CS 736
Spring 2016

Lecture 3: Modern Abstractions

Comments on reviews:

- What are password capabilities? Can they be manipulated?
  - An address, rights, check field (for revocation), encrypted in a key the kernel knows
- Why is clean up no harder?
- Why is coarse grained control a limitation?
- Why is copy-on-write not possible?
- What happens when try to access a resource it doesn't have access to?
  - By capability – capability check fails
  - Without a capability – how do you reference the object?
  - An address: try to fault in & fail ACL check.
- How does synchronization happen?
  - Same mechanisms – spin locks, kernel locks/sempahores
- Why built on Mach?
  - Need memory management, communication, device drivers

Papers:

- ExoKernel Jihad
- Opal
- Unix

Opening questions:

- What are the central Unix abstractions
- How do these central Unix abstractions for programs into certain designs?

1. Unix Timesharing System
   a. 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)
            a. NOTE: address space provides naming (memory) + protection (set of accessible memory)

        b. No sharing because cannot name memory of another process (unless explicitly establish shared memory)
    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
b. 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
c. 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)

2. 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
        v. Are they complete?
            1. E.g. Unix link() – doesn't work across mount points

1. Opal – abstractions for big addess spaces
    a. What changes: address space is plentiful: 1.8 x 10^19 bytes (18,000 petabytes)
a. QUESTION: What are the opportunities?
    i. Can give virtual addresses to everybody on earth
    ii. can give virtual addresses to all bytes of storage you might access
    iii. can give a virtual address to all servers (that's only 32 bits)
    iv. you don't have to free virtual address space
b. QUESTION: as a researcher, what do you do?
c. ANSWER: find a problem that this makes simpler
d. QUESTION: What is the problem Opal seeks to solve:
    i. Want better structuring technique than processes
        1. Threads in a process:
            a. share full access to data,
            b. Libraries work, but hard to coordinate independent threads
            c. Not reliabile – failure of one thread kills process
        2. Separate processes
            a. Clean interfaces over IPC (pipes, messages)
            b. Expensive: slow communication, must copy/convert data
            c. Shared memory: cannot guarantee same addresses
                i. Need to pre-reserve global address for shared data, feature doesn't exist
    ii. QUESTION: how do existing abstractions not work?
        1. Only one unit of memory protection: an process
            a. Too restrictive, ties together execution (thread), addressing, protection, resource management in one entity
    iii. Solution: decompose a process into all its contitutent parts as separate abstractions:
        1. Address space -> protection domain + segments
        2. File descriptor table -> capabilities, reference objects, resources groups
        3. Thread of execution -> protected procedure call, portals
2. Opal background
    a. Benefit of AS:
        i. Increase total address space to all programs
        ii. Provide hard memory protection boundaries – can't even express name for inaccessible memory (unless go outside processor to /dev/mem or /dev/kmem)

iii. Easy cleanup on exit; just teardown entire AS
iv. Common naming system: can have different, common things at same address in different address spaces -- e.g. main() could be at fixed location

b. Drawbacks
   i. Difficult to cooperate between AS because pointers must be changed or mapping addresses must be coordinated
      1. Pointers from shared data to private data confusing
   ii. Alternatives
      1. Either use pipes & separate processes
      2. Or threads w/o protection

c. Research questions: how do you provide #2 and #3, support fine grained protection domains, simplify sharing across domains?
   i. 64 bits provide:
      1. Enough address space to not need benefit #1
      2. Kernel protection of pages provides benefit #2
      3. Cleanup no worse for non-shared programs, just as bad for shared data programs

3. Opal Approach
   a. Separate addressing and protection
      i. Addressing == translate virtual to physical
      ii. Protection == can you access the physical memory?
      iii. Unify protection into protection and not have separate name space mechanism
   b. Decompose traditional process/task:
      i. Accessable memory (e.g. address space)
      ii. References to kernel objects (FD or handles)
      iii. Threads of execution
   c. Allow dynamic protection changes to share on-the-fly or for short periods
   d. No conventional "programs"
      i. Just code segments linked together, thread starts executing in one of them
4. Opal Architecture: abstractions / mechanisms
   a. Access control
      i. Password Capabilities
         1. Need not be stored in kernel, can't be fabricated
         2. 256 bit number
         3. Pointer, rights, cryptographic validity check
            a. Encrypted by key known to kernel, with checksum to verify wasn't modified
         4. Can be passed in shared mem; need not pass through kernel

5. Must pass to kernel (or someone who knows the key) to use it

b. Single address space for all things
   i. previously had local names (file descriptors) for global objects (e.g. files, pipes)
c. Protection domains
   i. Set of accessible memory to a thread
      1. Two threads in the same domain if can access the same memory
   ii. A thread can create sub-protection domains == new domain with access to subset of parent memory
d. Memory protection
   i. Segments are unit == multiple pages
      1. No hierarchy
         a.
   ii. Reference counted so can tell when to reclaim
   iii. Can attach/detach to protection domain given capability
      1. QUESTION: Is it a problem that when one thread attaches a segment, it is accessible to other threads in the domain
      2. Answser: not really; think about processes today – one thread opens a file, other threads can use the file descriptor. One thread maps a file, other threads can access it. It is part of the definition of a domain – if you don't want it, put the thread in a different domain.
   iv. QUESTION: How do you learn about segments:
      1. Can make public and fault in with ACL check
   v. Parents have control over child domains to attach/detach segments
   vi. QUESTION: Is this enough protection? Does it make any problems worse? Can you delete memory while others are using it?
      1. Apps with shared r/w must synchronize
      2. Memory regions ref counted, can't remove while other domain uses it
   vii. QUESTION: dynamic growth.
      1. How handled? grow on fault? Pre-reserve maximum space (at low cost – just prevent others from using it)
      2. Does SASOS change things?
         a. no
         b. Same problem occurs in private process – if allocate large structure, must have space to grow it
e. Communication

i. QUESTION: what do you want? Don't need message passing - just pass addresses
    1. ANSWER: what you need is a way to invoke code + pass a parameter.
ii. QUESTION: what code?
    1. ANSWER: specified entrypoints only (e.g. system call table)
iii. Portals = entry point for transferring control
    1. Unique 64 bit number (unguessable)
    2. Specifies VA for beginning execution (like trap handler)
    3. Perform Protected Procedure call by invoking RPC with portal capability; kernel verifies and transfers control to thread in target domain
iv. Launch process by
    1. creating domain,
    2. attaching segment,
    3. Create portal
    4. Create thread
    5. PPC into portal

f. Linking – CAN SKIP
    i. Modules sit at single global address
    ii. Global data (shared) also at single address
    iii. Private data to instance (e.g. to thread / domain invoking module) stored separately, accessed indirectly through register (register indirect addressing)
    iv. COMMENT: again, a layer of indirection.
    v. COMMENT: Much of linking problem due to compiler to producing correct code for environment
    vi. QUESTION: What is the cost? Indirection overhead on procedure call
    vii. QUESTION: What problems areise with linking
        1. A: fork.
            a. QUESTION: How big a problem is this?
        2. A: private per-module data
    viii. Linker allocates global address for code
    ix. Extra tools overcome standard tools assumption about address spaces (e.g. GP register can change)
    x. All modules that reference each other (e.g. call directly with protected procedure call) combined to use single private data segment
    xi. No dynamic linking; because link statically to procedure's addrss
    xii. QUESTION: What about third-party software that is pre-linked? How do you handle it?

1. A: relink at load time (happens in Windows, called rebasing, when there is a conflict between default addresses of a DLL)
2. Happens today already with shared libraries

g. Resource management
   i. QUESTION: What is the problem?
      1. in a process, you know what to reclaim (whole address space) and when (at program exit). You know who to charge for memory.
   ii. QUESTION: in Opal, how do you know when to reclaim phyiscal memory?
      1. ANSWER: when it is no longer in use (reference counting)
   iii. Reference counting on attach/detach
      1. Can't track # of capabilities, must add layer of indirection
   iv. How handle cleanup when a domain goes away?
      1. Hierarchical **resource groups** (like handle table, FD table)
      2. All reference counts credited to a group; removing group removes corresponding references
      3. Parent can delete child resources, is charged for all usage by child (like a process tree)
      4. COMMENT: is pulling apart notion of process
   v. QUESTION: how do you prevent someone from double-decrementing it while you are still using it?
      1. ANSWER: reference object isolates your private reference counts from others, coalesces your counts into a single object
         a. Create SegRef for each trust entity
         b. SegRef only allows matching attach/detaches
      2. COMMENT: Layer of indirection to separate different domains
   vi. QUESTION: How are dangling references handled?
      1. Is a bug in user-code

5. Evaluation
   a. Classic problem: you have a new system with new capabilities. No applications use it. How can you tell if it is a good idea?
      i. A: write new apps
      ii. B: port existing apps
      iii. Validate design effort to implement these, resulting performance
   b. Design:

        i. Can choose how to structure applications; where to place domains
            1. E,g. full protection with non-overlapping domains
        ii. Use read-only sharing + PPC notification instead of pipes/RPC to reduce overhead
    c. QUESTION: What do they evaluate?
    d. QUESTION: What are the comparison platforms?
        i. Monolithic OSF/1
    e. QUESTION: Do they answer the fundamental questions?
        i. Is it simpler to share using OPAL than Unix/Mach?
        ii. Is it faster?
        iii. Is it just as safe?
    f. QUESTION: do they let you know why it performs this way? What causes the expenses? Can they be sped up?
    g. QUESTION: Do the show Opal is worth it, vs. allowing shared segments in larger address space on Unix?
    h. COMMENTARY: don't really show that flexibility is needed, don't evaluate sharing patterns in programs to show how useful the features are.
6. Opal Commentary
    a. SASOS never caught on, but there is HW support in Itanium
    b. Virtual machine approach, with even more isolation and less sharing, says that even traditional address spaces aren't strong enough protection, need VMaching level isolation
    c. ASSESSMENT: Opal is an optimization for sharing. Makes sharing faster, easier, but may lead to more sharing-related bugs.
    d. Idea of separating protection from addressing valuable, I used it in Nooks
    e. Research style: take an idea, push it to an extreme, see what works and what doesn't. Real-world use typically picks up a few ideas where it makes more sense.

3. ExoKernel – exterminate abstractions
    a. Background:
        i. lots of effort at customizing/extensible kernels (e.g. hydra, mach, vino, spin). Mostly aimed at taking an existing kernel and adding extensions, building a new architecture that allows extensions to be downloaded, or moving functionality to user-mode where it can be replaced.
        ii. Basic challenge: all these approaches tend to be slow
    b. Context
        i. Predated modern virtual machines – they are like an exokernel
        ii. Paper presented at HotOS workshop – for new ideas, not fully-baked systems. Later published 2 papers demonstrating performance, flexibility, sharing
        iii. Trying to be inflammatory – to get attention.
    c. Problem:
        i. operating systems getting big and bloated. QUESTION: HOW BIG IS A KERNEL? Linux?
        ii. Abstractions offered useful for low-perf apps (sometimes), but mismatch between offering and what apps need makes programming difficult, hurts performance.
        iii. Example: automatic virtual memory. App may have better sense of its needs, e.g. what pages to replace and when
    d. Question: how do you get the extensibility, flexibility, sharing and protection with all the overhead?

    e. CLAIM: impossible to provide abstractions good for all applications and efficient
        i. Let's examine claim:
        ii. What is an OS? code that securely multiplexes + abstracts hardware.
            1. Code one can neither neither change nor avoid (baked in!)
            2. Microkernel move code around, perhaps allow some substitution but not much.
            3. Securely multiplex: why? so can run multiple programs
            4. Abstract: why?
                a. Makes multiplexing easier (at higher level, conceals some details. E.g. file systems)
                b. Easier to write program - don't need to deal with low-level details

5. Lots of good ideas proposed but not adopted by OS - demonstrates limits of current OS design
    a. how do we know all the ideas are good?
    b. lots of things are adopted into operating systems

f. Current design leads to:
    i. **poor reliability,** because complex systems needed (e.g. vm, COW, mmap, multithreading)
    ii. **Poor adaptability**: hard to add new policy, mechanism. Change not localized because OS applies to all apps
    iii. **Poor performance**: abstractions take time to execute; applications that don't need them still pay. Example: GC or DB that interacts poorly with VM and would do better managing memory itself
    iv. **Poor flexibility**: apps can't implement their own abstractions, just emulate on top of existing OS at high cost

g. Discussion of value of abstractions
    i. Benefits
        1. Higher level of programming
        2. Code reuse – don't have to rewrite low-level code
        3. Higher-level policy; can enforce policies with more knowledge
            a. E.g. more information about sharing & cooperation
        4. More security: can group information for finer-level access control
        5. Can share at a higher level – e.g. files instead of blocks, to get better consistency semantics
        6. Easier to write programs – can deal with things you think about (contiguous memory, files), not hardware
    ii. Drawbacks:
        1. Performance: wrong abstraction means what you want is very expensive
            a. E.g. file systems when you want block storage, or have large files, or billions of small files
            b. E.g. networking – can't get to inner access of how protocols work, hard to avoid copying
        2. Hard to change monolithic kernel – difficult, complex code
            a. E.g. optimize to deliver network packets right off disk to web server

b. Issue isn't as much one of booting into another OS, as to modifying to do what you want

3. End-to-end argument
    a. Application knows best
    b. application has to handle things anyway
    c. might as well give power to the application to do things their way
        i. Optimize for mutual trust instead of distrust

b. Solution:
    i. Big idea:
        1. Kernel provides minimum necessary to securely multiplex hardware, but no abstractions (if possible)
        2. **OS runs as a library attached to application, provides abstractions**
            a. **should be easier to extend, better tuned to applications**
            b. **provides same high level abstractions to make it easy to program**
            c. **EVERYTHING that used to be in the kernel other than sharing & protecting hardware moves to a library IN YOUR PROCESS.**
            d. **THEY IMPLEMENTED POSIX interface in a library, to run standard Unix code**
        3. Idea is NOT:
            a. have a trusted user-mode implementation of a service
        4. Idea IS:
            a. Run the OS code yourself in your process
            b. NO trusted third party
        5. **DISCUSSION: what happens to compatibility?**
            a. **Answer: you have this in the form of libraries, but you can bypass it or modify the libraries.**
            b. **E.g. can use raw packet send instead of standard TCP/IP implementation**
            c. **E.g. a database can directly write to disk and manage buffering, instead of using file system**
        6. **WHAT HAPPENS TO RELIABILITY?**
            a. **You run less code.**

b. **You can run common shared code if you want (but at lower performance).**
c. **Kernel crashes are worse than user-level crashes, as they impact everything and force a reboot**

ii. Minimal kernel, exokernel, that provides access to hardware if at all possible
    1. QUESTION: How?
        a. Provide base minimum: wired pages for exception handlers and page table, addresses of exception handlers
        b. Ensure safety at all times
            i. No use of other's resources (e.g. memory, CPU)
        c. Allow safe code downloaded to kernel
            i. For packet filter to decide which process gets a packet
    2. Expose hardware names – e.g. physical address
        a. So can do HW-dependent things, such as page coloring
        b. Use SECURE BINDINGS:
            i. When first using a resource, check access and cache.
                1. E.g. a TLB: verify mapping, then let it be used
                2. E.g. packet filter: verify selects right packets, then let it use
        c. For memory:
            i. Create capabilities to each page accessible by a process, which it presents to establish mappings.
            ii. Provide ability to insert into a TLB (could be software TLB in kernel) or remove
            iii. QUESTION: How compare with normal address space?
                1. Can use physical addresses directly, can use large pages if want, or page coloring`
        d. Processes:
            i. Provide address of an exception handler
                1. Know where to start code

2. What to do on seg fault, divide-by-zero, etc.
3. Know what to save/restore on context switch

3. Expose hardware events – e.g. swapping memory out
    a. So can choose what do do
4. Expose revocation: ask libos to do revocation
    a. Take away a CPU: libos may want to do some scheduling, or decide what state to save
    b. Take away memory: update PTE
5. **WHAT ABSTRACTIONS DO THEY KEEP?**
    a. **Capabilities: a right to use a HW resource**
        i. Check capabilities on use
        ii. Use HW support – e.g. TLB, or tables for disk blocks. Very lightweight
6. Provide SW TLB – apps can fill in own mappings safely
7. QUESTION: how do you handle resource sharing?
    a. Don't have enough policy in the kernel to make accurate decisions

iii. Provide abstractions in libraries – programs can choose what libraries they want
1. QUESTION: Who writes these?
    a. Could be OS developer, but leaves open modules for extension.
    b. Could be a port of an existing OS for compatibility
    c. Can run multiple simultaneously
2. QUESTION: What are implications? Do app developers have to write their own OS? Do app developers have to write own window routines in X because in user space, compared to windows? But can X be changed more easily?
    a. Can think of OS code being more modular – can replace parts of it, just for your app. Can leave interface alone, or subclass & add new features.
3. QUESTION: portability- how handled?
    a. A: library has hardware abstraction layer. But need not, if want extra performance

b. E.g. routines to manipulate page tables, handle specific exception formats
4. QUESTION: What are assumptions
    a. Not for timesharing; relatively trusted libOS because computer user chooses it,
    b. Probably not for malicious applications
5. Each application links to a libOS, which provides OS-level APIs
    a. Applications trusting each other can have a libOS that shares state
6. LibOS chooses what state to save/restore on context switch
7. LibOS can be customized to the application – different mechanisms and policices
8. QUESTION: Impact on complexity?
    a. Previously OS / device drivers hide complexity – only do once
    b. Now: all libOS have to do it, or share code
    c. Now: can have libOS that only has the features you need – leave out everything else
        i. makes libOS simpler
    d. When you need new functionality:
        i. need to change libOS
        ii. maybe copy code, maybe reimplement
        iii. e.g. static web server -> dynamic web server
        iv. Kernel handles dependencies between services, but in libOS, when add services, need to manage this separately

iv. Provide some functionality via safe downloaded code into kernel
    1. Code has guaranteed completion
    2. Limited access to memory
    3. Used for packet filters, event notification (wake up), file system block translation

v. Big issues are sharing
    1. If kernel doesn't handle it, how does it happen?
        a. Between libOS – but can optimize for trust relationship
            i. Mutual trust (e.g. unix processes of same user)

        1. Allow sharing w/o OS intervention
    ii. Unidirectional trust (e.g. process/kernel, microkernel server)
        1. Trusted sides retains ownership of shared resources, e.g. page tables
    iii. Mutual suspicion
        1. Provide kernel support via. Downloaded code
        2. LibOS must treat all data suspiciously
    iv. QUESTION: should you trust? How does that impact reliability / security?