UNIVERSITY OF BIRMINGHAM

Computer Systems
Operating System Structures & User Interface

Lecture Objective / Overview

In this lecture, we shall see:

- Operating System Services
- System Calls / OS Relationship
- OS Design and Implementation
- Operating System Structure
- User Classes & Interfaces

What is part of an Operating System?

- When you install an OS, what do you get?
  - **System programs** - program loader, command interpreter
  - **Language processors** - C compiler, assembler, linker
  - **Utilities** - text editor, terminal emulator
  - **Subroutine libraries** - standard C library, JVM

Operating System Services

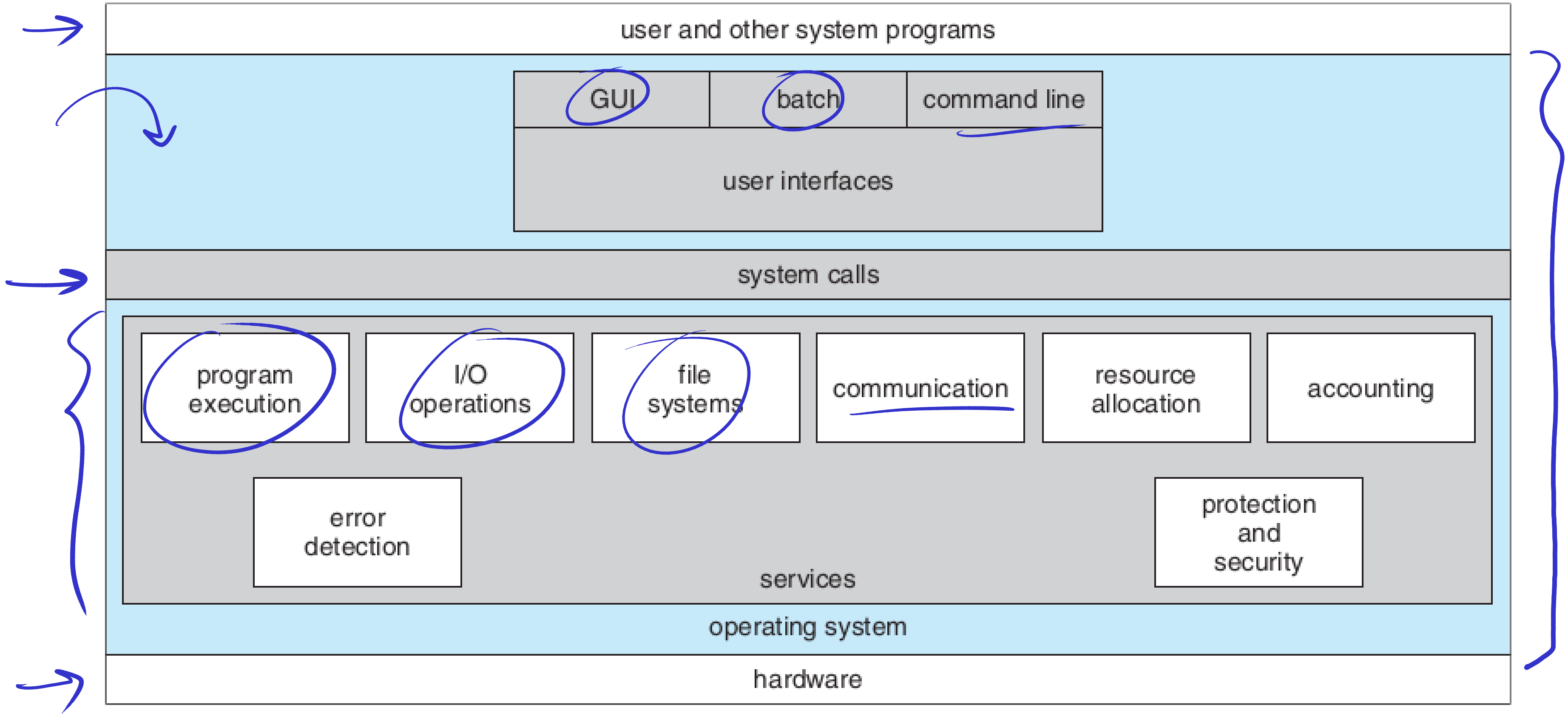


Figure 2.1 A view of operating system services.

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 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)

System Calls – Example

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:

100

```
#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.

API – System Call – OS Relationship

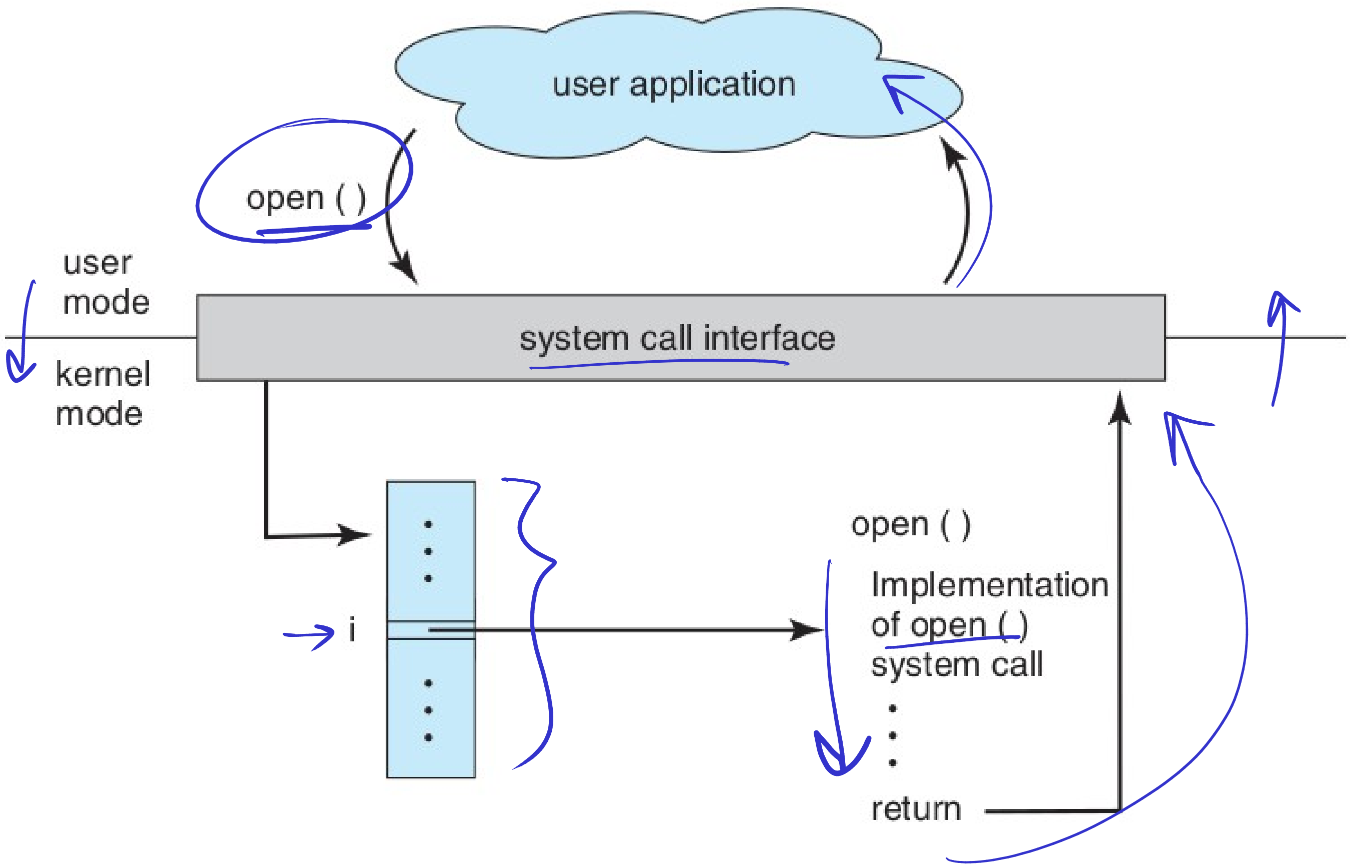


Figure 2.6 The handling of a user application invoking the open() system call.

System Call Implementation

- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller doesn't need to know implementation details
  - Just needs to obey API and understand what OS will do!
  - Most details of OS interface hidden from programmer by API
  - Managed by run-time support libraries (set of functions built into libraries included with compiler)

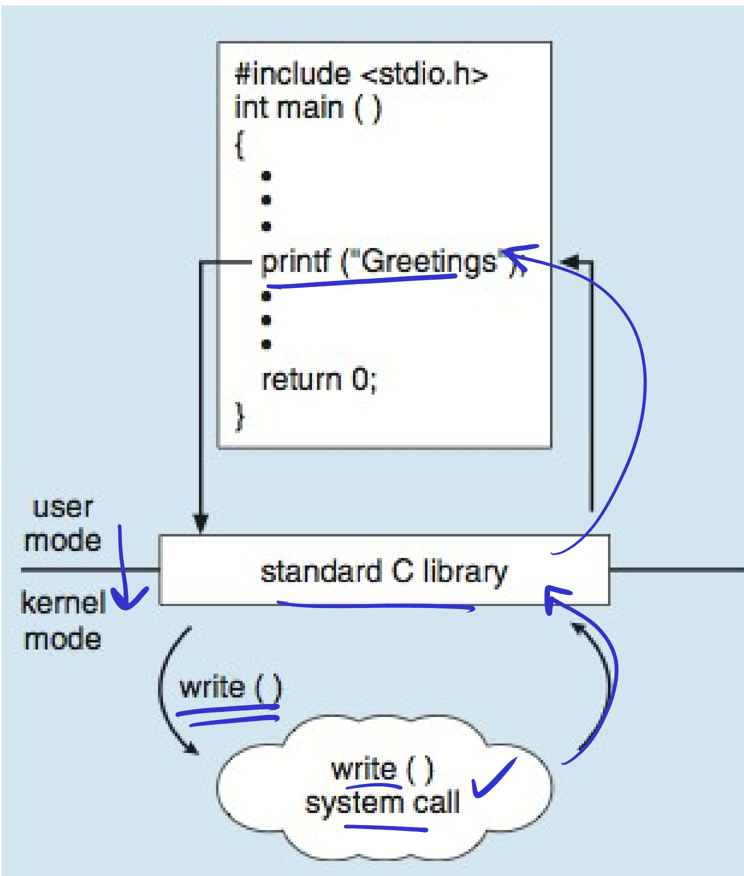
Examples of Windows and Unix System Calls

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

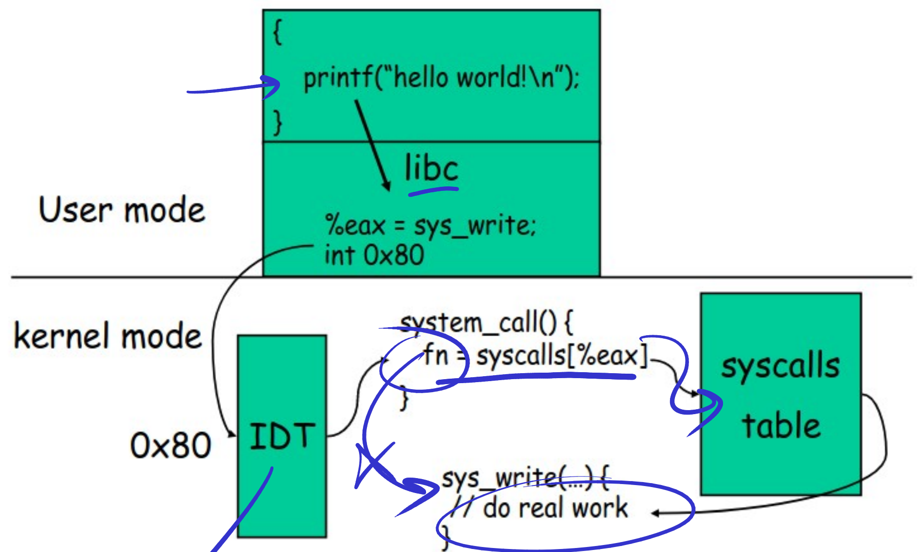
	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information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

Standard C Library Example

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



%eax = sys_write
int 0x80 0x82



Interrupt Descriptor Table

Operating System Design and Implementation

- Design and Implementation of OS not "solvable", but some approaches have proven successful
- Internal structure of 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

Implementation

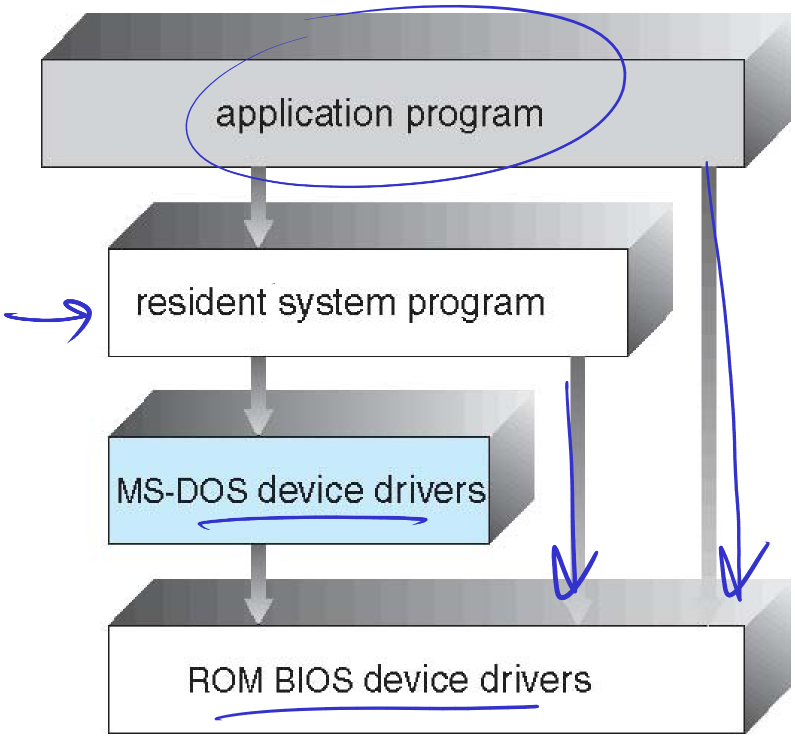
- Much variation
  - Early OSes in assembly language
  - Then system programming languages like Algol, PL/1
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to **port** to other hardware
  - But slower
- **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 OS
  - Modular and Hybrid

Simple Structure – MS DOS

- 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

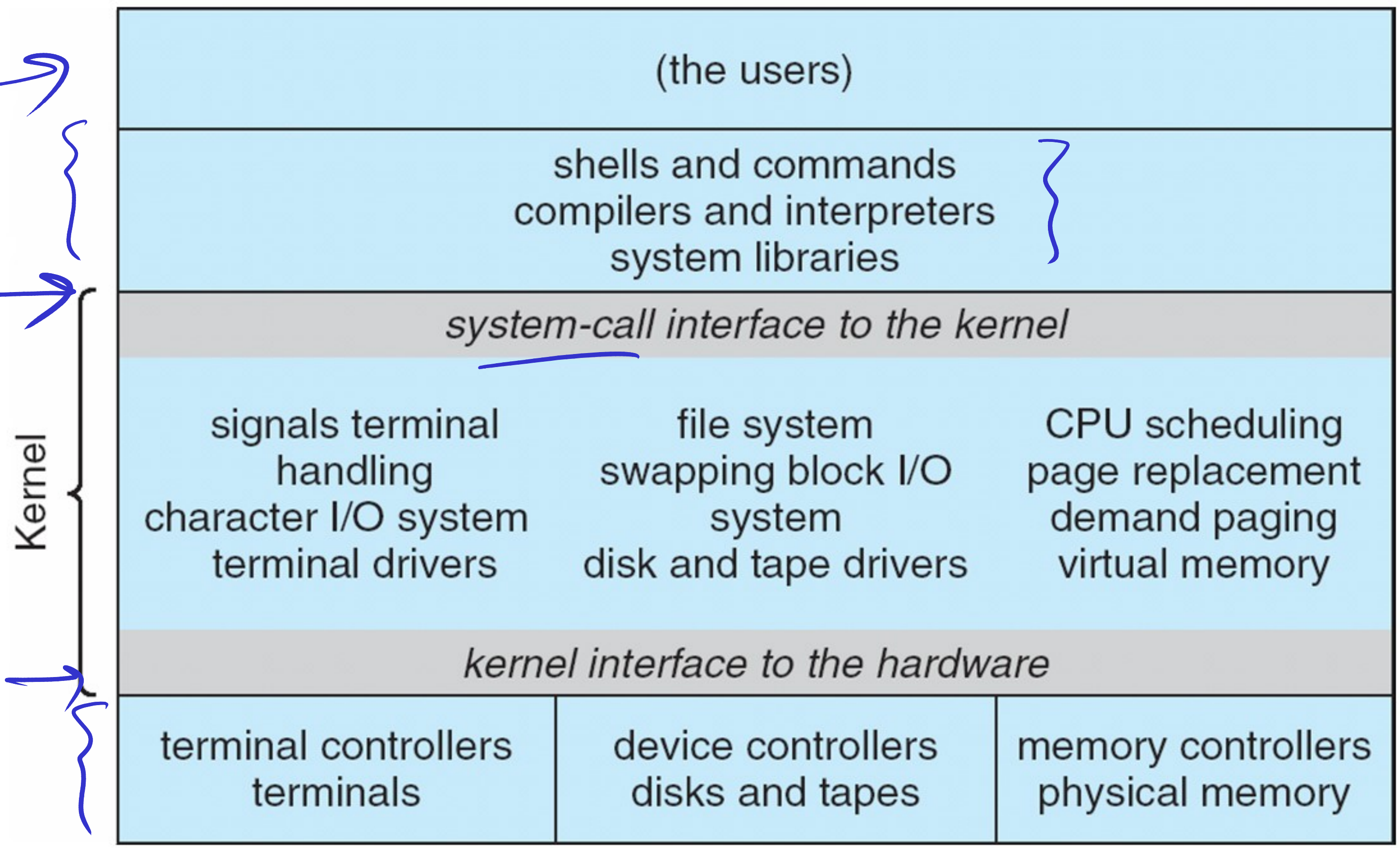


Non Simple Structure – UNIX

- 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

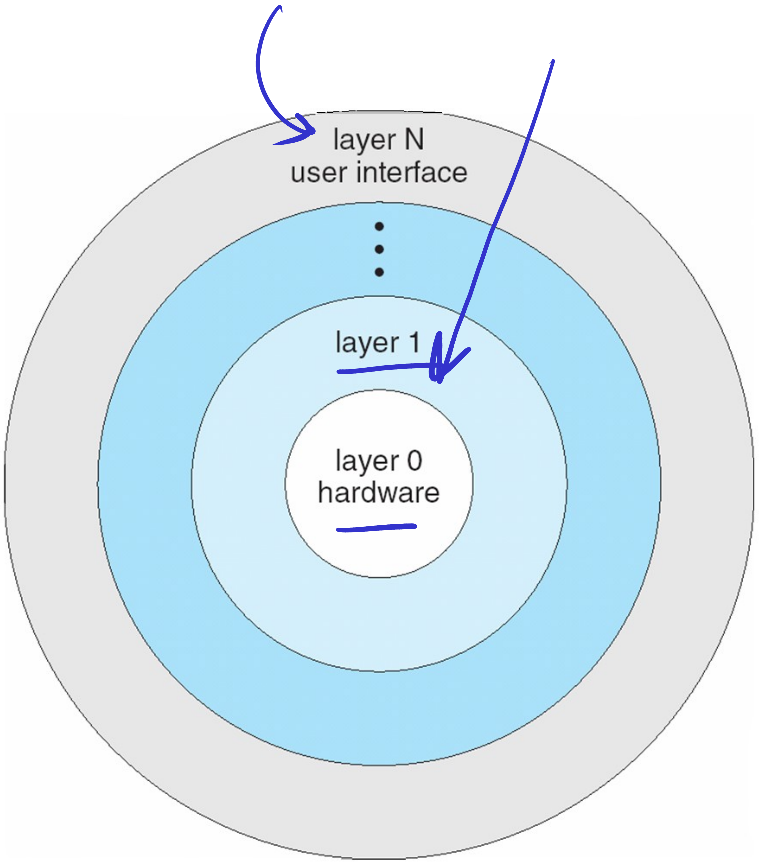
Traditional UNIX System Structure

- Beyond simple but not fully layered!



Layered Approach

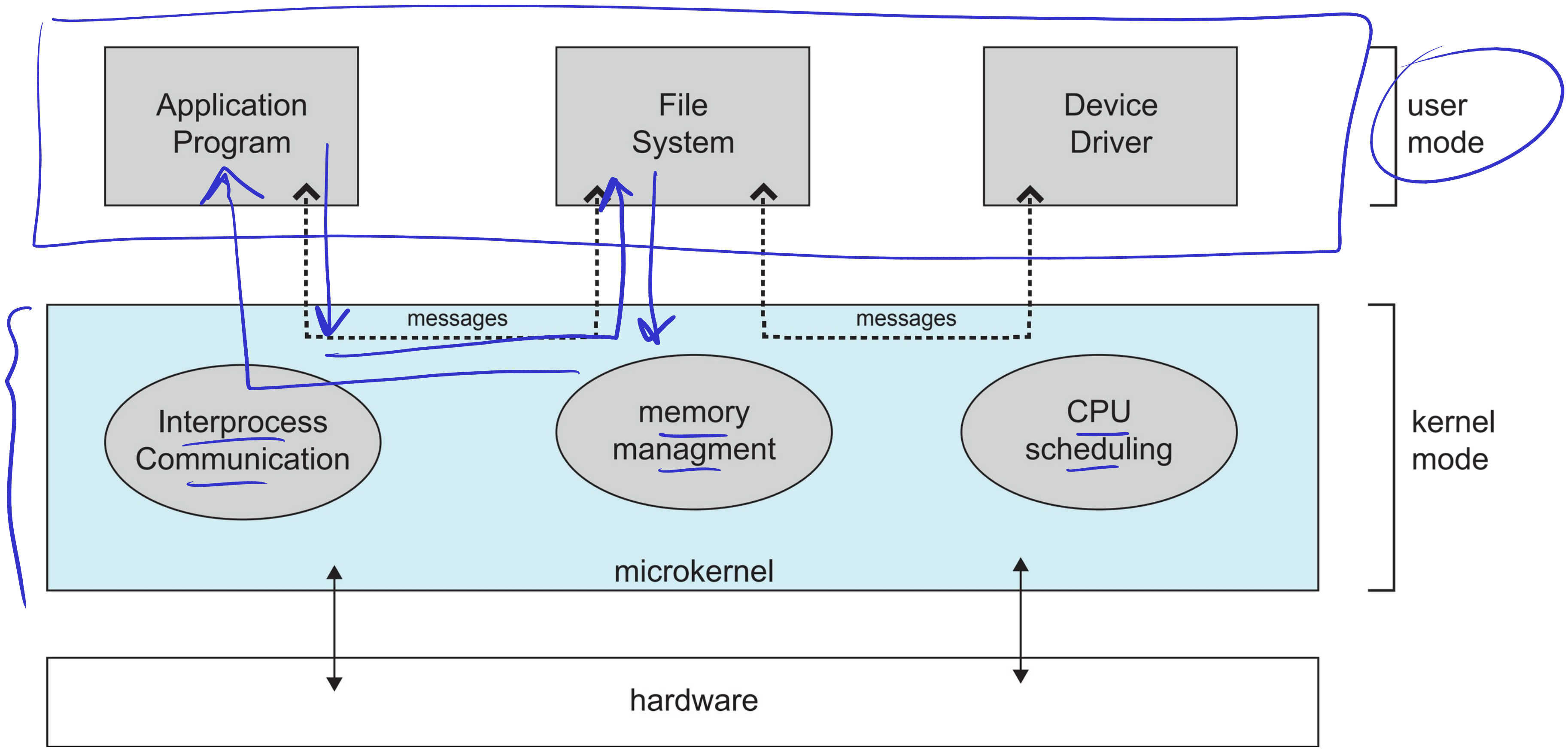
- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level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

- Many 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

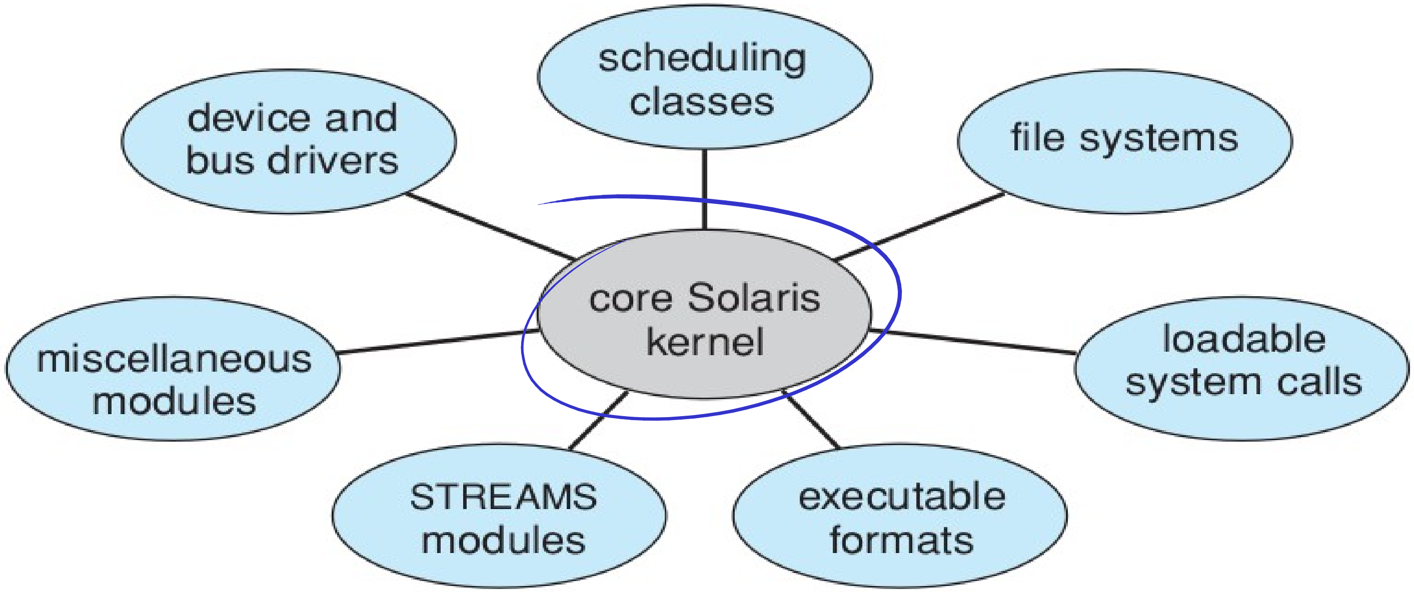
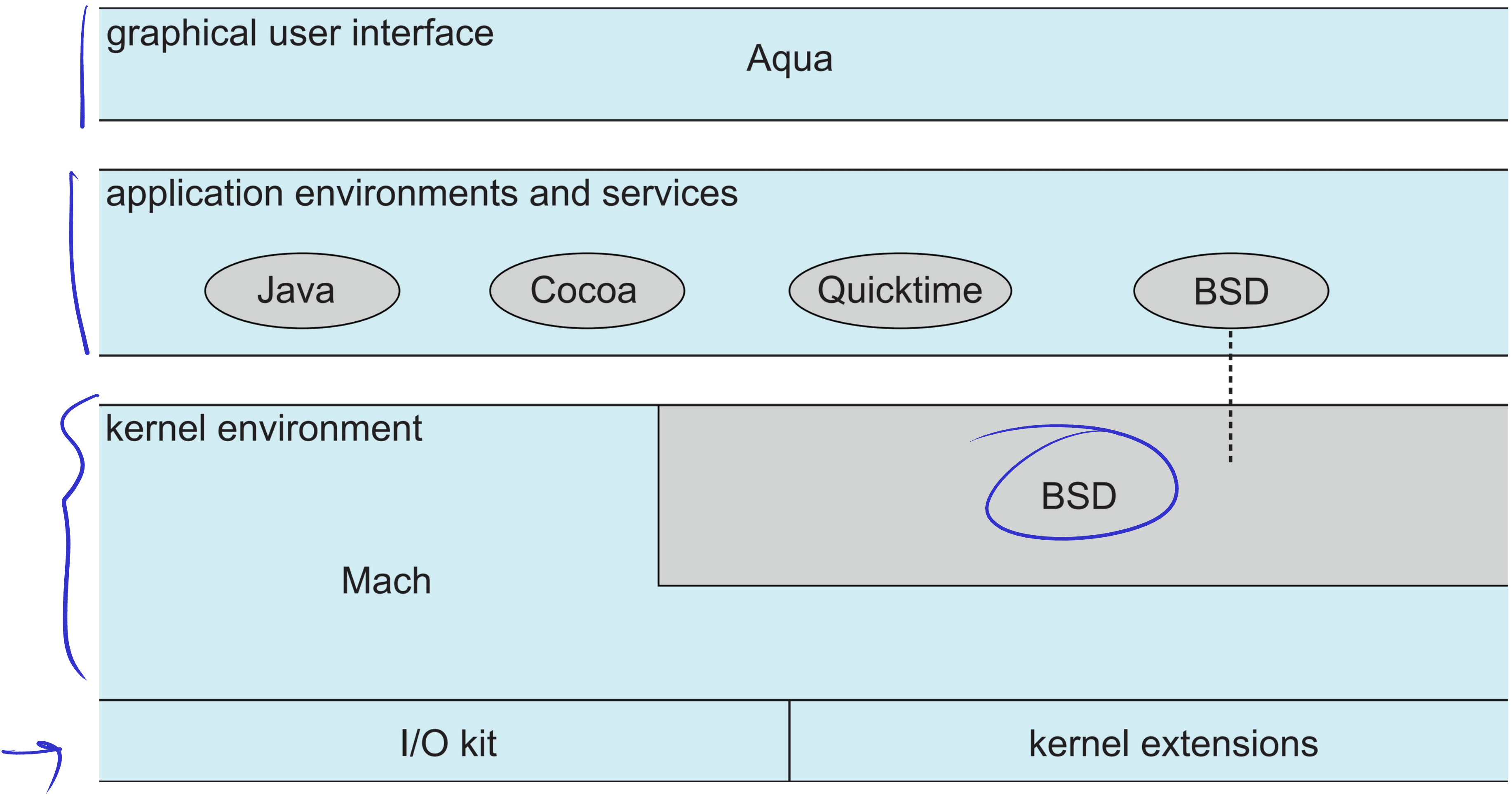


Figure 2.15 Solaris loadable modules.

Hybrid Systems

- Most modern operating systems are 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***
- Apple Mac OS X hybrid, layered, **Aqua** UI plus **Cocoa** programming environment
  - Next slides shows a kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

Mac OS X Structure



User Interfaces

Almost all operating systems have a user interface.

This interface can take several forms.

- Command-line Interface (CLI)
- Batch Interface - commands and directives entered into files, which are then executed
- Graphical User Interface (GUI)

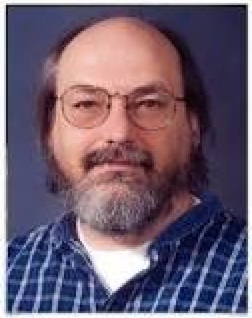
User Interfaces – Key Information

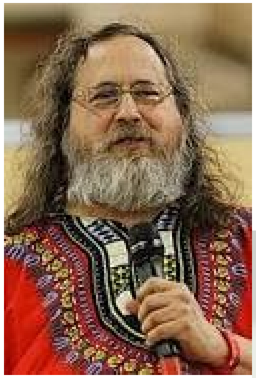
- Any UI requires a software link to hardware
  - This link may be buried under other software
- Most OS's provide a set of system calls that invoke low level operating system functions
- System calls can be invoked directly, but are often hidden from user

User Classes

1) Programmers

- Produce system or application software (and drink a lot of coffee)
  - System programmer = Operating Systems, compilers, devices drivers etc.
    - Require low level access to machine facilities e.g. System calls.
  - Application programmer = Spreadsheets, DBs, Mobile / Web Apps etc.






User Classes

2) Operational (Admins)

- Concerned with provision, operation and management of computing facilities

User Classes

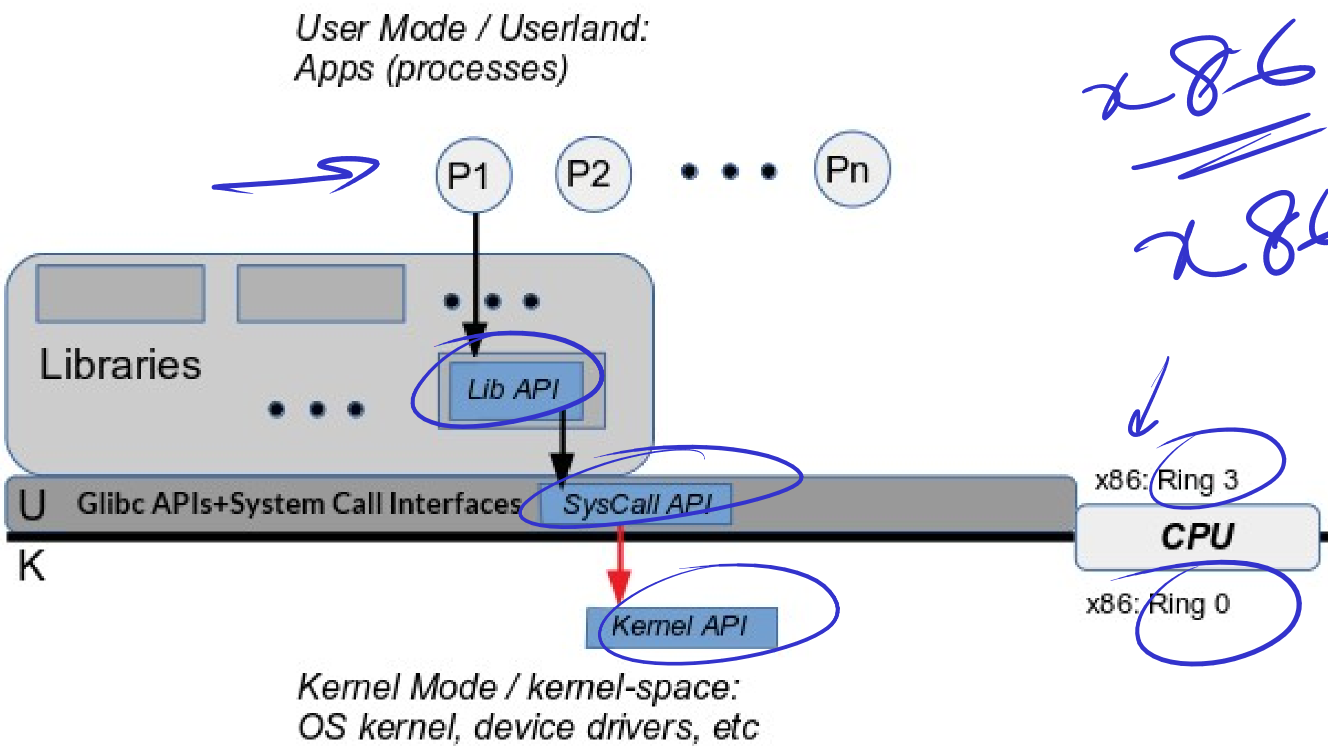
3) End-users

- Someone who applies software to some problem area. Two extremes:
  - Unaware they're interacting with a computer
  - Substantial understanding of computer

Types of Interface

1) System Calls:

- All interaction with hardware has to go though system call
- OS provides layer of subroutines called an API



x86 32bit

x86_64

Types of Interface

2) Command Language:

- Most OS's provide an interactive terminal
- Commands can be entered here
- Used to imitate programs
- Perform housekeeping control routines on system
- UNIX provides shell programs

Types of Interface

hamayun@dell-e6430: ~/workspace

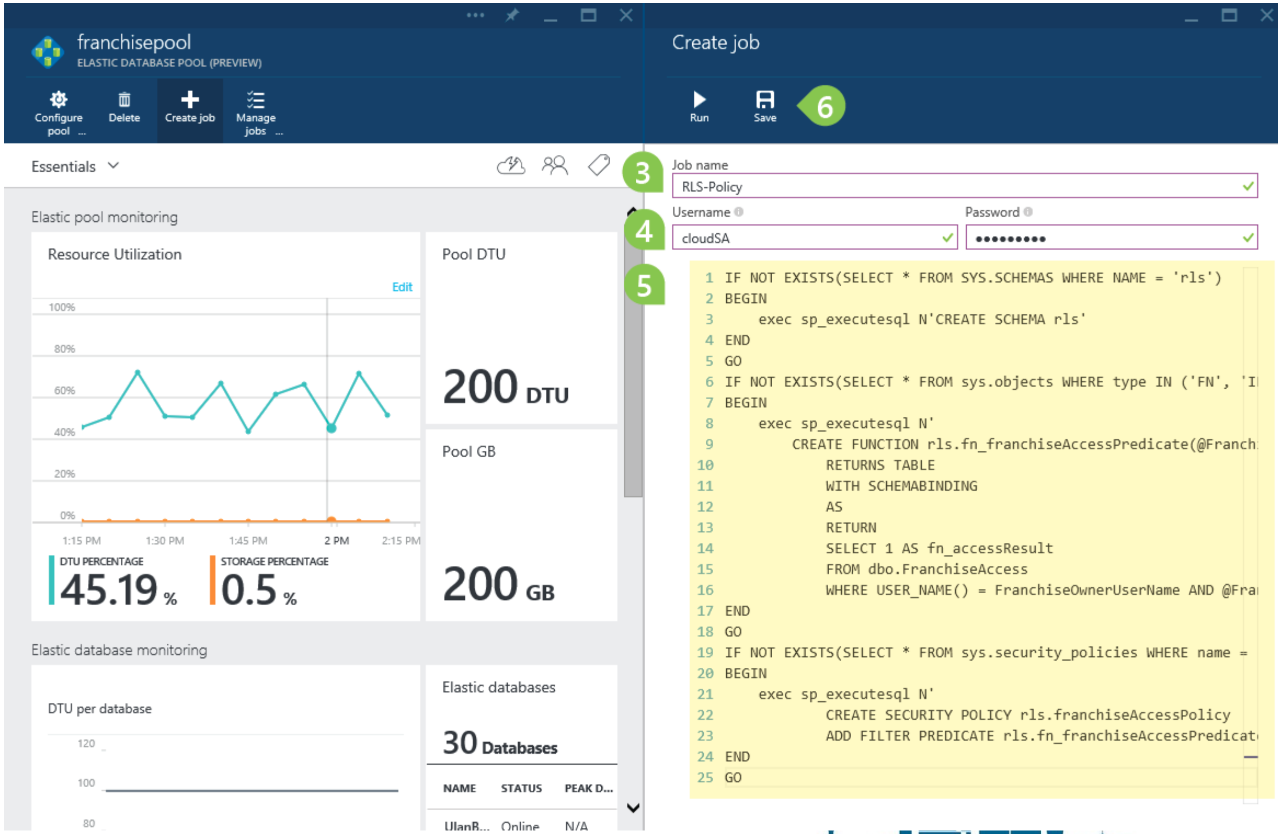
File Edit View Search Terminal Help

```
hamayun@dell-e6430:~/workspace$ date
Wed Jan 23 02:33:26 +04 2019
hamayun@dell-e6430:~/workspace$ echo "This is the Shell"
This is the Shell
hamayun@dell-e6430:~/workspace$ !!
echo "This is the Shell"
This is the Shell
hamayun@dell-e6430:~/workspace$ uptime
 02:33:44 up 14:46,  1 user,  load average: 0.17, 0.36, 0.54
hamayun@dell-e6430:~/workspace$ w
 02:33:52 up 14:47,  1 user,  load average: 0.14, 0.35, 0.53
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
hamayun  :0       :0               Tue11   ?xdm?  48:08   0.00s /usr/lib/gdm3/gd
m-x-session --run-s
hamayun@dell-e6430:~/workspace$ pwd
/home/hamayun/workspace
hamayun@dell-e6430:~/workspace$ 
```

Types of Interface

3) Job Control Language:

- Define requirements for work submitted to a batch system
- Used for DBs



Types of Interface

4) Graphical User Interface

- Interaction via windows and mouse driven environment
- Desktop, Icons, GUI APIs - Java X Windows
- Programmer's Perspective
  - Slightly more complex than shell script, but more rewarding
  - Event driven programming - responsive to user actions
- User viewpoint - friendlier?
- Increased processing load

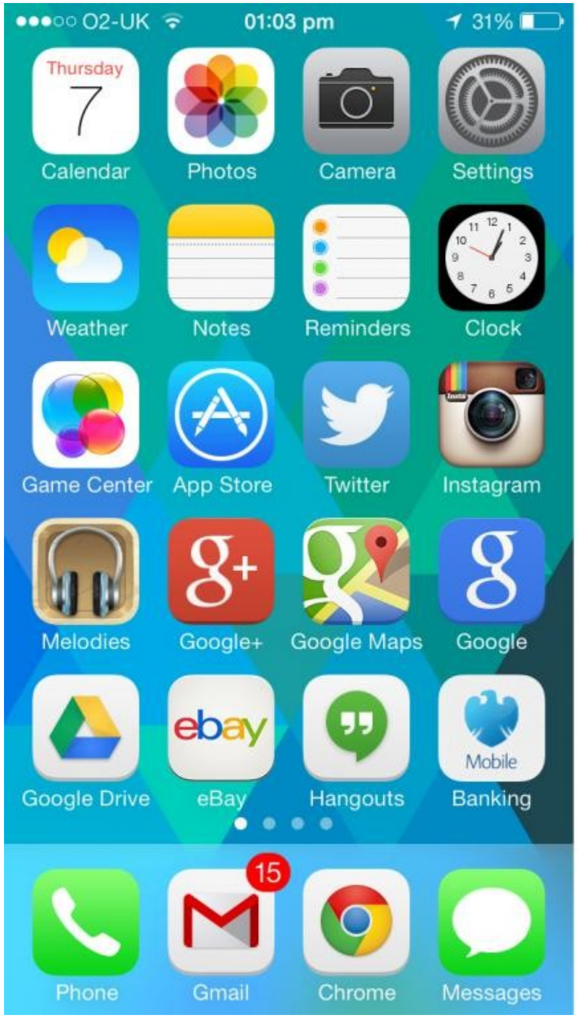
Desktop File Edit View Go Special Window Help
Finder
Computer
Home
Favorites
Apps
Docs
Users
3.1 GB avail., 17 items
Applications
Mac OS X
Computer
AddressBook
Calculator
Classic
Clock
GrabBag
Internet Explorer
Key Caps
Mail
Music Player
PPP Connect
Preview
QuickTime Player
Sherlock
Stickies
System Preferences
Audio CD
Ask
Mac OS X
About Mac OS X applications
About Classic applications
Late-Breaking News
Français
Deutsch
Software Update
Update Now
Software Update was last run at 4:43 PM on 12/31/69.
Update software automatically
Click the lock to make changes.

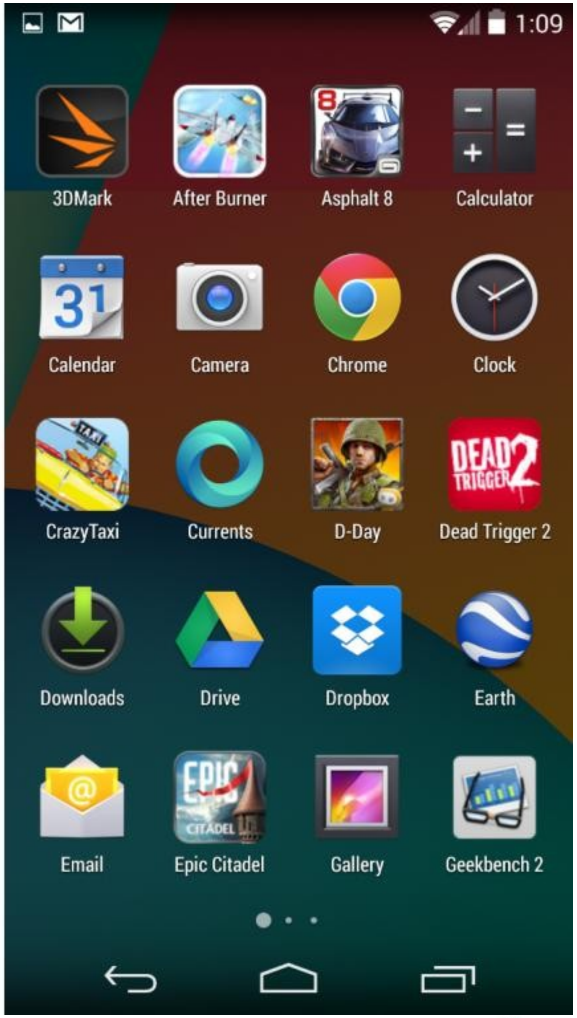
Types of Interface

5) Touchscreen Interfaces

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

Types of Interface





Summary

What elements of a computer system have we looked at:

- Operating System Services
- System Calls / OS Relationship
- OS Design and Implementation
- Operating System Structure
- User Classes & Interfaces

References / Links

- Chapter # 2: **Operating System Concepts** (9th edition) by Silberschatz, Galvin & Gagne