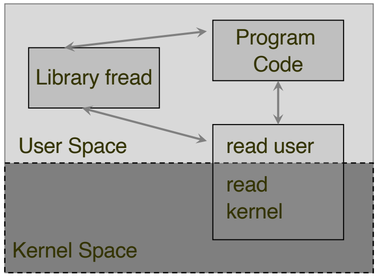
Operating Systems

Services and System Calls

1

System Calls

- Applications access OS services by making *system calls*
  - A function call that invokes the kernel
- This is the view of what the OS is and does from the application perspective



2

Operating System Services

- One set of functions of the OS provides services to the user:
  - User interface - Almost all operating systems have a user interface (UI)
    - Command-Line (CLI), Graphical User Interface (GUI), Batch
  - Program execution - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - I/O operations - A running program may require I/O, which may involve a file or an I/O device.
  - File-system manipulation - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

3

Operating System Services (Cont.)

- OS services for the user (con't):
  - Communications – Processes may exchange information, on the same computer or between computers over a network
    - Communications may be via shared memory or through message passing (packets moved by the OS)
  - Error detection – OS needs to be constantly aware of possible errors
    - May occur in the CPU and memory hardware, in I/O devices, in user program
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

4

Operating System Services (Cont.)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code.
  - **Accounting -** To keep track of which users use how much and what kinds of computer resources
  - **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
    - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

5

Operating System Interfaces

- Command Line Interface (CLI) – accepts commands from a user
  - The CLI could be part of the kernel
- Often implemented as a system program known as a shell. Examples:
  - The C Shell (csh) and its enhanced descendant tsch. The “t” comes from the OS Tenex, which included filename completion
  - The Bourne-again Shell (bash) is an enhanced descendant of the original UNIX shell (sh), named after its author Steven Bourne.
- Shells can invoke system calls or execute systems programs (which in turn interact with the kernel via system calls)

6

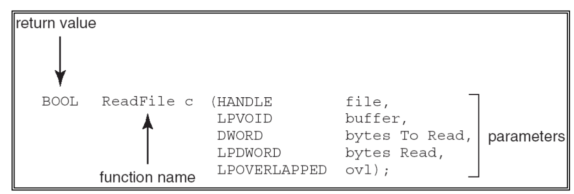
System Calls

- Programming interface to the services 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
  - For various reasons, but one key is that a “software interrupt” cannot be directly generated
- Three common APIs are Win32 API for Windows, POSIX API for POSIX-compliant systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

7

Example of Standard API

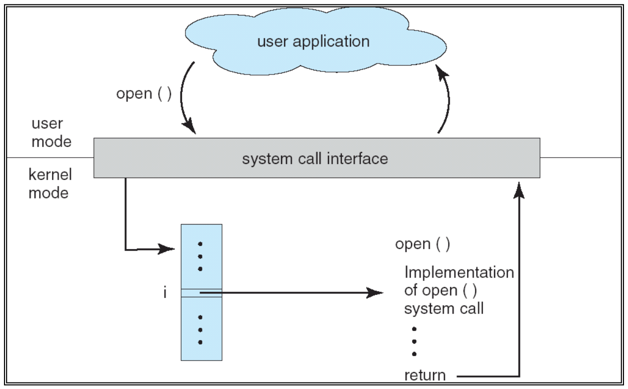
- Consider the ReadFile() function in the Win32 API—a function for reading from a file



- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read
  - LPOVERLAPPED ovl—indicates if overlapped I/O is being used

8

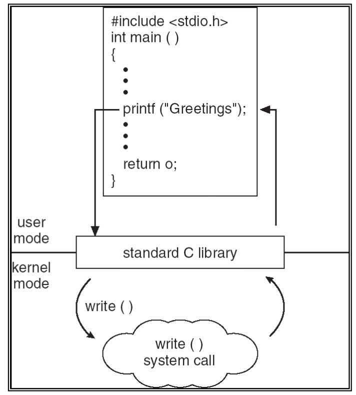
API – System Call – OS Relationship



9

Standard C Library Example

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



10

System Call Implementation

- Typically, a number is 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
  - 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)
- Why use APIs rather than system calls?

11

System call

- Directly calling a system call involves additional mechanism. Fragments from BSD:

```
#include <syscall.h>
       .globl _write, _errno
       # amtwritten =  write(fildes, address, count);

_write:                            # caller places arguments on stack
       lea   SYS_write,%eax        # select desired system call
       lcall $0x7,0                # call the system
       jb    1f                    # if system returns error, handle
       ret                         # otherwise return

1:     movl  %eax,_errno           # save error in global variable
       movl  $-1,%eax              # indicate error has occurred
       ret                         # and return
```

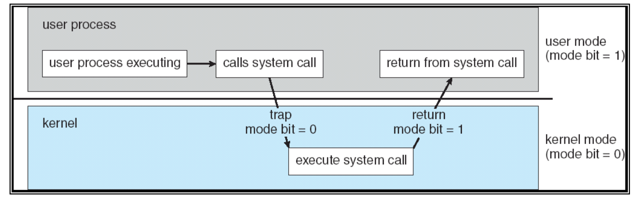
12

System call in Linux

```
#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name)); \
__syscall_return(type,__res); \
}
```

13

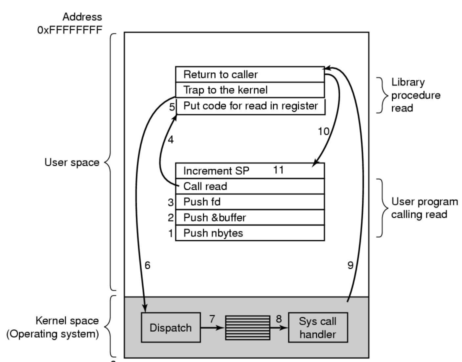
System Calls - User to Kernel Mode



- The Operating System gets control when a user process requests service via a system call
- These calls request service from the OS
  - or voluntarily yield the CPU
  - an interrupt causes involuntary yielding of the CPU

14

Steps in Making a System Call



There are 11 steps in making the system call
read (fd, buffer, nbytes)

15

System Calls: POSIX vs Win32

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

16

Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communications

17

Process Creation

Principal events that cause process creation

- System initialization
- Execution of a process creation system call
- User request to create a new process
- Initiation of a batch job

- In reality they all use the system call interface

18

Process Termination

Conditions which terminate processes

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)

19

OS Processes

- UNIX has a special process called init that starts other processes
  - terminal listeners
- Service processes that run in the background (**daemons**)
  - Some are started via scripts run by init
  - The “d” in sshd indicates a daemon (convention)
- The shell creates processes for the user
  - Or a graphical user interface can do so

20

Process Creation with the shell (UNIX)

- Most basic case
  - You type a command, and the shell runs the associated program
- fork()/exec*()
- One command string may launch multiple processes
  - Depending on the output of other processes

21

A UNIX Process

- Each UNIX process has a few features that are interesting to us
- A process usually has a few file descriptors
  - A non-negative integer
  - stdin, stdout, stderr
- A file descriptor is returned by **open()** (or **creat()**) and is an argument to other I/O calls
  - Like **read()** and **write()**

22

Default File Descriptors

- By convention, Unix processes associate certain file descriptors with roles
- 0 - STDIN_FILENO (or stdin)
- 1 - STDOUT_FILENO (or stdout)
- 2 - STDERR_FILENO
- Just convention (not a feature of the kernel) but many things would break if it weren't followed

23

I/O Redirection

- The shell has mechanisms to control the initial associations of these descriptors
- **<** -- attach stdin to a file
  - Process reading from stdin will read from the file
  - Can be anywhere in the input
- **>** -- attach stdout to a file
  - If it does not exist, it is created (with permission)
- **>>** -- attach stdout to a file and append all writes to end of the file
  - Just like **>** if the file doesn't exist

24

I/O Redirection and Pipes

- Many programs read from either a file specified as an argument or stdin
  - Again, only a convention
  - Thus "wc file" == "wc < file" == "cat file | wc"
- You can connect the stdout of one command to the stdin of another with the symbol **|**
  - Called a pipe

25

I/O Redirection

- You can send two file descriptors to one
  - In *sh 2>&1 will redirect stderr to stdout
  - command1 2>&1 | command2
  - In *csh, you can send both to a file with >& and to another process with |&
- cat < file | sort > output

26

UNIX Processes

- A C program in UNIX starts with a function called main()
  - int main(int argc, char *argv[])
- argc is the number of command line arguments
- argv is an array of pointers to the arguments
- argv[0] is the name of the program
- Or int main(int argc, char *argv[], char *envp[])
  - The environment is also available thru
    - extern char **environ;
    - so envp is often omitted
  - environ is a null-terminated array of pointers to environment variables of the form *variable=value*

27

Back to exec*()…

- Now we can discuss the variants of exec
  - All are front-ends for the same system call
- execl(), execlp() and execle() take a list of arguments
  - man has … meaning a variable number of arguments
- execv() takes a null-terminated *argv[]
  - Note this must be null-terminated, even though argv[argc] might not be null (K&R doesn't specify this)
- execvp() searches the path while execv() does not (like execlp())
- execve is used by all
  - Takes filename, argv, and envp

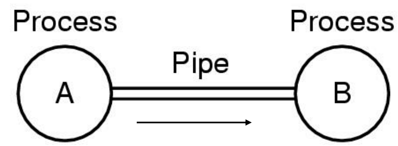
28

Process Hierarchies

- Parent creates a child process, child processes can create its own process
- Forms a hierarchy
  - UNIX calls this a "process group"
- Windows has no concept of process hierarchy
  - all processes are created equal
  - The creating process is returned a *handle* but can pass it on

29

IPC with a pipe



man -s 2 pipe or man 2 pipe

int
pipe(int *fildes);

The **pipe**() function creates a pipe, which is an object allowing unidirectional data flow, and allocates a pair of file descriptors.

fildes[1] is the write end, fildes[0] is the read end

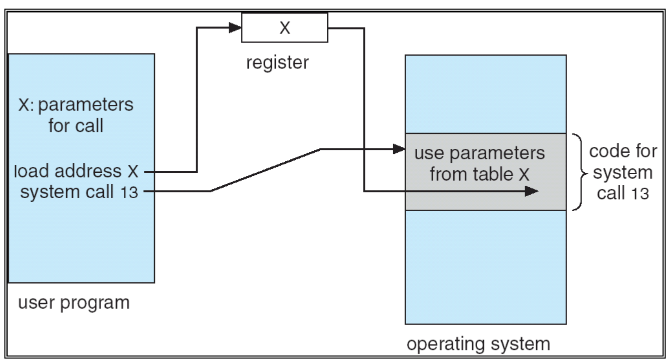
30

System Call Parameter Passing

- More information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in *registers*
    - In some cases, there may be more parameters than registers
  - Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

31

Parameter Passing via Table



32

Basic file I/O

- Processes keep a list of open files
- Files can be opened for reading, writing
- Each file is referenced by a *file descriptor* (integer)
- Three files are opened automatically
  - FD 0: standard input
  - FD 1: standard output
  - FD 2: standard error

33

File I/O system call: `open()`

- `fd = open(path, flags, mode)`
- path: string, absolute or relative path
- flags:
  - O_RDONLY - open for reading
  - O_WRONLY - open for writing
  - O_RDWR - open for reading and writing
  - O_CREAT - create the file if it doesn't exist
  - O_TRUNC - truncate the file if it exists
  - O_APPEND - only write at the end of the file
- mode: specify permissions if using O_CREAT

34

File I/O system call: `close()`

- `retval = close(fd)`
- Close an open file descriptor
- Returns 0 on success, -1 on error

35

File I/O system call: `read()`

- `bytes_read = read(fd, buffer, count)`
- Read up to count bytes from file and place into buffer
- fd: file descriptor
- buffer: pointer to array
- count: number of bytes to read
- Returns number of bytes read or -1 if error

36

File I/O system call: `write()`

- `bytes_written = write(fd, buffer, count)`
- Write count bytes from buffer to a file
- fd: file descriptor
- buffer: pointer to array
- count: number of bytes to write
- Returns number of bytes written or -1 if error

37

System call: `lseek()`

- `retval = lseek(fd, offset, whence)`
- Move file pointer to new location
- fd: file descriptor
- offset: number of bytes
- whence:
  - SEEK_SET - offset from beginning of file
  - SEEK_CUR - offset from current offset location
  - SEEK_END - offset from end of file
- Returns offset from beginning of file or -1

38

UNIX File access primitives

- open – open for reading, or writing or create an empty file
- creat - create an empty file
- close – close a file
- read - get info from file
- write - put info in file
- lseek - move to specific byte in file
- unlink - remove a file
- remove - remove a file
- fcntl - control attributes assoc. w/ file

39

File I/O using FILEs

- Most UNIX programs use higher-level I/O functions
  - `fopen()`
  - `fclose()`
  - `fread()`
  - `fwrite()`
  - `fseek()`
- These use the FILE datatype instead of file descriptors
- Need to include `stdio.h`

40

Using datatypes with file I/O

- The functions discussed so far use raw bytes for file I/O, but data is often stored as specific data types (int, char, float, etc.)
- `fprintf(), fputs(), fputc()` - used to write data to a file
- `fscanf(), fgets(), fgetc()` - used to read data from a file

41