Operating Systems-2

Programming Assignment 5: Syscall Implementation

Rajiv Shailesh Chitale
CS21BTECH11051

a) Working of Syscalls

When the user process makes a syscall, it places its parameters on the stack (from %esp+4), in accordance with the function declarations in “user.h”.

The function definitions are in assembly code in usys.s. The functions place the syscall number in the %eax register, and send an interrupt signal to the kernel.

In the kernel mode, syscall.c begins execution. It reads the syscall number and calls the required one from an array of syscalls. The syscalls are not directly fed arguments.

A syscall has to read its arguments beginning from %esp+4. It can run and call other helper functions. It returns an integer which syscall.c places in the %eax register.

From this assignment I learnt,

- Passing of arguments and return values between user and kernel.
- How to send an interrupt using assembly code. And that it can be used along with C code.
- That gcc’s preprocessor can be used on assembly files, and about the ## operator.
- The layout of bits and flags in the entries of page directory/tables. And how to efficiently manipulate them using bitshifts and bitmasks.
- The constant mapping between virtual address to physical address in case of kernel space. Also that kernel and user memory can lie very close physically. (I noticed that the physical address of the page table and pages were neighbouring despite virtual addresses being far apart.)
- How page table entries are mapped to a physical address taken from the freelist.
- That xv6 stack is limited to one page.
- What kind of data is maintained by kernel structs for processes and cpu.
- That clock time can be read from ports present in each processor.

b) 1. When changing the size of the global array of integers (4 bytes each), the following is observed. The number of pages is initially 2 (one is for the stack). The number of pages increases almost linearly and it is three orders of magnitude lower than the number of elements in the array. This is because 1024 elements in the array equate to 4KB which is the size of a page.

Size of Global Array	Number of Pages
0	2
100	2
1000	3
10,000	12
100,000	100
1000,000	978

2. When creating local arrays, we have to use them after the pgtPrint() call. Otherwise the compiler discards them. Further, the stack size is equal to one page size in xv6, ie. 4KB. When increasing the size of the local array, this is exceeded leading to a trap about a page fault.

Size of Local Array (N)	Number of Pages
$N \leq 956$	2
$N > 956$	Trap 14 (page fault)
Any size, if array is unused	2

3. On repeated execution of the program, the number of pages does not change. This is because the requirement of logical addresses by the code is the same and the page size is also fixed.

On multiple repetitions (or executing after some other programs have been run), we find that the virtual address remains the same but the physical address keeps changing. A program only sees the virtual address and it does not differ in its execution based on the availability of physical memory. The compiled code always accesses the same logical addresses.

But the physical addresses which are to be mapped to virtual addresses are decided using a freelist. When a process is allocated memory, physical addresses are taken out of the list to be mapped to pages. When a process frees memory, the corresponding physical addresses are added back to the list. When multiple processes are run, the available addresses and their order in the freelist keeps changing. This is why the physical address of pages of the same process changes over different iterations.