Operating Systems Fundamentals

Operating System Services, System Calls and the API

Operating System Services

Recall that these are:

- User interface (e.g. look and feel)
- Program execution
- I/O operations
- File-system manipulation
- Communications
- Error detection
- Resource allocation
- Accounting
- Protection and security

System Calls

- Programming interface to the services (on the previous page) are provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level

  Application Program Interface (API)

  rather than direct system call use

API's

- Common APIs are
  - **Win32 API** for Windows,
  - **POSIX API** for POSIX-based systems including virtually all versions of

    UNIX, Linux, and Mac OS X
  - **Java API** for the Java virtual machine (JVM)
  - **Android API** for android devices

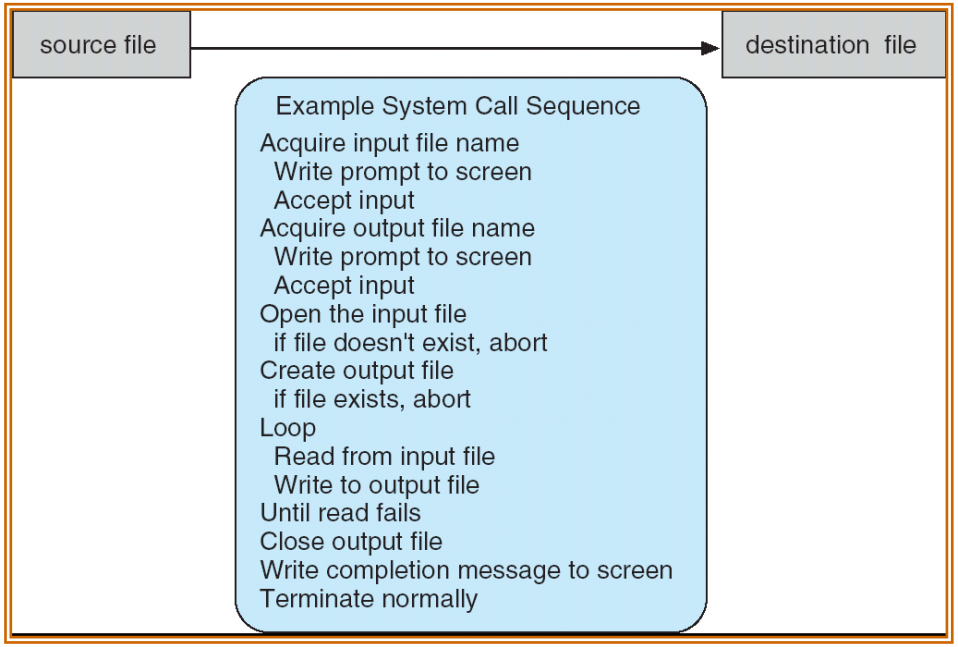
APIs Vs System calls?

1. API Portability
   - Should compile and run on any system that supports the same API

2. System calls can be more detailed and difficult to work with.
   - Typically not as portable

Example of System Calls

- A system call sequence to copy the contents of one file to another file



- What are the system calls here?

Example of System Calls

- The first input that the file will need is the names of the 2 files (i/p and o/p)
- One approach is for the program to ask the user of the names of the 2 files.

  In an interactive system this approach will require a sequence of system calls, first to write a prompting message on the screen and then to read from the keyboard the characters that define the 2 files.
- Once the 2 file names are obtained, the program must open the input file and create the output file. Each of these operations requires another system call.

Example of System Calls

- There are also possible error conditions for each operation.

  For example, when the program tries to open the input file, it may find that there is no file of that name or that the file is protected against access.

- In these cases, the program should print a Message on the console (another sequence of system calls) and then terminate abnormally (another system call).

- If the input file exists, then we must create a new output file. We may find that there is already an output file with same name and this may cause the program to abort (another system call) else we create a new o/p file (another system call).

Example of System Calls

- Now that both files are set up, we enter a loop that reads from the input file (SC), and writes to the output file (SC). These operations may lead to possible errors (What??) again invoking SCs.
- Finally, after the entire file is copied, the program closes both file (SCs), write a message to console (SC), and finally terminate normally (final SC).
- So…. As we can see, even SIMPLE programs make heavy use of system calls in the OS.
- Hence there are 1000's of sys calls per second. An application programmer who operates at the API high level will not see this level of detail, but an OS programmer does need to understand it

Example, continued

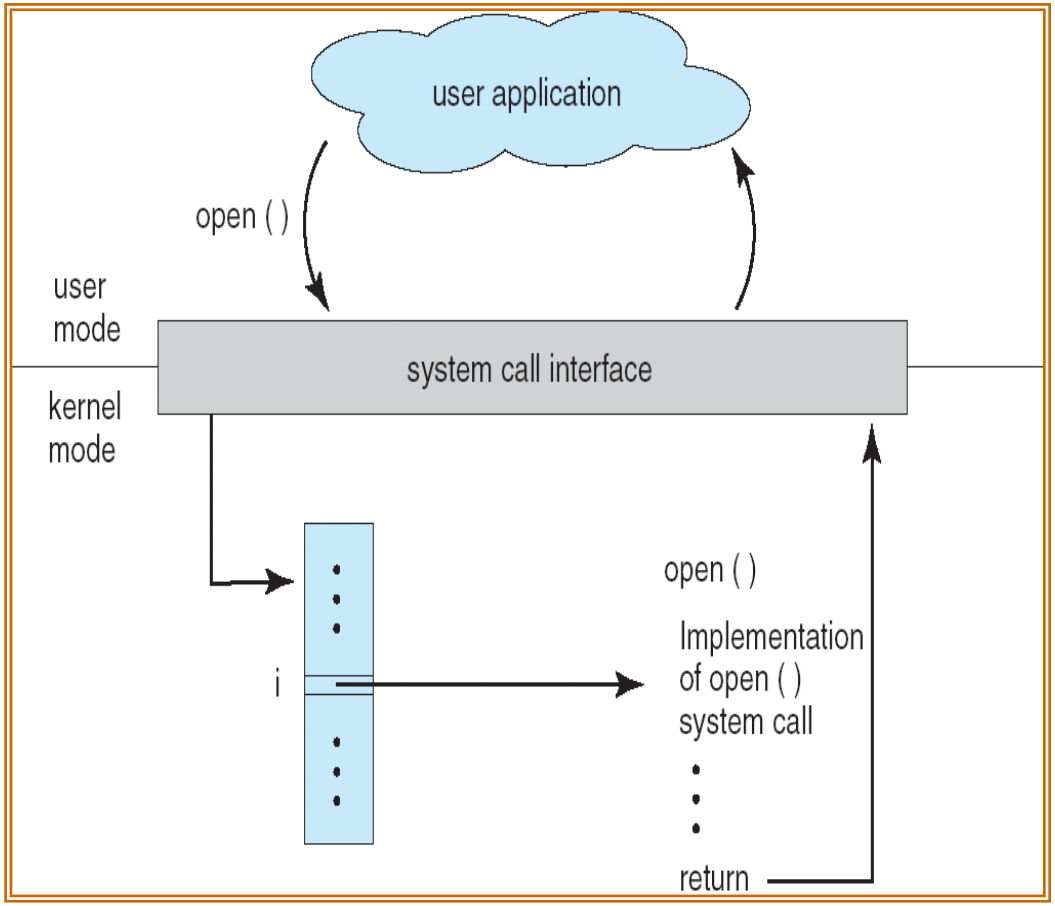
- As we can see from the previous example, even simple programs make heavy use of the operating system.

- Frequently, systems execute thousands of system calls *per second*.

System Call Implementation

- Typically, a number is associated with each system call
  - An indexed table is maintained by the **System Call Interface**
- The **system call interface** invokes the intended system call in the OS kernel and returns the status of the system call and any return values.
- The caller need know nothing about how the system call is implemented
  - Caller just needs to obey the API and understand what OS will do as a result
  - Most details of the OS interface is hidden from the programmer by the API

API – System Call – OS Relationship

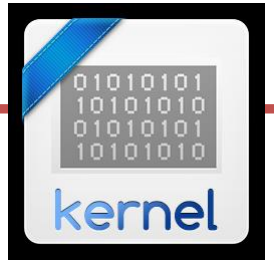


An illustration of how the OS handles a user application invoking the system call open()

Well actually…

The app calls the **API** of the system call

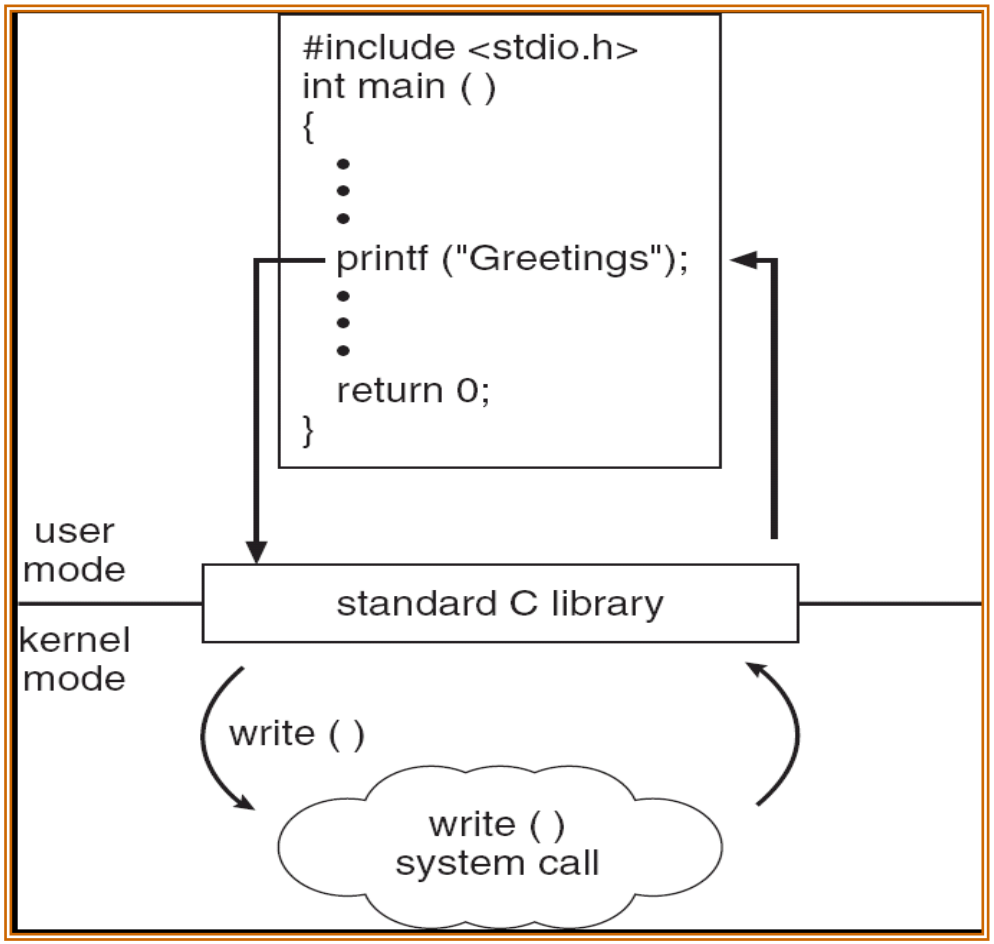
User Mode and Kernel Mode



- Recall that this is to distinguish between the execution of user-defined code and operating system code
- A “mode bit” is added to the hardware of the computer to indicate the current mode: kernel(0) or user(1)
- The “mode bit” is also known as the ‘Protection Ring’ or the ‘Supervisor mode’
- When a user application requests a service from the OS, it must switch from user to kernel mode to fulfil the request

https://www.geeksforgeeks.org/difference-between-user-mode-and-kernel-mode/

Standard C Library Example



- C program invoking printf() **library call**, found in the library glibc, in **user** mode
- which in turns calls the write() **system call** in **kernel** mode

System Call Parameter Passing

- Often, more information is required than simply the identity of the desired system call
- This information is passed in **parameters**
- This concept is same as passing parameter(s) to Python/Java methods
- The only different is the programming language
  - Typically the C programming language is used for OS system calls/API's

System Call Parameter Passing

Three general methods used to pass parameters to the OS

1. ***Registers***
2. ***Stack***
3. ***Block,*** or table

Block and stack methods do not limit the number or length of parameters being passed

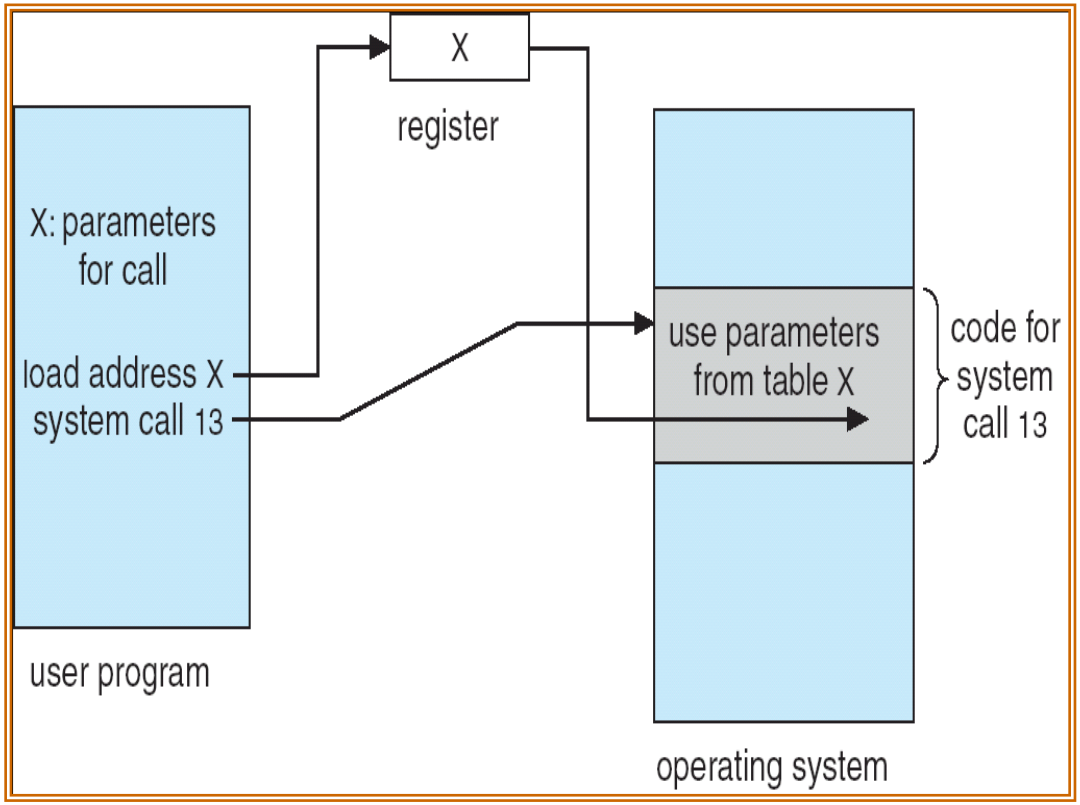
Why is there a limit on the Registers?

Register and Stack param passing

1. **Register**
   Simplest and Fastest:
   pass the parameters in ***registers***
   but registers are limited in number
   Where are these registers? …

2. **Stack**
   Parameters placed, or ***pushed,*** onto the ***stack***
   by the program calling the system call
   and ***popped*** off the stack by the operating system
   Why does the OS and not the calling program pop the params?

Parameter Passing via Table



- Address of block is X, this address is passed in a register
- This approach is taken by Linux

Types of System Calls

Type	Examples
• Process control	• exit, abort, load, execute, create, wait
• File management	• Create, delete, open, close, read, write, reposition
• Device management	• Request device, release device, read(from), write(to)...
• Information maintenance	• Get time/date, set time/date, get process/file/device attributes
• Communications	• Create/delete communications connection, send/receive messages, attach/detach remote devices
• Protection	• Control access to resources, Get and Set permissions, Allow/Deny access

Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Unix's system calls tend to be simpler than the Win32 API

System Programs

- System programs provide a convenient environment for program development and execution.

Some examples are:

- Desktop User Interface
- File manipulation
  - Create, delete, copy, rename
- Programming language support
  - Compilers, assemblers, debuggers
- Program loading and execution
  - Loaders, linkers, debuggers
- Communication between programs

System Programs

- Most users' view of the operation system is defined by system programs, not the actual system calls.
  - Which are abstracted under the system programs
- Normal Users will remember the User Interface
  - e.g. Apple products versus Android products
- Programming type users (s/w developers, IT support) will most likely be drawn to developer or scripting tools