Operating System Design

Chp 12, Tanenbaum

Good systems (not just operating) have clear goals. Tanenbaum notes for general purpose operating systems:

1. define abstractions
2. provide primitive operations
3. ensure isolation
4. manage the hardware

Why OS design is hard:

1. extremely large programs (Windows 2000 with 29 million lines of code)
2. must deal with concurrency
3. potential hostile users—protection
4. want to be able to share
5. last a long time, don't die off so easily
6. designers don't have a clear idea how systems will be used
7. portability to run with lots of hardware platforms
8. need for backward compatibility

Interface Design

Principle 1: simplicity

"Perfection is reached not when there is no longer anything to add, but when there is no longer anything to take away." (Antoine de St. Exupery)

Principle 2: Completeness

OS should do what is needed of it, but no more.

—minimum of mechanism and maximum of clarity

Tanenbaum cites Minix and Amoeba as have just send/receive primitives (but then all the hard work is somewhere else!).

Principle 3: Efficiency

costs of system calls should be intuitive

Paradigms

What model do users of the system see:

architectural coherence—do features hang together?

user interface paradigms—need consistency among applications

execution paradigms—algorithmic vs. event-driven code (threads vs. event-driven)

data paradigm—everything is a:
magnetic tape—FORTRAN
file—Unix
object—Windows
document—Web

System Call Interface

minimize system calls, easier if a unifying data paradigm

Tanenbaum's first law of software—"Adding more code adds more bugs."

Lamport: "Don't hide power"

Implementation

System structure:
layered
exokernels (push OS functions to be libraries for applications)
client-server
Mechanism vs. policy (as discussed in Intro). Implement mechanism such that the policy can be flexible.

Orthogonality—ability to combine separate concepts (fork and exec of Unix)

Naming—generally a high-level user visible name and a low-level (ugly) system name.

Binding Time—early vs. late. When is a decision made? Efficiency vs. flexibility.

Static vs. Dynamic structures—declare at compile time or allocate at run time. Again efficiency vs. flexibility.

Useful Techniques

hide hardware details: for example convert interrupts into thread invocations

conditional compilation for hardware details

use indirection to solve problems—more flexibility, less efficiency.

resuability—software engineering mantra

check return codes of system calls—they may fail!

Performance

more OS features, more time—self-inflicted performance hit

optimize where needed and until "good enough"—understand the performance problem and then expend just enough resources to solve it.

space-time tradeoffs. Classic software problem.

Caching is good. Allows systems to work.

Hints a similar idea of improving performance.

exploit locality

optimize the common case

Project Management

Testing is the hard part and simply determining "man-months" for a project does not work as not all of the work can be done in parallel.

Trends in Operating Systems Design

large address space OSes

networking—fundamental

multimedia

battery-power (power is another resource to be managed)

embedded systems