Hydra

CS380L: Mike Dahlin

February 5, 2002

1 Preliminaries

1.1 Review

- Unix
  - Theme: Simplicity and elegance
  - File I/O
  - Process management
  - Lessons: compare and contrast with Multics, THE

1.2 Outline

- "Hydra: The kernel of a multiprocessor operating system"
  - Introduction
  - Detailed mechanism
  - An example
  - Postscript

1.3 Preview

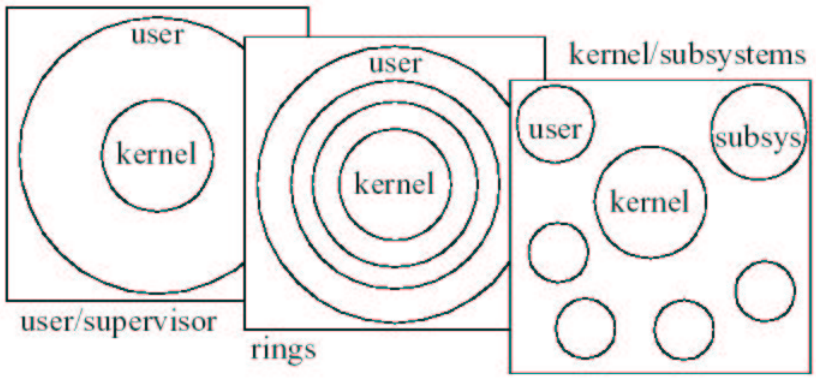
- Structure/extensibility: Synthesis, Exokernel, Disco
- Concurrency: Mesa, threads
- Communication: RPC, Active messages, duality of memory and communication

2 Introduction

- Background
  - Experimental machine and experimental OS
  - key concepts: (micro-) kernel and capability
  - Mcro-kernel
    - Minimal kernel in supervisor mode
    - Many OS services at user level
  - Push these ideas to the extreme
- Key ideas
  - Non-hierarchical protection
  - User-extensible type system
  - Capability-based protection
  - "Sealing/unsealing" for crossing protection boundaries

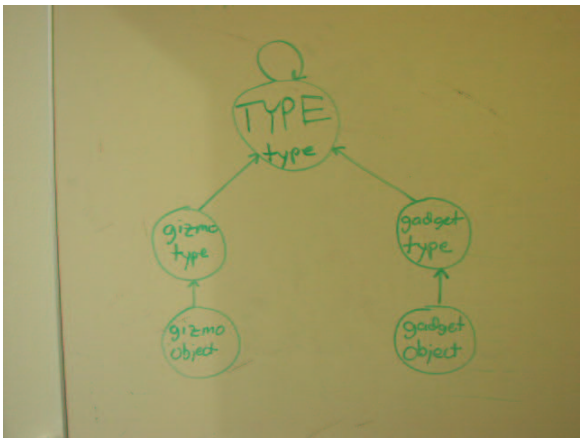
3 Detailed mechanism

3.1 Non-hierarchical protection



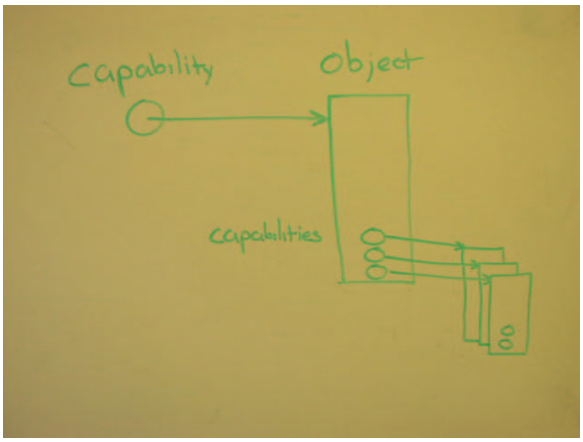
- Hierarchical systems
  - Inner rings have strictly more power
  - Doesnt scale to many components
- Non-hierarchical systems
  - Subsystem: protected collection of code and data
  - No one subsystem is intrinsically more powerful than the other

3.2 Extensible type system



- Objects
  - name
  - type, and
  - representation
- Three-level type hierarchy
  - objects
  - type objects
  - the TYPE type object

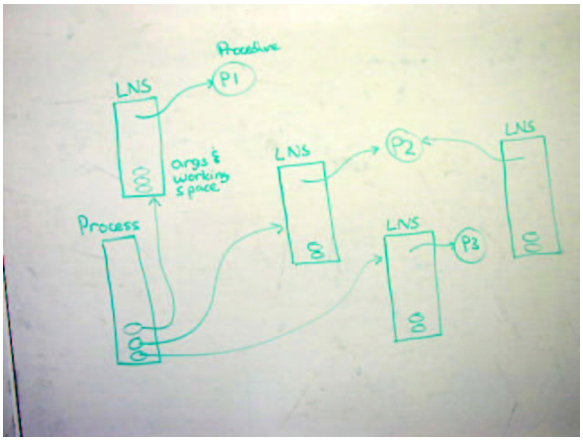
3.3 Capabilities and instance hierarchy



- Capability
  - object name
  - privileges
  - no other means of referencing objects
- Instance hierarchy
  - Embed capabilities as you would pointers
  - Can have cycles (so need garbage collection)

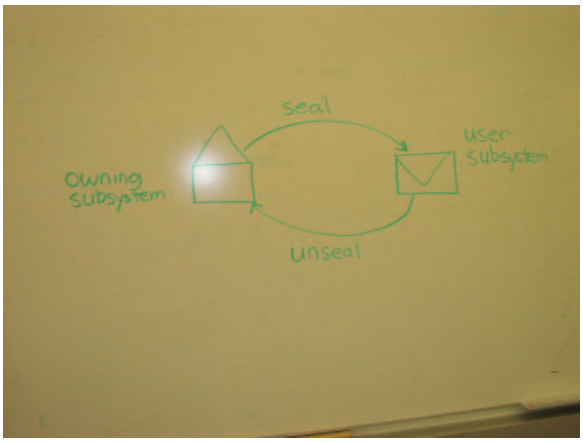
3.4 Built-in basic types

- Page
- Procedure
  - Static info associated with procedure
- Local name space (LNS)
  - - activation record
- Process
  - - stack of LNS
- Illustration:



3.5 Rights amplification and attenuation

- Example: Protected Queue subsystem; append method
  - Before calling, process should not be able to change the queue
  - → Call should amplify rights
  - During call, process (queue code) should not be able to modify the object being inserted
  - → Call should attenuate rights
- One implementation: Seal/unseal



- The "owning" subsystem
  - creates,
  - "seals", and
  - hands out an object
- "Other" subsystems
  - cant manipulate the object
  - can only hand it back to the owning system
- The "owning" subsystem
  - receives,
  - unseals, and

  - manipulates the object

- Seal/unseal can be explicit or implicit
  - Explicit: e.g., Pass encrypted objects
  - Implicit: e.g., Pass protected pointers (e.g., hydra)
  - Compare: Explicit allows distributed system; implicit allows pass by reference

- Queue example (with explicit seal):
  - Pass around encrypted queue
  - Pass queue encrypted stuff to add

- Hydra template: the "Fifth Element"
  - a tuple: (type, old privs, new privs)
  - one template for each argument in a procedure

- Protection boundary crossing
  - the kernel checks each argument
  - if type matches, and
  - old privs are sufficient
  - amplify rights to allow new privs

- Example: Queue::Append(Queue q, Data d);
  - Suppose you have a Queue that accepts SecretItems, but in the course of debugging the queue, you want to call the protected SecretItem::print method.
    * e.g.,
    * Queue::enqueue(Queue q [needs: pub, amplifies: rw], SecretItem i [needs: NULL, amplifies NULL]);
    * SecretItem::print(SecretItem s[needs: view, amplifies: rw])
  - 3 ways to do this
    1. Turn off security in SecretItem::print()

    * e.g.,
    * Queue::enqueue(Queue q [needs: pub, amplifies: rw], SecretItem i [needs: NULL, amplifies NULL])...i.print()...;
    * SecretItem::print(SecretItem s[**needs: NULL, amplifies: rw**])
    * DA: Turns off security for anyone that wants to print a secret item!
  2. Use capabilities passed in with SecretItem argument
    * e.g.,
    * Queue::enqueue(Queue q [needs: pub, amplifies: rw], SecretItem i [needs: NULL, amplifies NULL])...i.print()...;
    * SecretItem::print(SecretItem s[needs: view, amplifies: rw])
    * DA: Only get partial debugging info (e.g., for those items for which caller happened to include a permission he probably shouldn't have included...).
  3. Trust Queue class to access protected SecretItem methods
    * e.g.,
    * Queue::enqueue(Queue q [needs: pub, amplifies: rw], SecretItem i [needs: NULL, **amplifies view**]**)...i.print()...**;
    * SecretItem::print(SecretItem s[**needs: view, amplifies: rw**])
    * SecretItem declaration ... **friend Queue;**...
    * Note: Only can do this if you can modify both Queue and SecretItem code...that seems right.
    * "Friend" declaration is syntactic sugar...how could you implement it?
      - Creator of SecretItemClass uses SecretItemClass
      - Pass SICPower capability to Queue
- Easy to see how to amplify/attenuate "this" object
- Param0: Type = Queue, In = append, Out = read, write

- Limiting amplification **skip this – see hydra-example.pdf**
- Paper implies: all parameters checked and potentially amplified
- Given the above, it would seem the author of a procedure could take rights to everything.

- Example: Hand reference to SecretData to EvilQueue
- Example: Hand reference to disk to EvilQueue
  - How to prevent EvilQueue from amplifying its rights (e.g., to call SecretData::RevealSecret method or Disk::RawRead method) when it shouldn't?
- Answer: I'm confused...paper clearly implies that procedure can amplify permissions on incoming types, but that clearly can't be right (or at least it can't be the whole story).
  - Answer 1: Only allow Class X to amplify pointers of type X
    - Simple. "Type specific" checking all happens within type. Why amplify earlier than this, anyhow (the reference only can be used to call the object...do security checking then.)
  - Answer 2: Anly allow Class X to amplify pointers of type Y if type X has a capability to do so
    - Type X's type object contains capabilities
    - At instantiation time, check instantiated templates (from the type class) against the capabilities (from the type class)
    - This is my best guess at what they mean...
    - Could imagine Hydra support: class X has by default the capability to specify Templates for class X; class X can hand this capability to class Y's object if it wants
    - Could imagine language support to automate this in compiler...: "friend class" etc.
  - Answer 3...Suggestions/interpretations?

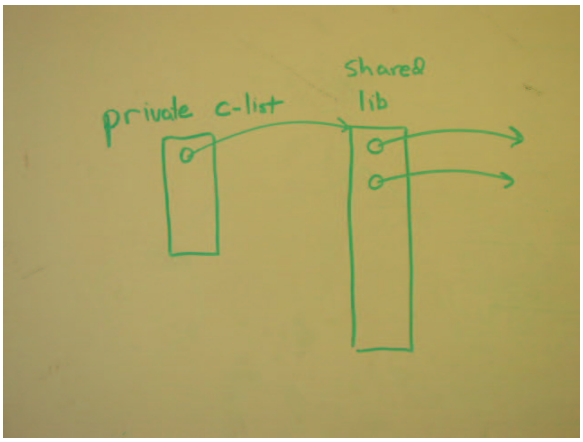
4 Admin

- Project

5 Example

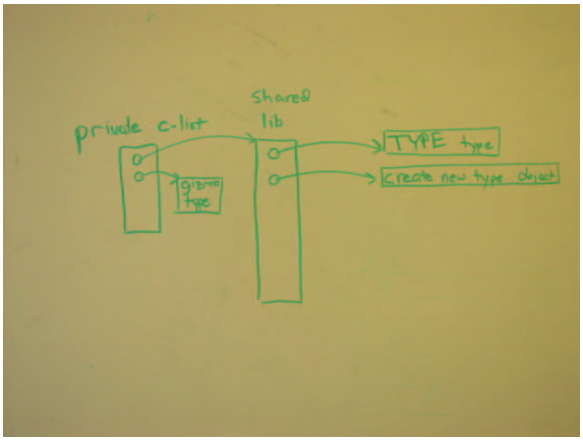
skip this – see separate hydra-example.pdf

1. Log in



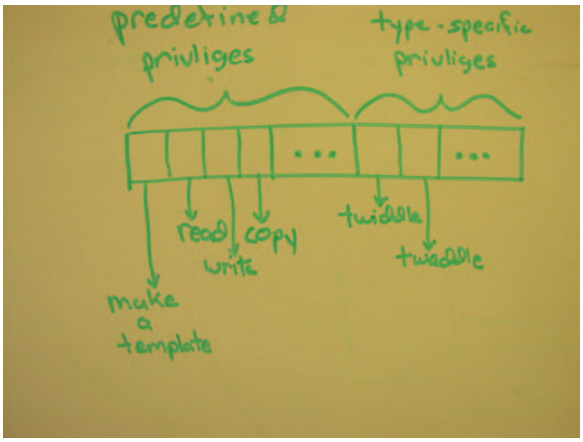
   - clist and shared library

2. Create gizmo type

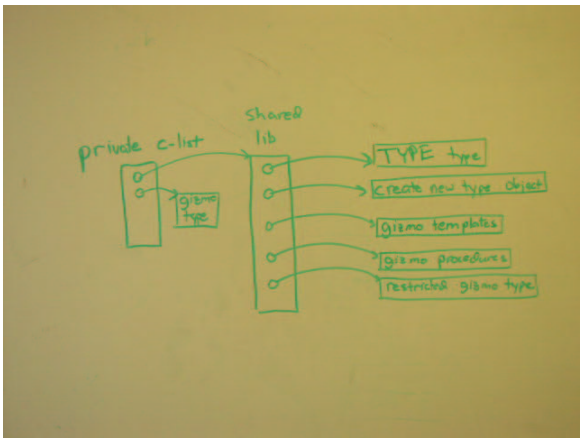


   - create new type using TYPE type object

- store capability in private clist
- Gizmo privliges



3. Package and publish gizmo

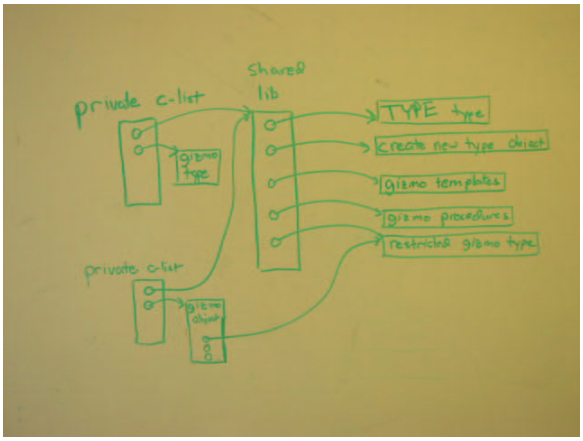


- create gizmo templates
- package and publish procedures and templates
- publish restricted copy of gizmo type
-

4. Another user logs in

- Gain capability for shared library

5. User 2 creates gizmo object



   - Invoke gizmo::create()

6. User 2 invokes gizmo::twiddle()

   - Rights amplification via template
     - type = gizmo
     - old priv = twiddle
     - new priv = read/write

6 Postscript

- A "Glorious failure"
- Persistent object stores keep re-inventing this idea
  - Java?
  - .net?

- Extra credit bounty: It seems to me that Multics, Hydra, and Java all do about the same thing with about the same mechanisms, but the differences in terminology are significant enough that it is hard to compare the mechanisms.

  Write a crisp report that compares and contrasts the fine-grained protection of (a) Multics, (b) Hydra, and (c) Java 2 (stack inspection and/or Wallach's "cleaner" formulation in his dissertation). There should be at least two goals. First, to teach the reader how these things really work. You might consider stepping through a detailed example of both a non-trivial shared object (such as the "bibliography" in the Hydra paper) and initialization of rights access (e.g., login in hydra, Java's policy file, etc.). Second, you should compare these mechanisms: What specific mechanisms are common across systems? What general abilities are similarly easy to support (even if via superficially different means)? What can some systems do easily but others not do? Can one/how does one implement system A in terms of system B. It might be useful to create a common terminology across all of the systems and describe each in those terms. (References: Multics paper, Hydra paper, Java 2 security documentation from javadocs.sun.com, and Dan Wallach's (formerly of Princeton, now of Rice) SOSP paper (for a description of the ideas in Stack Inspection in Java) and thesis (for a cleaner design that has not, to my knowledge, been implemented in most Java systems.)