Contents

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

2.6 3 types of Mode Transfer

How the OS gives control to a Process: Base and Bound change to that of the yellow process, but when the OS needs to get back the control you must keep track of where to return to (RTU - Return From User). The uPC (user program counter) becomes the next address to be executed in the process. Sysmode changes from 1 to 0 (restrict access of the program to privileged instructions)

- Syscall: process requires a system service e.g. exit. This is like a function call but outside the process - doesn't have the address of the system function to call
- Interrupt: external asynchronous event triggers context switch - often I/O - independent of user process
- Trap or Exception: internal synchronous event in process triggers context switch - protection violation like segfault/divide by 0

Interrupt Vector: for each interrupt you have a number, and you have a vector of interrupts with addresses of code to execute to handle that given interrupt.

When we return from a process (if say another process interrupts it), we use the RTU and store necessary data for the process in the Static data. The PC contains InterruptVector[i] corresponding to the interrupting process. Then you restart the method for the next process by setting the base, bound, uPC, and regs

Fragmentation: Kernel has to fit processes into contiguous memory blocks but there are random empty chunks from finished processes
Sharing: Very hard to share between processes or process and kernel

One potential solution is to use multiple segments for code, static data, heap, and stack. But this still doesn't allow for enough sharing.
Virtual Address Translation: Pages of fixed sizes fit very nicely into physical memory and each process has four sections mapped to real memory somewhere such that everything first nicely next to each other with standard page size

2.7 Process Control Block

Kernel represents each process as a PCB - status, register state, process ID, user, priority, execution time, memory space.
Kernel Scheduler: keeps track of the PCBs
Scheduler: mechanism for deciding which processes/threads to receive the CPU - different policies can use fairness or realtime guarantees or latency optimization etc.

The kernel and the processes must have separate stacks, as it needs its own stack and safe space to work
Kernel System Call Handler: vector though well-defined syscall entry points - locate arguments from user stack to kernel, copy arguments into kernel memory, validate arguments and copy results back into user memory

2.8 Interrupt Control

Interrupt processing not visible to user process - occurs between instructions - no change to process state (save state). You can observe the impact of the interrupt by things slowing down. If you 'disable' the interrupts it will run until completion and they are re-enabled upon completion. **Non-Maskable-Interrupt**: can't ignore these, like segfaults
Interrupt Controller chooses interrupt request to honor, masks enables/disables interrupts, priority encoder picks highest enabled interrupt. The CPU can disable interrupts.
Taking interrupts safely: there is an interrupt vector. Kernel handles interrupt regardless of state of user code - the handler is non-blocking - atomic transfer of control (nothing can mess the process up in the middle) - transparent restartable execution

2.9 Fork

Unique identity of process is the process ID (PID). Fork() system call creates a copy of current process with a new PID - parent and child process.
Return value from Fork(): integer greater than 0 in original (parent) process, 0 when in new child process, less than 0 means error

Other UNIX commands:
exec: change the program being run by the current process
wait: the parent uses to wait for the child to finish
signal: send notification to another process

2.10 System Call Interface

One type is read:
count = read(fd, buffer, nbytes)
First thing you do is put the arguments on the stack (3rd then 2nd then 1st), then the read system call number into a register then a trap to give control to OS
Trap is a synchronous interrupt to give control to the kernel
When the OS gets control it uses the syscall number, looks at the corresponding syscall handler table index and gets the syscall handler code, after this you Return to caller and go back to user space

3 Key Unix I/O Design Concepts

Uniformity: open, read/write, close - file operations, device I/O, and inter-process communication happen this way which allows simple composition of programs

- Open provides opportunity for access control and arbitration as well as sets up underlying machinery (data structures)
- Byte-oriented: even if blocks are transferred, addressing is in bytes
- Kernel buffered reads and writes- streaming and block devices look the same - read blocks process, yielding processor to other task and completion of out-going transfer decoupled from the application, allowing it to continue - flush writes all buffered information

3.1 The File System Abstraction

File: named collection of data in a file system - the data can be text/binary/linearized objections as well as metadata
Directory: Folder containing files and directories with hierarchal naming
Steams: sequence of bytes, whether text or data

3.2 Low Level I/O

open, create and close operate on file descriptors - an index to a data structure that contains all of the file information
Permissions are User Group Other separated by — like R—W—X (vertical lines)

3.3 SYSCALL

Low level lib parameters are set in registers
Internal OS File Descriptors: internal data structure describing everything about the file

3.4 Device Driver

Implemented in the kernel: the code that interfaces with the device - programs the device and receives interrupts from the device
They are typically divided into two halfs - the first provides common API to OS and the bottom half gets the interrupts from the devices

4 Communication Between Processes

Client issues write request and then waits while the server performs the read operation and then services the request.

4.1 Client-Server Model

Servers can have multiple clients (e.g. Webserver)
Socket: an abstraction of a network I/O queue - mechanism for inter-process communication that embodies one side of a communication channel - over any kind of network
File systems provide a collection of permanent objects in a structured name space (independent of process)
Sockets provide a means for processes to communicate (transfer data) to other processes

On the server side, create a new socket for every request, bind to TCP, listen and accept (when you accept you create a new socket - sereer connection socket)
On the client side, create socket, bind to TCP, connect on socket
Connection has five things: source and destination, two port numbers, and protocol type

4.2 Namespaces for Communication Over IP

Just an IP address for the machine isn't enough - we must specify each process from one another - this is where port numbers come in - always 80 for web

4.3 Multiplexing

The current state of process held in a PCB - the CPU decides what time to different processes and OS protects processes from each other
Context Switch: when the CPU switches form one process to another - can be non-trivial.
There are queues for each state of process (read queue, I/O queue) - PCBs switch between queues as they change state

4.4 Threads

Multithreading: a single program made up of a number of different concurring activities - done because its easy to program and save less state
Code, heap, global variables shared - registers and stack not shared
How to prevent stacks from writing on each other or on the heap??

4.5 ThreadFork

user level procedure that creates a new thread and places it on the ready queue - takes in pointer to routine, arguments, and size of stack - it initializes a TCB and Stack
ThreadRoot is the root for the thread routine - creates and runs the thread - ThreadFinish() then wakes the sleeping threads

4.6 Pintos

1 Thread per process only!

4.7 Pintos Interrupt Processing

Pushes interrupt number onto stack and then the general interrupt entry function
Timer triggers thread switch then thread tick to update thread counters then thread yield to set current thread back to ready then scheduler

4.8 Reasons for Multithreading

Sharing resources, speedup, and modularity

4.9 Atomic Operations

If two threads read the account balance, then both add their amount sepasrately, one will be lost - we have to make the read add and store happen without being interrupted or having the value changed
Atomic Operations run to completion or not at all = fundamental building block

4.10 Correctness Requirement

Threaded programs must work for all interleaving of thread instruction sequences

4.11 Definitions

Synchronization: using atomic operations to ensure cooperation between threads **Critical Section**: only one thread can execute at a time **Lock**: prevent someone from doing something

5 Locks, Semaphores, and Monitors

Where are we going with synchronization? In order to have high level API like locks and semaphores, we must have hardware support

5.1 Locks

Use when you want to prevent anyone else from executing the code you are executing - only one thread can acquire this lock at a time. Instead of busy waiting, you can go to sleep and be woken up at the correct time
How do we build multi-instruction atomic operations - you don't want to be interrupted or yield - these interrupts could be internal or external, you want to avoid context switching.

The **naive implementation** will simply disable interrupts on lock acquire and enable them on lock release. But the critical section could be while(true)...

A better implementation is to pick a lock variable with a value that will denote FREE or BUSY and put threads on the waiting queue if value is BUSY when you acquire. When you release you set value to be FREE. Additionally disable interrupts during the actual critical section in acquire and release.

Before you put a thread to sleep, you must enable the interrupts - you want to do this atomically with going to sleep so we need hardware support

Atomic Instruction Sequences: these instructions read a value and write a new value automatically - implemented in hardware

5.2 Semaphores

Easier to program than locks, but they have weaknesses

P() - wait for semaphore to become positive, then decrements
V() - increments semaphore by 1, waking up a waiting P if any

Use 1: mutual exlusion (same reason as using a lock)
Use 2: scheduling constraints - wait thread to finish before threadjoin

5.3 Monitors

They use locks for mutual exclusion and conditional variables for schedule constraints.
Always acquire and release lock when dealing with the critical section/shared variable

Hoare: when you signal, you give up the lock and CPU to the other thread, then the other thread gives it back when it's done
Mesa: when you signal, waiter is placed on the ready queue and the signaler keeps lock and CPU