System Calls & API Standards

Contents

- Implementation of API
- System call types
- API Standards
- Process Control Calls

System Calls

Previously stated that user applications interact with the kernel via system calls

- Typically invoked via a trap instruction
  - An intentional software-generated exception
- The kernel registers a handler for a specific trap
  - **int $0x80** for Linux system calls
  - **int $0x2e** for Windows system calls
- Can't easily pass arguments to system calls on the stack
  - Trap instruction causes the CPU to switch operating modes (i.e.from user mode to kernel mode)
  - Different operating modes have different stacks

System Calls

Typically, arguments to system calls passed in registers, and the return-value(s) come back in registers

- One of the arguments is an integer indicating which system call to invoke
- e.g. on Linux and Windows, %eax is set to operation to perform
- e.g. on UNIX systems, sys/syscall.h specifies these numbers

- Obvious constraint: system-call arguments can't be wider than the registers
- Several possible approaches:
- Can split larger arguments across multiple registers
- Can store larger arguments in a struct, then pass a pointer to the struct as an argument

System Calls

The operating system frequently exposes system calls via
a standard library

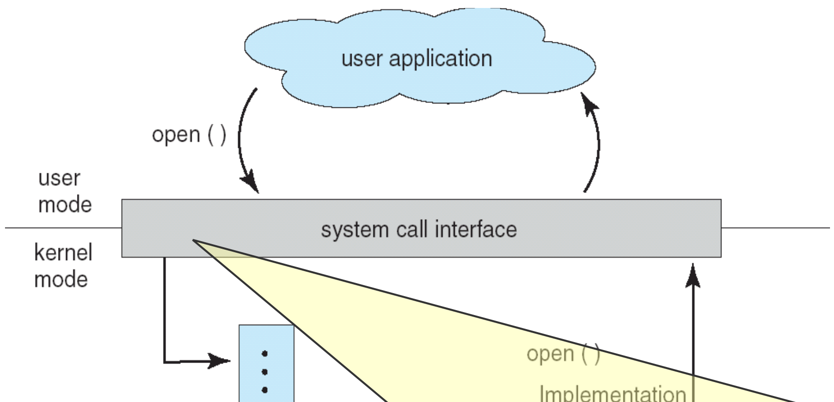
- e.g. UNIX syscalls are exposed via the C standard library (libc)
- e.g. Windows syscalls are exposed via the (largely undocumented) Native API (ntapi.dll)
- The library serves as an intermediary between apps and the operating system
- Some functions are direct wrappers for system calls

  e.g. ssize_t read(int fd, void *buf, size_t nbyte)

  Implementation stores arguments from stack into registers, invokes the system call entry-point (e.g. int $0x80), and returns result
- Others utilize system call wrappers internally

  e.g. malloc() is mainly implemented in user space, but uses system calls to increase the process' heap size

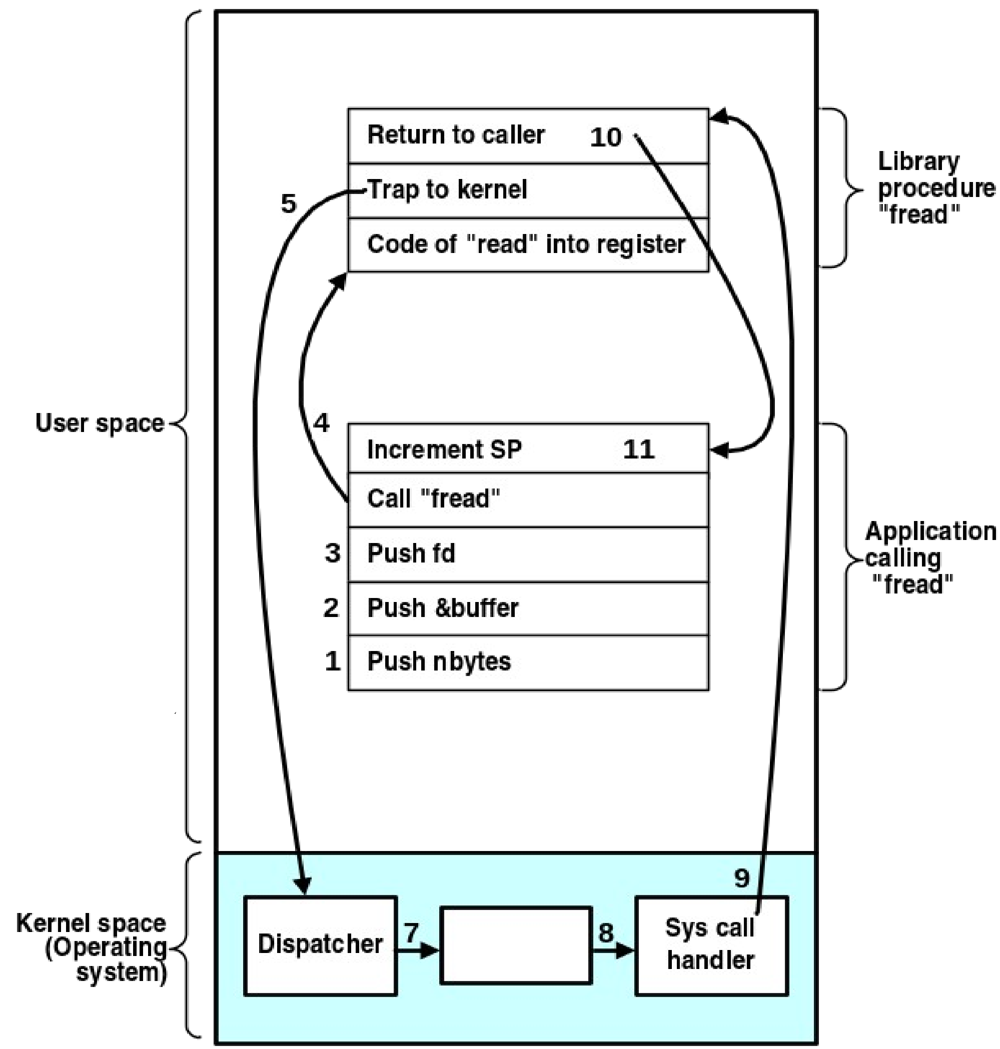
API – System Call Implementation



- The interface to the services provided by the OS has two parts:
  1. Higher language interface – a part of a system library
     - Executes in user mode
     - Implemented to accept a standard procedure call
     - Traps to the Part 2
  2. Kernel part
     - Executes in system mode
     - Implements the required system service
     - May cause blocking the caller (forcing it to wait)
     - After completion returns back to Part 1 (may report the success or failure of the call)

How the System Call Interface is Implemented

- **The application program makes a System Call:**
  - A system library routine is called first
  - 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 responsible part of the Kernel
  - Depending on the nature of the required service, the kernel may block the calling process
  - After the call is finished, the calling process execution resumes obtaining the result (success/failure) as if an ordinary function was called



11 steps to execute the service
fread (fd, buffer, nbytes)

System Calls

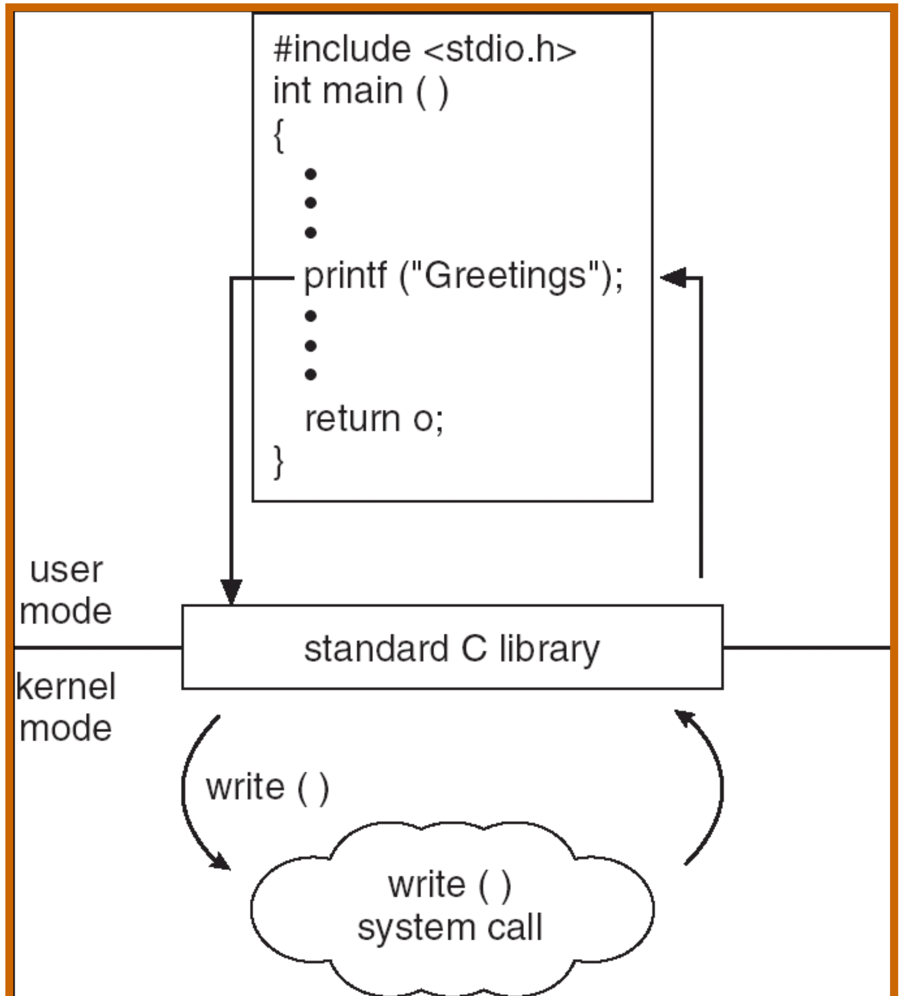
- Programming interface to the services provided by the OS
- Typically written in a higher-level language (C or C++)
- Mostly accessed by programs via a higher-level **Application Program Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- Why use APIs rather than the native system calls?

System Call Implementation

- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- 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 know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

Standard C Library Example

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



System API Standards

- Three most common API standards are
  - ***POSIX*** API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - ***Win32*** API for Windows
  - ***Java API*** for the Java virtual machine (JVM)
- POSIX (IEEE 1003.1, ISO/IEC 9945)
  - Very widely used standard based on (and including) C-language
  - Defines both
    - ***system calls*** and
    - compulsory ***system programs*** together with their functionality and command-line format
      - E.g. `ls -w dir` prints the list of files in a directory in a 'wide' format
  - Complete specification is at http://www.opengroup.org/onlinepubs/9699919799/nframe.html
- Win32 (Microsoft Windows based systems)
  - Specifies system calls together with many Windows GUI routines
    - VERY complex, no really complete specification

Parameter passing mechanism

- Pass the parameters in Registers
- Stored in in memory and the address of the memory is passed as parameter in a register
- Parameters are pushed on to the stack by the program and OS can poped off from the stack.

Types of System Calls

A set of (seemingly independent) groups of services:

- Process control and IPC (Inter-Process Communication)
- Memory management
  - allocating and freeing memory space on request
- Access to data in files
- File and file-system management
- Device management
- Communications
  - Networking and distributed computing support
- Other services
  - e.g., profiling
  - debugging
  - etc.

Process Control Calls (1)

- fork() – create a new process
  `pid = fork();`
  - The *fork*() function shall create a new process. The new process (child process) shall be an exact copy of the calling process (parent process) except some process' system properties
  - It returns 'twice'
    - return value == 0 ... child
    - return value > 0 ... parent (returned value is the child's *pid*)
- exit() – terminate a process
  `void exit(int status);`
  - The *exit*() function shall then flush all open files with unwritten buffered data and close all open files. Finally, the process shall be terminated and system resources owned by the process shall be freed
  - The value of 'status' shall be available to a waiting parent process
  - The *exit*() function should never return

Process Control Calls (2)

- 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 shall 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 shall replace the current process image with a new process image. The new image shall be constructed from a regular, executable file called the *new process image file*.
  - There shall be no return from a successful *exec*, because the calling process image is overlaid by the new process image; any return indicates a failure

Process Control Calls- Sample program

```
Int main()
 { pid_t pid1, pid2;
   int i=10;
   pid1= fork();
   if(pid1==0){
              i=i+5;
              printf("%d", i)
                   }
    i=i+20;
    Printf("%d", i);
    }
```


Memory Management Calls

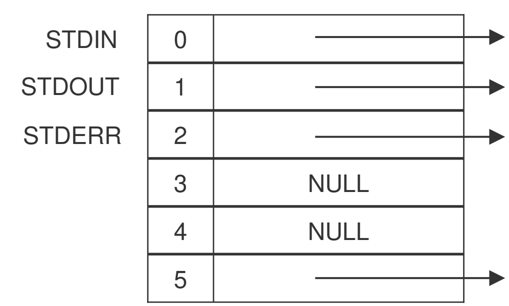
- System calls of this type are rather obsolete
  - Modern virtual memory mechanisms can allocate memory automatically as needed by applications
  - Important system API calls are:
- malloc() – a memory allocator

  `void *malloc(size_t size);`
  - The *malloc()* function shall allocate unused space for an object whose size in bytes is specified by `size` and whose value is unspecified.
  - It returns a pointer to the allocated memory space
- free() – free a previously allocated memory

  `void free(void *ptr);`
  - The *free()* function shall cause the space pointed to by `ptr` to be deallocated; that is, made available for further allocation.
  - If the argument does not match a pointer earlier returned by a *malloc*() call, or if the space has been deallocated by a call to *free()*, the behavior is undefined.

File Access Calls (1)

- POSIX-based operating systems treat a *file* in a very general sense
  - ***File*** is an object that can be written to, or read from, or both. A file has certain attributes, including access permissions and type.
  - File types include
    - regular file,
    - character special file ... a 'byte oriented device',
    - block special file ... a 'block oriented device',
    - FIFO special file,
    - symbolic link,
    - socket, and
    - directory.
  - To access any file, it must be first *open*ed using an *open()* call that returns a *file descriptor* (*fd*).
    - *fd* is a non-negative integer used for further reference to that particular file
    - In fact, *fd* is an index into a process-owned table of file descriptors
    - Any *open()* (or other calls returning *fd*) will always assign the LOWEST unused entry in the table of file descriptors



File Access Calls (2)

- open – open file
  `fd = open(const char *path, int oflag, ...);`
  - The *open()* function shall establish the connection between a file and a file descriptor. The file descriptor is used by other I/O functions to refer to that file. The `path` argument points to a pathname naming the file.
  - The parameter `oflag` specifies the open mode:
    - ReadOnly, WriteOnly, ReadWrite
    - Create, Append, Exclusive, ...
- close – close a file descriptor
  `err = close(int fd);`
  - The *close*() function shall deallocate the file descriptor indicated by *fd*. To deallocate means to make the file descriptor available for return by subsequent calls to *open*() or other functions that allocate file descriptors.
  - When all file descriptors associated with an open file description have been closed, the open file description shall be freed.

File Access Calls (3)

- **read** – read from a file

  `b_read = read(int fd, void *buf, int nbyte);`

  - The *read*() function shall attempt to read `nbyte` bytes from the file associated with the open file descriptor, `fd`, into the buffer pointed to by `buf`.
  - The return value shall be a non-negative integer indicating the number of bytes actually read.

- **write** – write to a file

  `b_written = write(int fd, void *buf, int nbyte);`

  - The *write*() function shall attempt to write `nbyte` bytes from the buffer pointed to by `buf` to the file associated with the open file descriptor `fd`.
  - The return value shall be a non-negative integer indicating the number of bytes actually written.

File Access Calls (4)

- lseek – move the read/write file offset
  `where = lseek(int fd, off_t offset, int whence);`
  - The *lseek*() function shall set the file offset for the open associated with the file descriptor `fd`, as follows:
    - If `whence` is `SEEK_SET`, the file offset shall be set to `offset` bytes.
    - If `whence` is `SEEK_CUR`, the file offset shall be set to its current location plus `offset`.
    - If `whence` is `SEEK_END`, the file offset shall be set to the size of the file plus `offset`.
  - The *lseek*() function shall allow the file offset to be set beyond the end of the existing data in the file creating a gap. Subsequent reads of data in the gap shall return bytes with the value 0 until some data is actually written into the gap (implements *sparse file*).
  - Upon successful completion, the resulting offset, as measured in bytes from the beginning of the file, shall be returned.
  - An interesting use is:

```
where = lseek(int fd, 0, SEEK_CUR);
```

    will deliver the “current position” in the file.

File Access Calls (5)

- dup – duplicate an open file descriptor
  `fd_new = dup(int fd);`
  - The *dup*() function shall duplicate the descriptor to the open fileassociated with the file descriptor `fd`.
  - As for *open()*, the LOWEST unused file descriptor should be returned.
  - Upon successful completion a non-negative integer, namely the file descriptor, shall be returned; otherwise, -1 shall be returned to indicate the error.
- stat – get file status
  `err = stat(const char path, struct stat *buf);`
  - The *stat*() function shall obtain information about the named file and write it to the area pointed to by the `buf` argument. The `path` argument points to a pathname naming a file. The file need not be open.
  - The `stat` structure contains a number of important items like:
    - device where the file is, file size, ownership, access rights, file time stapms, etc.

File Access Calls (6)

- chmod – change mode of a file
  `err = chmod(const char *path, mode_t mode);`
  - The *chmod*() function shall the file permission of the file named by the `path` argument to the in the `mode` argument. The application shall ensure that the effective privileges in order to do this.
- pipe – create an interprocess communication channel
  `err = pipe(int fd[2]);`
  - The *pipe*() function shall create a pipe and place two file descriptors, one each into the arguments `fd[0]` and `fd[1]`, that refer to the open file descriptors for the read and write ends of the pipe. Their integer values shall be the two lowest available at the time of the *pipe*() call.
  - A read on the file descriptor `fd[0]` shall access data written to the file descriptor `fd[1]` on a first-in-first-out basis.
  - The details and utilization of this call will be explained later.

File & Directory Management Calls (1)

- mkdir – make a directory relative to directory file descriptor
  `err = mkdir(const char *path, mode_t mode);`
  - The *mkdir*() function shall create a new directory with name `path`. The new directory access rights shall be initialized from `mode`.
- rmdir – remove a directory
  `err = rmdir(const char *path);`
  - The *rmdir*() function shall remove a directory whose name is given by `path`. The directory shall be removed only if it is an empty directory.
- chdir – change working directory
  `err = chdir(const char *path);`
  - The *chdir*() function shall cause the directory named by the pathname pointed to by the `path` argument to become the current working directory. Working directory is the starting point for path searches for *relative* pathnames.

File & Directory Management Calls (2)

- link – link one file to another file
  `err = int link(const char *path1, const char *path2);`
  - The *link*() function shall create a new link (directory entry) for the existing file identified by `path1`.
- unlink – remove a directory entry
  `err = unlink(const char *path);`
  - The *unlink*() function shall remove a link to a file.
  - When the file's link count becomes 0 and no process has the file open, the space occupied by the file shall be freed and the file shall no longer be accessible. If one or more processes have the file open when the last link is removed, the link shall be removed before *unlink*() returns, but the removal of the file contents shall be postponed until all references to the file are closed.
- chdir – change working directory
  `err = chdir(const char *path);`
  - The *chdir*() function shall cause the directory named by the pathname pointed to by the `path` argument to become the current working directory. Working directory is the starting point for path searches for *relative* pathnames.

Device Management Calls

- System calls to manage devices are hidden into ‘file calls’
  - POSIX-based operating systems do not make difference between traditional files and ‘devices’. Devices are treated as ‘special files’
  - Access to ‘devices’ is mediated by opening the ‘special file’ and accessing it through the device.
  - Special files are usually ‘referenced’ from the `/dev` directory.
- ioctl – control a device

  ```
  int ioctl(int fd, int request, ... /* arg */);
  ```

  - The *ioctl*() function shall perform a variety of control functions on devices. The `request` argument and an optional third argument (with varying type) shall be passed to and interpreted by the appropriate part of the associated with `fd`.

Other Calls

- kill – send a signal to a process or a group of processes
  `err = kill(pid_t pid, int sig);`
  - The *kill*() function shall send a signal to a process specified by `pid`. The signal to be sent is specified by `sig`.
  - *kill*() is an elementary inter-process communication means
  - The caller has to has to have sufficient privileges to send the signal to the target.
- signal – a signal management
  `void (*signal(int sig, void (*func)(int)))(int);`
  - The *signal*() function chooses one of three ways in which receipt of the signal `sig` is to be subsequently handled.
    - If the value of `func` is `SIG_DFL`, default handling for that signal shall occur.
    - If the value of `func` is `SIG_IGN`, the signal shall be ignored.
    - Otherwise, the application shall ensure that `func` points to a function to be called when that signal occurs. An invocation of such a function is called a "*signal handler*".

POSIX and Win32 Calls Comparison

- Only several important calls are shown

POSIX	Win32	Description
fork	CreateProcess	Create a new process
wait	WaitForSingleObject	The parent process may wait for the child to finish
execve	--	CreateProcess = fork + execve
exit	ExitProcess	Terminate process
open	CreateFile	Create a new file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from an open file
write	WriteFile	Write data into an open file
lseek	SetFilePointer	Move read/write offset in a file (file pointer)
stat	GetFileAttributesExt	Get information on a file
mkdir	CreateDirectory	Create a file directory
rmdir	RemoveDirectory	Remove a file directory
link	--	Win32 does not support “links” in the file system
unlink	DeleteFile	Delete an existing file
chdir	SetCurrentDirectory	Change working directory