1 What is an Operating System?

A search query goes to a DNS server, then goes through the Internet to its address. Along the way, a load balancer that uses algorithms to manage many machines that give partial responses that are put back together and sent back. Every single device involved in this process has an operating system.
Operating System: special layer of software that provides application software access to hardware resources. These resources include processors, screen, etc. An application may want to access many of these resources - you must provide some level of abstraction to application developers. The OS "visualizes the hardware." The OS does the following:

- Manage Resources: generally more than 1 application running on a device - protect against overload (Referee)
- Provide Abstraction: Different iPhones can run the same software from abstraction (Illusionist)
- Implements specific algorithms and techniques: scheduling, concurrency, transactions, security, etc. (Glue)

1.1 Examples of Operating Systems Design

Process: Some allocation of resources on a machine. Typical process: load the software into memory using OS, then begin using the processor. When you click on the screen, another process begins in the processor (**Context Switch**) when it finishes the new process, the OS goes back to the old application. More and more processes allocate different memory and the OS must manage these and protect them from one another in memory.

Protection Boundary: prevents the hardware from messing with the OS. Sometimes, however, devices will directly access memory without going to the processor so as to not put more load on it (**direct memory access**)

1.2 What Makes OS So Exciting

Moore's Law and microprocessors have increased in power and efficiency. All this computing power is also becoming cheaper. However, to make chips faster, we increase the frequency they are running at, but this increases power and pushes to physical limitations. The answer is to have more cores: ManyCore Chips. Parallelism must be exploited at all levels. Storage capacity is still growing incredibly fast (exponential). Network capacity is also growing exponentially. Where you have a processor, you have an OS.

1.3 Challenge of Complexity

Applications consist of a variety of software modules that urn on a variety of devices that implement different hardware architectures, run competing appli-

cations, fail in unexpected ways, can be under a variety of attacks. More and more cores also means these problems multiply. It isn't feasible to test software for all possible environments and combinations of components and devices - the question is how serious are the bugs, not if there are bugs.

Taming this complexity is complicated: every piece of computer hardware different - the OS insulates the application from the hardware and provides the abstraction to make this possible and easy to program

1.4 Virtual Machine

Software emulation of an abstract machine that give programs illusion they ow the machine. Make it look like hardware has features you want. This allows for programming simplicity - each process thinks it has all memory/CPU time/devices. Different devices appear to have same high level interface. Device interfaces more powerful than raw hardware.
Process VM supports execution of a single program; functionality type provided by OS.
System VM supports execution of entire OS and its applications

1.5 A Very Brief History of OS

Hardware used to be expensive and humans cheap, and slowly flipped to where hardware is very cheap now and humans are much more expensive. You want to maximize the efficiency of the programmer now.
Rapid change in hardware leads to a change in OS (batch to multiprogammig to timesharing to graphical UI to ubiquitous design) and migration of features into smaller machines

Most OS today have a long lineage, not developed from scratch

2 Four Fundamental OS Concepts

Loading: foo.c is compiled to make a.out, then you load into memory and execute. The memory from top to bottom is OS, stack, heap, data, instructions (code section), and the OS at biggest address, instructions at lowest address. The program counter (PC) pointing to the first memory in the instructions to be executed. Then "transfer control to program" and begin executing while protecting OS and program.

Execution: When you execute, you fetch instruction at PC, decode, execute, write results to registers/memory and update PC

2.1 Thread

Certain registers hold the context of thread. Stack pointer holds address of the top of the stack and one for heap and data etc. May be defined by instruction set architecture or compiler conventions
Thread: Single unique execution context (defined by PC, registers, etc). Only executing when its state is resident in the processor registers. PC register holds the address of executing instruction in the thread

2.2 Address Space with Translation

Address Space: set of accessible addresses and state associtated with them - for 32 bit process there are 2 raised to the 32 which is 4 billion addresses
When you read or write to an address many things can happen: nothing, acts like regularly memory, ignores writes, causes I/O operations (memory-mapped I/O), causes exception (fault)

2.3 Process

Process: execution environment with Restricted Rights - address space with one or more threads -owns memory (Address Space) - owns file descriptors, file system context - encapsulates one or more threads sharing process resources. Process are protected from each other and OS from them.

Fundamental tradeoff between protection and efficiency - communication easier within a process but harder between processes

2.4 Dual Mode Operation/Protection

"Kernel" mode: supervisor/protected
"User" mode:normal programs executed

You need one bit to determine which mode you are in. Certain operations and actions only permitted in kernel mode. Switching from user to kernel sets system mode AND saves the user PC (uPC). OS carefully puts aside user state then performs the necessary operation.
When program is given control and done you are in user mode and then you switch back to the kernel mode (kernel mode default) - interrupts and syscalls can take you from the user mode to the kernel mode.

Multiplex in time: if you have 10 processes, give 100 ms to each per second. This gives the appearance of multiple processes. Each virtual CPU has a PC and SP that you must save and load when you switch. To trigger the switch, use a timer, voluntary yield, I/O, or other things.

Problem of Concurency: if you need a resource (like a keyboard) it's hard

to multiplex - something hogs the resource (typically bugs). The OS provides the illusion you have all the resources but you really only have a fraction.

Properties of simple multigrpgramming technique: all virtual CPUs share same non-CPU resource, consequence of sharing is each thread can access the data of every other thread (good for sharing bad for protection) and threads can share instructions. The unprotected model is common in early versions on Windows (relied on process being well-behaved) **Protection**: system must protect itself from other programs and protects user from each other

Threads encapsulate concurrency ("active" component) and address space encapsulates protection ("passive" part) that keeps buggy programs from crashing the system

2.5 Multiplexing in Space the Memory

Multiplexing in time the memory is too slow.
To multiplex in space you must avoid processes overwriting each others memory - you must define regions of memory beginning with a base address and ending at a bound address (Base and Bound - B and B). This requires relocating the loader and still protects OS and isolates program - no addition on address path (fast)

This kind of translation is difficult, so we visualize the address - each program sees the same address space - but in the physical memory on the machine they are not all in the same place - you translate the addresses on the fly (**Virtual Address**) - the program still can't touch the OS