System calls

Saptarshi Ghosh and Mainack Mondal

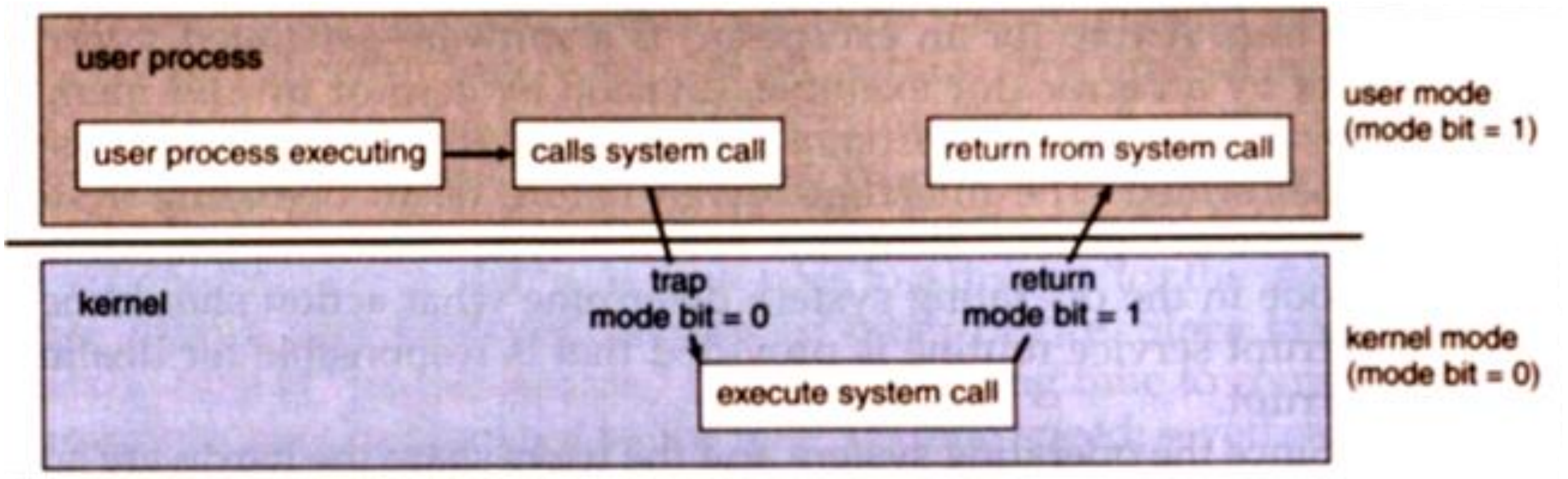
CS39002

Spring 2020-21

Some slides borrowed from slides of Don Porter, Univ of North Carolina (UNC)

What are system calls?

- The mechanism used by an application program to request service from the operating system
  - So how does it work?



This lecture

- How system calls work
  - Overview of interrupts
  - Earlier days – syscalls handled similar to interrupts
  - Nowadays – how syscalls work
- Examples of common syscalls

Earlier days: interrupt

- Originally, system calls issued using “int” instruction and handled similar to interrupts
  - Let us see a brief overview of how interrupts work

Interrupts

Interrupts

- Two types of interrupts usually provided by a processor

Interrupts

- Two types of interrupts usually provided by a processor
- Synchronous – will happen every time an instruction executes with a given program state
    - Divide by zero
    - Bad pointer dereference
    - Application wants some service (syscall)

Interrupts

- Two types of interrupts usually provided by a processor
- Synchronous – will happen every time an instruction executes with a given program state
  - Divide by zero
  - Bad pointer dereference
  - Application wants some service (syscall)
- Asynchronous – caused by an external event
  - By I/O device, timer, etc

Interrupts: Intel nomenclature

- Two types of interrupts usually provided by a processor

- Synchronous – will happen every time an instruction executes with a given program state
  - Divide by zero
  - Bad pointer dereference
  - Application wants some service (syscall)

Exceptions

- Asynchronous – caused by an external event
  - By I/O device, timer, etc

Interrupts

Interrupts: Intel nomenclature

- Two types of interrupts usually provided by a processor
- Synchronous – will happen every time an instruction executes with a given program state
  - Divide by zero
  - Bad pointer dereference
  - Application wants some service (syscall)

Exceptions

- Asynchronous – caused by an external event
  - By I/O device, timer, etc

Interrupts

From programmer's perspective, handled with the same abstractions

How interrupts work

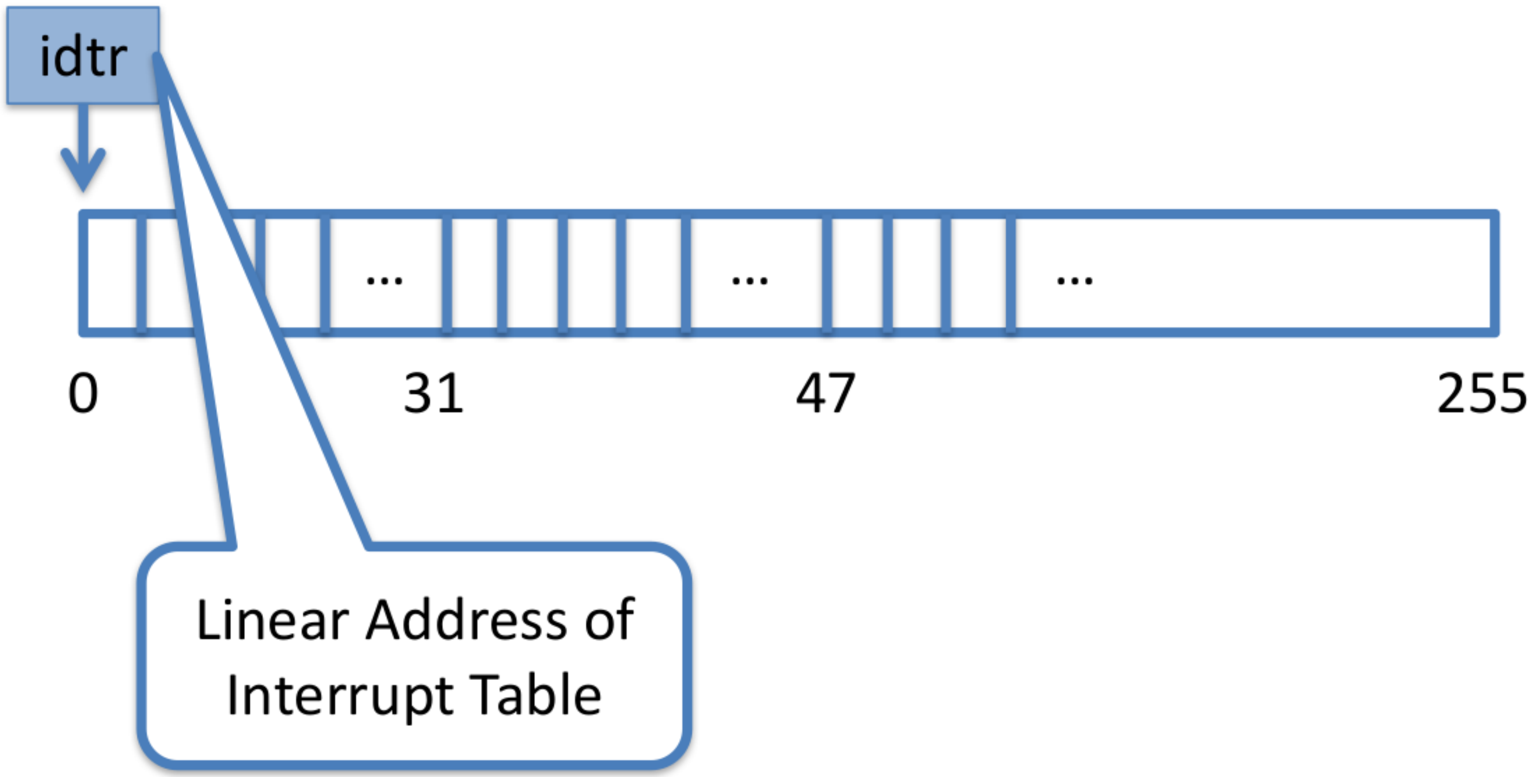
- Each interrupt / exception includes a number indicating its type (e.g., 14 is a page fault)
  - Kernel creates an array of Interrupt descriptors in memory, called Interrupt Descriptor Table, or IDT
  - The number is index into the IDT

How interrupts work

- Each interrupt / exception includes a number indicating its type (e.g., 14 is a page fault)
    - Kernel creates an array of Interrupt descriptors in memory, called Interrupt Descriptor Table, or IDT
    - The number is index into the IDT

- IDT can be anywhere in memory
- Pointed to by a special register (idtr)
- Each IDT entry *j* specifies things like
    - What code to execute for interrupt *j* (called interrupt handler)
    - Privilege level necessary to raise the interrupt (e.g., a user program cannot raise interrupt for page fault)

Example: x86 interrupt table



Software interrupts

- The "int <num>" instruction allows software to raise an interrupt
- What happens if a program issues "int <num>"?

Software interrupts

- The "int <num>" instruction allows software to raise an interrupt
- What happens if a program issues "int <num>"?
  - Control jumps to the kernel at a prescribed address (the interrupt handler)
  - Address of interrupt handler found using the IDT (index num)
  - The register state of the program is saved by the kernel
  - Interrupt handler runs and handles the interrupt
  - When handler completes, resume program

An important concept

- The state of a program's execution is succintly and completely represented by the CPU register state
- Pause a program (so that some other program can be allocated the CPU)
  - Dump the register contents into memory
- Resume a program
  - Read the register contents back from memory to the CPU

System calls

Earlier days

- Originally, system calls issued using “int” instruction

Earlier days

- Originally, system calls issued using “int” instruction
    - The system call handler routine was just an interrupt handler

Earlier days

- Originally, system calls issued using “int” instruction
  - The system call handler routine was just an interrupt handler
  - Like interrupts, system calls used to be arranged in a table

Earlier days

- Originally, system calls issued using “int” instruction
  - The system call handler routine was just an interrupt handler
  - Like interrupts, system calls used to be arranged in a table

- Whenever a syscall (interrupt driven) came

Earlier days

- Originally, system calls issued using “int” instruction
    - The system call handler routine was just an interrupt handler
    - Like interrupts, system calls used to be arranged in a table
- Whenever a syscall (interrupt driven) came
    - Program selects which syscall it wants by placing index in a particular register (eax)

Earlier days

- Originally, system calls issued using “int” instruction
    - The system call handler routine was just an interrupt handler
    - Like interrupts, system calls used to be arranged in a table

- Whenever a syscall (interrupt driven) came
    - Program selects which syscall it wants by placing index in a particular register (eax)
    - Arguments go in other registers as per convention
    - Return value goes in eax

Earlier days: summary

- Kernel leveraged interrupts to handle system calls

However this mechanism is slow

- Today, processors are totally pipelined
  - Pipeline stalls are very expensive
  - Cache misses can cause pipeline stalls

- Recall that IDT is in memory
  - May not be in cache
  - Cache misses makes it expensive, e.g., system calls said to take almost twice as long from P3 to P4

Idea: new instruction

- What if we cache the IDT entry for a system call in a special CPU register?
    - No more cache misses for the IDT!

- What is the cost?

Idea: new instruction

- What if we cache the IDT entry for a system call in a special CPU register?
  - No more cache misses for the IDT!

- What is the cost?
  - system calls should be frequent enough to be worth the transistor budget

How to leverage the new instruction?

- There is a machine instruction (new architectures)
  - Essentially for asking your processor to perform task
  - Hardware specific
  - Can be called “syscall”, “trap”, “svc”, “swi”
  - For x86-64 architecture it is called “syscall”

Example of syscall

- syscall machine instruction takes operands
  - syscall 10 // integer number for x86
    // some of these are fixed by intel

- The instruction used machine-specific registers that store syscall entry point (into kernel code) for every syscall
- Arguments, return value go in other registers as per convention

System calls in programs

Syscalls in programs

- When we are writing programs in HLL (higher level language, think C)
    - We think of syscalls in a somewhat higher level context

Characteristics of system calls

- Provide an interface to the services made available by an operating system

Characteristics of system calls

- Provide an interface to the services made available by an operating system
  - Typically executes hundreds of thousands time every second

Characteristics of system calls

- Provide an interface to the services made available by an operating system
  - Typically executes hundreds of thousands time every second
  - User application programs do not see this level of detail

Characteristics of system calls

- Provide an interface to the services made available by an operating system
    - Typically executes hundreds of thousands time every second
    - User application programs do not see this level of detail
    - Use Application programming interface (API)

Characteristics of system calls

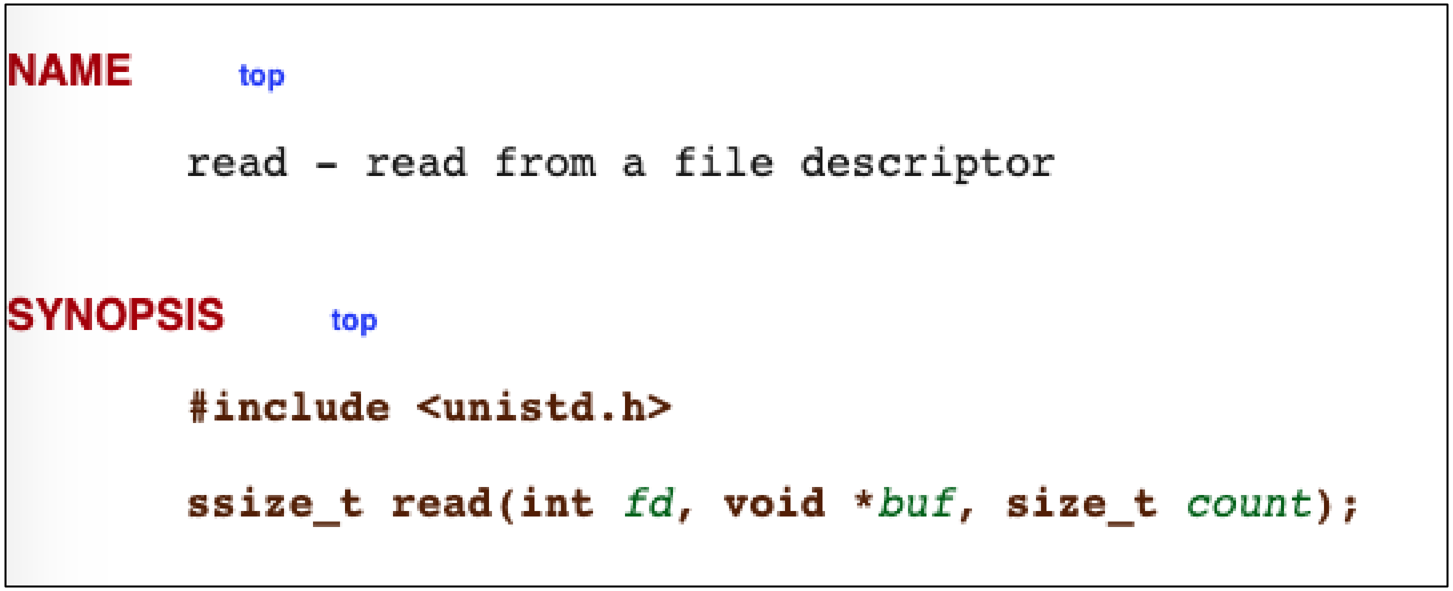
- Provide an interface to the services made available by an operating system
    - Typically executes hundreds of thousands time every second
    - User application programs do not see this level of detail
    - Use Application programming interface (API)
    - fork, pipe, execvp etc.

Characteristics of system calls

- Provide an interface to the services made available by an operating system
    - Typically executes hundreds of thousands time every second
    - User application programs do not see this level of detail
    - Use Application programming interface (API)
    - fork, pipe, execvp etc.
    - These APIs are also loosely termed as system calls

Example of a system call API

- There are no fopen, fgets, printf, and fclose system calls in the Linux kernel but open, read, write, and close
- http://man7.org/linux/man-pages/man2/read.2.html



NAME top

```
       read - read from a file descriptor
```

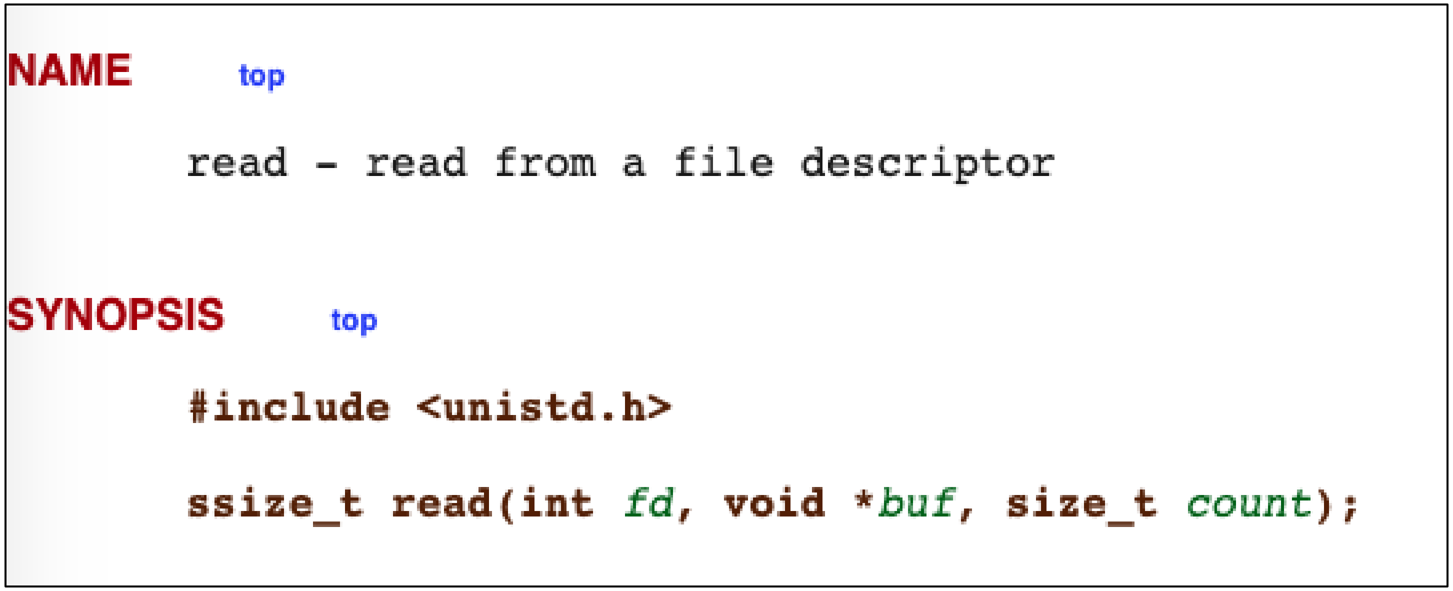
SYNOPSIS top

```
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);
```

Example of a system call API

- http://man7.org/linux/man-pages/man2/read.2.html



NAME top

```
       read - read from a file descriptor
```

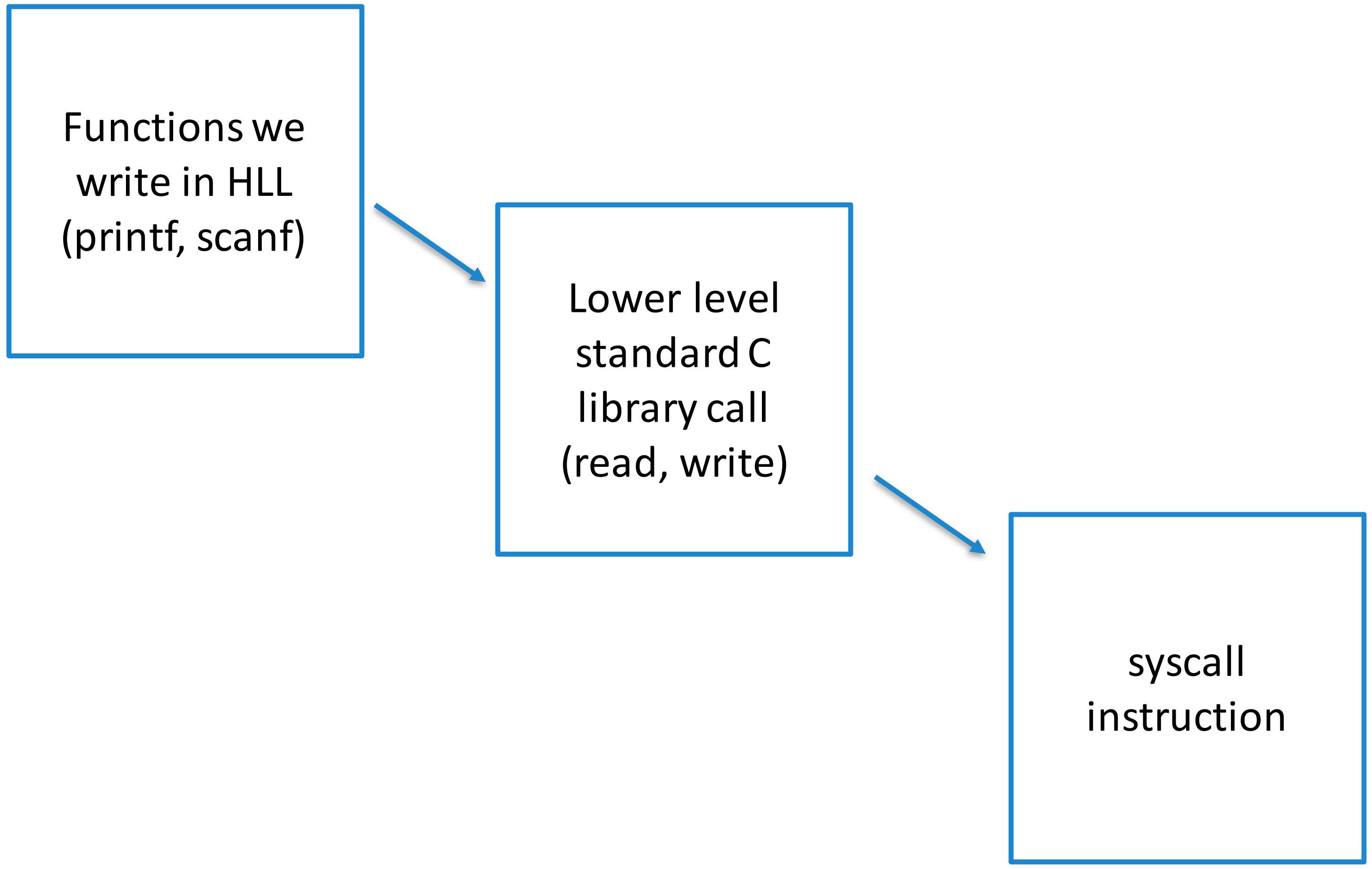
SYNOPSIS top

```
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);
```

- What are these function parameters?
- What is the return values?

The workflow of a syscall


Functions we write in HLL (printf, scanf)
Lower level standard C library call (read, write)
syscall instruction

Types of system calls

(From silberschatz’s slides)

Types of System Calls

- Process control (e.g., fork(), exit(), wait())
  - create process, terminate process (fork, exit)
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - Debugger for determining bugs, single step execution
  - Locks for managing access to share data between processes

Types of System Calls (Cont.)

- File management (e.g., open(), close(), read(), write())
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management (e.g., ioctl(), read(), write())
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

Types of System Calls (Cont.)

- Inter-Process Communications (e.g., pipe(), semget(), semop(), shmget(), shmcat(), shmdt(), shmctl(), signal(), kill())
  - create, delete communication connection
  - send, receive messages if message passing model to host name or process name
  - Shared-memory model create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices

Types of System Calls (Cont.)

- Protection (chmod(), chown(), umask())
    - Control access to resources
    - Get and set permissions
    - Allow and deny user access

More about system calls

- System call numbers in linux for x86 :
  - https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/arch/x86/entry/syscalls/syscall_32.tbl
- System call numbers in linux for x86 - 64:
  - https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/arch/x86/entry/syscalls/syscall_64.tbl
- System call implementations in x86-64 and x86
  - https://stackoverflow.com/questions/15168822/intel-x86-vs-x64-system-call