

EEL-4736/5737
**Principles of Computer System
Design**

Lecture Slides 10
Textbook Chapter 5
Enforcing Modularity in Memory

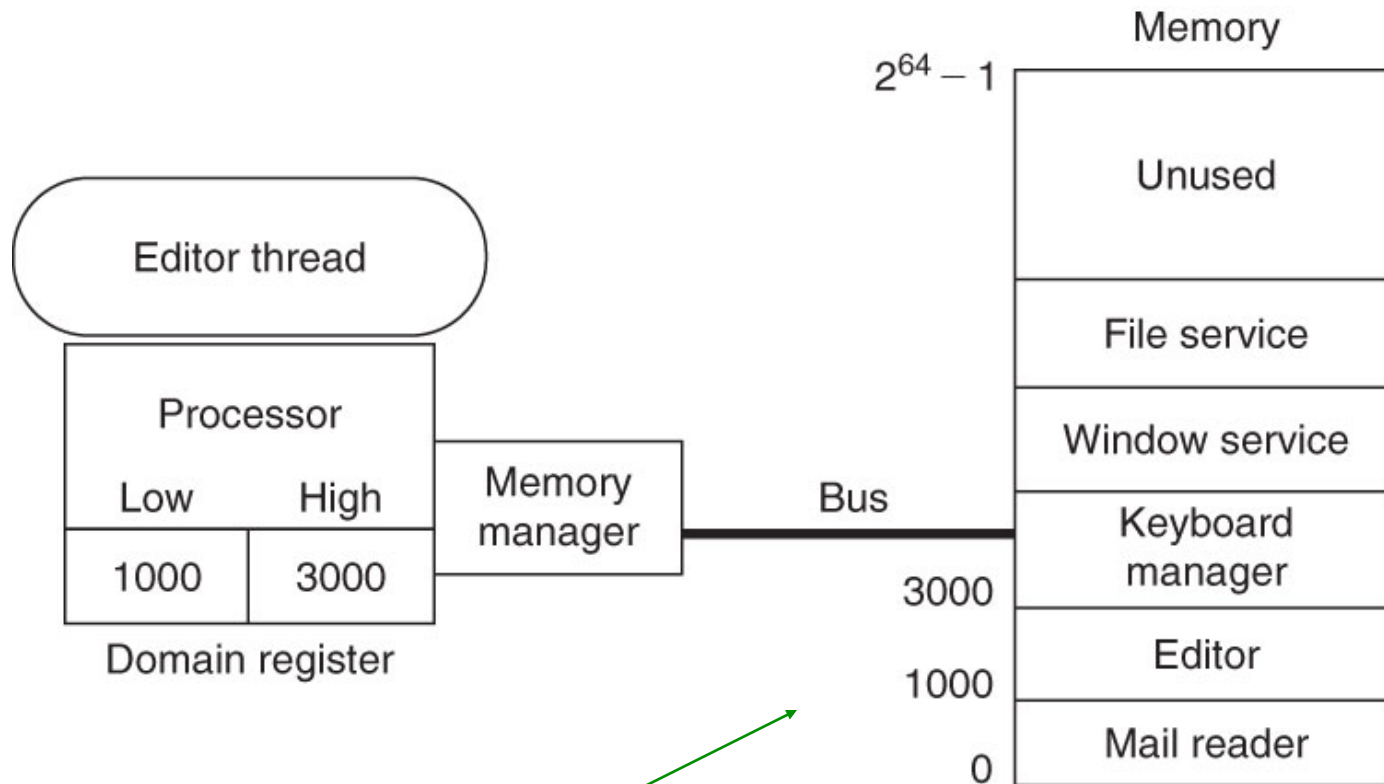
Introduction

- Shared memory does not enforce modularity
 - Recall procedure caller/callee: stack is shared memory, and stack discipline is a convention
 - RPC separates memory physically; shared memory without isolation would break the modularity enforcement of client/service
- For multiple threads to share a single computer, it is key to provide additional mechanisms to separate address spaces

Enforcing modularity - Domains

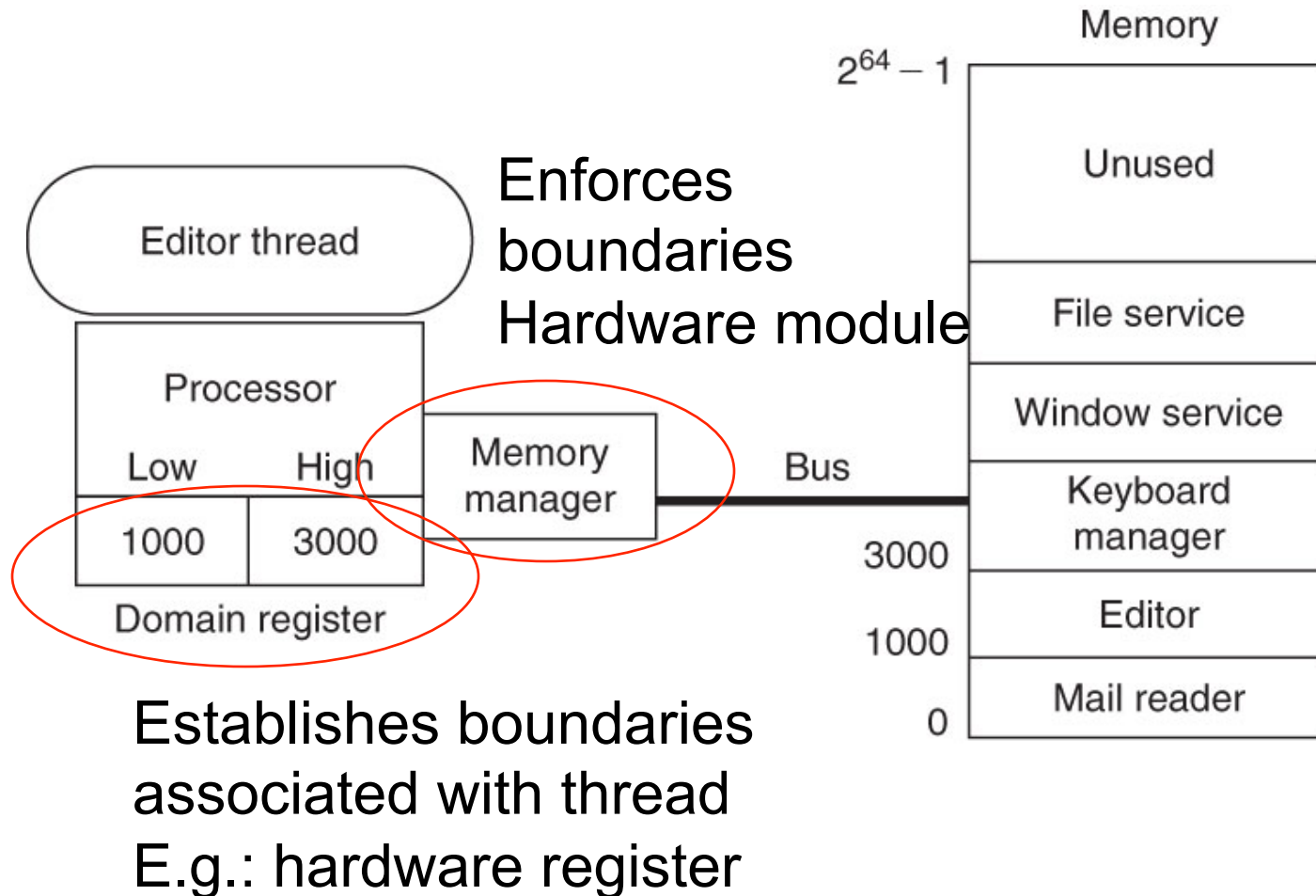
- Key idea: restrict the memory references of a thread
- Domains – contiguous range of memory addresses associated with threads
- Enforcing domain boundaries:
 - Need a “memory manager” module
 - Each thread has beginning/end boundary address
 - Memory references checked against boundaries

Domains

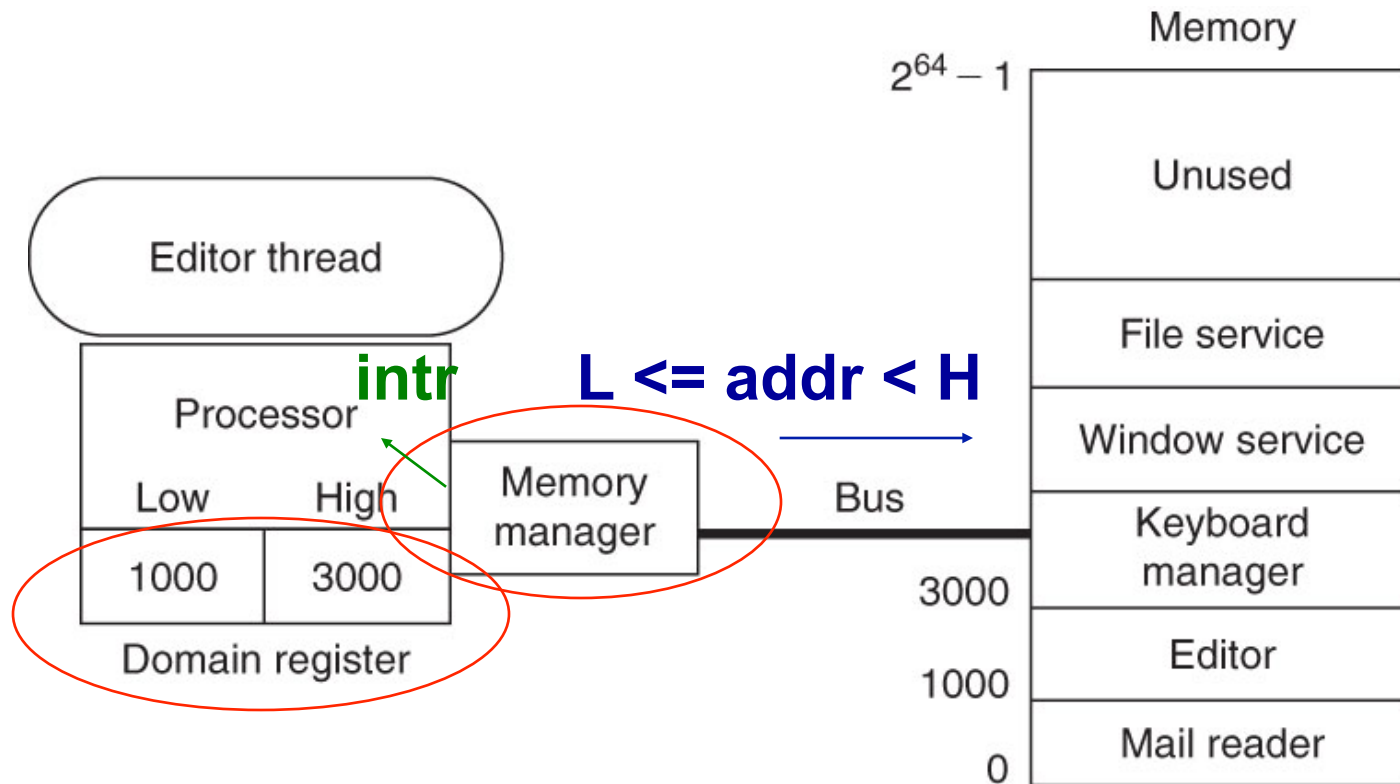


**Regions thread access memory separated
By hardware address boundaries**

Domains



Domains



If $\text{Low} \leq \text{addr} < \text{High}$
Proceed with access

Else

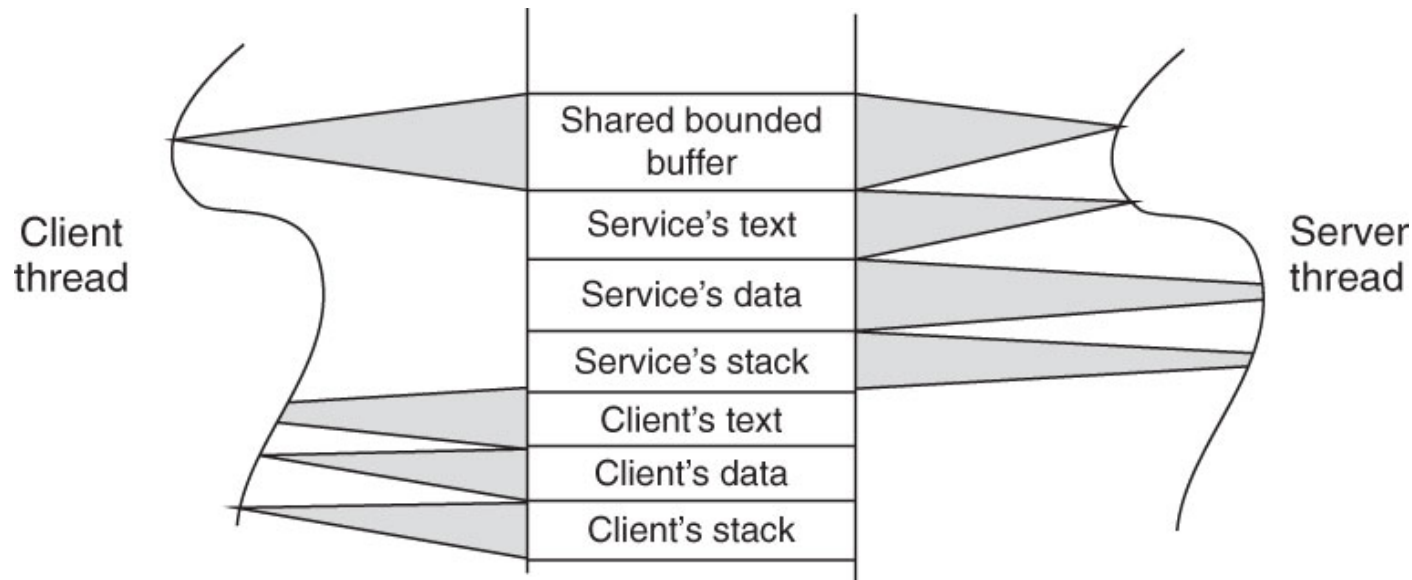
Interrupt processor (exception)

Issues

- One domain per thread is too restrictive
 - How to selectively share? Private and shared data for same thread
- Who controls the domain register?

Multiple Domains

- To support controlled sharing
 - Multiple domains per thread
 - Multiple domain registers



Basic interface

base_address <- ALLOCATE_DOMAIN (*size*)

Allocate a new domain of given size, return its base address

MAP_DOMAIN (*base_address*)

Add domain to calling thread's set of domains

Useful to classify permissions:

read, write, execute

Adding permissions

MAP_DOMAIN (*base_address, permission*)

Software/Hardware memory manager check

LOOKUP_AND_CHECK (addr, pneeded)

for each domain do

if CHECK_DOMAIN(addr, pneeded, domain) return domain
signal memory_exception

CHECK_DOMAIN (addr, pneeded, domain)

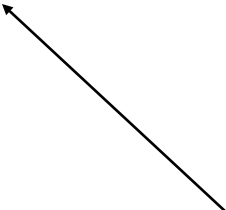
if domain.low <= addr < domain.high then

if PERMITTED (pneeded, domain.perm) then return TRUE

else signal permission_exception

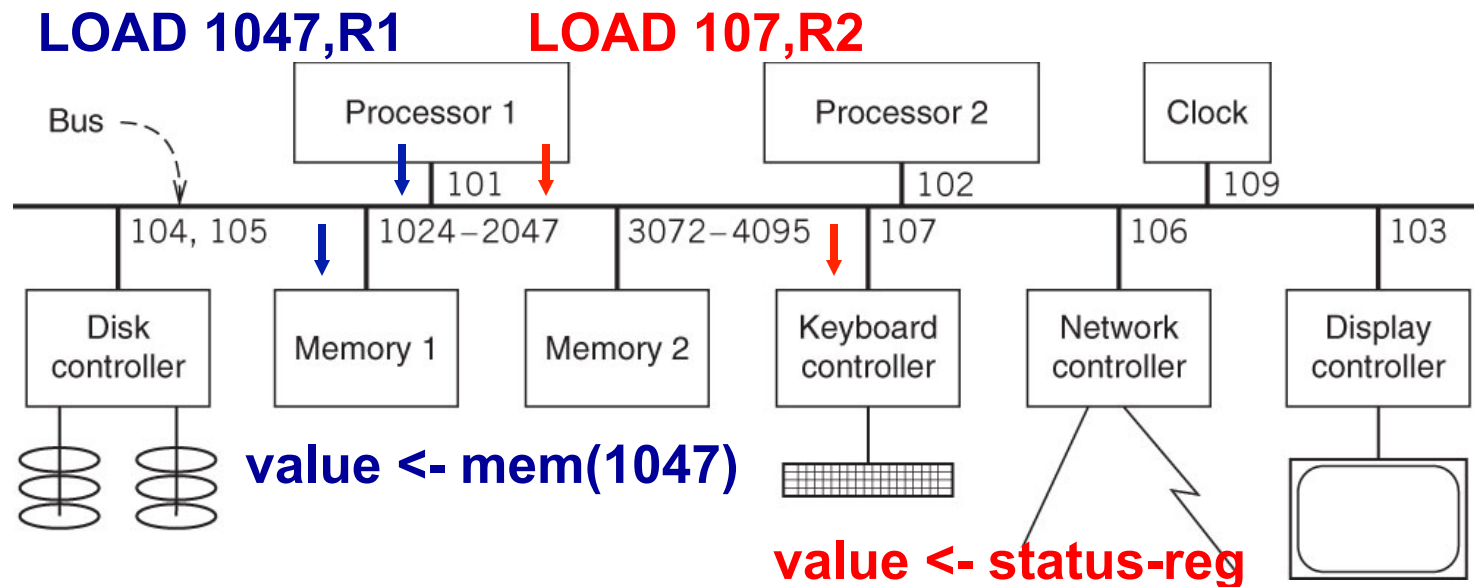
return FALSE

Load: read
Store: write
Fetch: exec



Memory-mapped I/O

- Domains with permissions can also be used to control access to I/O devices in memory-mapped systems



Issues

- One domain per thread is too restrictive
 - How to selectively share? Private and shared data for same thread
- Who controls the domain register?

Privilege separation

- Need mechanism to allow the manager to control domains, not threads themselves
 - Otherwise modularity is not fully enforced
 - A thread should not change its own domain boundaries or permissions, or that of others
- Kernel and user modes
 - Simplest case: a 1-bit register in processor
 - **Mode = kernel**: allowed to change domain reg
 - **Mode = user**: not allowed to change domain reg
 - Raise an exception

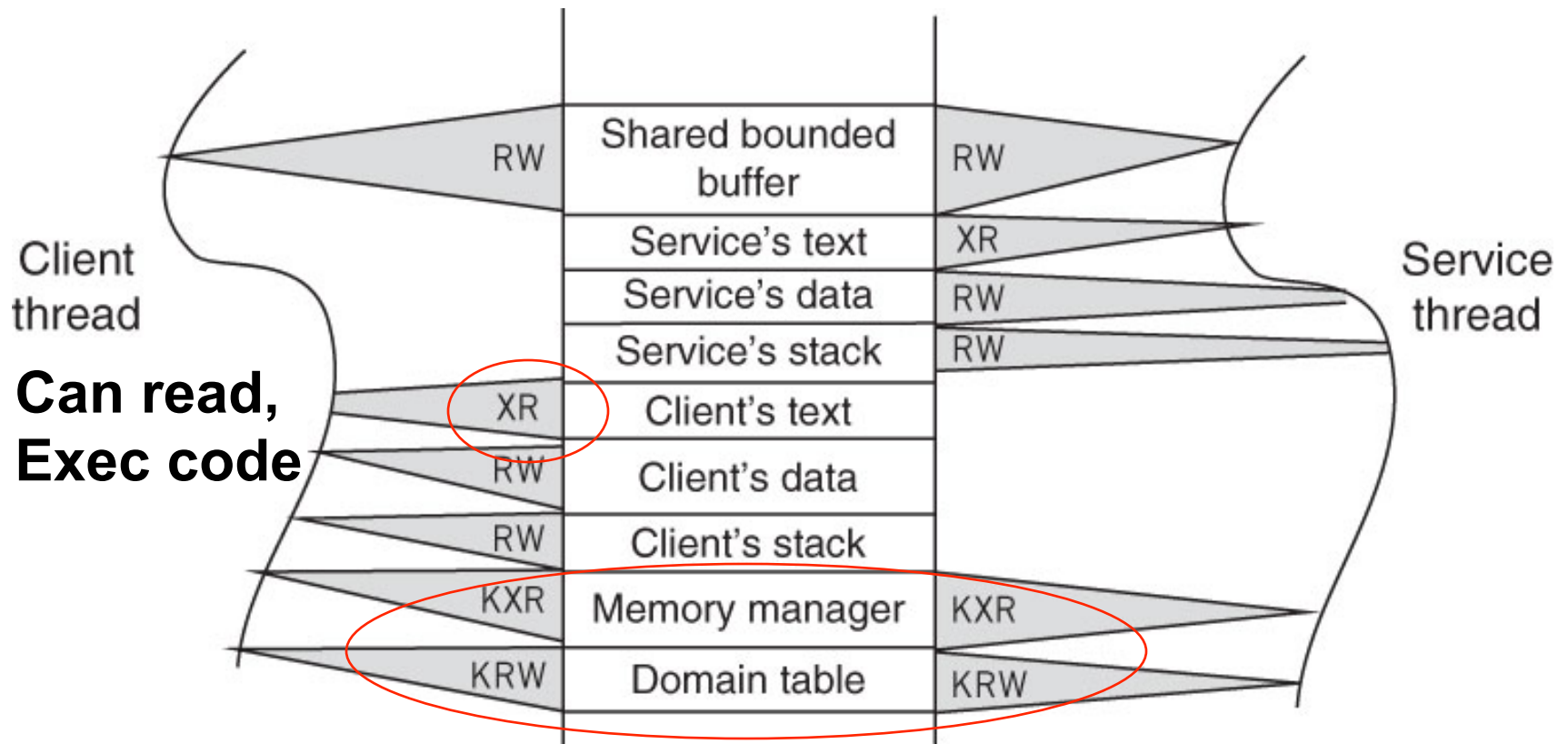
Kernel and user modes

- Add extra permission information about memory domain
 - `KERNEL_ONLY`: permission check includes a verification that `mode=kernel`
 - Prevent user-mode threads from accessing kernel memory
- Handling exceptions
 - Processor automatically switches to `mode=kernel` as a side-effect from an exception

Exceptions

- Illegal memory reference exception
 - Address doesn't fall within domain bounds
- Permission error exception
 - Address falls within bound, but thread has no sufficient permission
 - E.g. STORE to domain with RX permission; user-mode access to domain with K permission
- Illegal instruction exception
 - Attempt to change domain register in user mode
 - In general, any privileged operation in user mode

Domains revisited



Must be in kernel mode to execute memory manager and read/write domain table

Gates: Changing Modes

- To enforce modularity, threads in user mode must not directly invoke procedures of the memory manager
 - Calling arbitrary address - problematic
 - Must enter kernel mode in a controlled manner – e.g. system call
- Help from special processor instruction
 - *SVC – supervisor call*
 - Executes *atomically*:
 - Change mode from user to kernel
 - Set the PC to *predefined address* – *gate manager entry address*

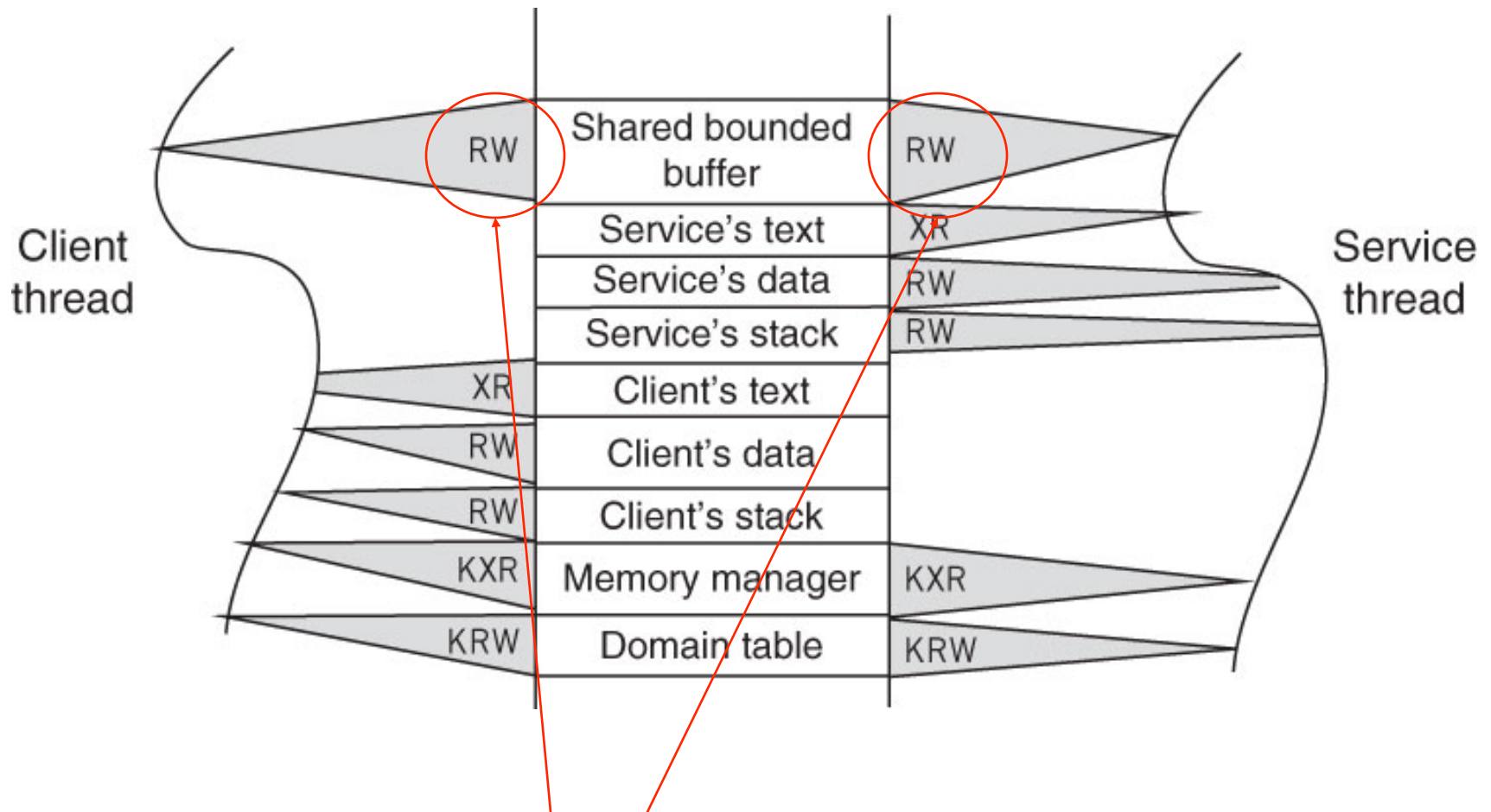
Gate manager

- Gates allow multiple kinds of entry
 - Multiple entry addresses (e.g. interrupt #)
 - Argument passed by instruction
 - Name is typically a number, and gate manager has a table mapping numbers to procedure entry points
- Gates can also be used to manage processor interrupts (e.g. I/O devices, timers) and exceptions (e.g. illegal memory reference)

Returning from kernel mode

- Two operations:
 - Change mode from kernel to user
 - Load PC value *from top of stack*
- Can be performed in two steps, but often single instruction (RETI)
- Going from user to kernel mode – predefined address; returning to user mode – stack. Why?

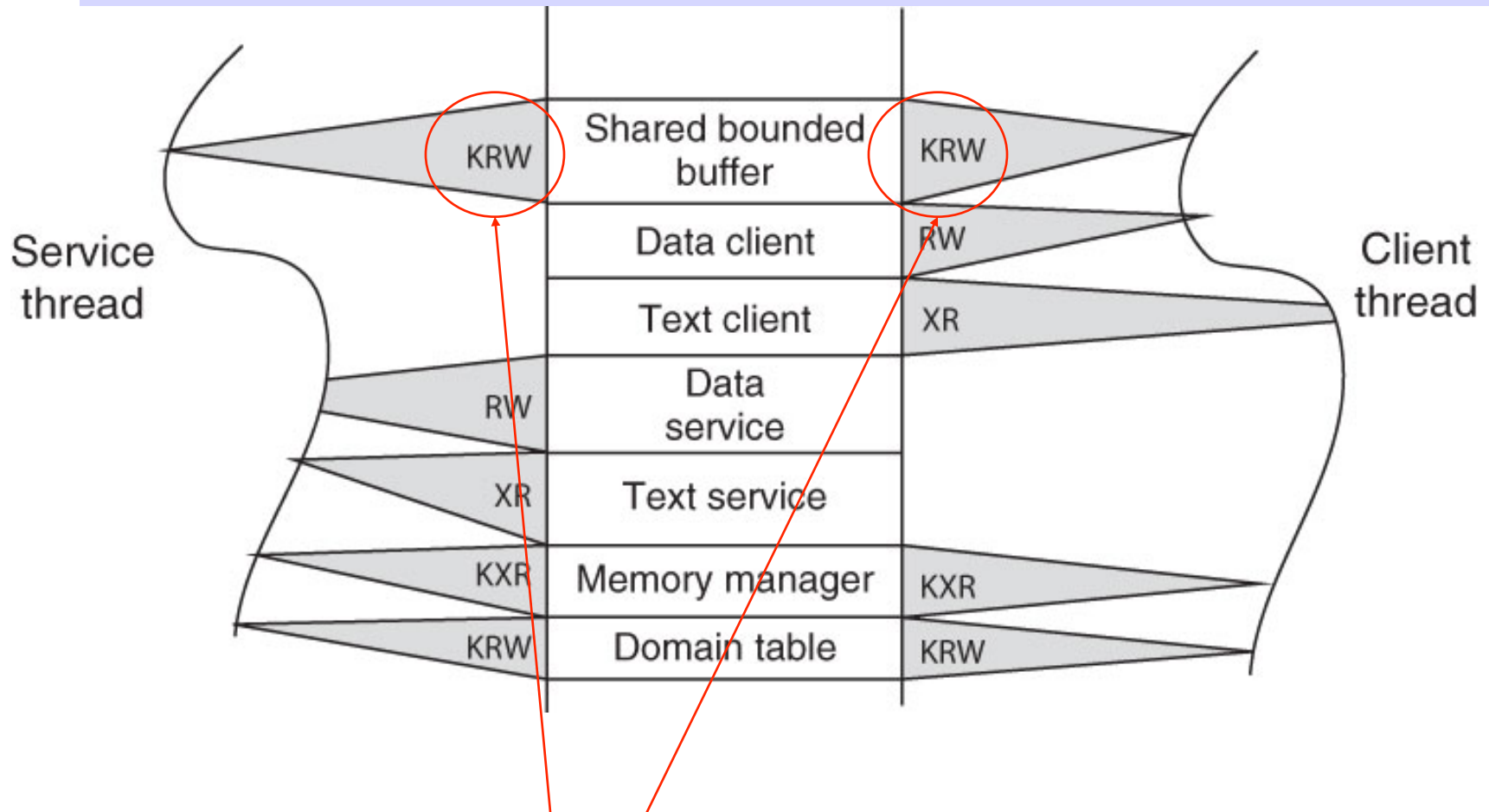
Bounded buffers



Good: other threads cannot read/write

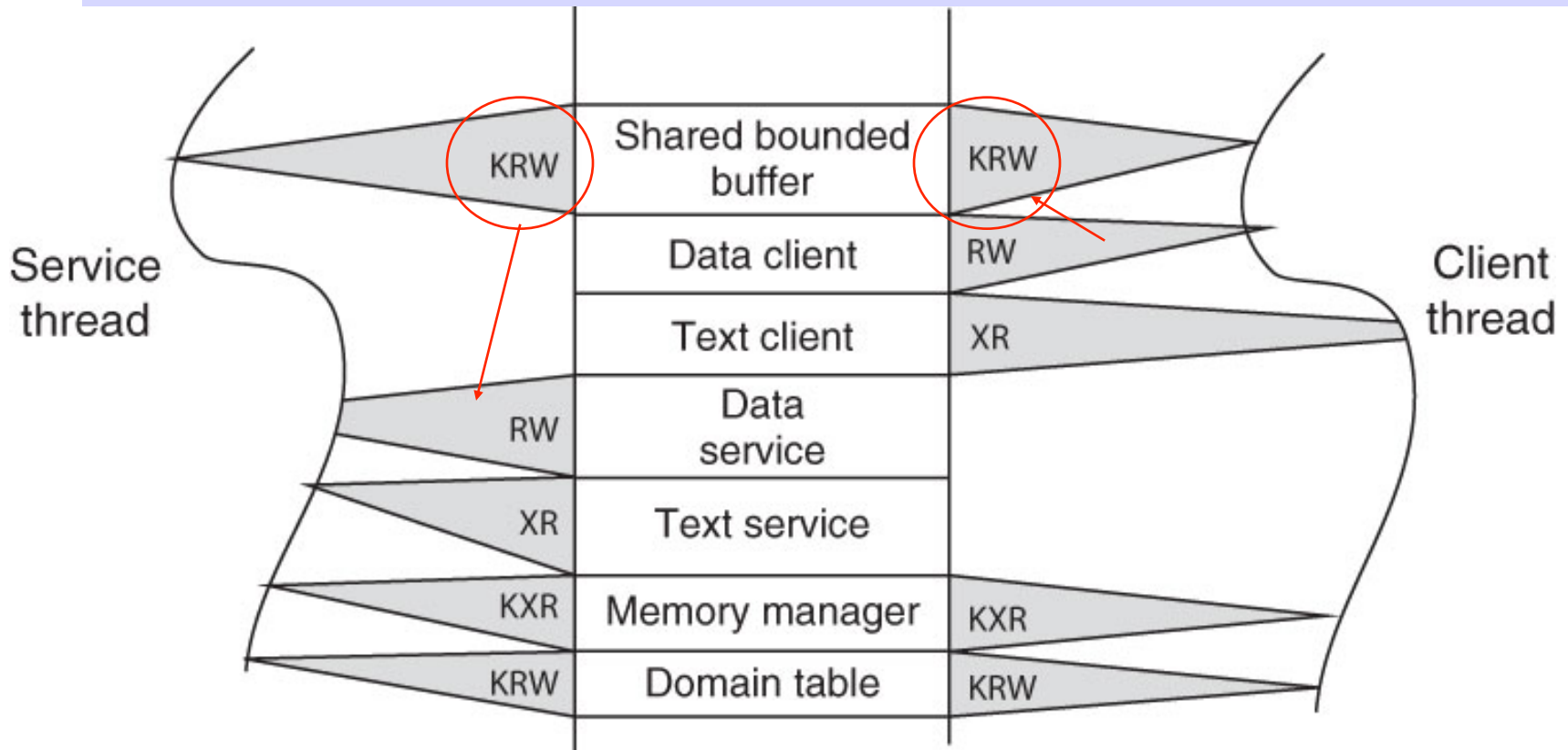
Bad: threads can bypass the send/receive API

Bounded buffers



Now threads need to go through a manager to read/write to shared bounded buffer

Bounded buffers



SEND/RECEIVE: supervisor calls (system calls)

There is a performance cost:

Copy from user domain to kernel on SEND

Copy from kernel to user domain on RECV

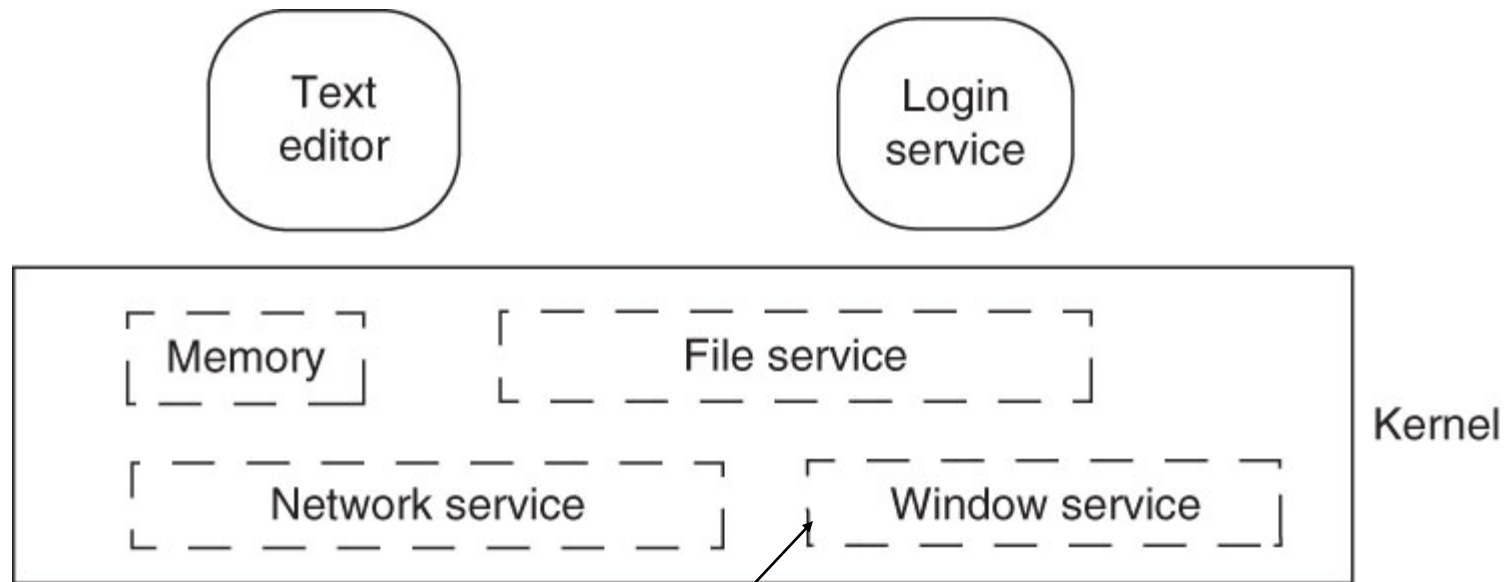
Overhead in entering/exiting gate

The kernel

- Collection of modules running in kernel mode – kernel program, or kernel
 - E.g. SEND/RECEIVE, domain memory managers – O/S kernel programs accessible through system call interface
- Kernel is a *trusted intermediary*
 - Only program that can execute “privileged” instructions

Monolithic kernel

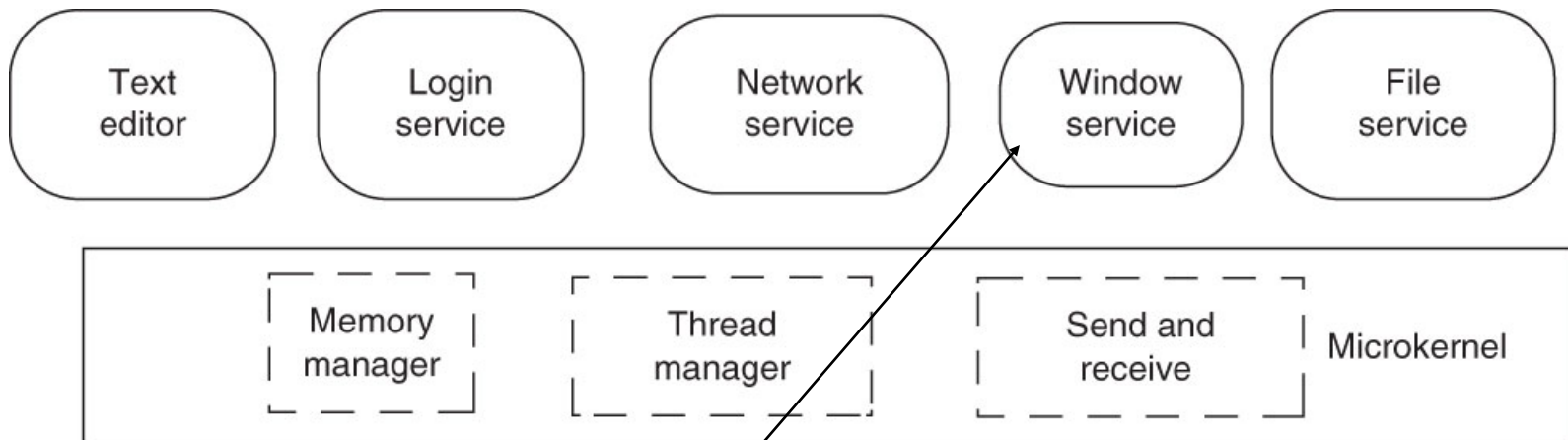
- Most of the operating system services run in kernel mode



An error here is fatal!
Blue screen of death

Microkernel

- Structures the operating system itself in a client/service architecture



An error here is not fatal
Error contained within module

Monolithic vs. client/service

- Most of today's O/S kernels are monolithic
 - Critical services that fail in either mode could render a system effectively useless
 - E.g. file service
 - It is simpler to program with shared data structures
 - HW#2
 - Performance overhead of inter-process communication can be significant
 - Legacy kernel code – porting to microkernel-style not trivial

Bootstrapping

- From zeroes to a system manager program (o/s kernel) in control of processor kernel mode and applications confined to user mode
- How does this bootstrapping process work?

Bootstrapping

- Start in most privileged mode
- Start with well-known location in memory – address 0
 - In ROM
 - Rudimentary file system to support loading kernel image from boot device (e.g. hard disk) into memory
 - Jump to beginning of kernel

Bootstrapping

- Kernel allocates a thread and memory domains for itself
 - Stack, domain tables
- Program addresses that are used for interrupt/exception gates
- Allocates threads for non-kernel services
 - Here the processor goes to user mode
 - Returns to kernel mode only through a gate

Reading

- Sections 5.5-5.6