Administrivia

- My office hours Wed 10:45am this week
- Antonio’s office hours moved to Wednesday 3:30pm
- Lab 3A-C due next three classes
Access control

- Access control can be viewed as a matrix
  - Model introduced in 70s by Bell & Lapadula
  - Subjects are rows, objects are columns
  - Each cell encodes access rights (read/write/etc.)

- Have already seen Unix users and groups

- Systems also use often use access control lists (ACLs)
  - For each object, store permissible subjects & rights
  - AFS does this, on class machines
Capabilities

- **Slicing matrix other way yields capabilities**
  - E.g., For each process, store a list of objects it can access
  - Process explicitly invokes particular capabilities

- **Three general approaches to capabilities:**
  - Hardware enforced (Tagged architectures like M-machine)
  - Kernel-enforced (Hydra, KeyKOS)
  - Self-authenticating capabilities (like Amoeba)
Hydra

- Machine & programing env. built at CMU in ’70s
- OS enforced object modularity with capabilities
  - Could only call object methods with a capability
- Agumentation let methods manipulate objects
  - A method executes with the capability list of the object, not the caller
- Template methods take capabilities from caller
  - So method can access objects specified by caller
KeyKOS

- Capability system developed in the early 1980s
- Goal: Extreme security, reliability, and availability
- Structured as a “nanokernel”
  - Kernel proper only 20,000 lines of C, 100KB footprint
  - Avoids many problems with traditional kernels
  - Traditional OS interfaces implemented outside the kernel (including binary compatibility with existing OSes)
- Basic idea: No privileges other than capabilities
  - Partition system into many processes akin to objects
  - Capabilities like pointers to objects in OO languages
Unique features of KeyKOS

• Single-level store
  - Everything is persistent: memory, processes, …
  - System periodically checkpoints its entire state
  - After power outage, everything comes back up as it was
    (may just lose the last few characters you typed)

• “Stateless” kernel design only caches information
  - All kernel state reconstructible from persistent data

• Simplifies kernel and makes it more robust
  - Kernel never runs out of space in memory allocation
  - No message queues, etc. in kernel
  - Run out of memory? Just checkpoint system
KeyKOS capabilities

- Refered to as “keys” for short
- Types of keys:
  - devices – Low-level hardware access
  - pages – Persistent page of memory (can be mapped)
  - nodes – Container for 16 capabilities
  - segments – Pages & segments glued together with nodes
  - meters – right to consume CPU time
  - domains – a thread context
- Anyone possessing a key can grant it to others
  - But creating a key is a privileged operation
  - E.g., requires “prime meter” to divide it into submeters
Capability details

- Each domain has a number of key “slots”:  
  - 16 general-purpose key slots  
  - address slot – contains segment with process VM  
  - meter slot – contains key for CPU time  
  - keeper slot – contains key for exceptions

- Segments also have an associated keeper  
  - Process that gets invoked on invalid reference

- Meter keeper (allows creative scheduling policies)

- Calls generate return key for calling domain  
  - (Not required–other forms of message don’t do this)
KeyNIX: UNIX on KeyKOS

- "One kernel per process" architecture
  - Hard to crash kernel
  - Even harder to crash system
- Proc’s kernel is it’s keeper
  - Unmodified Unix binary makes Unix syscall
  - Invalid KeyKOS syscall, transfers control to Unix keeper
- Of course, kernels need to share state
  - Use shared segment for process and file tables
Keynix I/O

- **Every file is a different process**
  - Elegant, and fault isolated
  - Small files can live in a node, not a segment
  - Makes the `namei()` function very expensive

- **Pipes require queues**
  - This turned out to be complicated and inefficient
  - Interaction with signals complicated

- **Other OS features perform very well, though**
  - E.g., fork is six times faster than Mach 2.5
Self-authenticating capabilities

- Every access must be accompanied by a capability
  - For each object, OS stores random check value
  - Capability is: \{Object, Rights, MAC(check, Rights)\}
- OS gives processes capabilities
  - Process creating resource gets full access rights
  - Can ask OS to generate capability with restricted rights
- Makes sharing very easy in distributed systems
- To revoke rights, must change check value
  - Need some way for everyone else to reacquire capabilities
- Hard to control propagation
Limitations of capabilities

- IPC performance, as we’ve discussed
- Capability programming model never took off
  - Requires changes throughout application software
  - Call capabilities “file descriptors” or “Java pointers” and people will use them
  - But discipline of pure capability system challenging so far