Linux System Calls

W’s of System Calls

User programs vs Kernel programs

- Runs in different spaces
- Runes with different privileges
- User space not allowed to access kernel space
- But they need kernel services

OS provides service points

- For user programs
- To request services from the kernel

In Linux these are called **System calls**

- About 300 in count
- Listing : /usr/include/asm/unistd.h
- Provide layer between
  - Kernel space (typically HW)
  - User space (typically user process)
- Serves three purposes
  - Provides a an abstracted HW interface to user space
  - Ensure system Security and stability
  - Makes Process Management easier

Working of a Linux System Call

- Implemented as an ordinary function in the Linux Kernel
- Executes like others in the Kernel Space
- However, the call to that function isn't ordinary
- When a user program makes a system call
  - Arguments are packaged up and handed to the kernel
  - A special procedure is required to transfer control to the kernel
  - Kernel takes over execution of the program until the call completes
  - Kernel transfers control back to the program with return value
- Special procedure is typically achieved using “trap”

Linux System Call Wrappers

- Every System Call has standard steps
- GNU C library (glibc) abstracts them
  - By wrapping with functions of same name
  - For easy invocation
- Examples
  - I/O functions: open, read, ...
- We rarely invoke direct system calls
  - But rather these system call (wrapper) functions
- Any Exception?
  - Custom defined system call – using syscall(sno, ...)

Contrast with a Library Function

- A library function is an ordinary function
- It resides in a library external to the program
  - But in the User Space only
- Moreover, the call to it is also ordinary
  - Arguments placed in processor registers or the stack
  - Execution transferred to the start of the function
    - Typically resides in a loaded shared library
    - In the User Space only
- Examples
  - fopen, printf, getopt, mkstemp (all from glibc)

Return Values

- Library functions often return pointers
  - Example: FILE * fp = fopen("harry","r");
  - NULL indicates failure
- System calls usually return an integer
  - Example: int res = open(“harry”, O_RDONLY);
  - Return value
    - >= 0 indicates success
    - < 0, typically -1 indicates failure, and error is set in errno
- Note the counter intuitive return of System Calls
  - Opposite way round
  - Cannot use as Boolean

System calls in Linux -examples

The operating system is responsible for

- Process Management (starting, running, stopping processes)
- File Management(creating, opening, closing, reading, writing, renaming files)
- Memory Management (allocating, deallocating memory)
- Other stuff (timing, scheduling, network management)
- An application program makes a system call to get the operating system to perform a service for it, like reading from a file.

One nice thing about syscalls is that you don't have to link with a C library, so your executables can be much smaller.

System calls in Linux -examples

exit - terminate current process		
In	eax	1
	ebx	return code
Out	*(This call does not return)*	
fork - create child process		
In	eax	2
Out	eax	0 in the clone; process id of clone or EAGAIN or ENOMEM in caller
read - read from file or device		
In	eax	3
	ebx	file descriptor
	ecx	address of the buffer to read into
	edx	maximum number of bytes to read
Out	eax	number of bytes actually read | EAGAIN | EBADF | EFAULT | EINTR | EINVAL | EIO | EISDIR
write - write to file or device		
In	eax	4
	ebx	file descriptor
	ecx	address of the buffer to write from
	edx	maximum number of bytes to write
Out	eax	number of bytes actually sent | EAGAIN | EBADF | EFAULT | EINTR | EINVAL | EIO | ENOSPC | EPIPE

Tracing System Calls

- Command: strace <program> [args]
- Traces the execution of <program>
- And Lists
  - System Calls made by <program>
  - Signals received by <program>
- Controlled by various options
  - An interesting one is “-e”
- Example
  - strace cat /dev/null

Pros & Cons

- Pros
  - System calls provide direct & hence more control over the kernel services
  - Library functions abstract the nitty-gritty of architecture or OS specific details of the system calls
  - Library functions can provide wrappers over repeated set of system calls
- Cons
  - Library functions may have overheads
  - System calls at times may expose the underlying system dependency

Let's try some Examples

- System Call Invocation
- System calls vs Library functions
  - File Operations
- Observe the various system calls invoked
  - Use strace