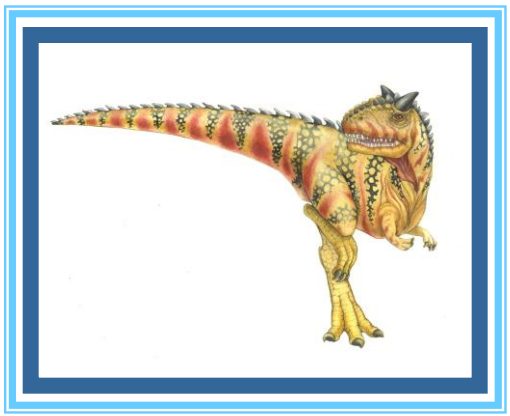
Chapter 2: Operating-System Services

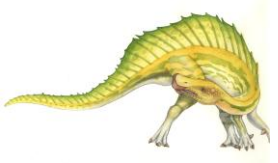


Silberschatz, Galvin and Gagne ©2018



Outline

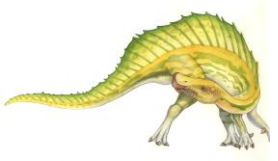
- Operating System Services
- User and Operating System-Interface
- System Calls
- System Services
- Linkers and Loaders
- Why Applications are Operating System Specific
- Design and Implementation
- Operating System Structure
- Building and Booting an Operating System
- Operating System Debugging





Objectives

- Identify services provided by an operating system
- Illustrate how system calls are used to provide operating system services
- Compare and contrast monolithic, layered, microkernel, modular, and hybrid strategies for designing operating systems
- Illustrate the process for booting an operating system
- Apply tools for monitoring operating system performance
- Design and implement kernel modules for interacting with a Linux kernel



Troy University Resources



Operating System Services

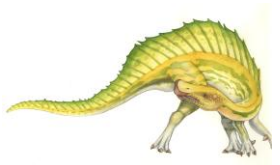
- Operating systems provide an environment for execution of programs and services to programs and users
- One set of operating-system services provides functions that are helpful to the user:
  - **User interface** - Almost all operating systems have a user interface (**UI**).
    - Varies between **Command-Line** (**CLI**), **Graphics User Interface** (**GUI**), **touch-screen**, **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. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.





Operating System Services (Cont.)

- One set of operating-system services provides functions that are helpful to the user (Cont.):
  - **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

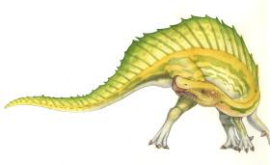


Troy University Resources



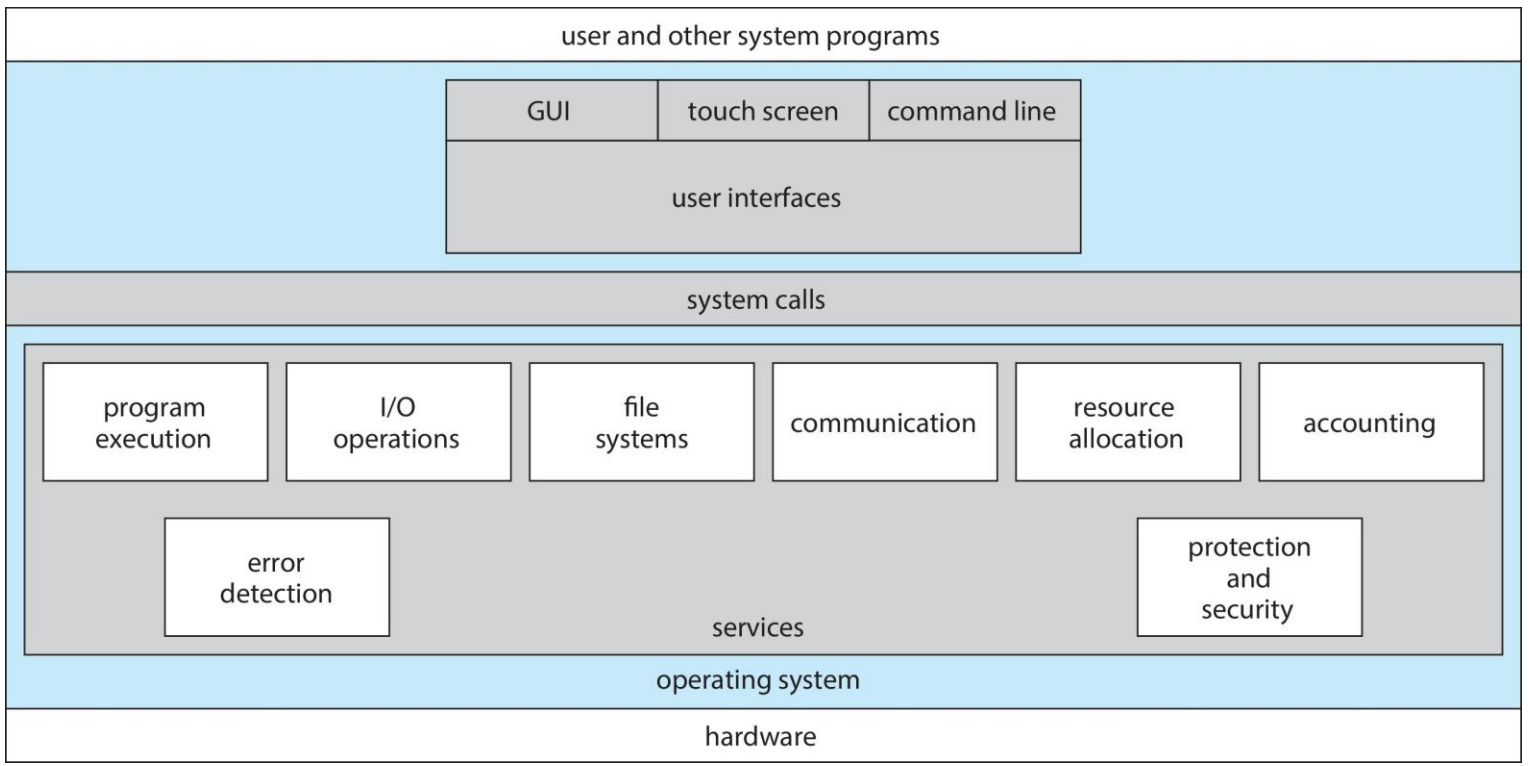
Operating System Services (Cont.)

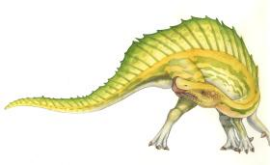
- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
  - **Logging -** 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





A View of Operating System Services



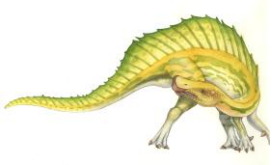


Troy University Resources



Command-Line Interface and Shell

- CLI means command-line interface: the user enters commands as text.
- A command interpreter parses and executes those commands. Unix command interpreters are commonly called shells.
- A shell usually runs as a user-space program. Historical systems also placed command interpreters in the kernel.
- A command may name a shell built-in, a function or an external executable.
- Adding an external executable normally needs no change to the shell. New shell syntax or built-in behavior may require a change.



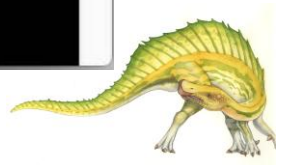


Bourne Shell Command Interpreter

1. root@r6181-d5-us01:~ (ssh)

root@r6181-d5-u... ⌘1 | ssh ⌘2 | root@r6181-d5-us01... ⌘3

```
Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbg$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
 06:57:48 up 16 days, 10:52,  3 users,  load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
                       50G   19G   28G  41% /
tmpfs                 127G  520K  127G   1% /dev/shm
/dev/sda1             477M   71M  381M  16% /boot
/dev/dssd0000         1.0T  480G  545G  47% /dssd_xfs
tcp://192.168.150.1:3334/orangefs
                       12T  5.7T  6.4T  47% /mnt/orangefs
/dev/gpfs-test         23T  1.1T   22T   5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root      97653 11.2  6.6 42665344 17520636 ?   S<Ll Jul13 166:23 /usr/lpp/mmfs/bin/mmfsd
root      69849  6.6  0.0        0        0 ?        S    Jul12 181:54 [vpthread-1-1]
root      69850  6.4  0.0        0        0 ?        S    Jul12 177:42 [vpthread-1-2]
root       3829  3.0  0.0        0        0 ?        S    Jun27 730:04 [rp_thread 7:0]
root       3826  3.0  0.0        0        0 ?        S    Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfsd
-r-x------ 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfsd
[root@r6181-d5-us01 ~]#
```

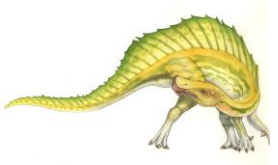


Troy University Resources



User Operating System Interface - GUI

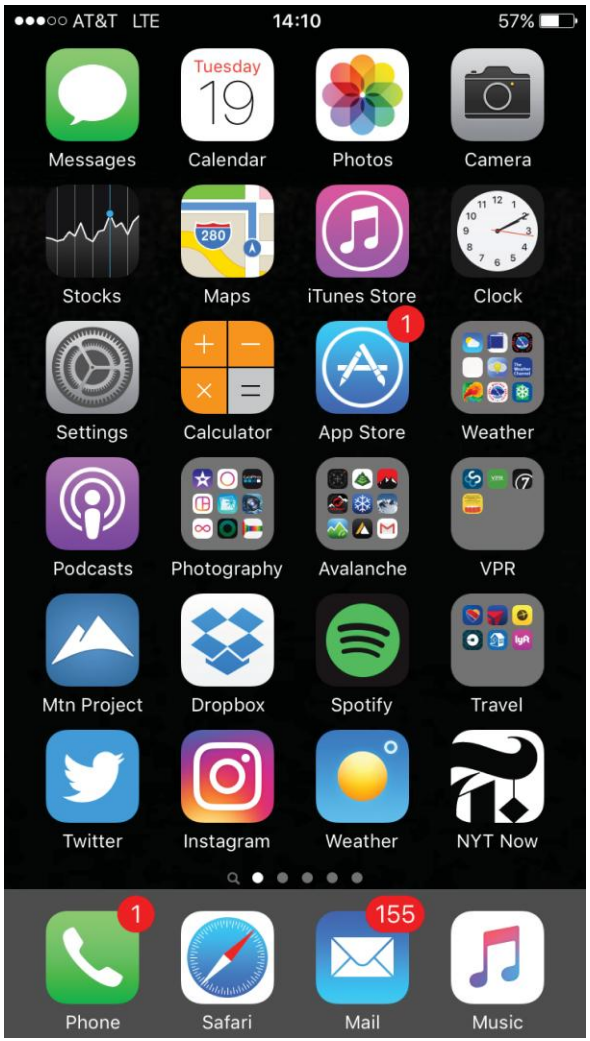
- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
  - Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI "command" shell
  - Apple Mac OS X is "Aqua" GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

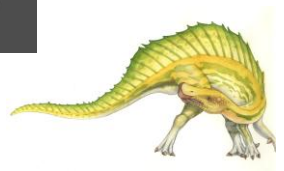




Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
- Voice commands

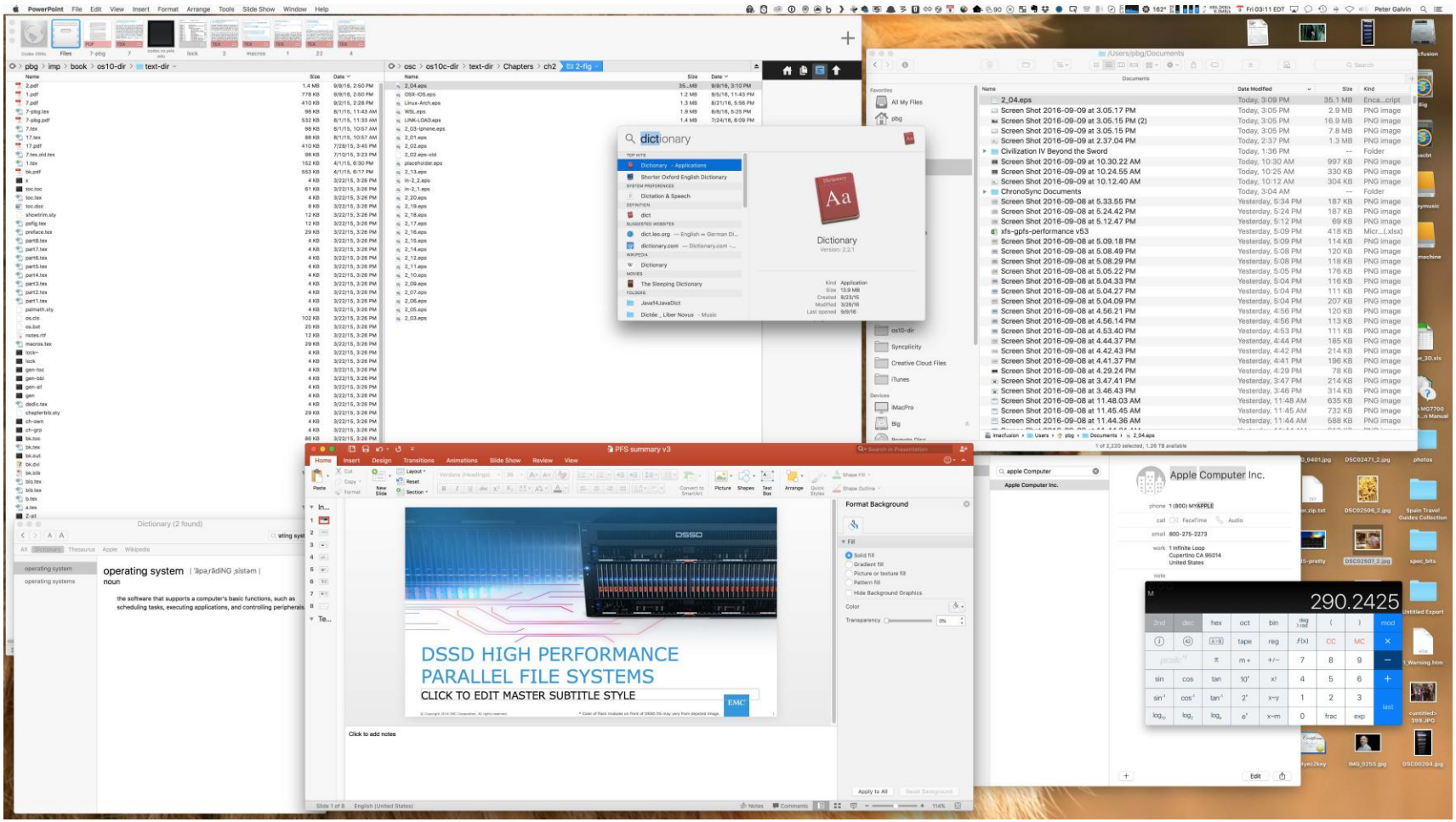


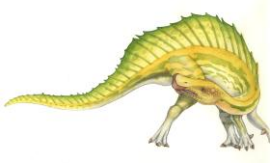


Troy University Resources



The Mac OS X GUI



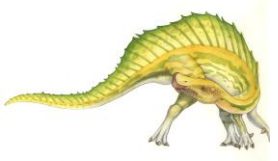




System Calls

- Programming interface to the services provided by the OS
- Applications normally invoke library functions such as C or C++ API wrappers
- Mostly accessed by programs via a high-level **Application Programming 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)

Note that the system-call names used throughout this text are generic

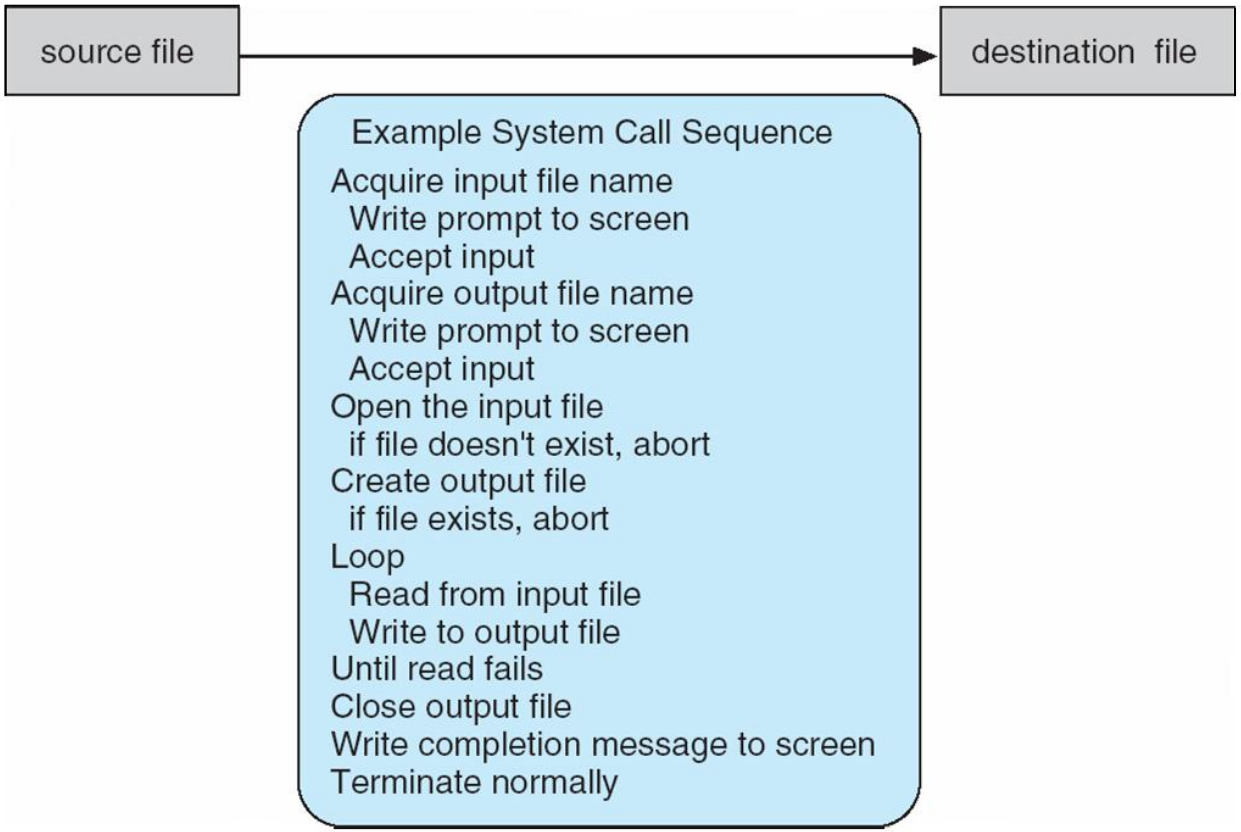


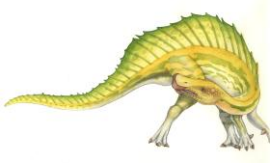
Troy University Resources



Example of System Calls

- System call sequence to copy the contents of one file to another file







Example of Standard 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 into which the data will be read
- `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.

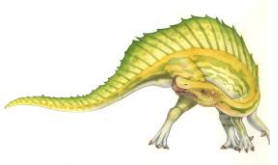


Troy University Resources



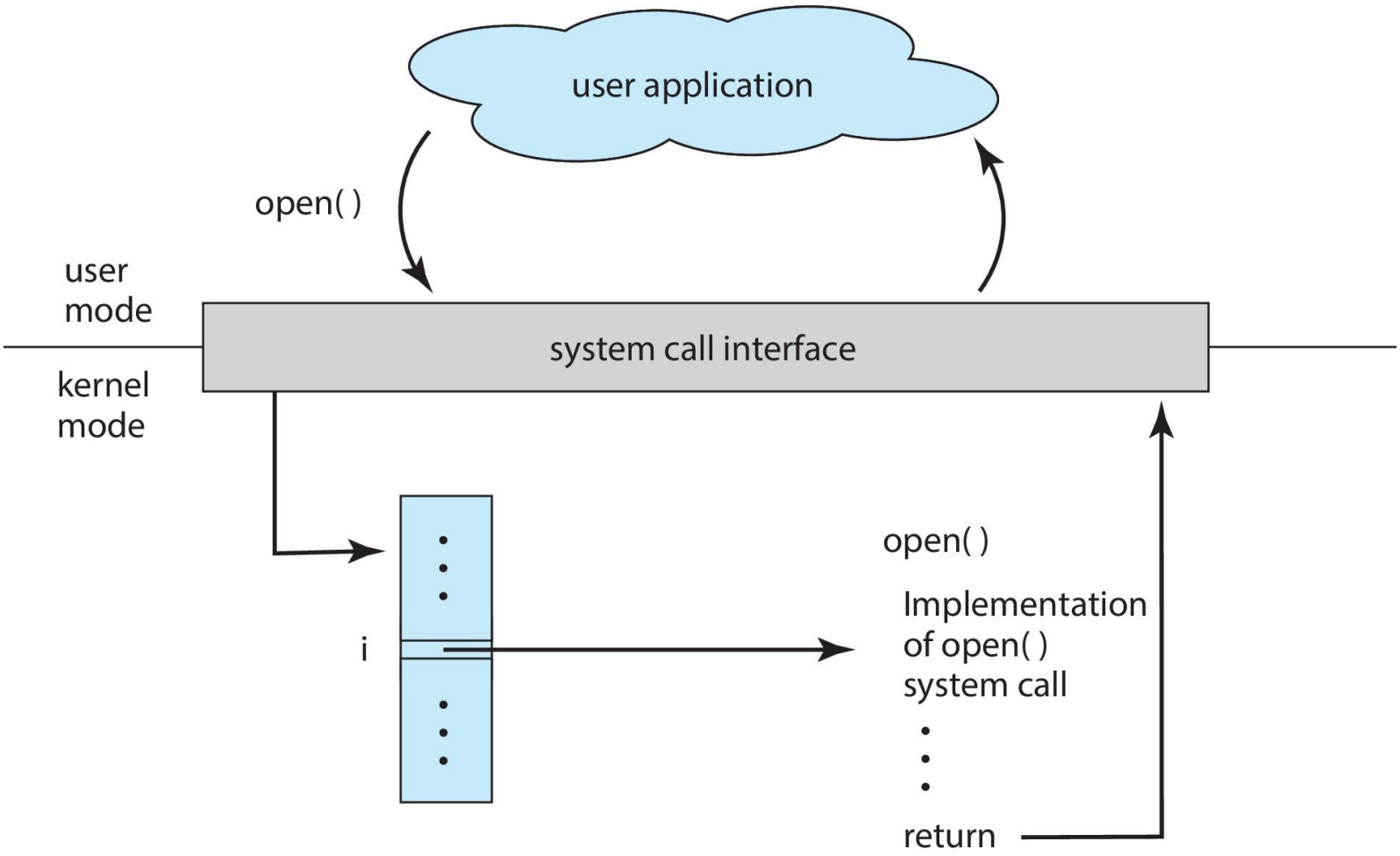
System Call Implementation

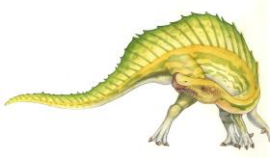
- The OS and CPU ABI define how a system-call number and arguments are supplied.
- The kernel dispatches the request to the appropriate service and returns a result or error.
- Applications usually call a library API wrapper that prepares this interface.
- One API call may use zero, one, or several system calls.
- Programs rely on the API contract rather than the kernel's internal implementation.
- POSIX read() returns ssize_t; its byte-count argument has type size_t.





API – System Call – OS Relationship



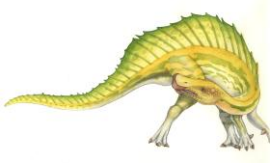


Troy University Resources



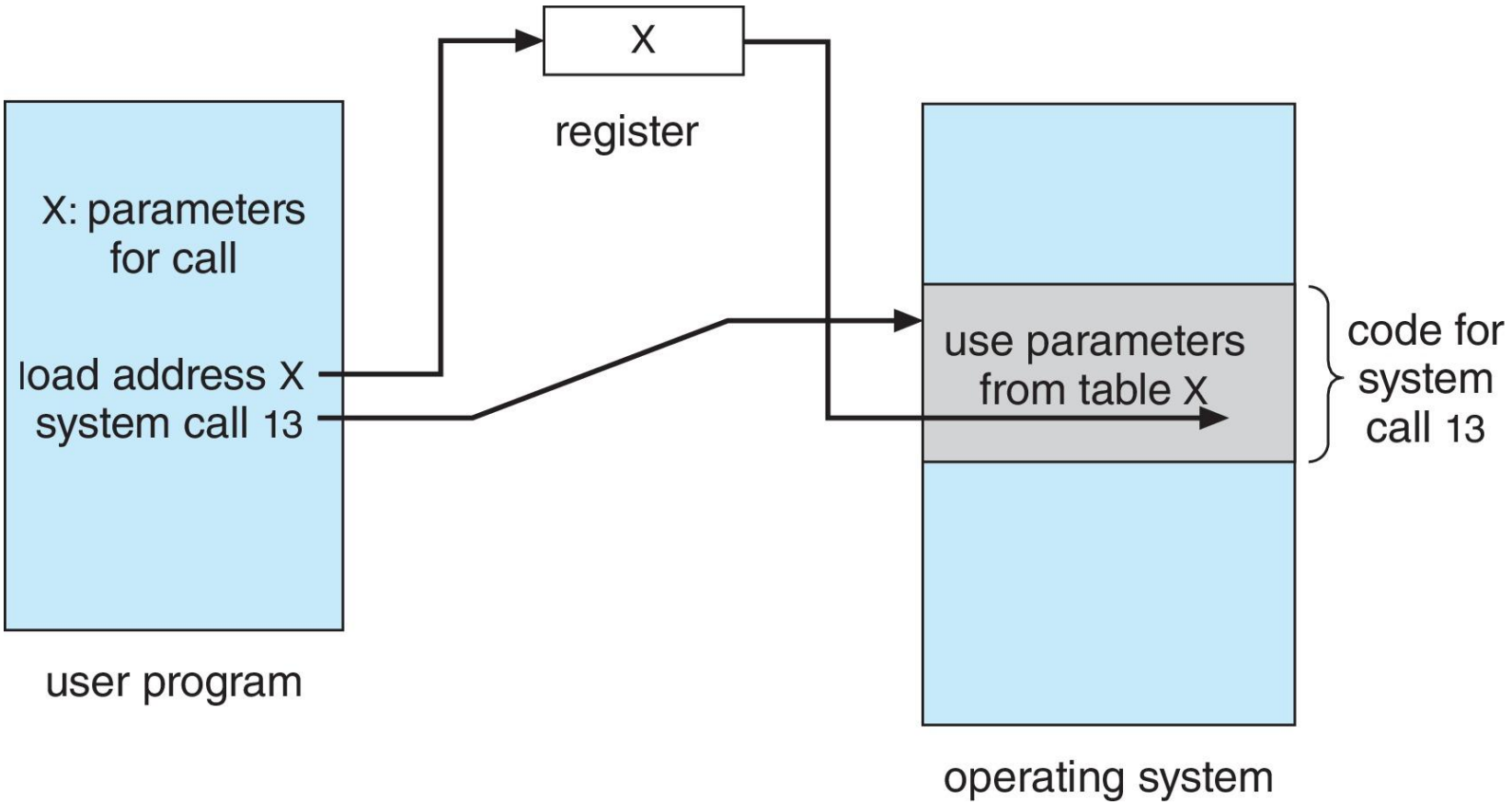
System Call Parameter Passing

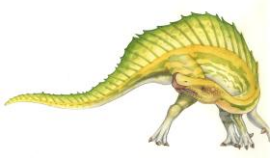
- The calling convention depends on the OS and CPU architecture.
- Registers commonly hold the system-call number and argument values.
- A register can hold a pointer to a structure or buffer in user memory.
- Some interfaces use stack arguments or a combination of registers and memory.
- For example, Linux x86-64 passes up to six syscall arguments in registers. A pointer argument can refer to a larger data structure.
- Memory blocks avoid the register-count limit for a structure, but memory and API limits still apply.





Parameter Passing via Table



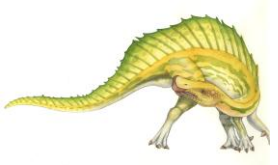


Troy University Resources



Types of System Calls

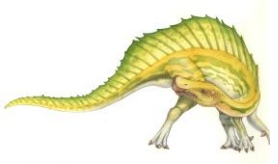
- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - **Debugger** for determining **bugs, single step** execution
  - **Locks** for managing access to shared data between processes





Types of System Calls (Cont.)

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

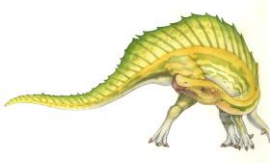


Troy University Resources



Types of System Calls (Cont.)

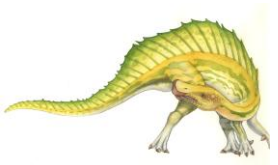
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages if **message passing model** to **host name** or **process name**
    - From **client** to **server**
  - **Shared-memory model** create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices





Types of System Calls (Cont.)

- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access



Troy University Resources



Examples of Windows and Unix System Calls

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix
Process control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File management	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device management	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	`write()`
Information maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communications	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shm_open()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`



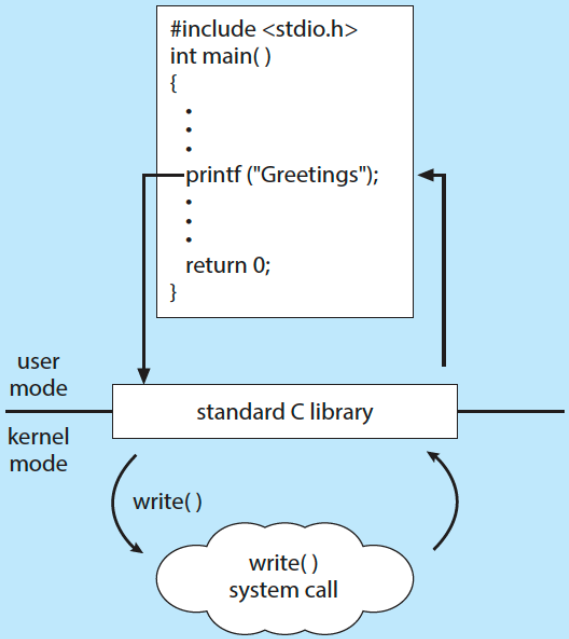


Standard C Library Example

- An example path: printf() formats buffered output; flushing may invoke write().

THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:



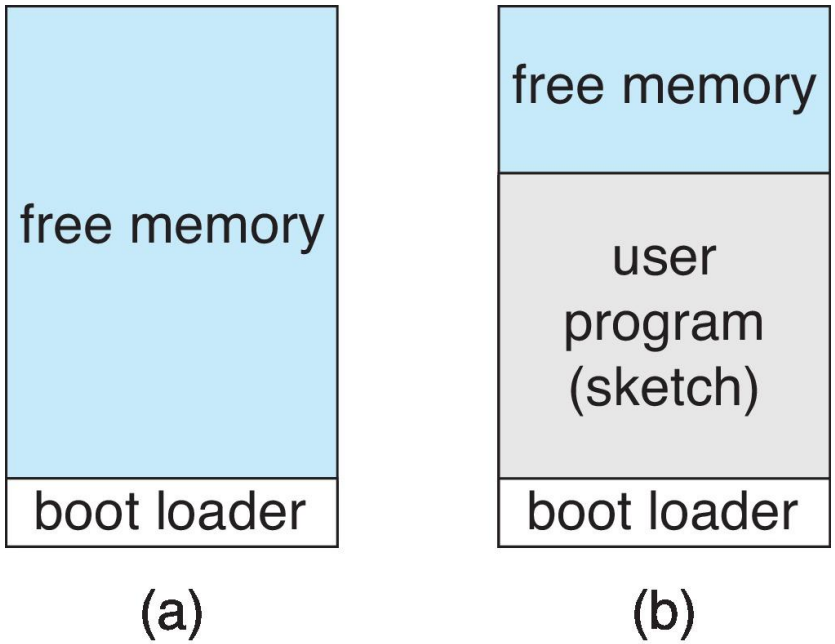


Troy University Resources

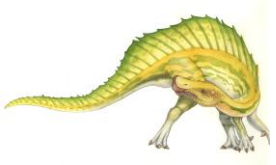


Example: Arduino

- Classic AVR Arduino example: one sketch runs without a general-purpose OS.
- The upload tool writes the compiled sketch into flash, commonly through a bootloader.
- After reset, startup code calls setup() once, then repeatedly calls loop().
- This example has no command shell to reload when a sketch finishes.
- The figure is conceptual. AVR program flash and SRAM are separate memory spaces.



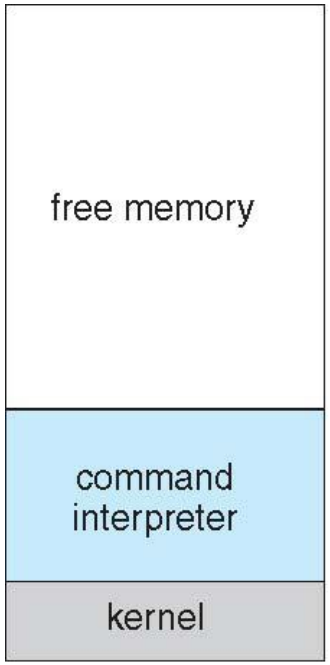
At startup (a), running a sketch (b)



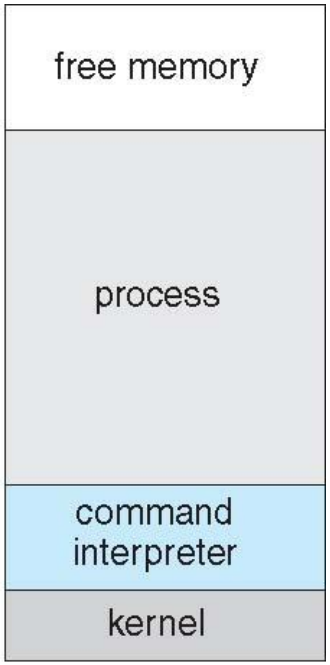


Historical Example: MS-DOS

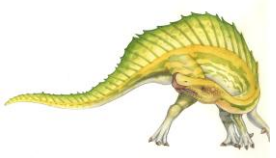
- MS-DOS normally runs one foreground application at a time.
- COMMAND.COM provides the command interpreter.
- Programs share the machine without modern per-process memory protection.
- An application may overwrite the transient part of COMMAND.COM.
- After the application returns, DOS reloads that part of the shell if needed.



(a)



(b)

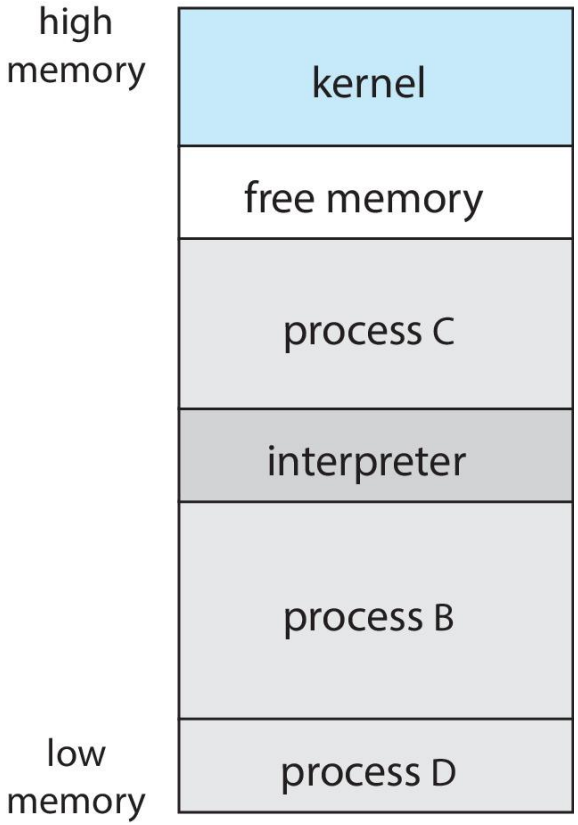


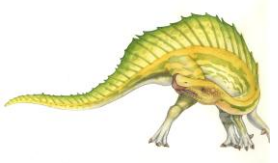
Troy University Resources



Example: FreeBSD

- Unix variant
- Multitasking
- User login -> invoke user's choice of shell
- Shell executes fork() system call to create process
  - Executes exec() to load program into process
  - Shell waits for process to terminate or continues with user commands
- By convention, a program reports:
  - an exit status of 0 for success
  - a nonzero exit status for failure

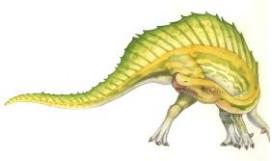






System Services

- System programs provide a convenient environment for program development and execution.
- File manipulation and file modification
- Status information and configuration tools
- Programming-language support
- Program loading and execution
- Communications and background services
- Users often interact with these programs rather than with system calls directly.

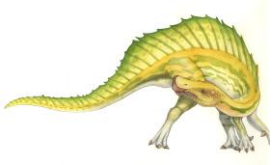


Troy University Resources



System Services (Cont.)

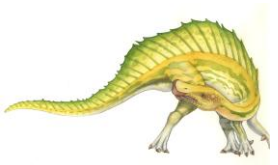
- Provide a convenient environment for program development and execution
  - Some of them are simply user interfaces to system calls; others are considerably more complex
- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output to the terminal or other output devices
  - Some systems implement a **registry** - used to store and retrieve configuration information





System Services (Cont.)

- **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

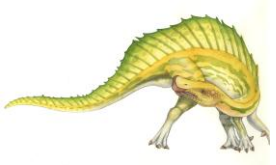


Troy University Resources



System Services (Cont.)

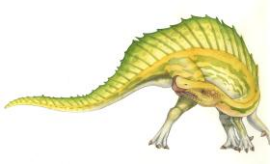
- **Background Services**
  - Launch at boot time
    - Some for system startup, then terminate
    - Some from system boot to shutdown
  - Provide facilities such as logging, printing, scheduled jobs and network services
  - Run in user context not kernel context
  - Known as **services**, **subsystems**, **daemons**
- **Application programs**
  - Don't pertain to system
  - Run by users
  - Not typically considered part of OS
  - Launched by command line, mouse click, finger poke





Linkers and Loaders

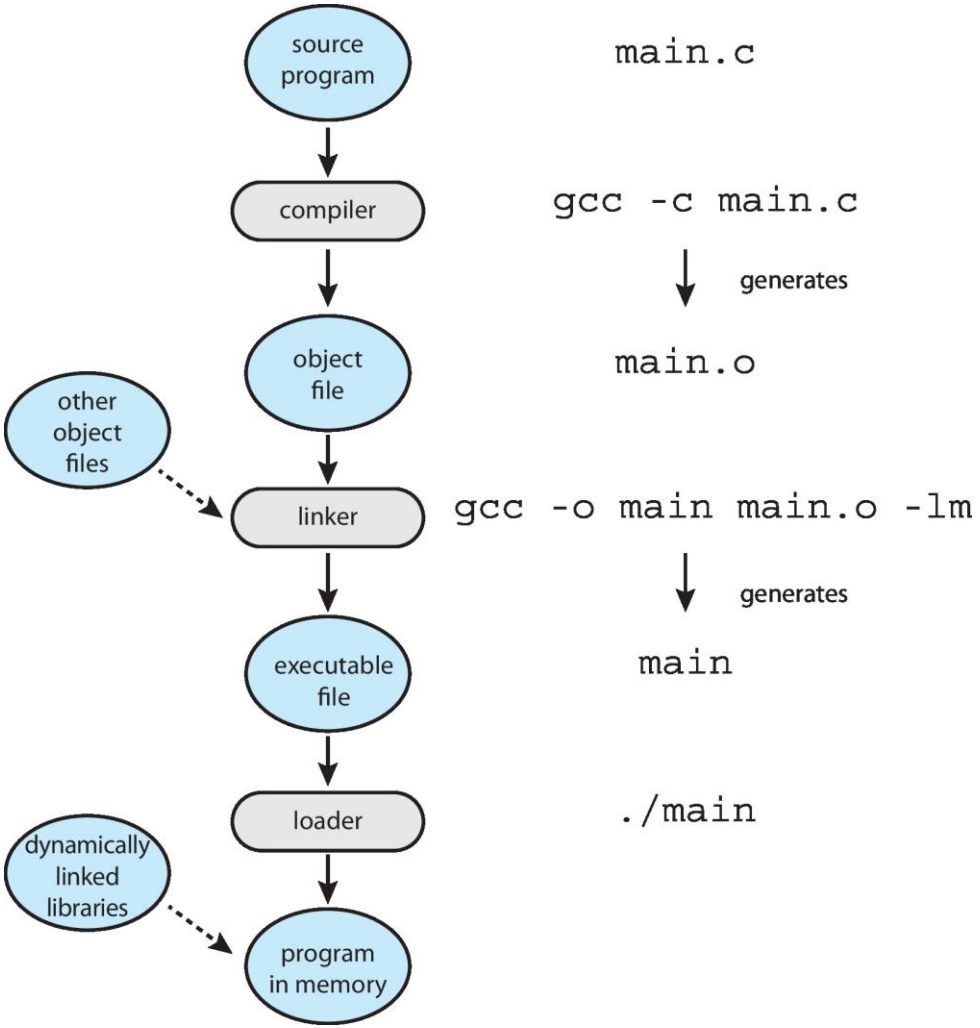
- The compiler and assembler produce relocatable object files.
- The linker resolves symbol references and combines object files into an executable or shared library.
- Static linking copies required library code into the executable.
- Dynamic linking resolves shared-library references at load time or during execution.
- The loader maps the executable and required libraries into the process address space.
- Relocation adjusts addresses or references as needed. Mapped code pages can be shared while each process retains its own writable state.



Troy University Resources



The Role of the Linker and Loader

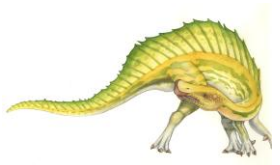






Why Applications are Operating System Specific

- Apps compiled on one system usually not executable on other operating systems
- Each operating system provides its own unique system calls
  - Own file formats, etc.
- Apps can be multi-operating system
  - Written in interpreted language like Python, Ruby, and interpreter available on multiple operating systems
  - App written in language that includes a VM containing the running app (like Java)
  - Use standard language (like C), compile separately on each operating system to run on each
- **Application Binary Interface** (**ABI**) is architecture equivalent of API, defines how different components of binary code can interface for a given operating system on a given architecture, CPU, etc.

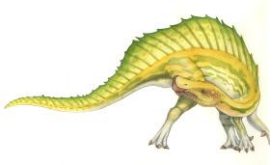


Troy University Resources



Design and Implementation

- Design and Implementation of OS is not “solvable”, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system
- **User** goals and **System** goals
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- Specifying and designing an OS is highly creative task of **software engineering**





Policy and Mechanism

- **Policy**: **What** needs to be done?
  - Example: choose a 10 ms time quantum
- **Mechanism**: **How** to do something?
  - Example: timer
- Important principle: separate policy from mechanism
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later.
  - Example: change the quantum from 10 ms to 20 ms

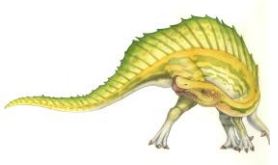


Troy University Resources



Implementation

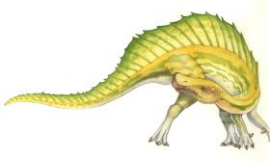
- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to **port** to other hardware
  - Performance depends on the implementation, compiler and workload
- **Emulation** can allow an OS to run on non-native hardware





Operating System Structure

- General-purpose OS is very large program
- Various ways to structure ones
  - Simple structure – MS-DOS
  - More complex – UNIX
  - Layered – an abstraction
  - Microkernel – Mach

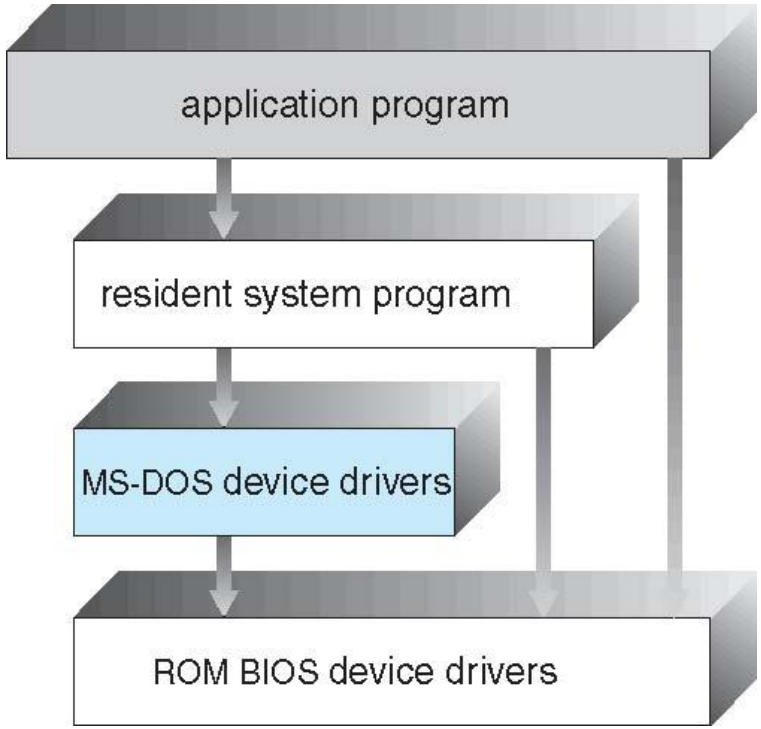


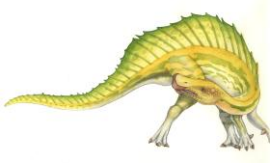
Troy University Resources



Historical Structure: MS-DOS

- MS-DOS aimed to provide substantial functionality in limited memory.
- Its interfaces and layers were only loosely separated.
- Applications could bypass higher-level services and access lower-level facilities.
- The diagram contrasts this weak separation with the structured designs that follow.







Monolithic Structure – Original 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

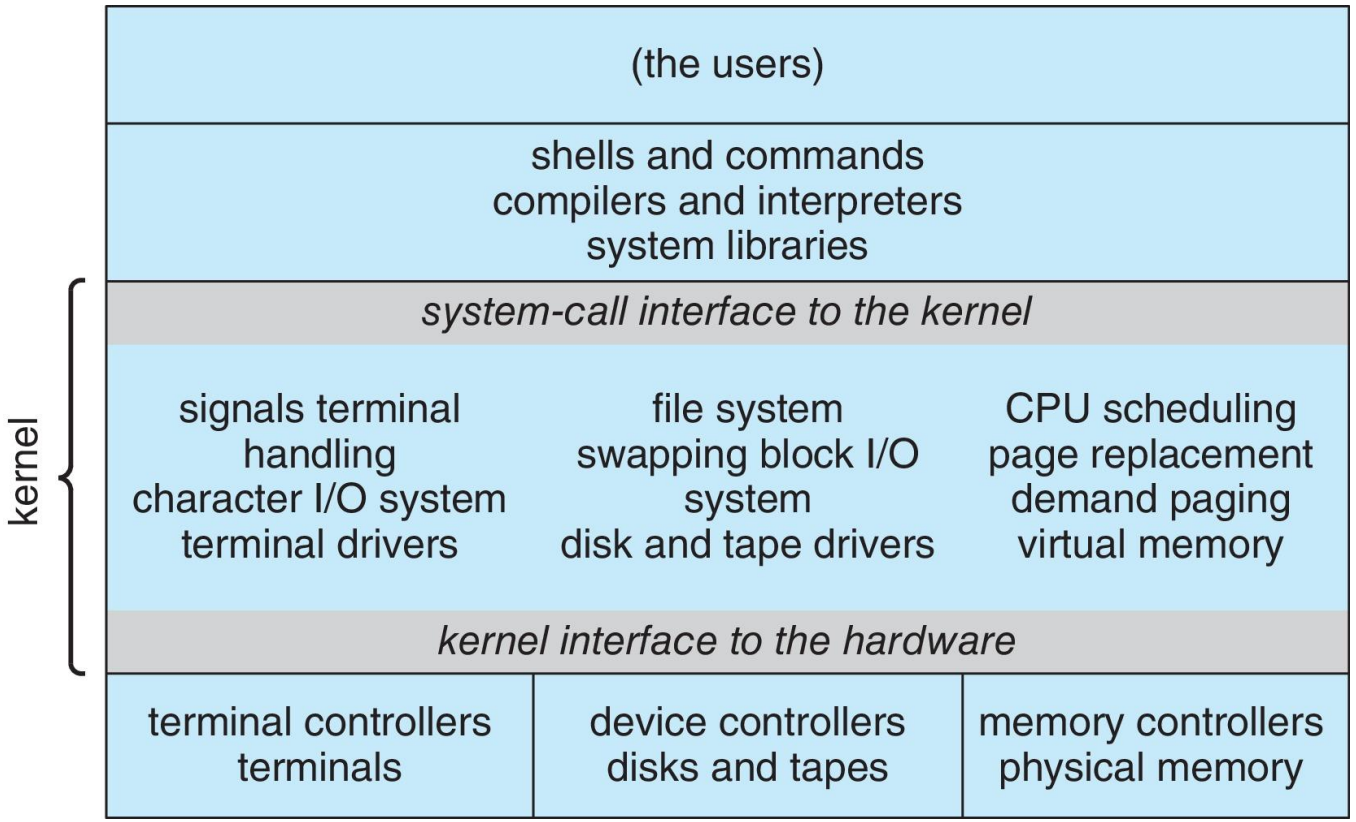


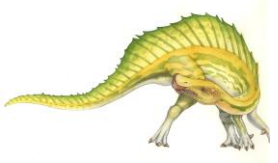
Troy University Resources



Traditional UNIX System Structure

Beyond simple but not fully layered

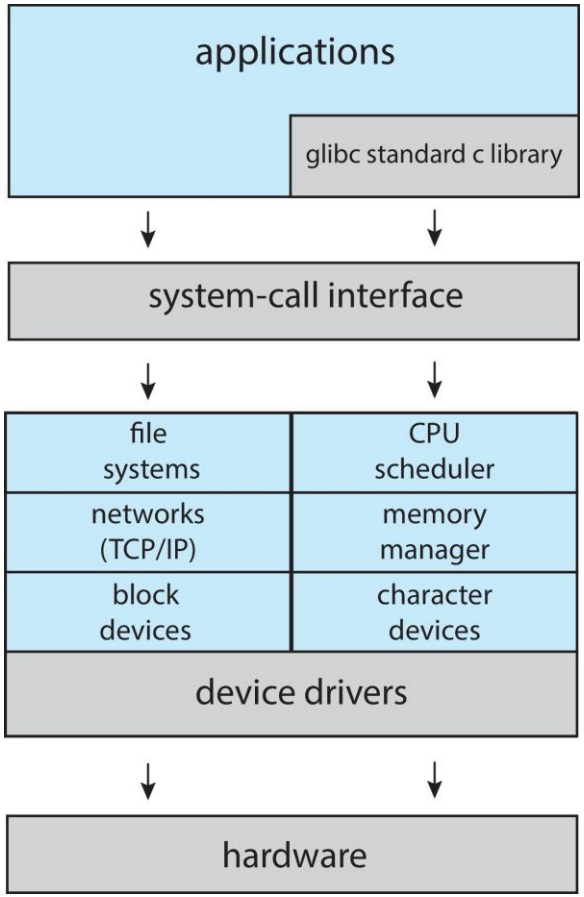


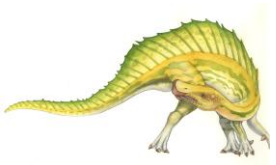




Linux System Structure

Monolithic plus modular design



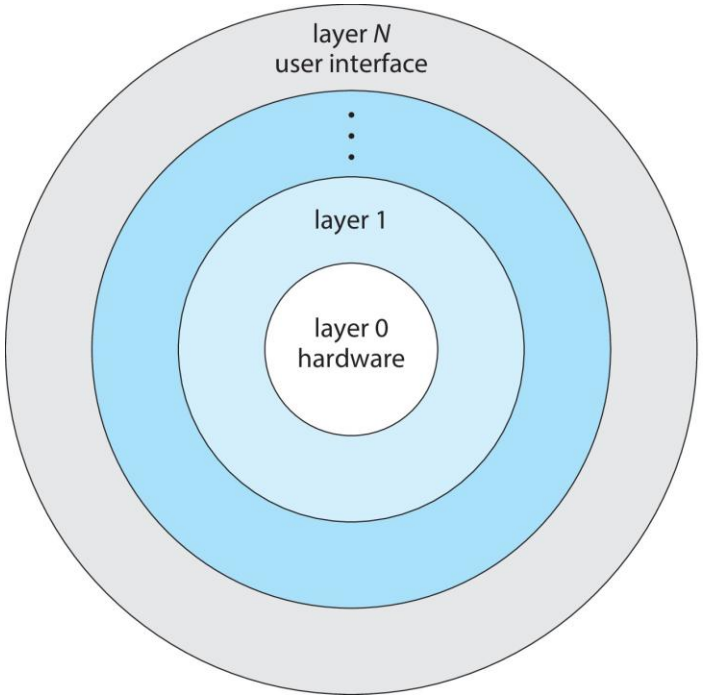


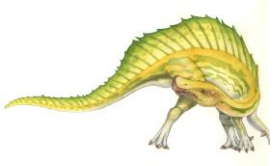
Troy University Resources



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







Microkernels

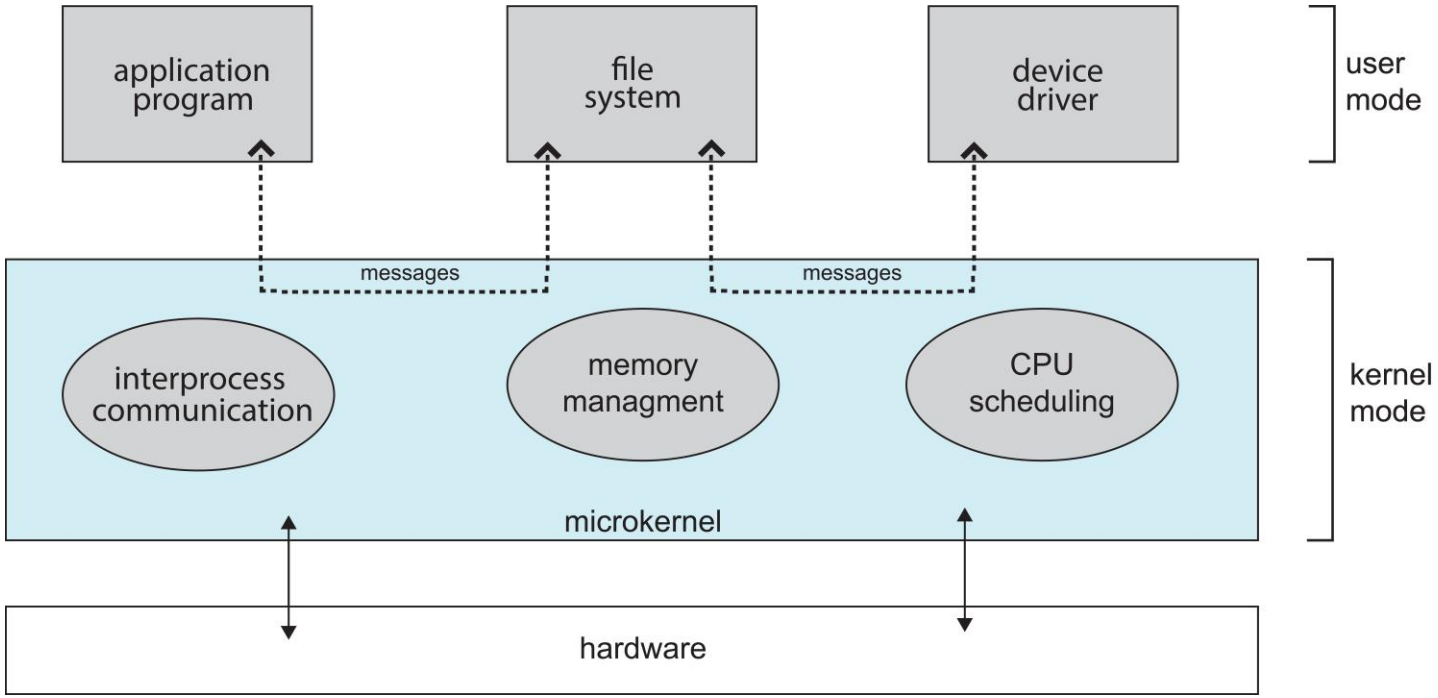
- Moves as much from the kernel into user space
- **Mach** is an example of **microkernel**
  - Apple's XNU hybrid kernel combines Mach, BSD and I/O Kit components.
- Communication takes place between user modules using **message passing**
- Benefits:
  - Easier to extend a microkernel
  - Easier to 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

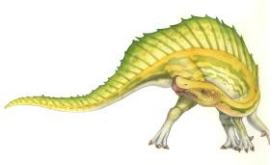


Troy University Resources



Microkernel System Structure

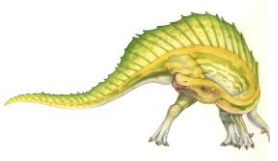






Modules

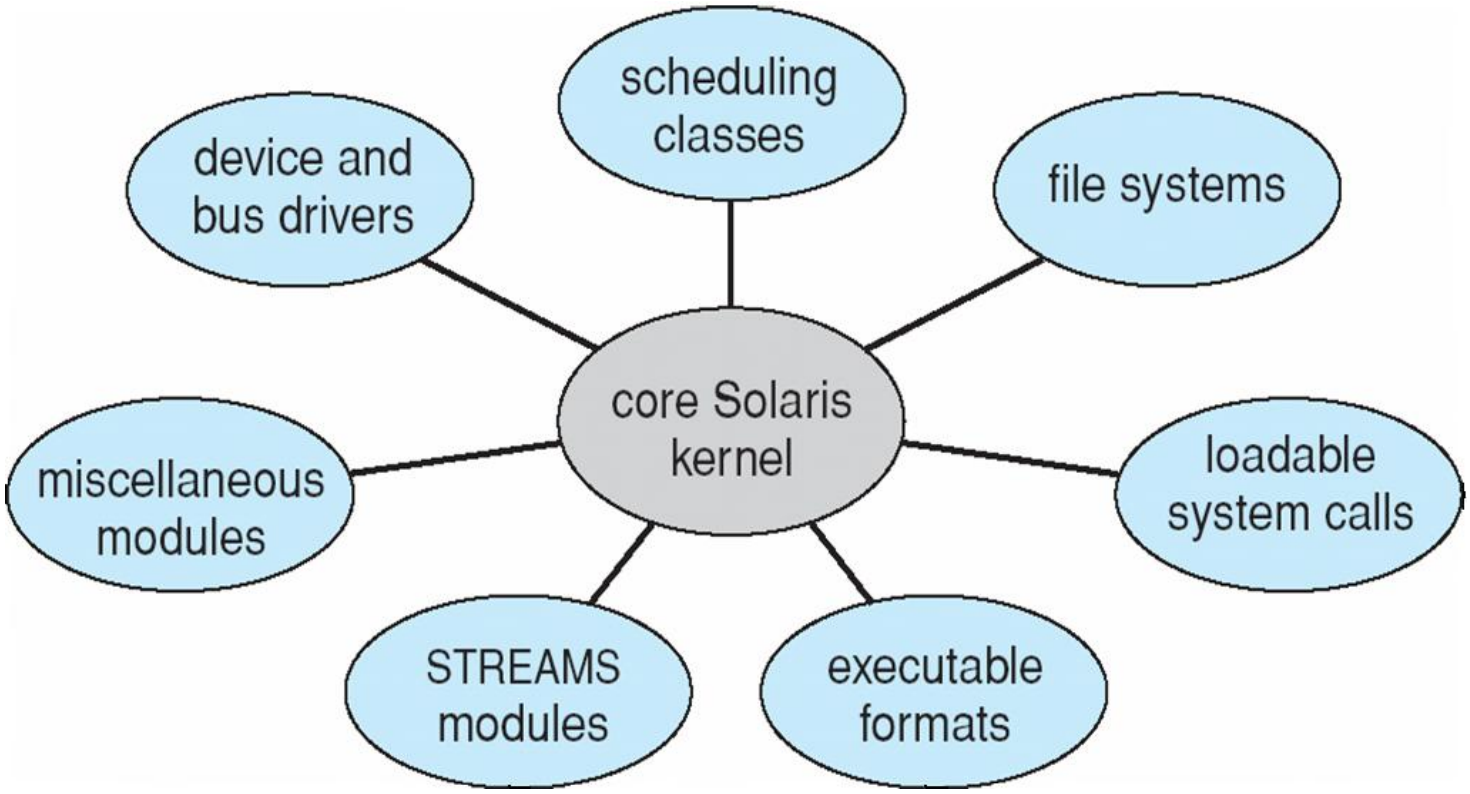
- Many modern operating systems implement **loadable kernel modules** (**LKMs**)
  - Uses modular components with defined interfaces
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible
  - Linux, Solaris, etc.



Troy University Resources



Solaris Loadable Kernel Modules



Loadable modules run in kernel space and use defined interfaces.
Modularity alone does not provide microkernel-style isolation.





Hybrid Systems

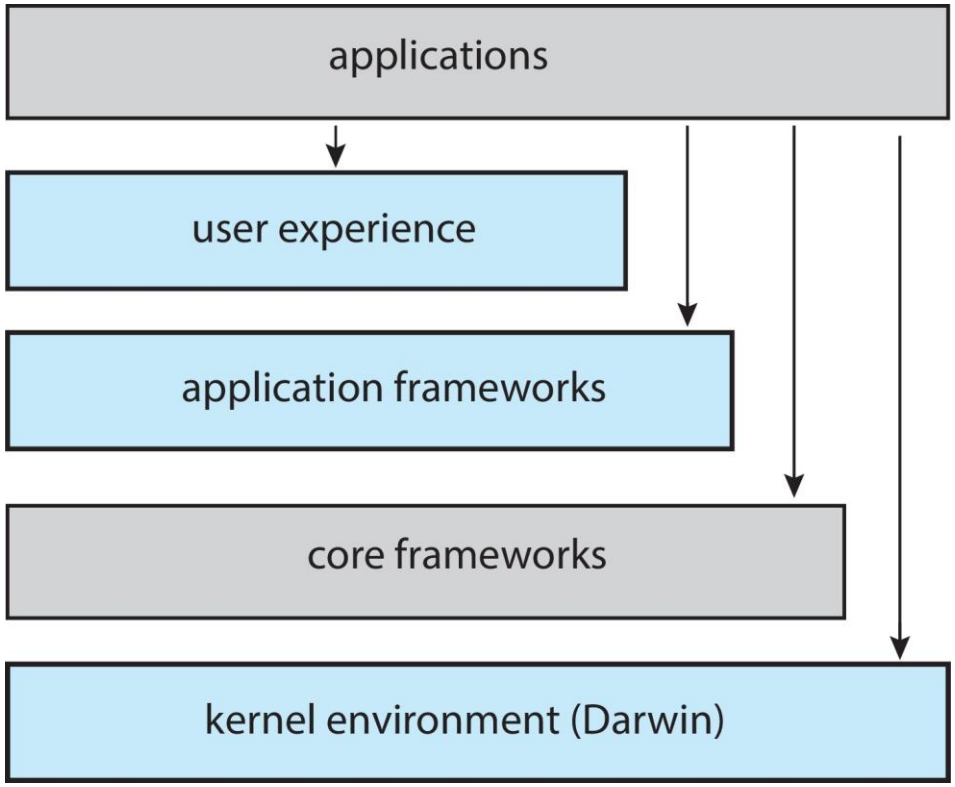
- Most modern operating systems are not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, plus microkernel for different subsystem ***personalities***
- Apple Mac OS X hybrid, layered, **Aqua** UI plus **Cocoa** programming environment
  - Below is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

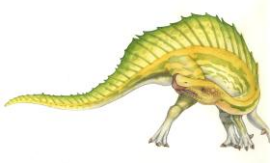


Troy University Resources



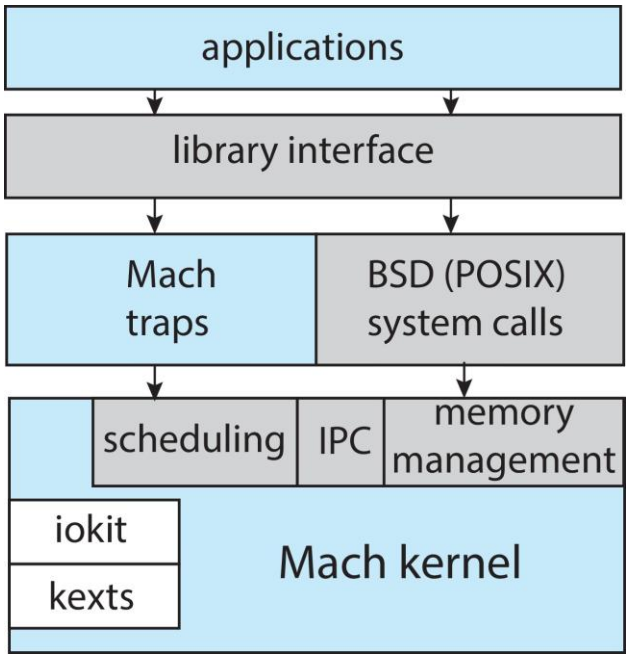
macOS and iOS Structure







Darwin



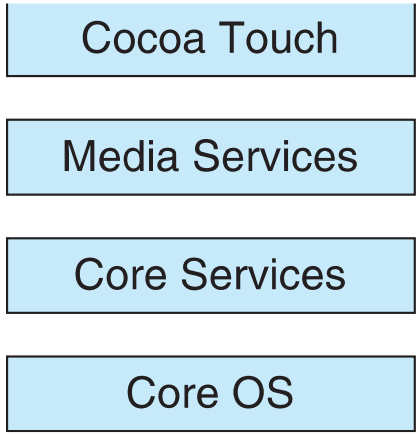


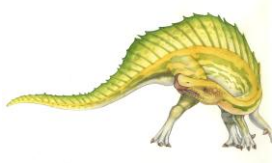
Troy University Resources



iOS

- iOS is Apple's operating system for iPhone. iPad uses the related iPadOS.
- iOS and macOS share Darwin foundations, including Mach and BSD components.
- Applications target platform-specific APIs and security rules.
- ARM versus Intel describes the historical comparison in this course. Modern Macs also use Apple silicon.
- Shared CPU architecture alone does not make an iOS binary a general macOS application.

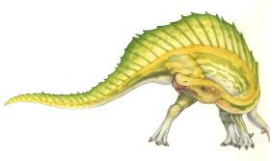






Android

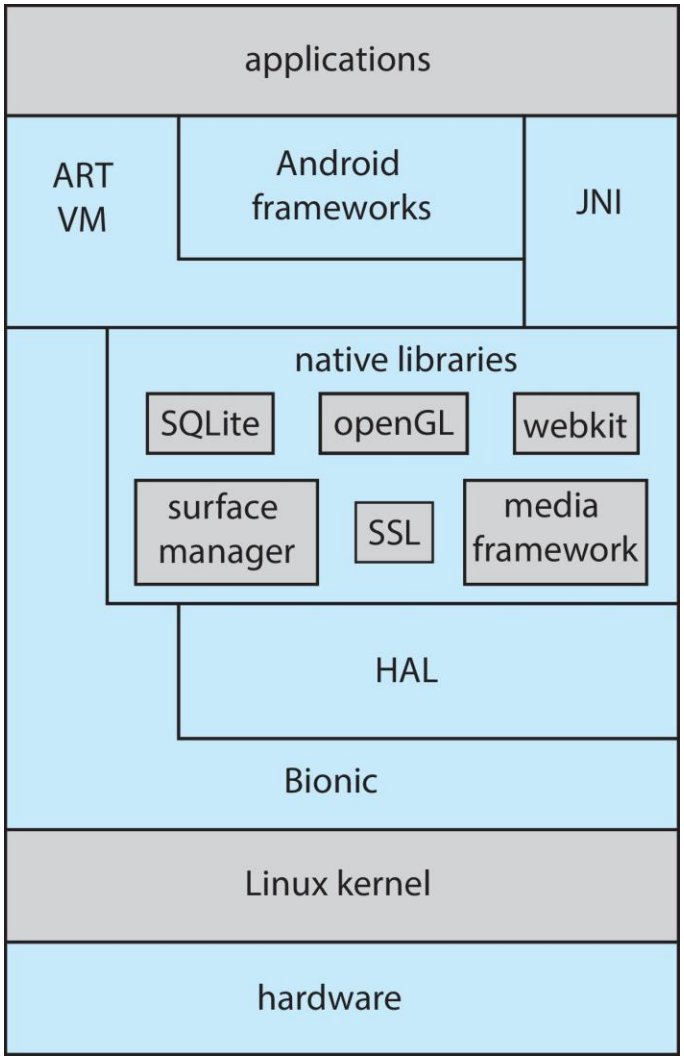
- Android uses a Linux kernel together with native libraries, a runtime and application frameworks.
- The kernel manages processes, memory, devices and security mechanisms.
- Android Runtime (ART) executes DEX bytecode and supports compilation and interpretation.
- Dalvik is ART's predecessor and remains relevant to older Android versions.
- Java or Kotlin source can compile to DEX. Apps may also use native C/C++ code through the NDK.

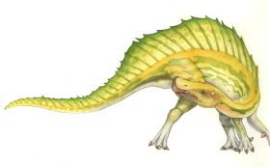


Troy University Resources



Android Architecture

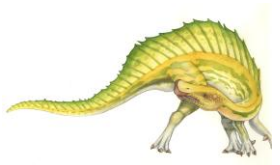






Building and Booting an Operating System

- Operating systems generally designed to run on a class of systems with variety of peripherals
- Commonly, operating system already installed on purchased computer
  - But can build and install some other operating systems
  - If generating an operating system from scratch
    - Write the operating system source code
    - Configure the operating system for the system on which it will run
    - Compile the operating system
    - Install the operating system
    - Boot the computer and its new operating system

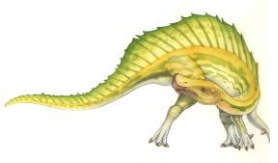


Troy University Resources



Building and Booting Linux

- Obtain the kernel source and build dependencies.
- Configure the kernel, for example with make menuconfig.
- Build the kernel and selected loadable modules with make.
- make modules_install installs modules under /lib/modules/<kernel-release>/.
- Install the kernel image and arrange boot-loader and initramfs integration as the distribution requires.
- make install behavior depends on the architecture and distribution. Loadable modules remain separate from the kernel image.





System Boot

- When power initialized on system, execution starts at a fixed memory location
- Operating system must be made available to hardware so hardware can start it
  - Firmware initializes hardware and starts a boot loader. The boot loader loads the kernel and transfers control to it.
  - Sometimes two-step process where **boot block** at fixed location loaded by ROM code, which loads bootstrap loader from disk
  - Modern systems replace BIOS with **Unified Extensible Firmware Interface** (**UEFI**)
- Common bootstrap loader, **GRUB**, allows selection of kernel from multiple disks, versions, kernel options
- Kernel loads and system is then **running**
- Boot loaders frequently allow various boot states, such as single user mode



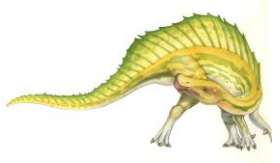
Troy University Resources



Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
- Also **performance tuning**
- OS 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
  - Sometimes using ***trace listings*** of activities, recorded for analysis
  - **Profiling** is periodic sampling of instruction pointer to look for statistical trends

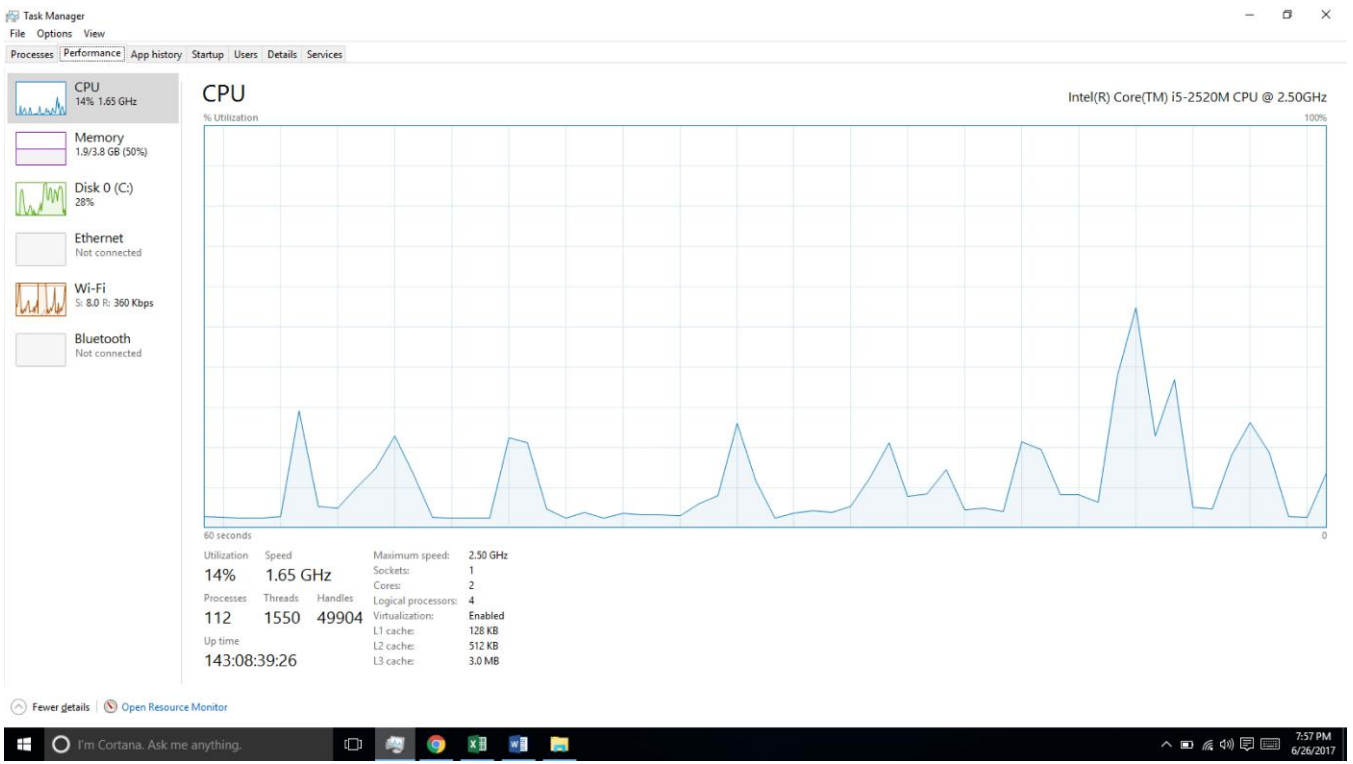
Kernighan's Law: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."





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



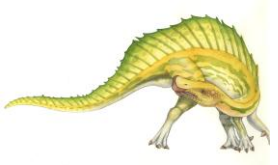


Troy University Resources



Tracing

- Collects data for a specific event, such as steps involved in a system call invocation
- Tools include
  - strace – trace system calls invoked by a process
  - gdb – source-level debugger
  - perf – collection of Linux performance tools
  - tcpdump – collects network packets

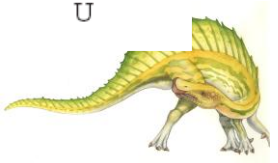




Dynamic Tracing with DTrace

- DTrace instruments a running system using probes and actions.
- Providers expose events such as system calls, scheduling changes and function entry or return.
- When an enabled probe fires, its action can record data or update an aggregation.
- The figure follows a user function into kernel activity 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
```



Troy University Resources



DTrace Scheduling Example

DTrace can aggregate CPU time for threads of a selected user. Scheduling probes mark entry to and exit from a CPU.

```
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.

timestamp is measured in nanoseconds. Per-thread self-> variables track intervals, and aggregations combine samples by process name.

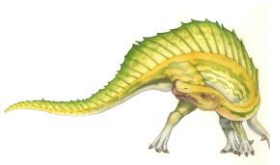




BCC

- BCC (BPF Compiler Collection) provides Linux tracing tools built around eBPF.
- Tracing can connect user-space operations with the kernel events they trigger.
- For example, disksnoop.py records disk I/O events and their latency.
- The following slide shows other tracing tools.

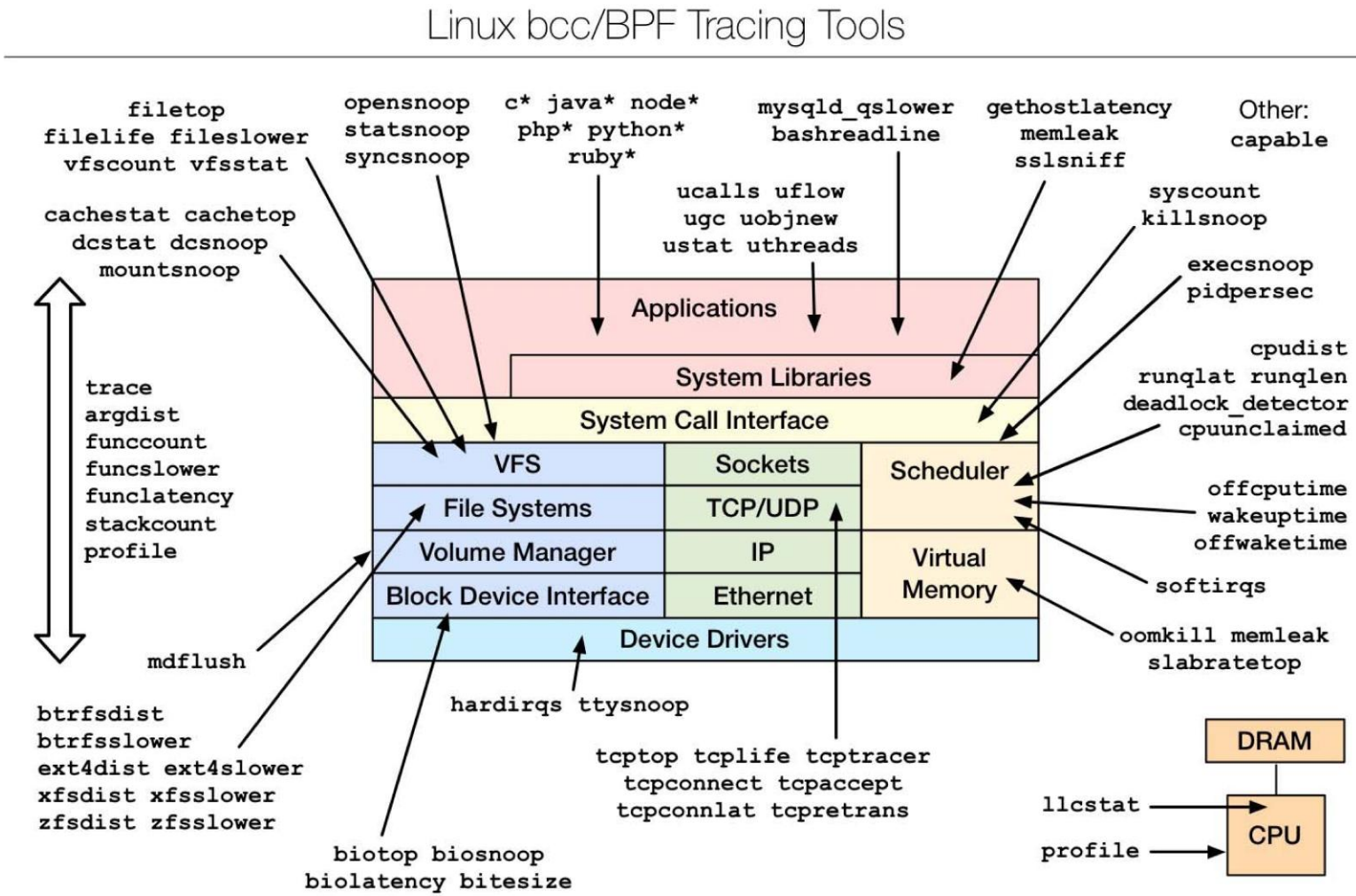
```
TIME(s)          T      BYTES      LAT(ms)
1946.29186700    R      8              0.27
1946.33965000    R      8              0.26
1948.34585000    W      8192           0.96
1950.43251000    R      4096           0.56
1951.74121000    R      4096           0.35
```

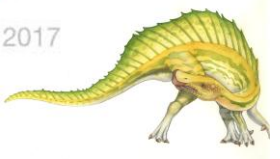


Troy University Resources

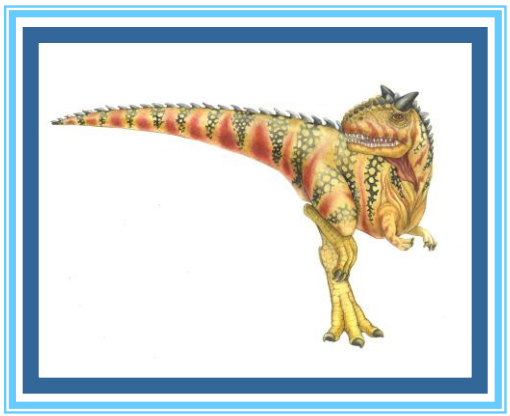


Linux bcc/BPF Tracing Tools





End of Chapter 2



Silberschatz, Galvin and Gagne ©2018