CS 344 OPERATING SYSTEMS
LABORATORY - 03

GROUP 19 **Drishti Chouhan (180101021)** **Mridul Garg (180123028)**
Kousik Rajesh (180101094) **Eklavya Jain (180123065)**

PART - A:

1. In the patch file provided as part of the assignment, the page allocation process has been eliminated in the **sbrk(n)** system call in **sysproc.c.** This causes a trap error when running the command ***echo hi.***
2. The output of the command makes it clear that the **trap number is 14(tf->trapno)** and the **error number is 6(tf->error)**. The virtual address of the address that caused the page fault is stored in the cr2 register and the rcr2 function reads its value. The output of the error tells us that the virtual address is 0x4004.
3. In traps.h, **T_PGFLT** corresponds to trap 14.
4. There are multiple switch-case statements in trap.c, that handle the various trap errors. We add another case, **case T_PGFLT** to handle the page fault error, that isn't being dealt with originally.

// trap.c

```
case T_PGFLT:
  // rcr2() return the address that caused the page fault
  if (lazy_alloc(rcr2()) < 0) {
    myproc()->killed = 1;
  }
  break;
```

The function **int lazy_alloc(uint addr)** manually handles the page fault in trap.c by making use of functions that are used by kernel functions like **growproc()**. The argument passed to the function lazy_alloc is the virtual address at which the page fault occurs. All other parameters of the process structure are fetched using myproc(). The function returns 0 on success and -1 if the function fails, in which case, the process is killed.

```
int
lazy_alloc(uint addr) {
  uint a = PGROUNDDOWN(addr); //to round the faulting virtual address down to the start of a page boundary.

  char *mem = kalloc(); // method of allocating memory for objects smaller than page size in the kernel
  if (mem == 0)return -1;
  memset(mem, 0, PGSIZE);
  if (mappagesLazy(myproc()->pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W | PTE_U) < 0)return -1;
  return 0;
}
```

The new function **lazy_alloc** is declared in **vm.c**. growproc() function uses **allocuvm()** in order to allocate pages correctly. We take a look at this function in order to handle page faults correctly. Declaring lazy_alloc in vm.c also allows us to use the function mappages.

The arguments this function takes are pgdir(page directory), oldsz (old page size), and newsz (new page size). It allocates a new page to the passed directory using mappages() function.
Mappages() uses the **V2P** function to convert the virtual address to physical address and returns an integer depending on whether the pages were successfully allocated or not.

Some functions which will be required to manually handle the page fault that the system encounters are as follows:

memset(): makes sure that the allocated page is empty

kalloc(): simply allocates a page of 4096 bytes and returns a pointer to it

PGROUNDUP(address) is a macro that rounds up the address sent to a multiple of the PGSIZE.

mappagesLazy(): Creates a new **page table entry(PTE)** for the given virtual address. It maps the page to the processes' page directory by using V2P to convert the rounded virtual address to a physical address. Since our function lazy_alloc in vm.c makes use of mappages, we would have to either remove the static declaration in vm.c. Instead, we declared another function mappagesLazy in vm.c, which is the same as mappages, just without the static declaration, so lazy alloc can make a call to it in trap.c

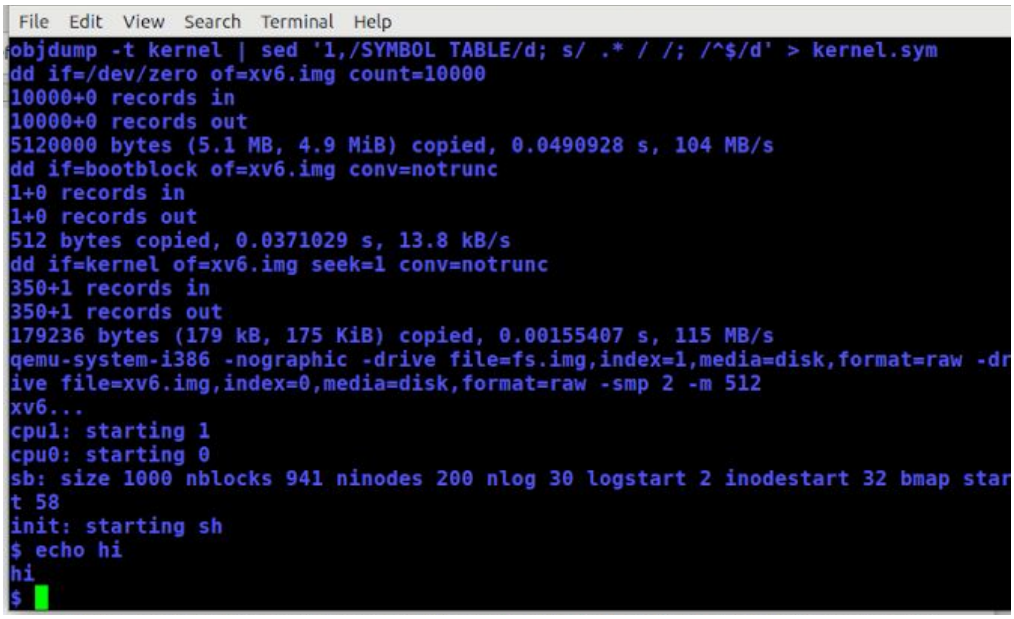
// vm.c

```
int
mappagesLazy(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

The function lazy_alloc makes use of all these functions. It takes as argument the directory of the current process (myproc()) and the address causing page fault(given by rcr2). We implement the rest of the function so the program executes without interruption in the userspace.

Lazy_alloc, therefore, takes a faulty address, aligns it correctly and rounds it up, and then allocates a new page from the physical memory to the process that raised the fault using the mappagesLazy() function. The process then continues to execute correctly, as is shown by the error-free output of the command **echo hi**



PART B:

Task 1: Kernel Process

The implementation of the function create_kernel_process resembles fork() , allocproc(), and userinit(). Although there are significant differences, for instance, fork() copies the address space, registers, and other process parameters from the parent process, while create_kernel_process does not proceed in this way. It sets up the data from scratch in a manner similar to allocproc() and userinit().

The difference is illustrated in the following example :
When the fork sets up the trap frame for the child processes, it copies from the parent using:

```
fork(){
...
*np->tf = *proc->tf
...
}
```

On the other hand, create_kernel_process() instead sets up the entire trap frame like userinit():

```
create_kernel_process(){
...

  memset(newp->tf, 0, sizeof(*newp->tf)); // Populate newp->tf with zeros

  newp->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  newp->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  newp->tf->es = newp->tf->ss = newp->tf->ds;
```

```
  newp->tf->eflags = FL_IF;
  newp->tf->esp = 4096; // Set page size

  newp->tf->eax = newp->tf->eip = 0;
...
}
```

At the end of the function create_kernel_process, it sets np->context->eip to the entrypoint function pointer that was provided in one of the parameters. . The process begins running at the function entrypoint when it starts.

Task 2 & 3: Swapping out, Swapping in Mechanism

//vm.c:

select_victim(): In order to swap out a page, we must first select a victim page. We select a page table entry which is mapped but not valid. We try to find a page whose valid bit cannot be set. If such a page is not present, we randomly reset the valid bits of 10% of the allocated pages and call select_victim again.

clear_valid_bits(): This function randomly clears the valid bits of 10% of the allocated pages.

get_swapped_block(): The function returns the disk-id of the swapped block if the virtual address was swapped, and -1 in the other case. It extracts the disk-id of the swapped block from the first 20 bits of the page table entry.

//bio.c

write_page_to_disk(): The function writes 4KB of the page to the eight consecutive memory blocks starting at the address initial_block. It writes the physical page to the disk by dividing it into 8 pieces. (1 block = 512 bytes)

read_page_from_disk(): Reads 4KB from eight consecutive memory blocks starting from address initial_block into the page. We read from the disk into a temporary buffer and from the buffer into the page.

//paging.c

map_physical_to_virtual(): This function in paging.c maps a physical page to a virtual address. If the page table entry is pointing to a swapped block, then the function restores the page contents from the swapped block and frees the swapped block.

swap_page(): The function swaps pages after selecting a victim frame using the function select_victim(). The select_victim() function makes two attempts to find a victim frame, and it prints on the kernel accordingly. Then it calls swap_pages_from_pte to swap the victim page from the disk.

swap_page_from_pte(): This function allocates 8 consecutive blocks from disk and writes the dirty page in the physical memory to disk and updates the page table entries. It saves the content of the page in the PTE to the disk and saves the block-id into the PTE.

//fs.c

balloc_page() : The function is similar to balloc() function, except the fact that it allocates 8 consecutive empty blocks. We can safely assume that the first block is always 8 bytes aligned.

bfree_page(): Is used to free disk blocks that were allocated using balloc_page.

Task 4 : Sanity Test

In the program memtest we create 20 child processes using the fork command and each process has a for loop of 10 iterations each and in each iteration we allocate a memory block of 4096 bytes using the malloc statement. We store the pointer for each of these memory blocks in a global array of pointers.
When the iterations complete we call the validate function on each of the memory pointers that we malloced and print and exit if any of the blocks fail to validate
Finally we wait for each child process to complete execution.

```
Creating child processes with PID's :
4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,

Validated child process 4
Validated child process 5
Validated child process 6
Validated child process 7
Validated child process 8
Validated child process 9
Validated child process 10
Validated child process 11
Validated child process 12
Validated child process 13
Validated child process 14
Validated child process 15
Validated child process 16
Validated child process 17
Validated child process 18
Validated child process 19
Validated child process 20
Validated child process 21
Validated child process 22
Validated child process 23
--------------------------------------------------------------------------
Test completed successfully!
```