Programming Assignment 5: Syscall Implementation

CO21BTECH11004

To add a system call (date & pgtPrint here) in xv6, change the following files: -

- syscall.h : -
    - Define a number to your system call
    - Syntax: -

```
#define SYS_date     22
#define SYS_pgtPrint 23
```

- syscall.c: -
    - Do extern int declaration for the system call, which tells the compiler that our system call is defined somewhere else in the code.
    - Syntax: -

```
extern int sys_date(void);
extern int sys_pgtPrint(void);
```

    - Add the entry for our system call in the array of function pointers to system call (*systemcalls[])
    - Syntax: -

```
[SYS_date]     sys_date,
[SYS_pgtPrint] sys_pgtPrint,
```

- sysproc.c: -
    - Implement your system call here.
    - Eg date system call: -

```
int sys_date(void)
{
     struct rtcdate *r;
     if(argptr(0, (void*)&r, sizeof(*r)) < 0)
         return -1;
     cmostime(r);
     return 0;
}
```

- usys.S
    - Updating assembly file so that user program can call the system call
    - Syntax: -

```
SYSCALL(date)
SYSCALL(pgtPrint)
```

- user.h
    - Define a prototype wrapper to invoke a system call.
    - Syntax: -

```
int date(struct rtcdate*);
int pgtPrint(void);
```

- User program is written to call system call. Here mydate.c ang mypgtPrint.c .
- Makefile:- In UPROGS, the file name of the user program is added.

How system call works (high-level overview): -

When the user-level program executes syscall instruction, the syscall number is passed in %eas register, generates trap (trap.c) handled by trapasm.S, and there is transfer to kernel mode. After identifying the syscall kernel performs appropriate functions. After execution, the control is passed back to user mode.

Part 1: -

Following modifications are made in syscall.c: -

Uncomment these cprintf statements and syscallNames array for part 1.

- In the syscall() function for a valid system call, cprintf is used to print the required: -

```
cprintf("%s: ", curproc->name);          // print process name
cprintf("%s->", sysCallNames[num]);      // print system call name
cprintf("%d->", num);                    // print system call number
cprintf("%d\n", curproc->tf->eax);       // print return value
```

- Array to hold system call names is made syscallNames[]

```
static char *syscallNames[] = {
[SYS_fork]    "fork",
[SYS_exit]    "exit",
. . .
```

Part 2: -

Comment on the part1 things from syscall.c to avoid extra printing.

System call date is called when mydate runs from the xv6 shell. date system call is added using the steps mentioned above.

- In mydate.c date systemcall is called, in which costime() is used to get real-time.
- This time is stored in struct rtcdate r.
- Then time and date are printed.

Part 3: -

Uncomment the local array and global array part from mypgtPrint.c before use.

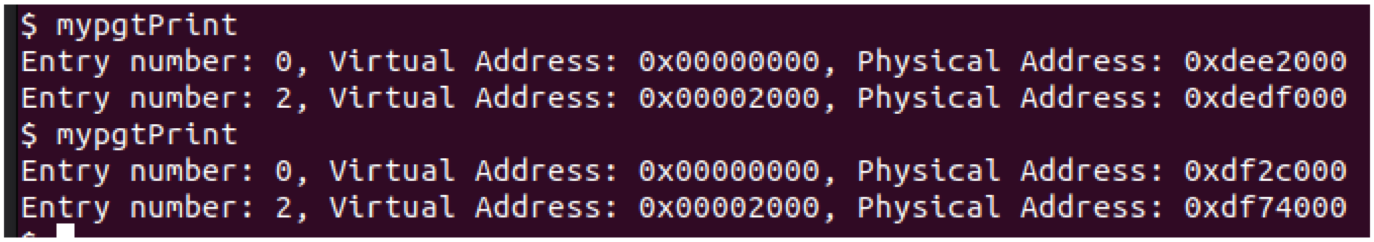
System call pgtPrint is called when mypgtPrint runs from the xv6 shell. pgtPrint system call is added using the steps mentioned above.

In sysproc.c, sys_pgtPrint: -

- Page directory (page table) pointer (pgdir) for the current process using p->pgdir, where struct proc *p = myproc(), which represents the process and contains information about it.
- A page table is stored in physical memory as a two-level tree.
- For traversing through the page directory (level 1, NPDENTRIES(1024) entries) f**or loop (i)** is used, and if the page table page is present (checked using PTE_P), using (level 2, NPTENTRIES(1024) entries) **for loop (j),** iterate through it and if page table entry is valid (checked using PTE_P) and in user mode access (checked using PTE_U), print entry number, virtual address and physical address of that page table entry.

Observations for different experiments for Part 3: -

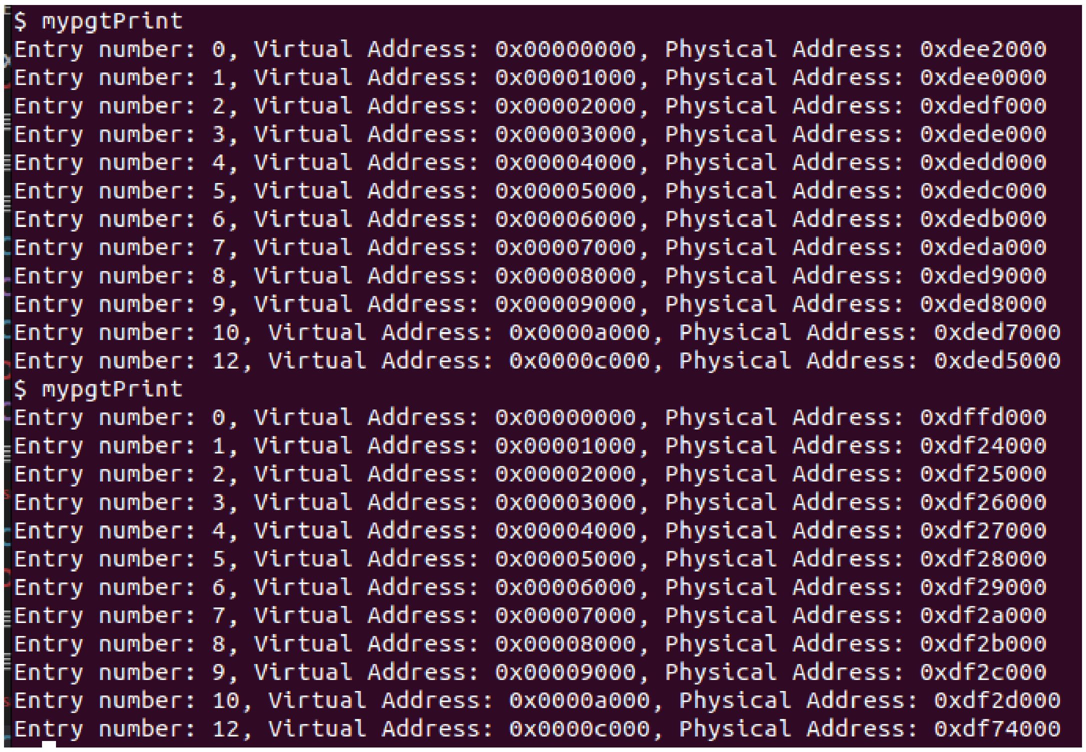
Output without any array (called pgtPrint systemcall two times): -



```
$ mypgtPrint
Entry number: 0, Virtual Address: 0x00000000, Physical Address: 0xdee2000
Entry number: 2, Virtual Address: 0x00002000, Physical Address: 0xdedf000
$ mypgtPrint
Entry number: 0, Virtual Address: 0x00000000, Physical Address: 0xdf2c000
Entry number: 2, Virtual Address: 0x00002000, Physical Address: 0xdf74000
```

- Running multiple times doesn’t change the number of entries.

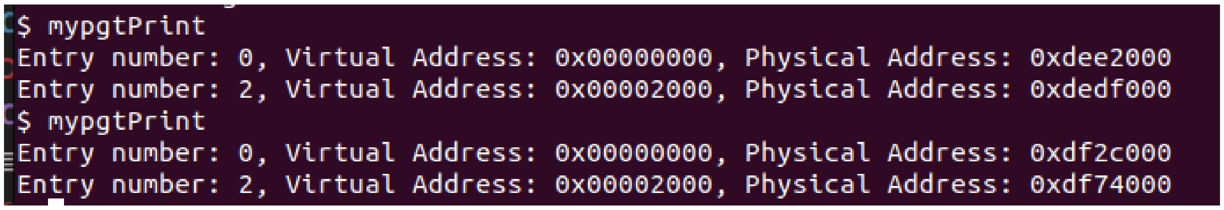
1. int arrGlobal[10000]
   Output (called pgtPrint systemcall two times): -



```
$ mypgtPrint
Entry number: 0, Virtual Address: 0x00000000, Physical Address: 0xdee2000
Entry number: 1, Virtual Address: 0x00001000, Physical Address: 0xdee0000
Entry number: 2, Virtual Address: 0x00002000, Physical Address: 0xdedf000
Entry number: 3, Virtual Address: 0x00003000, Physical Address: 0xdede000
Entry number: 4, Virtual Address: 0x00004000, Physical Address: 0xdedd000
Entry number: 5, Virtual Address: 0x00005000, Physical Address: 0xdedc000
Entry number: 6, Virtual Address: 0x00006000, Physical Address: 0xdedb000
Entry number: 7, Virtual Address: 0x00007000, Physical Address: 0xdeda000
Entry number: 8, Virtual Address: 0x00008000, Physical Address: 0xded9000
Entry number: 9, Virtual Address: 0x00009000, Physical Address: 0xded8000
Entry number: 10, Virtual Address: 0x0000a000, Physical Address: 0xded7000
Entry number: 12, Virtual Address: 0x0000c000, Physical Address: 0xded5000
$ mypgtPrint
Entry number: 0, Virtual Address: 0x00000000, Physical Address: 0xdffd000
Entry number: 1, Virtual Address: 0x00001000, Physical Address: 0xdf24000
Entry number: 2, Virtual Address: 0x00002000, Physical Address: 0xdf25000
Entry number: 3, Virtual Address: 0x00003000, Physical Address: 0xdf26000
Entry number: 4, Virtual Address: 0x00004000, Physical Address: 0xdf27000
Entry number: 5, Virtual Address: 0x00005000, Physical Address: 0xdf28000
Entry number: 6, Virtual Address: 0x00006000, Physical Address: 0xdf29000
Entry number: 7, Virtual Address: 0x00007000, Physical Address: 0xdf2a000
Entry number: 8, Virtual Address: 0x00008000, Physical Address: 0xdf2b000
Entry number: 9, Virtual Address: 0x00009000, Physical Address: 0xdf2c000
Entry number: 10, Virtual Address: 0x0000a000, Physical Address: 0xdf2d000
Entry number: 12, Virtual Address: 0x0000c000, Physical Address: 0xdf74000
```

- The number of valid entries increases in the case of a global array compared to when the global array was not declared.
- A typical explanation is that the global array memory is allocated statically at compile time and stored in the data segment of the process's virtual address space.

2. int arrLocal[10000]
   Output (called pgtPrint systemcall two times): -



```
$ mypgtPrint
Entry number: 0, Virtual Address: 0x00000000, Physical Address: 0xdee2000
Entry number: 2, Virtual Address: 0x00002000, Physical Address: 0xdedf000
$ mypgtPrint
Entry number: 0, Virtual Address: 0x00000000, Physical Address: 0xdf2c000
Entry number: 2, Virtual Address: 0x00002000, Physical Address: 0xdf74000
```

- The number of valid entries remains the same in the case of a local array compared to when the local array was not declared as the local array is dynamically allocated at runtime,

3. Repeated Execution: -
   - On repeated execution, the number of valid entries remains the same.
   - On repeated execution of the system call, the Virtual address remains the same while the physical address changes.
   - Physical address changes due to the operating system's dynamic allocation and management of physical memory.