Operating Systems

Dr. Shu Yin

Part II: Process Management

- Processes
- Threads
- Process Synchronization
- CPU Scheduling
- Deadlocks

Goals

- Intros to processes

Recall: What's an OS?

- Special layer of SW that provides app SW access to HW resources
  - Convenient abstraction of complex HW devices
  - Protected access to shared resources
  - Security and authentication
  - Communication amongst logical entities

What's an OS

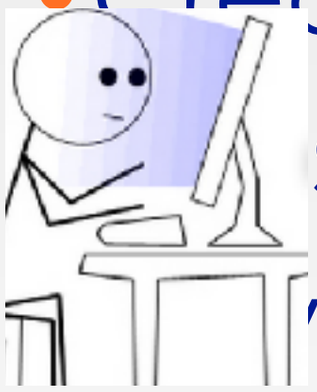
- Referee
  - Resource allocation, isolation, communication
- Illusionist
  - Infinite memory, dedicated machine
  - Higher level objects: files, users, messages
  - Masking limitations, virtualization
- Glue
  - Storage, Networking
  - Sharing, authorization
  - UI

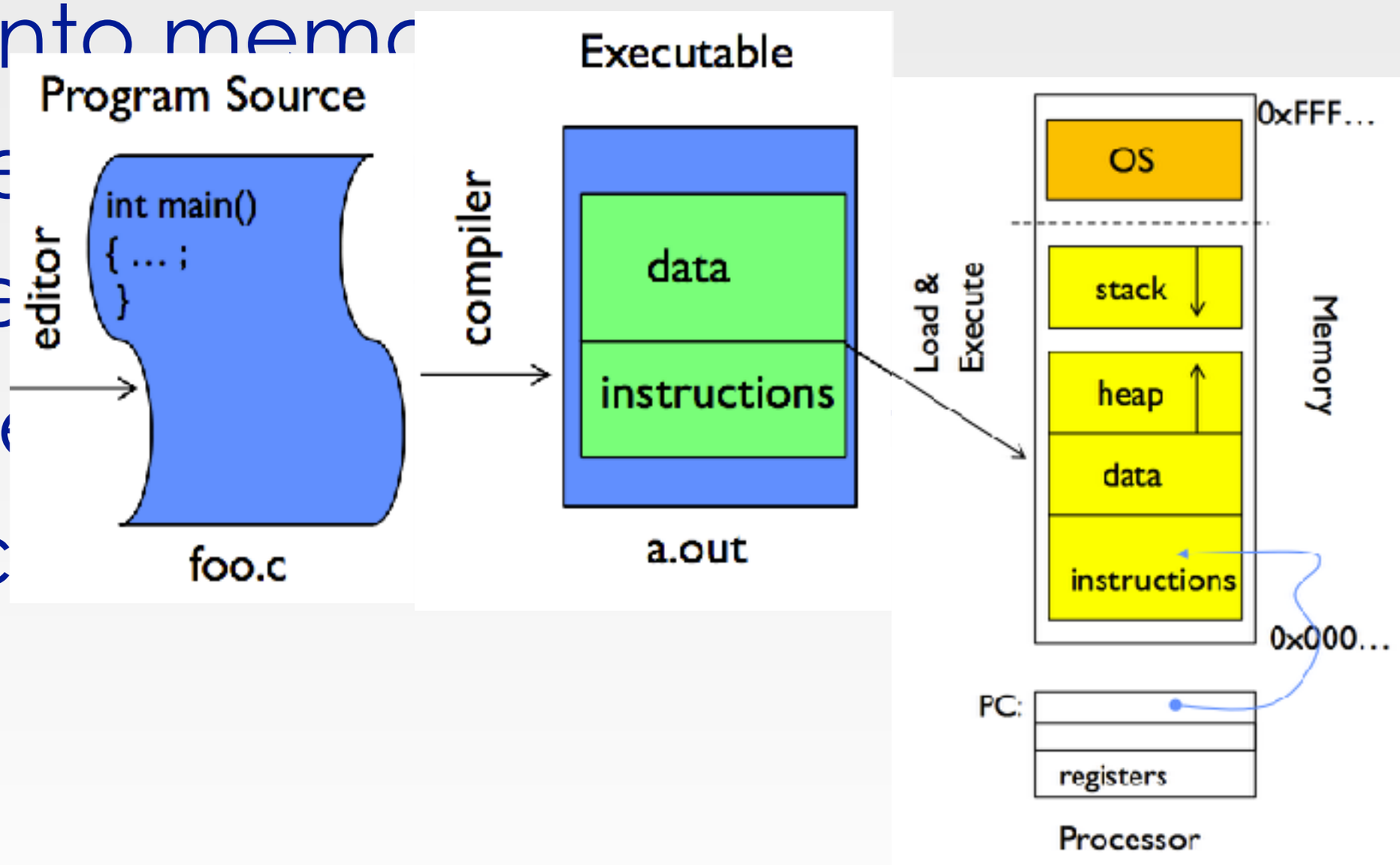
OS Bottom Line: Run Programs

- Load into memory
- Create stack and heap
- Transfer control to program
- Provide services to program
- Protect OS and program

OS Bottom Line: Run Programs

- Load into memo
- Create
- sfe
- ride
- Protec





4 Fundamental OS Concepts

- Thread
- Address Space with Translation
- Process
- Dual Mode Operation/Protection

Recall: Instruction Cycles

Recall: Instruction Cycles

- Fetch

Recall: Instruction Cycles

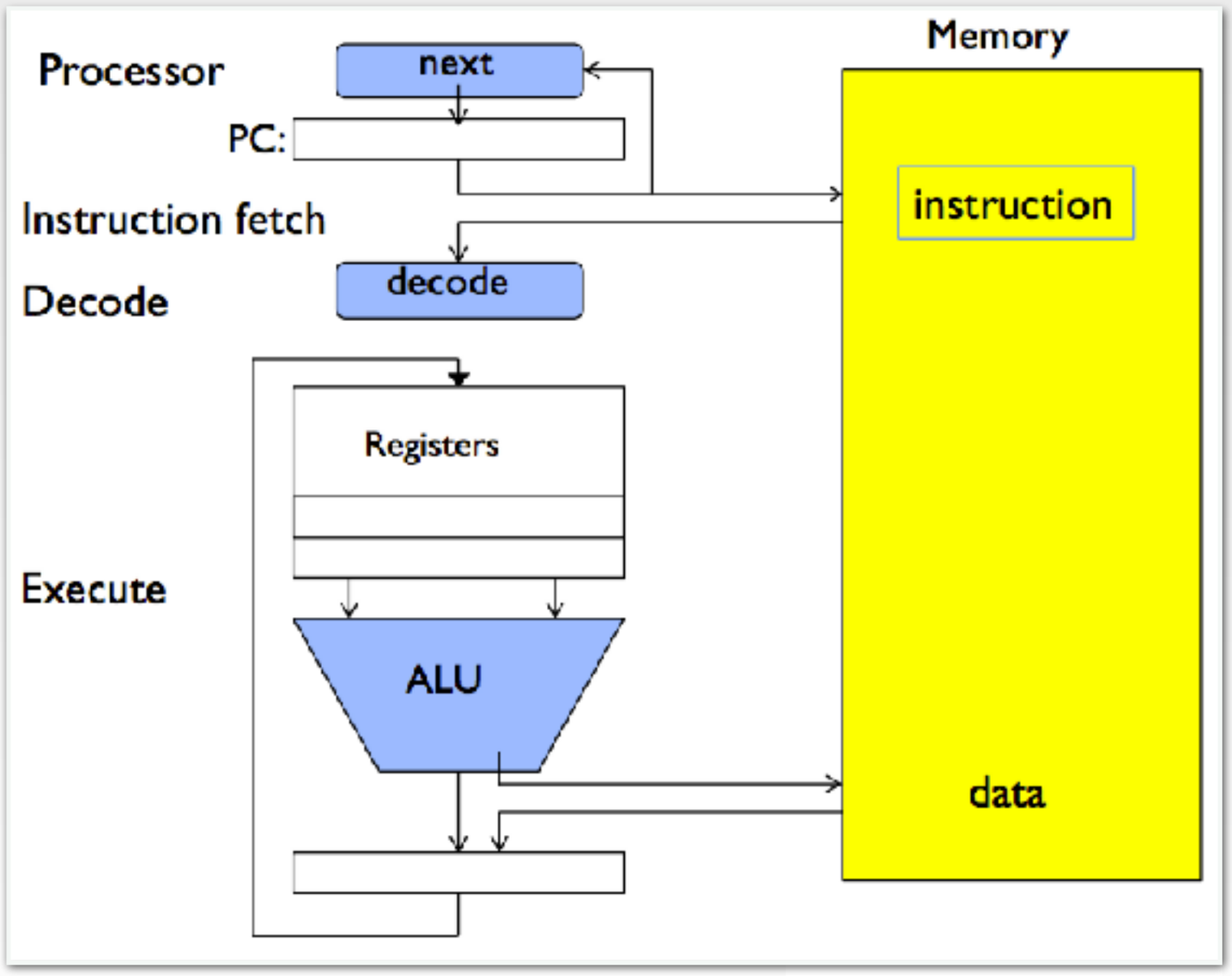
- Fetch
- Decode

Recall: Instruction Cycles

- Fetch
- Decode
- Execute

Recall: Instruction Cycles

- Fetch
- Decode
- Execute



Recall: Execution Sequence

- What happens during program execution?

Recall: Execution Sequence

- What happens during program execution?
  - Fetch instruction at PC

Recall: Execution Sequence

- What happens during program execution?
  - Fetch instruction at PC
  - Decode

Recall: Execution Sequence

- What happens during program execution?
  - Fetch instruction at PC
  - Decode
  - Execute (possibly involves registers)

Recall: Execution Sequence

- What happens during program execution?
  - Fetch instruction at PC
  - Decode
  - Execute (possibly involves registers)
  - Write results to registers/memory

Recall: Execution Sequence

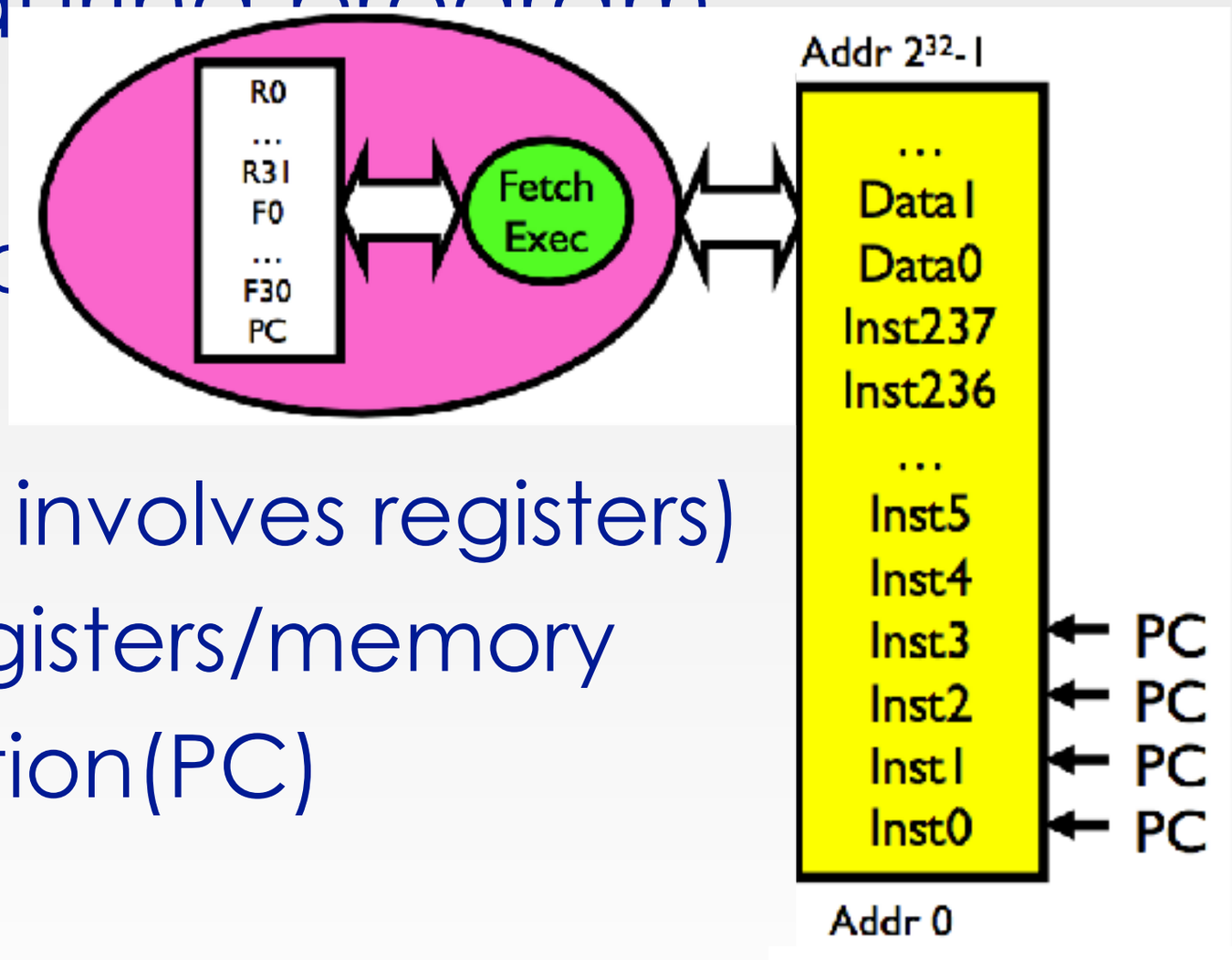
- What happens during program execution?
  - Fetch instruction at PC
  - Decode
  - Execute (possibly involves registers)
  - Write results to registers/memory
  - PC = Next instruction(PC)

Recall: Execution Sequence

- What happens during program execution?
  - Fetch instruction at PC
  - Decode
  - Execute (possibly involves registers)
  - Write results to registers/memory
  - PC = Next instruction(PC)
  - Repeat

Recall: Execution Sequence

- What happens during program execution?
  - Fetch instruction
  - Decode
  - Execute (possibly involves registers)
  - Write results to registers/memory
  - PC = Next instruction(PC)
  - Repeat



Four Fundamental OS Concepts(1)

- Thread
  - Single unique execution context
    - PC, Registers, Execution Flags, Stack

Thread of Control

- Certain registers hold the context
  - Stack pointer (address of the top of stack)
  - Defined by the ISA or by compiler
- Single unique execution context
- Executing when resides in the registers
- PC holds the address of instruction in the thread
- Registers hold the root state

Four Fundamental OS Concepts(2)

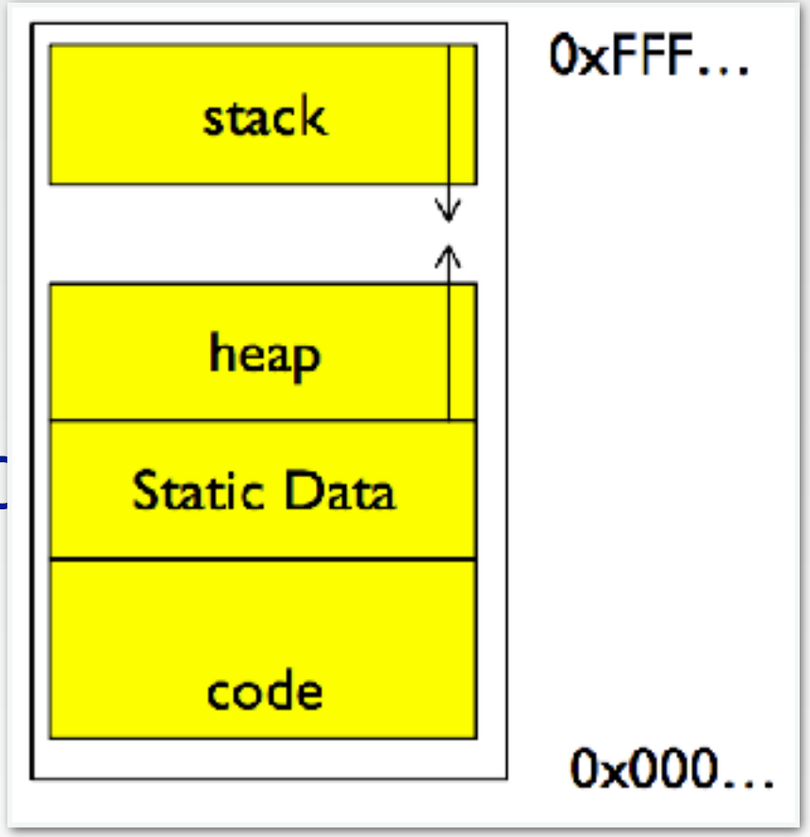
- Address Space with Translation
  - Programs execute in an address space
  - Distinct from the memory space of the physical machine

Program's Address Space

- Address space:
  - the set of accessible address +
  - state associate with them
  - For a 32-bit processes: 2^{32} addresses

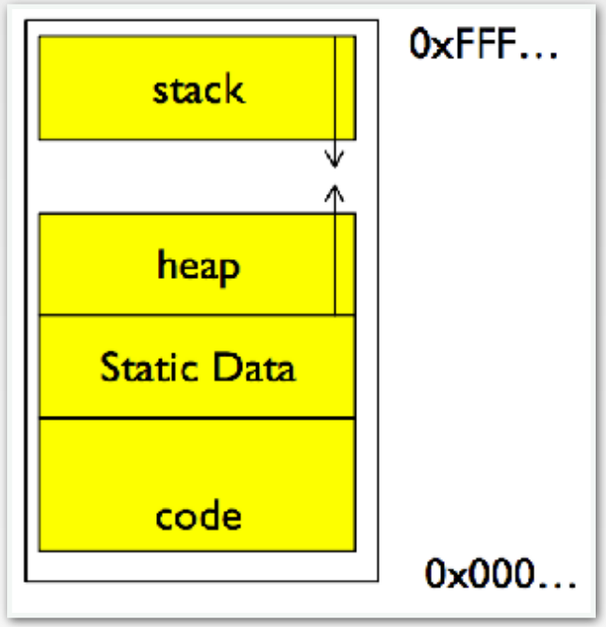
Program's Address Space

- Address space:
  - the set of accessible address
  - state associate with them
  - For a 32-bit processes: 2^{32} ad

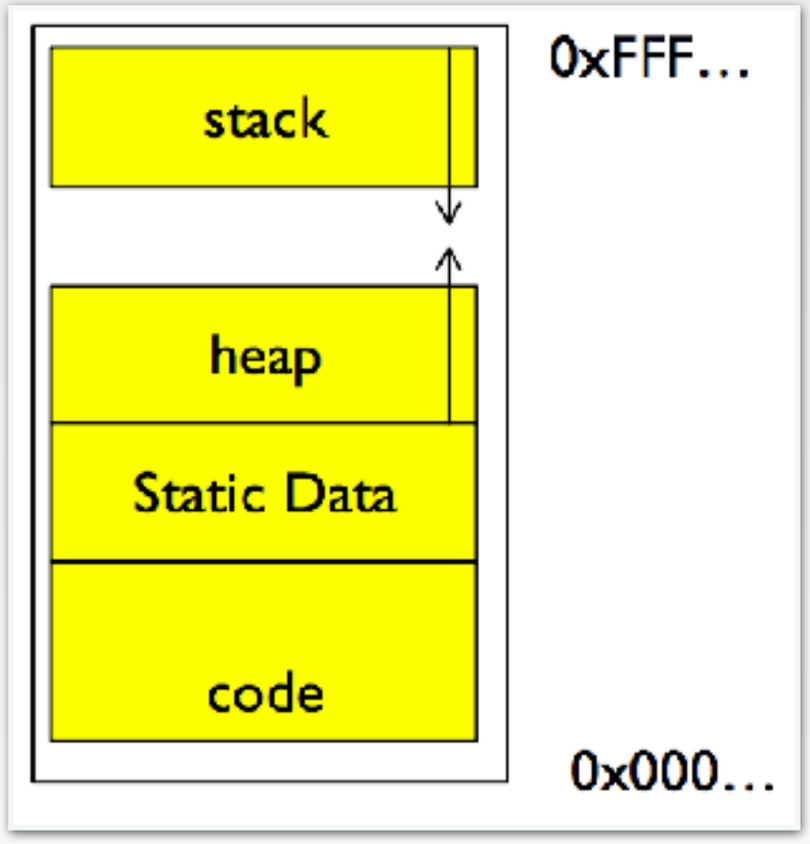


Program's Address Space(cont.)

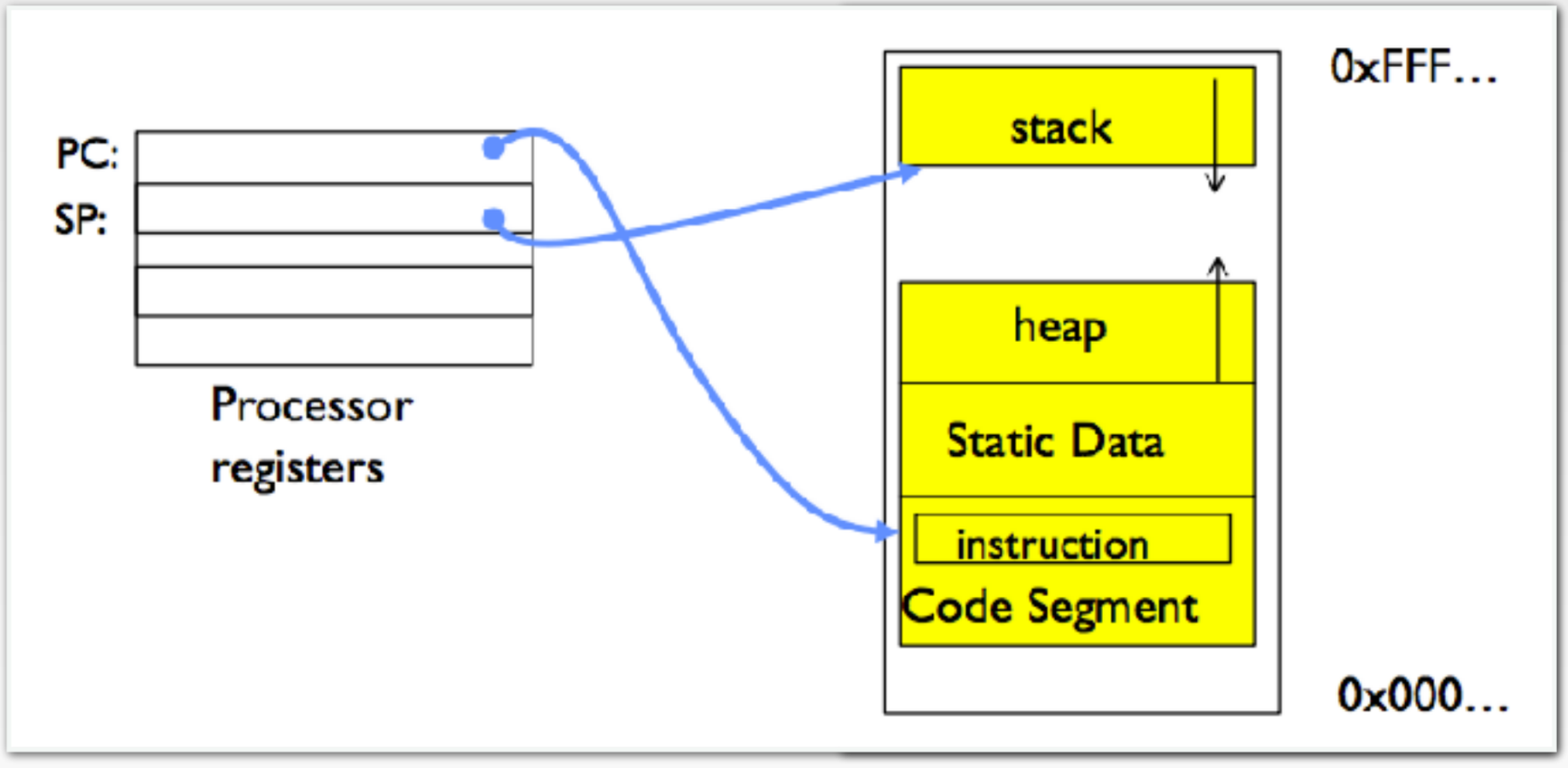
- What happens when you R/W to an address?
  - Perhaps nothing
  - Perhaps acts like regular memory
  - Perhaps ignores writes
  - Perhaps causes I/O operation
  - Perhaps causes exception



Address Space: In a Picture

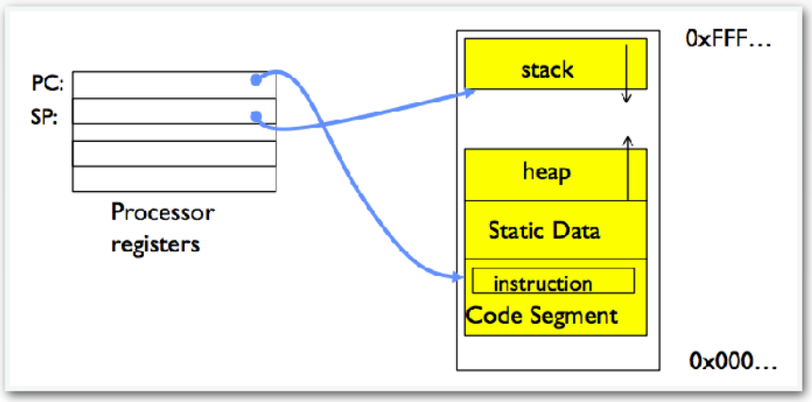


Address Space: In a Picture



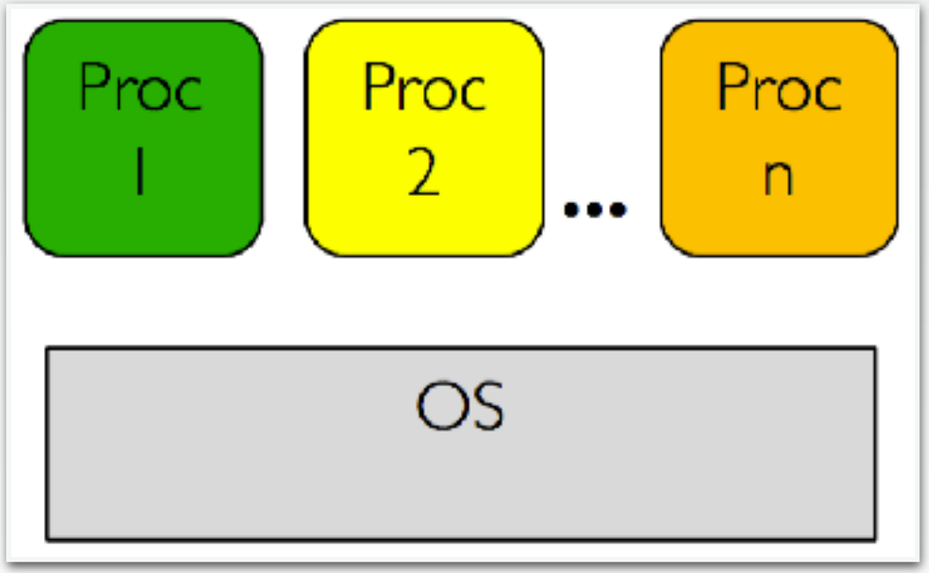
Address Space: In a Picture

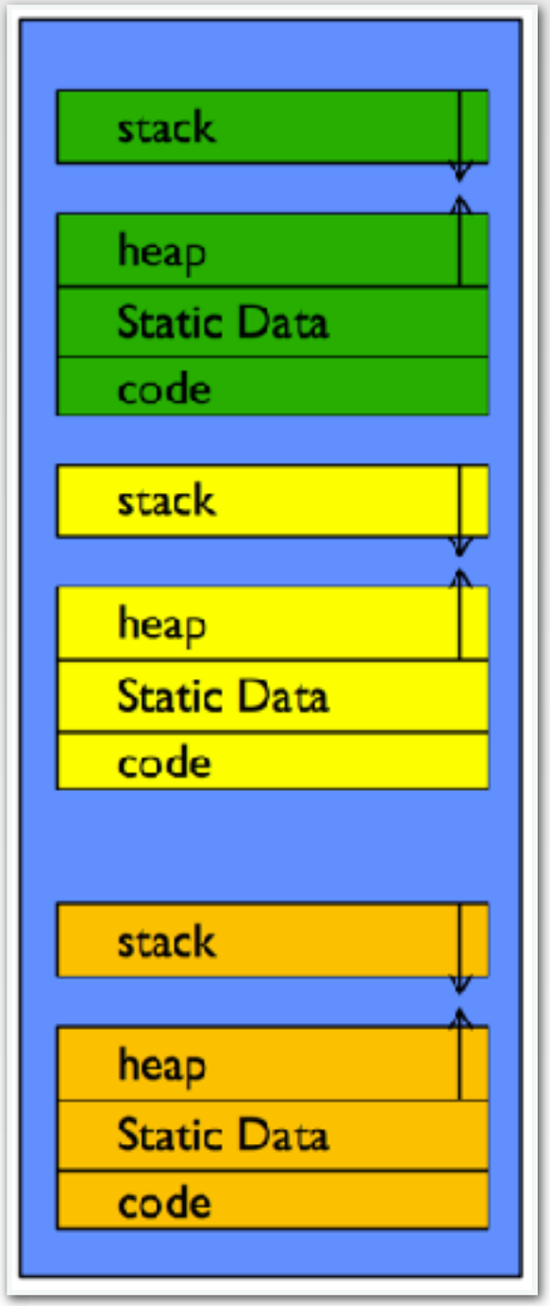
- What's in the Code Segment?
  - What's Static Data Segment?
- What's in the Stack Segment?
  - How is it allocated? How big is it?
- What's in the Heap Segment?
  - How is is allocated? How big is it?



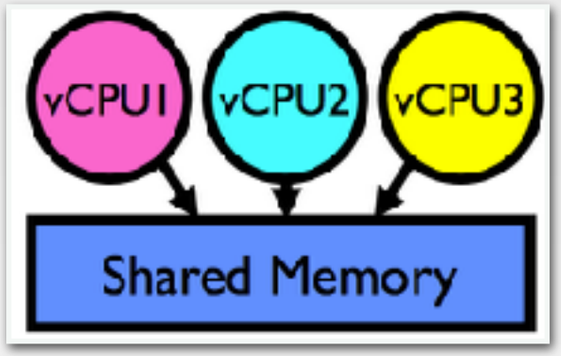
Multiprogramming

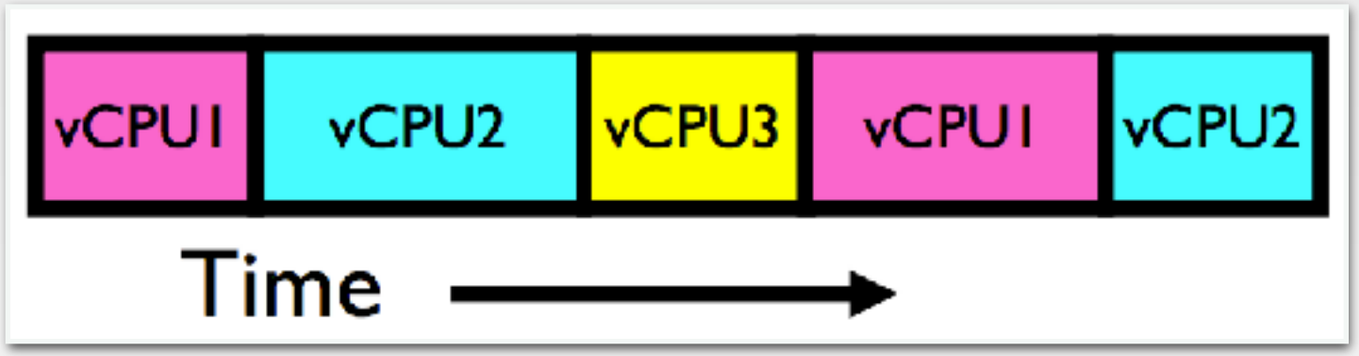
- Multiple threads of control





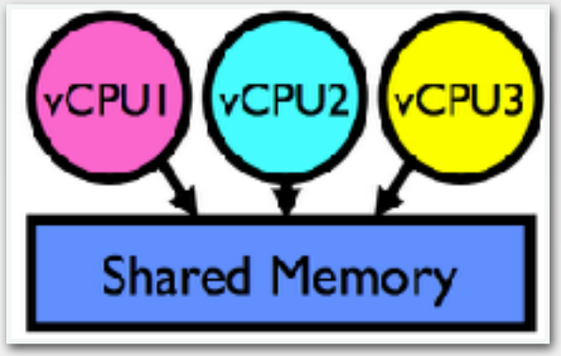
Illusion of Multiple Processors

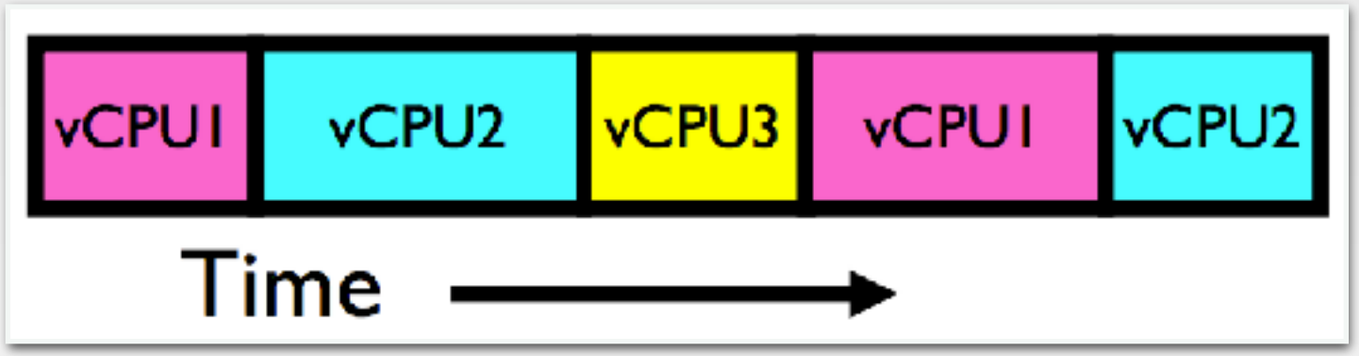




- Assume a single processor:
  - How do we provide the illusion of multiple processors? (Pentium 4)

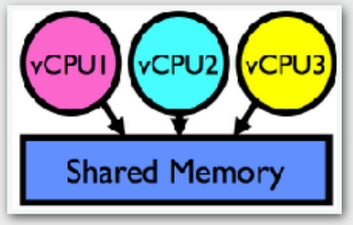
Illusion of Multiple Processors

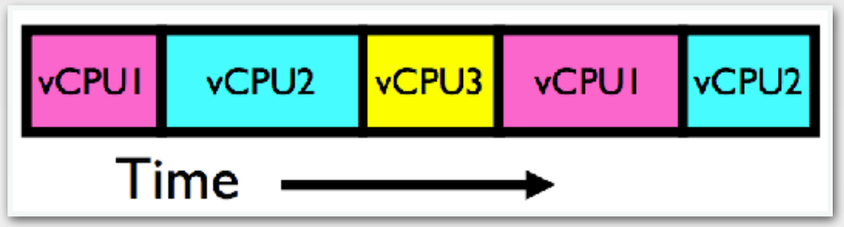




- Assume a single processor:
  - How do we provide the illusion of multiple processors? (Pentium 4)
  - Multiplex in time

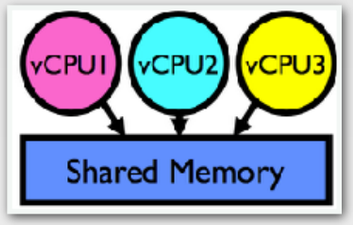
Illusion of Multiple Processors

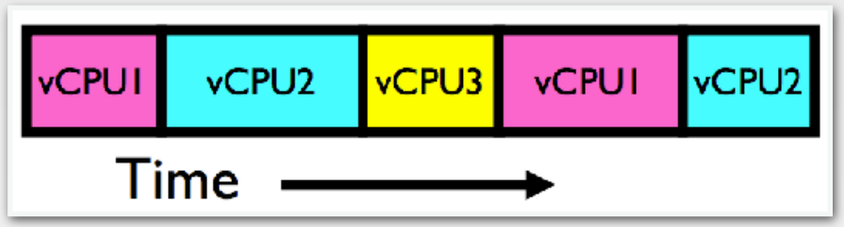




- Each virtual CPU
  - Needs a structure to hold
    - PC, SP
    - Registers

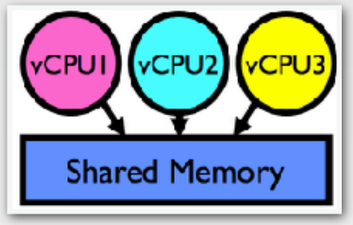
Illusion of Multiple Processors

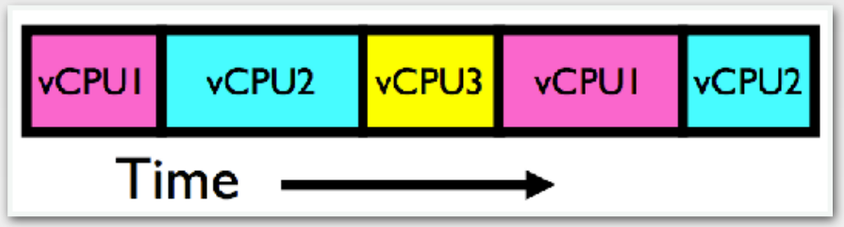




- Each virtual CPU
  - Needs a structure to hold
    - PC, SP
    - Registers
  - How to switch from one virtual CPU to the next?

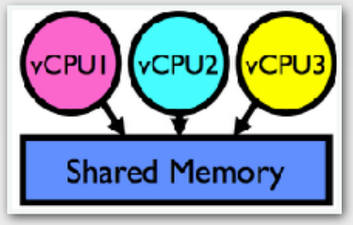
Illusion of Multiple Processors

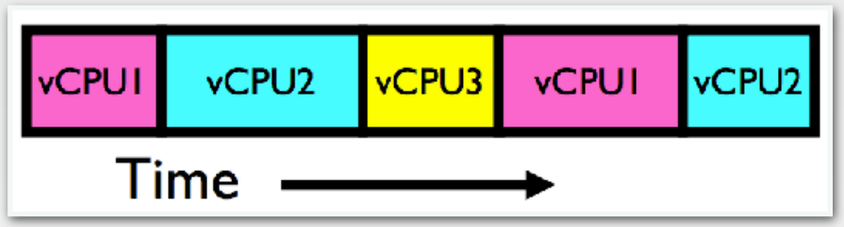




- Each virtual CPU
  - Needs a structure to hold
    - PC, SP
    - Registers
  - How to switch from one virtual CPU to the next?
    - Save PC, SP, Registers in the current state block

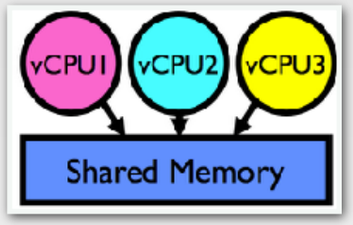
Illusion of Multiple Processors

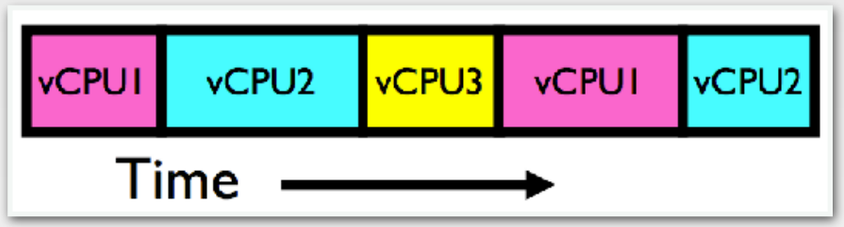




- Each virtual CPU
  - Needs a structure to hold
    - PC, SP
    - Registers
  - How to switch from one virtual CPU to the next?
    - Save PC, SP, Registers in the current state block
    - Load PC, SP, Registers from the new state block

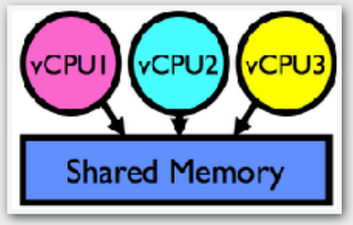
Illusion of Multiple Processors

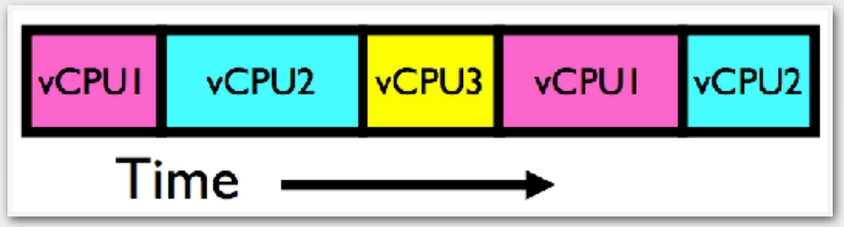




- Each virtual CPU
  - Needs a structure to hold
    - PC, SP
    - Registers
  - How to switch from one virtual CPU to the next?
    - Save PC, SP, Registers in the current state block
    - Load PC, SP, Registers from the new state block
  - What triggers switch?

Illusion of Multiple Processors





- Each virtual CPU
  - Needs a structure to hold
    - PC, SP
    - Registers
  - How to switch from one virtual CPU to the next?
    - Save PC, SP, Registers in the current state block
    - Load PC, SP, Registers from the new state block
  - What triggers switch?
    - Timer, Voluntary yield, I/O, others

Basic Problem of Concurrency

- Resources
  - HW: single processor, DRAM, I/O devices
  - Multiprogramming API
- OS has to coordinate all activity
  - Multiple processes, I/O interrupts, …
  - How can it do?
- Basic Idea: VM abstraction
- Dijkstra THE system

Properties of this Multiprogramming Technique

- All virtual CPUs share same non-CPU resources
  - I/O devices the same
  - Memory the same

Properties of this Multiprogramming Technique

- Consequence of sharing
  - Each thread can access the data of every other thread (good for sharing, bad for protection)
  - Threads can share instructions (good for sharing, bad for protection)
  - Can threads overwrite OS functions?

Properties of this Multiprogramming Technique

- This (unprotected) model is common in:
  - Embedded applications
  - Windows 3.1/Early Macintosh (switch only with yield)
  - Windows 95 - ME (switch with both yield and timer)

Protection

- Operating System must protect itself from user programs
  - Reliability
  - Security
  - Privacy
  - Fairness
- It must protect User programs from one another

Protection (cont.)

- Primary Mechanism
- Additional Mechanisms

Four Fundamental OS Concepts(3)

- Process
  - An instance of an executing program is a process consisting of an address space and one or more threads of control

Process

- Execution environment with Restricted Rights
  - Address Space with One or More Threads
  - Owns memory (address space)
  - Owns file descriptors, file system context, …
  - Encapsulate one or more threads sharing processes resources

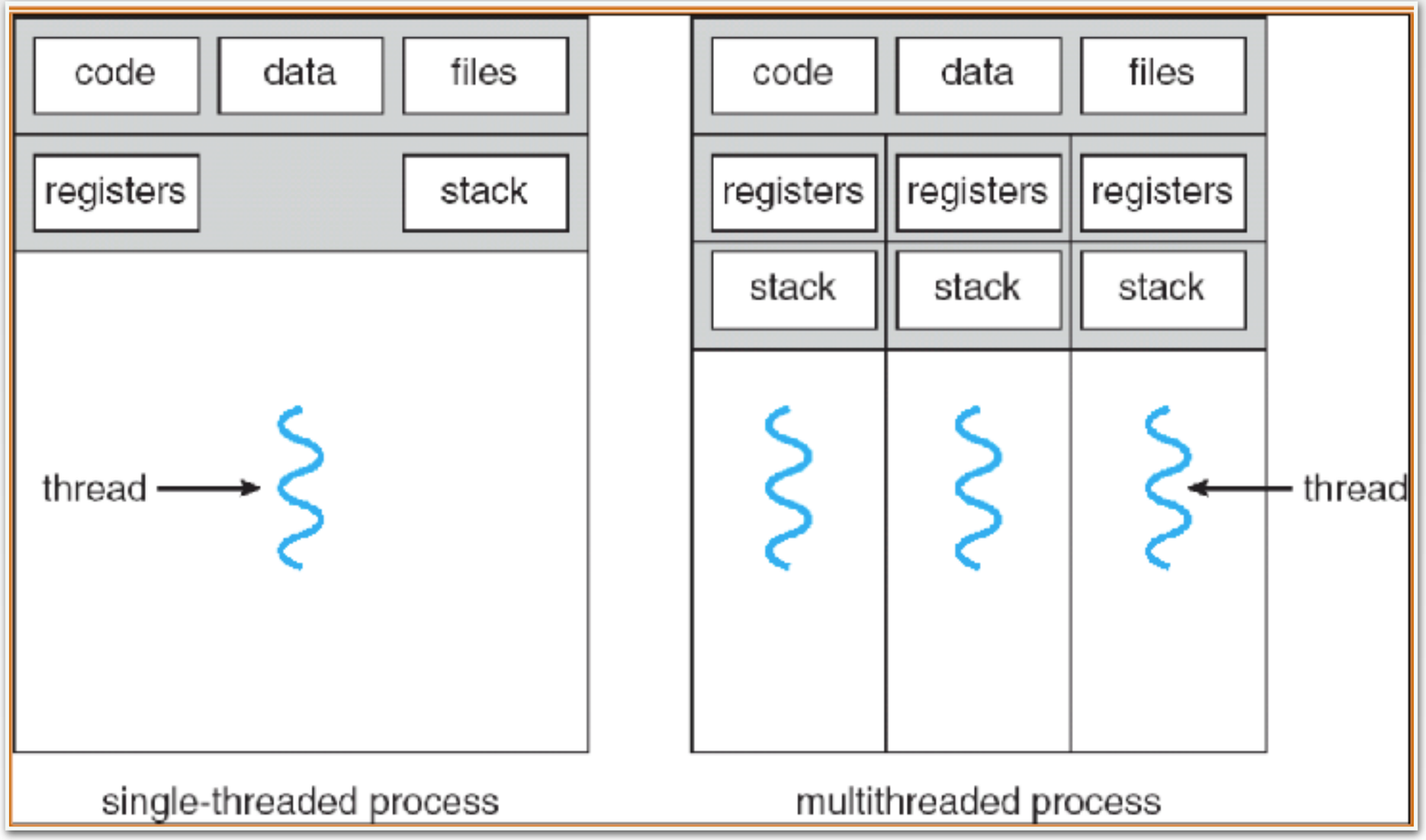
Process (cont.)

- Why processes?
  - Protected from each other
  - OS Protected from them
  - Processes provides memory protection
  - Threads more efficient than processes(later)
- Fundamental tradeoff between protection and efficiency

Process (cont.)

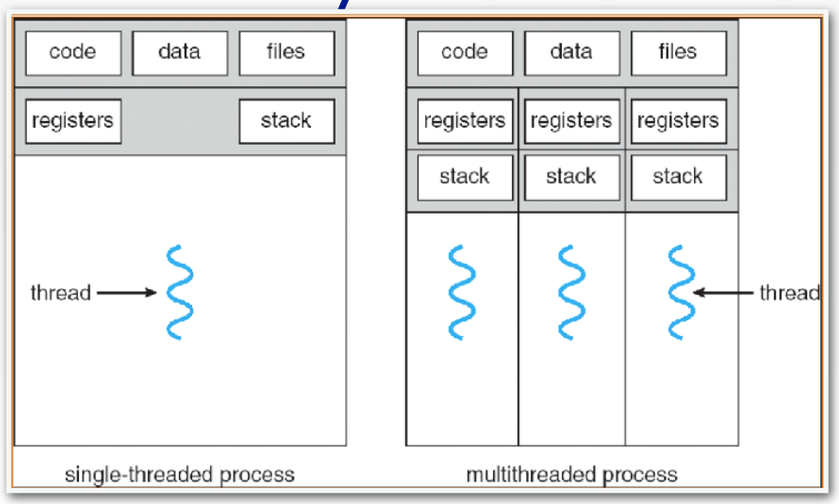
- Application instance consists of one or more processes

Single and Multithreaded Processes



Single and Multithreaded Processes

- Threads encapsulate concurrency
  - "Active" component
  - Address spaces encapsulate protection
    - "Passive " part
    - Keeps buggy program from trashing the system
  - Why have multiple threads per address space



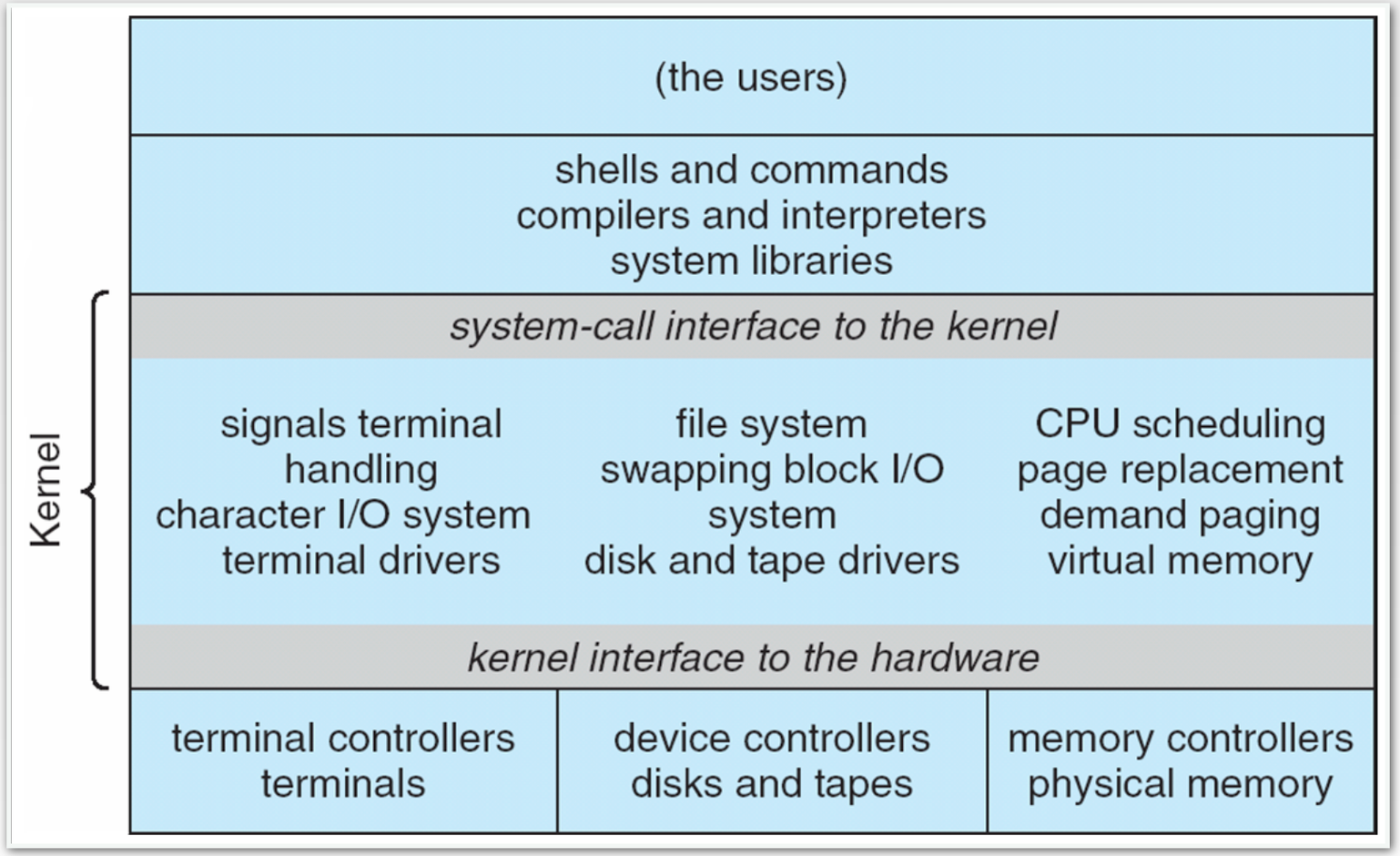
Four Fundamental OS Concepts(4)

- Dual Mode operation/Protection
  - Only the "system" has the ability to access certain resources
  - OS and HW protected from user programs
  - User programs isolated from one another
  - Virtual address vs. Physical address

Dual Mode Operation

- Hardware provides at least two modes:
  - Kernel mode
  - User mode
- What is needed to support "dual mode"
  - A bit of state (user/system mode bit)
  - Certain operations only permitted in kernel
  - User -> Kernel transition
  - Kernel -> User transition

Example: UNIX System Structure

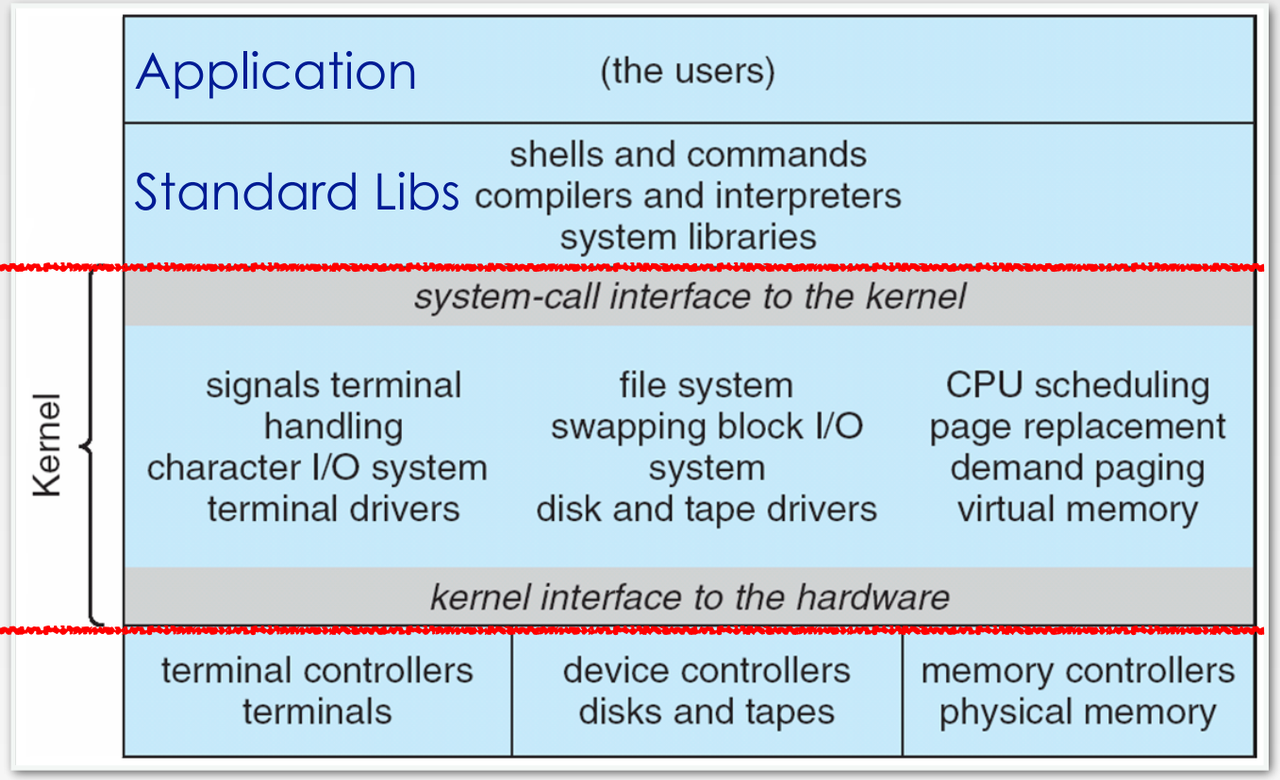


Example: UNIX System Structure

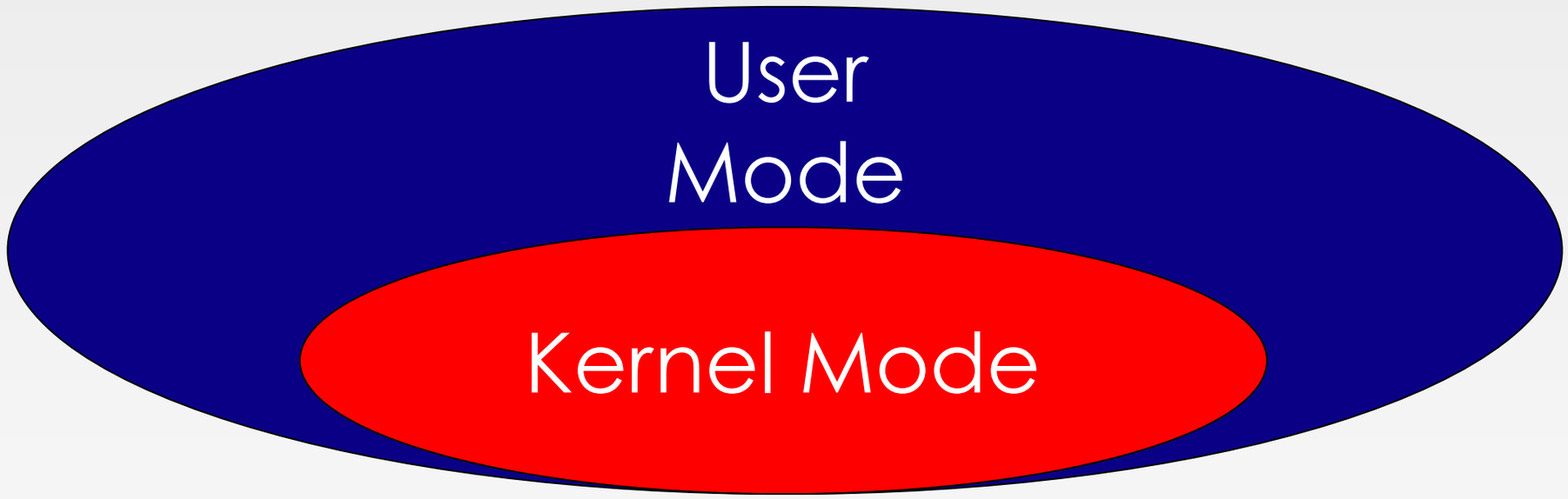
User Mode

Kernel Mode

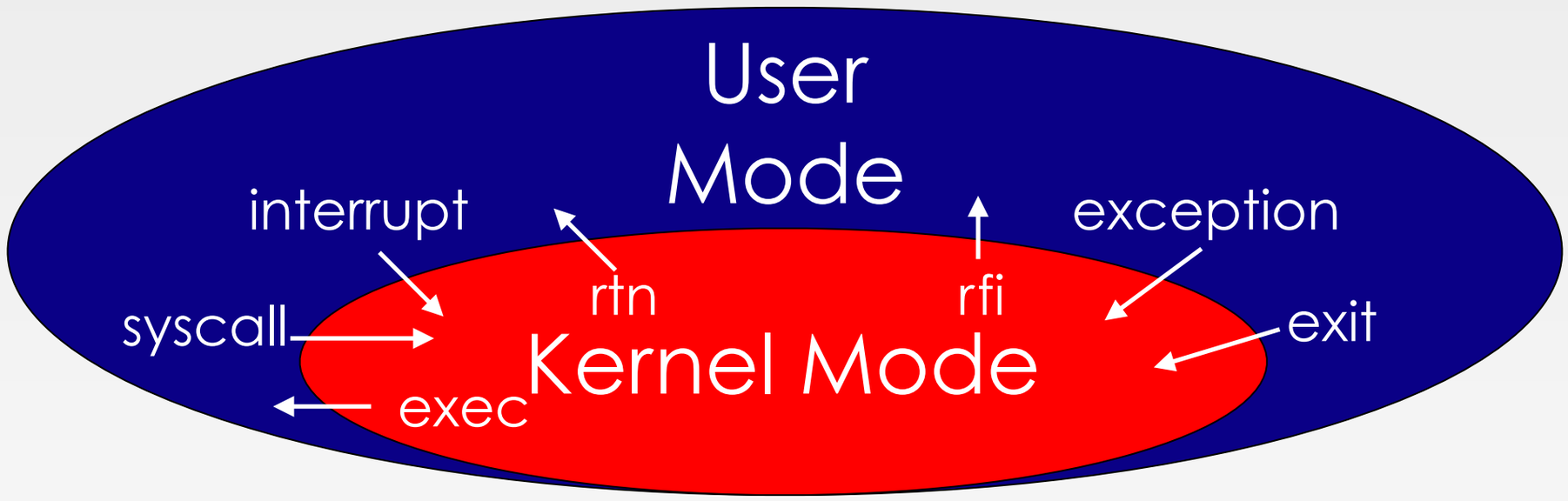
Hardware



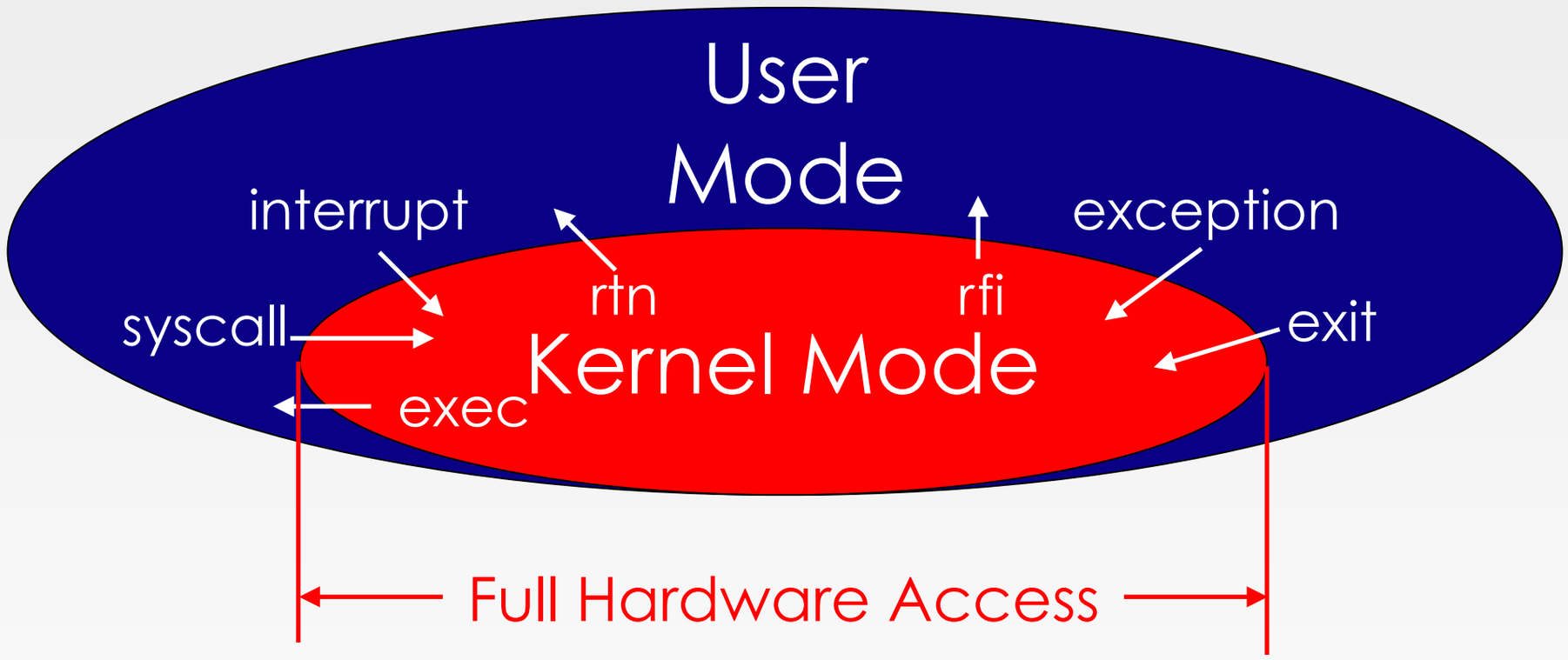
User/Kernel Mode



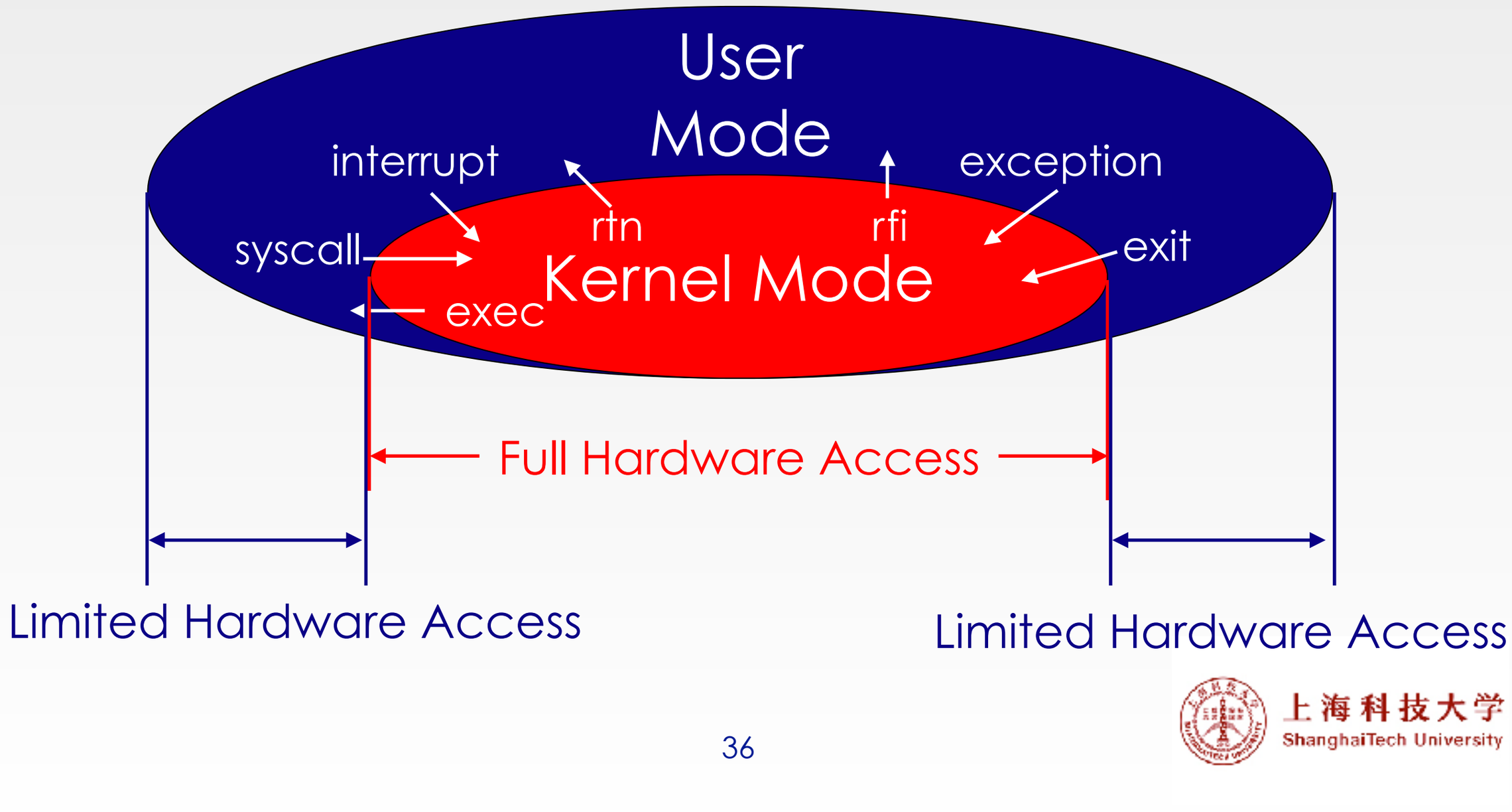
User/Kernel Mode



User/Kernel Mode

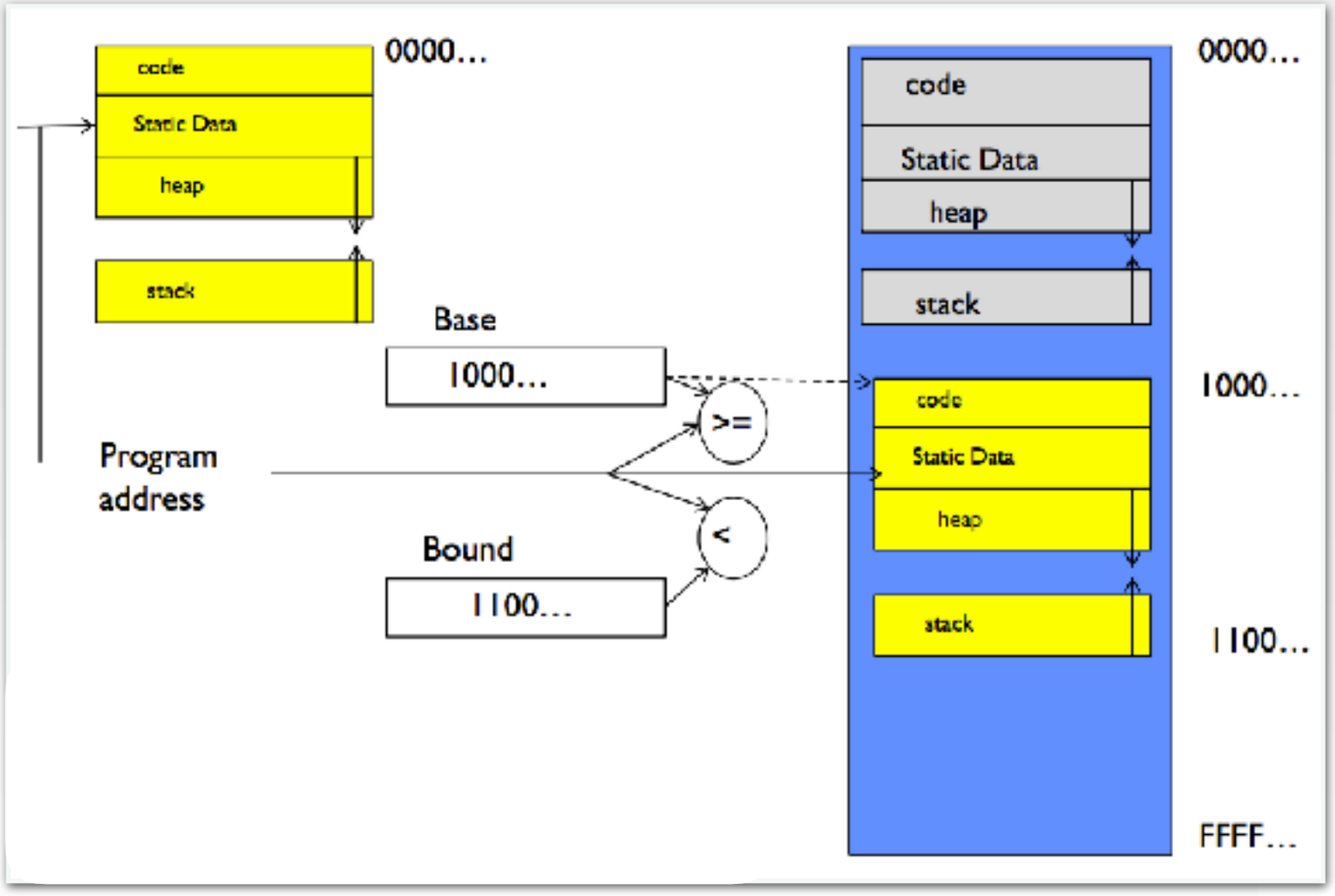


User/Kernel Mode



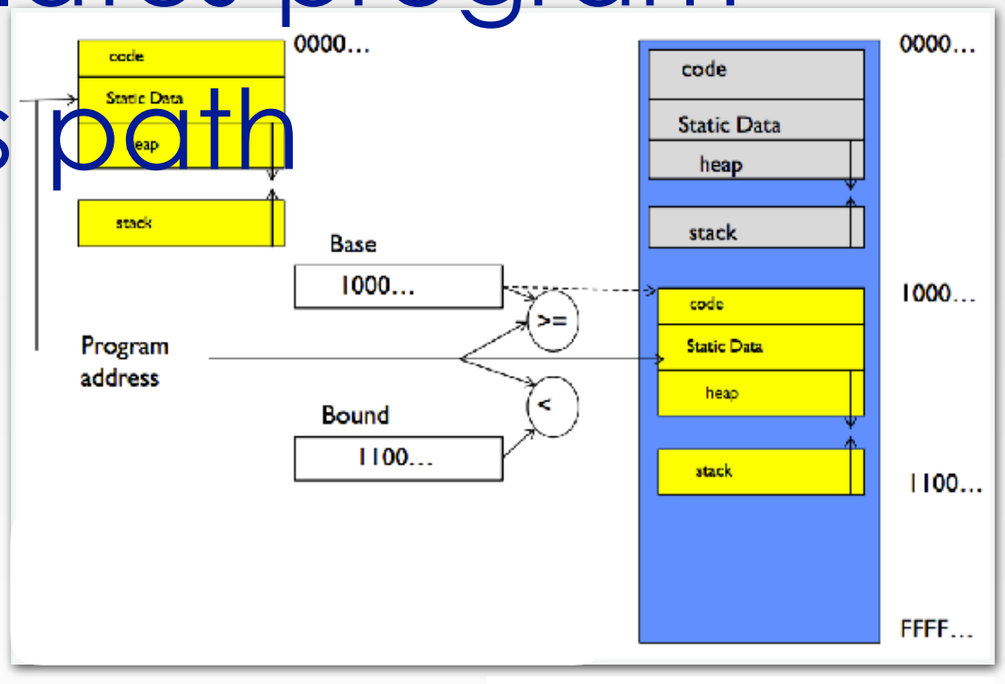
Simple Protection

- Base and Bound (B&B)



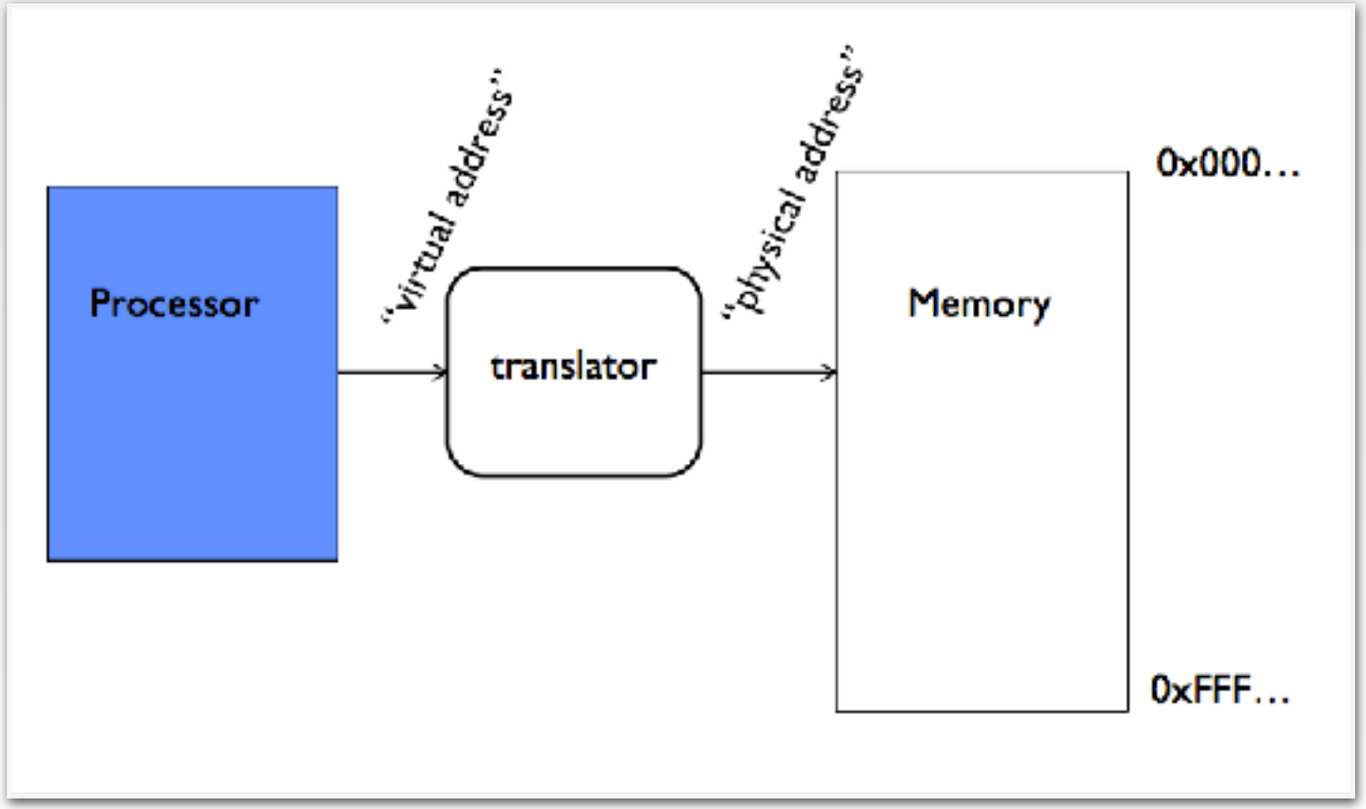
Simple Protection

- Base and Bound (B&B)
- Requires relocating loader
- Still protects OS and isolates program
- No addition on address path

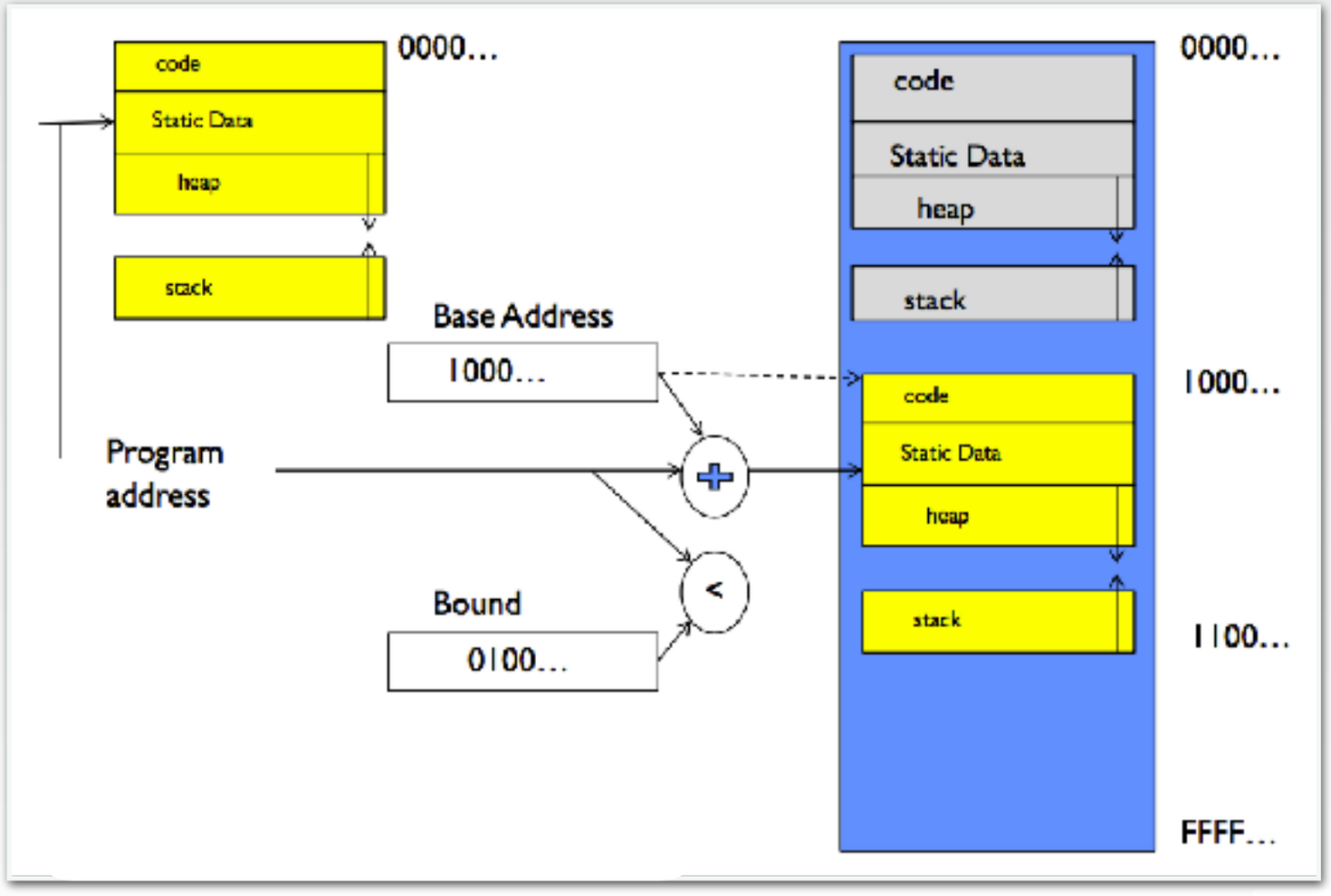


Another Idea

- Address Space Translation
  - Distinct from the physical memory space

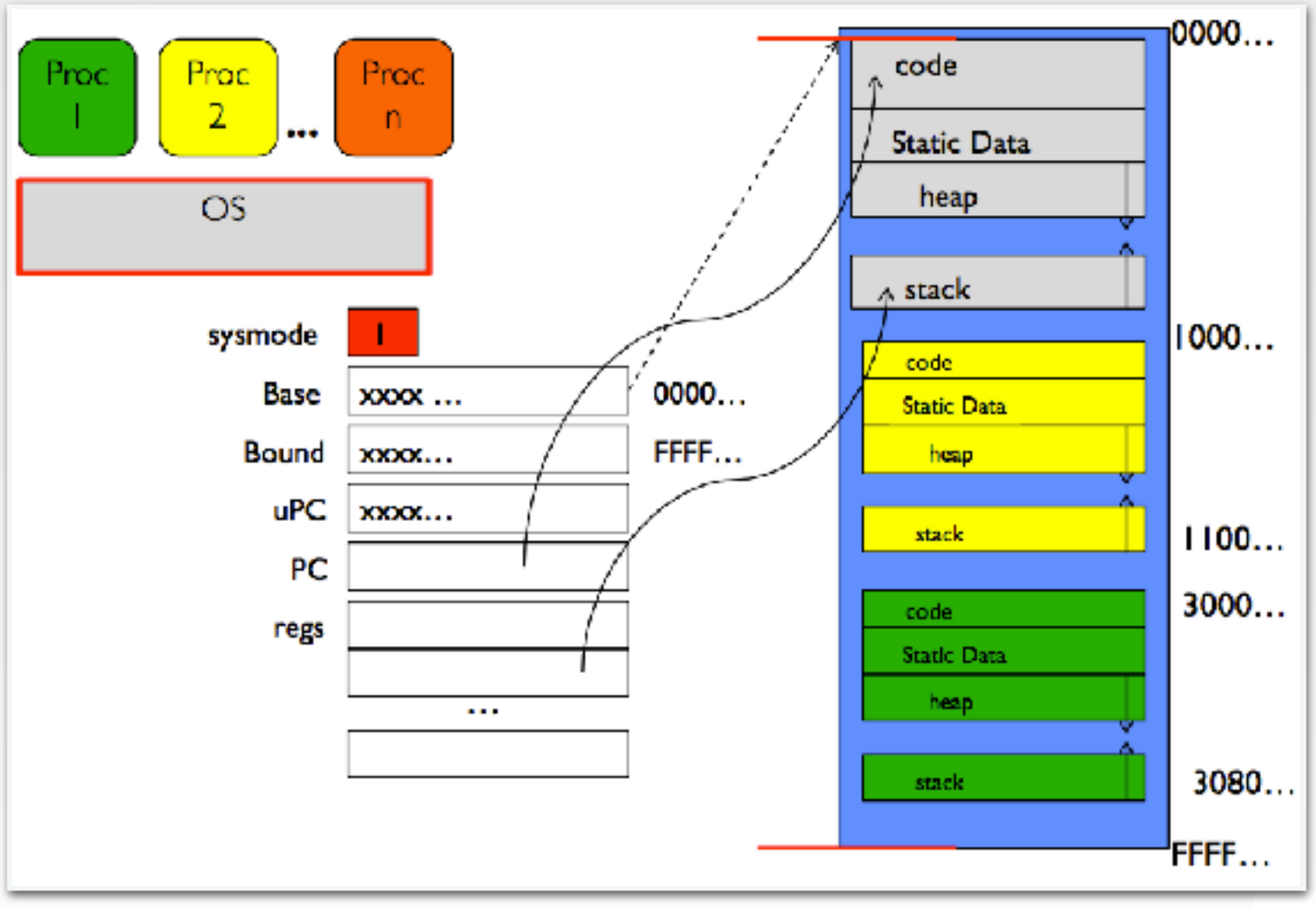


- A simple address translate + B&B



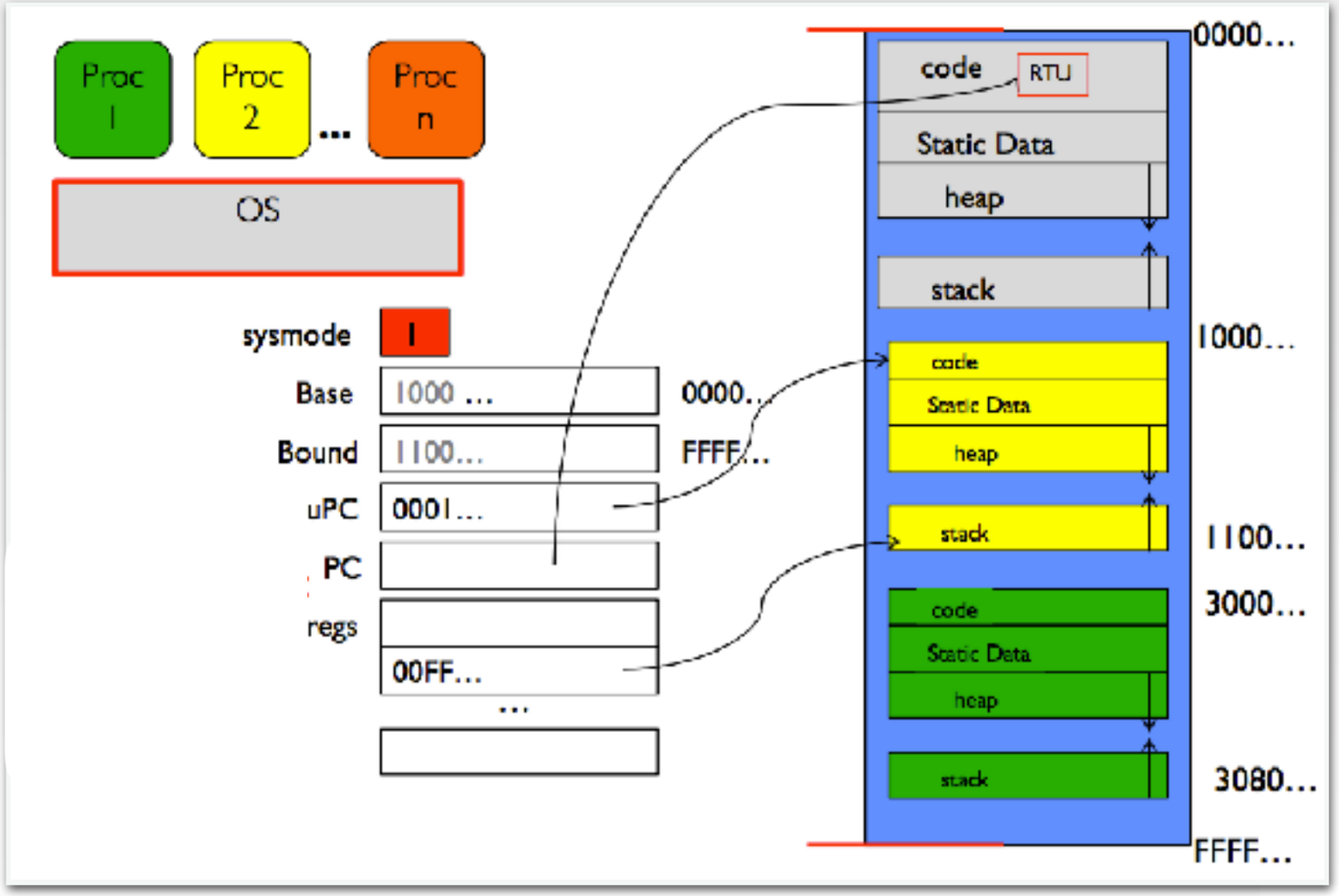
Trying it Together: Simple B&B

- OS loads process



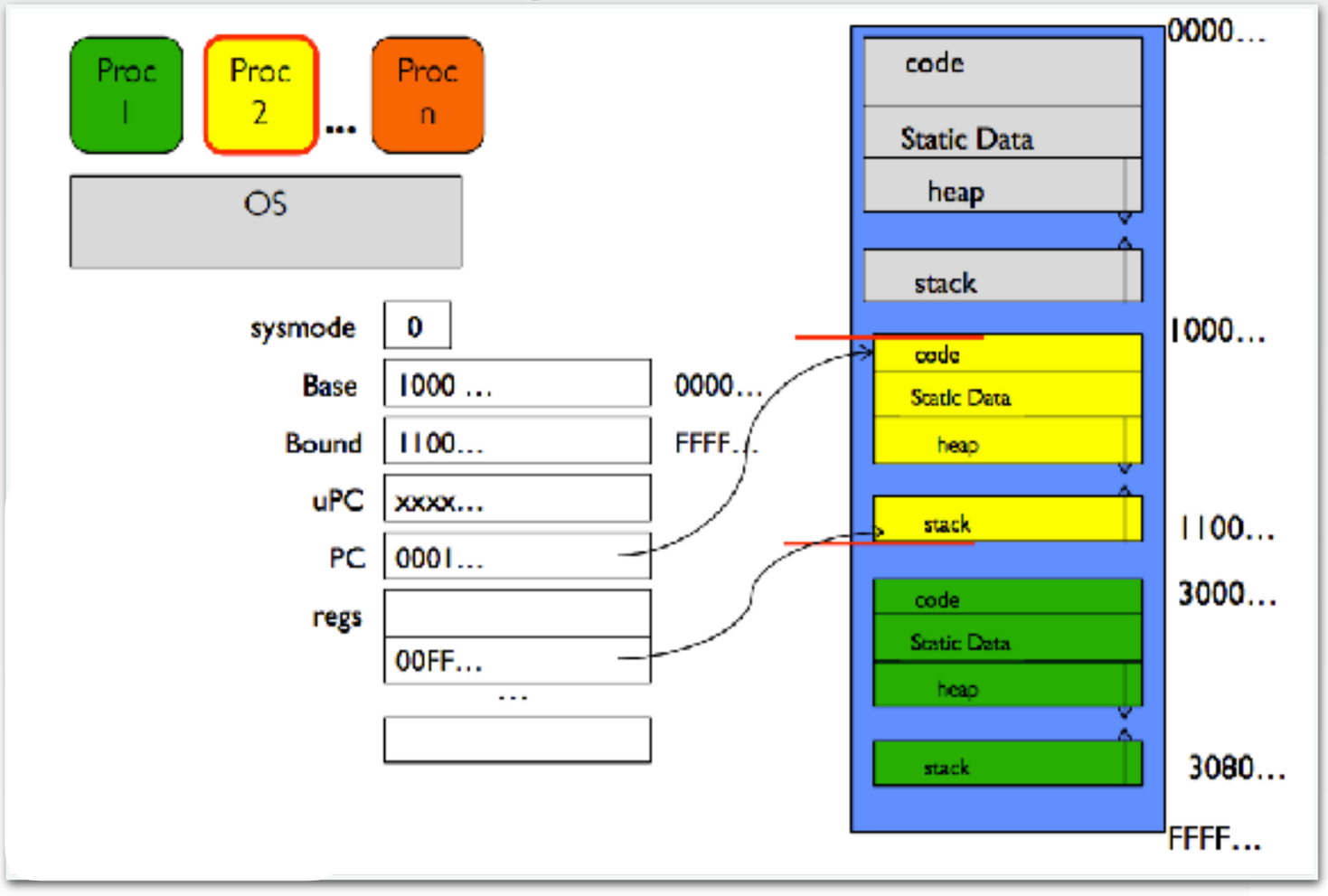
Trying it Together: Simple B&B

- OS gets ready to execute process



Trying it Together: Simple B&B

- User code running



Three Types of Mode Transfer

- Syscall
- Interrupt
- Trap/Exception

Syscall

- Process requests a system service
  - e.g., exit
- Like a function call
  - but outside the process
- Does NOT have the address
- Like a RPC
- Marshall the syscal id and args in registers and exec syscal

Interrupt

- External asynchronous event triggers context switch
- e.g., Timer, I/O device
- Independent of user process

Trap/Exception

- Internal synchronous event in process triggers context switch
- e.g., Protection violation
  - segmentation fault
  - Divide by zero

Three Types of Mode Transfer

- Syscall
- Interrupt
- Trap/Exception

All three are an UN-PROGRAMMED CONTROL TRANSFER

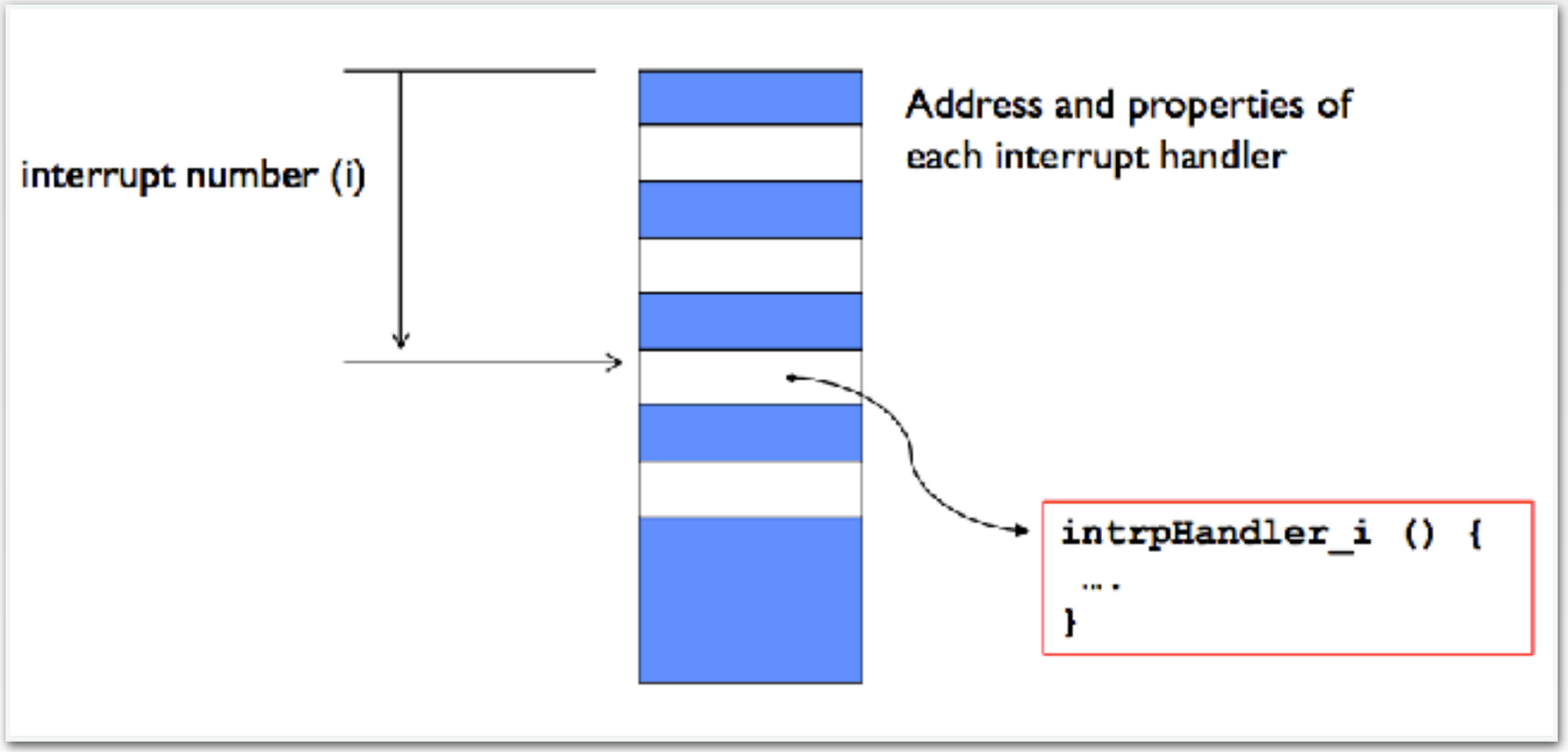
Three Types of Mode Transfer

- Syscall
- Interrupt
- Trap/Exception

How do we get the system target address of the "UN-PROGRAMMED CONTROL TRANSFER"?

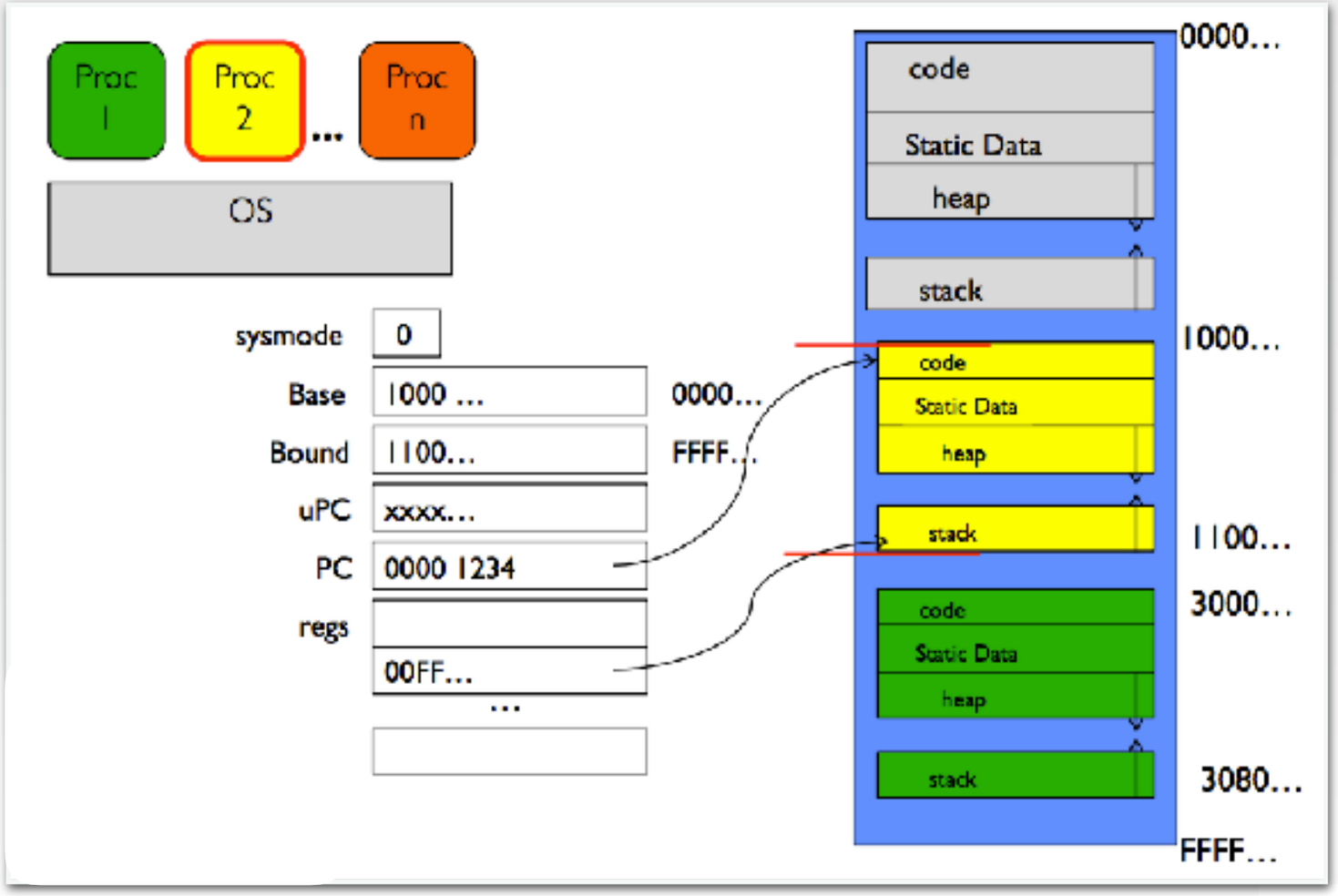
How do we get the system target address of the "UN-PROGRAMMED CONTROL TRANSFER"?

- Interrupt Vector



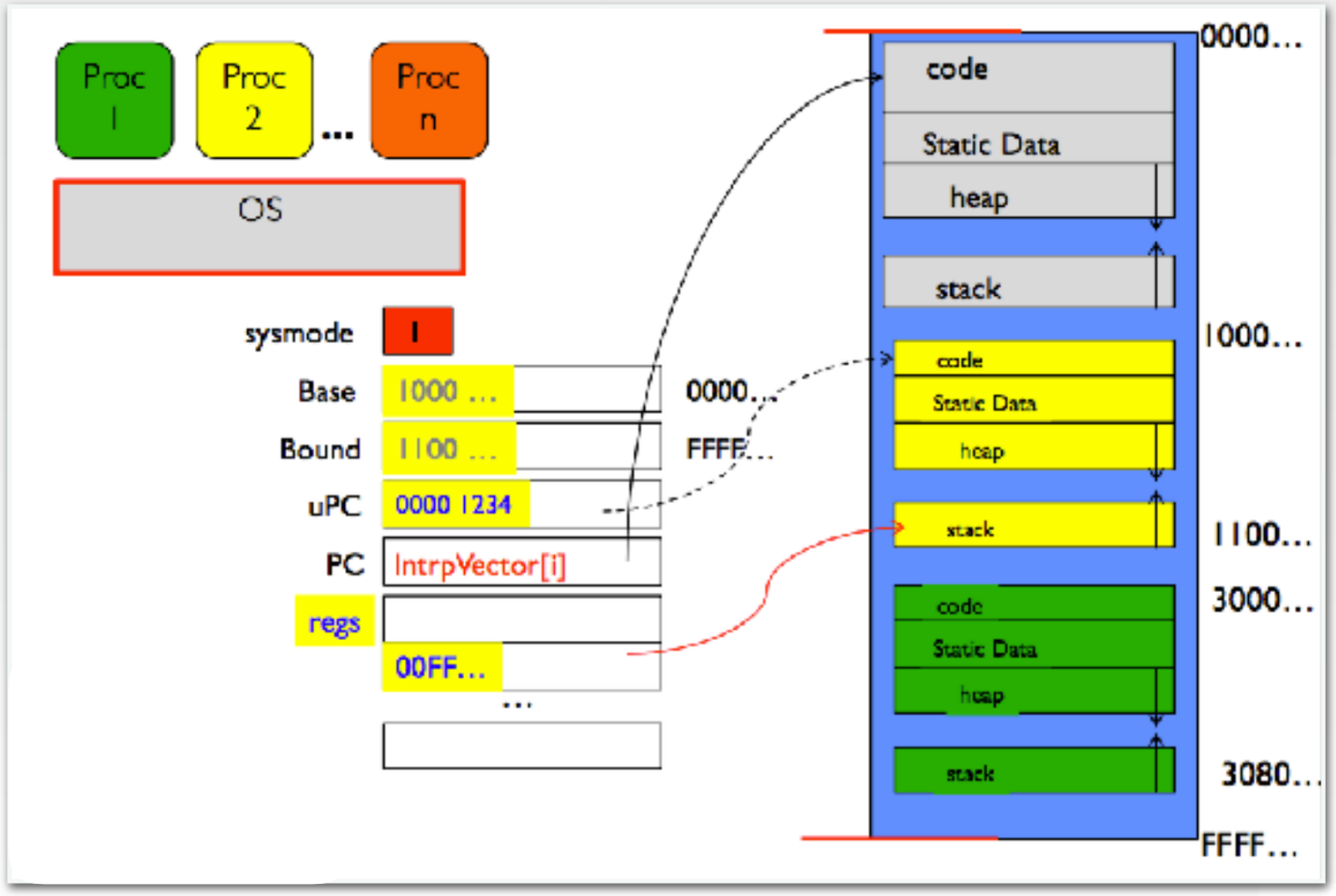
How do we get the system target address of the "UN-PROGRAMMED CONTROL TRANSFER"?

- Simple B&B: User→Kernel



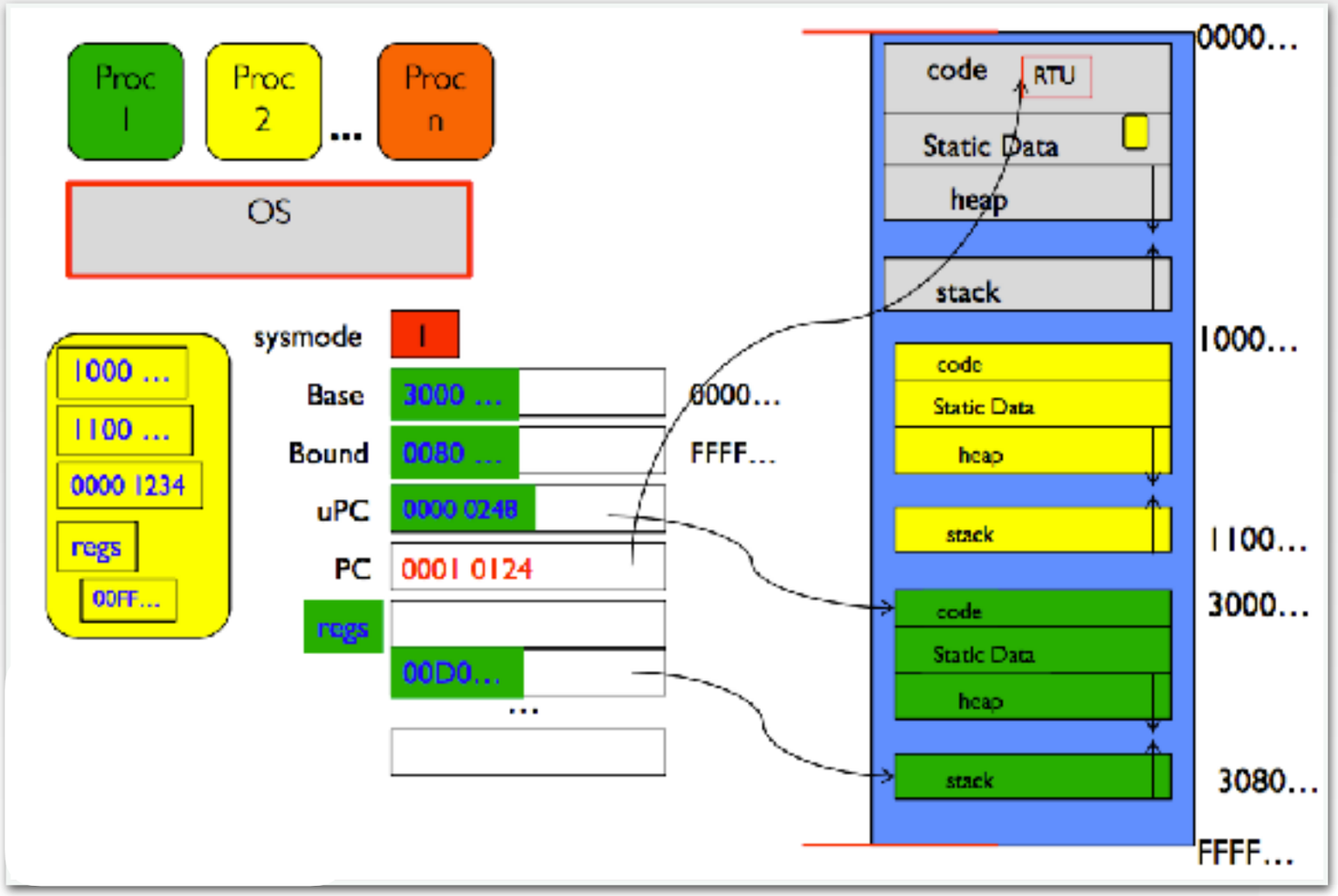
How do we get the system target address of the "UN-PROGRAMMED CONTROL TRANSFER"?

- Simple B&B: Interrupt



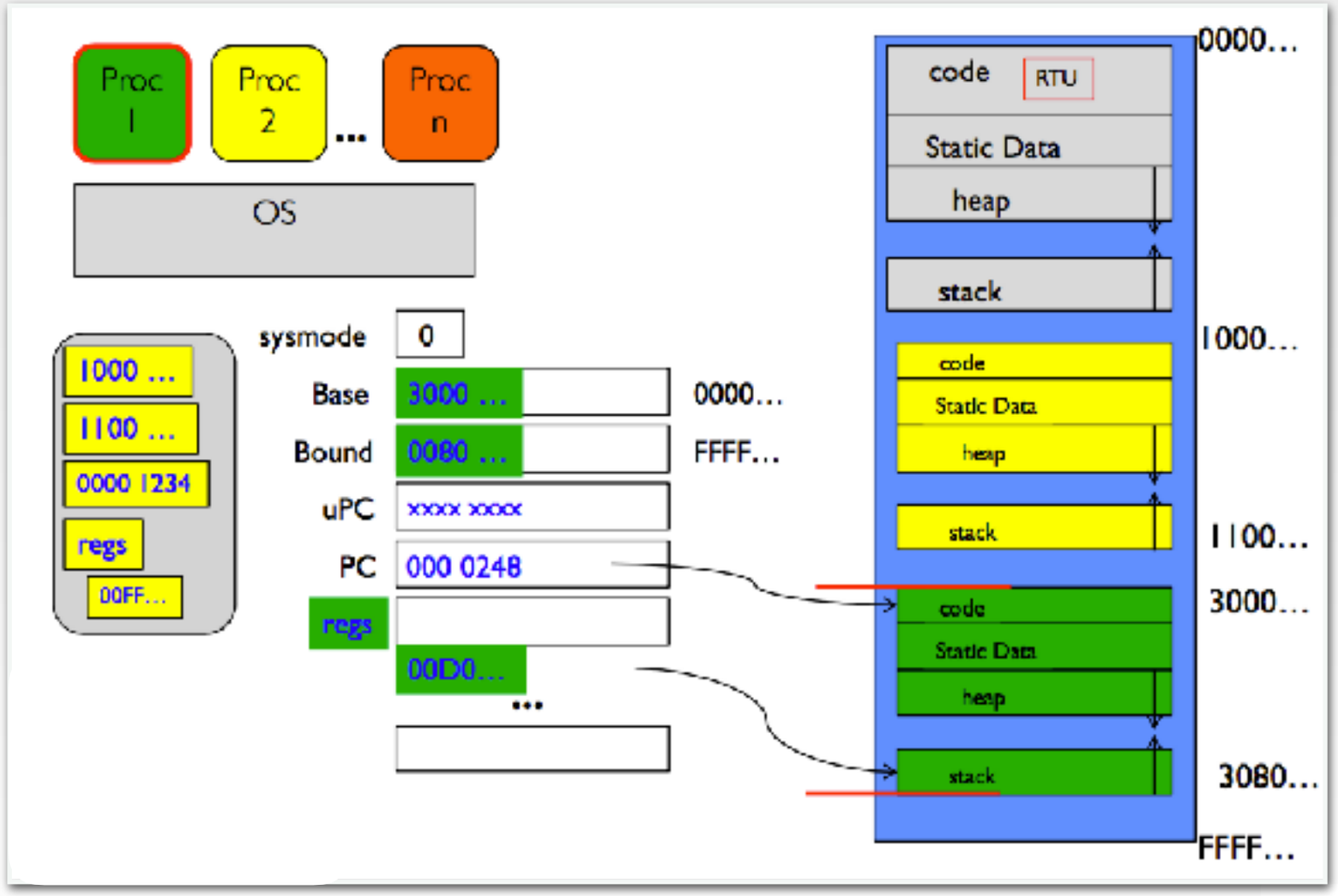
How do we get the system target address of the "UN-PROGRAMMED CONTROL TRANSFER"?

- Simple B&B: Switch User Process



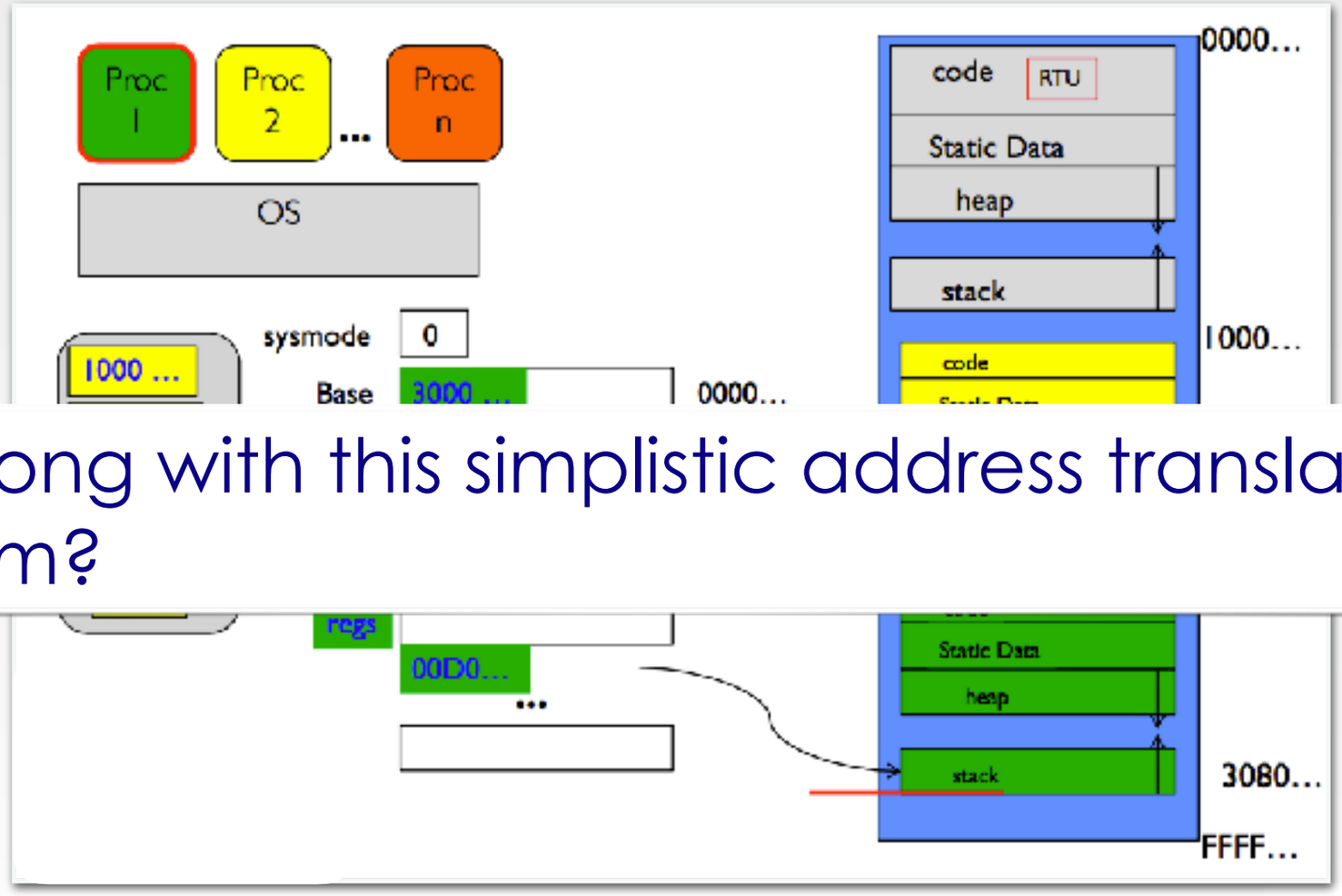
How do we get the system target address of the "UN-PROGRAMMED CONTROL TRANSFER"?

- Simple B&B: Resume



How do we get the system target address of the "UN-PROGRAMMED CONTROL TRANSFER"?

- Simple B&B: Resume

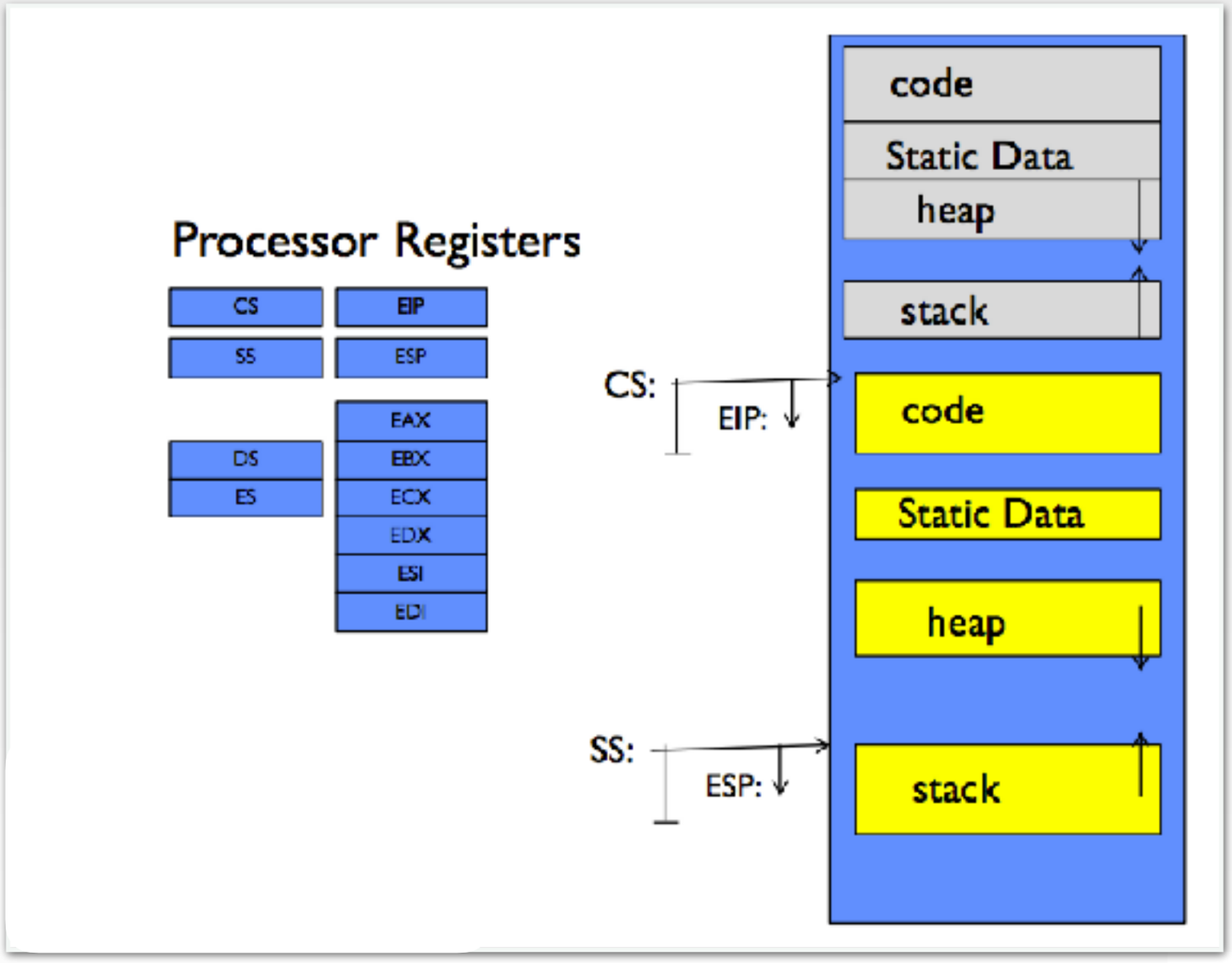


What's wrong with this simplistic address translation mechanism?

What's wrong with this simplistic address translation mechanism?

- Fragmentation
  - Contiguous block memory
  - After a while, memory becomes fragmented
- Sharing
  - Very hard to share any data
    - between processes, or
    - between process and kernel
  - Simple segmentation

x86 Segments and Stacks

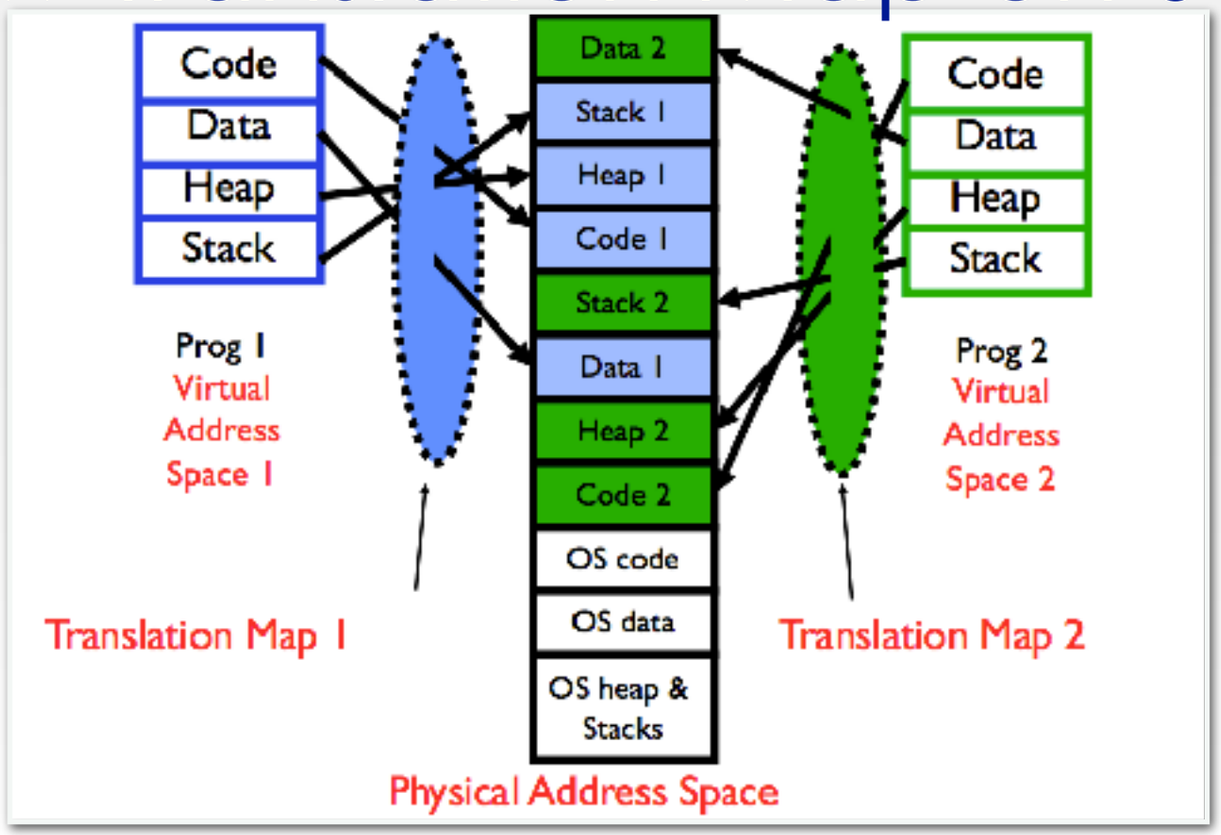


Virtual Address Translation

- Simpler
- More useful schemes
- Illusion of Big Flat Address Space
  - Break it into pages
  - More on this later

Virtual Address Translation (cont.)

- Providing illusion of separate address space:
- Load new Translation Map on Switch



Running Many Programs?

- We have the Basic Mechanism to
  - Switch between user processes and kernel
  - Kernel can switch among user processes
  - Protect OS from user processes and processes from each other
- But…

Running Many Programs?

- How do we decide which user process to run?
- How do we represent user processes in OS?
- How do we pack up the process and set it aside?
- How do we get a stack and heap for the kernel?
- Aren't we wasting a lot of memory?
- …

Process Control Block

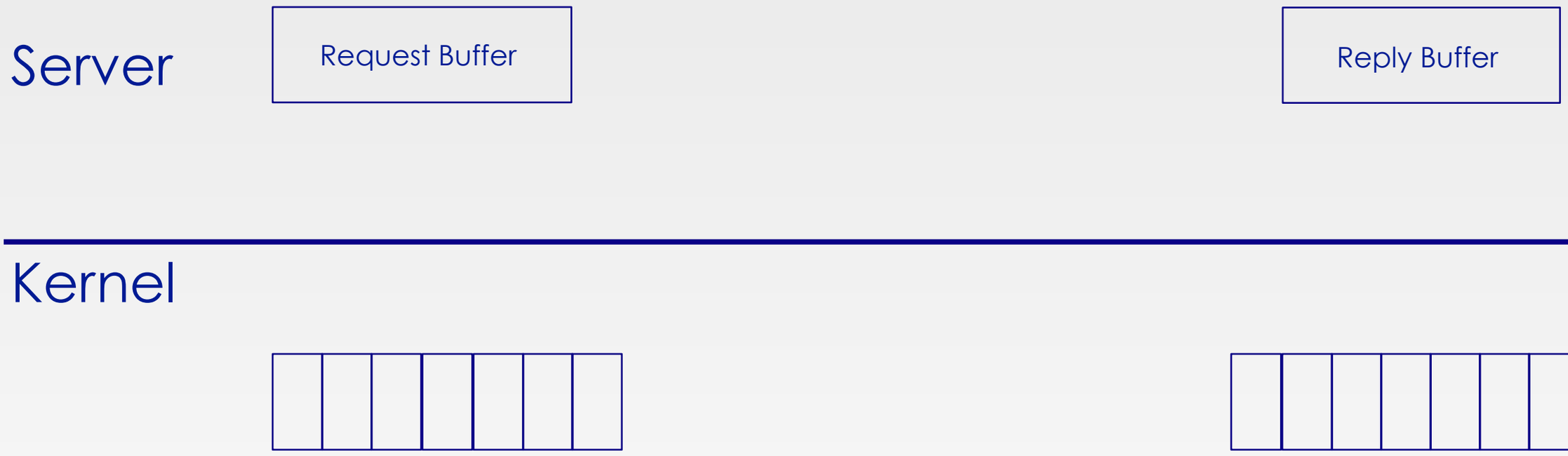
- Kernel represents each process as a PCB
  - Status (running, ready, blocked, …)
  - Register state (when not ready)
  - Process ID (PID), User, Executable, Priority, …
  - Execution time, …
  - Memory space, translation, …
- Kernel Scheduler
- Scheduling algorithm selects the next one to run

Scheduler

```
if ( readyProcesses(PCBs) ) {
        nextPCB = selectProcess(PCBs);
        run( nextPCB );
} else {
        run_idle_process();
}
```

- Scheduling
- Lots of different scheduling polices provide:
  - Fairness, or
  - Realtime guarantees, or
  - Latency optimization, …

Putting it Together: Web Server



Putting it Together: Web Server

Server

Request Buffer

Reply Buffer

1. Network Socket Read

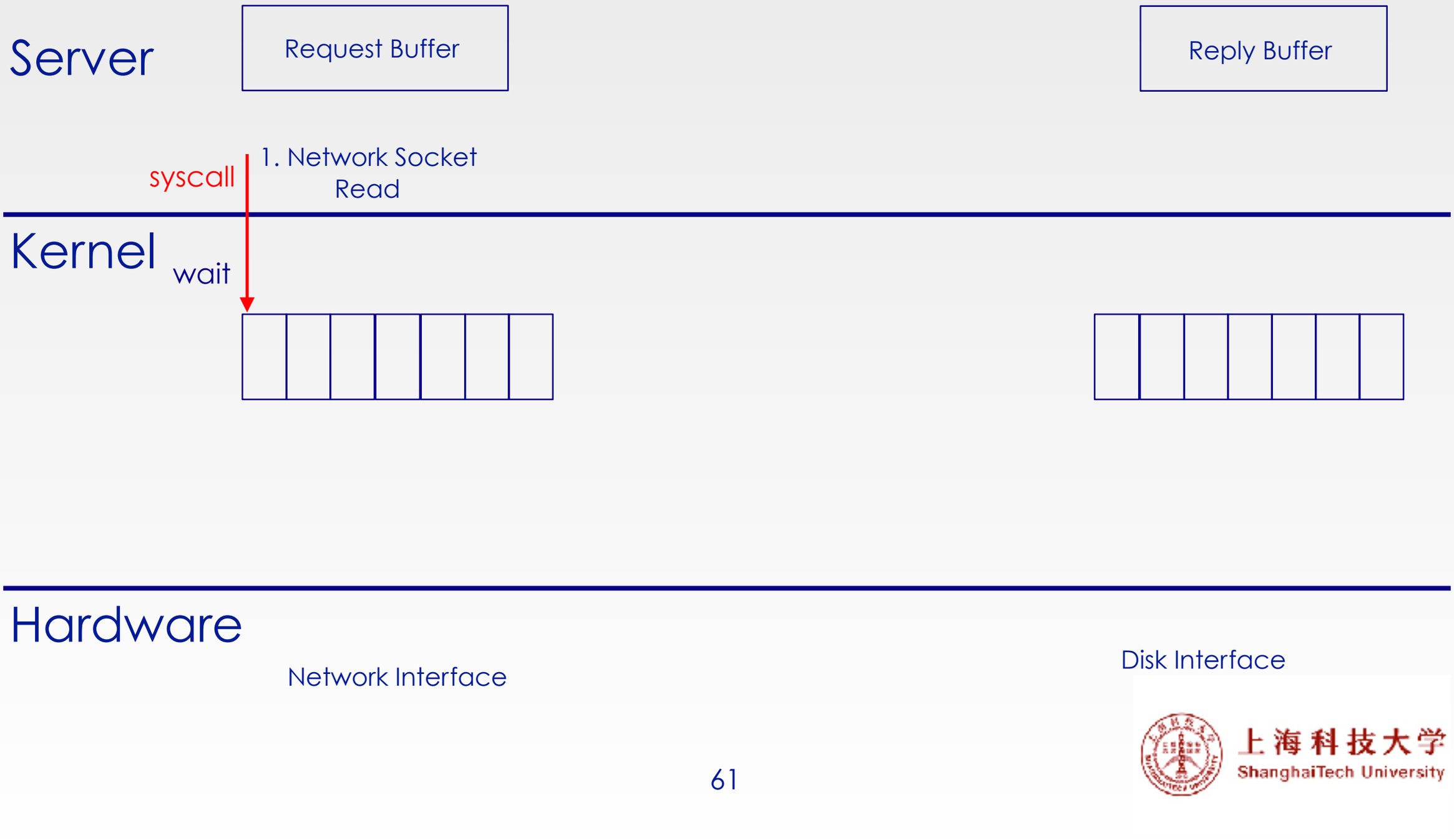
Kernel

Hardware

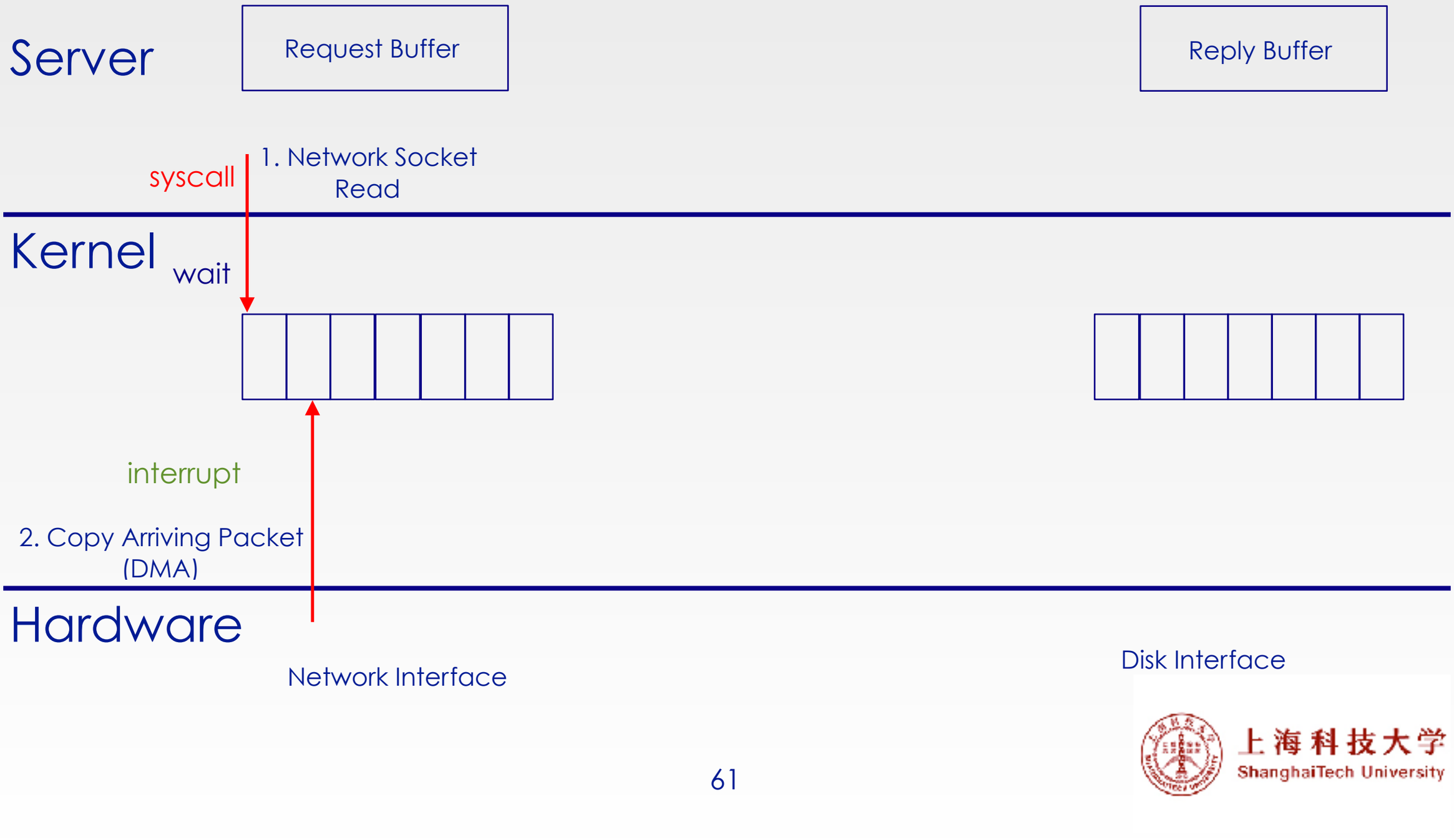
Network Interface

Disk Interface

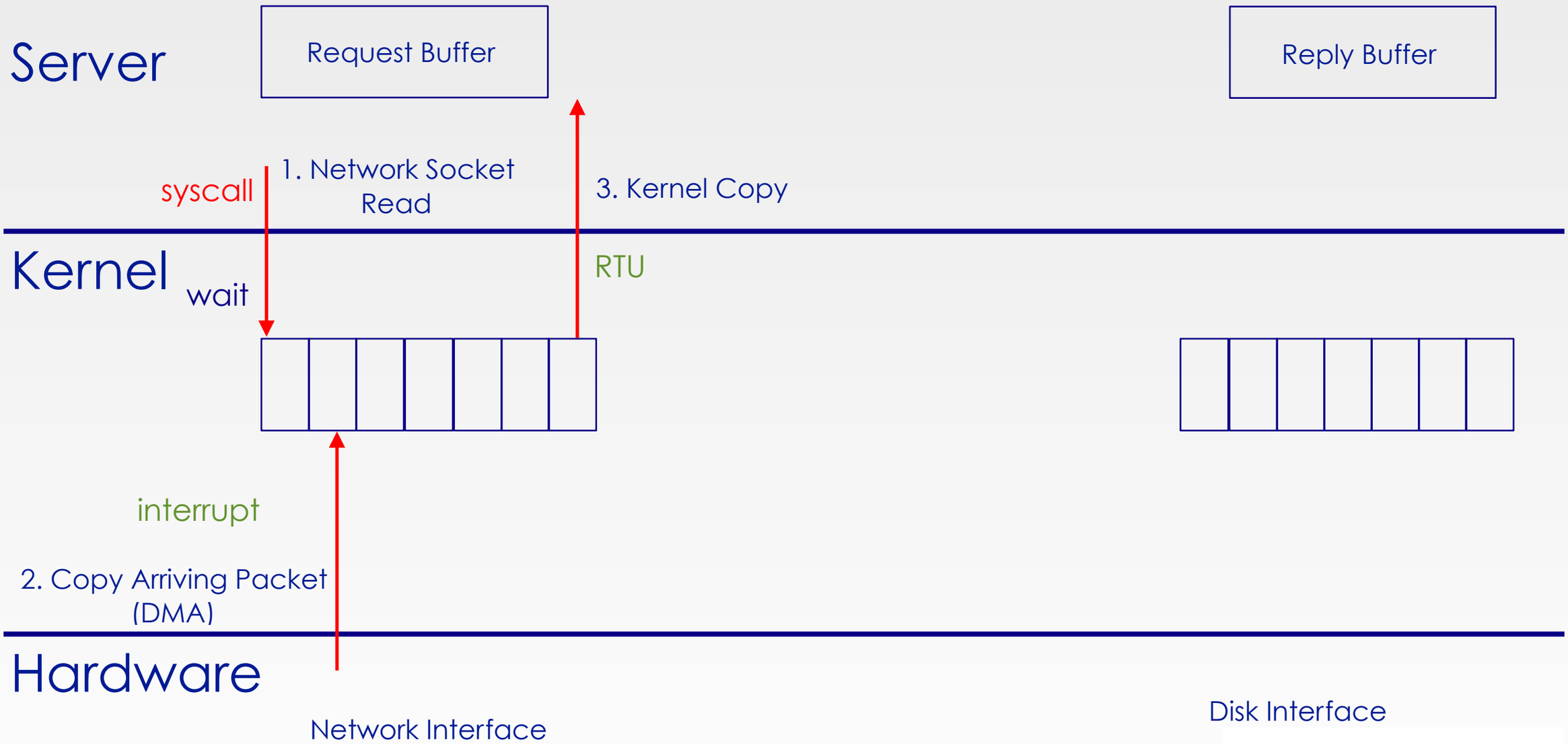
Putting it Together: Web Server



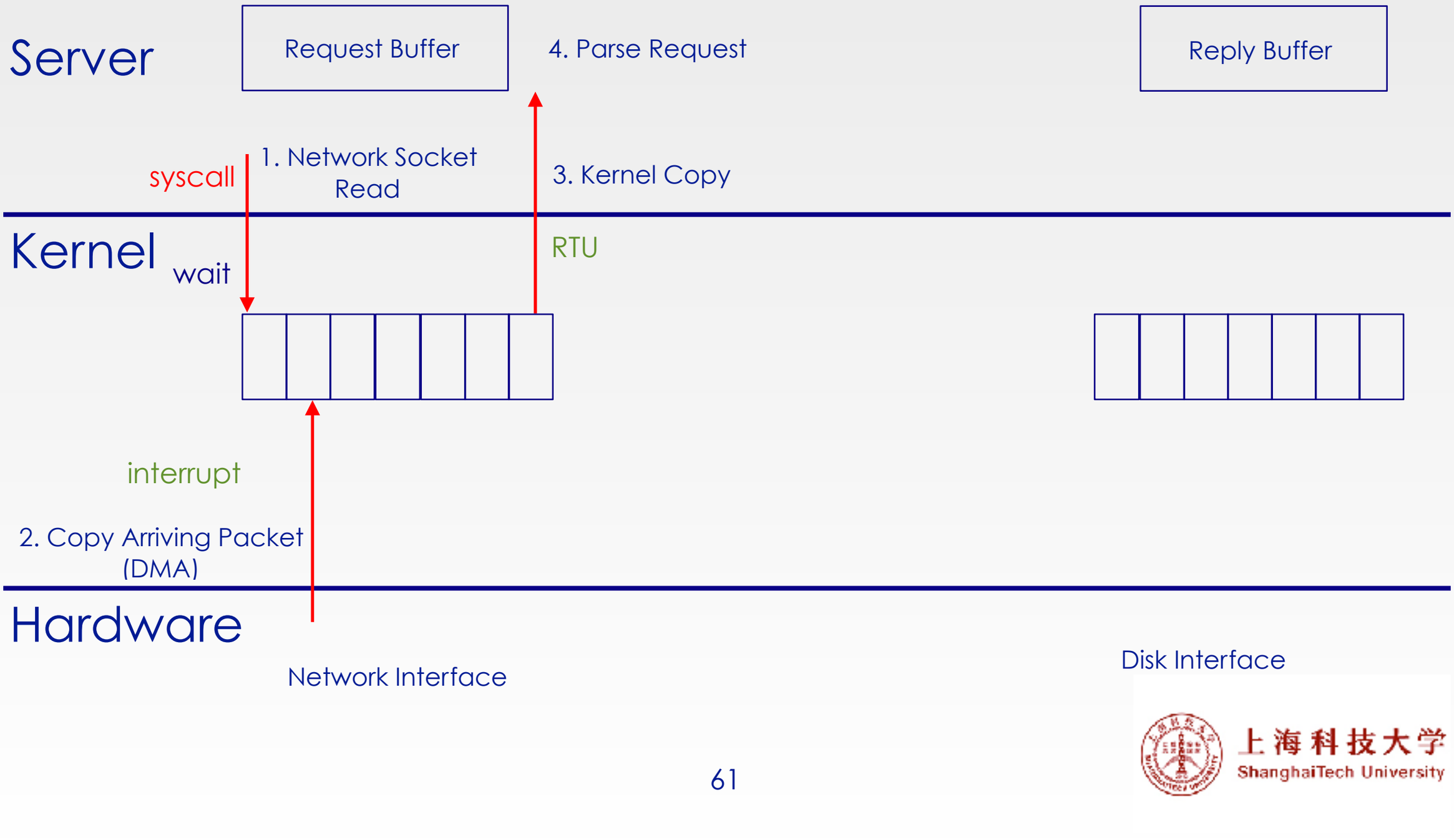
Putting it Together: Web Server



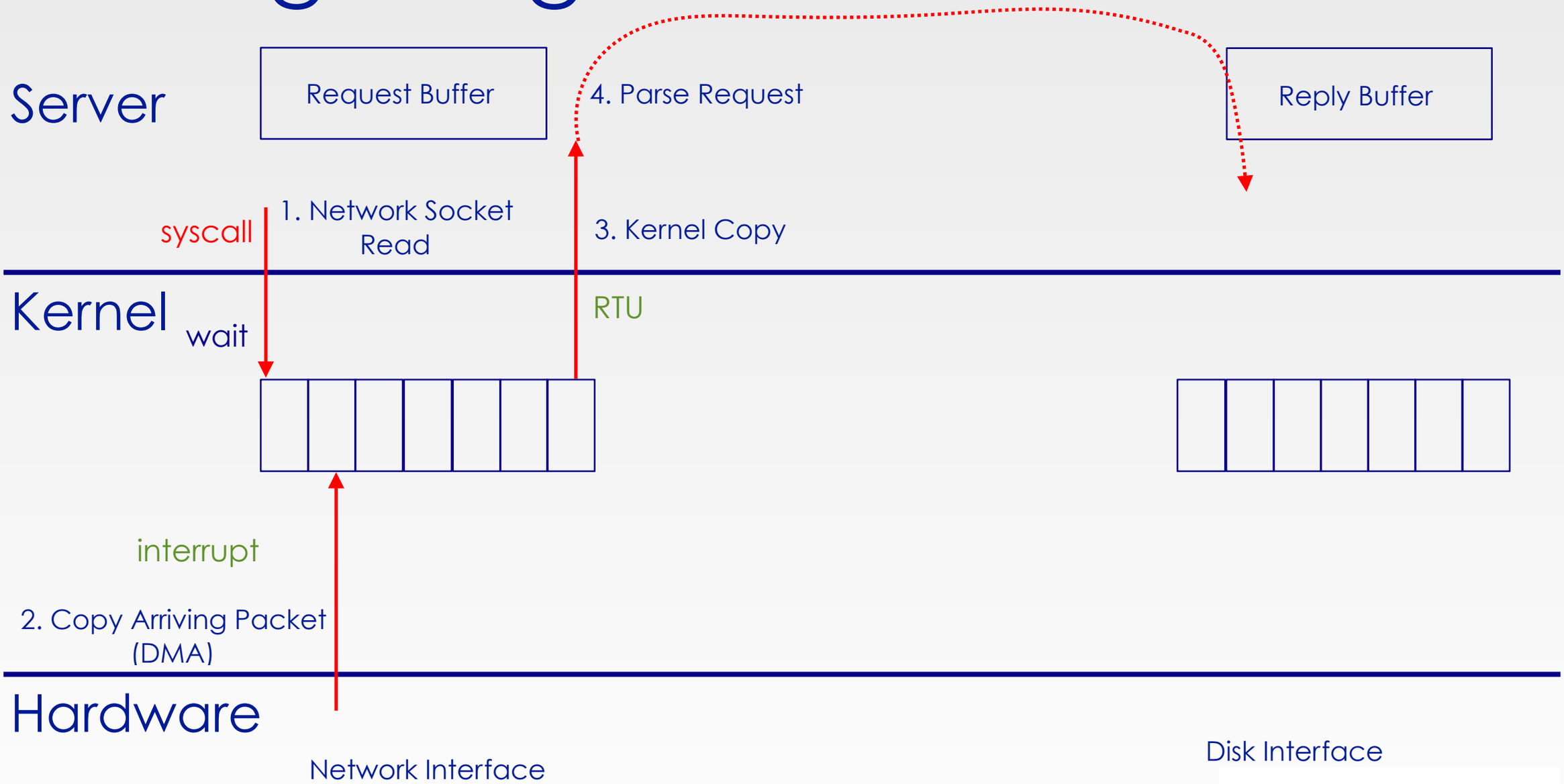
Putting it Together: Web Server



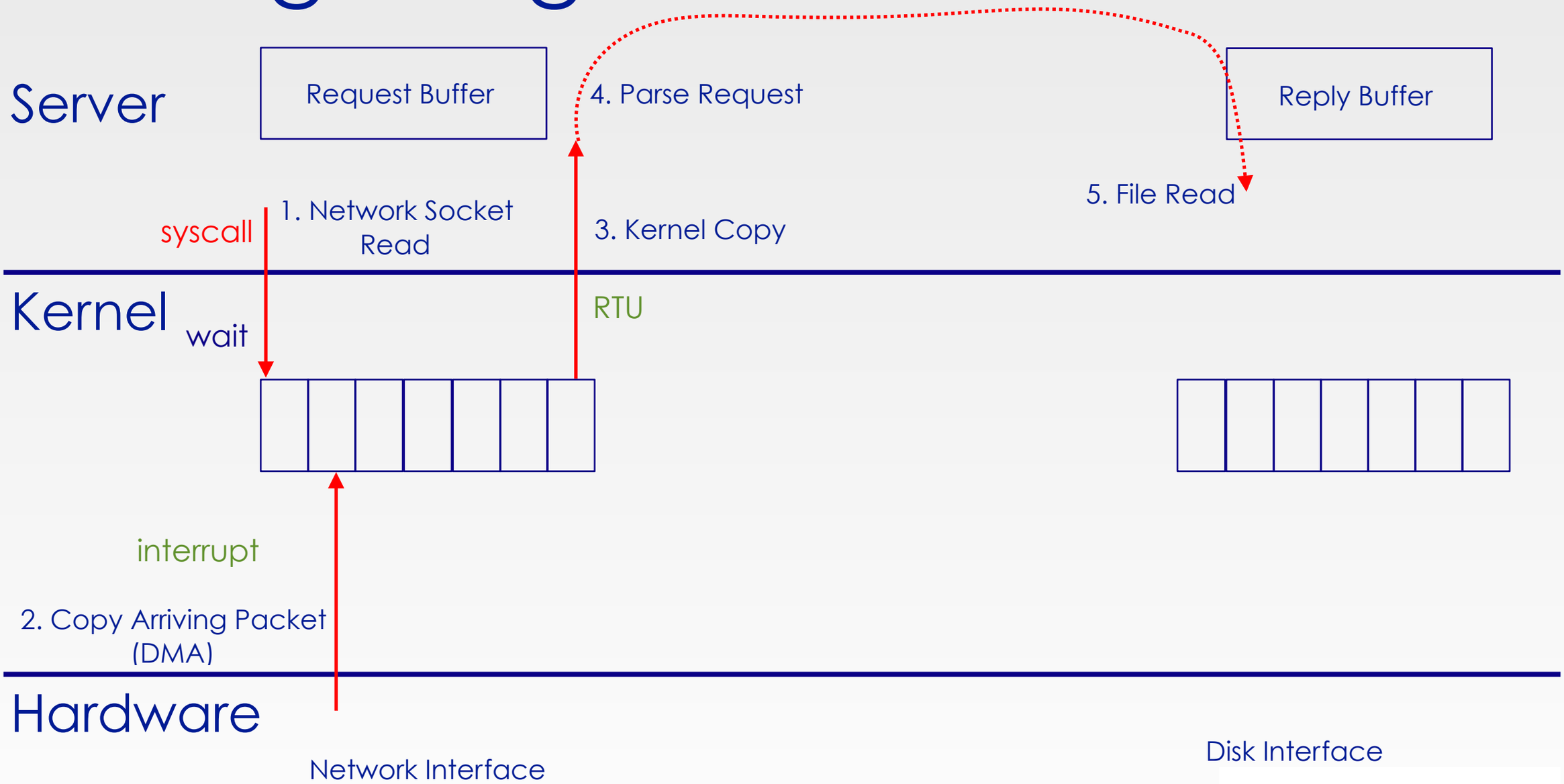
Putting it Together: Web Server



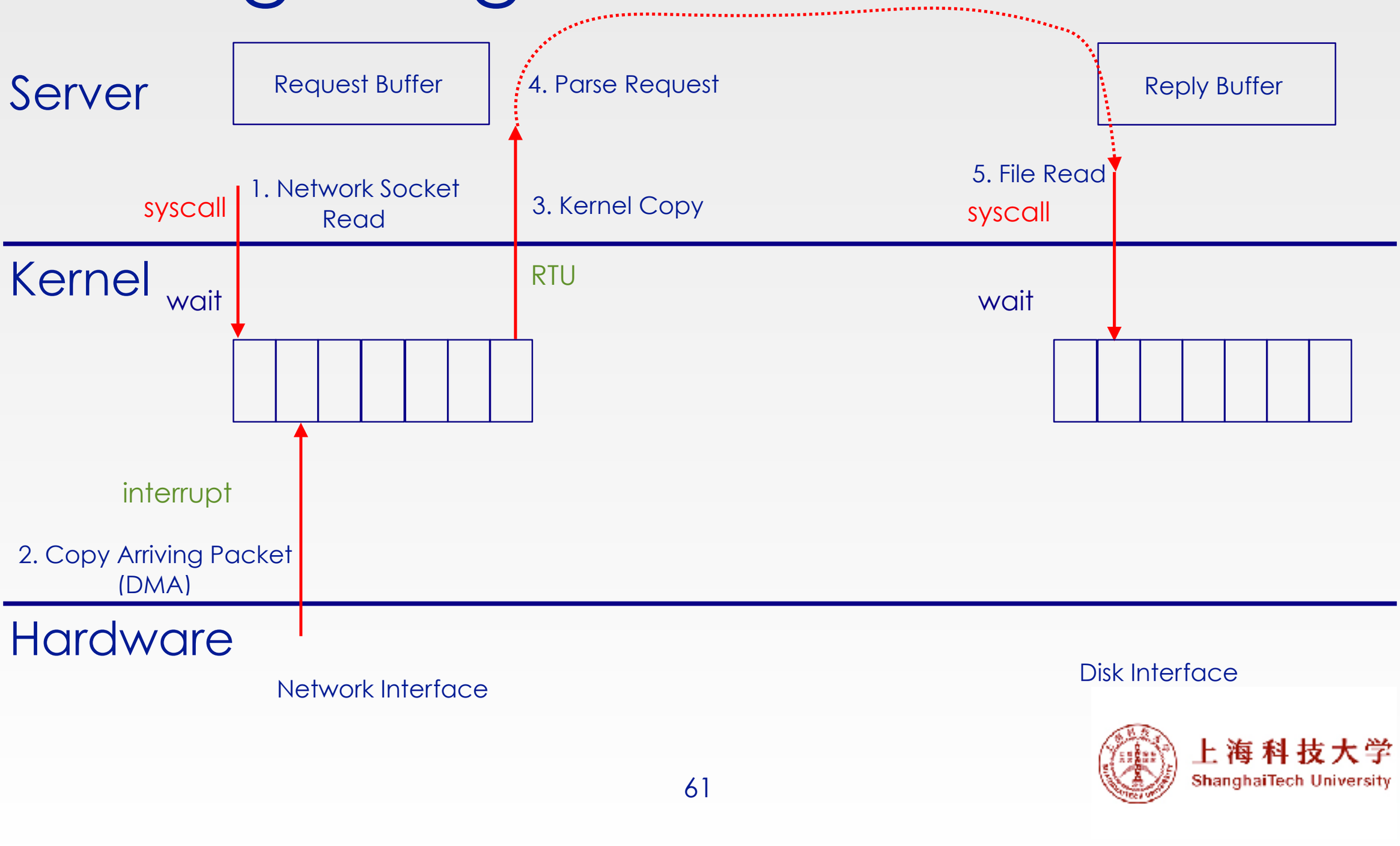
Putting it Together: Web Server



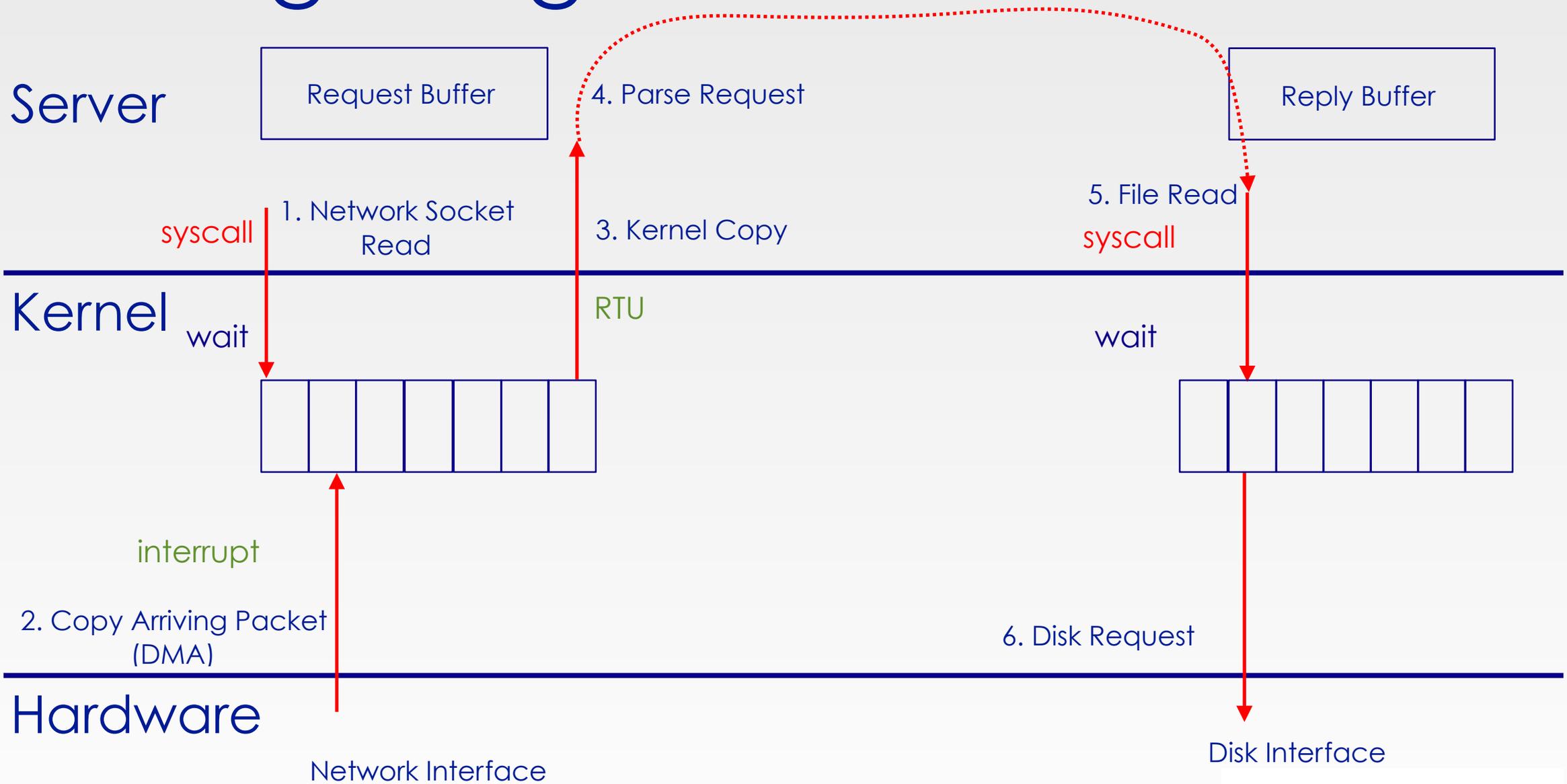
Putting it Together: Web Server



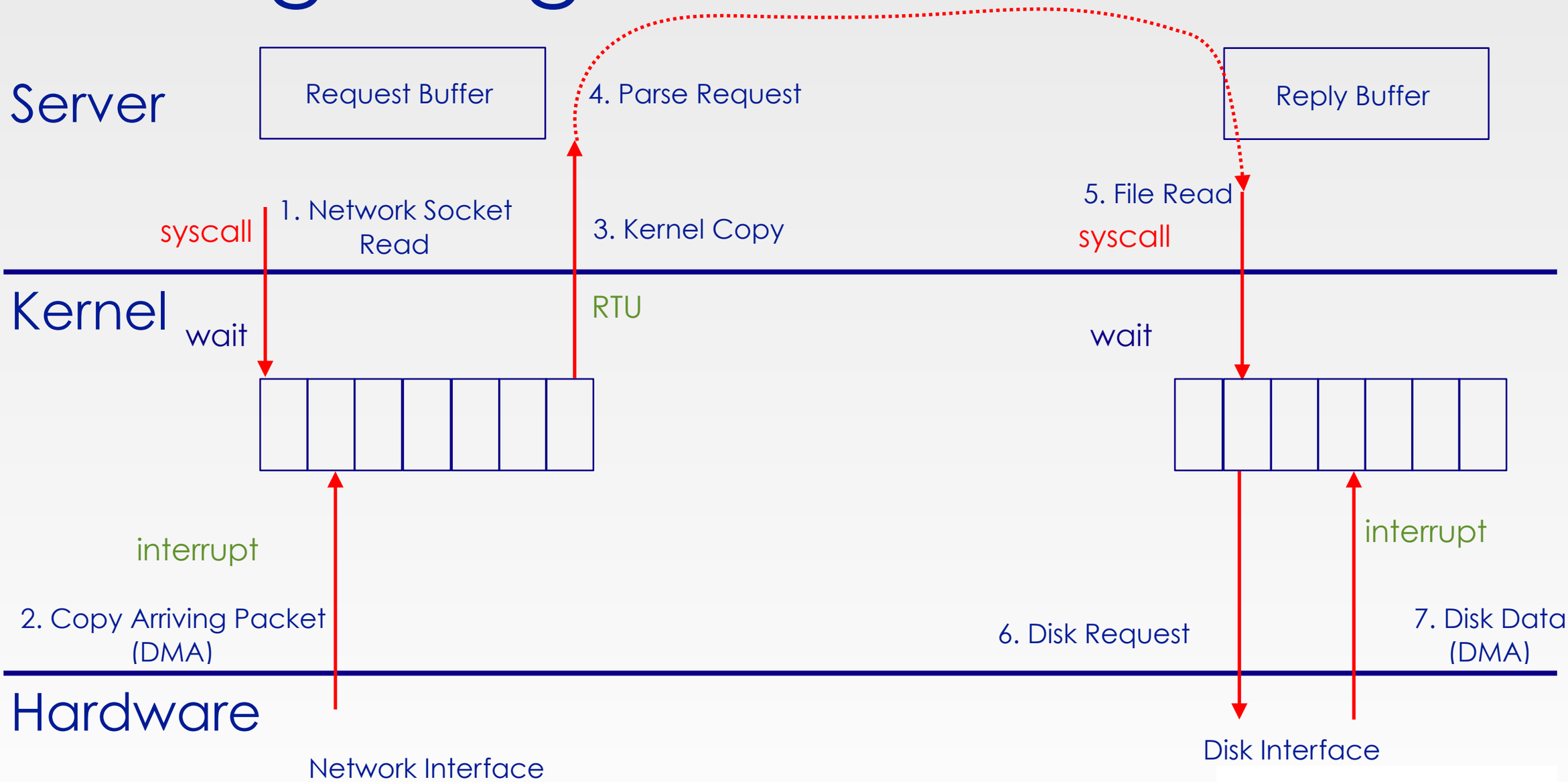
Putting it Together: Web Server



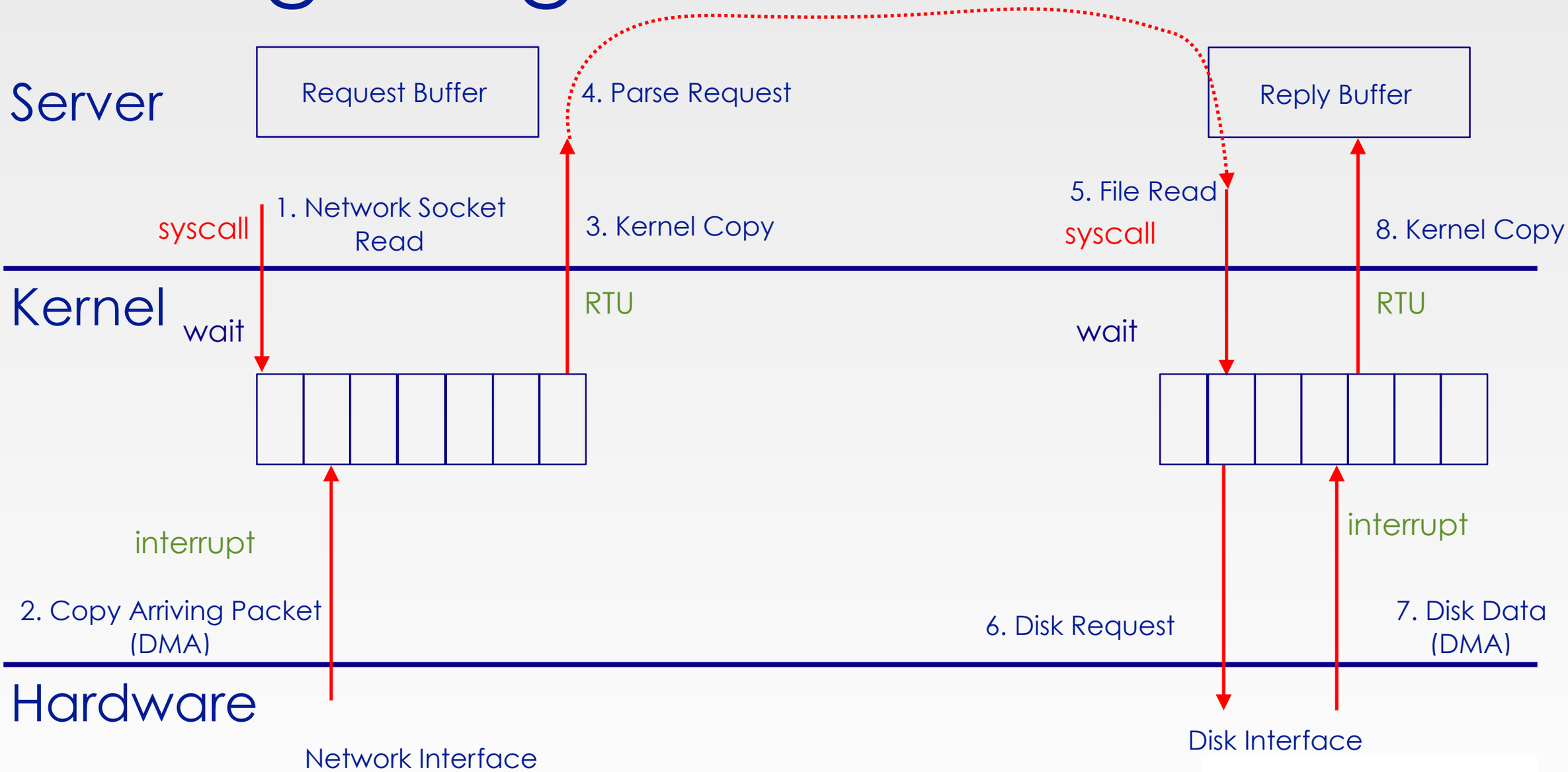
Putting it Together: Web Server



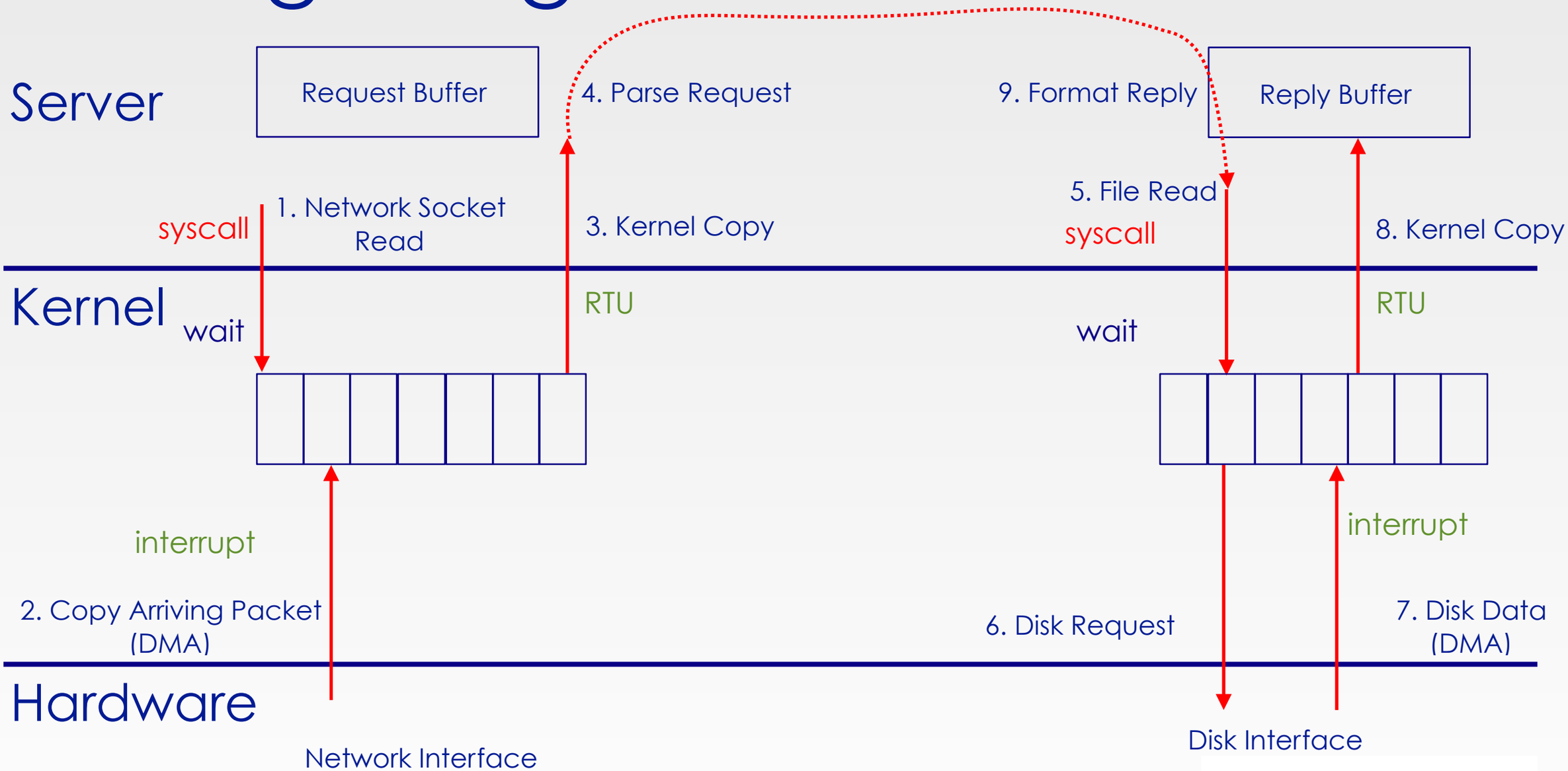
Putting it Together: Web Server



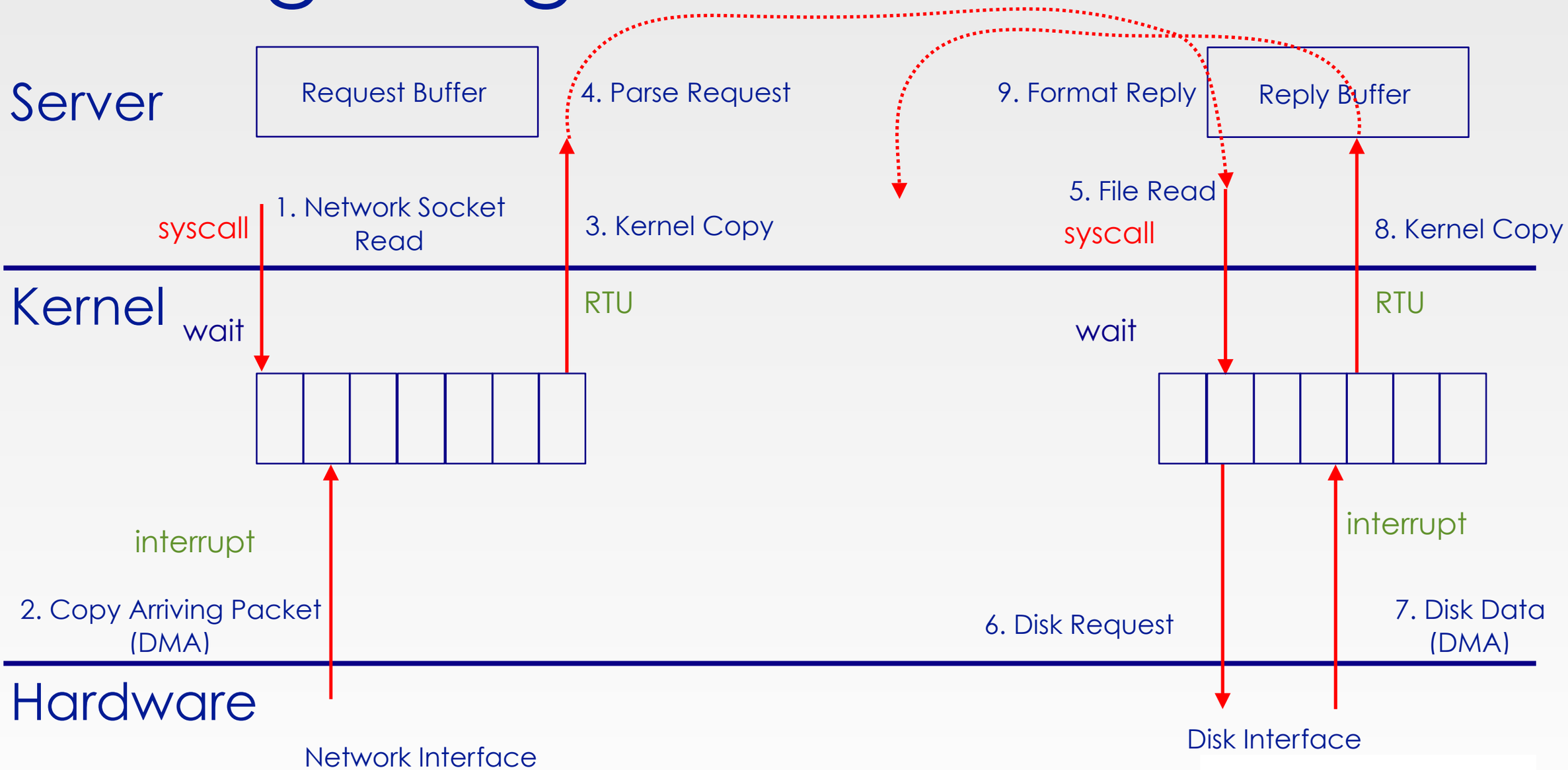
Putting it Together: Web Server



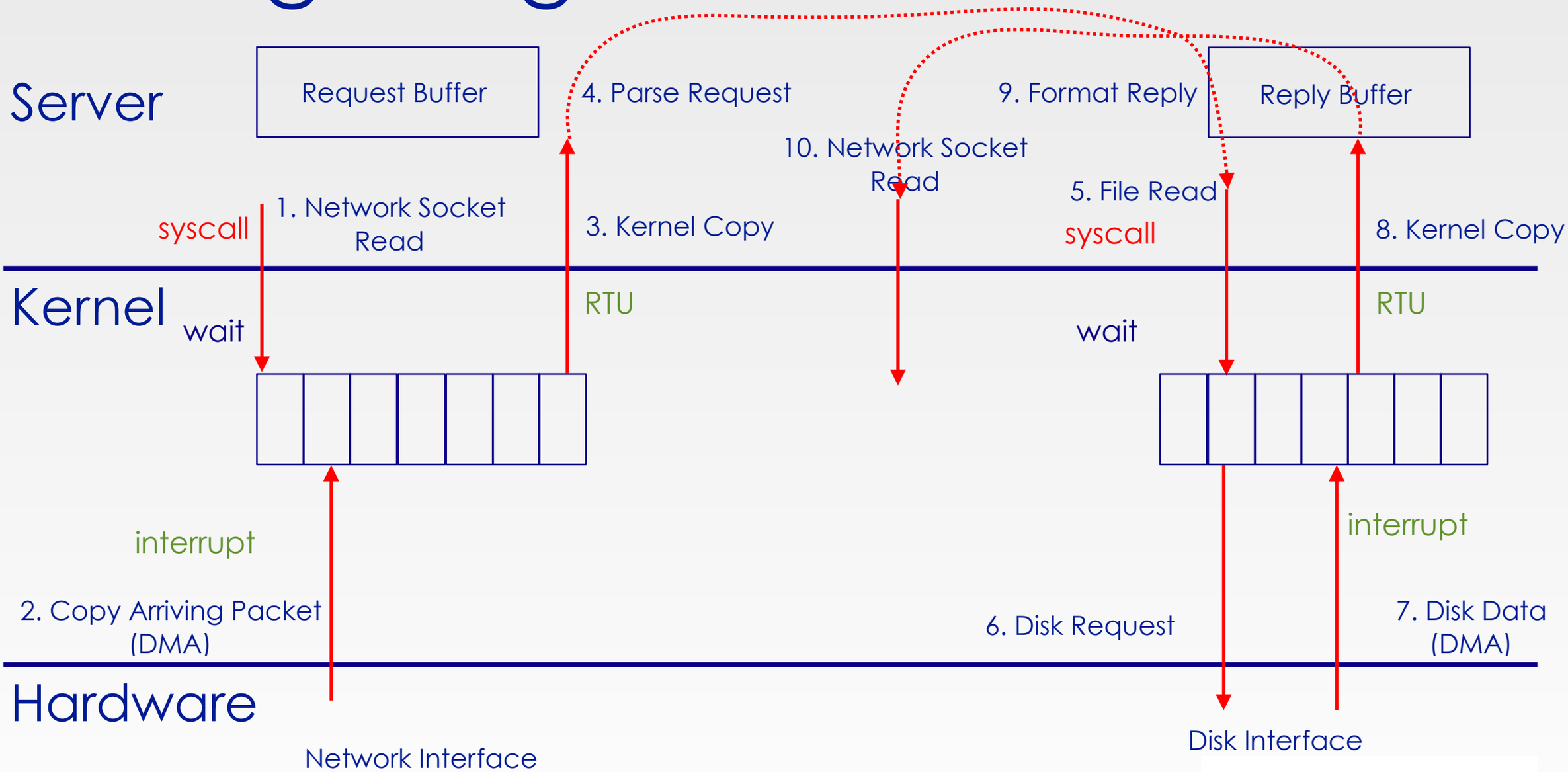
Putting it Together: Web Server



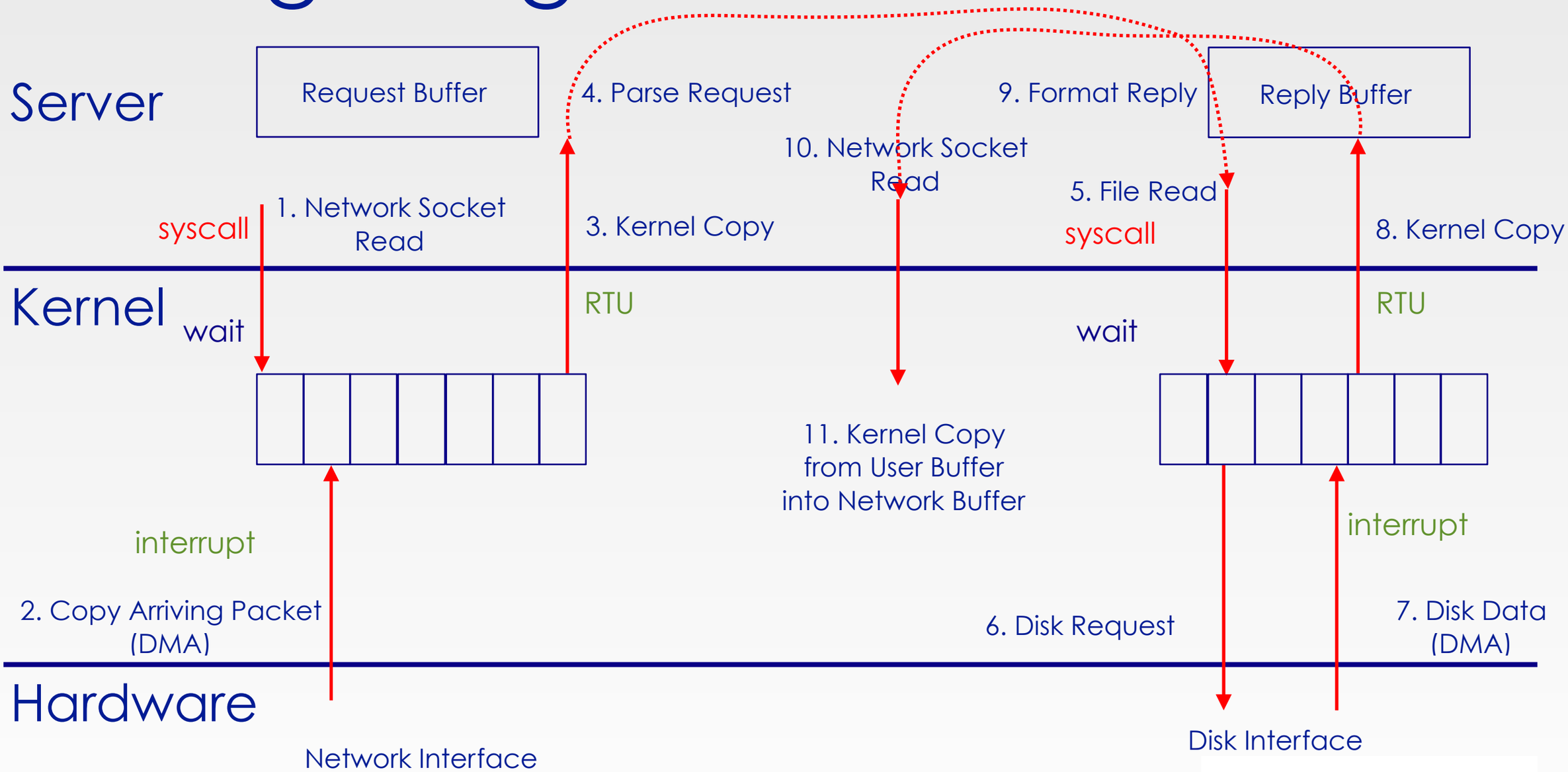
Putting it Together: Web Server



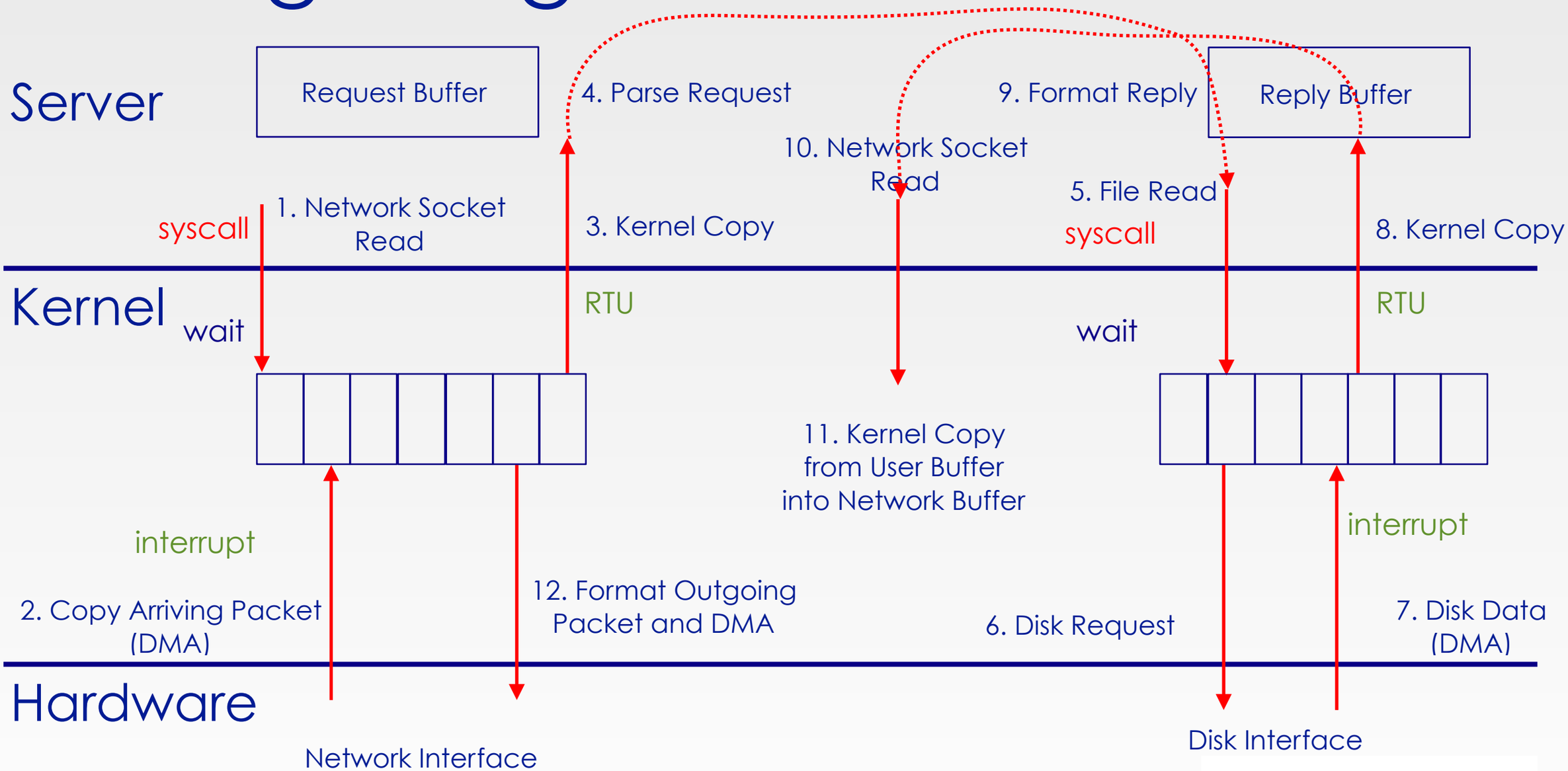
Putting it Together: Web Server



Putting it Together: Web Server



Putting it Together: Web Server



Conclusion

- Four fundamental OS concepts
- Thread
- Address Space with Translation
- Process
- Dual Mode Operation/Protection