CS 736
Spring 2016

Lecture 2: OS structure and Abstractions

Papers:

- THE
- Nucleus
- Unix

Opening questions:

- What were organizing principles of the three systems?
- What were the central abstractions of the three systems?

Context for these papers:

a. Context: computers up to this point programmed at low level:
    i. Assembly language (PL/1 coming along)
    ii. Not much agreement about abstractions
    iii. Not much rigor/correctness given to design
b. Context: complexity starting to get serious:
    i. OS/360 was being designed – huge effort, thousands of programmers, late, buggy, not rigorously designed (e.g. corner cases involving interrupts very sloppy)
c. OS had not really settled down
    i. What services should it offer?
    ii. What applications is it for?
    iii. How should it be constructed internally?
        1. As a bunch of libraries?
        2. As a bunch of a layers
        3. As a hierarchy?
        4. As modules/subsystems?
d. Problems being solved:
    i. What is right way to organize OS to provide
        1. Protection
        2. Flexibility
        3. Simplicity / correctness
        4. Handle I/O efficiently (abstracted from processes)
    ii. How do you battle the complexity of:
        1. Multiprogramming – e.g. different users, different tasks, different programs, different priorities
        2. Interrupts; re-entrant code
        3. Control: who controls things and how
        4. Flexibility: not much is known about how to do things, want to have flexibility to change things in the future

e. THE - designed in 1965, while work IBM's OS-360 was being finished

1. **Topics of the day: Abstraction and Organization**
    a. QUESTION: What is abstraction?
        i. The operating system provides **a layer of abstraction** between the user and the bare machine. Users and applications do not see the hardware directly, but view it through the operating system.
        ii. This abstraction can be used to **hide certain hardware details** from users and applications. Thus, changes in the hardware are not seen by the user (even though the OS must accommodate them).
        iii. This is particularly advantageous for venders that want offer **a consistent OS interface** across an entire line of hardware platforms.
        iv. Another way that abstraction can be used is to **make related devices appear the same** from the user point of view. For example, hard disks, floppy disks, CD-ROMs, and even tape are all very different media, but in many operating systems they appear the same to the user.
    b. QUESTION:
        i. What are some common OS abstractions?
        ii. What hardware (or sets of hardware) do they abstract?
    c. QUESTION: where do abstractions exist?
        i. Within the operating system (e.g., as layers in THE)
        ii. At the operating system API (system calls), e.g. files
        iii. Between operating systems and applications: runtime layers (C's struct FILE)
2. THE – structure and abstractions
    a. Background:
        i. Batch system,
            1. 5 user processes, 10 I/O processes
            2. Spooling system: read from paper tape, printer for output
            3.
        ii. Old problem: I/O timing
            1. Many devices had specific timing
                a. How long does a disk/drum take to rotate?
                b. How long does a printer take to print one character?
            2. Interactions with devices would use real time to decide when to input or output
                a. **QUESTION: What is the problem with this?**
                b.
    b. Organizing principle: **layered abstractions**
        i. Each abstraction builds upon the layers below it
        ii. Abstractions at each layer are visible to the layers above
        iii. Like **nested virtual machines** gradually adding more features
    c. Abstractions:
        i. Sequential processes: the CPU

            1. Executes a sequence of instructions sequentially, but not contiguously
            2. Hides effects of interrupts, timing
            3. Provides explicit synchronization routines (semaphores)
            4. QUESTION: What is not allowed?
                a. Data doesn't disappear if you don't read it in time
                b. No synchronization that is not explicit via semaphores
                c. I/O devices use buffered input/output to avoid revealing timing to user processes
        ii. Memory segments: virtual addresses
            1. Abstracts physical location (memory or disk), which are both independent
            2. Can switch to another process while transferring data
        iii. Virtual console
            1. Attaches keyboard to the correct process
        iv. Virtual devices
            1. Buffering of output to avoid dependence on real time
            2. Buffering of input to avoid dependence on real time
        v. User programs
            1. Can use all services in all layers below
    d. System organization: layers
        i. QUESTION: Why layers?
            1. Solves the question of dependencies:
                a. What code is allowed to use what othe code?
                b. Need to avoid circular dependencies
            2. Addresses testing:
                a. Can test an abstraction and no code above it is supposed to affect the abstraction
                b. Reduces M x M tests for component interactions to M tests (one at each layer)
            3. Allows code to depend on the abstraction, not the implementation
                a. Can vary (e.g. if disk breaks, do things in memory)
        ii. QUESTION: what are the drawbacks?
            1. Rigid: hard to add new features, like networking
            2. Protection: no protection in this system, expensive to add protection at each layer
            3. Performance: hard to perform cross-layer optimizations, such as transmit data from tape directly to disk

3. Nucleus – structure and abstractions
    a. Background:
        i. Frustration with multiple operating systems for different purposes
        ii. QUESTION: Why would you do this?
            1. Scheduling types (real time, batch, time sharing)

b. QUESTION: What are possible solutions to this problem?
    i. Defining better abstractions?
    ii. Building systems that can be customized
        1. Modify the code
        2. Add extensions
        3. Parameterized – e.g. scheduling classes/priorities
    iii. QUESTION: can you build the base of an operating system that meets all needs?
        1. Are some things incompatible?
            a. Example: real time systems cannot really do page faults, as they introduce variability
            b. Real time systems typically have hard guarantees on the number of processes, when they run.
                i. Mismatch for systems like Unix that create many processes for short tasks
c. Organizing principle:
    i. A nucleus with primitive operations
        1. **QUESTION: What kinds of operations go in the nucleus?**
            a. Privileged operations
                i. Loading memory maps
                ii. Context switching
                iii. communication
            b. Hardware-dependent operations
                i. I/O
    ii. Extends hardware to provide a virtual machine with:
        1. Interrupt response
        2. Control of I/O
        3. Storage protection
        4. Processes
        5. Operations to create, control, and communicate between processes
        6. Mediates communication
d. Abstractions:
    i. QUESTION: What are Execution abstractions:
        1. Program: bits of code and data in storage
        2. Internal Processes:
            a. execution of a program (same as today)
            b. Has a global name
            c. Has memory (storage) inherited from parent
    ii. QUESTION: What are I/O abstractions
        1. Peripheral device:
            a. Device with an ID and a data channel
        2. Document

        a. Collection of data stored on a medium, like a block page, segment or file
    3. External process: a process controlling a device
        a. Launches when I/O requested; may not be persistent
        b. Runs as part of nucleus, but presents interface of a process
    4. QUESTION: What does this accomplish?
        a. Programs can refer to documents, without having to know how to store/retrieve
        b. External processes are launched to do the storing/retrieving
        c. Uses same communication mechanisms as between normal processes for I/O

iii. QUESTION: Why is communication so important an abstraction?
    1. Allows moving code out of a process, out of the OS
        a. Can communicate with another process for functionality

iv. Communication Abstractions
    1. Message queue + message buffers
        a. Mailbox for each process that can receive messages
    2. Operations:
        a. Send message / wait message
        b. Send answer / wait answer
        c. QUESTION: why asymmetric?
            i. Answers reuse buffering of message
            ii. Guarantees responses don't block waiting for space
        d. QUESTION: Why blocking on wait(), non-blocking on send()?
            i. Efficient: allows overlapping execution
            ii. Properties:
                1. Fault tolerant: failure of a process causes a dummy response or answer to be ignored
                2. Decoupled: processes don't interact directly through memory (as in THE)
                3. Asynchronous: buffering sends
            iii. Benefits:
                1. Can replace a process with another that responds to the same operations
                    a. Example: I/O; can implement in a process rather than nucleus
                2. Common interface for all I/O
                    a. Access a file: send request for file
                        i. External process reads into memory
                        ii. Answers requests by reading/writing file contents

b. Access clock:
    i. Send messages to ask clock about time, wait for things to happen (when answer is sent)

v. Process Hierarchy
    1. QUESTION: Why inherit from parent's memory?
        a. Acts like an OS that allocates memory
        b. Parent can launch a process – load code into memory, modify it if need be (e.g. for swapping)
    2. Parent can schedule child by starting/stopping a process
        a. Otherwise runs in a global scheduling queue
    3. Parent can do swapping:
        a. Stop child
        b. Write child memory out
        c. Read child memory in
        d. Start child
    4. QUESTION: How implement scheduling with this
        a. Select the set of processes you want to run (e.g. high priority), and OS will not run others
    5. QUESTION: What is not handled by hierarchy?
        a. Scheduling: all active processes are run round-robin
        b. Message delivery (see below): all processes use global names visible to other processes
    6. QUESTION: How use this to implement an OS?
        a. Could have a single process implementing THE abstractions (above layer 1) from memory of a parent process
        b. Could implement Unix system calls as messages to the parent process, which invokes nucleus

vi. QUESTION: IS this enough to replace OS?
    1. No: need to virtualize naming (having all message go to parent, which then forwards) to replace nucleus components transparently
    2. Hard to fairly manage resources at top level, as don't know how many children each process will end up having

e. QUESTION: What is evaluated?
    i. Basics:
        1. Size of implementation
        2. Speed of primitive operations / break down of time
    ii. QUESTION: is this enough?
        1. Will applications be fast or slow?
            a. Depends on how often these operations take place

        2. Depends on the hardware
            a. Are other systems faster?
            b. What are the limits in hardware for these operations?
    iii. Overall: performed terribly due to messaging overhead

4. Unix Timesharing System
    a. Context
        i. Written by software people for their own use, not to make a point
            1. Smart users, no other customers
            2. Limited hardware, budget
                a. More emphasis on something useful, working, than fully general
            3. Not trying to show how to build a better system
                a. Most research systems push ideas to an extreme
                b. Commercial systems try to predict "what other people want" but have a hard time getting it right
            4. They paid the full cost of all design choices
                a. Simpler OS but harder to program?
        ii. Not tied to a specific piece of hardware
            1. Previous systems typically written by vendor for a specific machine, in assembly language
            2. Written in C, ported many places
        iii. Many ideas in Unix came from other places
            1. Hierarchical file systems – CTSS, Multics
            2. Fork/Exec – Berkely SDS system
            3. I/O redirection – Multics
            4. Pipes – Dartmouth DTSS
        iv. Much of Unix flows from a few design decisions
            1. File system design
            2. Process management
    b. Abstractions: QUESTION: What are they?
        i. Processes: running program
            1. Code + data in an address space
            2. Unlimited number, fast to create
                a. Note: other systems allow **one process per user**
            3. No OS state in address space (all is in kernel)
            4. Standard input/output streams
            5. Abstracts processors (sequential processes) and memory (address spaces)
        ii. Files:
            1. Sequences of bytes + metadata
            2. Sequential access by default, can reposition with seek
            3. Limited set of permissions
            4. Abstracts disks
        iii. Device files

            1. Devices show up in file system as special files
            2. Access them invokes driver code rather than file system
            3. Uses same interface as files, but are a different abstraction
        iv. File systems:
            1. Unit of files stored on a device/partition
            2. Can be mounted/removed from a directory
            3. Abstracts independent disks / partitions
        v. Directories:
            1. Collection of strings pointing to files (that may also be interpreted as directories)
            2. Single pre-defined root directory
        vi. Pipes
            1. Uni-directional streams of bytes
            2. Share an interface with files, but not the same abstraction
                a. No seek
        vii. Users
            1. Identifier associated with a person who has access to things
            2. Identity attached to a process
            3. Can set via special system calls, or as a side-effect of launching a program (setuid programs)
        viii. Root user
            1. Allowed to make all system calls, access all files
    c. Abstractions have operations
        i. Processes: fork, wait, exec
        ii. File: open/close/create/read/write/seek
        iii. File systems: mount/unmount
        iv. Directories: mkdir/chdir/rmdir/link/unlink
        v. Pipes: pipe()
        vi. Users: setuid
    d. Usage style
        i. Usage of Unix is **not dictated by the system calls**
            1. It is encouraged (system calls make things easy)
        ii. Example:
            1. Many small programs connected by pipes
                a. Easy because processes are fast to create, easy to create, communication is fast, no limit on number of processes
            2. Programmable command shell
                a. Many processes, ability to redirect output
                b. Command multitasking: using fork/wait() to run things in background
                c. Shell scripts: redirecting standard input for shell to a file
            3. Extending OS via setuid() programs
                a. E.g. passwd(), login(), mount()

            i. All use setuid, then access files or make system calls only allowed to superusers to
        iii. Requires some thing to be in the kernel:
            1. Device drivers; without IPC, alternative would be to fork a program to do I/O (like nucleus external processes)
5. Big picture on abstractions:
    a. QUESTION: How do we know if abstractions are good or bad?
        i. Do they add power: make it easy to do what you want to do
        ii. Are the efficient to implement: slow abstractions (e.g. Nucleus message) encourage people to bypass
            1. Unix files: just low level bytes, no record or indexed access
            2.
        iii. Do they encourage simplicity?
            1. THE layers
        iv. Do they allow flexibility?
            1. Unix setuid() programs
            2. Nucleus messages()
        v. Are they complete?
            1. E.g. Unix link() – doesn't work across mount points