Thinking About Capabilities

An EROS-Centric View

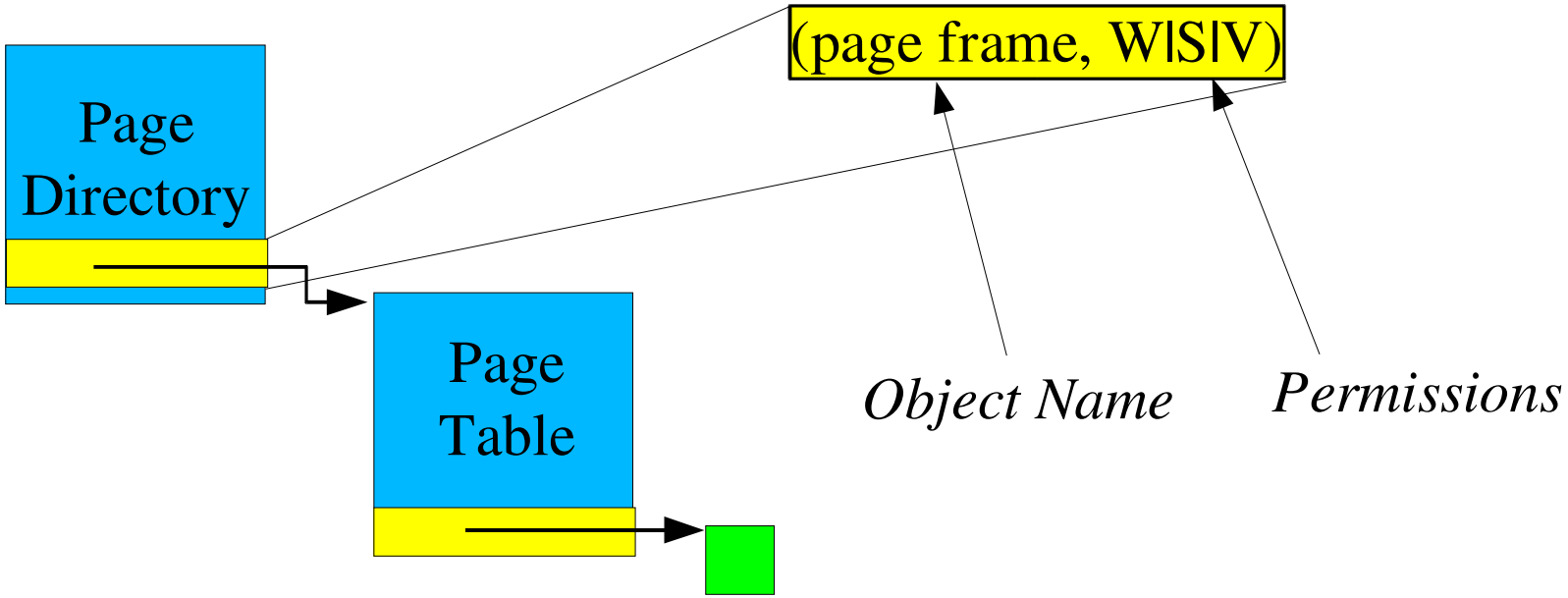
Jonathan S. Shapiro
Johns Hopkins University

shap@eros-os.org
www.eros-os.org

Plan of Talk

- Capabilities: a conceptual view
- Bits of Architecture
- Why is this idea so compelling?
- System-scale overview

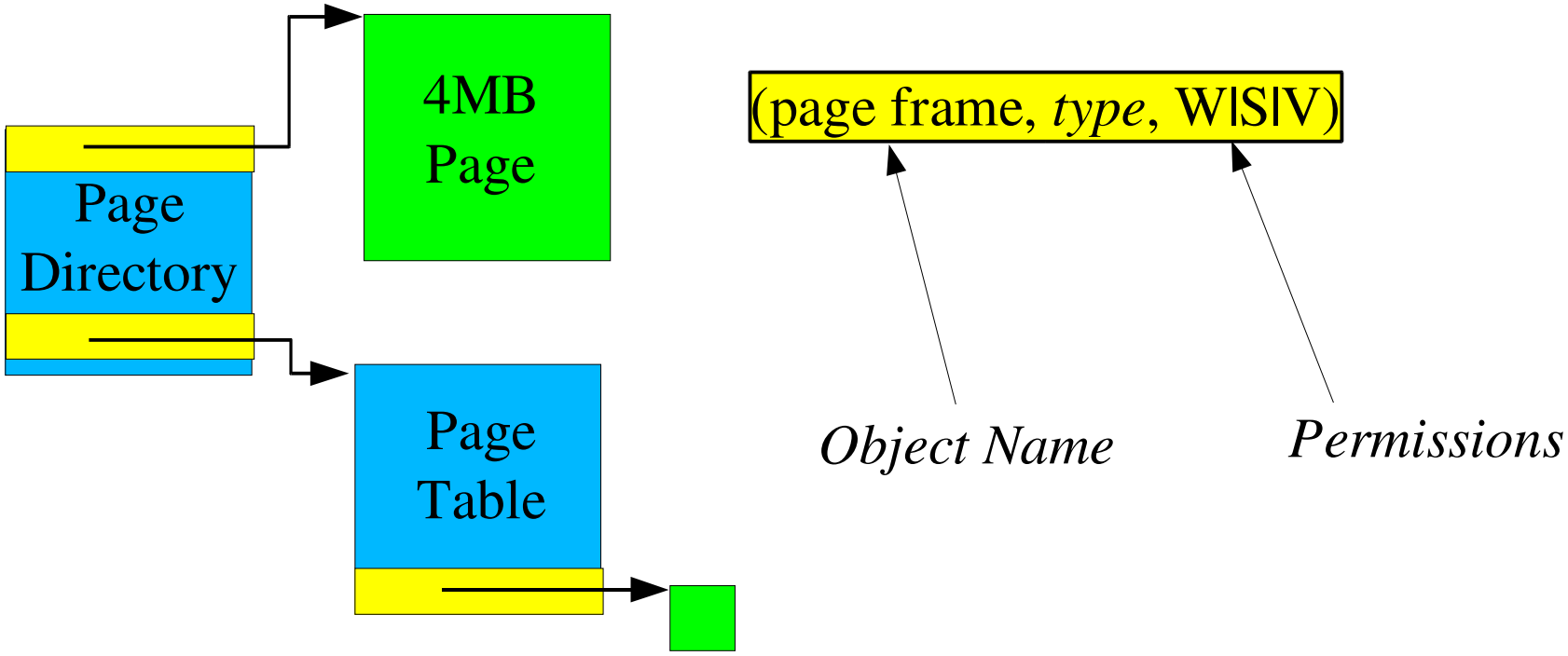
Conventional Page Tables



Classical Definition

- Term “capability” is due to Dennis and van Horn, 1966, *Programming Semantics for Multiprogrammed Computations*
- A capability is an (object name, access rights pair)
- The term “object name,” in this context, has been commonly (mis) understood to mean “the global name of some system resource.”
- A page table entry is a capability in exactly this sense, but the concept is much more general.

Modern Page Tables



Note that the *type* field was always latent, implied by containership in a structure (the page table/page directory) that contained typed slots

From Memory Protection to Objects

- Latent in this view of capability is a generalized notion of object semantics
  - Most of the uses have been in memory naming and protection
  - Capability is not just a memory idea
- Type + permission == interface (structure in *ML* terminology)
  - This implies that capabilities provide a *general* model for naming the interface to an arbitrary object.
  - OS people don't use the term “object” correctly
    - Object = behavior + representation state
    - OS people focus almost exclusively on the representation state

Examples from Real Systems

- UNIX:
  - Socket descriptor: capability to socket connection
  - File descriptor: capability to a file with RO or RW permissions
  - CWD, “root” descriptor: capabilities to file system, directory
- Windows has many of the same
- EROS uses capabilities pervasively:
  - Pages
  - Mapping structures
  - Processes
  - A few kernel services

What is EROS?

- A pure, capability-based operating system
  - It is an *object-based*, not a client-server architecture
- High performance invocation (includes IPC)
- Transparent persistence
- Built on a decidable access model
  - Questions of policy enforceability are decidable (and the outcome is good)
  - Confinement mechanism is verified
  - Implementation is not (and won't be)

Everything is an Object

- Kernel Implemented
  - Pages (hold data)
  - Nodes (hold capabilities)
  - Wrappers
  - Processes
  - Void object

(type, object-id, permissions)

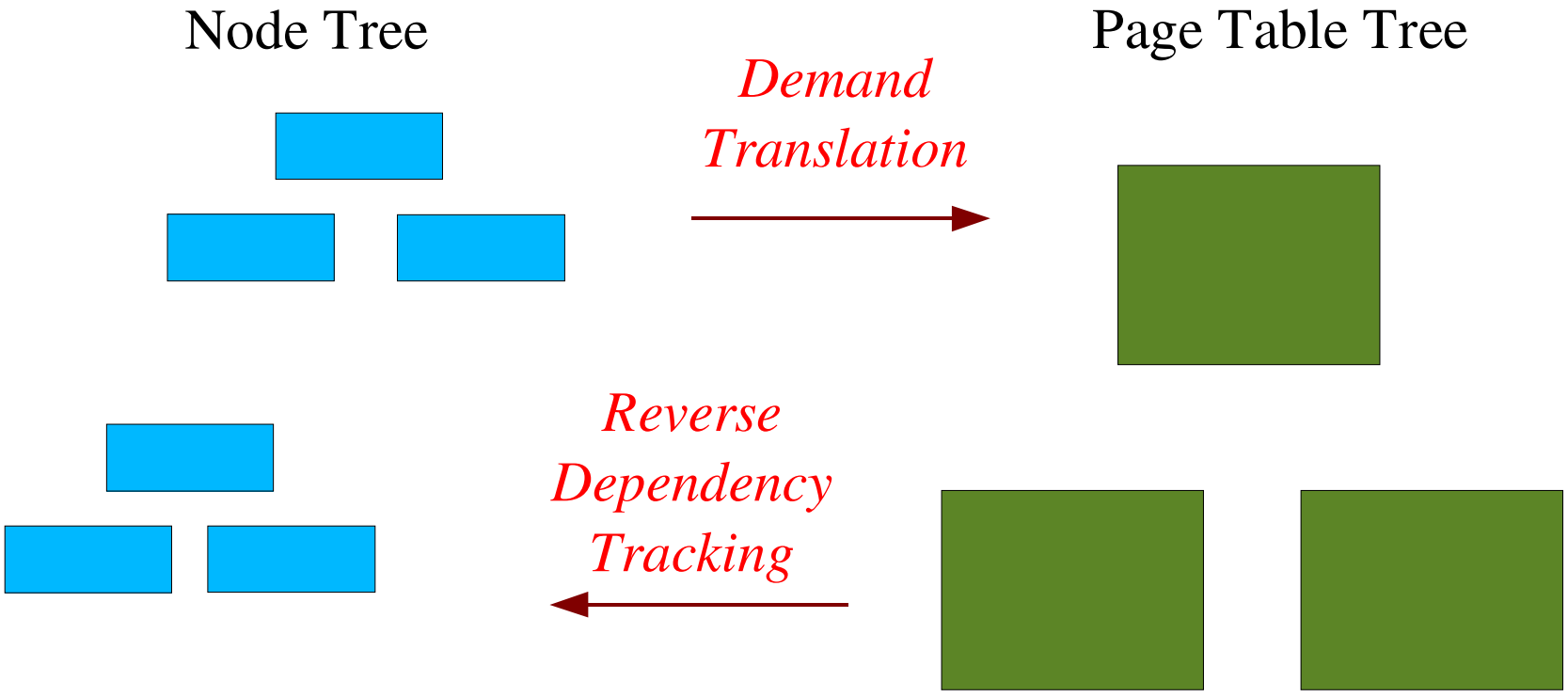
- User Implemented
  - Implemented by some user process
  - One process can implement multiple objects, multiple interfaces, or mutiple facets on a single object

(type, object-id, facet-id)

Nodes

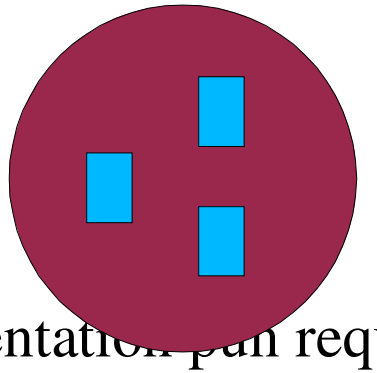
- Since capabilities are not user-accessible data, we need some container object to hold them: Nodes
- Each node is fixed size, holds 32 capabilities
  - Could be page-sized, but this was not space efficient.
  - In hindsight, probably should have made them page sized anyway
- Side effect: a type partitioning between data and authority that is carried through all the way to the disk

Nodes Define Address Spaces



Processes

- From the kernel perspective, interesting process state is capability state. For this reason, process state is represented as an arrangement of Nodes
  - New capability type “number capability” to hold the register bits.



- A representation pun required because of persistence
  - Not needed in a non-persistent design

Kernel Objects

- The kernel-implemented capabilities implement interfaces to the core kernel abstractions: pages, nodes, processes, and address spaces
- Because these are kernel objects, the kernel understands their semantics, and can implement permissions on them.
  - This is *NOT* true of user-implemented objects.
  - Example: kernel cannot tell when a user-implemented object is read-only
  - Given current kernel technology, user-mode (extended) objects are *necessarily* second-class w.r.t. The primitive protection system.

Capability Rescind

- Allocation Count
  - Most capability types carry a version number: the **allocation count**.
  - Every object likewise carries a version number.
  - Version is incremented on object rescind.
  - No match => capability is void.
- Call Count
  - Special mechanism for call/return. Similar to allocation count
  - Every node has a call count. Incremented by every call.
  - Call generates a resume key that contains call count for node.
  - No match => capability is void

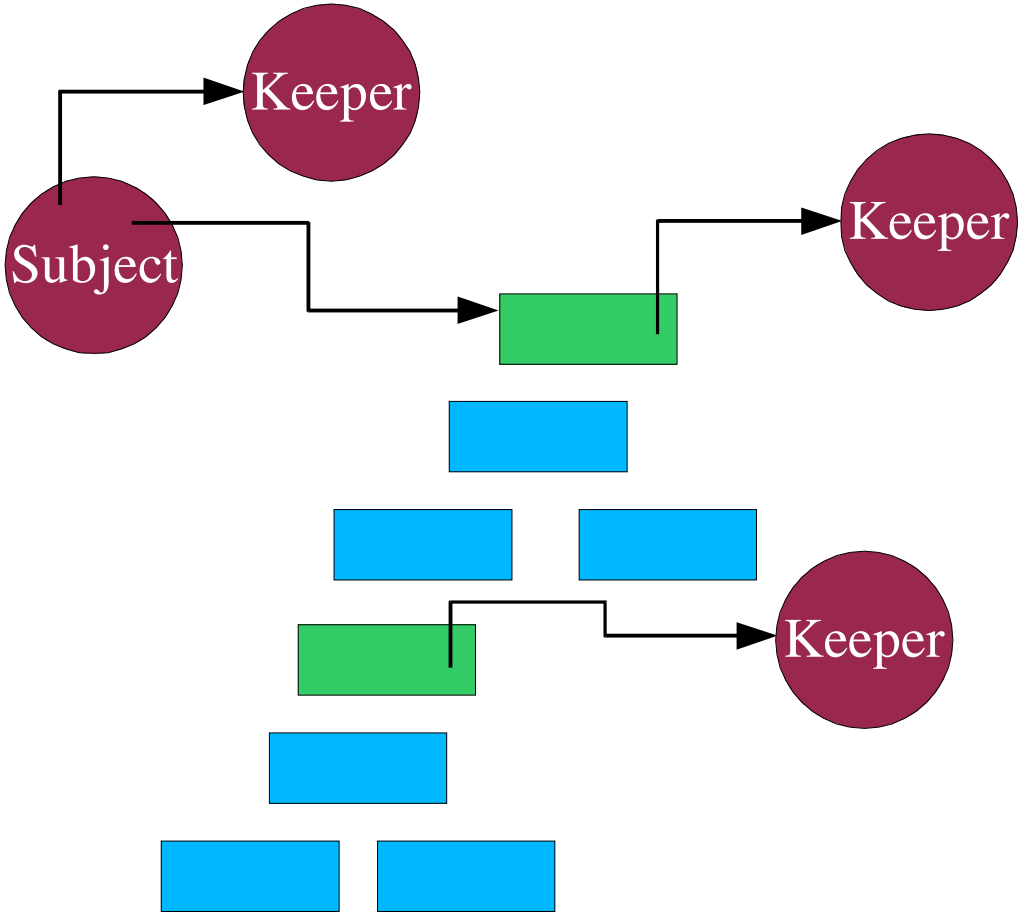
Protection Issue: Transitivity

- Capability systems present a problem: a read-only object may contain a read-write capability
  - Similar to non-const pointer within const object.
- Sometimes, the real issue is *transitive* read-only access.
- This motivates a new access restriction: *weak*
- Any capability that is fetched by invoking a weak capability will have read-only, weak access restrictions imposed on it.

Exceptions

- EROS distinguishes two types of exceptions:
  - Memory exceptions occur when accessing address spaces
  - Non-memory exceptions occur from mis-executing exceptions
- Memory exceptions are first delivered to the “appropriate” memory keeper (fault handler).
  - If no memory keeper is defined, they go to the process keeper.
  - Memory keeper can patch the problem and restart the instruction.
- All other exceptions go to the process keeper.
  - Identified by a per-process capability slot

Spaces, Processes have “Keepers”



Fault goes to *nearest enclosing keeper*

Process keeper encloses all memory keepers

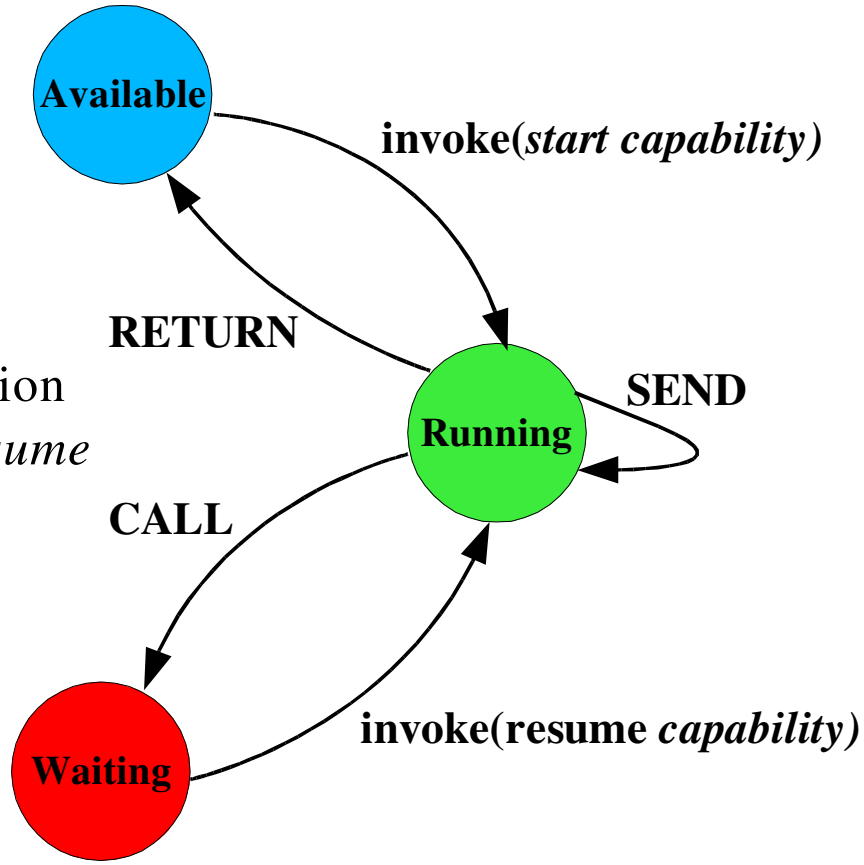
Interrupt-Style Kernel

- Originally: Every operation has three phases:
  - Prepare (includes all exceptions, access checks)
  - Commit
  - Mutate
- Now: certain operations cheat
  - Exceptions allowed during mutate
  - These restart the operation from the beginning
  - Restricted to mutations that do not alter security state
  - Security state updates only legal after success guaranteed

Persistence

- Entire system is periodically (efficiently) checkpointed
- Motivation: simplest path to secure bootstrap
  - Do not need to argue successful reduction of authority
  - Argue instead that saved state is successfully resumed
  - Argue that any saved state resulted from a correctness-preserving sequence of operations proceeding from an initially safe state
  - Check the base case separately
    - Via assurance (trusted components)
    - Via reachability (initial capabilities)

Capability Invocation



CALL invocation generates a *resume* capability

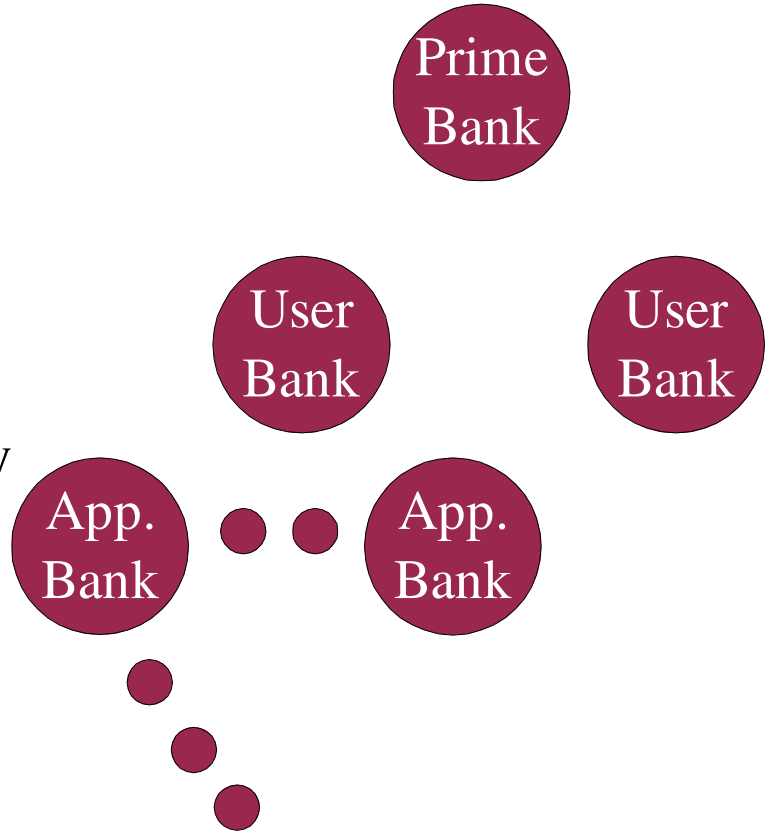
Invocation type determines *invoker* transition

Capability type determines *invokee* transition

Rule: kernel capabilities behave exactly as if a call was made to a *start* cap. to some process that returned using the generated *resume* cap. after producing the result by magic.

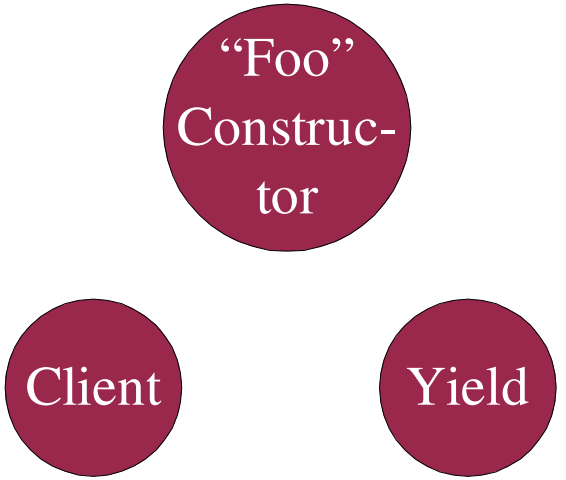
Space Bank Hierarchy

- All storage allocated from some space bank
- Space banks exist in logical hierarchy (all one program)
- Allocates *disk* space, not memory space
- Destroying bank either
  - Destroys all allocated storage, or
  - Propagates storage ownership to parent



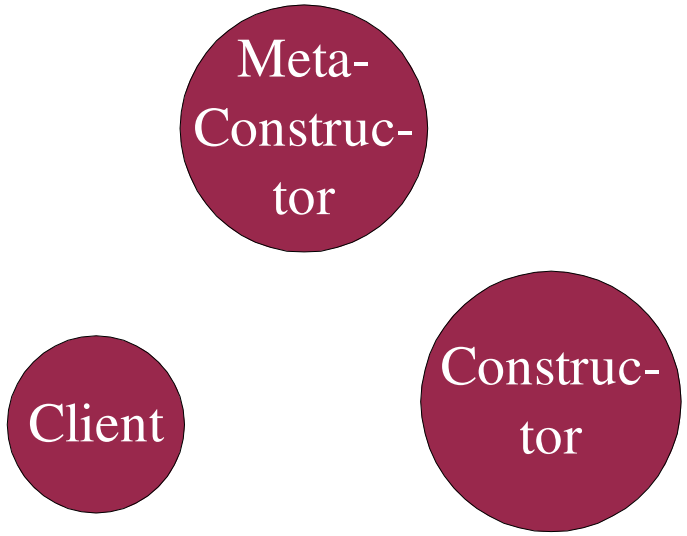
Constructor

- Constructs instances of some program
- Tests for confinement
  - By testing initial capabilities
  - New instance can *only* write to client at creation time.
  - Any further permission must come from client
- Definition is recursive
  - Capability to constructor of confined thing is considered safe

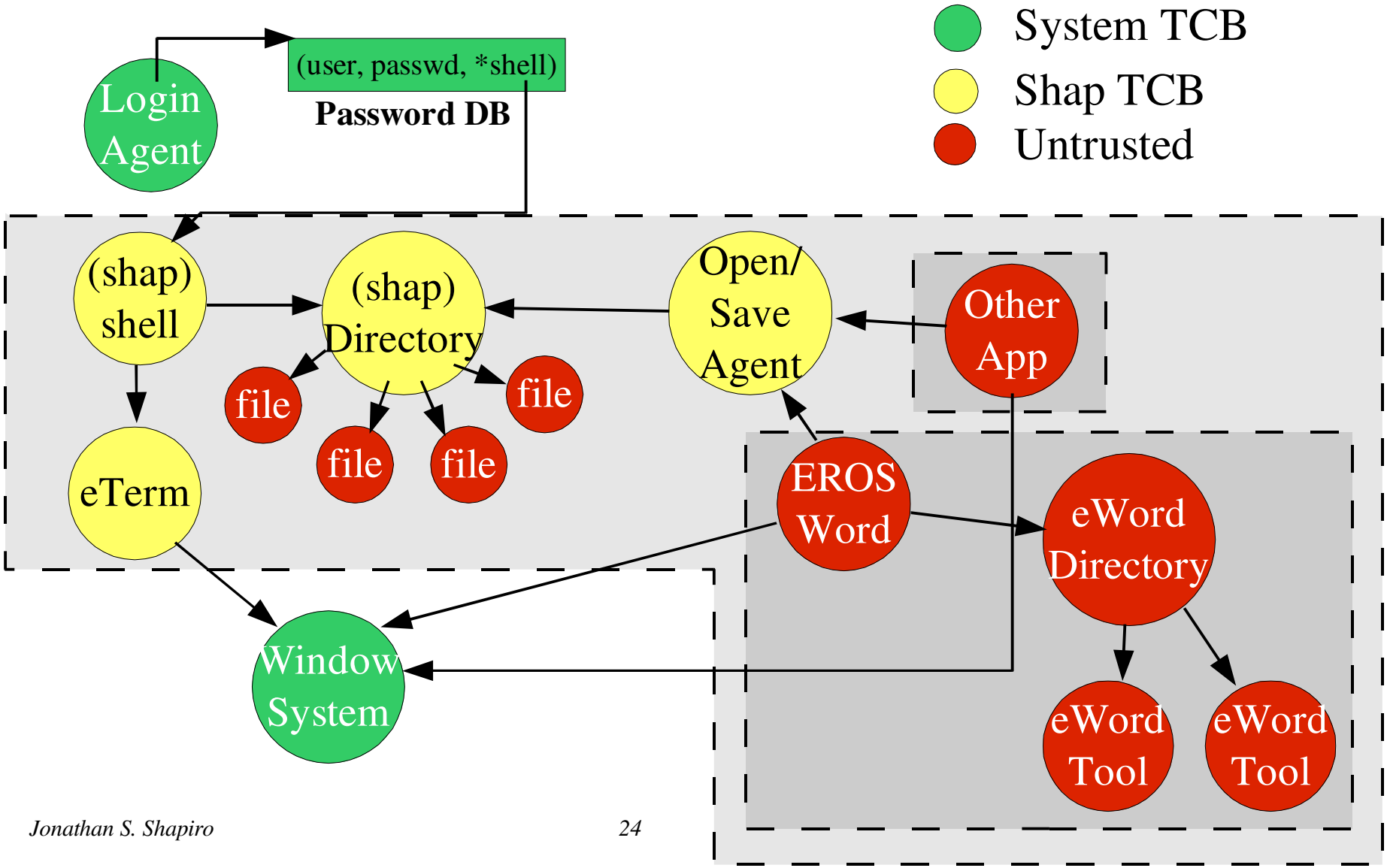


MetaConstructor

- Constructors are build by the singleton metaconstructor
- Space bank and metaconstructor are “primordial objects”



Big Picture


System TCB
Shap TCB
Untrusted
(user, passwd, *shell)
Password DB
Login Agent
(shap) shell
(shap) Directory
Open/ Save Agent
Other App
file
file
file
file
eTerm
EROS Word
eWord Directory
Window System
eWord Tool
eWord Tool

Why is This Idea So Compelling?

- Capability concept dates back to early 1960's; perhaps earlier.
- It has been conclusively discredited two or three times a decade, from both a theoretical and a practical perspective
- Yet it refuses to die, and the participants are a very unusual collection of operating system architects:
  - System architects: Needham, Lampson, Wulf, Fabry, Wilkes, Rashid (probably didn't know it), Neumann, Schroeder, Hardy, myself, many others
  - Theorists: Jones, Boyer, Levitt, Snyder, Lipton, Bishop, Boyer
- What do these people share in common, and why has this idea categorically refused to die?

Semantics

- Note the key word in the Dennis and van Horn title:
  - Programming *Semantics* for Multiprogrammed Computations
  - Largely unnoticed by the mainstream operating system community
- Hint
  - (object-name + interface) ≡ (closure + continuation)
  - Capability semantics ≡ lambda calculus w/ side effects
- The capability model is currently the *only* model offering a semantics that allows us to reason all the way from user-level object interactions down to machine-level instructions in a uniform and consistent way.
  - Or indeed, *any semantics of systems computation at all*

Models and Results

- Anita K. Jones, 1973
  - *Protection in Programmed Systems*
- Harrison, Ruzzo, Ullman, 1976
  - *Protection in Operating Systems*
- Jones, Lipton Snyder, 1976
  - *A Linear-Time Algorithm for Deciding Security*
- Neumann, Boyer, *et al.*, 1980
  - *A Provably Secure Operating System: The System, Its Applications, and Proofs*
- Shapiro, Weber, 2000
  - *Verifying the EROS Confinement Mechanism*
- Notably *not*:
  - Lampson, *Protection*
  - Static snapshots reveal very little about the evolution of dynamic systems

Recent Events: L4 Summit Meeting

- L4x3 (evolution from L4*x*2) will be a capability system
  - Now provides descriptors for all system resources
- EROS and L4 groups appear to be merging into a single effort to provide a high-performance, protected system
- Extended “team” includes several groups interested in formal verification.

Invocation Performance

- Not measurably different from L4 in common case
  - Usual case: 1 resume capability in call, 0 in return
  - Rest of path nearly identical
- No intrinsic reason to believe that this should change