Syscall Implementation in xv6

Aaryan, CO21BTECH11001

In this assignment, we learnt how system calls work in the xv6 operating system. System calls in xv6 works as follows:

a. Implementation of the system call is present in `sysproc.c` file.
   Function corresponding to a system call named `<sysCallName>` is generally defined as `int sys_<sysCallName>(void)`.
   For example the system call `uptime` has a function name `sys_uptime`.
b. The system call is assigned a number in `syscall.h` file in this format:

```
#define SYS_<sysCallName> <sysCallNumber>
```

c. The function defined in `sysproc.c` is declared in `syscall.c` in the following format:

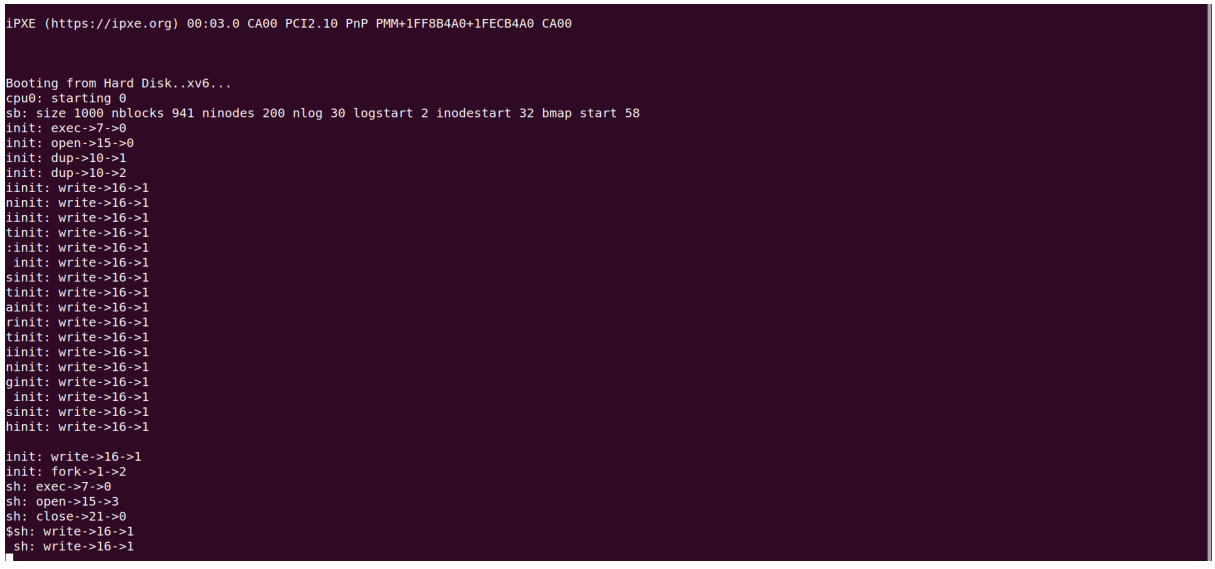
```
extern int <functionName>(<args>)
```

   It is also written in the array `static int (*syscalls[])(void)`
d. The function is also declared in the `user.h` file.
e. In `usys.S` file, the system call is given an entry in following format:

```
SYSCALL(<sysCallName>)
```

Part - 1:

A line is printed out for each system call invocation. To see the result of this, uncomment the lines in which all system call names are listed and also, uncomment the line 187 in which the result is being printed.Here is a result of the same:



Part - 2:

Time is printed out by making a system call named `mydate`. Here is a result of same:

```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00



Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap8
init: starting sh
$ mydate
Year: 2023
Month: 4
Date: 7
Hour: 10
Minute: 26
Second: 18
$
```

Part - 3:

Page table entries which are valid and are accessible in user mode are printed.
When a large size local array is declared, there is no change in number of entries:

```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00



Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap8
init: starting sh
$ mypgtPrint
Entry number: 0, Virtual address: 0x0, Physical address: 0xdee2000
Entry number: 2, Virtual address: 0x2000, Physical address: 0xdedf000
$
```

Whereas, when a large size global array is declared, the number of entries increases:

```


Booting from Hard Disk...
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap8
init: starting sh
$ mypgtPrint
Entry number: 0, Virtual address: 0x0, Physical address: 0xdee2000
Entry number: 1, Virtual address: 0x1000, Physical address: 0xdee0000
Entry number: 2, Virtual address: 0x2000, Physical address: 0xdedf000
Entry number: 3, Virtual address: 0x3000, Physical address: 0xdede000
Entry number: 4, Virtual address: 0x4000, Physical address: 0xdedd000
Entry number: 5, Virtual address: 0x5000, Physical address: 0xdedc000
Entry number: 6, Virtual address: 0x6000, Physical address: 0xdedb000
Entry number: 7, Virtual address: 0x7000, Physical address: 0xdeda000
Entry number: 8, Virtual address: 0x8000, Physical address: 0xded9000
Entry number: 9, Virtual address: 0x9000, Physical address: 0xded8000
Entry number: 10, Virtual address: 0xa000, Physical address: 0xded7000
Entry number: 12, Virtual address: 0xc000, Physical address: 0xded5000
$
```

When a large size global array is declared, its memory is allocated in the data section of the program's virtual memory space. When the array is of very large size, additional pages may be allocated to accommodate the data. Thus, the number of page table entries increases.
However, when a large size local array is declared, it is stored on the stack, which is a part of the process’s existing virtual memory space. Therefore, it doesn’t require additional pages to be allocated to store the variables. Thus, the number of page table entries does not increase when a local array is declared.
When a system call is executed various times, the virtual address remains the same, however the physical address changes at every execution because of demand paging i.e, a page is allocated to a process only when it is accessed. One more possible reason can be address space randomization which is used as a security feature to protect the OS against attacks.