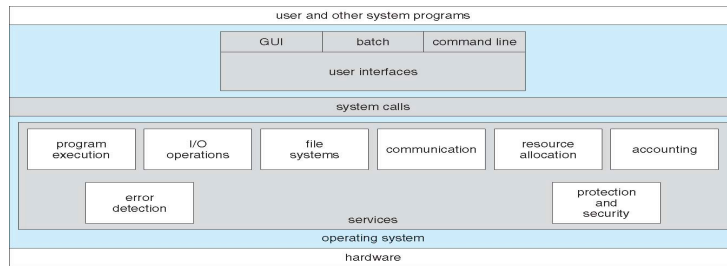


Operating System Services



- **User services:**
 - **User interface** - Almost all operating systems have a user interface (UI)
 - **Command-Line (CLI), Graphics User Interface (GUI), Batch**
 - **Program execution** - Load a program into memory, run that program, end execution (either normally or abnormally)
 - **I/O operations** - Involving a file or an I/O device
 - **File-system manipulation** - Read, write, create, delete files and directories, list file information, permission management.

Operating System Services (cont'd)

- **More user services:**
 - **Communications** – Inter-process, locally or over a network
 - Via shared memory or message passing
 - **Error detection**
 - In CPU, memory, I/O devices, programs
 - Take the appropriate action to ensure correct and consistent computing
 - Debugging facilities also useful
- **Efficient operation of the system itself**
 - **Resource allocation** - multiple users or multiple jobs run concurrently
 - Some resources (CPU time, memory, file storage) have special allocation code, others (I/O devices) have general request and release code
 - **Accounting**
 - **Protection and security** - The owners of information controls the use; concurrent processes should not interfere with each other
 - Ensures that all access to system resources is controlled
 - User authentication, defend external I/O devices from invalid access attempts
 - A chain is only as strong as its weakest link.

System Calls

- Programming interface to the services provided by the OS, forms an **Application Binary Interface (ABI)**
- Typically written in a C-like language (C, C++, Objective-C)
- Accessed by programmers via an **Application Programming Interface (API)**
- Most common APIs: Win32 (Windows), POSIX (virtually all versions of UNIX, Linux, and Mac OS X), Java (Java VM)
- In Unix, the description of all the API calls are in **Section 2 of the manual pages**
 - Some other, useful functions found in **Section 3** (Standard C library but non-OS-related)
- **Example of API: file manipulation**

<code>open</code>	Prepares the file for operation; returns a file descriptor
<code>close</code>	Terminates the use of a previously opened file/device
<code>read</code>	Obtains data from file/device
<code>write</code>	Writes data to file/device
<code>lseek</code>	Moves to some position in file/device (not applicable to everything)

File I/O (cont'd)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>

char result[256];
int file = open("echo", O_RDONLY);
if (file == -1)
    { perror("test.c"); return 1;}
read(file, result, 255);
close(file);

int main (int argc, char** argv) {
    int file = open(argv[1], O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
    int i = 0; char prefix[12]; char message[256] = "something";
    if (file == -1) return 1;
    while (true) {
        i++;
        printf("Message: ");
        fgets(message, 255, stdin);
        if (message[strlen(message)-1] == '\n')
            message[strlen(message)-1] = '\0';
        if (strlen(message) == 0) break;
        sprintf(prefix, 12, "%d: ", i);
        write(file, prefix, strlen(prefix));
        write(file, message, strlen(message));
        write(file, "\n", 1);
    }
    close(file);
}
```

File I/O (cont'd)

```

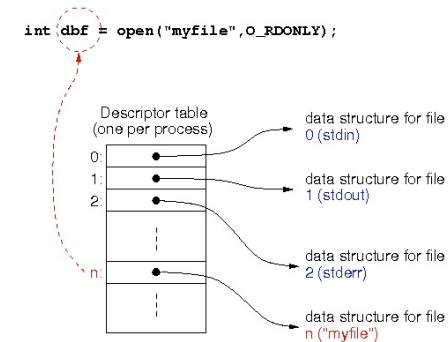
const int recv_nodata = -2;

int readline(const int fd, char* buf, const size_t max) {
    size_t i;
    int begin = 1;

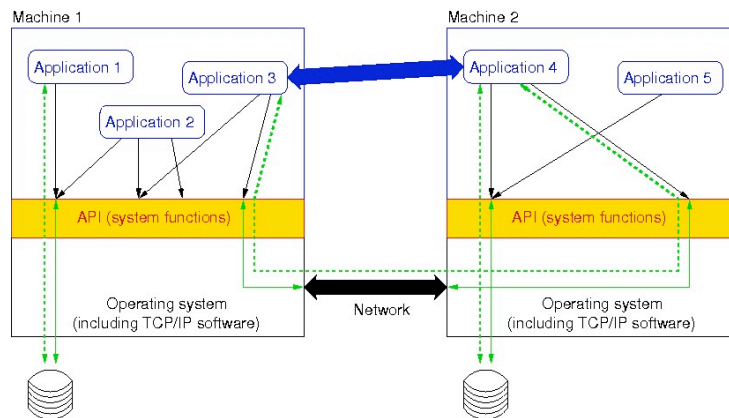
    for (i = 0; i < max; i++) {
        char tmp;
        int what = read(fd, &tmp, 1);
        if (what == -1) return -1;
        if (begin) {
            if (what == 0)
                return recv_nodata;
            begin = 0;
        }
        if (what == 0 || tmp == '\n') {
            buf[i] = '\0';
            return i;
        }
        buf[i] = tmp;
    }
    buf[i] = '\0';
    return i;
}

```

File I/O (cont'd)

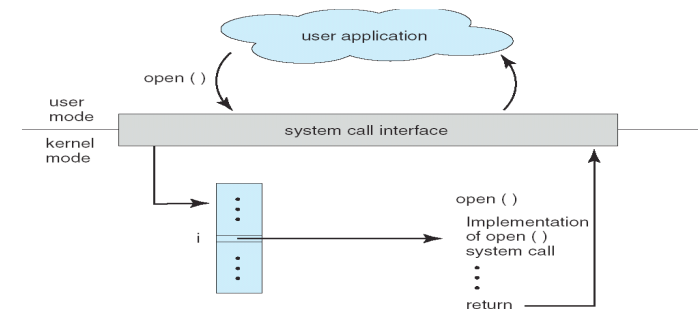


System calls (cont'd)



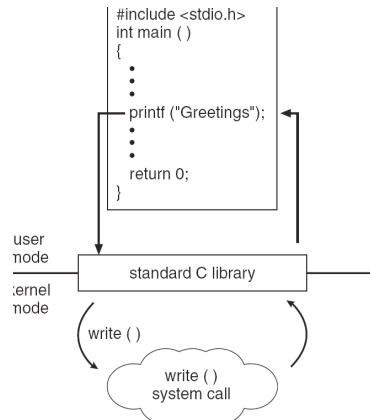
System Call Implementation

- 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 kernel and returns status and any return values upon completion
- The caller need know nothing about how the system call is implemented
 - Details of OS interface hidden from programmer, managed by run-time libraries



Other Functions in the Standard C Library

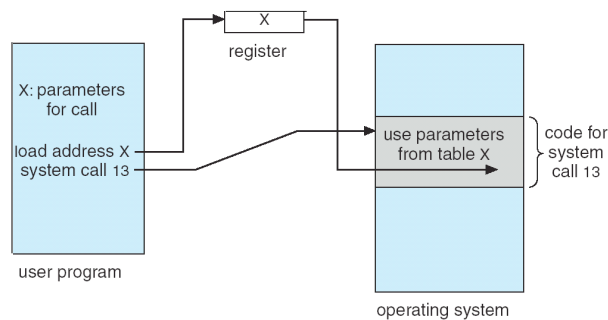
- C program calls `printf()` (**library call**)
- `printf()` in turn calls `write()` (**system call**)



System Call Parameter Passing

- Often, 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**
 - * What if we have more parameters than registers??
 - Parameters stored in a **block**, or **table** in memory, and address of block passed as a parameter in a register
 - * Approach taken by Linux and Solaris
 - Parameters **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 size of parameters

System Call Parameter Passing (cont'd)



Types of System Calls

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

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() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

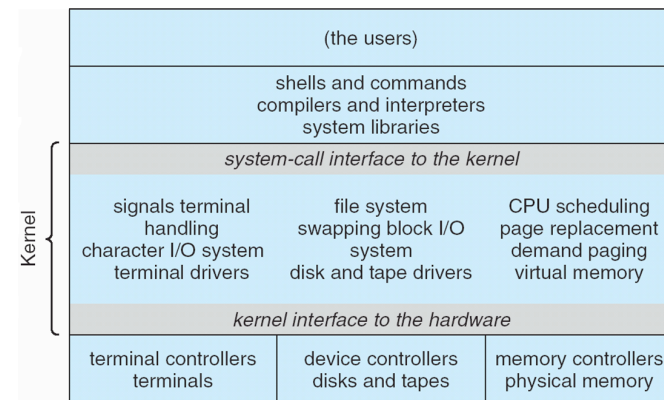
More wrappers: System Programs

- Provide a convenient environment for program development and execution
 - File manipulation
 - Status information
 - File modification
 - Programming language support
 - Program loading and execution
 - Communications
 - Application programs
- Most users' view of the operation system is defined by system programs, not the actual system calls
- Some are simply user interfaces to system calls; others are considerably more complex
- Example: File management: ls, cp, mv, rm, cat, more, ...
- Example: Status information
 - Some ask the system for info - date, time, amount of available memory, ...
 - Others provide detailed performance, logging, and debugging information

Design and Implementation

- Many intractable problems, OSes are built based on good approximations
- **Policy** (what will be done) vs. **Mechanism** (how to do it)
 - The separation of policy from mechanism allows maximum flexibility if policy decisions are to be changed later
- Layered approach
 - number of layers (levels), each built on top of lower layers; the bottom layer (layer 0), is the hardware; the highest is the user interface.
 - layers selected such that each uses only services of lower-level layers
- The original UNIX had limited structuring. It consists of two separable parts
 - Systems programs
 - The kernel (**monolithic kernel**)
 - * Everything below the system-call interface and above the physical hardware
 - * Provides file system, CPU scheduling, memory management, etc - a large number of functions for one level

Traditional UNIX System Structure



Design and Implementation (cont'd)

- **Microkernel approaches:**
 - Move as much from the kernel into “user” space
 - Communication takes place between user modules using message passing
 - Easier to extend, easier to port to new architectures
 - More reliable (less code is running in kernel mode), more secure
 - **But** performance overhead of user space to kernel space communication
- Most modern operating systems implement **kernel modules**
 - Object-oriented approach
 - Each core component is separate
 - Each talks to the others over known interfaces
 - Each is loadable as needed within the kernel
 - More flexible than layers
- Layers to the extreme: **virtual machines**

Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
- OSes generate **log files** containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- Operating system failure can generate **crash dump** file containing kernel memory
- Beyond crashes, performance tuning can optimize system performance
- Kernighan’s Law: “Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”
- DTrace tool in Solaris, FreeBSD, Mac OS X allows live instrumentation on production systems
 - **Probes** fire when code is executed, capturing state data and sending it to consumers of those probes

Solaris 10 dtrace Following System Call

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
0 -> XEventsQueued U
0 -> _XEventsQueued U
0 -> _XllTransBytesReadable U
0 <- _XllTransBytesReadable U
0 -> _XllTransSocketBytesReadable U
0 <- _XllTransSocketBytesreadable U
0 -> ioctl U
0 -> ioctl K
0 -> getf K
0 -> set_active_fd K
0 <- set_active_fd K
0 <- getf K
0 -> get_udatamodel K
0 <- get_udatamodel K
...
0 -> releasef K
0 -> clear_active_fd K
0 <- clear_active_fd K
0 -> cv_broadcast K
0 <- cv_broadcast K
0 <- releasef K
0 <- ioctl K
0 <- ioctl U
0 <- _XEventsQueued U
0 <- XEventsQueued U
```