Operating Systems Assignment 4

Meenakshi Madhu B180390CS

Group 4

Topic:

System call-working (From application program to until IRETURN) with process related system calls as examples.

1 System Calls

1.1 Introduction

System calls provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly), may need to be written using assembly-language instructions.

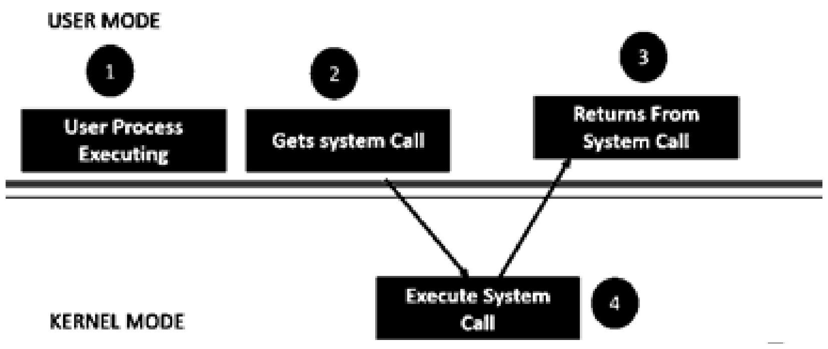


Fig: Architecture of a system call

In a general system, there is a lowest level- hardware, above that is the OS, above that is the application program. Application program can interact with OS via the OS interface, which is the system call interface i.e. application program interacts with OS via system calls.

For e.g. we have system calls open, close, read, write, etc available for file related system calls, and fork etc which are process related system calls.

We can invoke these system calls without knowing their underlying implementation in the OS. We implement this using the software interrupt instructions available in the machine.

For e.g.: Linux systems use int 80h and DOS and Windows OS uses int 21h to implement the system calls.

The OS defines the ISR (interrupt service routine) corresponding to those interrupts. ISR is a set of functions defined in the OS kernel.

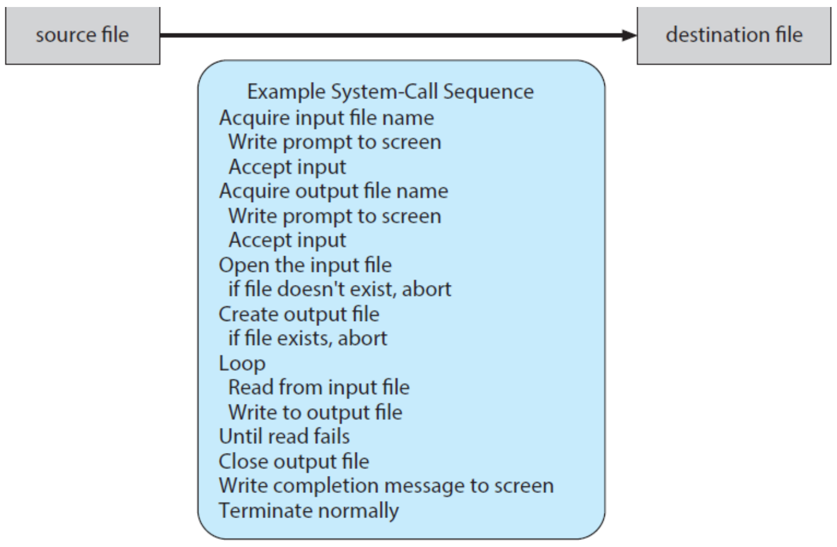


Figure 2.5 Example of how system calls are used.

2 Application Program Interface

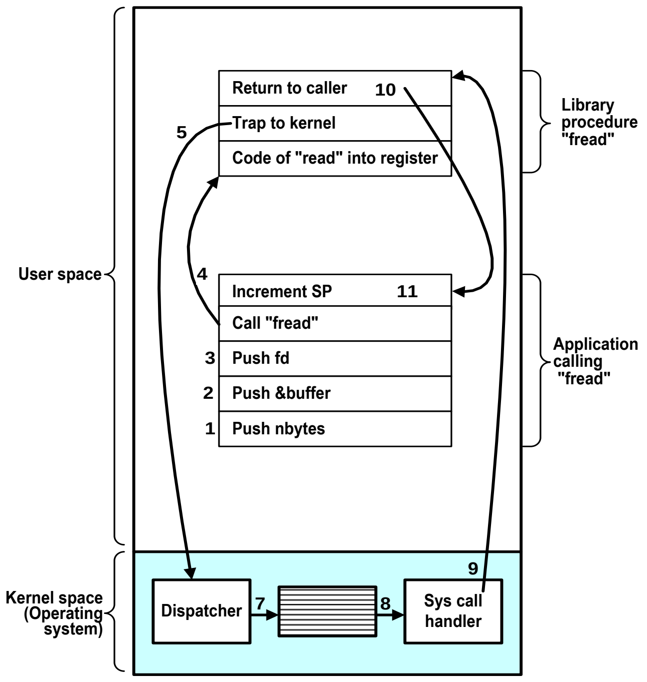
When we invoke a system call from an application program, it actually invokes an Application Program Interface (API) rather than a direct system call.

Some standard APIs are: Win32 API for Windows, POSIX API for POSIX-based systems (including all versions of UNIX, Linux and Mac OS X) and Java API for the Java virtual machine (JVM).

Advantages of using APIs rather than native system calls:-

- Portability: as long as a system supports an API, any program using that API can compile and run.
- Ease of Use: using the API can be significantly easier than using the actual system call, as the user need not know the underlying implementation of the system calls.

Taking an example to learn more about this, consider the implementation of a system call for file reading.



11 steps to execute the service

fread (fd, buffer, nbytes)

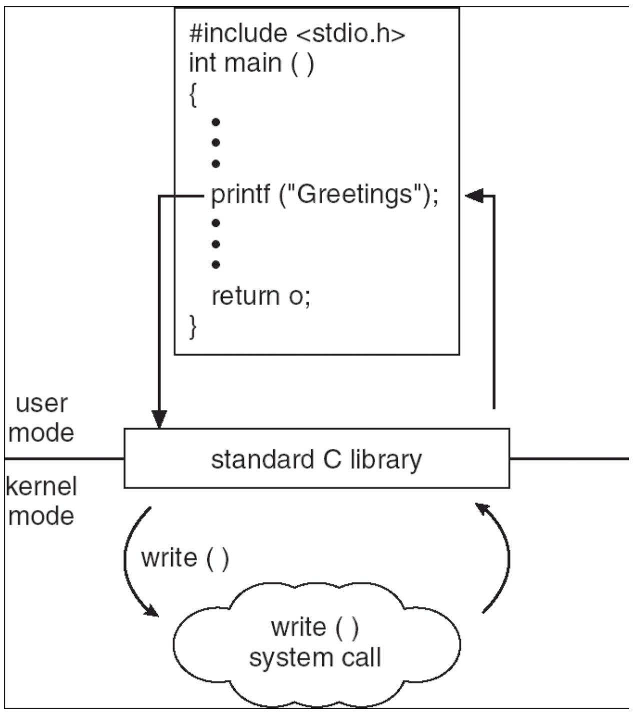
The steps when the application program makes a system call are:

- A system library routine is called.
- It transforms the call to the system standard (native API) and traps to the kernel.
- Control is taken by the kernel running in the system mode.
- According to the service code, the *Call dispatcher* invokes the corresponding responsible part of the Kernel.
- Depending on the nature of the required service, the kernel may block the calling process (for eg: reading from user, with scanf).
- After the call is finished, the calling process execution resumes obtaining the result (success/failure) as if an ordinary function was called.

3 System Call Implementation

- A number is associated with each system call.
  - System call interface maintains a table indexed according to these numbers.
  - This table contains pointers that point to the corresponding interrupt service routine to the system call.
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need not know details of how the system call is implemented.
  - Only need to obey API and understand result of call.
  - Most details of OS interface are hidden from the programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)
  - Compiler generates the call to the wrapper function in the library. And the library routine will actually call the system call.

3.1 Standard C Library Example


```
#include <stdio.h>
int main ( )
{
   •
   •
   •
   printf ("Greetings");
   •
   •
   •
   return o;
}
```

C program invoking printf() library call, which calls write() system call.

This is used to write output to the monitor, and hence is an I/O operation. So, we need to invoke a sys call.

The printf() call is converted into a standard C library function, ie. it invokes a standard C library function in the C library.

And the C library invokes the generic system call, corresponding to printf() which is the write() system call.

The OS executes the corresponding ISR for the write() system call and control returns back to the library, and from the library, the normal return function returns the control back to the user program.

3.2 Parameter Passing Mechanism

We need to pass parameters to the system calls. For eg: for the file read system call, we passed three parameters, the file handler, buffer address and the number of bytes from the user level program.

How?

- Pass the parameters in registers.
  - Registers are hardware entities that can be accessed from both user as well as OS area.
  - Can be used for simple parameters like integers, floating points or characters.
  - Can be used when the number of parameters is limited.
- Stored in memory and the address of the memory is passed as parameter in a register.
  - For open() call when a path name is the parameter, we can store the entire path in a block of memory and store the address of the memory block in the register.
- Parameters are pushed on to the stack by the program and OS can pop off from the stack.
  - Same as previous case except that the registers are implicit registers ie. the stack pointers.

4 Types of System Calls

- **Process Control and IPC (Inter-Process Communication)**
  - end, abort
  - load, execute
  - create, terminate
  - get/set attributes
  - wait for time, event
  - signal event
  - allocate/free memory

- **<u>Memory Management</u>**
  - allocating and freeing memory space on request.
  - malloc()
  - free()
- **<u>File Access</u>**
  - To access a file it must first be opened using open() call that returns a file descriptor (fd).
  - open/close, read/write
  - lseek (move the read/write file offset)
  - dup (duplicate file descriptor)
  - stat (get file status)
  - chmod (change mode of a file)
  - pipe (create an inter process communication channel)
- **<u>File and File-System Management</u>**
  - create/delete
  - open/close
  - read/write/reposition
  - get/set attributes
  - Some common system calls are create, delete, read, write, reposition, or close. Also, there is a need to determine the file attributes – get and set file attribute. Many times, the OS provides an API to make these system calls.
- **<u>Device Management</u>**
  - request/release device
  - read/write/reposition
  - get/set attributes
  - logically attach/detach
  - Process usually require several resources to execute, if these resources are available, they will be granted and control returned to the user process. These resources are also thought of as devices. Some are physical, such as a video card, and others are abstract, such as a file.
  - User programs request the device, and when finished they release the device. Similar to files, we can read, write, and reposition the device.
- **<u>Information Maintenance</u>**
  - get/set time/date/system data
  - get/set process, file, device attributes
  - Some system calls exist purely for transferring information between the user program and the operating system. An example of this is time, or date.
  - The OS also keeps information about all its processes and provides system calls to report this information.
- **<u>Communications</u>**
  - create/delete communication connection
  - send/receive messages
  - There are two models of inter process communication, the message-passing model and the shared memory model.
    - Message-passing uses a common mailbox to pass messages between processes.

    - Shared memory use certain system calls to create and gain access to create and gain access to regions of memory owned by other processes. The two processes exchange information by reading and writing in the shared data.
- <u>**Other calls**</u>
  - kill (send a signal to a process or a group of processes)
  - signal (a signal management)
- <u>**Other services**</u>
  - e.g. profiling, debugging etc.

<u>4.1 Process Control Calls</u>

- **fork() -create a new process**
  - pid = fork() ;
  - The fork() function creates a new process. The new process (child process) will be an exact copy of the calling process (parent process) except some process' system properties.
  - It returns 'twice'
    - return value == 0 for the child
    - return value > 0 for the parent (returned value is the child's *pid*)
- **exit() – terminate a process**
  - void exit(int status);
  - The exit() function then flushes all open files with unwritten buffered data and then closes all open files.
  - Finally, the process is terminated and system resources owned by the process are freed.
  - The value of 'status' is available to a waiting parent process
  - The exit() function should never return
- **wait, waitpid – wait for a child process to stop or terminate**
  - pid = wait(int *stat_loc) ;
    pid = waitpid(pid_t pid, int *stat_loc, int options) ;
  - The wait() and waitpid() functions suspend the calling process and obtain status information pertaining to one of the caller's child processes.
  - Various options permit status information to be obtained for child processes that have terminated or stopped.
- **execl, execle, execlp, execv, execve, execvp – execute a file**
  - int execl(const char *path, const char *arg0, …) ;
  - The members of the exec family of functions differ in the form and meaning of the arguments.
  - The *exec* family of functions will replace the current process image with a new process image. The new image will be constructed from a regular, executable file called the 'new process image file'.
  - There will be no return from a successful exec, because the calling process image is overlaid by the new process image; any return indicates a failure.

References:

- Operating System Concepts, 10th Edition by Abraham Silberschatz, Peter B. Galvin and Greg Gagne
- https://www.geeksforgeeks.org/introduction-of-system-call/
- http://faculty.salina.k-state.edu/tim/ossg/Introduction/sys_calls.html