BUCKNELL UNIVERSITY
Computer Science

CSCI 315 Operating Systems Design

Operating Systems Structures

Notice: This set of slides is based on the notes by Professor Perrone of Bucknell and the textbook authors Silberschatz, Galvin, and Gagne

Operating System Services

- One set of services for users
- The other set of services for system operations

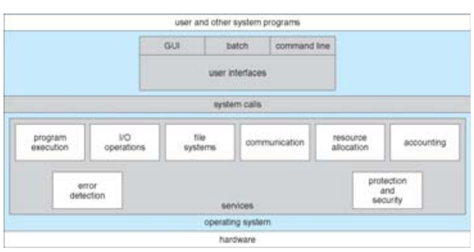
User Services

- One set for the users:
  - **User interface** - Almost all operating systems have a user interface (**UI**).
    - Varies between **Command-Line** (**CLI**), **Graphics User Interface** (**GUI**), **Batch**
  - **Program execution** - The system must be able to load a program into memory and to run that program
  - **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. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
  - **Communications** – Processes may exchange information, on the same computer or between computers over a network
  - **Error detection and handling** – Deal with errors

System Operation Services

- 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
  - **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

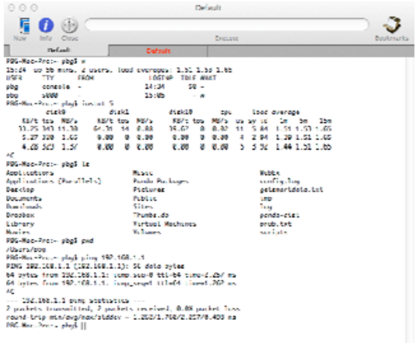
A View of Operating System Services



User Operating System Interface - CLI

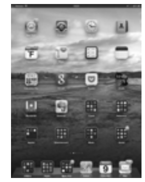
- CLI or **command line interpreter** allows direct command entry
  - User types in a command as text
  - The CLI (a.k.a. shell) takes the command and sends it to the operating system kernel for proper action and display the result of the action to the user
- It is *interactive.*

Bourne Shell Command Interpreter



Other Types of User Interfaces

- Graphics User Interface (GUI)
  - Touchscreen Interface



System Calls

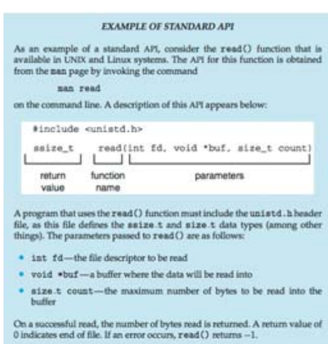
- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Remember *syscall* in MIPS?
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use

Example of System Calls

MIPS system call:

```
li  $v0, 1                  # service 1 is print integer
add $a0, $t0, $zero  # load value to register $a0
syscall
```

Example of System Call API



EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

return value | function name | parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

Example of Use System Call

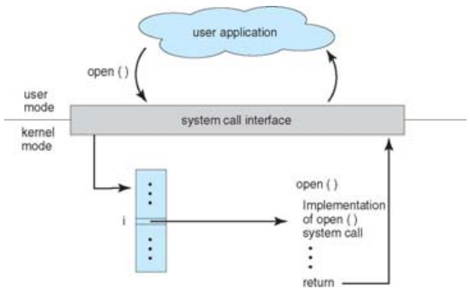
```
#include <unistd.h>

/* read a entire file*/
char * read_file(int fd) {

    char * content = malloc(MAXLEN + 1);
    ssize_t size_read;

    size_read = read(fd, content, MAXLEN)
    content[MAXLEN] = 0;
    return content;
}
```

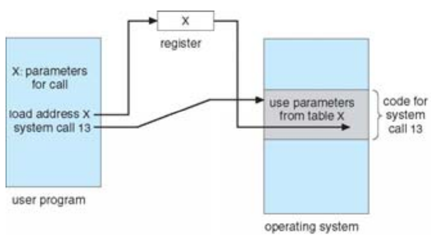
API – System Call – OS Relationship



System Call Parameter Passing

- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in 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

Parameter Passing via Table



Types of System Calls

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

Examples of Windows and Unix System Calls

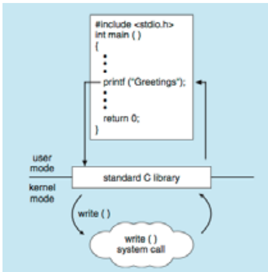
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information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

C Library Calls

- In addition to system calls, which map directly to the services provided by an OS, programming languages provide some libraries.
- C has a rich set of libraries, ranging from input/output to mathematics operations.

Standard C Library Example

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



System Programs

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

System Program Examples

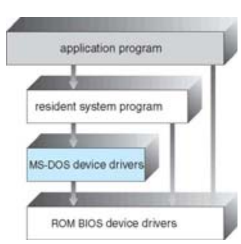
- Text editors such as vi and emacs
- Compilers and interpreters such as Java, Python, and C
- Assembler, loader, linker
- Web browsers, web servers
- Command line interpreter (a.k.a. shells)

Operating System Structure

- General-purpose OS is very large program
- Various ways to structure one as follows

Simple Structure

- MS-DOS – written to provide the most functionality in the least space
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated

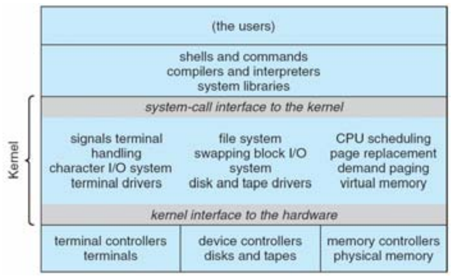


UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

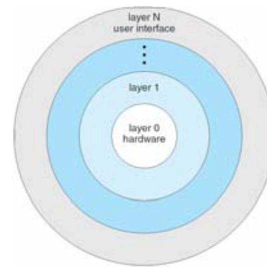
Traditional UNIX System Structure

Beyond simple but not fully layered



Layered Approach

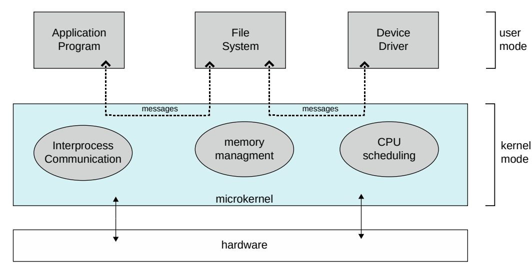
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers



Microkernel System Structure

- Moves as much from the kernel into user space
- **Mach** example of **microkernel**
  - Mac OS X kernel (**Darwin**) partly based on Mach
- Communication takes place between user modules using **message passing**
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

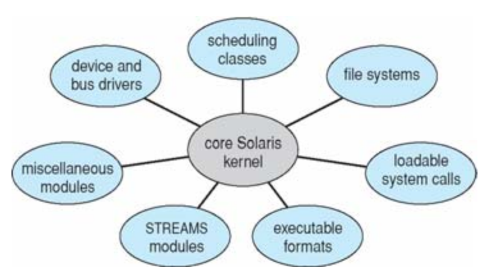
Microkernel System Structure



Modules

- Most modern operating systems implement **loadable kernel modules**
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible
  - Linux, Solaris, etc

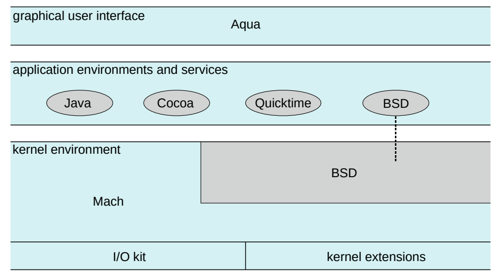
Solaris Modular Approach



Hybrid Systems

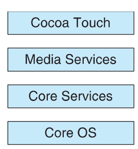
- Most modern operating systems actually not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, plus microkernel for different subsystem ***personalities***

Mac OS X Structure



iOS

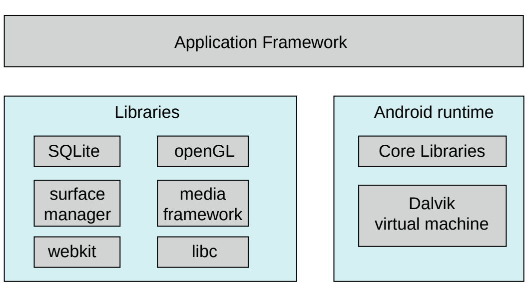
- Apple mobile OS for ***iPhone***, ***iPad***
  - Structured on Mac OS X, added functionality
  - Does not run OS X applications natively
    - Also runs on different CPU architecture (ARM vs. Intel)
  - **Cocoa Touch** Objective-C API for developing apps
  - **Media services** layer for graphics, audio, video
  - **Core services** provides cloud computing, databases
  - Core operating system, based on Mac OS X kernel



Android

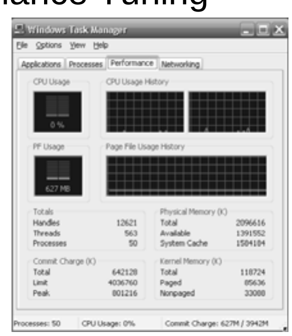
- Developed by Open Handset Alliance (mostly Google)
- Similar stack to IOS
- Based on Linux kernel but modified
  - Provides process, memory, device-driver management
  - Adds power management
- Runtime environment includes core set of libraries and Dalvik virtual machine
  - Apps developed in Java plus Android API
    - Java class files compiled to Java byte code then translated to executable that runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc

Android Architecture



Performance Tuning

- Improve performance by removing bottlenecks
- OS must provide means of computing and displaying measures of system behavior
- For example, “top” program or Windows Task Manager



DTrace

- DTrace tool in Solaris, FreeBSD, Mac OS X allows live instrumentation on production systems
- **Probes** fire when code is executed within a **provider**, capturing state data and sending it to **consumers** of those probes
- Example of following XEventsQueued system call move from libc library to kernel and back

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                          U
  0   -> _XEventsQueued                       U
  0     -> _X11TransBytesReadable             U
  0     <- _X11TransBytesReadable             U
  0     -> _X11TransSocketBytesReadable       U
  0     <- _X11TransSocketBytesreadable       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
```

DTrace

- DTrace code to record amount of time each process with UserID 101 is in running mode (on CPU) in nanoseconds

```
sched:::on-cpu
uid == 101
{
  self->ts = timestamp;
}

sched:::off-cpu
self->ts
{
  @time[execname] = sum(timestamp - self->ts);
  self->ts = 0;
}
```

```
# dtrace -s sched.d
dtrace: script 'sched.d' matched 6 probes
^C
    gnome-settings-d          142354
    gnome-vfs-daemon          158243
    dsdm                      189804
    wnck-applet               200030
    gnome-panel               277864
    clock-applet              374916
    mapping-daemon            385475
    xscreensaver              514177
    metacity                  539281
    Xorg                     2579646
    gnome-terminal           5007269
    mixer applet2            7388447
    java                    10769137
```

Figure 2.21 Output of the D code.

Operating System Generation

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site
- **SYSGEN** program obtains information concerning the specific configuration of the hardware system
  - Used to build system-specific compiled kernel or system-tuned
  - Can general more efficient code than one general kernel

System Boot

- When power initialized on system, execution starts at a fixed memory location
  - Firmware ROM used to hold initial boot code
- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader**, stored in **ROM** or **EEPROM** locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where **boot block** at fixed location loaded by ROM code, which loads bootstrap loader from disk
- Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then **running**