

Chapter 2

Design Principles

2.1. Functionality of OS

The operating system acts as an interface between the hardware and the programs requesting I/O. It is the most fundamental of all system software programs. In a desktop computer, these resources include such things as the processor, memory, disk space and more (On a cell phone, they include the keypad, the screen, the address book, the phone dialer, the battery and the network connection).

Responsibilities of the OS include:

- Hiding the complexities of hardware from the user
- Managing between the hardware's resources which include the processors, memory, data storage and I/O devices
- Handling "interrupts" generated by the I/O controllers
- Sharing of I/O between many programs using the CPU
- It provides a stable, consistent way for applications to deal with the hardware without having to know all the details of the hardware.

The most well-known Operating Systems include:

- **System Software** - programs that manage the operation of a computer.
- **Application Software** - programs that help the user perform a particular task.
- The first task, managing the hardware and software resources, is very important, as various programs and input methods compete for the attention of the **central processing unit** (CPU) and demand memory, storage and input/output (I/O) bandwidth for their own purposes. In this capacity, the operating system plays the role of the good parent, making sure that each application gets the necessary resources while playing nicely with all the other applications, as well as husbanding the limited capacity of the system to the greatest good of all the users and applications.
- The second task, providing a consistent application interface, is especially important if there is to be more than one of a particular type of computer using the operating system, or if the hardware making up the computer is ever open to change. A consistent **application program interface** (API) allows a software developer to write an application on one computer and have a high level of confidence that it will run on another computer of the same type, even if the amount of memory or the quantity of storage is different on the two machines.

Even if a particular computer is unique, an operating system can ensure that applications continue to run when hardware upgrades and updates occur. This is because the operating system -- not the application -- is charged with managing the hardware and the distribution of its resources. One of the challenges facing developers is keeping their operating systems flexible enough to run hardware from the thousands of vendors manufacturing computer equipment.

Today's systems can accommodate thousands of different printers, disk drives and special peripherals in any possible combination.

2.2. Client Server and Hand-held Devices

A handheld computer is a computer that can conveniently be stored in a pocket (of sufficient size) and used while you're holding it.

Handheld systems include personal digital assistants (PDAs), such as Palm- Pilots, or cellular telephones with connectivity to a network such as the Internet. Developers of handheld systems and applications face many challenges, most of which are due to the limited size of such devices. For example, a PDA is typically about 5 inches in height and 3 inches in width, and it weighs less than one-half pound. Due to this limited size, most handheld devices have a small amount of memory, include slow processors, and feature small display screens. We will take a look now at each of these limitations.

The first issue being memory, the second issue of concern to developers of handheld devices is the speed of the processor used in the device. Processors for most handheld devices often run at a fraction of the speed of a processor in a PC. Faster processors require more power. To include a faster processor in a handheld device would require a larger battery that would have to be replaced (or recharged) more frequently. To minimize the size of most handheld devices, smaller, slower processors which consume less power are typically used. Therefore, the operating system and applications must be designed not to tax the processor.

The last issue confronting program designers for handheld devices is the small display screens typically available. Whereas a monitor for a home computer may measure up to 21 inches, the display for a handheld device is often no more than 3 inches square. Familiar tasks, such as reading e-mail or browsing web pages, must be condensed onto smaller displays. One approach for displaying the content in web pages is web clipping, where only a small subset of a web page is delivered and displayed on the handheld device.

Some handheld devices may use wireless technology, such as Bluetooth, allowing remote access to e-mail and web browsing. Cellular telephones with connectivity to the Internet fall into this category.

2.3. Design Issues

Functional correctness is usually the most important quality of a system. To achieve functional correctness, we should understand the specifications well (we need to know what is the 'correct' behavior expected in the first place).

In addition, a system is required to have many non-functional qualities (i.e. qualities that are not directly related to its functionality), some explicitly mentioned, some only implied. The nature of the system decides which qualities are more important. Find out the relative importance of each quality, as you might have to sacrifice one to gain another. There are many to choose from such as Analyzability, Adaptability, Changeability, Compatibility, Configurability, Conformance, Efficiency, Fault tolerance, Install ability/uninstall ability, Interoperability Learnability, Localizability,

Maintainability, Portability, Performance, Replicability, Reliability, Reusability, Security, Scalability, Stability, Supportability, Testability, Understandability, and Usability.

Most grading schemes have some marks allocated for certain NF qualities. Note that some NF qualities are not easily visible during a product demo. Be sure to talk about them during the presentation, and in the report. Given next are some common NF qualities and some tips on achieving each.

2.3.1. Reliability

You need to show that your system can perform without failure for the specified operating environment. If the system is expected to run for a long period, you should ensure that it indeed can. For example, monitor the memory usage while the system is in operation; if it keeps increasing, your system might have a memory leak, which will crash it eventually.

2.3.2. Robustness

Can your system withstand incorrect input? Will it gracefully terminate when pushed beyond the operating conditions for which it was designed? Does it warn the user when approaching the breaking point or simply crash without warning? If you can crash your system by providing wrong input values, then it is not a very robust system.

2.3.3. Maintainability

Do not think maintainability is unimportant because this is just a class project or because the instructor did not mention it specifically. The truth is that you will 'maintain' your code from the moment you write it. You will really regret writing unmaintainable code especially during bug fixing and modifying existing features towards the end of the project.

Keeping the design and eventually, its implementation, simple and easy-to-understand can really boost the maintainability of your system. A comprehensive suite of automated test cases can detect undesirable ripple effects of a change on the rest of the system. Having such tests indirectly helps maintainability. "Always code as if the person who ends up maintaining your code will be a violent psychopath who knows where you live." – Anonymous. Maintainability is of utmost importance if you plan to release the product to the public and keep it alive after the course.

2.3.4. Efficiency

As software engineers, we should always give due consideration to the efficiency of what we are building. It becomes especially important if multiple teams are building similar products and more efficient products stand to gain extra marks. Here are some guidelines you can consider adopting.

- **Define.** Find out which kind of efficiency you should be focussing on. It could be the space efficiency (how efficiently you use storage space) that you should worry about. It could also be the time efficiency (how fast the system can work). In some cases, it can be a more specific aspect such as the time it takes to process a certain input.
- **Do not speculate; find out.**

- o If you try to optimise the parts that you *think* might be slow, you could end up optimizing the wrong part, especially, given your inexperience in the field. While you are busy looking for a super-fast sorting algorithm, your system could be slow simply because you use *Strings* instead of *StringBuffers*.
- o Use appropriate tools (e.g. profilers such as CLRProfiler and DotTrace) to find out where in the system you need to optimise for performance. Note that black-box performance testing is different from profiling. Remember to stay focussed on the specific aspect you want to profile. Do not get distracted by different kinds of data produced by the profiling tool. Sometimes, it is good to use multiple tools to confirm a finding, especially if the findings appear counter-intuitive at first.
- o You can use profiling to test hypotheses too, but if you limit yourself to hypothesis testing only, you will miss the chance to discover other interesting things you did not even think of when formulating your hypotheses.
- o If you want to test how the system responds to one parameter, you should vary only that parameter while keeping other parameters fixed.
- o Document your findings. Clearly state under which circumstances and using which tools the profiling was done. Include a short summary of the findings. Dumping a lot of data is not very helpful by itself.
- o Show evidence of your findings/conclusions. Including a screenshot adds to the credibility (i.e. it proves that you did not simply make up the numbers).
- **Prove your claims.** Provide results of performance testing to show how efficient your system is. If your system should be able to handle a million data items, provide evidence of how well it handles a million data items.
- **Benchmark.** You can also consider benchmarking your system against other existing alternatives. You can also benchmark against previous versions of your own system to show how you improved the efficiency of your system over time.

2.3.5. Security

Security is a prime concern for most systems, especially for multi-user systems with open access (such as Internet-based applications). If security is a top priority for your system, make sure you have it built into the system from the very beginning. It is much harder to secure a system built based on an unsecure architecture.

2.3.6. Reusability

Under this aspect, you can address the following questions:

- How well have reused code in the part you wrote? For example, did you simply copy-paste modify functions instead of writing generic reusable functions?
- How well did reuse code across different parts of your system (written by different team members)?
- How reusable is code in other similar systems or future versions of this system?

2.3.7. Testability

Testability does not come by default; some designs are more testable than others. To give a simple example, methods that return a value are easier to test than those that do not. If you give up testing some parts of your system because it is too difficult to test, you have a testability problem. If you have good testability, it is possible to test each part of your system in isolation and with reasonable ease. During integration, a system with good testability can be tested at each stage of the integration, integrating one component at a time.

2.3.8. Scalability

How well can your system handle bigger loads (bigger input/output, more concurrent users, etc.)? Does the drop in performance (if any) reasonably reflect the increase in workload?

2.3.9. Flexibility/Extensibility

How easily can you change a certain aspect of your system? For example, can you change the database vendor or the storage medium (say, from a relational database to a file-based system) without affecting the rest of the system? Can you replace a certain algorithm in your system by replacing a component?

How easily can you extend your system (to handle other types of input, do other types of processing, etc.)? For example, can you add functionality by simply plugging in more code, or do you need extensive changes to existing code to add functionality?

2.3.10. Usability

Does the usability of your system match the target user base? Is the operation intuitive to the target user base? Have you produced good user manuals? Does your system let users accomplish things with a minimal number of steps?

2.4. Influence of Security

This topic (Security) deals with protecting systems from deliberate attacks, either internal or external, from individuals intentionally attempting to steal information, damage information, or otherwise deliberately wreak havoc in some manner.

Some of the most common types of **violations** include:

- **Breach of Confidentiality** - Theft of private or confidential information, such as credit card numbers, trade secrets, patents, secret formulas, manufacturing procedures, medical information, financial information, etc.
- **Breach of Integrity** - Unauthorized **modification** of data, which may have serious indirect consequences. For example a popular game or other program's source code could be modified to open up security holes on users systems before being released to the public.
- **Breach of Availability** - Unauthorized **destruction** of data, often just for the "fun" of causing havoc and for bragging rights. Vandalism of web sites is a common form of this violation.

2.4.1. Viruses

- A virus is a fragment of code embedded in an otherwise legitimate program, designed to replicate itself (by infecting other programs), and (eventually) wreaking havoc.
- Viruses are more likely to infect PCs than UNIX or other multi-user systems, because programs in the latter systems have limited authority to modify other programs or to access critical system structures (such as the boot block.)
- Viruses are delivered to systems in a **virus dropper**, usually some form of a Trojan Horse, and usually via e-mail or unsafe downloads.
- **Theft of Service** - Unauthorized use of resources, such as theft of CPU cycles, installation of daemons running an unauthorized file server, or tapping into the target's telephone or networking services.

- **Denial of Service, DOS** - Preventing legitimate users from using the system, often by overloading and overwhelming the system with an excess of requests for service.
- Some of the forms of viruses include:
 - **File** - A file virus attaches itself to an executable file, causing it to run the virus code first and then jump to the start of the original program. These viruses are termed **parasitic**, because they do not leave any new files on the system, and the original program is still fully functional.
 - **Boot** - A boot virus occupies the boot sector, and runs before the OS is loaded. These are also known as **memory viruses**, because in operation they reside in memory, and do not appear in the file system.
 - **Macro** - These viruses exist as a macro (script) that are run automatically by certain macro-capable programs such as MS Word or Excel. These viruses can exist in word processing documents or spreadsheet files.
 - **Source code** viruses look for source code and infect it in order to spread.
 - **Polymorphic** viruses change every time they spread - Not their underlying functionality, but just their **signature**, by which virus checkers recognize them.
 - **Encrypted** viruses travel in encrypted form to escape detection. In practice they are selfdecrypting, which then allows them to infect other files.
 - **Stealth** viruses try to avoid detection by modifying parts of the system that could be used to detect it. For example the read() system call could be modified so that if an infected file is read the infected part gets skipped and the reader would see the original unadulterated file.
 - **Tunneling** viruses attempt to avoid detection by inserting themselves into the interrupt handler chain, or into device drivers.
 - **Multipartite** viruses attack multiple parts of the system, such as files, boot sector, and memory.
 - **Armored** viruses are coded to make them hard for anti-virus researchers to decode and understand. In addition many files associated with viruses are hidden, protected, or given innocuous looking names such as "...".
- In 2004 a virus exploited three bugs in Microsoft products to infect hundreds of Windows servers (including many trusted sites) running Microsoft Internet Information Server, which in turn infected any Microsoft Internet Explorer web browser that visited any of the infected server sites. One of the back-door programs it installed was a **keystroke logger**, which records users keystrokes, including passwords and other sensitive information.
- There is some debate in the computing community as to whether a **monoculture**, in which nearly all systems run the same hardware, operating system, and applications, increases the threat of viruses and the potential for harm caused by them.

2.4.2. System and Network Threats

Most of the threats described above are termed program threats, because they attack specific programs or are carried and distributed in programs. The threats in this section attack the operating system or the network itself, or leverage those systems to launch their attacks.

2.4.3. Worms

- A **worm** is a process that uses the fork / spawn process to make copies of itself in order to wreak havoc on a system. Worms consume system resources, often blocking

out other, legitimate processes. Worms that propagate over networks can be especially problematic, as they can tie up vast amounts of network resources and bring down large-scale systems.

- One of the most well-known worms was launched by Robert Morris, a graduate student at Cornell, in November 1988. Targeting Sun and VAX computers running BSD UNIX version 4, the worm spanned the Internet in a matter of a few hours, and consumed enough resources to bring down many systems.
- This worm consisted of two parts:
 1. A small program called a **grappling hook**, which was deposited on the target system through one of three vulnerabilities, and
 2. The main worm program, which was transferred onto the target system and launched by the grappling hook program.
- The three vulnerabilities exploited by the Morris Internet worm were as follows:
 1. **rsh (remote shell)** is a utility that was in common use at that time for accessing remote systems without having to provide a password. If a user had an account on two different computers (with the same account name on both systems), then the system could be configured to allow that user to remotely connect from one system to the other without having to provide a password. Many systems were configured so that **any** user (except root) on system A could access the same account on system B without providing a password.
 2. **finger** is a utility that allows one to remotely query a user database, to find the true name and other information for a given account name on a given system. For example, "finger joeUser@somemachine.edu" would access the finger daemon at somemachine.edu and return information regarding joeUser. Unfortunately, the finger daemon (which ran with system privileges) had the buffer overflow problem, so by sending a special 536character user name the worm was able to fork a shell on the remote system running with root privileges.
 3. **sendmail** is a routine for sending and forwarding mail that also included a debugging option for verifying and testing the system. The debug feature was convenient for administrators, and was often left turned on. The Morris worm exploited the debugger to mail and execute a copy of the grappling hook program on the remote system.

2.5. Structuring Methods

For efficient performance and implementation an OS should be partitioned into separate subsystems, each with carefully defined tasks, inputs, outputs, and performance characteristics. These subsystems can then be arranged in various architectural configurations:

2.5.1. Simple Structure

When DOS was originally written its developers had no idea how big and important it would eventually become. It was written by a few programmers in a relatively short amount of time, without the benefit of modern software engineering techniques, and then gradually grew over time to exceed its original expectations. It does not break the system into subsystems, and

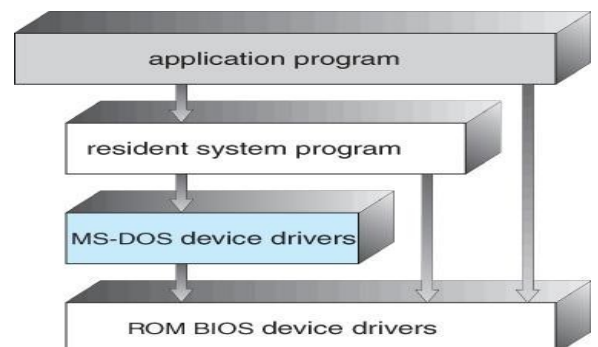


Figure 1 MS-DOS layer structure

has no distinction between user and kernel modes, allowing all programs direct access to the underlying hardware. (Note that user versus kernel mode was not supported by the 8088-chip set anyway, so that really wasn't an option back then)

The original UNIX OS used a simple layered approach, but almost all the OS was in one big layer, not really breaking the OS down into layered subsystems:

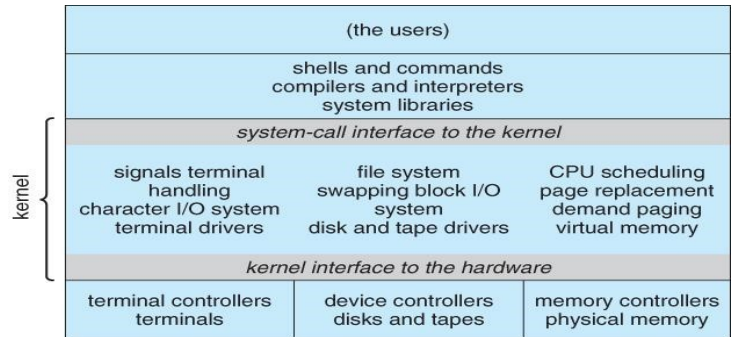


Figure 2 Traditional UNIX system structure

2.5.2. Layered Approach

- Another approach is to break the OS into a number of smaller layers, each of which rests on the layer below it, and relies solely on the services provided by the next lower layer.
- This approach allows each layer to be developed and debugged independently, with the assumption that all lower layers have already been debugged and are trusted to deliver proper services.
- The problem is deciding what order in which to place the layers, as no layer can call upon the services of any higher layer, and so many chicken-and-egg situations may arise.
- Layered approaches can also be less efficient, as a request for service from a higher layer has to filter through all lower layers before it reaches the HW, possibly with significant processing at each step.

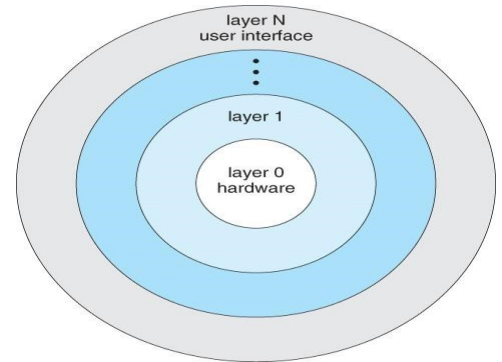


Figure 3 A layered operating system

2.5.3. Microkernels

- ✓ The basic idea behind micro kernels is to remove all non-essential services from the kernel, and implement them as system applications instead, thereby making the kernel as small and efficient as possible.
- ✓ Most microkernels provide basic process and memory management, and message passing between other services, and not much more.
- ✓ Security and protection can be enhanced, as most services are performed in user mode, not kernel mode.
- ✓ System expansion can also be easier, because it only involves adding more system applications, not rebuilding a new kernel.
- ✓ Mach was the first and most widely known microkernel, and now forms a major component of Mac OSX.

- ✓ Windows NT was originally microkernel, but suffered from performance problems relative to Windows 95. NT 4.0 improved performance by moving more services into the kernel, and now XP is back to being more monolithic.
- ✓ Another microkernel example is QNX, a real-time OS for embedded systems.

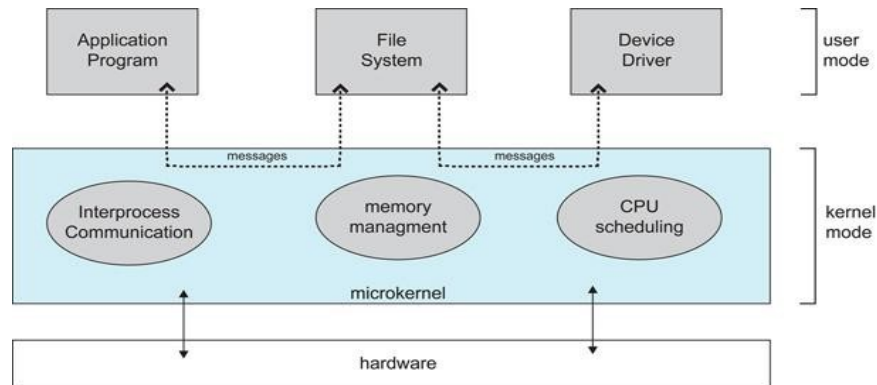


Figure 4 Architecture of a typical microkernel

2.5.4. Modules

- Modern OS development is object-oriented, with a relatively small core kernel and a set of **modules** which can be linked in dynamically. See for example the Solaris structure, as shown in Figure 2.3 below.
- Modules are similar to layers in that each subsystem has clearly defined tasks and interfaces, but any module is free to contact any other module, eliminating the problems of going through multiple intermediary layers, as well as the chicken-and-egg problems.
- The kernel is relatively small in this architecture, similar to microkernels, but the kernel does not have to implement message passing since modules are free to

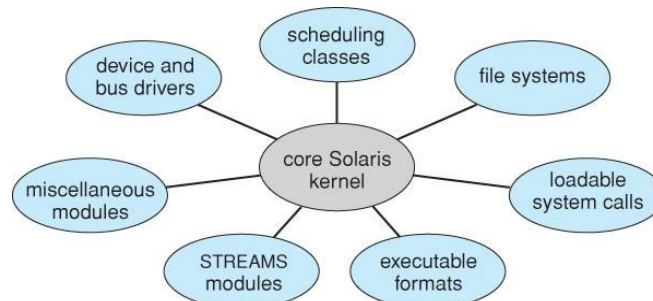


Figure 5 Solaris loadable modules

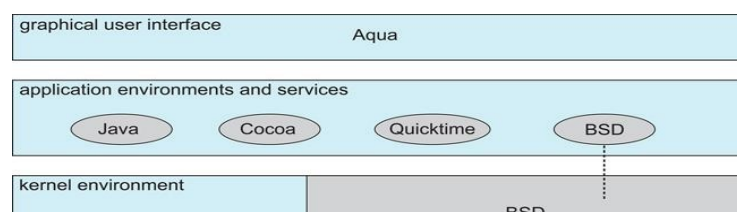
contact each other directly.

2.5.5. Hybrid Systems

- ✓ Most Operating systems today do not strictly adhere to one architecture, but are hybrids of several.

A) Mac OS X

- ✓ The Mac OS X architecture relies on the Mach microkernel for basic system management services, and the BSD kernel for additional services. Application



services and dynamically loadable modules (kernel extensions) provide the rest of the OS functionality:

B) iOS

- ✓ The iOS operating system was developed by Apple for iPhones and iPads. It runs with less memory and computing power needs than Max OS X, and supports

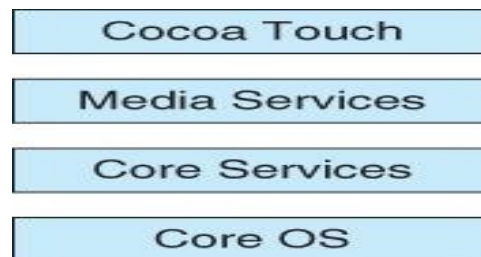


Figure 7 Architecture of Apple's iOS

touchscreen interface and graphics for small screens:

C) 2.4.5.3 Android

- ✓ The Android OS was developed for Android smartphones and tablets by the Open Handset Alliance, primarily Google.
- ✓ Android is an open-source OS, as opposed to iOS, which has led to its popularity.
- ✓ Android includes versions of Linux and a Java virtual machine both optimized for small platforms.
- ✓ Android apps are developed using a special Java-for-Android development

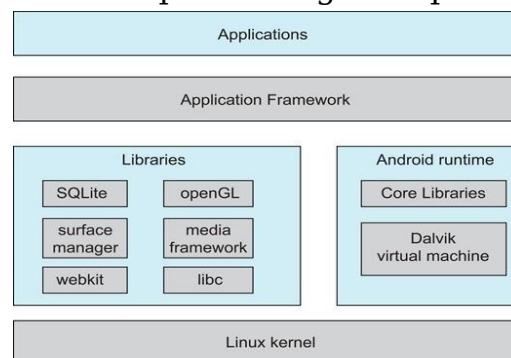


Figure 8 Architecture of Google's Android

environment.

2.6. Concept of Application Program Interfaces

You often have to rely on others to perform functions that you may not be able or permitted to do by yourself, such as opening a bank safety deposit box. Similarly, virtually all software has to request other software to do some things for it.

To accomplish this, the asking program uses a set of standardized requests, called **application programming interfaces (API)**, that have been defined for the program being called upon. Almost every application depends on the APIs of the underlying operating system to perform such basic functions as accessing the file system. In essence, a program's API defines the proper way for a developer to request services from that program.

2.6.1. Types of APIs

There are many different types of APIs for operating systems, applications or for websites. Windows, for example, has many API sets that are used by system hardware and applications when you copy and paste text from one application to another, it is the API that allows that to work.

Most operating environments, such as MS-Windows, provide an API so that programmers can write applications consistent with the operating environment. Today, APIs are also specified by websites. For example, Amazon or eBay APIs allow developers to use the existing retail infrastructure to create specialized web stores. Third-party software developers also use Web APIs to create software solutions for end-users.

2.6.2. Popular API Examples

Programmable Web, a site that tracks more than 9,000 APIs, lists Google Maps, Twitter, YouTube, Flickr and Amazon Product Advertising as the most popular APIs.

1. **Google Maps API:** Google Maps APIs lets developers embed Google Maps on webpages using a JavaScript or Flash interface. The Google Maps API is designed to work on mobile devices and desktop browsers.
2. **YouTube APIs:** YouTube API: Google's APIs lets developers integrate YouTube videos and functionality into websites or applications. YouTube APIs include the YouTube Analytics API, YouTube Data API, YouTube Live Streaming API, YouTube Player APIs and others.
3. **Flicker API:** The Flickr API is used by developers to access the Flickr photo sharing community data. The Flickr API consists of a set of callable methods, and some API endpoints.
4. **Twitter APIs:** Twitter offers two APIs. The REST API allows developers to access core Twitter data and the Search API provides methods for developers to interact with Twitter Search and trends data.

2.7. Interrupt

An interrupt is a signal from a device attached to a computer or from a program within the computer that causes the main program that operates the computer (the operating system) to stop and figure out what to do next. Almost all personal (larger) computers today are *interruptdriven* - that is, they start down the list of computer instructions in one program (perhaps an application such as a word processor) and keep running the instructions until either (A) they can't go any further or (B) an interrupt signal is sensed. After the interrupt signal is sensed, the computer either resumes running the program it was running or begins running another program.

Basically, a single computer can perform only one computer instruction at a time. But, because it can be interrupted, it can take turns in which programs or sets of instructions that it performs. This is known as multitasking. It allows the user to do a number of different things at the same time.

There are three types of interrupts:

1. **Hardware Interrupts** are generated by hardware devices to signal that they need some attention from the OS. They may have just received some data (e.g., keystrokes on the keyboard or an data on the ethernet card); or they have just completed a task which the operating system previous requested, such as transferring data between the hard drive and memory.
2. **Software Interrupts** are generated by programs when they want to request a system call to be performed by the operating system.
3. **Traps** are generated by the CPU itself to indicate that some error or condition occurred for which assistance from the operating system is needed.

Interrupts are important because they give the user better control over the computer. Without interrupts, a user may have to wait for a given application to have a higher priority over the CPU to be ran. This ensures that the CPU will deal with the process immediately.

2.7.1. CPU Execution Mode

There are two modes of execution, known as user mode and kernel or supervisor mode. User mode is restricted in that certain instructions cannot be executed, certain registers cannot be accessed, and I/O devices cannot be accessed. Kernel mode has none of these restrictions. A system call will set the CPU to kernel mode, as will traps and interrupts. Application programs cannot do this.

Mode bit: **Supervisor** or **User** mode

- *Supervisor mode*
 - Can execute all machine instructions
 - Can reference all memory locations
- *User mode*
 - Can only execute a subset of instructions
 - Can only reference a subset of memory locations

2.7.2. CPU Response to Interrupts

A key point towards understanding how operating systems work is to understand what the CPU does when an interrupt occurs. The hardware of the CPU does the exact same thing for each interrupt, which is what allows operating systems to take control away from the current running user process. The switching of running processes to execute code from the OS kernel is called a context switch.

CPUs rely on the data contained in a couple registers to correctly handle interrupts. One register holds a pointer to the process control block of the current running process. This register is set each time a process is loaded into memory. The other register holds a pointer to a table containing pointers to the instructions in the OS kernel for interrupt handlers and system calls. The value in this register and contents of the table are set when the operating system is initialized at boot time.

The CPU performs the following actions in response to an interrupt:

1. Using the pointer to the current process control block, the state and all register values for the process are saved for use when the process is later restarted.
2. The CPU mode bit is switched to *supervisory* mode.
3. Using the pointer to the interrupt handler table and the interrupt vector, the location of the kernel code to execute is determined. The interrupt vector is the IRQ for hardware interrupts (read from an interrupt controller register) and an argument to the interrupt assembly language instruction for software interrupts.
4. Processing is switched to the appropriate portion of the kernel.