OPERATING SYSTEMS II

PROGRAMMING ASSIGNMENT 5 : *REPORT*

By: Devraj, ES22BTECH11011

WORKING OF SYSTEM CALLS

With the help of system calls, we provide a way for user level processes to request for services from the kernel (operating system). This happens in the following manner:

- A user program invokes a system call using special instructions that transfer control to the kernel.
- The processor switches from user mode to kernel mode to execute some privileged instructions.
- The specific system call is identified by a system call number passed as an argument in a register.
- The kernel executes the system call, the system call handler performs the requested operation on behalf of the user process.
- After the instruction is executed, control is returned to user space.

INSIGHTS FROM THIS ASSIGNMENT

Through this assignment, I learned how to implement system calls in xv6.
I also learned how to check the page table entries of a process to determine the permissions associated with the process. With this, I learned to get the virtual and physical addresses of a page using the page table entries.
It was also interesting to learn to isolate various parts of a process's memory space into readable, writable, etc and use them to implement demand paging.
This assignment also helped in the understanding of bitmaps/bitmasks and their uses in storing information regarding bitwise comparisons between addresses and numbers.
Finally, this assignment helped in the overall understanding of how operating systems implement various methods in order to ensure the correct and smooth functioning of a computer.

OUTPUTS AND THE REASONING BEHIND THEM

TASK 1

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

// int arrGlobal[10000];

int main() {
    // int arrLocal[10000];
    // arrLocal[5] = 5;
    // arrLocal[7] = arrLocal[5];
    pgtPrint();
    exit(0);
}
```

```
$ mypgtPrint
PTE No: 0, Virtual page address: 0x0000000000000000, Physical page address: 0x0000000087f40000
PTE No: 1, Virtual page address: 0x0000000000001000, Physical page address: 0x0000000087f3d000
PTE No: 3, Virtual page address: 0x0000000000003000, Physical page address: 0x0000000087f3b000
$ mypgtPrint
PTE No: 0, Virtual page address: 0x0000000000000000, Physical page address: 0x0000000087f54000
PTE No: 1, Virtual page address: 0x0000000000001000, Physical page address: 0x0000000087f63000
PTE No: 3, Virtual page address: 0x0000000000003000, Physical page address: 0x0000000087f6e000
$
```

This output is without the local/global array initialization. As we can see, only three page table entries are valid, i.e., only three pages are needed to store the program segments.

Also, we can observe that upon multiple executions the virtual addresses remain the same while the physical addresses may change.

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int arrGlobal[10000];

int main() {
    // int arrLocal[10000];
    // arrLocal[5] = 5;
    // arrLocal[7] = arrLocal[5];
    pgtPrint();
    exit(0);
}
```

```
$ mypgtPrint
PTE No: 0, Virtual page address: 0x0000000000000000, Physical page address: 0x0000000087f40000
PTE No: 1, Virtual page address: 0x0000000000001000, Physical page address: 0x0000000087f3d000
PTE No: 2, Virtual page address: 0x0000000000002000, Physical page address: 0x0000000087f3c000
PTE No: 3, Virtual page address: 0x0000000000003000, Physical page address: 0x0000000087f3b000
PTE No: 4, Virtual page address: 0x0000000000004000, Physical page address: 0x0000000087f3a000
PTE No: 5, Virtual page address: 0x0000000000005000, Physical page address: 0x0000000087f39000
PTE No: 6, Virtual page address: 0x0000000000006000, Physical page address: 0x0000000087f38000
PTE No: 7, Virtual page address: 0x0000000000007000, Physical page address: 0x0000000087f37000
PTE No: 8, Virtual page address: 0x0000000000008000, Physical page address: 0x0000000087f36000
PTE No: 9, Virtual page address: 0x0000000000009000, Physical page address: 0x0000000087f35000
PTE No: 10, Virtual page address: 0x000000000000a000, Physical page address: 0x0000000087f34000
PTE No: 12, Virtual page address: 0x000000000000c000, Physical page address: 0x0000000087f32000
$ mypgtPrint
PTE No: 0, Virtual page address: 0x0000000000000000, Physical page address: 0x0000000087f4b000
PTE No: 1, Virtual page address: 0x0000000000001000, Physical page address: 0x0000000087f4e000
PTE No: 2, Virtual page address: 0x0000000000002000, Physical page address: 0x0000000087f4f000
PTE No: 3, Virtual page address: 0x0000000000003000, Physical page address: 0x0000000087f50000
PTE No: 4, Virtual page address: 0x0000000000004000, Physical page address: 0x0000000087f51000
PTE No: 5, Virtual page address: 0x0000000000005000, Physical page address: 0x0000000087f52000
PTE No: 6, Virtual page address: 0x0000000000006000, Physical page address: 0x0000000087f53000
PTE No: 7, Virtual page address: 0x0000000000007000, Physical page address: 0x0000000087f54000
PTE No: 8, Virtual page address: 0x0000000000008000, Physical page address: 0x0000000087f55000
PTE No: 9, Virtual page address: 0x0000000000009000, Physical page address: 0x0000000087f6f000
PTE No: 10, Virtual page address: 0x000000000000a000, Physical page address: 0x0000000087f63000
PTE No: 12, Virtual page address: 0x000000000000c000, Physical page address: 0x0000000087f6e000
$
```

This output is with the large global array defined. As we can see, the number of page table entries increases by quite a lot. This is because the global variables are stored in the data segment of the process, memory is allocated for this array when it is loaded in the memory.
Hence, the number of page table entries increases because separate pages are required for this array.

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

// int arrGlobal[10000];

int main() {
    int arrLocal[10000];
    arrLocal[5] = 5;
    arrLocal[7] = arrLocal[5];
    pgtPrint();
    exit(0);
}
```

```
$ mypgtPrint
PTE No: 0, Virtual page address: 0x0000000000000000, Physical page address: 0x0000000087f40000
PTE No: 1, Virtual page address: 0x0000000000001000, Physical page address: 0x0000000087f3d000
PTE No: 3, Virtual page address: 0x0000000000003000, Physical page address: 0x0000000087f3b000
$ mypgtPrint
PTE No: 0, Virtual page address: 0x0000000000000000, Physical page address: 0x0000000087f54000
PTE No: 1, Virtual page address: 0x0000000000001000, Physical page address: 0x0000000087f63000
PTE No: 3, Virtual page address: 0x0000000000003000, Physical page address: 0x0000000087f6e000
$
```

This output is with the large local array initialized inside the user function. As we can see, the number of page table entries remains the same as that of the user program with no large arrays.

This is because local variables have their memory allocated in the stack segment. The pages used for the local array might reuse the existing entries within the page table that were already set up in the stack segment.

TASK 2

```
14    for (int i = 1; i < N; i++)
15    {
16      glob[i] = glob[i - 1];
17      if (i % 1000 == 0)
18      {
19        printf("instance: %d\n", i/1000);
20        pgtPrint();
21      }
22    }
23
24    int *arrHeap;
25    arrHeap = malloc(10000*sizeof(int));
26    arrHeap[5] = 5;
27    arrHeap[7] = arrHeap[5];
28
29    printf("Printing final page table (after heap allocation):\n");
30    pgtPrint();
```

```
$ mydemandPaging
page fault occured, doing demand paging for address: 0x0000000000005000
page fault occured, doing demand paging for address: 0x0000000000014000
page fault occured, doing demand paging for address: 0x0000000000001000
global addr from user space: 0x0000000000001010
instance: 1
PTE No: 0, Virtual page address: 0x0000000000000000, Physical page address: 0x0000000087f4e000
PTE No: 1, Virtual page address: 0x0000000000001000, Physical page address: 0x0000000087f52000
PTE No: 5, Virtual page address: 0x0000000000005000, Physical page address: 0x0000000087f4a000
page fault occured, doing demand paging for address: 0x0000000000002000
instance: 2
PTE No: 0, Virtual page address: 0x0000000000000000, Physical page address: 0x0000000087f4e000
PTE No: 1, Virtual page address: 0x0000000000001000, Physical page address: 0x0000000087f52000
PTE No: 2, Virtual page address: 0x0000000000002000, Physical page address: 0x0000000087f73000
PTE No: 5, Virtual page address: 0x0000000000005000, Physical page address: 0x0000000087f4a000
page fault occured, doing demand paging for address: 0x0000000000003000
page fault occured, doing demand paging for address: 0x0000000000006000
page fault occured, doing demand paging for address: 0x000000000000c000
Printing final page table (after heap allocation):
PTE No: 0, Virtual page address: 0x0000000000000000, Physical page address: 0x0000000087f4e000
PTE No: 1, Virtual page address: 0x0000000000001000, Physical page address: 0x0000000087f52000
PTE No: 2, Virtual page address: 0x0000000000002000, Physical page address: 0x0000000087f73000
PTE No: 3, Virtual page address: 0x0000000000003000, Physical page address: 0x0000000087f72000
PTE No: 5, Virtual page address: 0x0000000000005000, Physical page address: 0x0000000087f4a000
PTE No: 6, Virtual page address: 0x0000000000006000, Physical page address: 0x0000000087f71000
PTE No: 12, Virtual page address: 0x000000000000c000, Physical page address: 0x0000000087f70000
Value: 2
$
```

This is the output for the size of the global integer array equal to *3000* and a dynamically allocated integer array of size *10000* is declared inside *main()*.

As we can see in each instance, a page fault occurs for a particular virtual address and the same address is present in the page table in the next instance. This is because of the read/write operations happening with the global array in each iteration of the for loop in the user program.

Since the array is large, it might not be entirely loaded into physical memory. Hence when we access indices not previously accessed/loaded into memory, it can trigger page faults and new pages might need to be allocated.

TASK 3

```
28      buf = malloc(32 * PGSIZE);
29      if (pgaccess((uint64)buf, 32, (uint64)&abits) < 0)
30          printf("pgaccess failed");
31      pgtPrint();
32      printf("\nbitmask before access\n");
33      printBinary(abits);
34
35      buf[PGSIZE * 1] += 1;
36      buf[PGSIZE * 2] += 1;
37      buf[PGSIZE * 30] += 1;
38      if (pgaccess((uint64)buf, 32, (uint64)&abits) < 0)
39          printf("pgaccess failed");
40      pgtPrint();
41      printf("\nbitmask after access\n");
42      printBinary(abits);
```

```
$ mypageAccess
page fault occured, doing demand paging for address: 0x0000000000005000
page fault occured, doing demand paging for address: 0x0000000000014000
pgaccess_test starting
page fault occured, doing demand paging for address: 0x0000000000001000
page fault occured, doing demand paging for address: 0x0000000000004000
PTE No: 0, Virtual page address: 0x0000000000000000, Physical page address: 0x0000000087f4e000
PTE No: 1, Virtual page address: 0x0000000000001000, Physical page address: 0x0000000087f52000
PTE No: 3, Virtual page address: 0x0000000000003000, Physical page address: 0x0000000087f4a000
PTE No: 4, Virtual page address: 0x0000000000004000, Physical page address: 0x0000000087f73000

bitmask before access
0000000000000000000000000000000000000000000000000000000000000011
page fault occured, doing demand paging for address: 0x0000000000005000
page fault occured, doing demand paging for address: 0x0000000000006000
page fault occured, doing demand paging for address: 0x0000000000022000
PTE No: 0, Virtual page address: 0x0000000000000000, Physical page address: 0x0000000087f4e000
PTE No: 1, Virtual page address: 0x0000000000001000, Physical page address: 0x0000000087f52000
PTE No: 3, Virtual page address: 0x0000000000003000, Physical page address: 0x0000000087f4a000
PTE No: 4, Virtual page address: 0x0000000000004000, Physical page address: 0x0000000087f73000
PTE No: 5, Virtual page address: 0x0000000000005000, Physical page address: 0x0000000087f72000
PTE No: 6, Virtual page address: 0x0000000000006000, Physical page address: 0x0000000087f71000
PTE No: 34, Virtual page address: 0x0000000000022000, Physical page address: 0x0000000087f70000

bitmask after access
0011000000000000000000000000000000000000000000000000000000111100

pgaccess_test: OK
```

This is the output for a buffer of size *32*PGSIZE* and a *64*-bit bitmask is printed to show the changes before and after accessing a memory location within the buffer.

In this bitmask, the first two bits from the left indicate the access bit and the dirty bit respectively for the first page of the buffer, the next two bits for the second page and so on.

As we can see, before accessing the buffer, only the bits for the first page are set to 1. This could be because we are accessing the first byte/address of the buffer to allocate space for it in the first place.

After the memory accesses (incrementing the values by 1) for pages 1, 2 and 30 we can see that the bits for these pages have been set to 1.