CS 344 : OPERATING SYSTEMS LAB

LAB 3

GROUP NUMBER : 12

GROUP MEMBERS:

MOHAMMAD HUMAM KHAN (180123057)

KARTIKEYA SINGH (180123021)

SIDDHARTHA JAIN (180101078)

AB SATYAPRAKASH (180123062)

PART A
Lazy Memory Allocation

A brief description:

*In this part of the lab, we have implemented the **Lazy Memory Allocation** for xv6, which is a feature in most modern operating systems. Incase of the original xv6, it makes use of the **sbrk()** system call, to allocate physical memory and map it to the virtual address space. In the first section, we modified the **sbrk()** system call to remove the memory allocation, and cause a page fault. In the second section we have modified the trap.c file to resolve this page fault via lazy allocation.*

1. <u>Eliminate allocation from **sbrk()**</u>

   In this section, we have modified the **sbrk()** system call (also provided to us in the patch file). After initial declarations and error handling, the sbrk() system call has 4 essential lines in it.

```
1. addr = myproc()->sz;
2. myproc()->sz += n;

3. if(growproc(n) < 0)
    return -1;
4. return addr;
```

Line 1: Assigns **addr** to the start of the newly allocated region

Line 2: Increases the size for the current process by a factor n

Line 3: Calls the **growproc(n)** function in **proc.c**, which allocates n bytes of memory for the process.

Line 4: Returns the **addr**

Now we comment-out the line 3, and this makes the process to believe that it has got it's requested memory, while in reality it does not. This will cause a **trap error, with the code 14** when we try to run something like **echo hi** or **ls**. The code **14** corresponds to the **page fault error**, or **T_PGFLT**.

The modified **code** for system call **sbrk()** is a below:

```c
int
sys_sbrk(void)
{
  int addr;
  int n;
  if(argint(0, &n) < 0)
    return -1;

  addr = myproc()->sz;
  myproc()->sz += n;

  //  if(growproc(n) < 0)
  //     return -1;
  return addr;
}
```

On running echo hi in the terminal, we get the following **output**.

```
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ echo hi
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x112c addr 0x4004--kill proc
$ 
```

2. Lazy Allocation in **xv6**

In this section, we handle the page fault resulting from the changes in part A.1. For this we make use of the following observations :

a. The file **trap.c** has the code that produces the trap error as observed in part A.1. This is present in the **default** case for the **switch(tf->trapno)** as follows:

```
// In user space, assume the process misbehaved.

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,tf->err, cpuid(), tf->eip, rcr2());

myproc()->killed = 1;
```

b. Comparing with the above output we realise that **rcr2()** represents the contents of the control register 2 which in turn has the faulting virtual address. This is what goes into the input of **PGROUNDDOWN(va)** later.

c. Inside the **T_PGFLT** case, we make use of **PGROUNDDOWN(va)** to round down the **virtual address** to the start of the page boundary.

d. In **vm.c**, we have the function **allocuvm()** which is what **sbrk()** makes use of via the **growproc()** function.

e. Studying **allocuvm()**, makes it clear that it assigns 4KB (**PGSIZE**) of pages to a function making use of **kalloc()**, in a loop for as many pages as are needed. In our case we need a similar thing, except that we can do away with the loop and assign 1 page of size 4KB (**PGSIZE**) as and when a page fault occurs.

f. A final observation is to remove the **static** keyword for the mappages function in **vm.c** and declare it as **extern** in **trap.c**. This will make sure that we can call it inside the switch case. We also add a **break;** statement to make sure that **fall-through** does not occur and the default statements are not executed.

The code changed in various files are as below:

In **trap.c**

```c
extern int mappages(pde_t *pgdir, void *va, uint size, uint pa,
int perm);
```

```c
case T_PGFLT:
  {
    // code from allocuvm
    // cprintf("Trap Number : %x\n", tf->trapno);
    cprintf("rcr2() : 0x%x\n", rcr2());

    uint newsz = myproc()->sz;
    uint a = PGROUNDDOWN(rcr2());
    if(a < newsz){
      char *mem = kalloc();
      if(mem == 0) {
        cprintf("out of memory\n");
        exit();
        break;
      }
      memset(mem, 0, PGSIZE);
      mappages(myproc()->pgdir, (char*)a, PGSIZE, V2P(mem),
PTE_W|PTE_U);
    }
    break;
  }
```

In **vm.c**

```c
int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
```

5

The output on running commands like **echo** and **ls** is shown below. As can be seen, the trap error does not occur any more. Additionally, we have used **cprintf** in our code to print the **faulting virtual address** in each case, which shows up in the terminal output.

```
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ echo hi
rcr2() : 0x4004
rcr2() : 0xbfa4
hi
$ ls
rcr2() : 0x4004
rcr2() : 0xbfa4
.                1 1 512
..               1 1 512
README           2 2 2286
cat              2 3 13620
echo             2 4 12632
forktest         2 5 8060
grep             2 6 15496
init             2 7 13216
kill             2 8 12684
ln               2 9 12580
ls               2 10 14768
mkdir            2 11 12764
rm               2 12 12740
sh               2 13 23228
stressfs         2 14 13412
usertests        2 15 56344
wc               2 16 14160
zombie           2 17 12404
console          3 18 0
```

PART B
Memory and Paging in xv6

A brief description:

In this part of the lab, we examine the memory and paging aspects in xv6, and implement several features like creating a kernel kernel process for paging, swapping in and out from disk, and writing sanity tests.

1. **xv6 Memory:**

In this section we first understand the concept of memory management in xv6. After looking and understanding the code in relevant files like **mmu.h**, **kalloc.c** and **vm.c**, we now answer the following questions:

a. **How does the kernel know which physical pages are used and unused?**

The kernel uses the **struct run** (line 16 of **kalloc.c**) data structure to address an unused (free) page. This structure simply stores a pointer to the next free page, and the rest of the page is filled with garbage.

Note that *the kernel code that allocates and frees pages always returns the virtual address of the page in the kernel address space, and not the actual physical address. The V2P macro is used when one needs the physical address of the page, say to put into the page table entry*

b. **What data structures are used to answer this question?**

The list of free pages are maintained as a **linked list**. The pointer to the next page is stored within the page itself. Pages are added to this list whenever they are either initialised or freed up.

c. **Where do these reside?**

Each free page's run structure is stored within the page itself, since there is nothing else stored there.

d. Does xv6 memory mechanism limit the number of user processes?

The number of user processes that can be created is limited by **NPROC = 64** defined in **param.h**, that is at a time maximum processes can be created is **64**. However, all of these processes may not be present in main memory at same time i.e. the state of the processes may b**e RUNNING, RUNNABLE, SLEEPING, EMBRYO**, etc. Only the process that is currently **running** will be in memory. The physical address space as well as virtual address space in xv6 is equal to **4GB**. Also, xv6 keeps a copy of **kernel** code with each user process i.e. **2GB** is occupied by kernel and 2GB occupied by kernel.

*In the virtual address space of every process, the kernel code and data begin from **KERNBASE** (2GB in the code), and can go up to a size of **PHYSTOP** (whose maximum value can be 2GB). This virtual address space of **[KERNBASE, KERNBASE+PHYSTOP]** is mapped to **[0,PHYSTOP]** in physical memory. The kernel is mapped into the address space of every process, and the kernel has a mapping for all usable physical memory as well, restricting xv6 to using no more than 2GB of physical memory*

e. If so, what is the lowest number of processes xv6 can 'have' at the same time (assuming the kernel requires no memory whatsoever)?

If we ignore the kernel space of a process then at a time maximum 2 processes can be present in main memory.

2. <u>xv6 Paging:</u>

In this section we will implement swapping of pages, which is an important aspect for xv6. We will implement both swap in, and out of the disk and also create sanity tests to validate our implementation.

Task 1: Kernel process

To implement the paging mechanism, we will make a kernel process. For this we create a function,

void create_kernel_process(const char *name, void (*entrypoint)())

in **proc.c**. To implement the swapper, we make use of 2 functions, **swapin()** and **swapout()**. In the **main()** function of main.c, we call the create_kernel_process() function, which creates the processes needed to run swapin() and swapout(). As is obvious, the create_kernel_process() is called twice for swapping in and out respectively in the main.c file.

```
int
main(void){
    ......
    userinit();
    create_kernel_process("swapin", swapin);
    create_kernel_process("swapout", swapout);
    mpmain();
}
```

The implementation of **create_kernel_process()** is quite similar to **fork(), allocproc()** and **userinit()**. In particular, it has a general structure comparable to fork, but with certain key differences between them:

a. While **fork()** copies the address space, registers, etc. from parent processes, **create_kernel_process()** does no such thing. Instead it sets up the data from scratch in the same way that **allocproc()** and **userinit().**

b. As an example, we can see the way **fork()** sets up the trap frame for the child process, it simply copies it from the parent in one line:

```
void
fork(){
    ......
    *np->tf = *proc->tf; // note this line
    ......
}
```

However, in case of **create_kernel_process()**, it sets up the trapframe in

9

several lines of code similar to the way **userinit()** does.

```
void
create_kernel_process(){
    ......
    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
    np->tf->eax = 0; // clear %eax so that fork returns 0 in the
child
    ......
}
```

At the end of **create_kernel_process()**, it sets **np->context->eip** to the
entrypoint function pointer that was provided in the parameter. This means that
the process will start running at the function entrypoint when it starts.

```
void
create_kernel_process(const char *name, void (*entrypoint) ()){
    ......
    np->context->eip = (uint)entrypoint;
    ......
}
```

Task 2: Swapping out mechanism:

Whenever the kernel tries to allocate memory for a process, but is unable to do
so, we need to swap out a page into the backing store. For this we first need to
select a victim, which in our case we do using **LRU** replacement policy in
select_a_victim() function. Once this is done, we need to **write** the page into a

dedicated file with the name **<pid>_<VA[20:]>.swp**, where VA[20:] represents the **20 MSB of the virtual address**. Let's have a complete walkthrough about how swapping out has been implemented.

a. In **trap.c**, we have the T_PGFLT case under **switch(tf->trapno).** This case calls a function **handle_pgfault()**.

```
switch(tf->trapno){
  case T_PGFLT:
    handle_pgfault();
    break;
```

b. Now inside the function **handle_pgfault()**, we call another function **map_address(curproc->pgdir, addr)**, and we pass the **page table** and **virtual address of the page where page fault occured** to it.

```
/* page fault handler */
void
handle_pgfault()
{
  unsigned addr;
  struct proc *curproc = myproc();
  asm volatile ("movl %%cr2, %0 \n\t" : "=r" (addr));
  addr &= ~0xfff;
      cprintf("%d\n",addr);

  map_address(curproc->pgdir, addr);
  return;
}
```

c. The **map_address()** function calls the **kalloc()** function to assign a physical page. When it fails to allocate the page, we need to swap out a page. This comes under the case of **mem=0** as in the given code. If that is the case we

call yet another function **swap_page(pgdir)**, that handles the case of swapping out.

```
void
map_address(pde_t *pgdir, uint addr)
{
 .......
 char *mem=kalloc(); //allocate a physical page

 if(mem==0){
    swap_page(pgdir); //swap out a page
    mem=kalloc(); //Now a physical page has been swapped to file
and free, so this time we will get a physical page for sure.
    cprintf("kalloc success\n");
    cprintf("%p\n",mem);
 }
 ......
}
```

    d. The function **swap_page()**, first needs to select a victim page for swapping out. For this we make use of the function **select_a_victim(pgdir)**, which makes use of LRU policy to choose a victim (we will see about the working of this in the next point). It's just possible that no victim was found in the first attempt. In the event of occurrence of this we will clear the access bits of all the pages using the function **clearaccessbit(pgdir)**, which will actually reset the access bits for 10% of the pages, thus ensuring that a victim is available for sure in the second attempt. After selecting a victim, it calls the **swap_page_from_pte(pte_t *pte , uint pageNumber)**, which actually does the work of making a **.swp file** with the specifications as stated in the assignment.

```
int
swap_page(pde_t *pgdir)
{
 pte_t* pte=select_a_victim(pgdir); //returns *pte
 if(pte==0){  //If this is true, the victim is not found in the
```

```
1st attempt.
    cprintf("No victim found in 1st attempt. Clearing access
bits.\n");
    clearaccessbit(pgdir); //Access bits are cleared,

    cprintf("Finding victim again, after clearing access bits of
10%% pages.\n");
    pte=select_a_victim(pgdir);
    ......
  }

 swap_page_from_pte(pte, me);
 ......
}
```

e. Let's now briefly look at the function of **select_a_victim(pgdir),** which implements the LRU. The function begins with a loop which goes over all pages, and then makes use of 2 bits, **PTE_P** and **PTE_A**. **PTE_P** denotes if the page is present and **PTE_A** denotes if the page was accessed. If we get a page that is **present and was not accessed (LRU)**, we immediately choose that page as the victim page.

```
pte_t*
select_a_victim(pde_t *pgdir){
 pte_t *pte;
 for(long i=4096; i<KERNBASE;i+=PGSIZE){
    if((pte=walkpgdir(pgdir,(char*)i,0))!= 0){//if mapping exists
        if(*pte & PTE_P){  //present bit is set
            if(*pte & ~PTE_A){ //access bit is NOT set
                me = i;
                return pte;
            }
        }
    }
    else{
```

```
        cprintf("walkpgdir failed \n ");
      }
  }
  return 0;
}
```

f. Finally as already stated before **swap_page_from_pte(pte_t *pte , uint pageNumber)** is called which makes use of the 20 MSB of the virtual address (here the page number), and then makes a file, and writes to it making use of the **filewrite()** function in file.c.

```
filewrite(f,(char*)pageNumber,PGSIZE)
```

Task 3: Swapping in mechanism:

In this task we will implement the swapping in for xv6. Let's look at the steps involved for swapping in a similar way as we had explained for swapping out.
a. We have already seen the function **map_address()**, for the swapping out mechanism. Inside that function itself we have 2 more cases.
   ● In the first case, if the **PTE_SWAPPED** bit in the page table entry is set which means that the page fault occurred because the page is not present and was swapped out previously. In this case we need to call the **swapIn(uint pid , uint pageNumber , char* pageAddress)**, which will swap in the page from the backing store and then update the page table.
   ● The second case is the default case i.e. the memory is allocated by kalloc() and the page was not previously swapped out of memory. In such a case, we have nothing to swap in, so we update the page table only.

```
void
map_address(pde_t *pgdir, uint addr) {
```

```
......
if(pte!=0){
  if(*pte & PTE_SWAPPED){
    swapIn(curproc->pid , a , mem);
    cprintf("SWAPPING IN DONE PERFECTLY!!!\n");

     *pte=V2P(mem) | PTE_W | PTE_U | PTE_P;
     *pte &= ~PTE_SWAPPED;
     lcr3(V2P(pgdir));
  }
  else{
    memset(mem,0,PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_P |
PTE_W | PTE_U )<0){
       panic("allocuvm out of memory xv6 in mappages/n");
       deallocuvm(pgdir,cursz+PGSIZE, cursz);
       kfree(mem);
    }
     else{
       cprintf("mappages working");
    }
  }
  }
}
```

b. We now take a look at the **swapIn()** function which is present in **paging.c**. This function first makes use of its parameters to create the filename. Then it opens this file from the backing store using the **fileread()** function in **file.c.** Finally the read information is swapped in to the memory, using the **memmove()** function.

```
fileread(f , buf , 4096);
memmove(pageAddress , buf , 4096);
```

There are 3 test programs for our xv6 - memtest, memtest2, and memtest3. Please run them for sanity testing.

In this task, we validate the implementations for swapping in and out that we have done so far. The important thing here is to reduce the value of **PHYSTOP** to **0x450000**. This will make it run out of memory faster, because the kernel won't be able to hold all the processes in the **RAM**.

For this part of the assignment we wrote a user process by the name **memtest.c**, in which the main process forks **20** child processes.

```c
int
main(int argc, char* argv[])
{
  int i, pid;
  int pids[20];

  // As per the assignment spec, fork 20 children.
  for (i = 0; i < 20; i++) {
    pid = fork();
    if (pid == 0){
      child_proc();
    }
    pids[i] = pid;
  }

  printf(1, "first child pid: %d\n", pids[0]);

  while(wait() >= 0);
  exit();
}
```

Now, each child process runs a loop to perform memory sanity tests, which is

done in the function called **child_proc()**. Since the value of **PHYSTOP** is reduced, we actually make sure that swapping occurs and the values that have been assigned to the memory locations are intact.