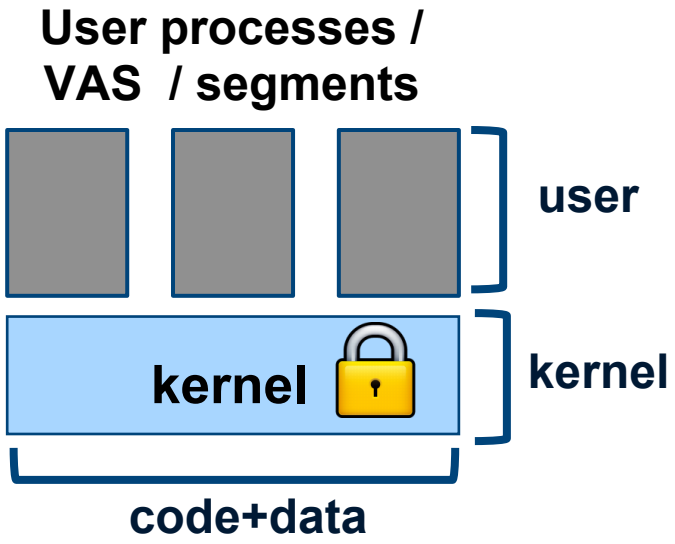
Singularity
Part 2

Jeff Chase

Today

- **Singularity: abstractions**
- **How do processes interact?**
  - Communicate / share
  - Combine
  - (De)compose
  - Extend
- **How to invoke the kernel?**
- **How does the kernel decide what to allow?**



Platform abstractions

- Platforms provide “building blocks”…
- …and APIs to use them.
  - Instantiate/create/allocate
  - Manipulate/configure
  - Attach/detach
  - Combine in uniform ways
  - Release/destroy

The choice of abstractions reflects a philosophy of how to build and organize software systems.

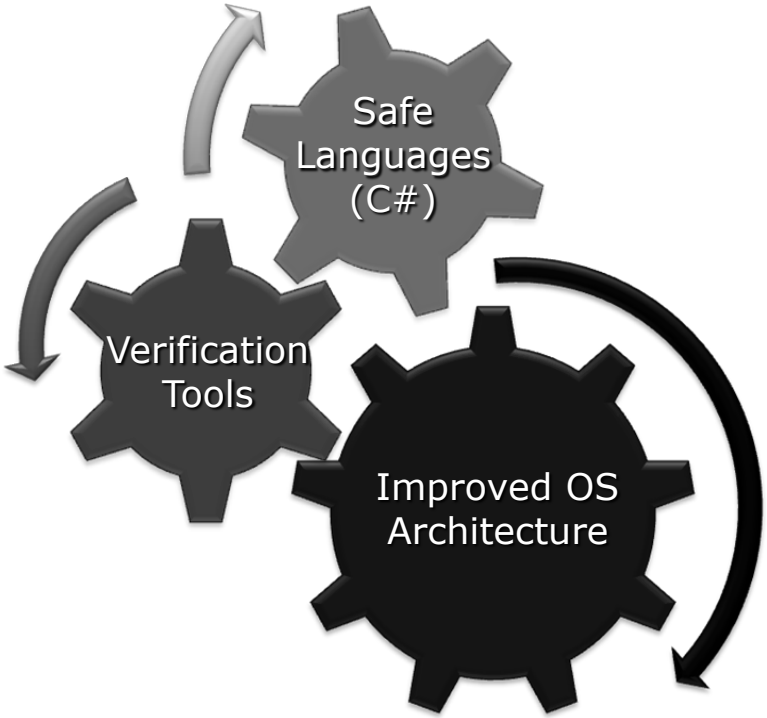
Singularity: Rethinking the Software Stack

Galen C. Hunt and James R. Larus
Microsoft Research Redmond

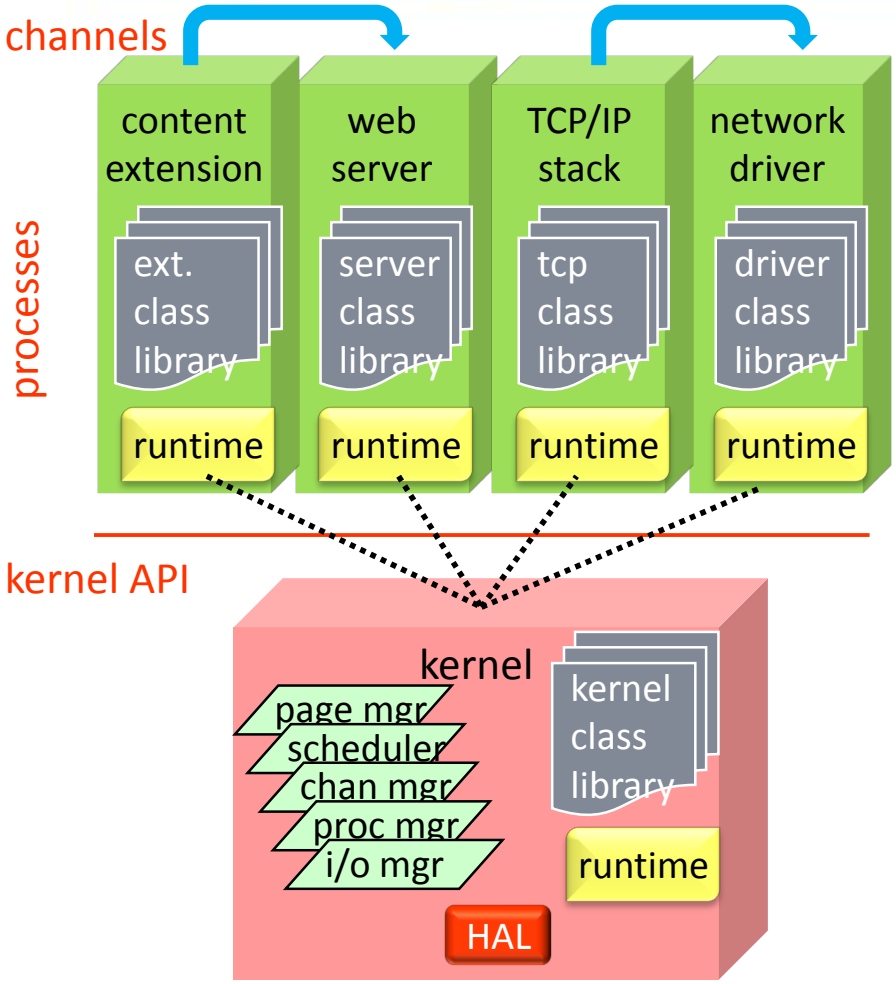
galenh@microsoft.com

ABSTRACT

Every operating system embodies a collection of design decisions. Many of the decisions behind today's most popular operating systems have remained unchanged, even as hardware and software have evolved. Operating systems form the foundation of almost every software stack, so inadequacies in present systems have a pervasive impact. This paper describes the efforts of the Singularity project to re-examine these design choices in light of advances in programming languages and verification tools. Singularity systems incorporate three key architectural features: software-isolated processes for protection of programs and system services, contract-based channels for communication, and manifest-based programs for verification of system properties. We describe this foundation in detail and sketch the ongoing research in experimental systems that build upon it.



Singularity OS Architecture



- Safe micro-kernel
  - 95% written in C#
    - 17% of files contain unsafe C#
    - 5% of files contain x86 asm or C++
  - services and device drivers in processes
- Software isolated processes (SIPs)
  - all user code is verifiably safe
  - some unsafe code in trusted runtime
  - processes and kernel sealed at start time
- Communication via channels
  - channel behavior is specified and checked
  - fast and efficient communication
- Working research prototype
  - not Windows replacement

Sealing OS Processes to Improve Dependability and Safety

Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson,
James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber

Microsoft Research
One Microsoft Way
Redmond, WA 98052 USA

singqa@microsoft.com

ABSTRACT

In most modern operating systems, a process is a hardware-protected abstraction for isolating code and data. This protection, however, is selective. Many common mechanisms—dynamic code loading, run-time code generation, shared memory, and intrusive system APIs—make the barrier between processes very permeable. This paper argues that this traditional *open process architecture* exacerbates the dependability and security weaknesses of modern systems.

As a remedy, this paper proposes a *sealed process architecture*, which prohibits dynamic code loading, self-modifying code, shared memory, and limits the scope of the process API. This paper describes the implementation of the sealed process architecture in the Singularity operating system, discusses its merits and drawbacks, and evaluates its effectiveness. Some benefits of this sealed process architecture are: improved program analysis by tools, stronger security and safety guarantees, elimination of redundant overlaps between the OS and language runtimes, and improved software engineering.

General Terms

Design, Reliability, Experimentation.

Keywords

Open process architecture, sealed process architecture, sealed kernel, software isolated process (SIP).

1. INTRODUCTION

Processes debuted, circa 1965, as a recognized operating system abstraction in Multics [48]. Multics pioneered many attributes of modern processes: OS-supported dynamic code loading, run-time code generation, cross-process shared memory, and an intrusive kernel API that permitted one process to modify directly the state of another process.

Today, this architecture—which we call the *open process architecture*—is nearly universal. Although aspects of this architecture, such as dynamic code loading and shared memory, were not in Multics' immediate successors (early versions of UNIX [35] or early PC operating systems), today's systems, such as FreeBSD, Linux, Solaris, and

The open process architecture

Processes debuted, circa 1965, as a recognized operating system abstraction in Multics [48]....Today, this architecture—which we call the **open process architecture**—is nearly universal. Although aspects of this architecture, such as dynamic code loading and shared memory, were not in Multics' immediate successors (early versions of UNIX [35] or early PC operating systems), today's systems, such as FreeBSD, Linux, Solaris, and Windows, embrace … the open process architecture.

The open process architecture is commonly used to extend an OS or application by dynamically loading new features and functionality directly into a kernel or running process. For example, Microsoft Windows supports over 100,000 third-party, in-kernel modules ranging in functionality from device drivers to anti-virus scanners. Dynamically loaded extensions are also widely used as web server extensions (e.g., ISAPI extensions for Microsoft's IIS or modules for Apache), stored procedures in databases, email virus scanners, web browser plug-ins, application plug-ins, shell extensions, etc.

Sealing

The **sealed process architecture** imposes two restrictions: the code in a process cannot be altered once the process starts executing and the state of a process cannot be directly manipulated by another process.

The system prohibits dynamic code loading, self-modifying code, cross-process sharing of memory, and provides a process-limited kernel API.

A **sealed kernel** is an OS kernel that conforms to the same two restrictions: the code in the kernel cannot be altered once the kernel starts executing and the state of the kernel cannot be directly manipulated by any process.

2.1. Sealed Process Invariants

1. The **fixed code** invariant: Code within a process does not change once the process starts execution.

2. The **state isolation** invariant: Data within a process cannot be directly accessed by other processes.

3. The **explicit communication** invariant: All communication between processes occurs through explicit mechanisms, with explicit identification of the sender and explicit receiver admission control over incoming communication.

4. The **closed API** invariant: The system's kernel API respects the fixed code, state isolation, and explicit communication invariants.

The fixed code invariant does not limit the code in a process to a single executable file, but it does require that all code be identified before execution starts. A process cannot dynamically load code and should not generate code into its address space.

Microkernel and extensions

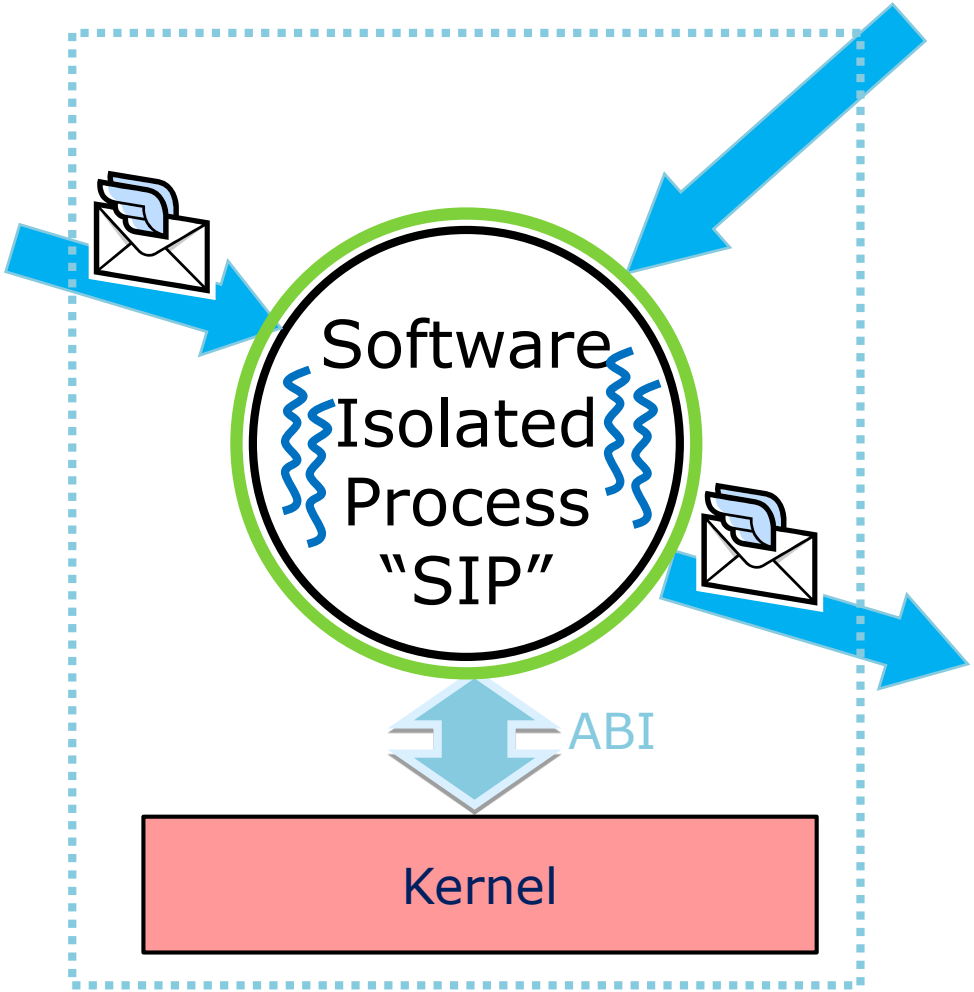
The Singularity kernel is a **microkernel**; all device drivers, network protocols, file systems, and user replaceable services execute in SIPs outside the kernel. Functionality that remains in the kernel includes scheduling, mediating privileged access to hardware resources, managing system memory, managing threads and thread stacks, and creating and destroying SIPs and channels.

Microkernel operating systems, such as Mach [1], L4 [24], SPIN [6], VINO [38], Taos/Topaz [45], and the Exokernel [11], partition the components of a monolithic operating system kernel into separate processes to increase the system's failure isolation and reduce development complexity.

The sealed process architecture generalizes this sound engineering methodology (modularity) to the entire system. Singularity provides lightweight processes and inexpensive interprocess communication, which enable a partitioned application to communicate effectively.

How is Singularity different?

Process Model



- Process contains only safe code
- No shared memory
  - communicates via *messages*
- Messages flow over channels
  - well-defined & verified
- Lightweight threads for concurrency
- Small binary interface to kernel
  - threads, memory, & channels
- Seal the process on execution
  - no dynamic code loading
  - no in-process plug-ins
- Everything can run in ring 0 in kernel memory!

SIPs can execute privileged instructions!

SIPs may run in kernel mode, in a single kernel address space!

Because type and memory safety assure the execution integrity of functions, Singularity can place privileged instructions, with safety checks, in trusted functions that run inside SIPs.

For example, privileged instructions for accessing I/O hardware can be safely in-lined into device drivers at installation time. Other ABI functions can be in-lined into SIP code at installation time as well.

Singularity takes advantage of this safe in-lining to optimize channel communication and the performance of language runtimes and garbage collectors in SIPs.

ABI

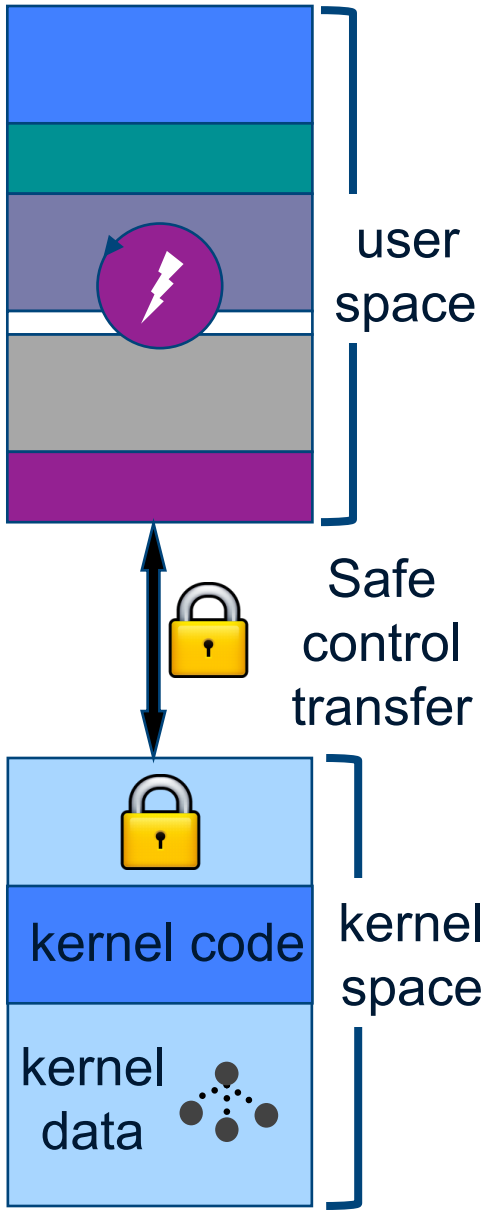
SIPs communicate with the kernel through a **limited API** that invokes static methods in kernel code. This interface isolates the kernel and SIP object spaces. All parameters to this API are values, not pointers, so the kernel's and SIP's garbage collectors need not coordinate.

What state belongs to a SIP?

The call to stop a child SIP stops its threads and then destroys its state.

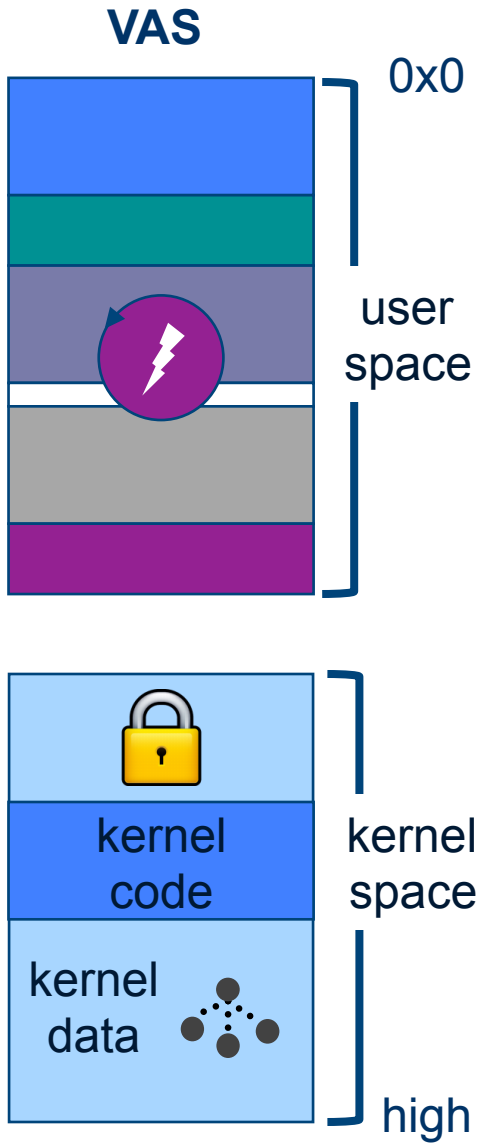
How do we know which objects in the shared address space below to a particular SIP and its threads?

User/kernel

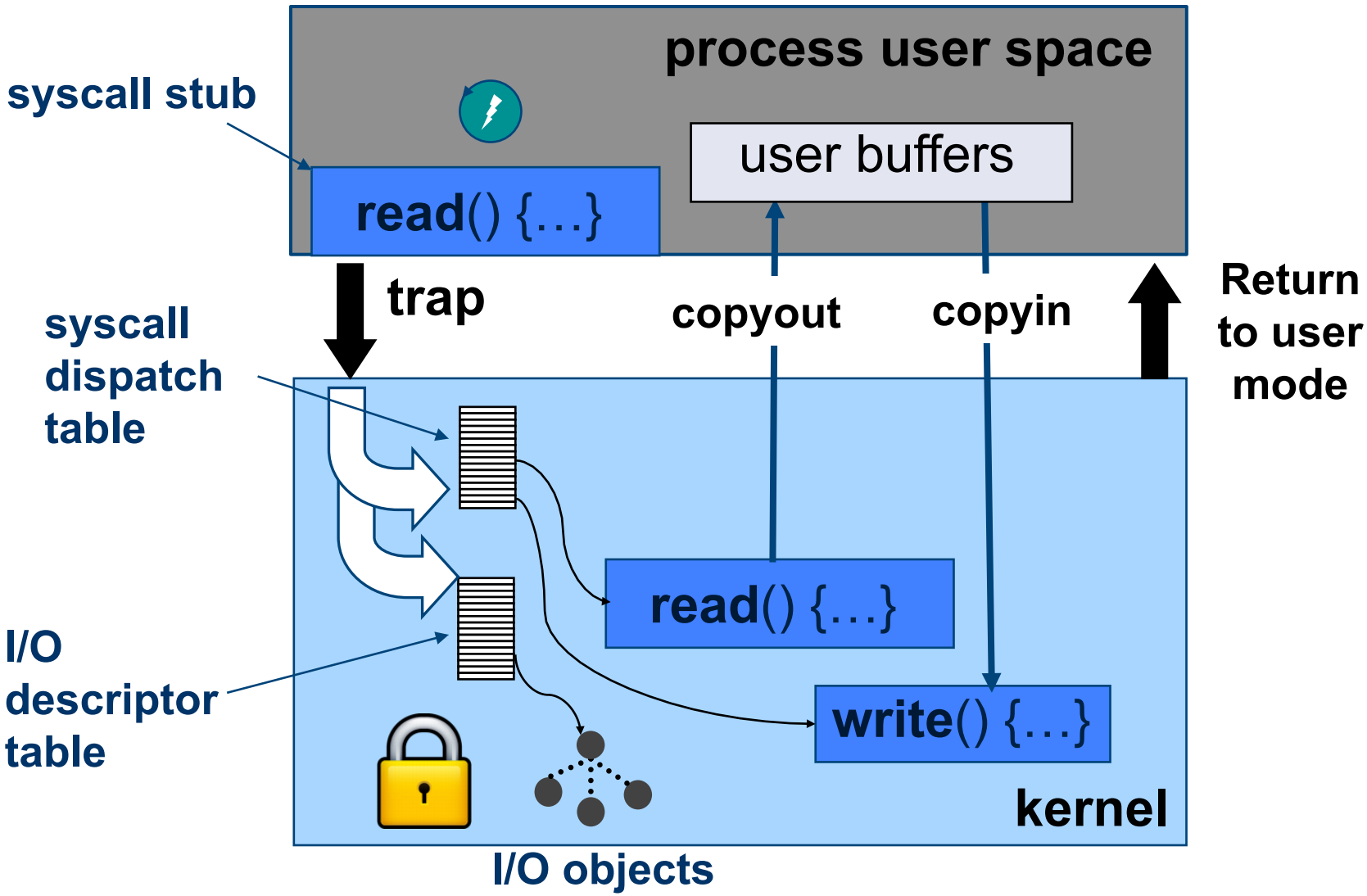

user
space
Safe
control
transfer
kernel code
kernel
space
kernel
data

The kernel

- The kernel is just a program: a collection of linked modules and their state (data).
- E.g., the kernel may be written in C and compiled/linked a little differently.
  - E.g., linked with –static option: no dynamic libs
- At runtime, kernel resides in a protected range of virtual addresses: **kernel space.**
  - It is (or may be) "part of" every VAS, but **protected**.
    - (Details vary by machine and OS configuration)
  - Access to kernel space is denied for user programs.
  - Portions of kernel space may be non-pageable and/or direct-mapped to machine memory.



Process, kernel, and syscalls

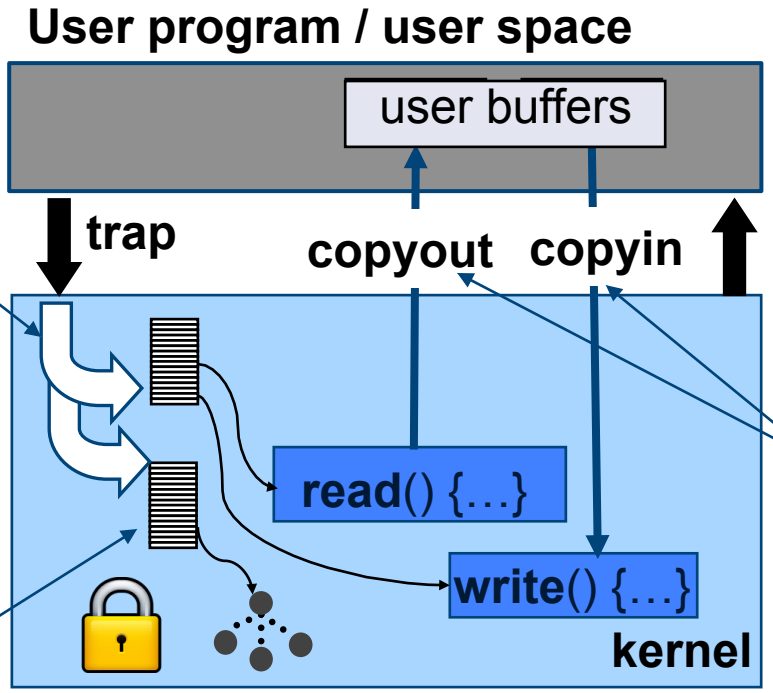


The kernel must be bulletproof

Secure kernels handle system calls verrry carefully.

Syscalls indirect through **syscall dispatch table** by syscall number. No direct calls to kernel routines from user space!

What about references to kernel data objects passed as syscall arguments (e.g., file to read or write)?



Kernel copies all arguments into kernel space and validates them.

Kernel interprets pointer arguments in context of the user VAS, and copies the data in/out of kernel space (e.g., for **read** and **write** syscalls).

Use an integer index into a kernel table that points at the data object. The value is called a **handle** or **descriptor**. No direct pointers to kernel data from user space!

Channels

2.2 Contract-Based Channels

- All communication between SIPs in Singularity flows through *contract-based channels*…
- A channel provides a lossless, in-order message conduit with exactly two endpoints. Semantically, each endpoint has a receive queue. Sending on an endpoint enqueues a message on the other endpoint's receive queue.
- Sends are non-blocking and non-failing. Receives block synchronously until a message arrives or the send endpoint is closed.

Threads own endpoints

A channel **endpoint** belongs to exactly one **thread** at a time. Only the endpoint's owning thread can dequeue messages from its receive queue or send messages to its peer.

How do threads get endpoints?

How do threads get endpoints?

The call to create a child SIP specifies the child's manifest (identifying of all code allowed to run in the child) and gives an initial set of channel endpoints before the child SIP begins execution.

Channel endpoints can be sent in messages. Thus, a communication network can evolve dynamically while conforming to the explicit communication invariant.

Unix Pipes

Example: cat | cat

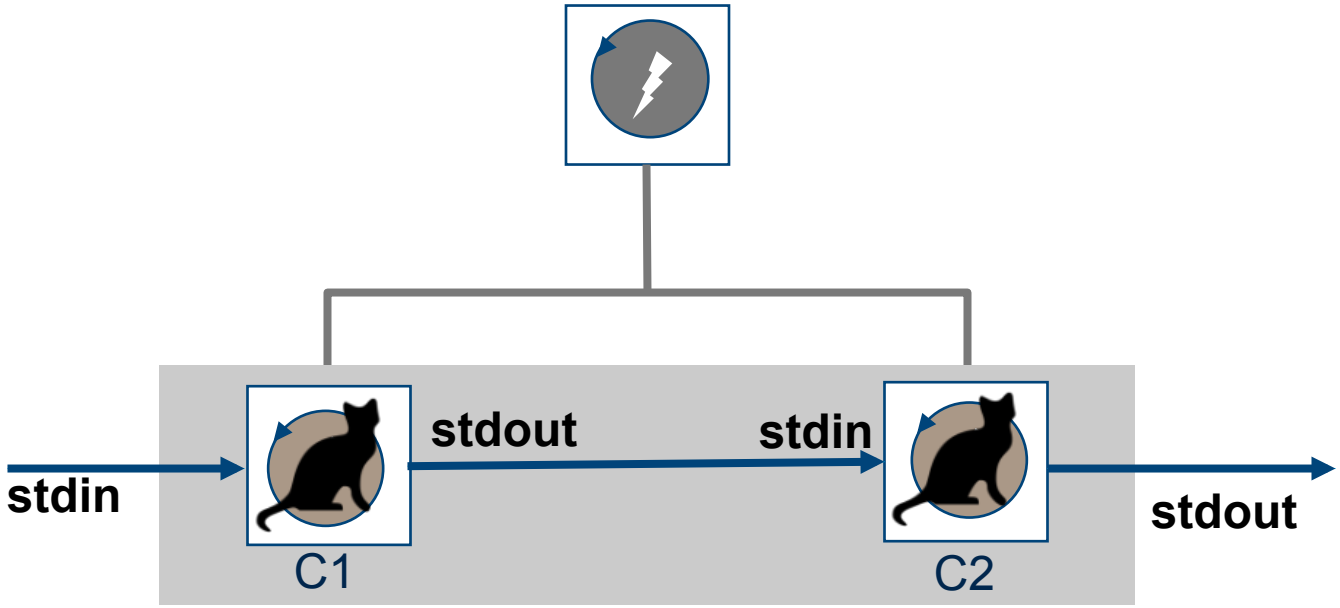
cat pseudocode (user mode)

```
while(until EOF) {
        read(0, buf, count);
        compute/transform data in buf;
        write(1, buf, count);
}
```

Kernel pseudocode for pipes: Producer/consumer bounded buffer

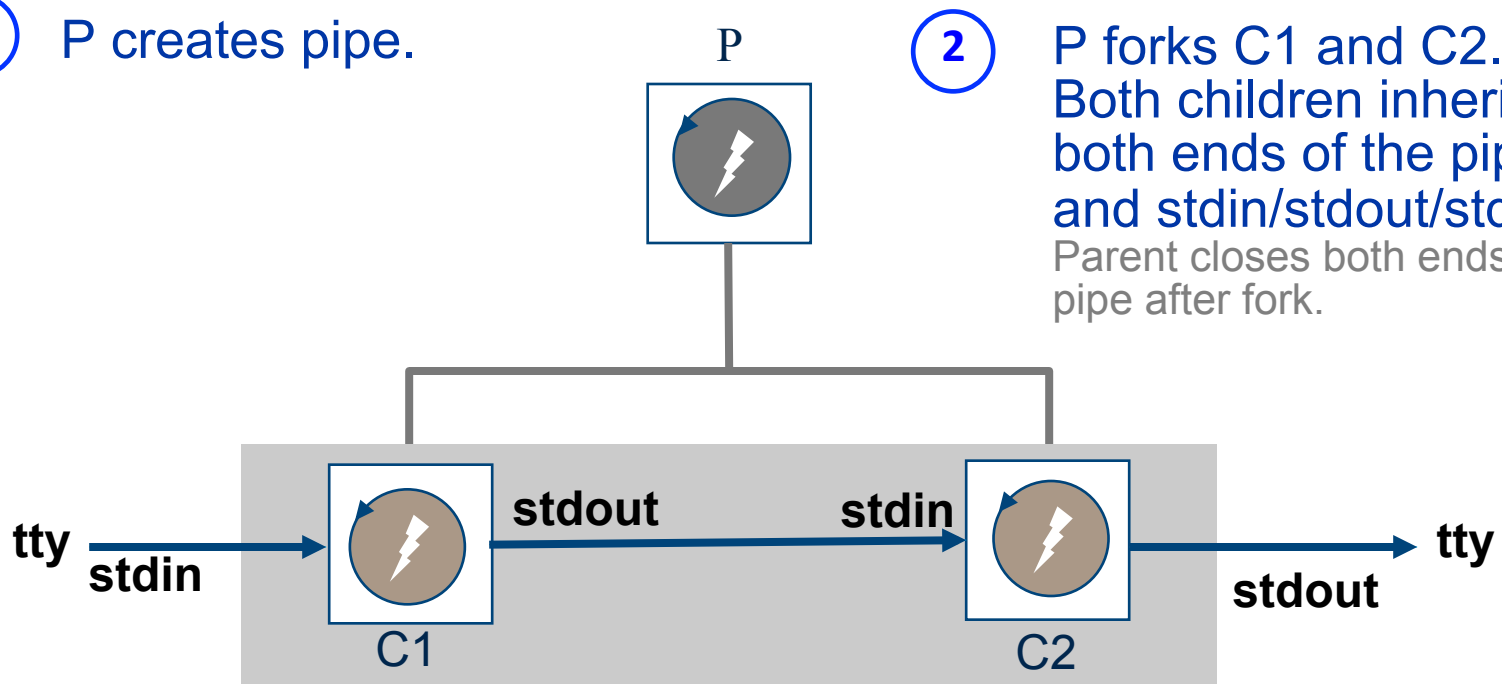
Pipe write: copy in bytes from user buffer to in-kernel pipe buffer, blocking if k-buffer is full.

Pipe read: copy bytes from pipe's k-buffer out to u-buffer. Block while k-buffer is empty, or return EOF if empty and pipe has no writer.



How to plumb the pipe?

1 P creates pipe.



2 P forks C1 and C2. Both children inherit both ends of the pipe, and stdin/stdout/stderr.
Parent closes both ends of pipe after fork.

3A C1 closes the read end of the pipe, closes its stdout, "dups" the write end onto stdout, and execs.

3B C2 closes the write end of the pipe, closes its stdin, "dups" the read end onto stdin, and execs.

Manifest

SIPs are created from a signed manifest [39]. The manifest describes the SIP's code, resources, and dependencies on the kernel and on other SIPs. All code within a SIP must be listed in the manifest. Singularity SIP manifests are entirely declarative. They describe the desired state of the application configuration after an installation, not the algorithm for installing the application.

Channel contracts

Communication across a channel is described by a **channel contract**…
Channel contracts are declared in the Sing# language.

The two ends of a channel are not symmetric in a contract. One is the importing end (Imp) and the other is the exporting end (Exp).

A contract consists of message declarations and a set of named protocol states. Message declarations state the number and types of arguments for each message and an optional message direction. Each state specifies the possible message sequences leading to other states in the state machine.

```
contract NicEvents {
  enum NicEventType {
    NoEvent, ReceiveEvent, TransmitEvent, LinkEvent
  }

  out message NicEvent(NicEventType e);
  in message AckEvent();

  state READY: one {
    NicEvent! → AckEvent? !READY;
  }
}
```

Listing 2. Contract for network device events.

```
contract NicDevice {
  out message DeviceInfo(...);
  in  message RegisterForEvents(NicEvents.Exp:READY c);
  in  message SetParameters(...);
  out message InvalidParameters(...);
  out message Success();
  in  message StartIO();
  in  message ConfigureIO();
  in  message PacketForReceive(byte[] in ExHeap p);
  out message BadPacketSize(byte[] in ExHeap p, int m);
  in  message GetReceivedPacket();
  out message ReceivedPacket(Packet * in ExHeap p);
  out message NoPacket();

  state START: one {
    DeviceInfo! → IO_CONFIGURE_BEGIN;
  }
  state IO_CONFIGURE_BEGIN: one {
    RegisterForEvents? →
      SetParameters? → IO_CONFIGURE_ACK;
  }
  state IO_CONFIGURE_ACK: one {
    InvalidParameters! → IO_CONFIGURE_BEGIN;
    Success! → IO_CONFIGURED;
  }
  state IO_CONFIGURED: one {
    StartIO? → IO_RUNNING;
    ConfigureIO? → IO_CONFIGURE_BEGIN;
  }
  state IO_RUNNING: one {
    PacketForReceive? → (Success! or BadPacketSize!)
      → IO_RUNNING;
    GetReceivedPacket? → (ReceivedPacket! or NoPacket!)
      → IO_RUNNING;
    ...
  }
}
```

Listing 1. Contract to access a network device driver.

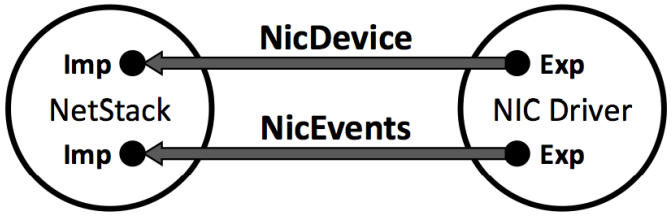


Figure 2. Channels between a network driver and stack.

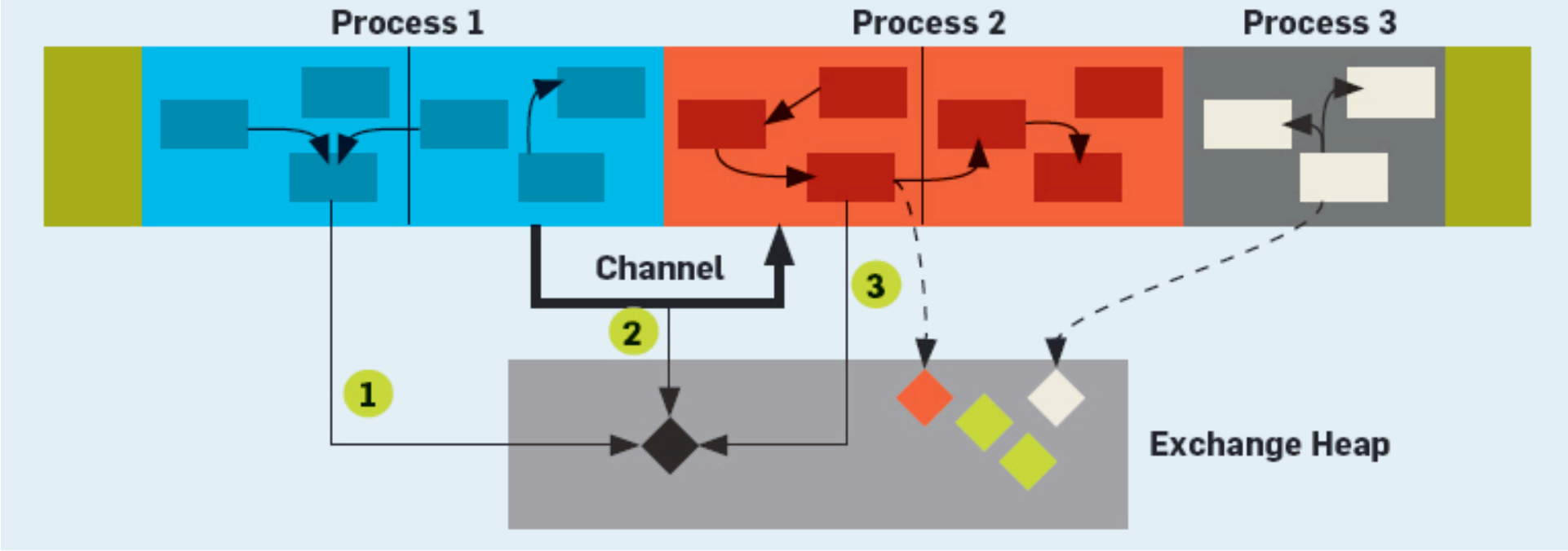
Verifying IPC

…Channels enable efficient and analyzable communication between SIPs. …the compiler can statically verify that send and receive operations on channels never are applied in the wrong protocol state. A separate contract verifier can read a program's byte code and statically verify which contracts are used within a program and that the code conforms to the state machine described in the contract's protocol.

The Exchange Heap

Endpoints and message data reside in a special set of pages known as the **Exchange Heap**.... Messages are tagged collections of values or message blocks in the Exchange Heap.

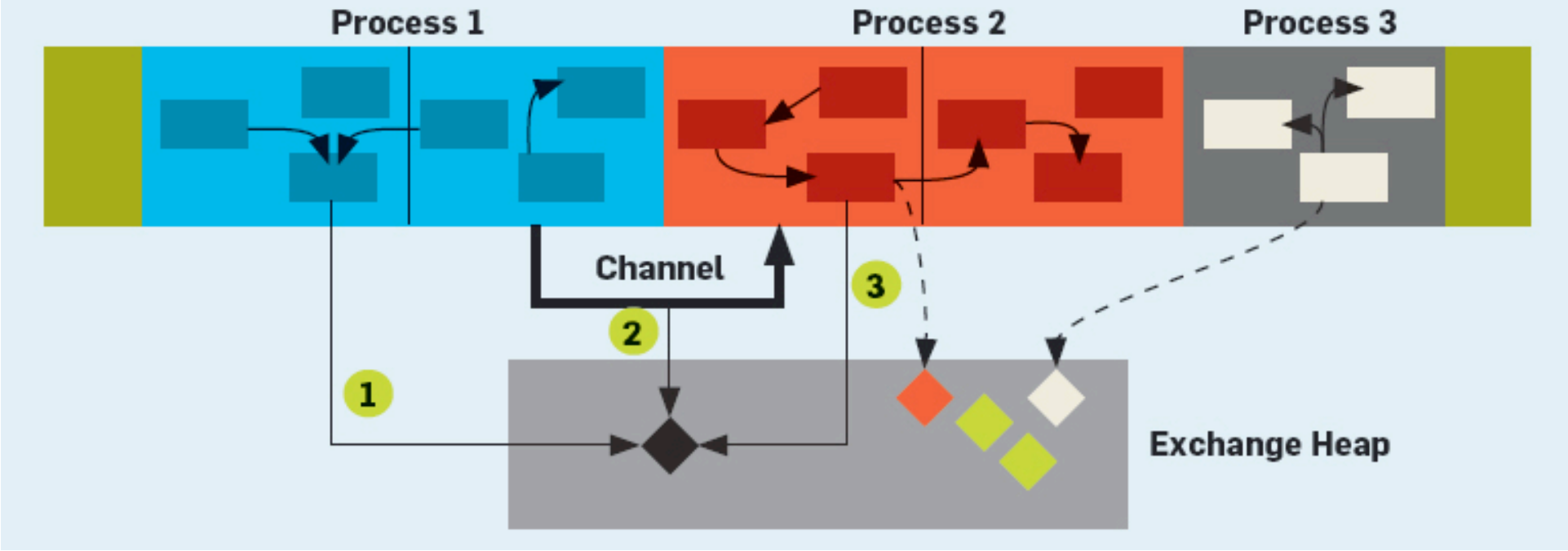
SIPs can contain pointers into their own heap and into the exchange heap. The exchange heap only holds pointers into the exchange heap itself.



Sending a message

A message sender passes ownership by storing a pointer to the message in the receiving endpoint, at a location determined by the current state of the message exchange protocol.

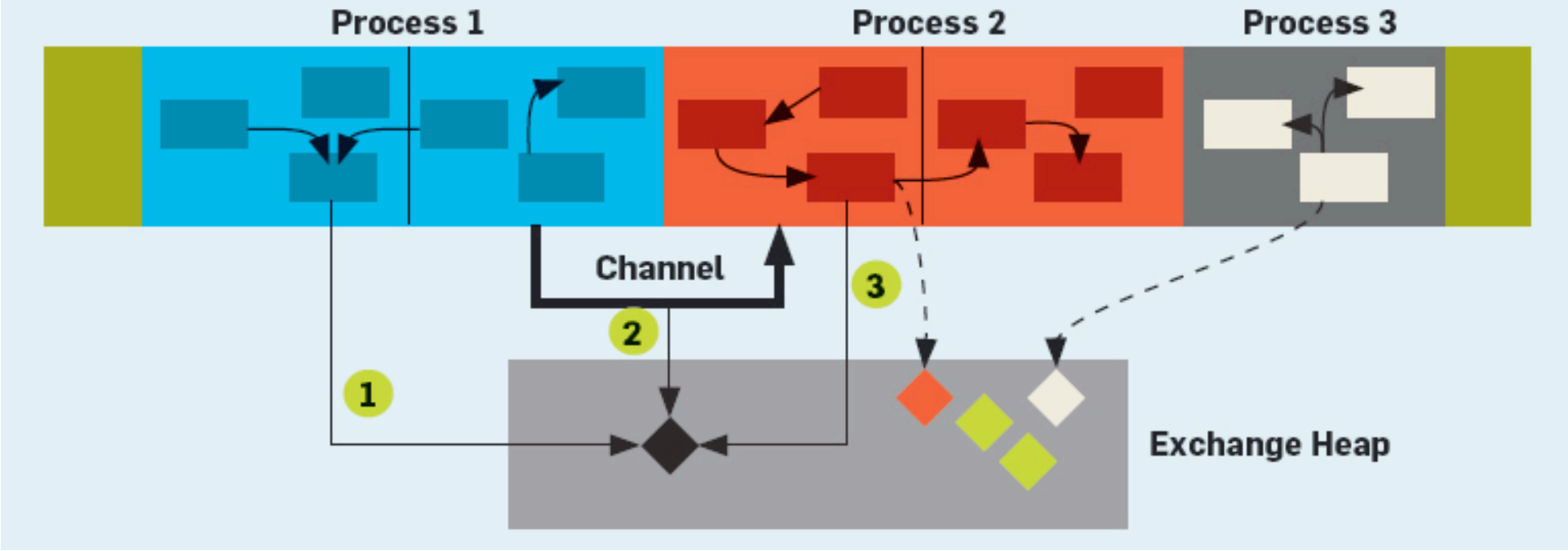
The sender then notifies the scheduler if the receiving thread is blocked awaiting to receive a message.



The Exchange Heap

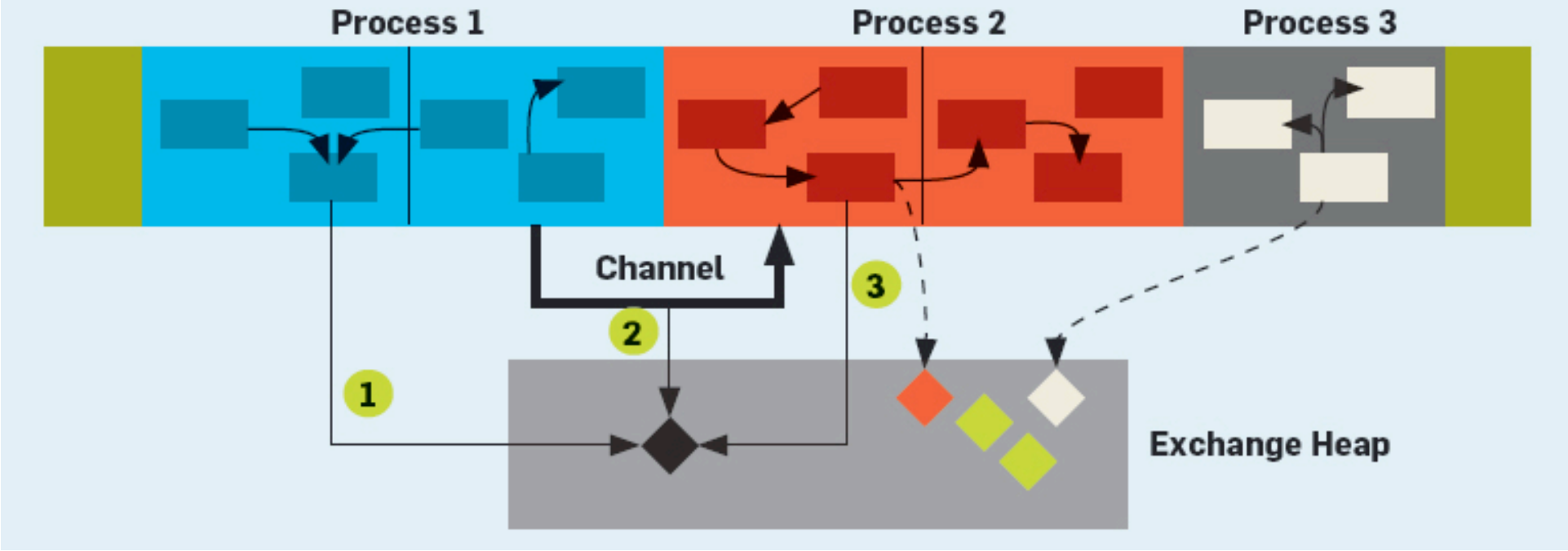
The operation of sending and receiving, as opposed to creating a message, entails no memory allocation.

Do we still need to copy the data?



Zero-copy I/O

Pre-allocating endpoint queues and passing pointers to exchange heap memory naturally allow **zero copy** implementations of multi-SIP subsystems, such as the I/O stack. For example, disk buffers and network packets can be transferred across multiple channels, through a protocol stack and into an application SIP, without copying.



The Exchange Heap is not garbage collected

It uses reference counts to track usage of blocks of memory.

Every block of memory in the exchange heap is owned (accessible) by at most one SIP at any time during the execution of the system. When data ...is sent over a channel, ownership passes from the sending SIP...to the receiving SIP.

Ownership of a block can only be transferred to another SIP by sending it in a message across a channel. Singularity ensures that a SIP does not access a block after it has sent it in a message.

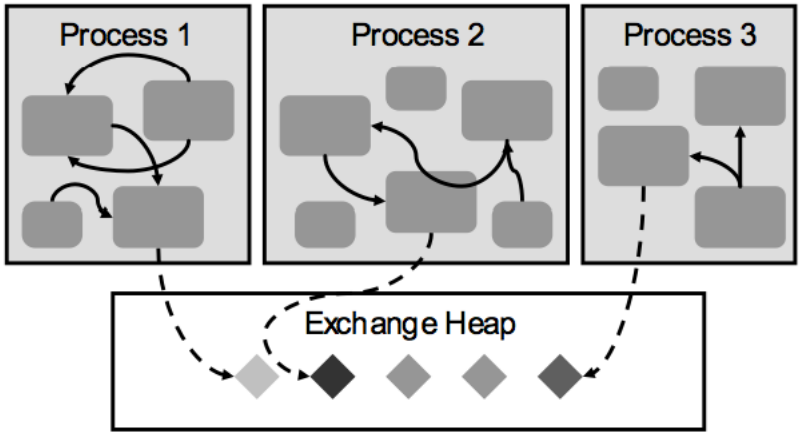
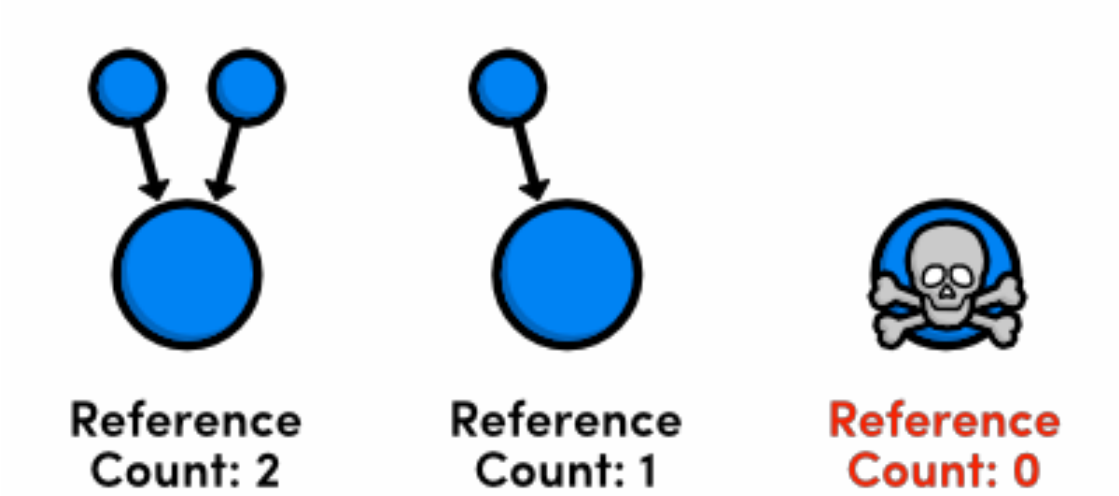


Figure 2. The Exchange Heap.

Reference counting

Used in various applications and programming language environments, and in the kernel, e.g., Unix file management.

- Keep a count of references to the object.
- Increment count when a new reference is created (shallow copy).
- Decrement count when a reference is destroyed.
- Free object when the count goes to zero.



[http://rypress.com/tutorials/objective-c/memory-management.html]

3.6 Principals and Access Control

In Singularity, applications are security principals. More precisely, principals are compound in the sense of Lampson *et al.* [16, 29]: they reflect the application identity of the current SIP, an optional role in which the application is running, and an optional chain of principals through which the application was invoked or given delegated authority. Users, in the traditional sense, are roles of applications (for example, the system login program running in the role of the logged in user). Application names are derived from MBP manifests which in turn carry the name and signature of the application publisher.

…

Security principals

Most systems control access to data based on the identity of an authenticated user [51].

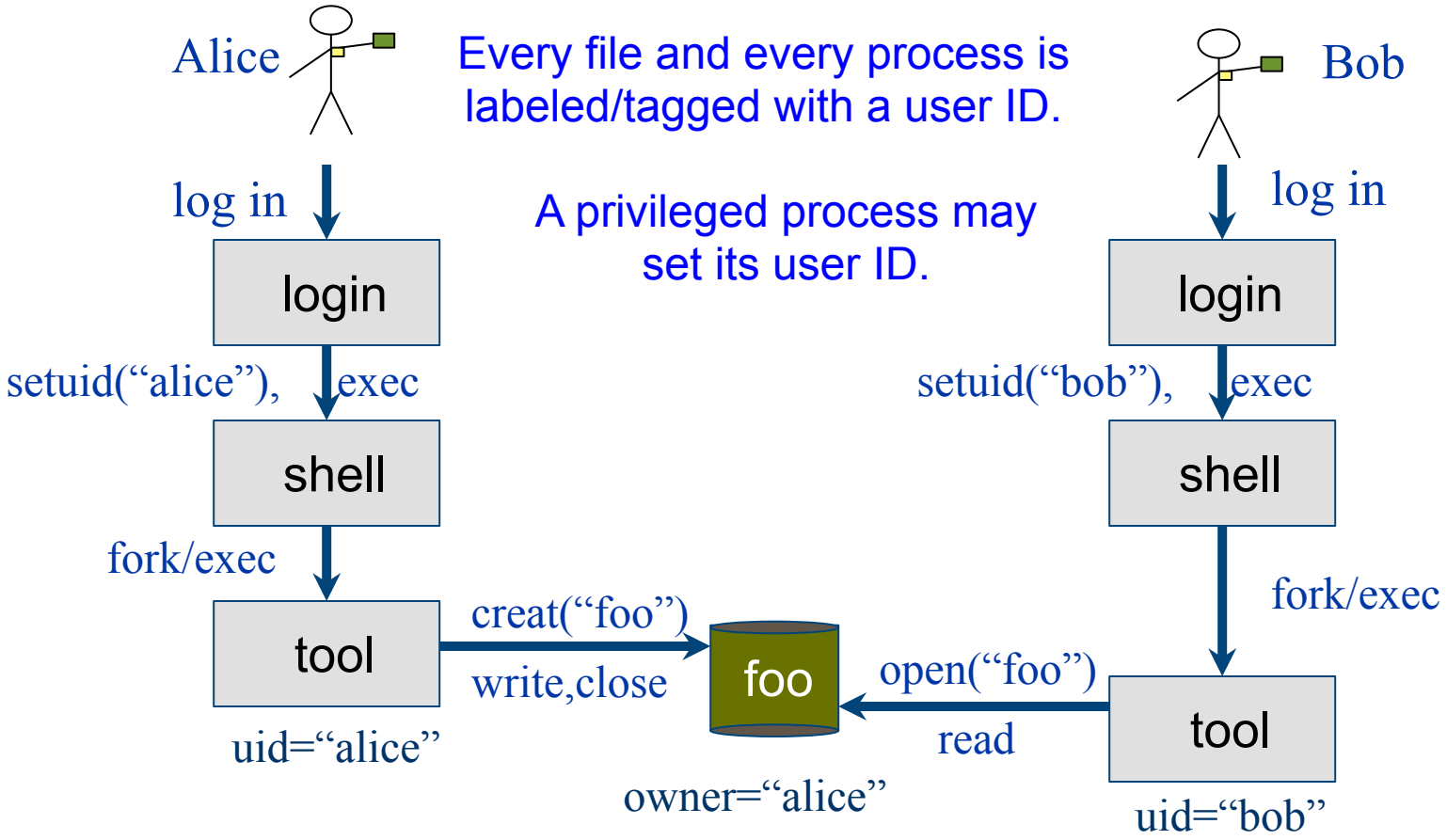
When the code loaded in a process is known a priori, its identity can be verified against certificates provided by the code's publisher. This identity makes it is possible to assign access rights to the process [51]. For example, a process might have a security principal consisting of its authenticated user, authenticated program, and a publisher. Making a program part of a security principal enables the system to limit access to the application's data files to the application itself or its trusted peers.

With a sealed process, the certification of disk contents can be extended to the executable contents of a process. A sealed architecture can guarantee that a program will execute in its certified form because of the state isolation and closed API invariants....

When coupled with hardware support for attestation [46] sealed processes can enable an execution model in which a process is a trustable entity.

Extra slides from CPS 310

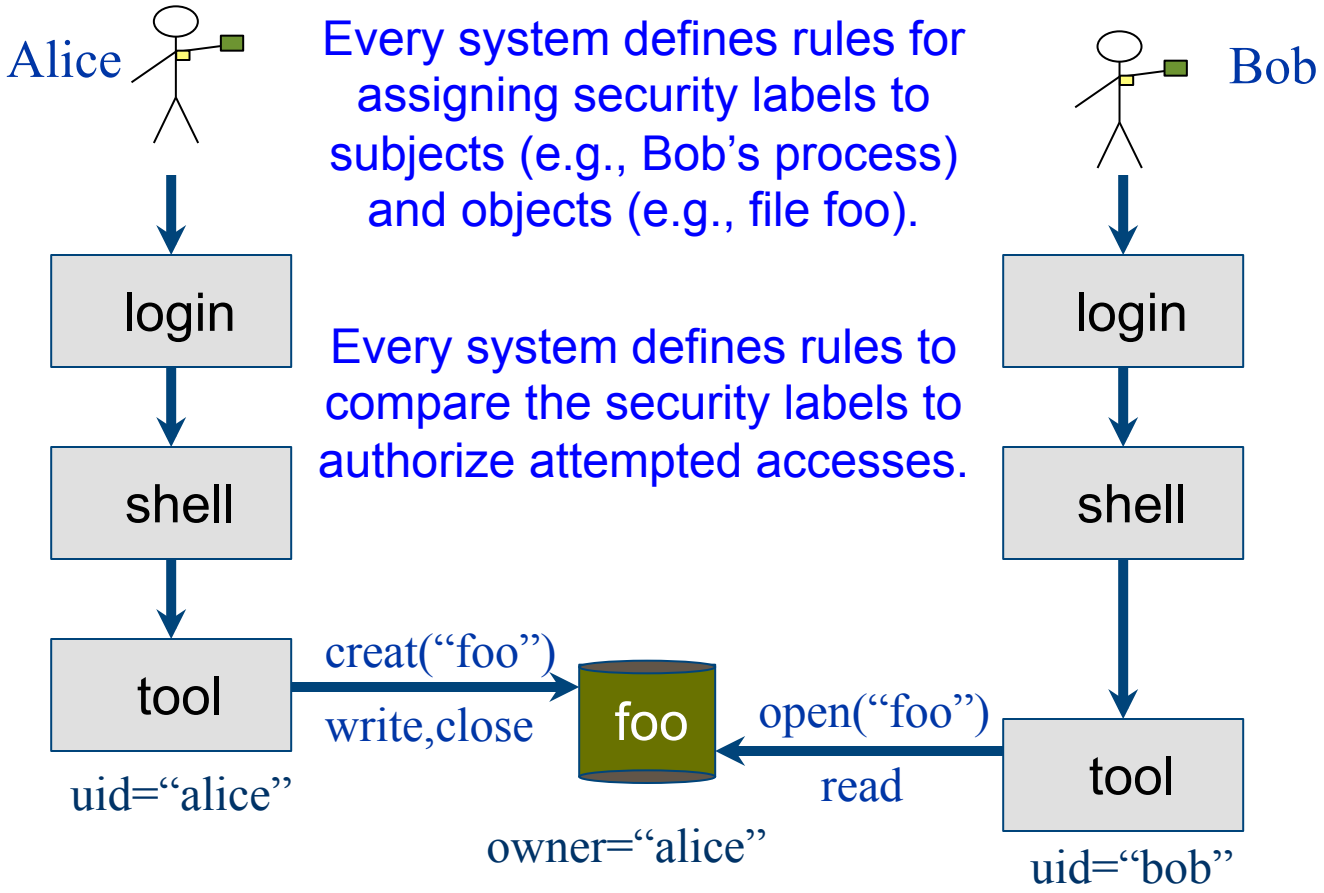
Labels and access control



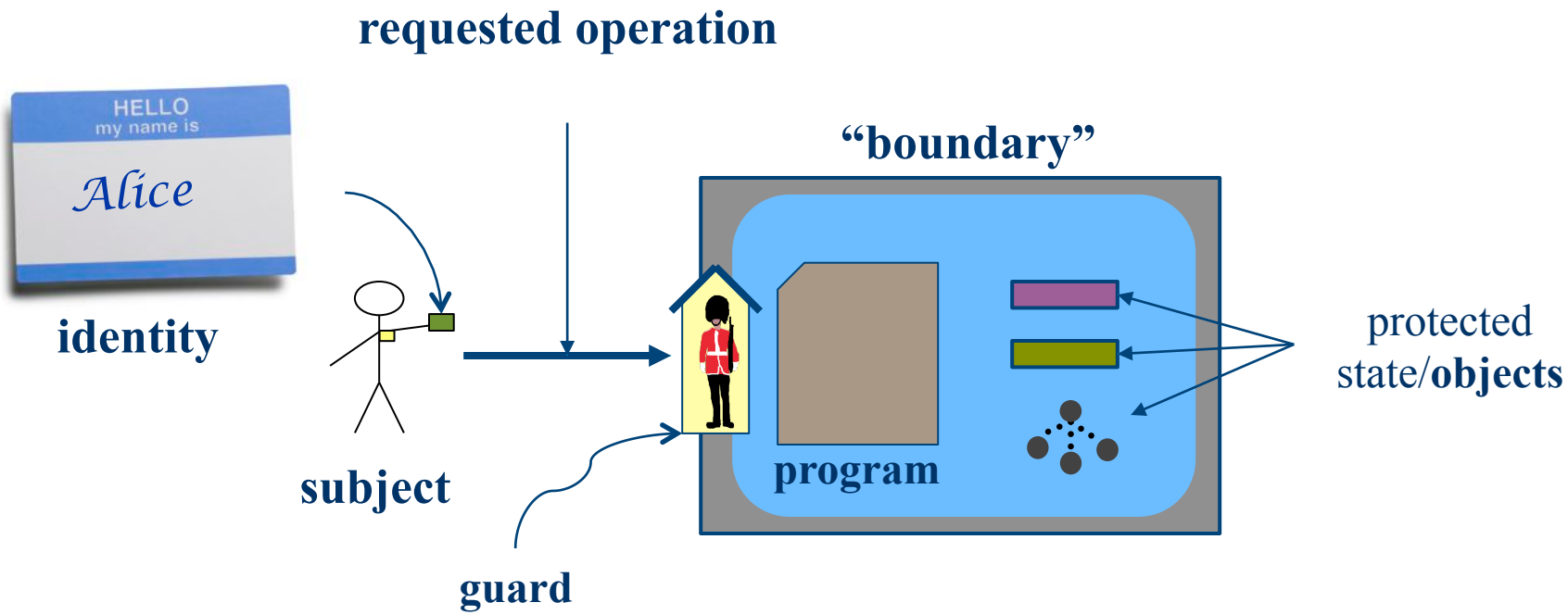
A process inherits its userID from its parent process.

A file inherits its owner userID from its creating process.

Labels and access control

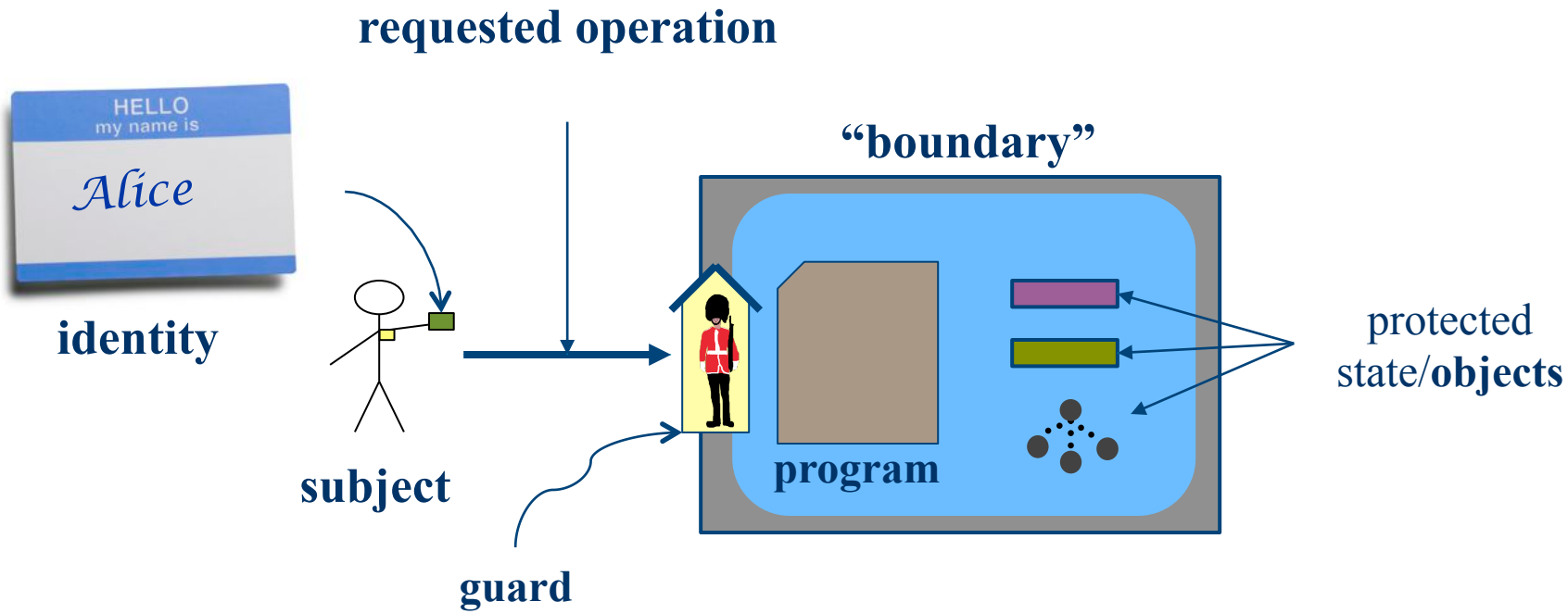


Concept: reference monitor



A **reference monitor** is a program that controls access to a set of objects by other programs. The reference monitor has a **guard** that checks all requests against an **access control policy** before permitting them to execute.

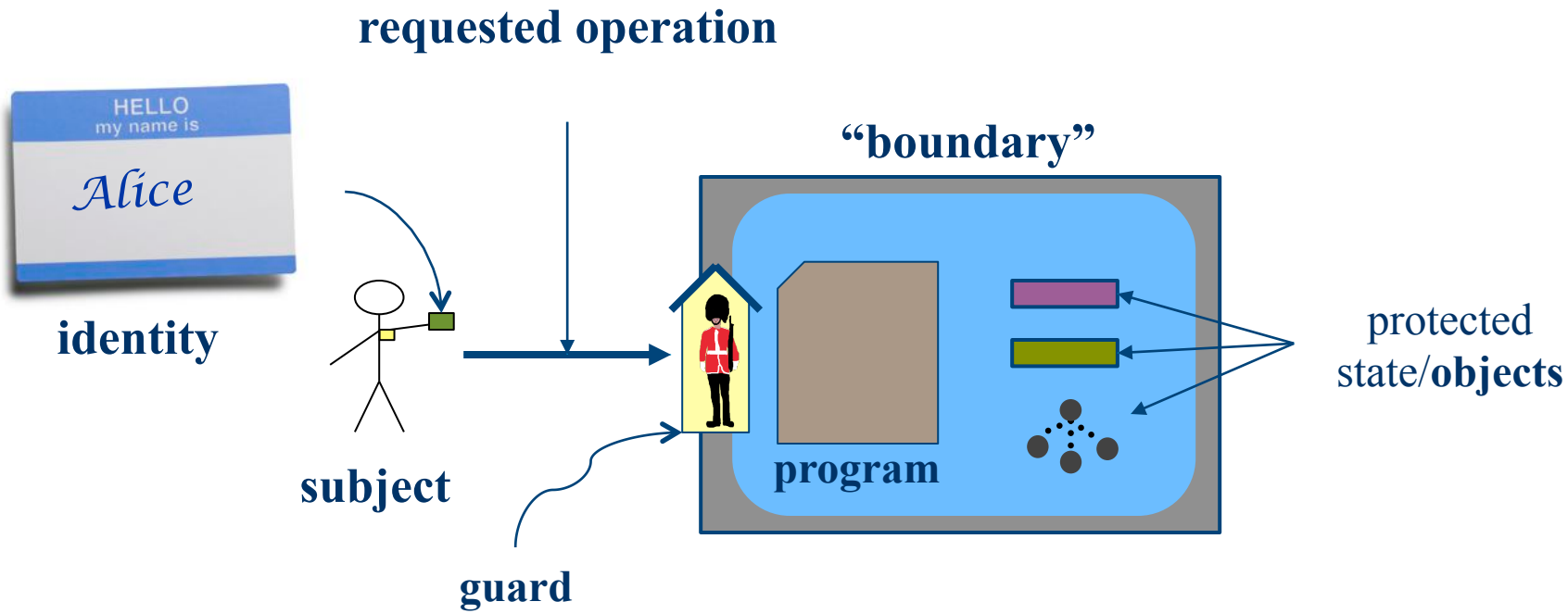
Reference monitor



What is the nature of the isolation boundary?

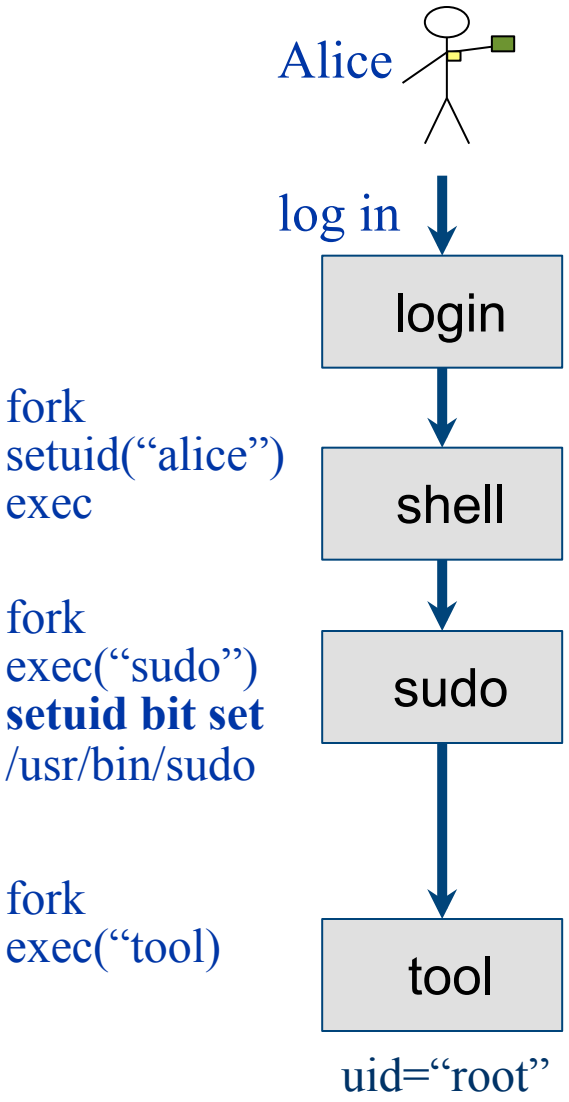
If we're going to post a guard, there should also be a wall.
Otherwise somebody can just walk in past the guard, right?

Reference monitor



How does the guard decide whether or not to allow access?
We need some way to represent **access control policy**.

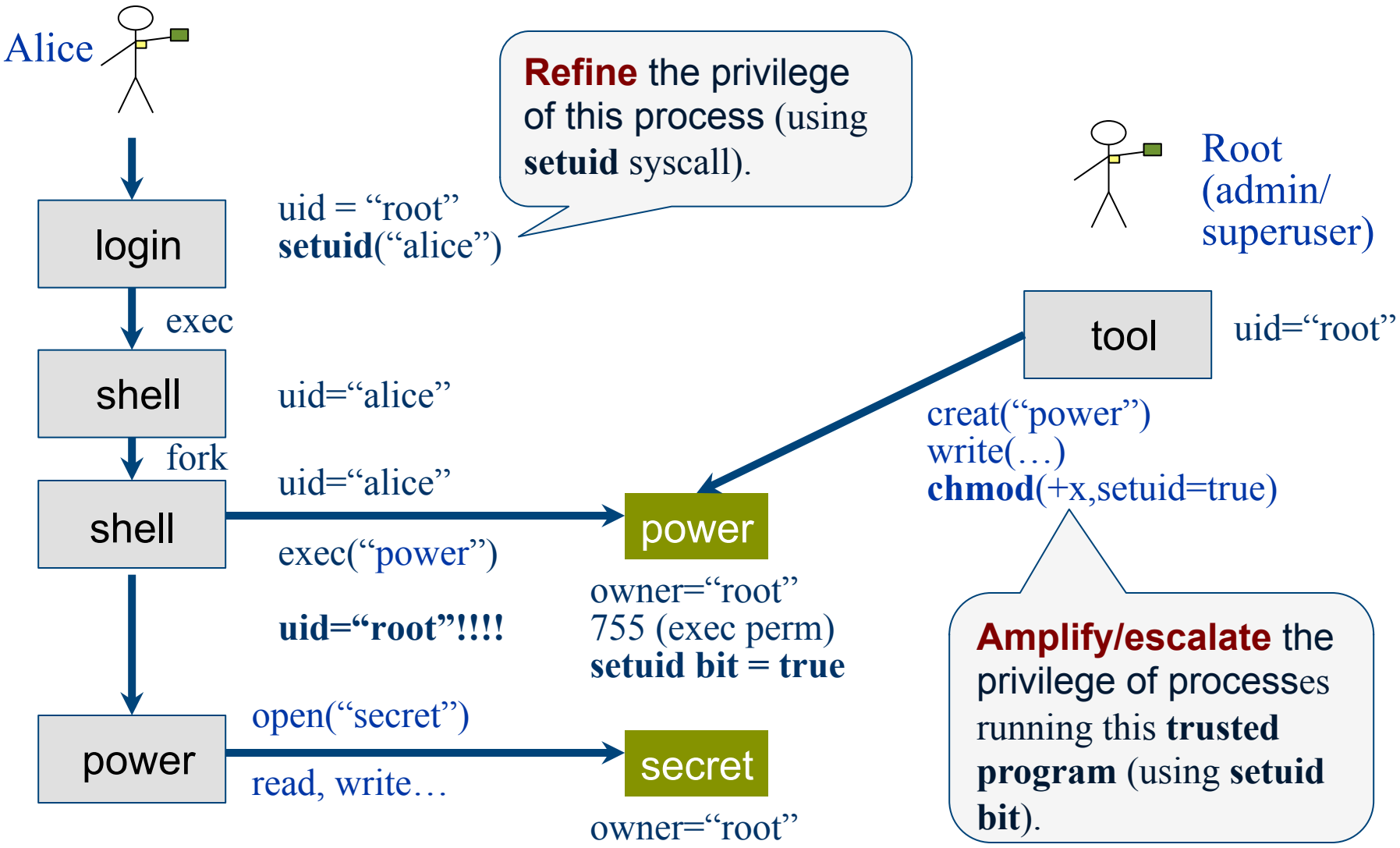
Another way to setuid



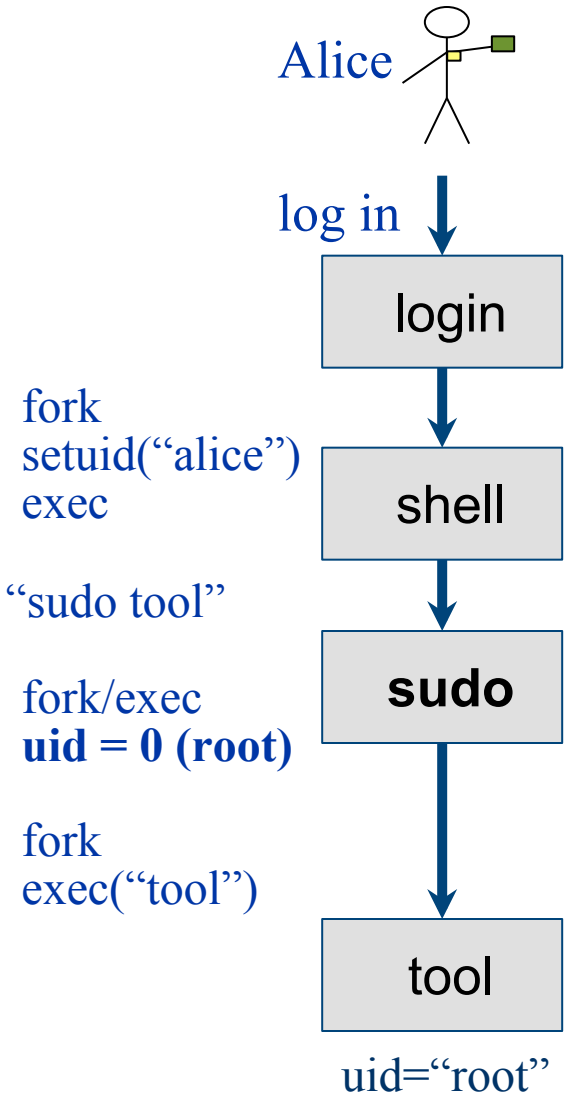
- The mode bits on a program file may be tagged to setuid to owner's uid on **exec***.
- This is called **setuid bit**. Some call it the most important innovation of Unix.
- It enables users to request protected ops by running **secure programs**.
- The user cannot choose the code: only the program owner can choose the code to run with the owner's uid.
- Parent cannot subvert/control a child after program launch: a property of Unix **exec***.

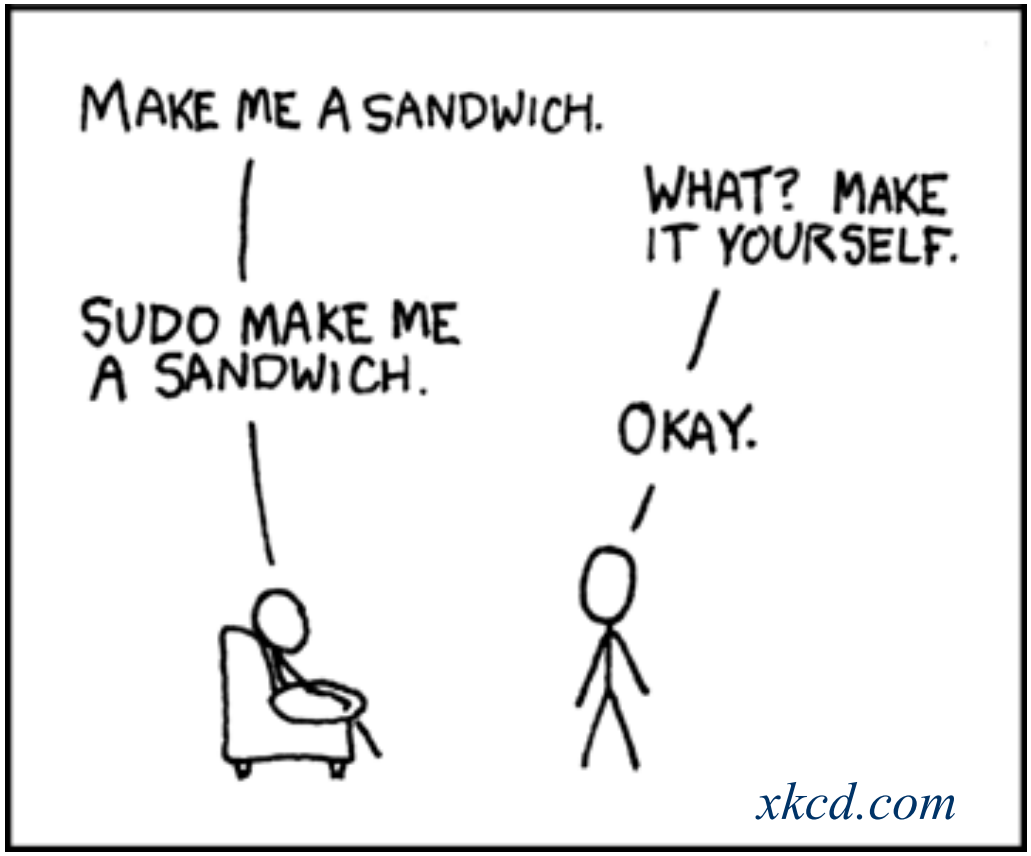
Example: **sudo** program runs as root, checks user authorization to act as superuser, and (if allowed) executes requested command as root.

Unix setuid: recap



A trusted program: sudo





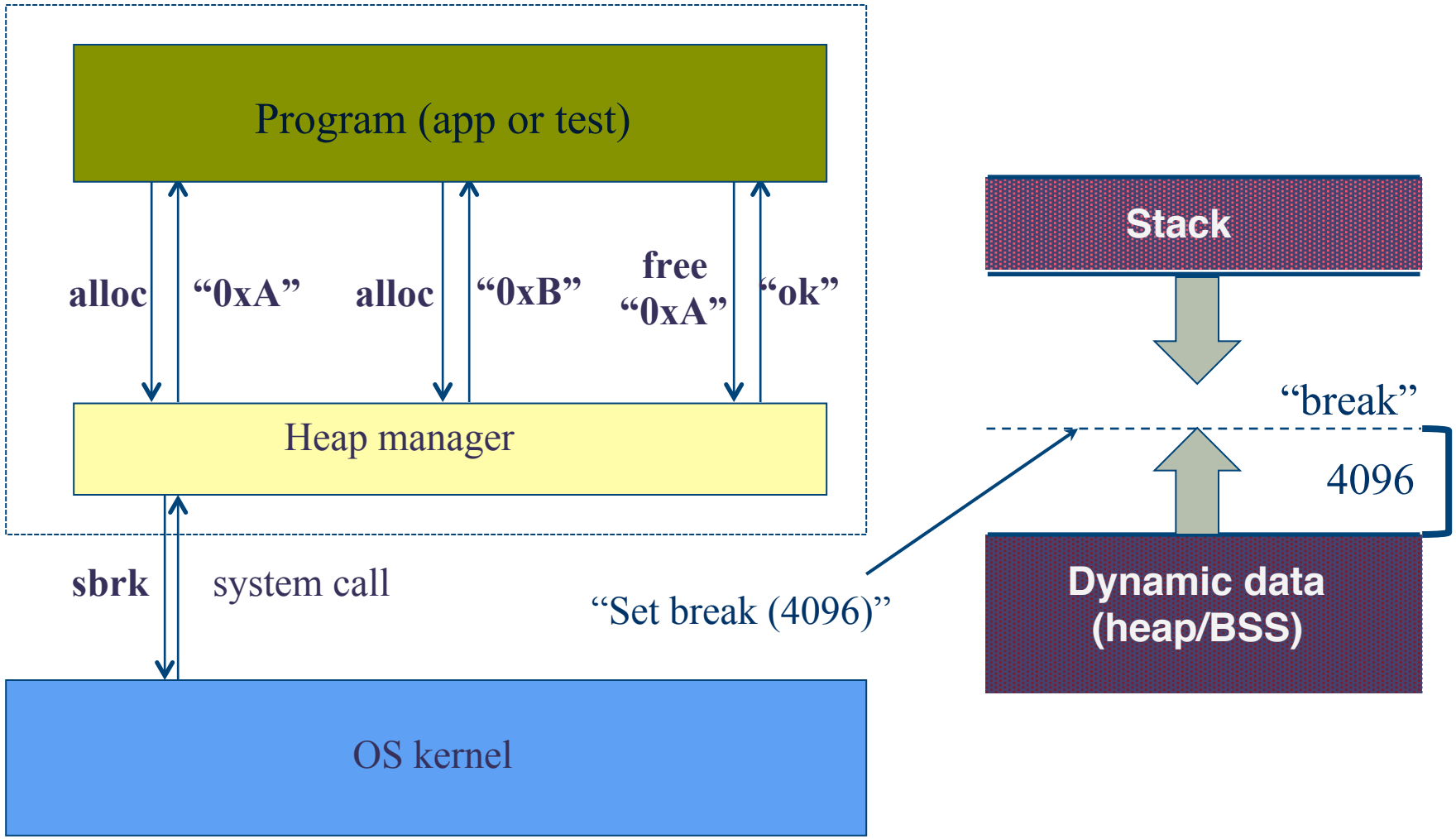
With great power comes great responsibility.

Example: **sudo** program runs as root, checks user authorization to act as superuser, and (if allowed) executes requested command as root.

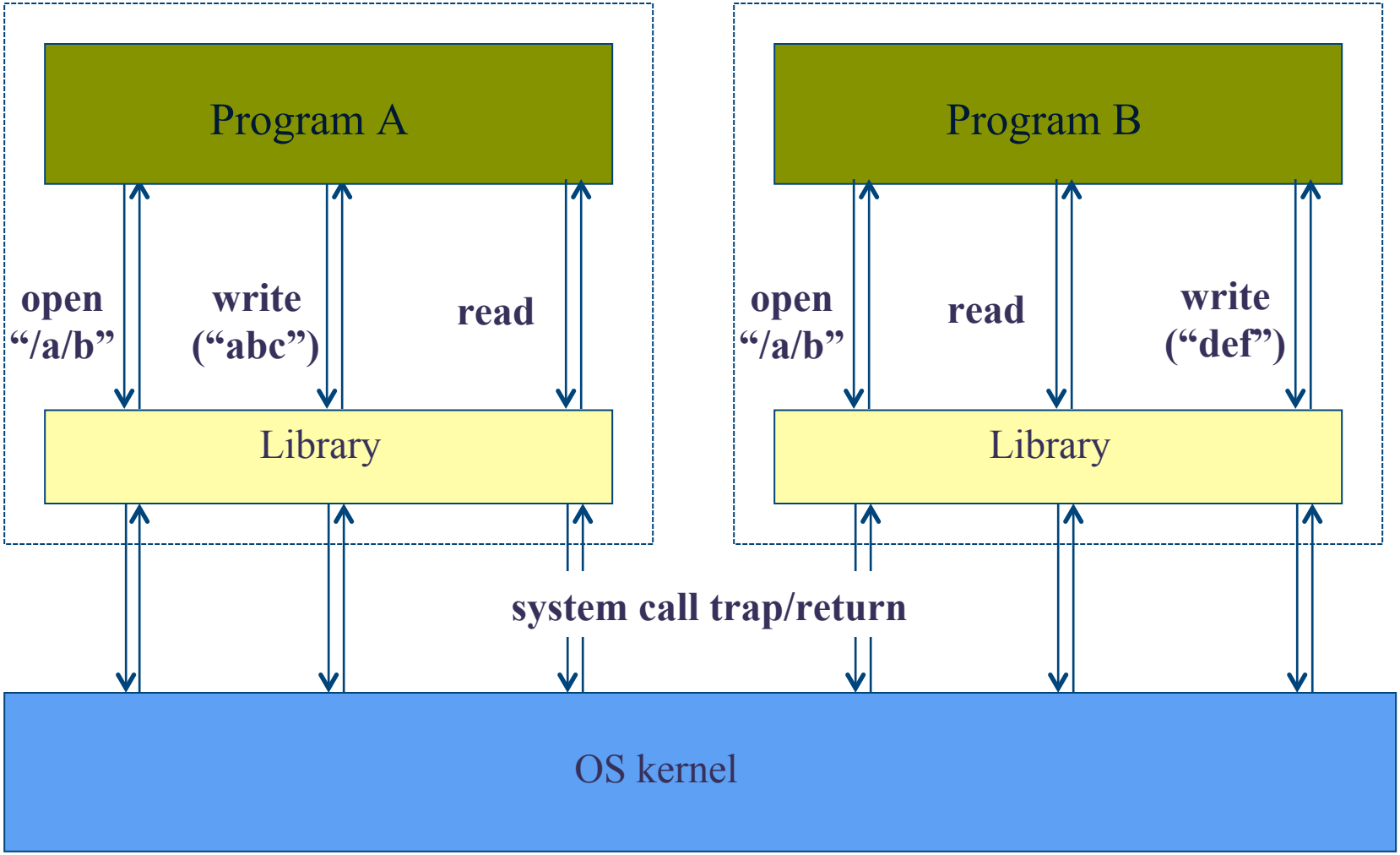
The secret of setuid

- The **setuid bit** can be seen as a mechanism for a trusted program to function as a **reference monitor**.
- E.g., a **trusted program** can govern access to files.
  - Protect the files with the program owner's uid.
  - Make them inaccessible to other users.
  - Other users can access them via the trusted program.
  - **But only in the manner permitted by the trusted program.**
  - Example: "moo accounting problem" in 1974 paper (cryptic)
- What is the reference monitor's "isolation boundary"? What protects it from its callers?

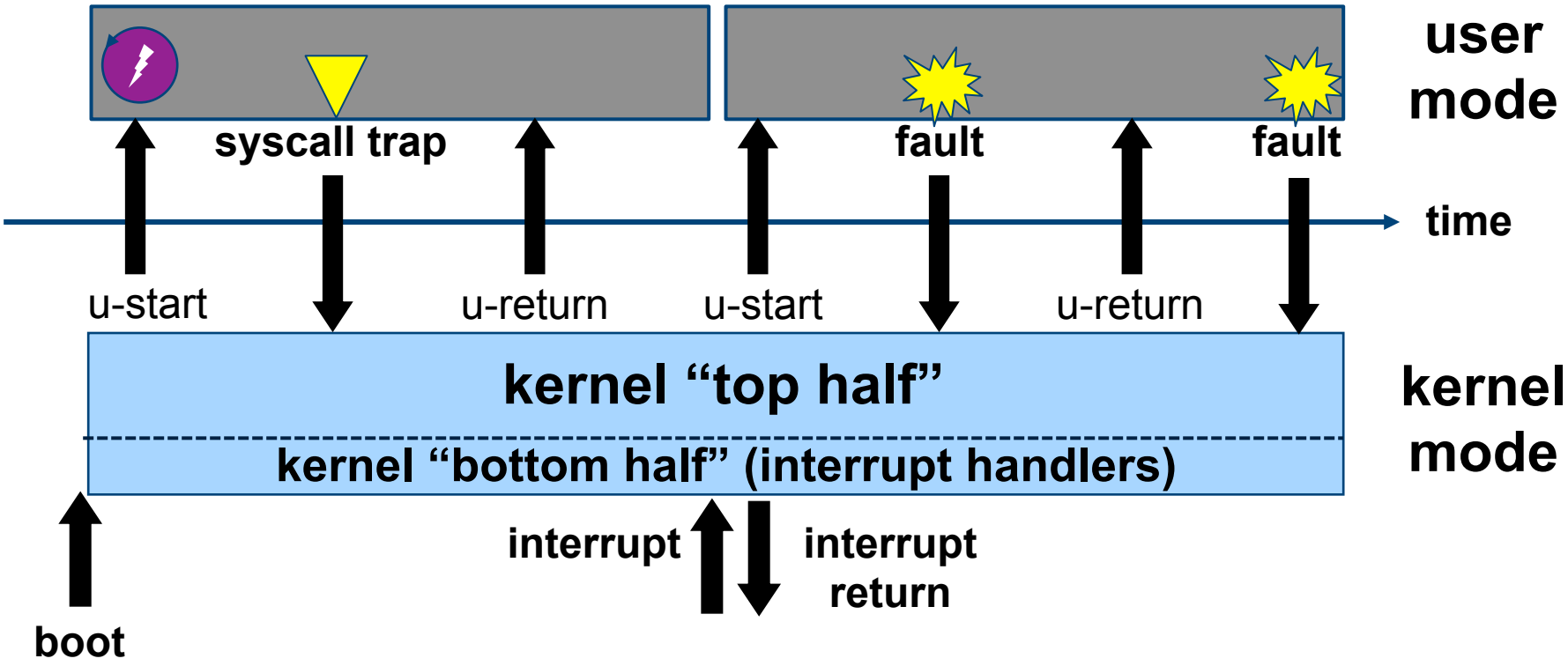
Heap manager



File abstraction



“Limited direct execution”



User code runs on a CPU core in user mode in a user space. If it tries to do anything weird, the core transitions to the kernel, which takes over.

The kernel executes a special instruction to **transition to user mode** (labeled as “u-return”), with selected values in CPU registers.

Linux x64 syscall conventions (ABI)

1. User-level applications use as integer registers for passing the sequence `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9`. The kernel interface uses `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8` and `%r9`.

2. A system-call is done via the `syscall` instruction. The kernel destroys registers `%rcx` and `%r11`.

3. The number of the syscall has to be passed in register `%rax`.

4. System-calls are limited to six arguments, no argument is passed directly on the stack.

5. Returning from the `syscall`, register `%rax` contains the result of the system-call. A value in the range between -4095 and -1 indicates an error, it is `-errno`.

(user buffer addresses)

6. Only values of class INTEGER or class MEMORY are passed to the kernel.

Illustration only: the details aren't important.

MacOS x86-64 syscall example

```
section .data
hello_world     db      "Hello World!", 0x0a

section .text
global start

start:
mov rax, 0x2000004      ; System call write = 4
mov rdi, 1              ; Write to standard out = 1
mov rsi, hello_world    ; The address of hello_world string
mov rdx, 14             ; The size to write
syscall                 ; Invoke the kernel
mov rax, 0x2000001      ; System call number for exit = 1
mov rdi, 0              ; Exit success = 0
syscall                 ; Invoke the kernel
```

Illustration only: this program writes "Hello World!" to standard output (fd == 1), ignores the syscall error return, and exits.

http://thexploit.com/secdev/mac-os-x-64-bit-assembly-system-calls/