Boston University, Systems Group, 2008

On the Design and Implementation of Mutable Protection Domains Towards Reliable Component-based Systems

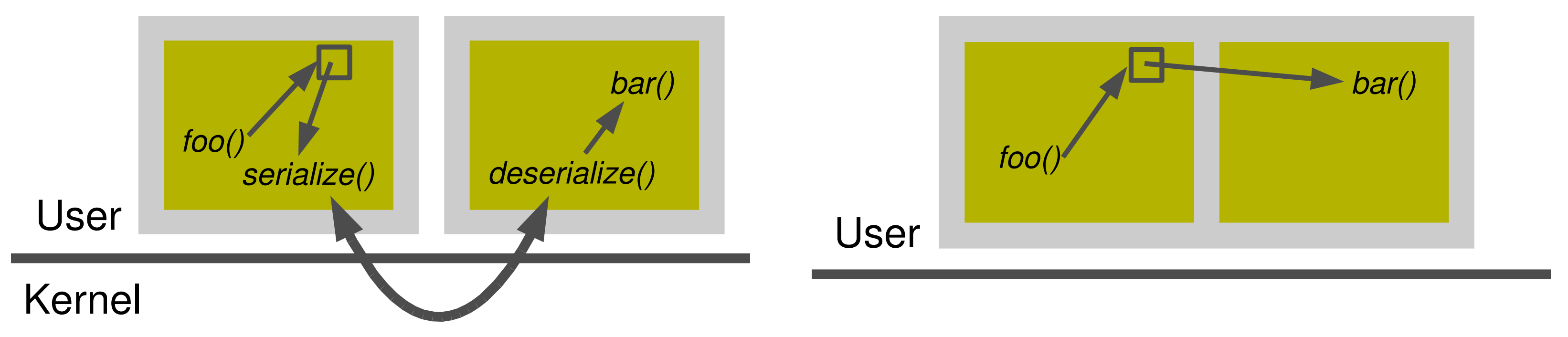
Gabriel Parmer and Richard West

Component-based systems

- A component is a reusable software unit of composition
  - Contractually specified interfaces
  - Explicit interface context dependencies
  - Independently deployable
- Abstraction via separation of implementation and interface
- Smaller/more specific components allow more compositional flexibility

Why Mutable Protection Domains (MPD)?

- Systems often go through phases of different distributions of communication between components
- Static placement of protection domains allows either
  - good performance **or** pervasive fault tolerance
- Dynamic placement: performance **and** fault tolerance

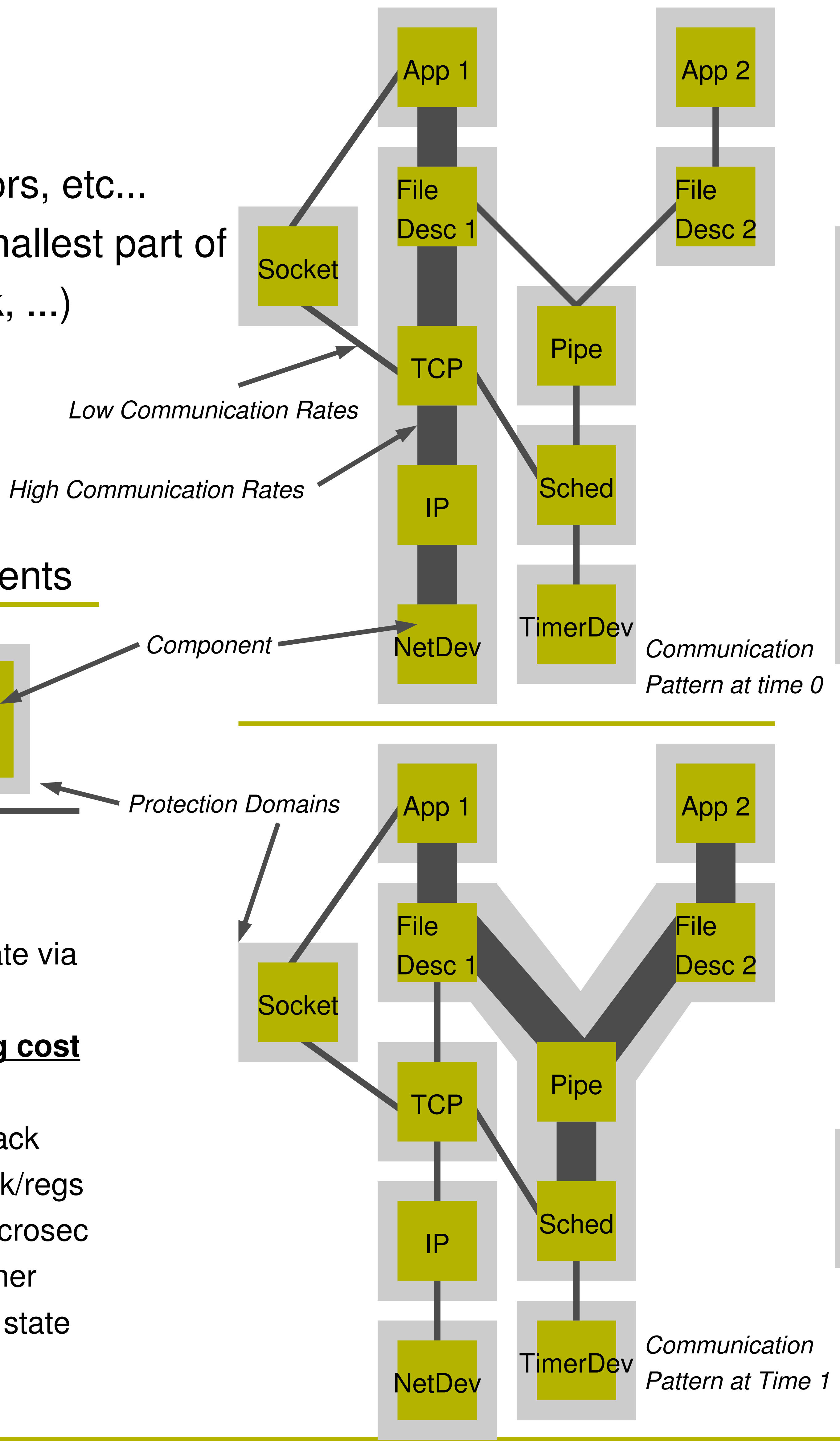


MPD in the Composite OS

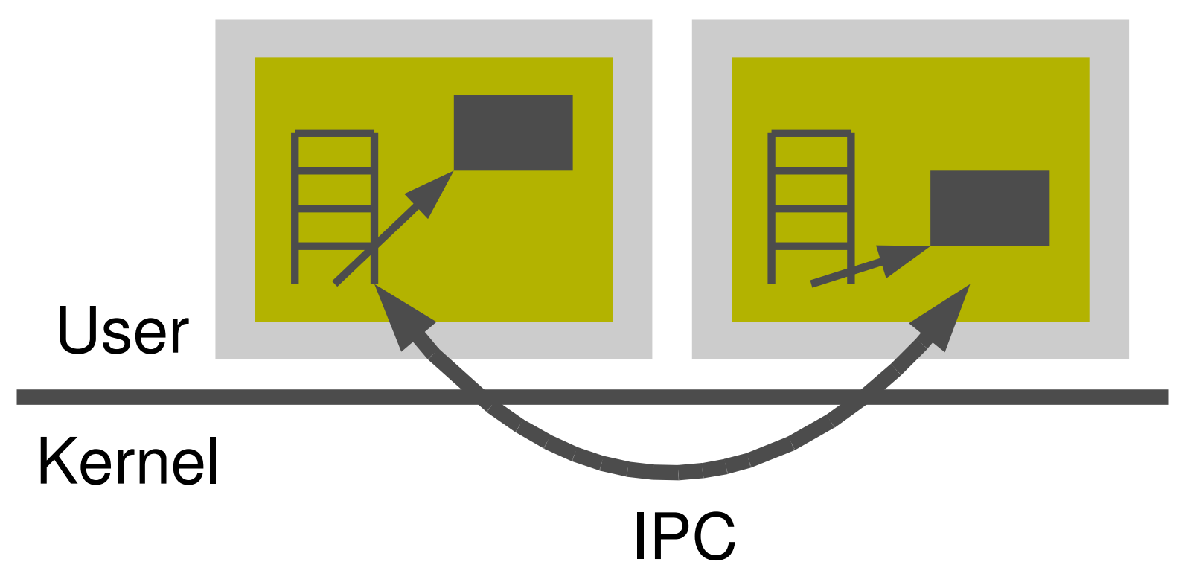
- Small region shared with kernel in each component points to code relevant for current protection type
  - Serialize function arguments and IPC
  - Direct invocation of destination function
- Kernel can dynamically change protection type by altering this structure and memory context (page-tables)

Protection Domains in Operating Systems

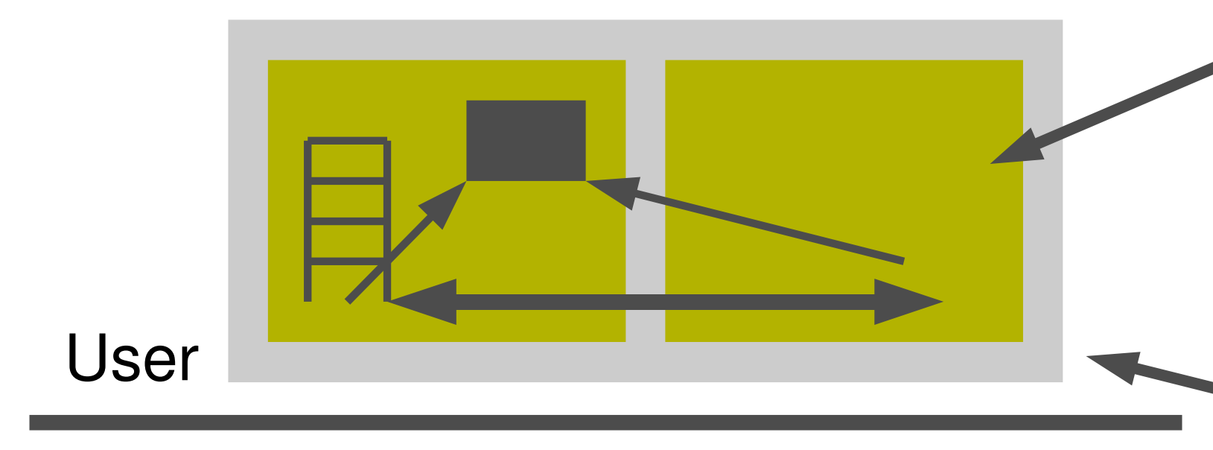
- Protection domains provide a basis for
  - Resource usage accountability – memory, file descriptors, etc...
  - Fault isolation – a manifested error should effect the smallest part of the system possible (deadlock, ptr corruption, mem leak, ...)
    - Essential for reliable systems – bugs are inevitable!
- Significant processing costs for inter-protection domain communication



Trade-off in Placing Protection Domains Around Components

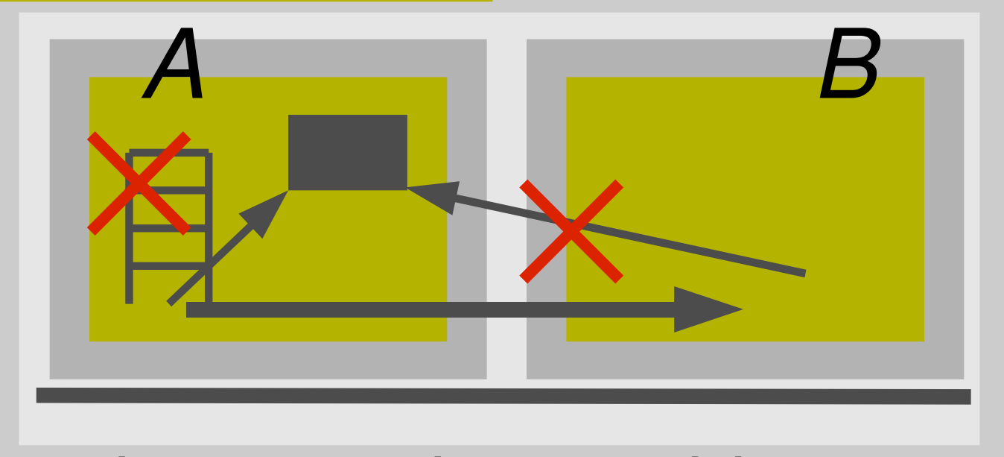


- Inter-process Communication (IPC) used to communication between protection domains
- **High invocation processing overhead**
  - 4 user <---> kernel-level switches
  - 2 hardware memory context switches
  - Switch execution stacks
  - Pass any function arguments (copy)
  - Invocation takes ~0.63 microsec
- **Increased reliablity** - fault in one component doesn't necessarily effect the other



- Components in the same protection domain communicate via direct function calls
- **Low invocation processing cost**
  - Function call overhead
  - Single shared execution stack
  - Arguments passed via stack/regs
  - Invocation takes ~0.022 microsec
- **Less reliability** – fault in either component can easily corrupt state in the other, no fault isolation

Reality Interjects: Implementation Complications

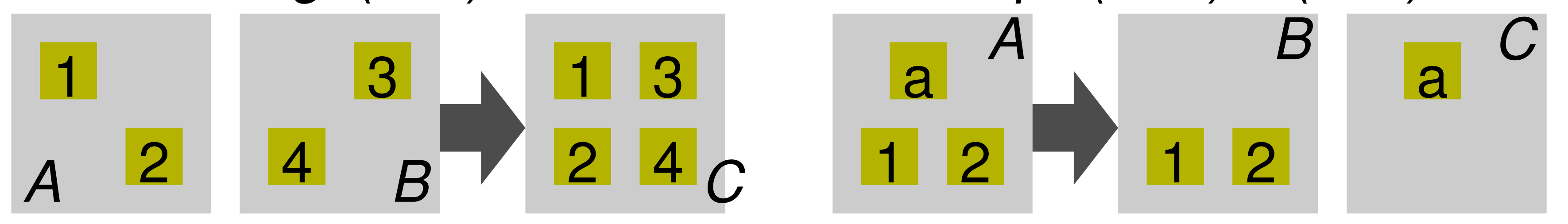


- Problem:
  1. A thread make invocation from component *A* to *B*
  2. *A* and *B* are split into separate protection domains
- The thread immediately faults: Can't access arguments or its execution stack!
- Solution:
  - Threads maintain view of protection domains in current components **until** they return
  - Garbage collect these stale protection domains (predictable via reference counting)

Mutable Protection Domains Canonical Operations

- Policies for protection domain placement defined in components: What should the kernel interface to manipulate them look like?
  - Removing protection boundaries is performance-sensitive
    - Requires predictability – remove overhead in a bounded time
  - Avoid creating many stale protection domains

merge(A,B) = C *split(A, a) = (B,C)*



move(A,B,a) = (C,D) is a composite of split and merge

- Moves *a* in *A* to *B*, output as *C* and *D*