CSE221 Lecture 3

Aronya Baksy

October 2024

1 HYDRA: The Kernel of a Multiprocess OS

1.1 Design Philosophy

- Protection: extended not just to files but all OS resources
- Separation of mechanism and policy (as argued by Hansen in RC4000 paper)
- Rejection of strict hierarchical layering: limits flexibility, strangles experimentation
- Provide an environment for construction of a multiprocessor OS
- Reliability: since all h/w components are redundant, make the software reliable
- Integration of the design with implementation methodology.

1.2 Goals

- Decompose the OS into sub-systems and create a microkernel that supports these sub-systems
- Define mechanisms for protection between the sub-systems

1.3 Important Concepts in Protection

- **Protection Domain**: Space in which the entity has a specific set of access rights which go away once you leave that domain (e.g.: LNS in Hydra, concept of user/kernel space in UNIX, each user in UNIX)
- **Protected Control Transfer**: Move between execution domains (e.g. the WALK primitive in Hydra, syscalls in UNIX)
- **Rights Augmentation**: process for (temporarily) increasing some object's privilege access (e.g. setuid in UNIX, templates in Hydra)

1.4 Objects and Types

- All physical and logical resources available to Hydra programs are viewed as **objects**
- Hydra kernel supports creation and manipulation of new object types, instances of those types, and **capabilities**
- all object operations are performed through calls to the kernel.
- Each object has a unique **name**, a **type** that defines the ops that can be performed on it, and a **representation** which holds the current state of the object (data and C-list)
- A type object is the representative for all instances of a given resource. It contains info for creation of new instances of the type, and capabilites for procedures to operate on instances of the type
- The type object creates new objects of its type and performs all operations on them
- The Hydra kernel supports directly some types like: process, procedure, LNS (dynamic repr of executing procedure), device, semaphore, policy and type
- LNS (Local Namespace) is the *activation record* for a procedure call (several LNS can exist, one per invocation of the process)

1.5 The Hydra Environment

- A **procedure** has code, data, parameters and return values
- Procedure objects also contain list of objects to be accessed during execution (capabilities). These may be caller independent or specified by the caller.
- A **Local Name Space** (LNS) is the record of the *runtime environment* of a procedure when it is called
- A **capability** is a reference to an object along with a list of possible access rights. Possession of a capability is a token for accessing only those access rights on that object
- Capabilities are manipulated only by the kernel
- **Templates** characterize the actual parameters expected by the procedure
- When a procedure is called, the template is filled with the capabilites from parameters (this is the parameter checking), this allows the callee to have temporarily higher access to an object than the caller

1.6 Protection and Security

- Protection is a mechanism, security is a policy
- Not maintaining a hierarchy of privilege is super important to Hydra architects
- Everything in Hydra is either an object or a reference to an object (i.e. a capability)
- Hydra maintains reference counts and deletes objects whose refcounts are zero
- Some Access Rights are common to all objects (kernel rights) while some are type dependent (auxiliary rights)
- Kernel primitives CALL and RETURN allow transfer of control between procedures

1.7 Systems and Subsystems

- Hydra views the OS as objects and procedures that act on objects
- A "sub-system" consists of a distinct object-type to denote the style of object supported by that system, and a collection of procedures for dealing with that style of object (e.g. a file system)

2 Protection: Butler W. Lampson

2.1 Motivations

- Original motivation for protection was to ensure that users don't affect each other's data or access to hardware
- Added motivation is to ensure that processes don't do the same to each other

2.2 Protection Domain

- Programs can find themselves in different protection contexts, each with their own powers (e.g. programs in user mode vs kernel/*supervisor* mode)
- Essential property of domain is that it has potentially different access than other domains.
- Each domain must have its own memory address space
- Example provided is of a pure message passing system, where A and B are two processes communicating with each other (each process is a single domain)
- The advantages of this system are:
    1. A and B determine their points of entry (i.e. the points at which they wait for messages/responses) hence no arbitrary transfer of control
    2. Returns are also protected as the caller can ignore all messages from the callee except the required one in the format required by caller

3. If B does not return to A, then A must simply arrange before calling B, some reliable timer process C to interrupt A after a certain timeout
4. Callee can validate that it is being called only by authorized processes based on ID present in the message

- Drawbacks of this system:

1. No control over runaway processes
2. Process co-ordination requires elaborate systems

2.3 Objects and Access Matrices

- Need to provide mechanisms for controlling processes, as well as controlling what is to be shared with which process
- This is realized by the object system described, with 3 components, the *objects*, the *domains* and the *access function* (or access matrix A)
- Rows of A are *domains*, columns are *objects*
- A_{ij} is a set of *access attributes* (e.g.: read, write) that domain i has on object j
- Rules for populating A:

1. Domain d_1 can remove access rights from d_2 if it has **control** rights on d_2
2. d_1 can copy to $A[d_2, x]$ any attribute which has the copy bit set, and mention if the destination will have the copy bit set
3. If d_1 has **owner** access to object x then d_1 can add rights (with or without copy set) to any $A[d_2, x]$
4. If d_1 has owner access to object x and d_2 does not have **protected** access to x then d_1 can take rights away from $A[d_2, x]$

2.4 Implementing Access Control

- Simplest method: **linear list** T of 3-tuples $< d, x, A[d, x] >$
- Cons of this method:
  - Protected by hardware which does not use T hence outside OS designer's control
  - Since at any given point of time, most objs and domains are inactive, hence inconvenient to load entire table to main memory
  - Groupings may lead to inefficient use (e.g. a public file needs one entry for every single domain)
  - Can't retrieve access rights by object or domain all together easily
- **Approach 2:** Group table by **domain**, using entries of the form $< x, A[d, x] >$ attached to domain d. This is called a **capability list** and each $< x, A[d, x] >$ is a **capability**
- **Approach 3:** Group table by **object**, wherein the owner of each object provides a procedure to access $A_X(d)$
- Since domain names are hard to remember for processes, each requestor asks the kernel for an **access key** which is used to index