Assignment 5
Swapnil Bag
ES22BTECH11034

(Note that the output for task 1 is recorded before demand paging was implemented, hence the output with demand paging will different from what is shown below)

System call

When a user function invokes a system call then the OS triggers an exception and switches from user mode to kernel mode. The trap handler services the request based on the system call number as in (kernel/ syscall.h). The actual process to be executed is in kernel/sysproc.c where the system call is serviced. After completion of execution it returns to user mode.

Task 1

Method

- The function pgtPrint was implemented in kernel/syspoc.c System call number defined in kernel/syscall.h
- Function definition added in kernel/syscall.c (kernel definition) and also in user/user.h (userspace definition)
- Function call added in usys.S and usys.pl

pgtPrint(): It is a recursive function that obtains the page table as argument and traverses the page table structure level by level, at each level the virtual address is constructed (by bitwise right shift based on level) and is checked; if it points to lower level (R,W,X bits are 0) or is the final level and prints it

1) For global array 100 PTE entries are printed (top and bottom entries shown). It is probably because when the program runs all the entries of the array are initialized to 0 by the compiler (hence the pages are fetched and made valid) and mapped to physical memory . (Virtual address starts from 0x0000 and gets contiguous allocated).

```
$ mypgtPrint
PTE entry 0: virtual address 0x0000000000000000 physical address 0x0000000087f40000
PTE entry 1: virtual address 0x0000000000001000 physical address 0x0000000087f3d000
PTE entry 2: virtual address 0x0000000000002000 physical address 0x0000000087f3c000
PTE entry 3: virtual address 0x0000000000003000 physical address 0x0000000087f3b000
PTE entry 4: virtual address 0x0000000000004000 physical address 0x0000000087f3a000
PTE entry 5: virtual address 0x0000000000005000 physical address 0x0000000087f39000
PTE entry 6: virtual address 0x0000000000006000 physical address 0x0000000087f38000
PTE entry 7: virtual address 0x0000000000007000 physical address 0x0000000087f37000
PTE entry 8: virtual address 0x0000000000008000 physical address 0x0000000087f36000
PTE entry 9: virtual address 0x0000000000009000 physical address 0x0000000087f35000
```

…..

```
PTE entry 90: virtual address 0x000000000005a000 physical address 0x0000000087f4f000
PTE entry 91: virtual address 0x000000000005b000 physical address 0x0000000087f50000
PTE entry 92: virtual address 0x000000000005c000 physical address 0x0000000087f51000
PTE entry 93: virtual address 0x000000000005d000 physical address 0x0000000087f52000
PTE entry 94: virtual address 0x000000000005e000 physical address 0x0000000087f53000
PTE entry 95: virtual address 0x000000000005f000 physical address 0x0000000087f54000
PTE entry 96: virtual address 0x0000000000060000 physical address 0x0000000087f55000
PTE entry 97: virtual address 0x0000000000061000 physical address 0x0000000087f6f000
PTE entry 98: virtual address 0x0000000000062000 physical address 0x0000000087f63000
PTE entry 100: virtual address 0x0000000000064000 physical address 0x0000000087f6e000
$
```

2) In my implementation of the local array, I initialized only the first element to be 0 and printed it. Only 3 valid PTE entries are printed. Here possibly since only a single element initialized lesser number of pages allocated for the given process and hence lower PTE

```
$ mypgtPrint
arrLocal: 0
PTE entry 0: virtual address 0x0000000000000000 physical address 0x0000000087f54000
PTE entry 1: virtual address 0x0000000000001000 physical address 0x0000000087f63000
PTE entry 3: virtual address 0x0000000000003000 physical address 0x0000000087f6e000
$ mypgtPrint
arrLocal: 0
PTE entry 0: virtual address 0x0000000000000000 physical address 0x0000000087f49000
PTE entry 1: virtual address 0x0000000000001000 physical address 0x0000000087f4c000
PTE entry 3: virtual address 0x0000000000003000 physical address 0x0000000087f3c000
```

3) Global array: upon multiple executions the physical address remains unchanged, indicating that the program is being located in the same location of memory probably because that space in Physical memory has the necessary contiguous space
Local array: upon multiple executions the physical address changes, the program gets loaded in different portion of DRAM depending on the availability of physical space

Task 2

Method:

- In kernel/ exec.c memory is being allocated to only hold the size of file segment . If the size of memsz>filesz then it means that more space has to be allocated (uninitialized data) and hense increment the size by this difference
- In kernel/trap.c, check for the cause of trap; if theres a page fault, then report. Allocate space in the physical memory and if it succeeds , map the given faulting virtual address to the allocated physical memory using mmapages
- In kernel/ vm.c remove the panic command and continue if pagetable is valid

Here the program is executed for N=3000, every 1000 iterations a page fault occurs and page table is printed. When page fault occurs for a given virtual memory, physical memory is allocated and mapped to the corresponding faulting virtual memory. For N=5000 and N=100000 more number of page faults occur and the number of PTE increases per 1000 th iteration as more number of pages are being allocated (1000, 2000…..) as the array is being filled.

When memory is allocated dynamically using malloc, the page is being allocated in the heap section using sbrk() system call. Now by only loading code/text segment page fault occurs and hence demand paging is initiated

```
$ mydemandPage
page fault occured, doing demand paging for 0x0000000000001010
global addr from user space: 1010
PTE entry 0: virtual address 0x0000000000000000 physical address 0x0000000087f40000
PTE entry 1: virtual address 0x0000000000001000 physical address 0x0000000087f44000
PTE entry 5: virtual address 0x0000000000005000 physical address 0x0000000087f3c000

page fault occured, doing demand paging for 0x0000000000002000
PTE entry 0: virtual address 0x0000000000000000 physical address 0x0000000087f40000
PTE entry 1: virtual address 0x0000000000001000 physical address 0x0000000087f44000
PTE entry 2: virtual address 0x0000000000002000 physical address 0x0000000087f73000
PTE entry 5: virtual address 0x0000000000005000 physical address 0x0000000087f3c000

page fault occured, doing demand paging for 0x0000000000003000
Printing final page table:
PTE entry 0: virtual address 0x0000000000000000 physical address 0x0000000087f40000
PTE entry 1: virtual address 0x0000000000001000 physical address 0x0000000087f44000
PTE entry 2: virtual address 0x0000000000002000 physical address 0x0000000087f73000
PTE entry 3: virtual address 0x0000000000003000 physical address 0x0000000087f72000
PTE entry 5: virtual address 0x0000000000005000 physical address 0x0000000087f3c000
Value: 2
```

(for global array)

```
$ mydemandPage
page fault occured, doing demand paging for 0x0000000000001000
page fault occured, doing demand paging for 0x0000000000003ef0
arr: 1
PTE entry 0: virtual address 0x0000000000000000 physical address 0x0000000087f40000
PTE entry 1: virtual address 0x0000000000001000 physical address 0x0000000087f44000
PTE entry 3: virtual address 0x0000000000003000 physical address 0x0000000087f73000
PTE entry 5: virtual address 0x0000000000005000 physical address 0x0000000087f3c000
PTE entry 6: virtual address 0x0000000000006000 physical address 0x0000000087f72000
```

(for malloc)

Task 3

- **pgaccess()** system call was added in kernel/ sysproc.c. The function accepts start address, number of pages and buffer to store results. In the function it uses the page table structure and traverses PTE for a certain range (using the walk function in kernel/ vm.c). It then checks if PTE_A flag (6th bit is set or not) records it in the bit mask and resets. The bitmask is returned to the user program using copyout (to return from kernel to user).
- PTE_A flag is defined in kernel/ riscv.h
- The same procedure for making a system call is followed as in task 1

dirtyPage.c is created in the user file to test this function. An array of size 10000 is created at and pgaccess() is called

```
$ dirtyPage
arr: 90
bit_buff: 0x0000000000003FB8
```

The bitmask prints 0x3FB8 that translates to
0011 1111 1011 1000 This implies that when a search was made for 32 pages starting from the virtual address 0x0000 10 pages have been modified