Operating Systems

Dr. Shu Yin

Part II: Process Management

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

Goals

- Intros to processes (cont.)
- Fork

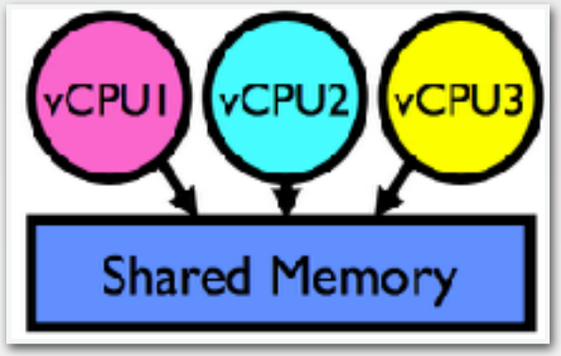
Recall1: Fundamental Concepts?

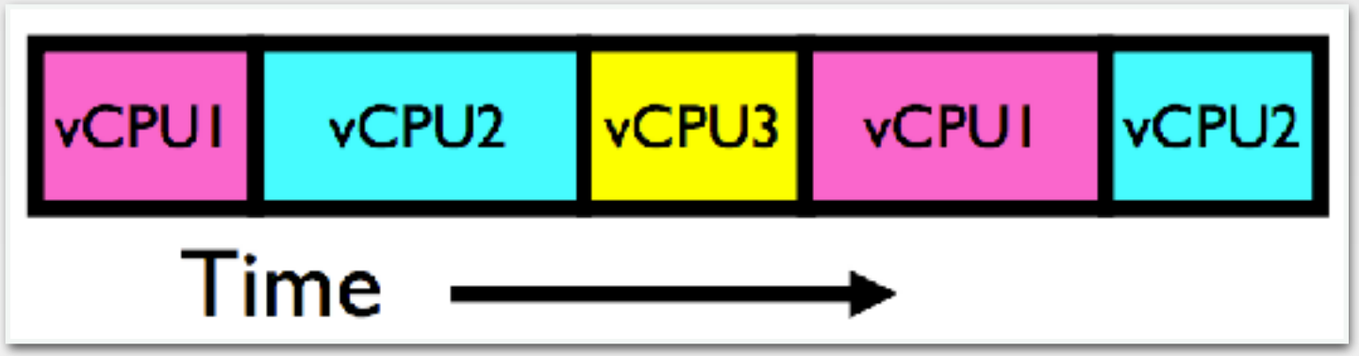
- Thread
- Address Space with Translation
- Process
- Dual Mode Operation/Protection

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

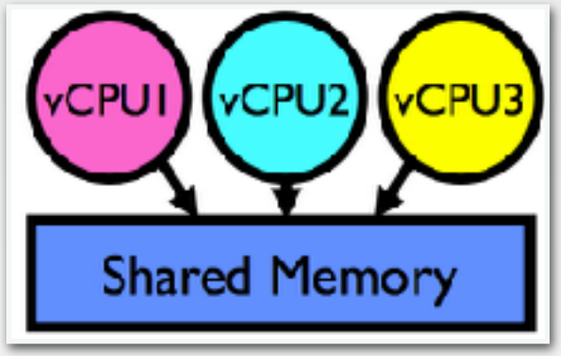
Recall2: Illusion of Multiple Processors

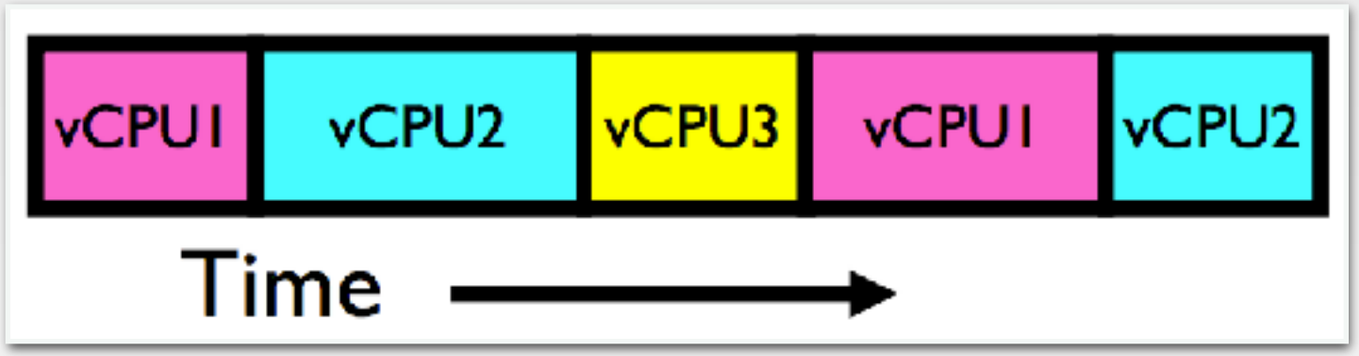




- Assume a single processor:
  - How do we provide the illusion of multiple processors? (Pentium 4)

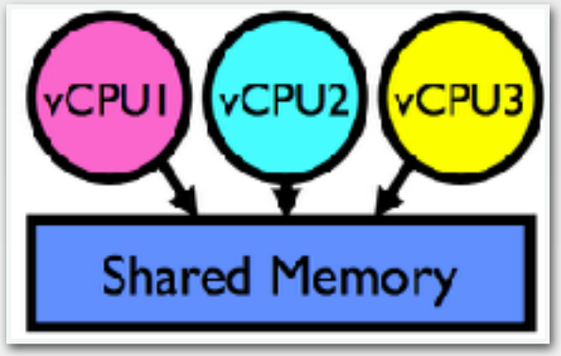
Recall2: Illusion of Multiple Processors

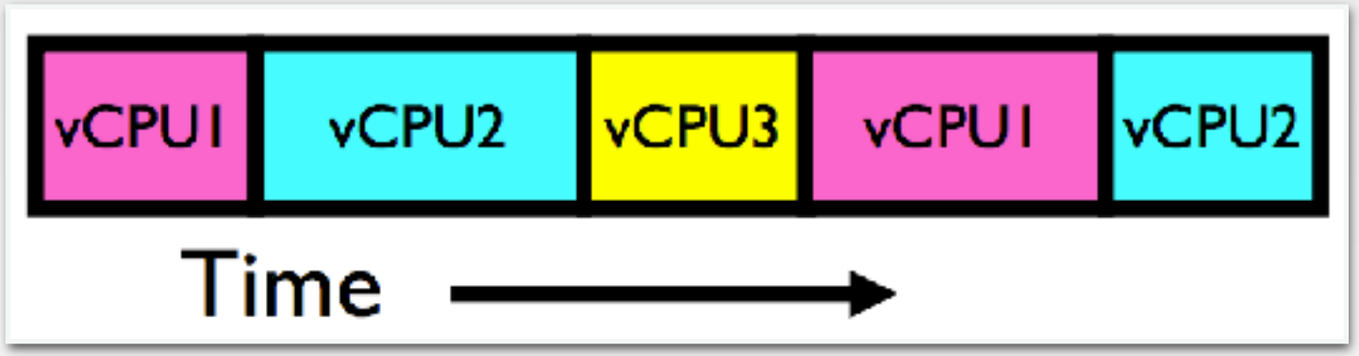




- Assume a single processor:
  - How do we provide the illusion of multiple processors? (Pentium 4)
  - Multiplex in time

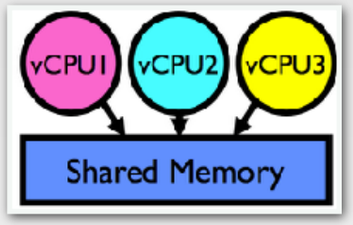
Recall2: Illusion of Multiple Processors

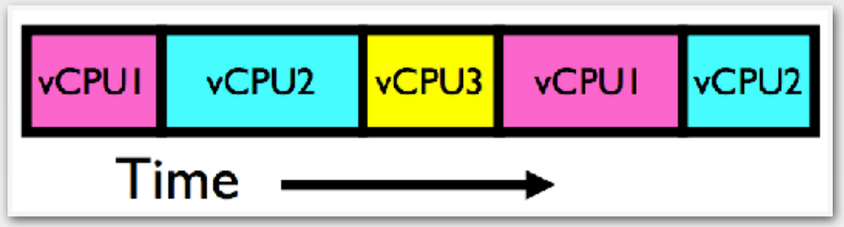




- Assume a single processor:
  - How do we provide the illusion of multiple processors? (Pentium 4)
  - Multiplex in time
  - Multiple "virtual CPUs"

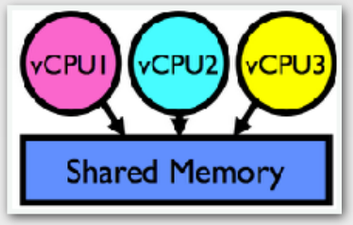
Illusion of Multiple Processors

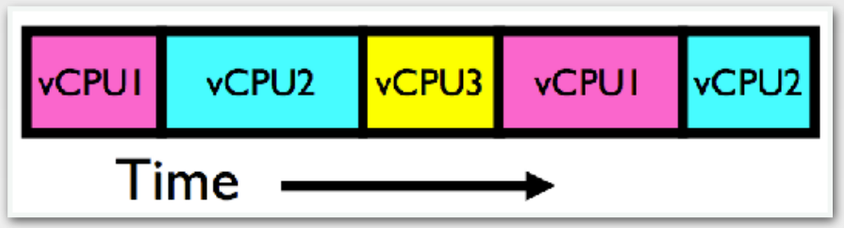




- Each virtual CPU
  - Needs a structure to hold
    - PC, SP
    - Registers

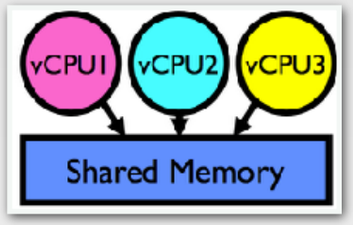
Illusion of Multiple Processors

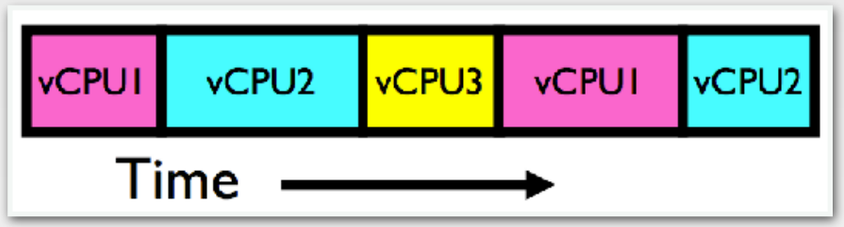




- Each virtual CPU
  - Needs a structure to hold
    - PC, SP
    - Registers
  - How to switch from one virtual CPU to the next?

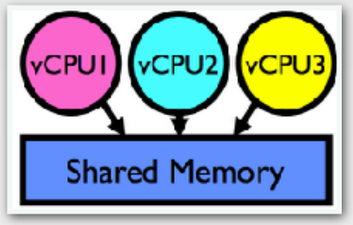
Illusion of Multiple Processors

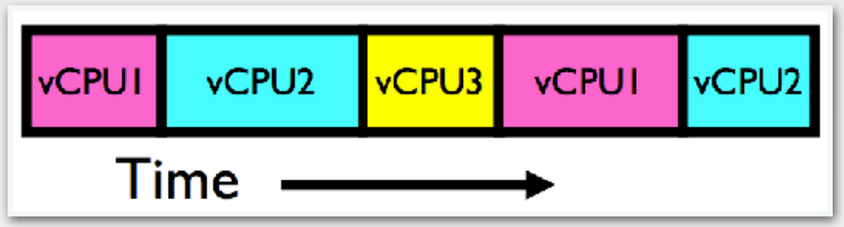




- Each virtual CPU
  - Needs a structure to hold
    - PC, SP
    - Registers
  - How to switch from one virtual CPU to the next?
    - Save PC, SP, Registers in the current state block

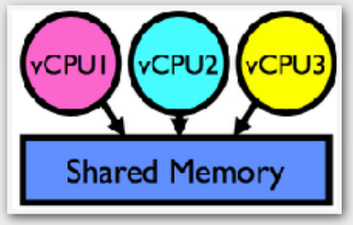
Illusion of Multiple Processors

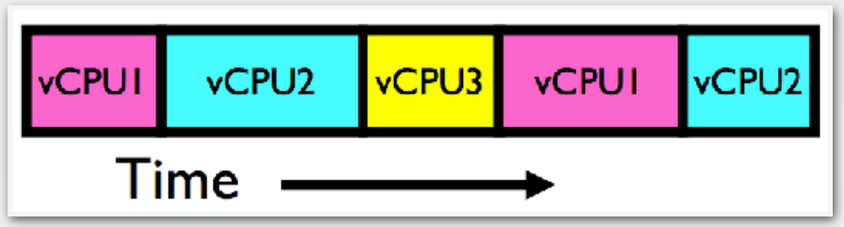




- Each virtual CPU
  - Needs a structure to hold
    - PC, SP
    - Registers
  - How to switch from one virtual CPU to the next?
    - Save PC, SP, Registers in the current state block
    - Load PC, SP, Registers from the new state block

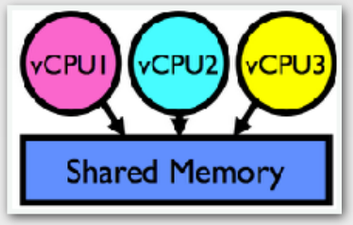
Illusion of Multiple Processors

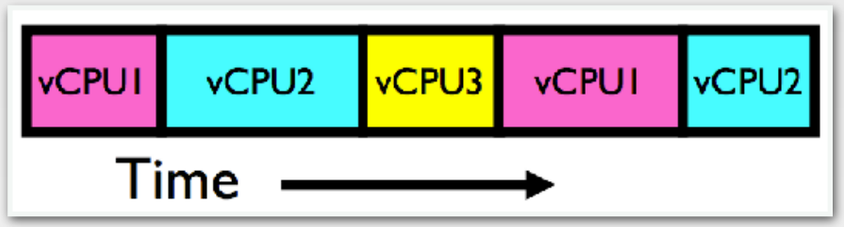




- Each virtual CPU
  - Needs a structure to hold
    - PC, SP
    - Registers
  - How to switch from one virtual CPU to the next?
    - Save PC, SP, Registers in the current state block
    - Load PC, SP, Registers from the new state block
  - What triggers switch?

Illusion of Multiple Processors





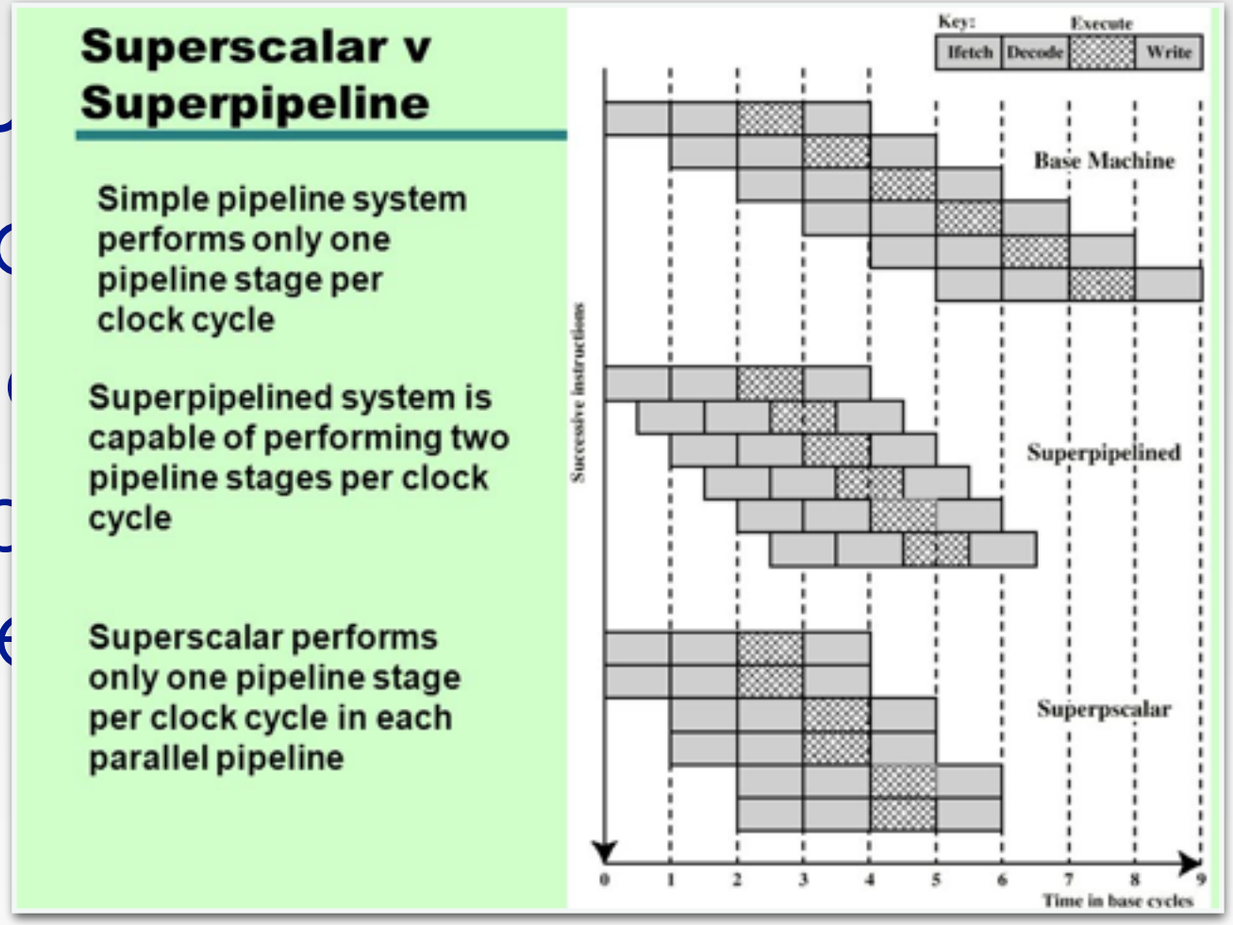
- Each virtual CPU
  - Needs a structure to hold
    - PC, SP
    - Registers
  - How to switch from one virtual CPU to the next?
    - Save PC, SP, Registers in the current state block
    - Load PC, SP, Registers from the new state block
  - What triggers switch?
    - Timer, Voluntary yield, I/O,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

Simultaneous MultiThreading/Hyperthreading

- Hardware technique
  - Superscalar processors
  - Execute multiple independent instructions
  - Hyper threading duplicates register state to make a second “thread”, allowing more instructions to run

Simultaneous MultiThreading/Hyperthreading

- Hardware techniqu
  - Superscalar processo
  - Execute multiple ind
  - Hyper threading dup make a second "thre instructions to run



MultiThreading (cont.)

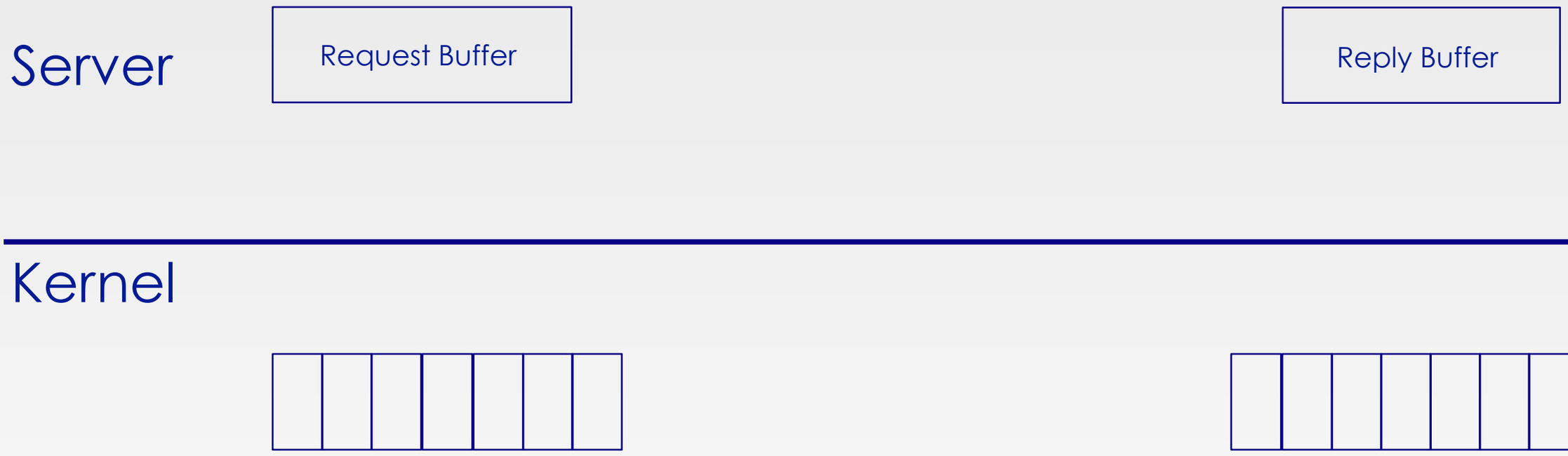
- Can schedule each thread as if were separated CPU
  - but, sub-linear speedup
- Original technique called “simultaneous multithreading”
  - SPARC, Pentium 4/Xeon (HT), Power5

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

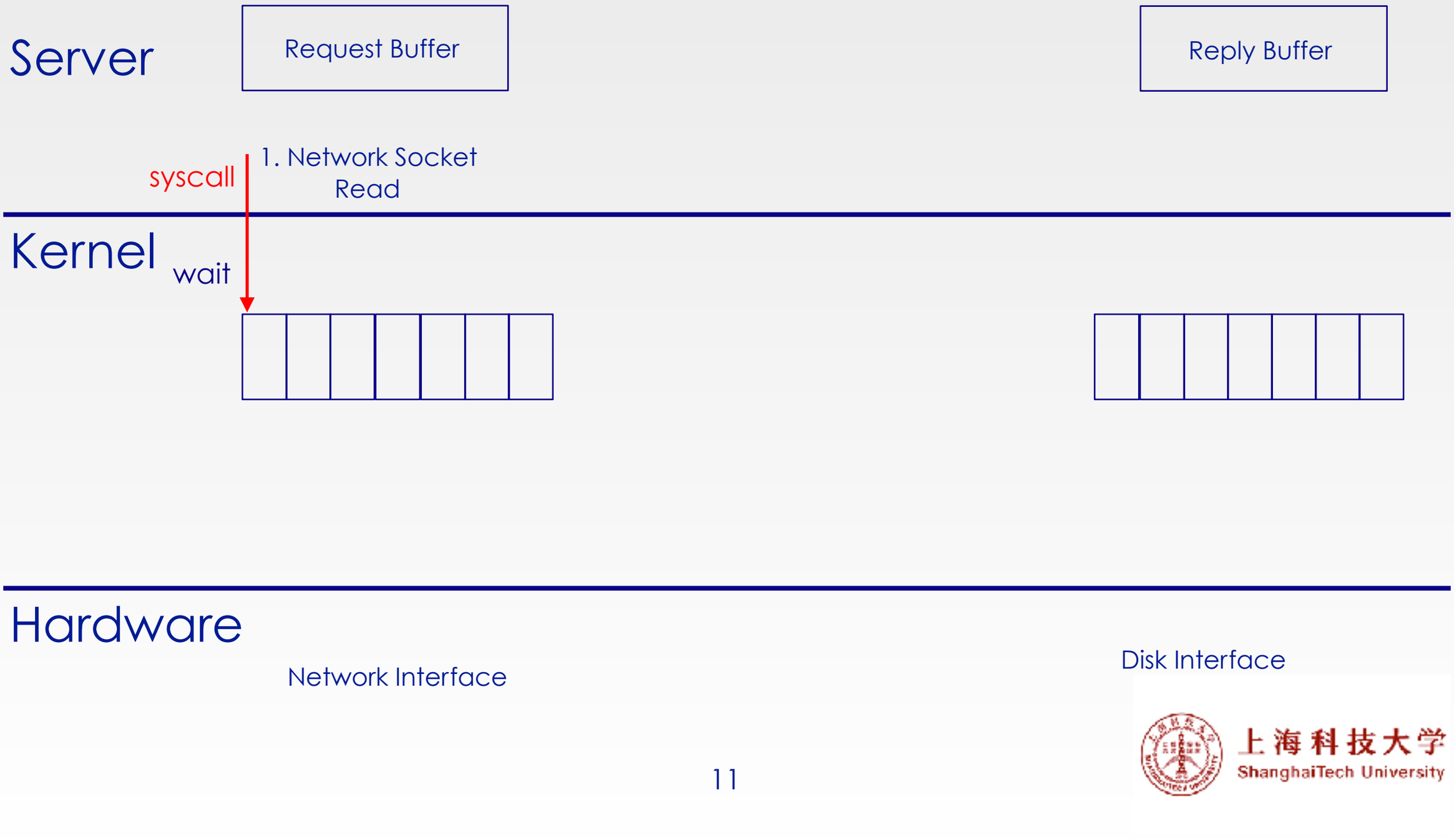
Kernel

Hardware

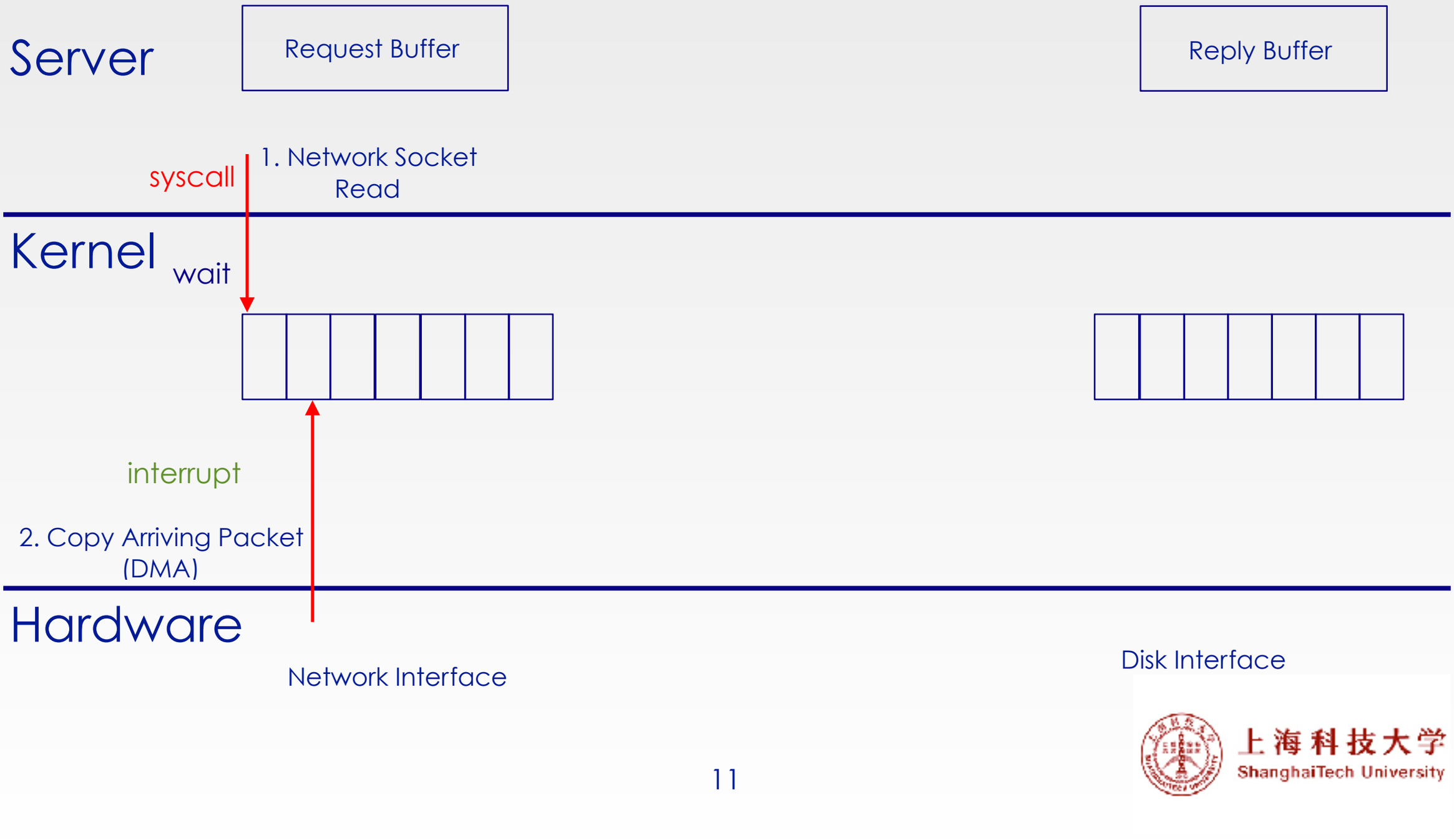
Network Interface

Disk Interface

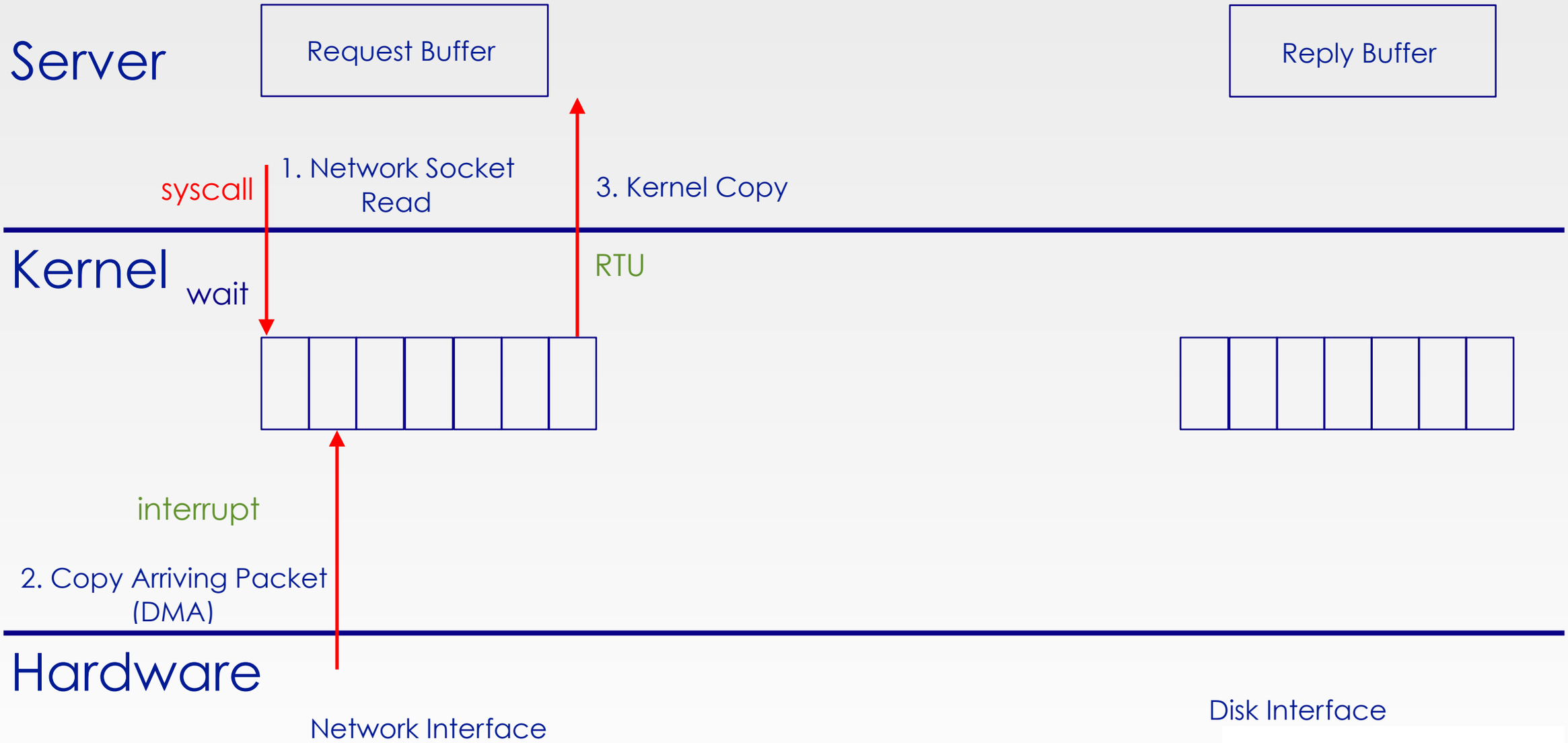
Putting it Together: Web Server



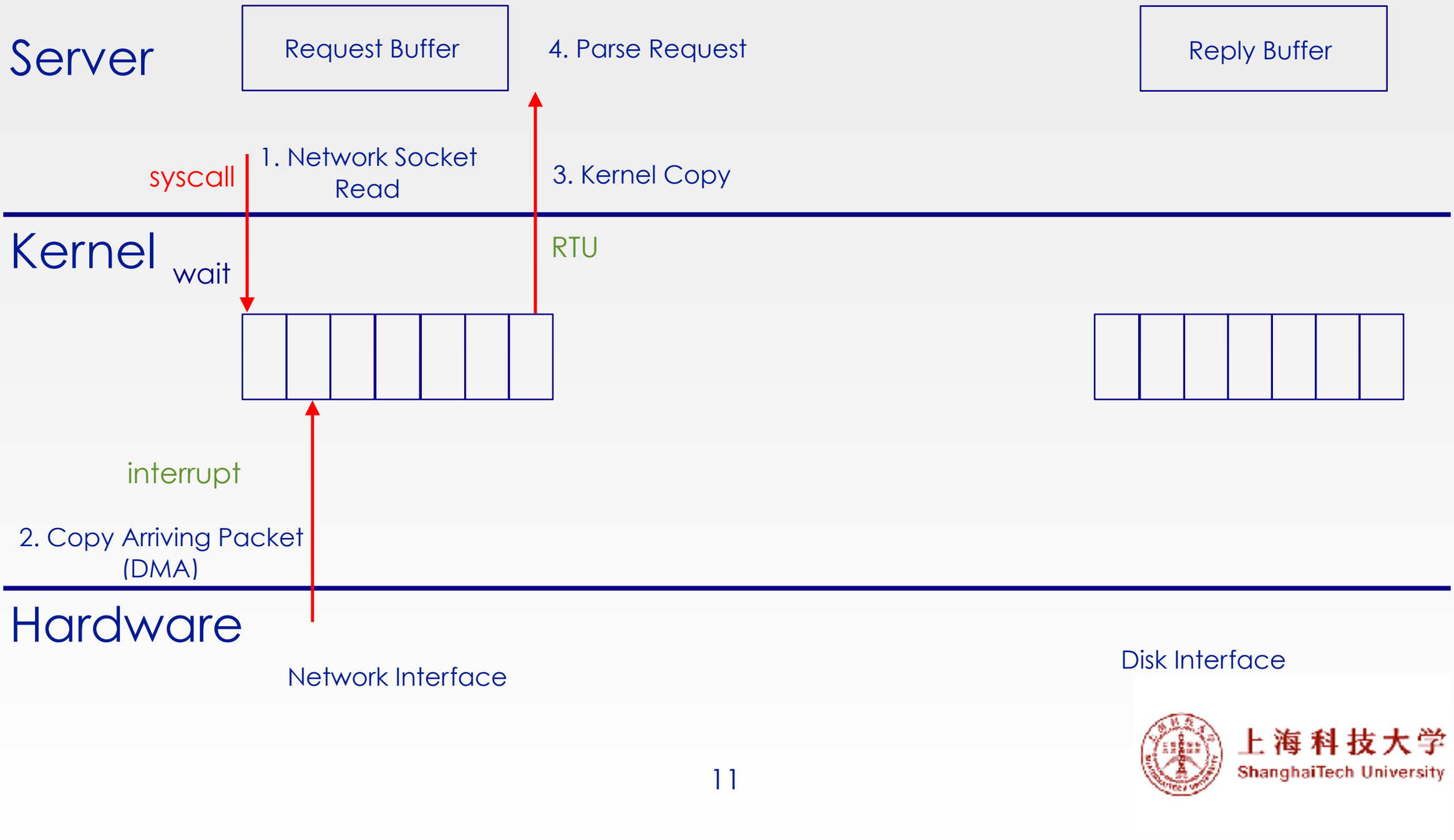
Putting it Together: Web Server



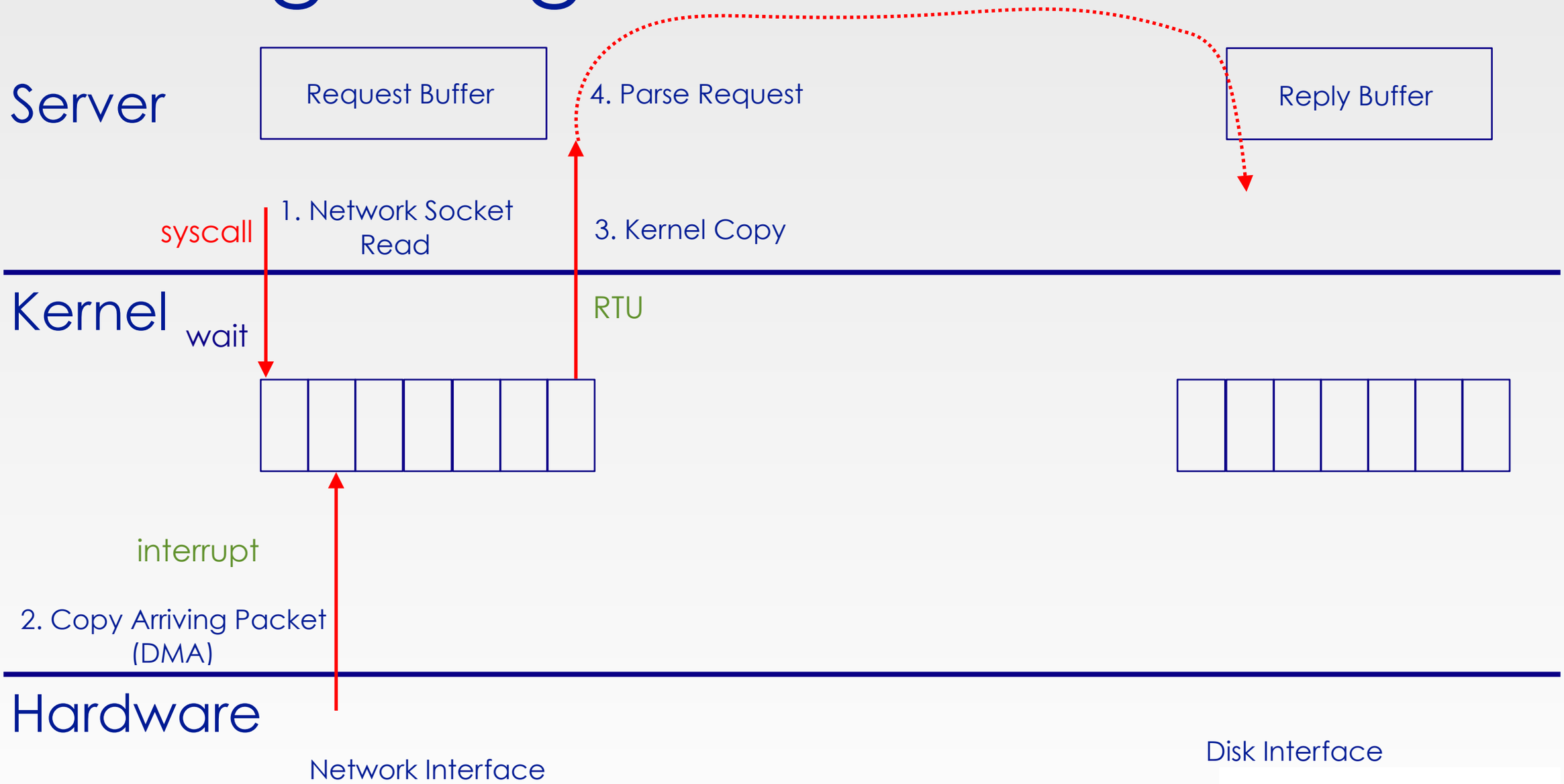
Putting it Together: Web Server



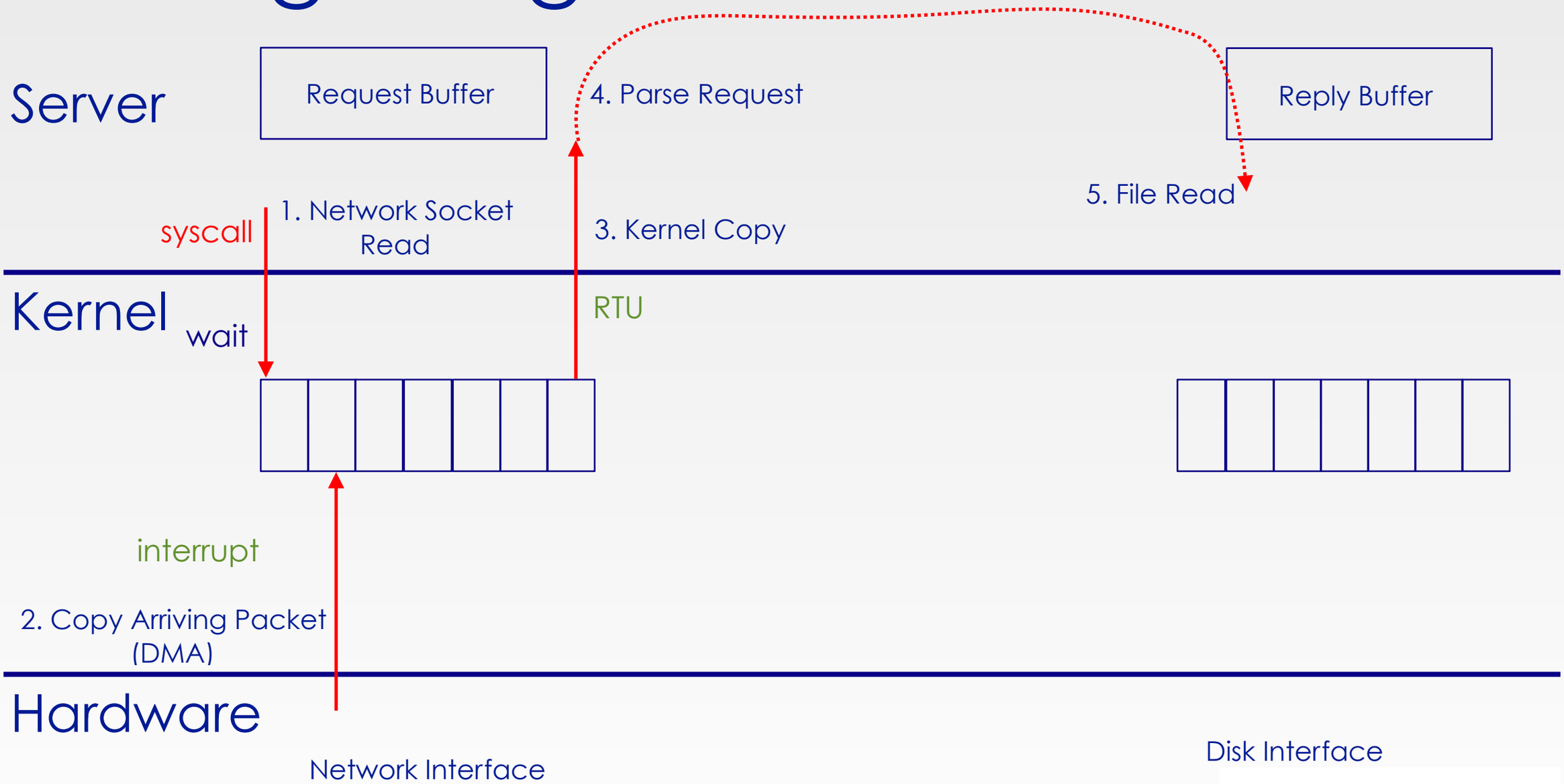
Putting it Together: Web Server



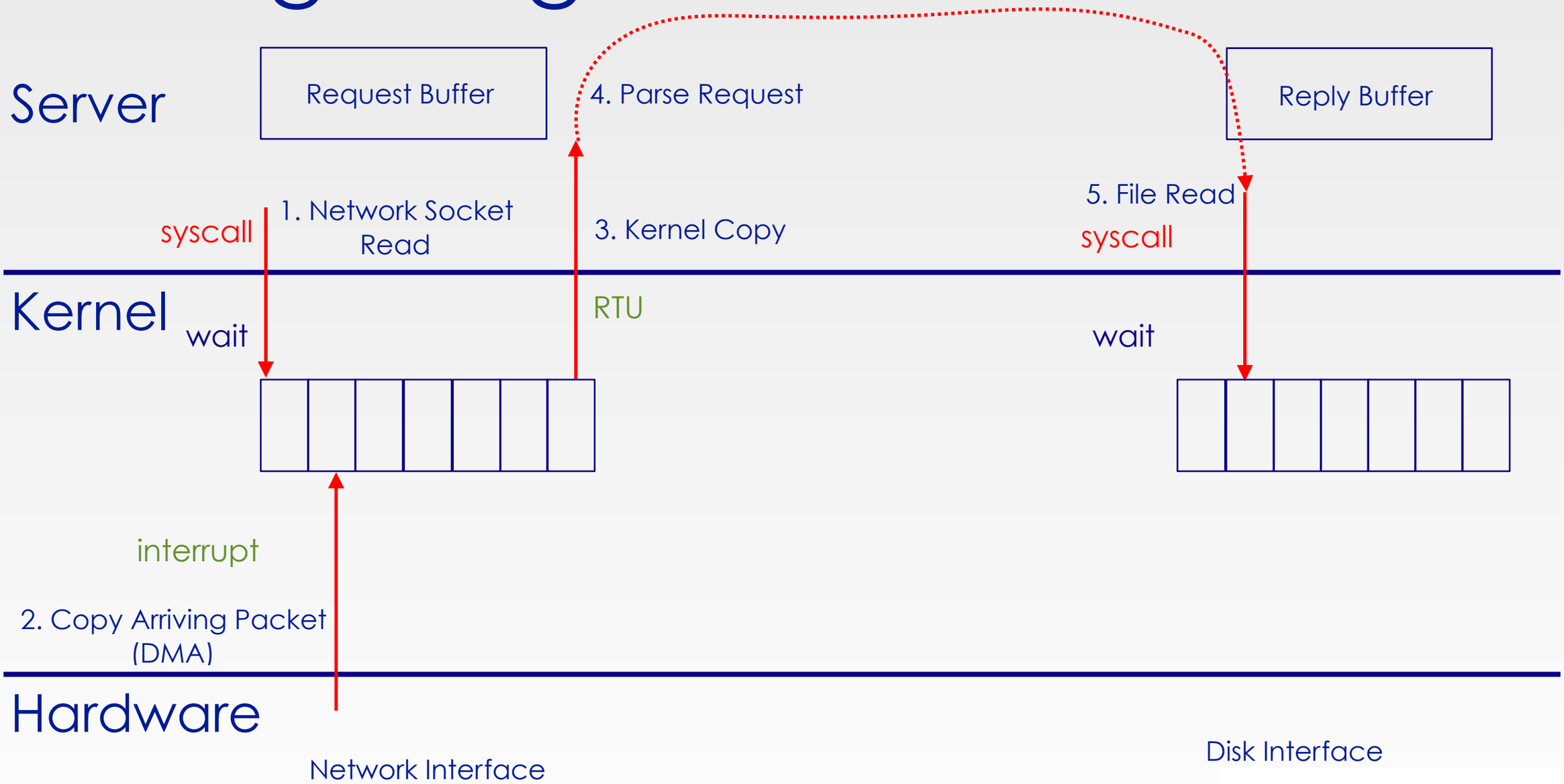
Putting it Together: Web Server



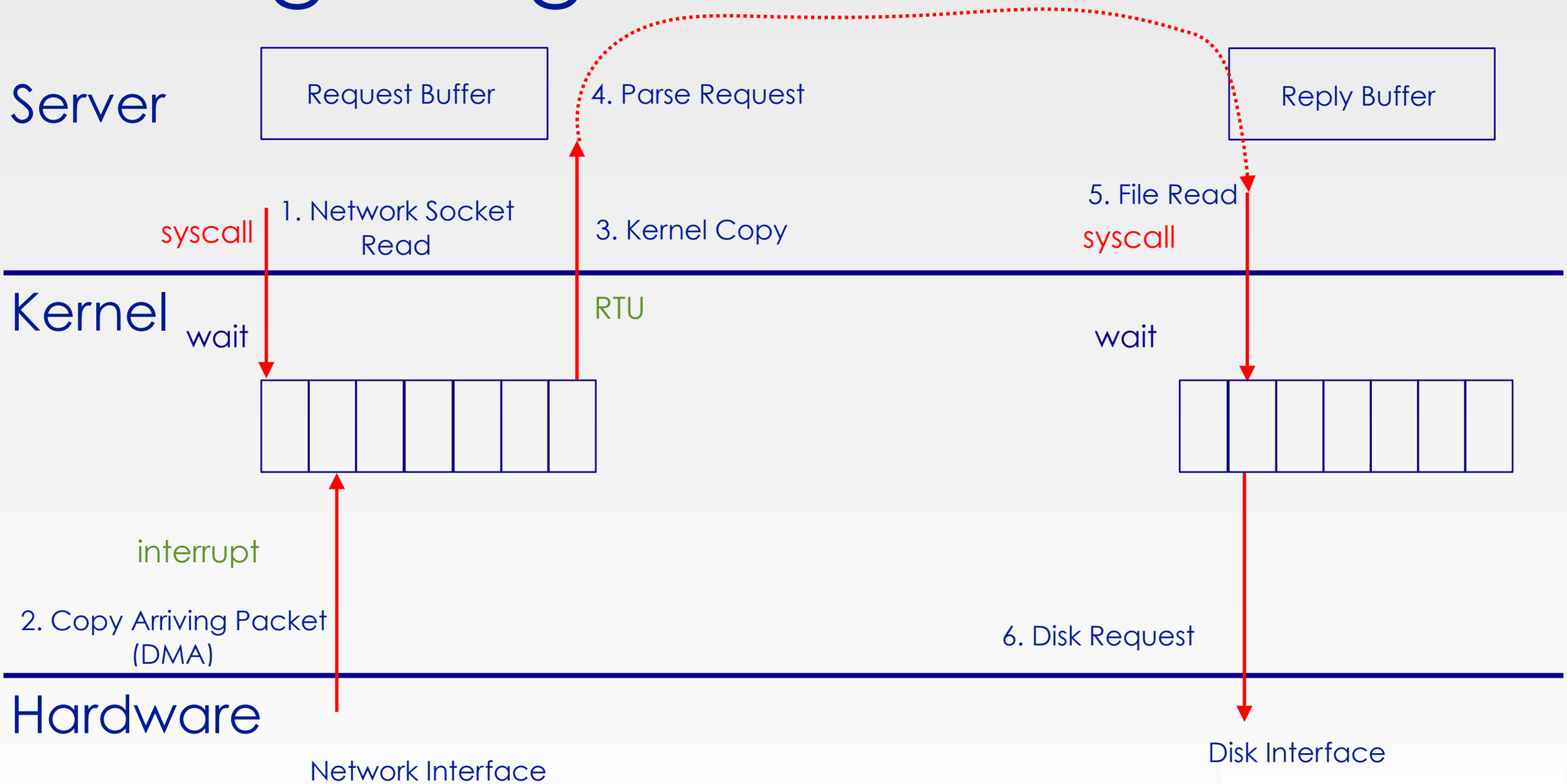
Putting it Together: Web Server



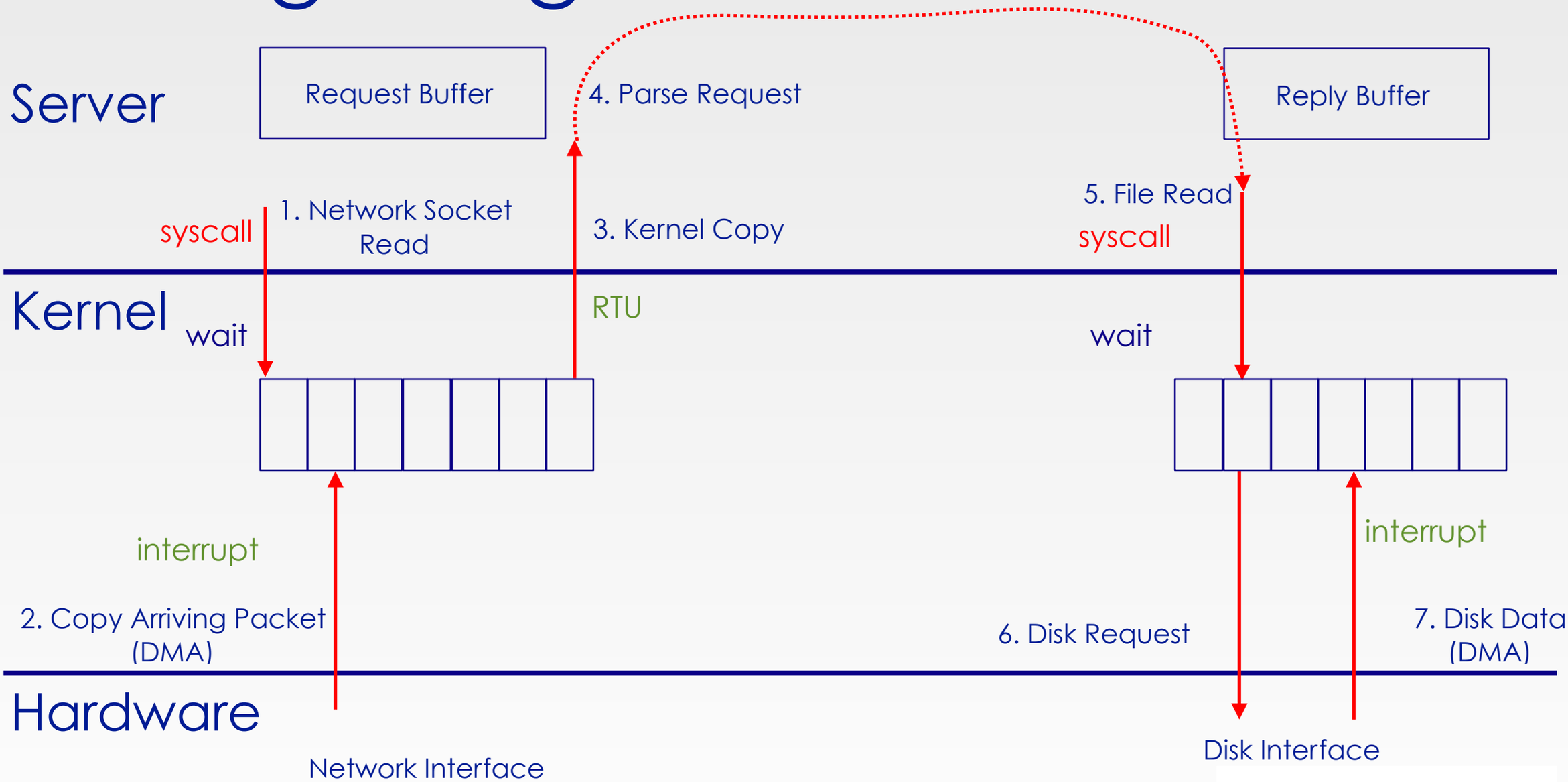
Putting it Together: Web Server



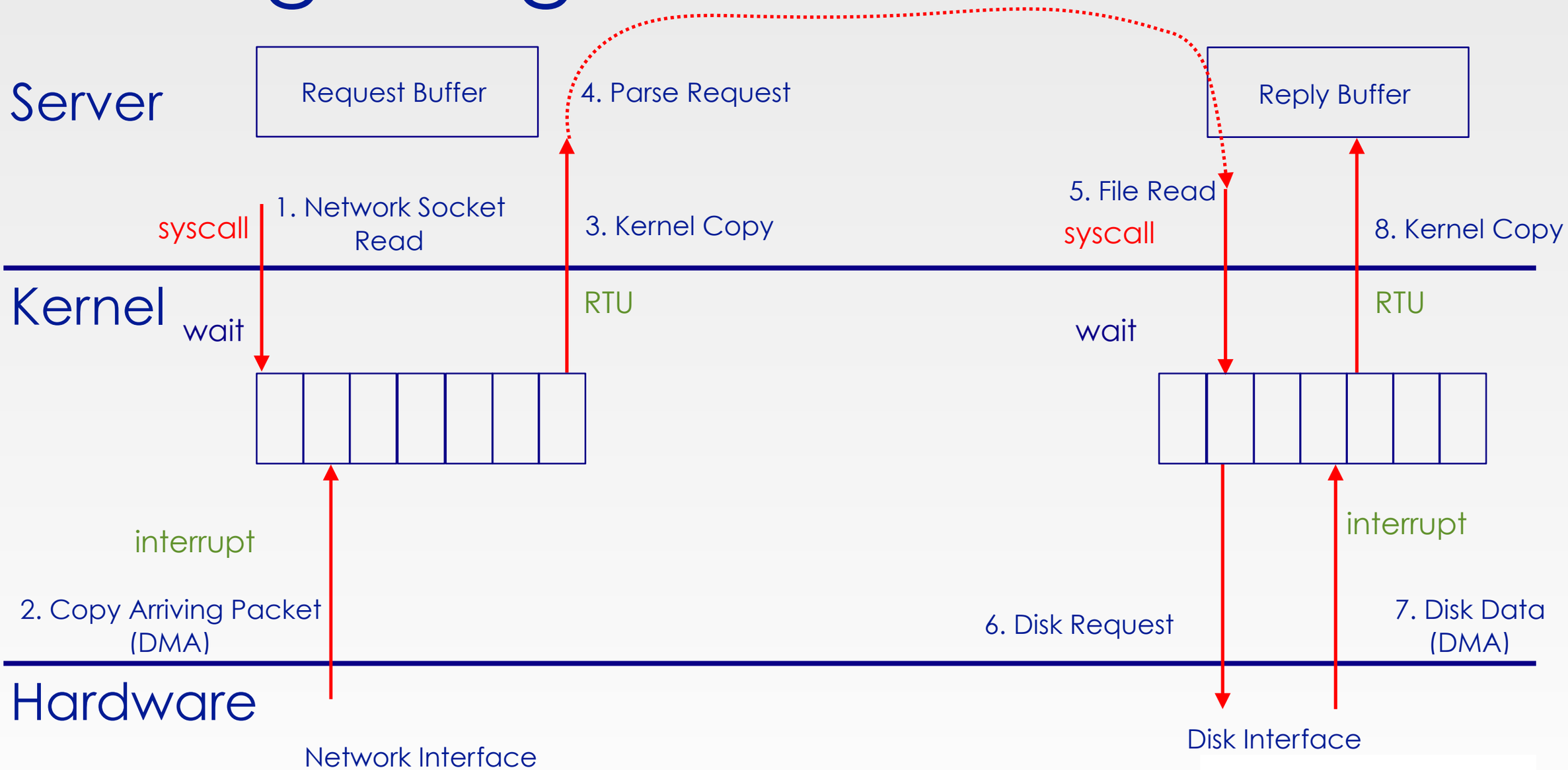
Putting it Together: Web Server



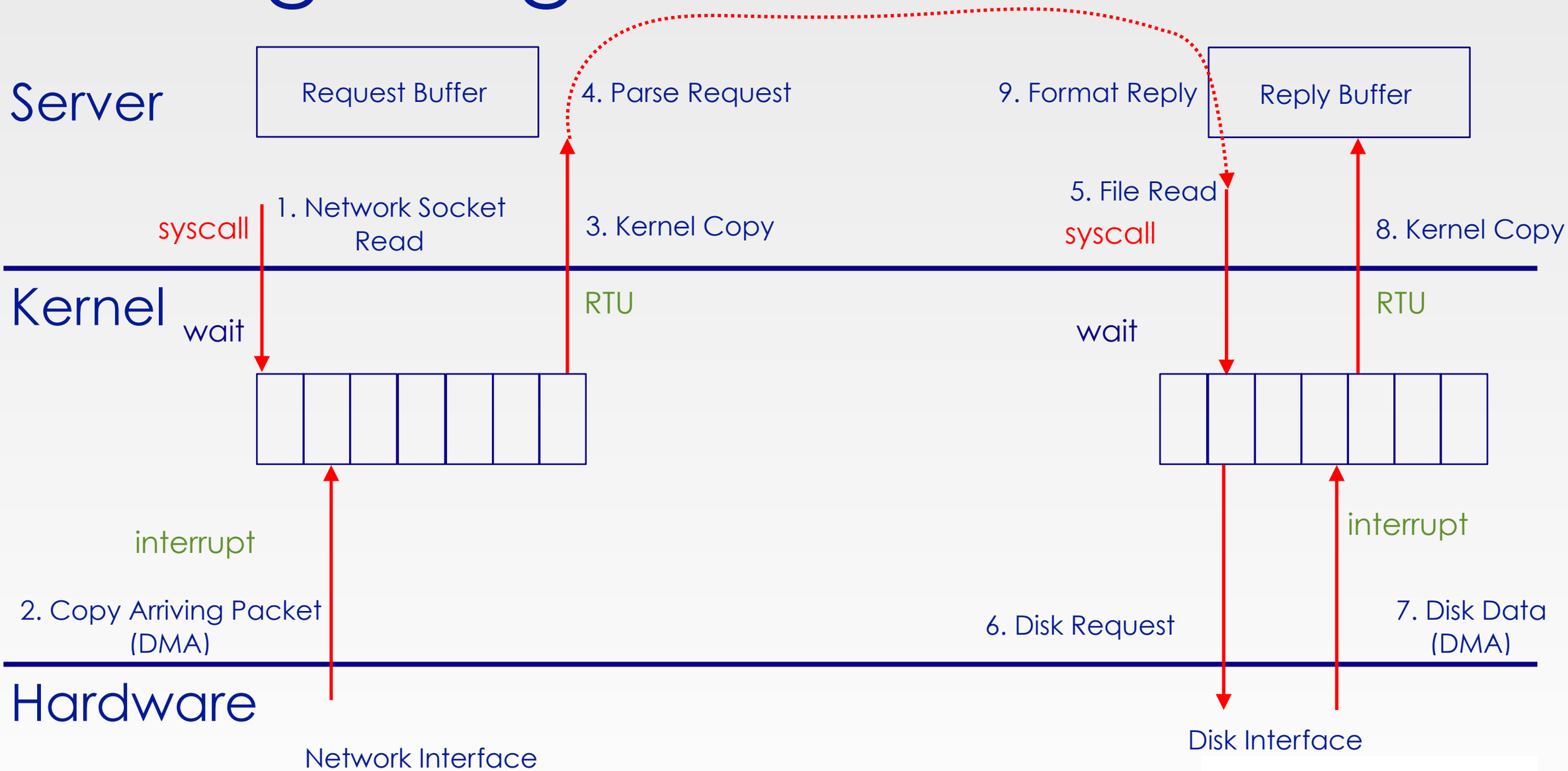
Putting it Together: Web Server



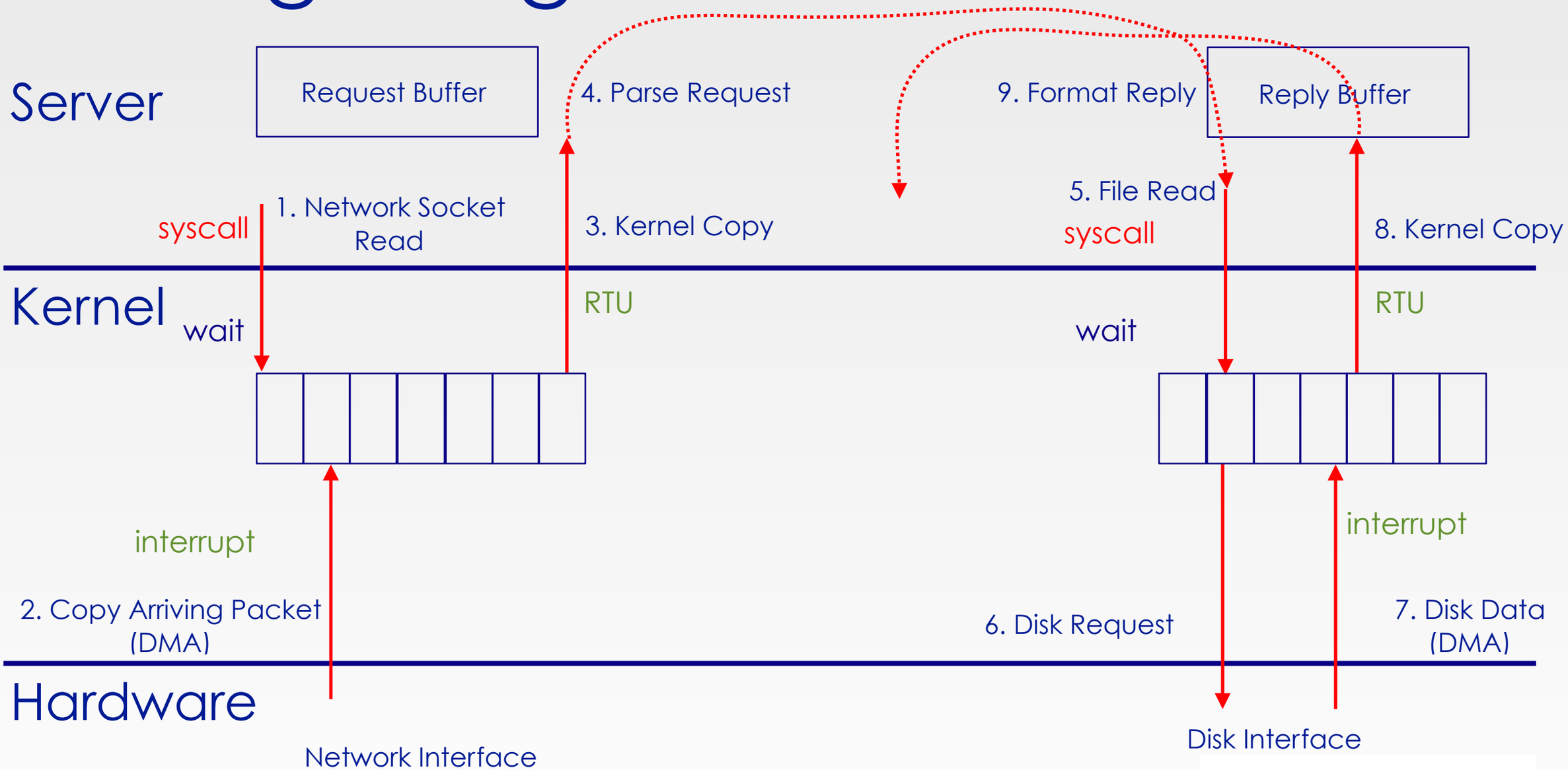
Putting it Together: Web Server



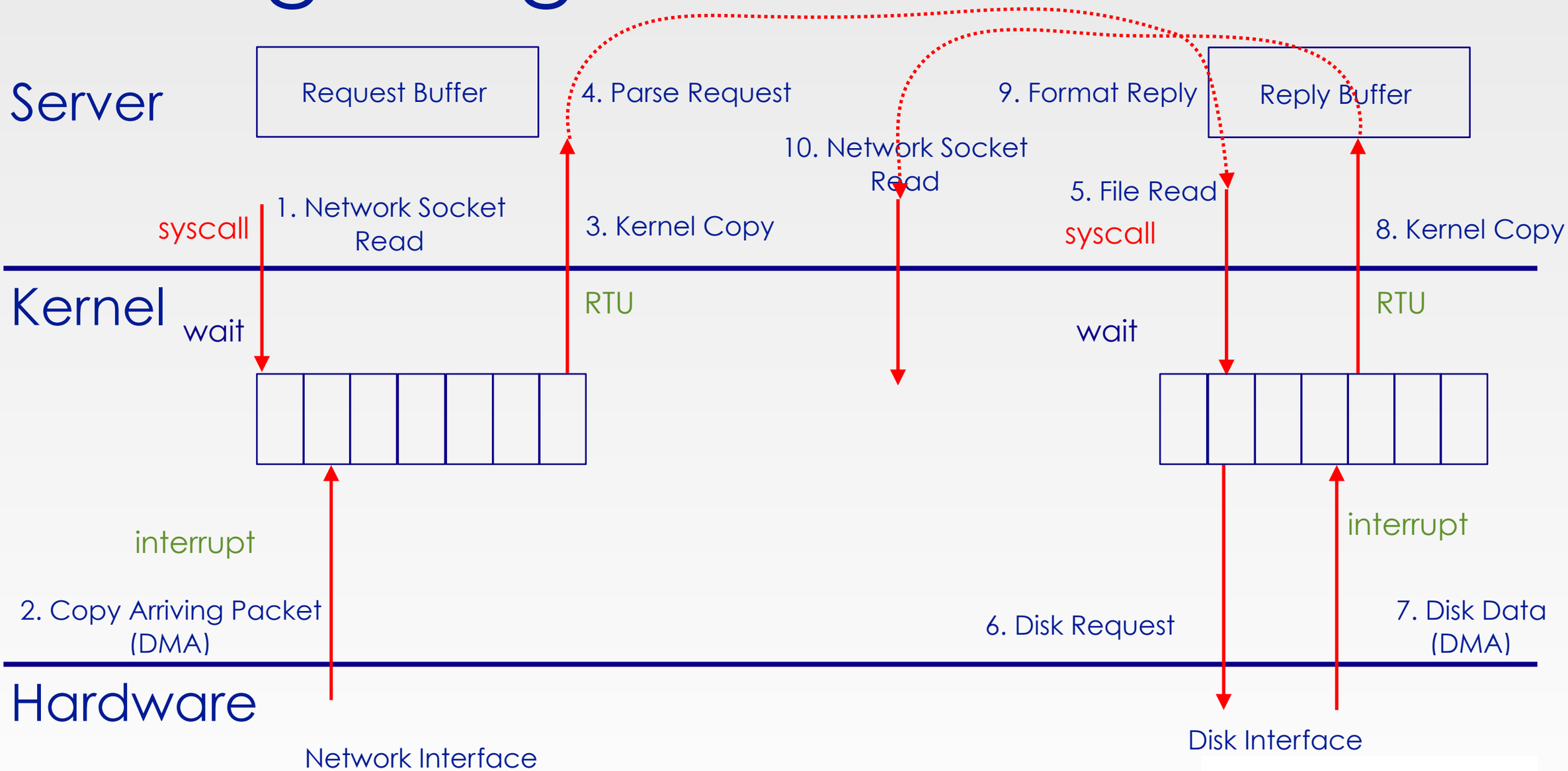
Putting it Together: Web Server



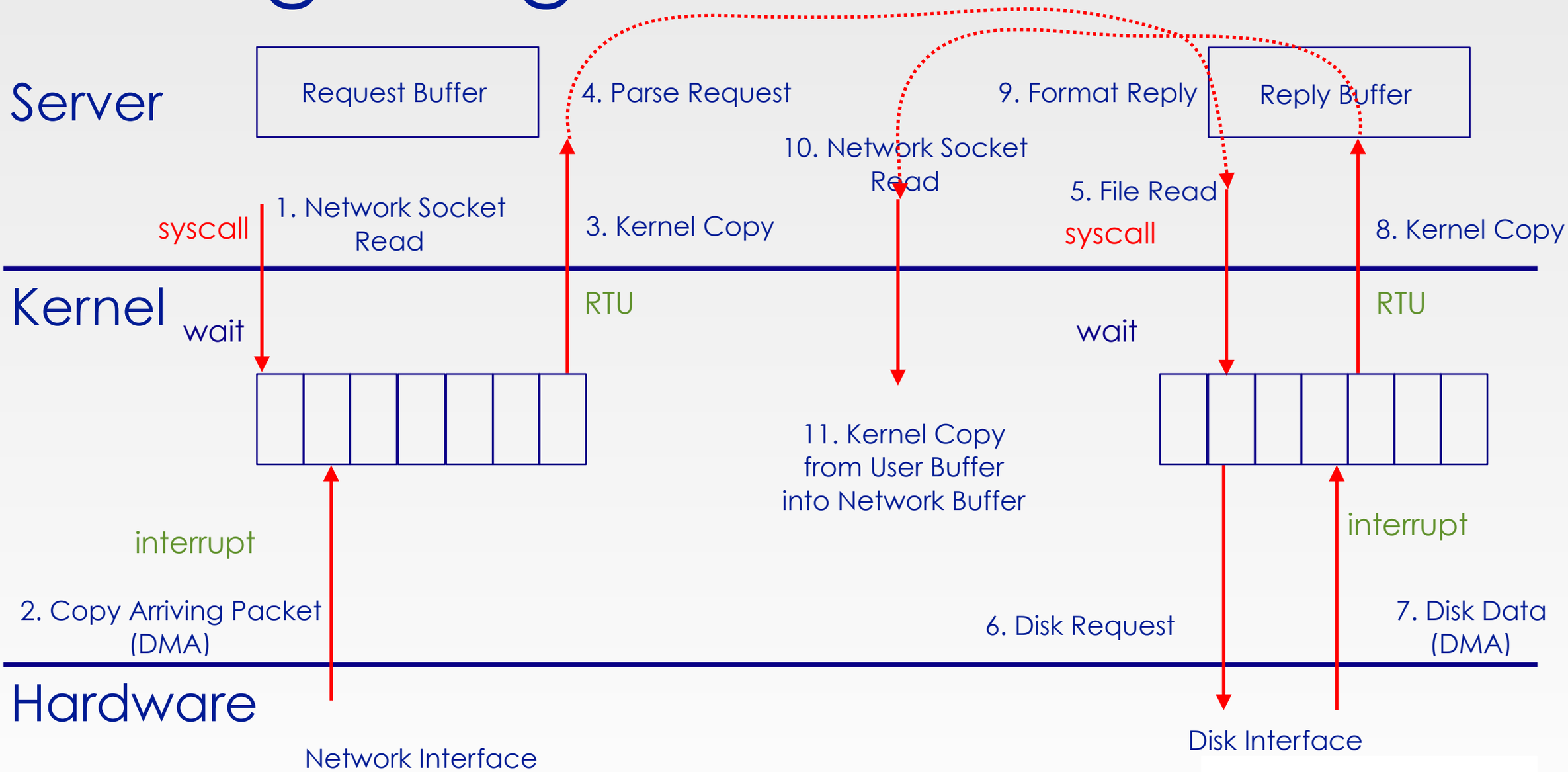
Putting it Together: Web Server



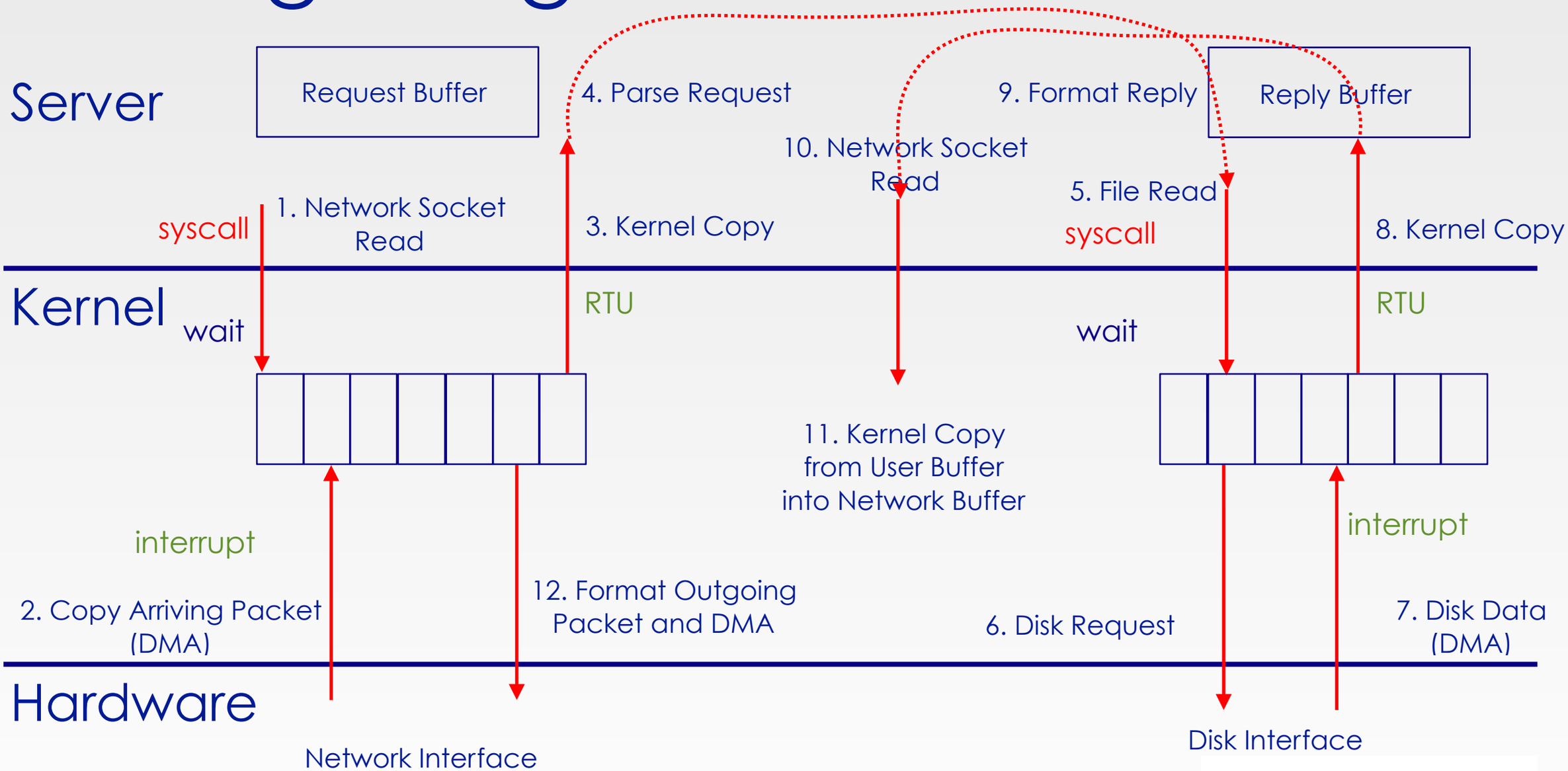
Putting it Together: Web Server



Putting it Together: Web Server



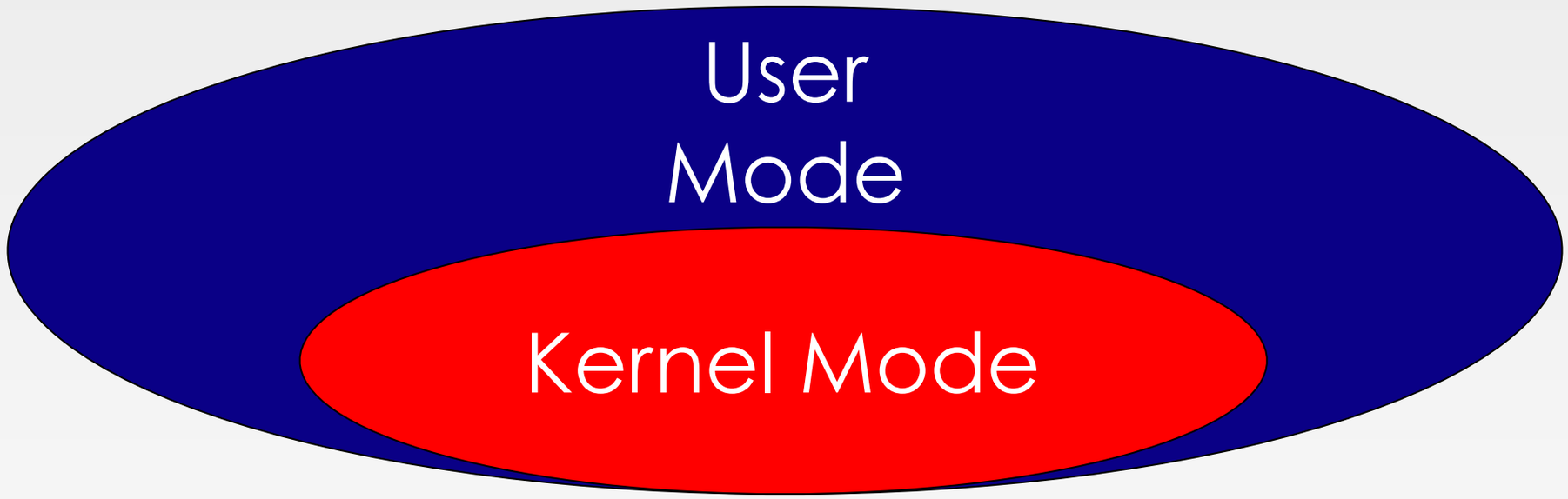
Putting it Together: Web Server



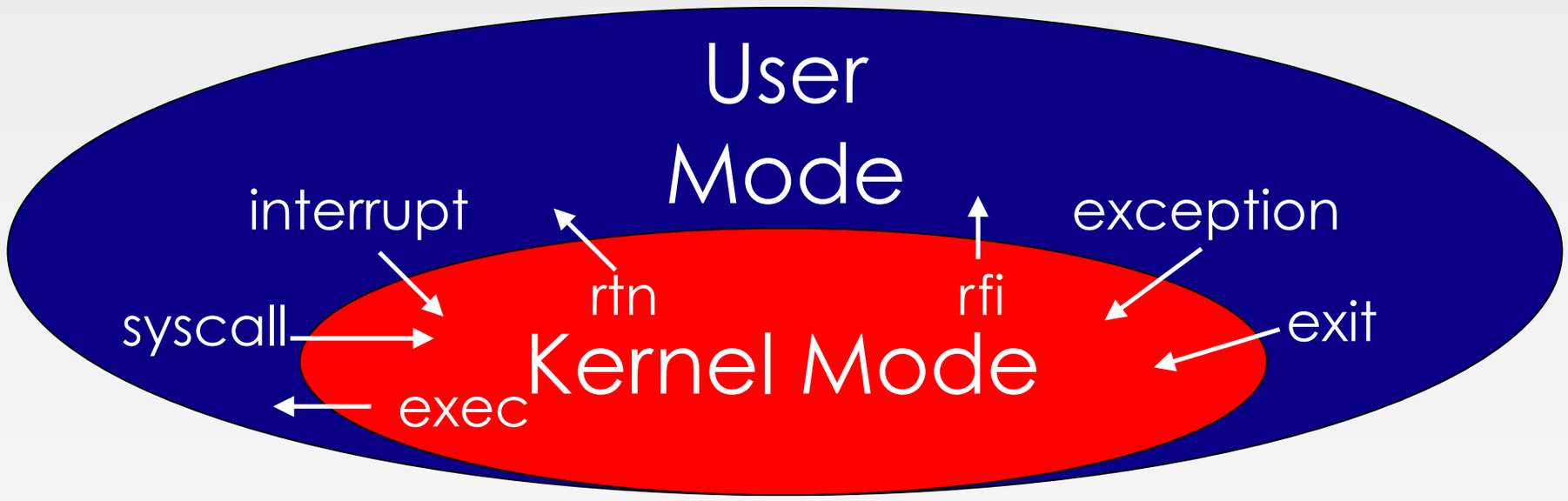
Recall3: Types of Kernel Mode Transfer

- syscall
- interrupt
- trap/ exception

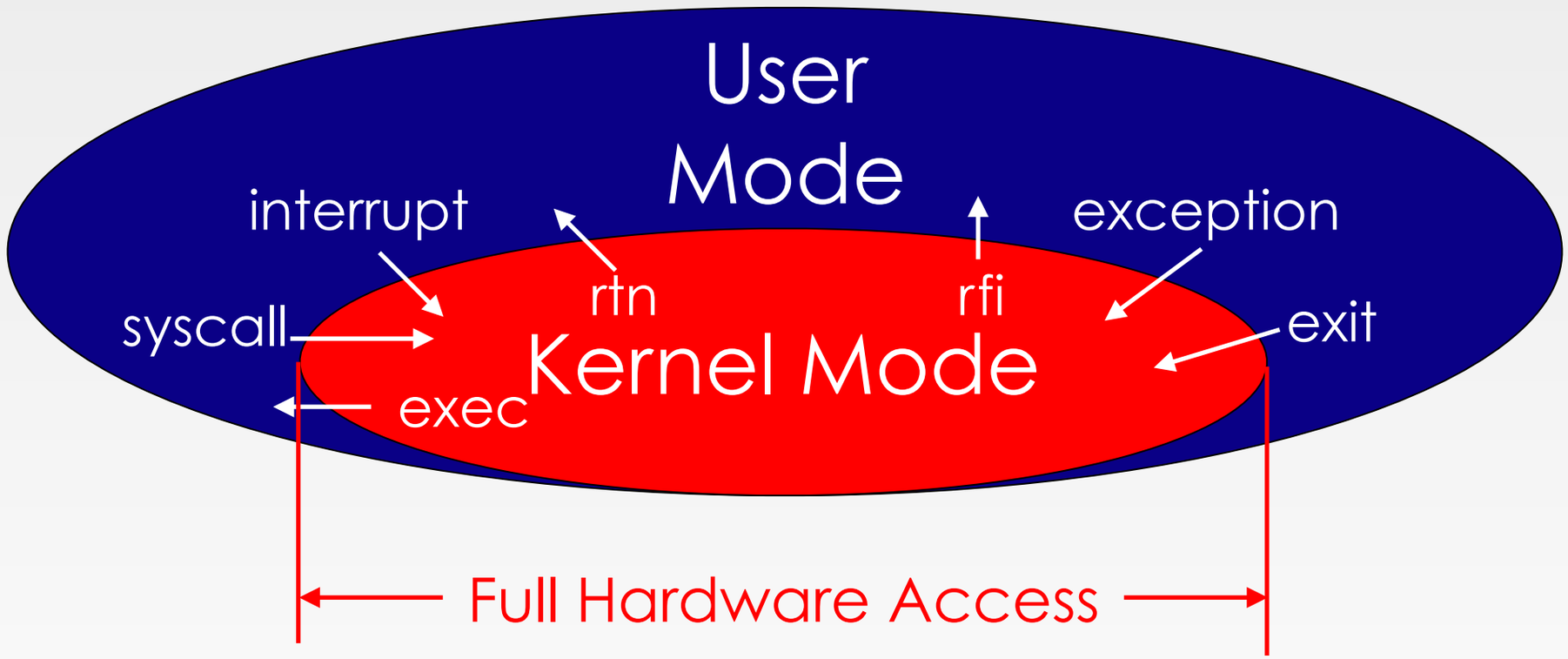
User/Kernel Mode



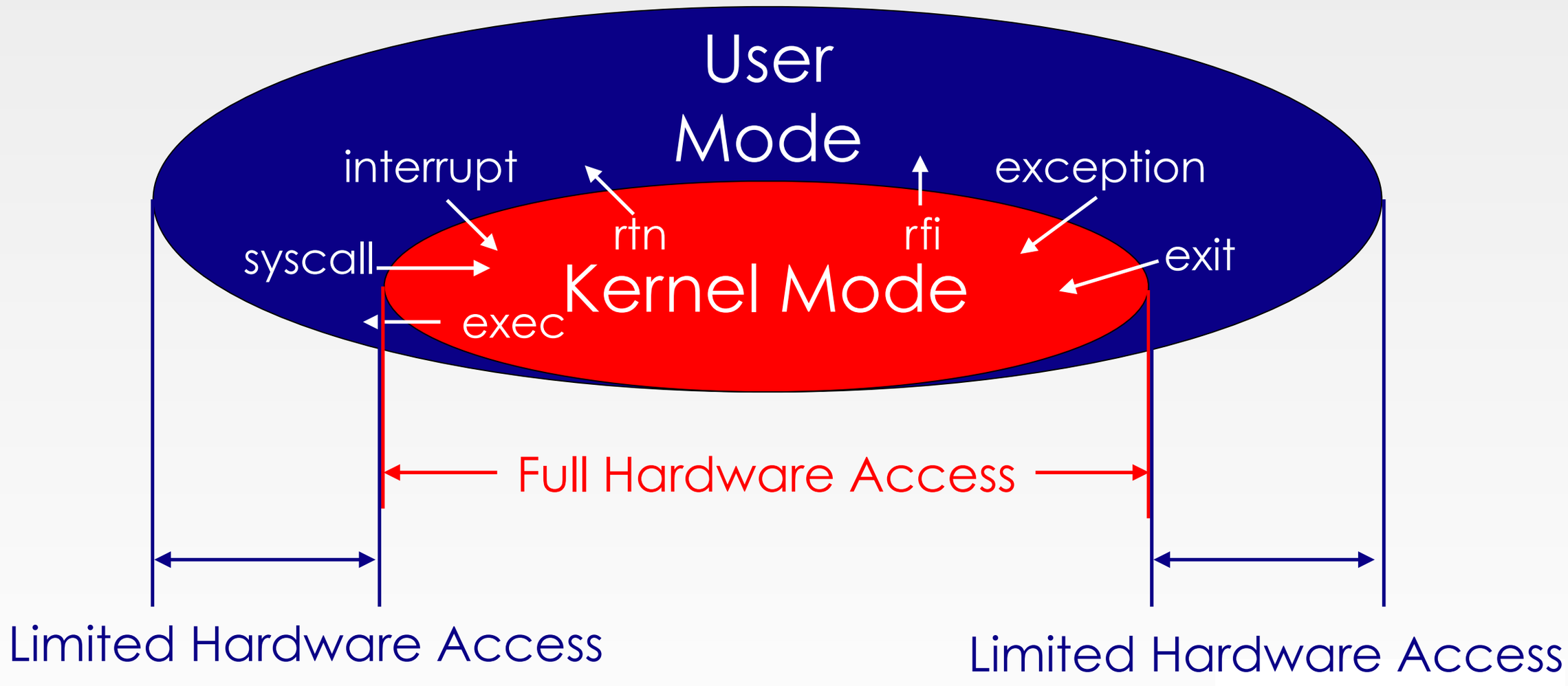
User/Kernel Mode



User/Kernel Mode



User/Kernel Mode



Implementing Safe Kernel Mode Transfers

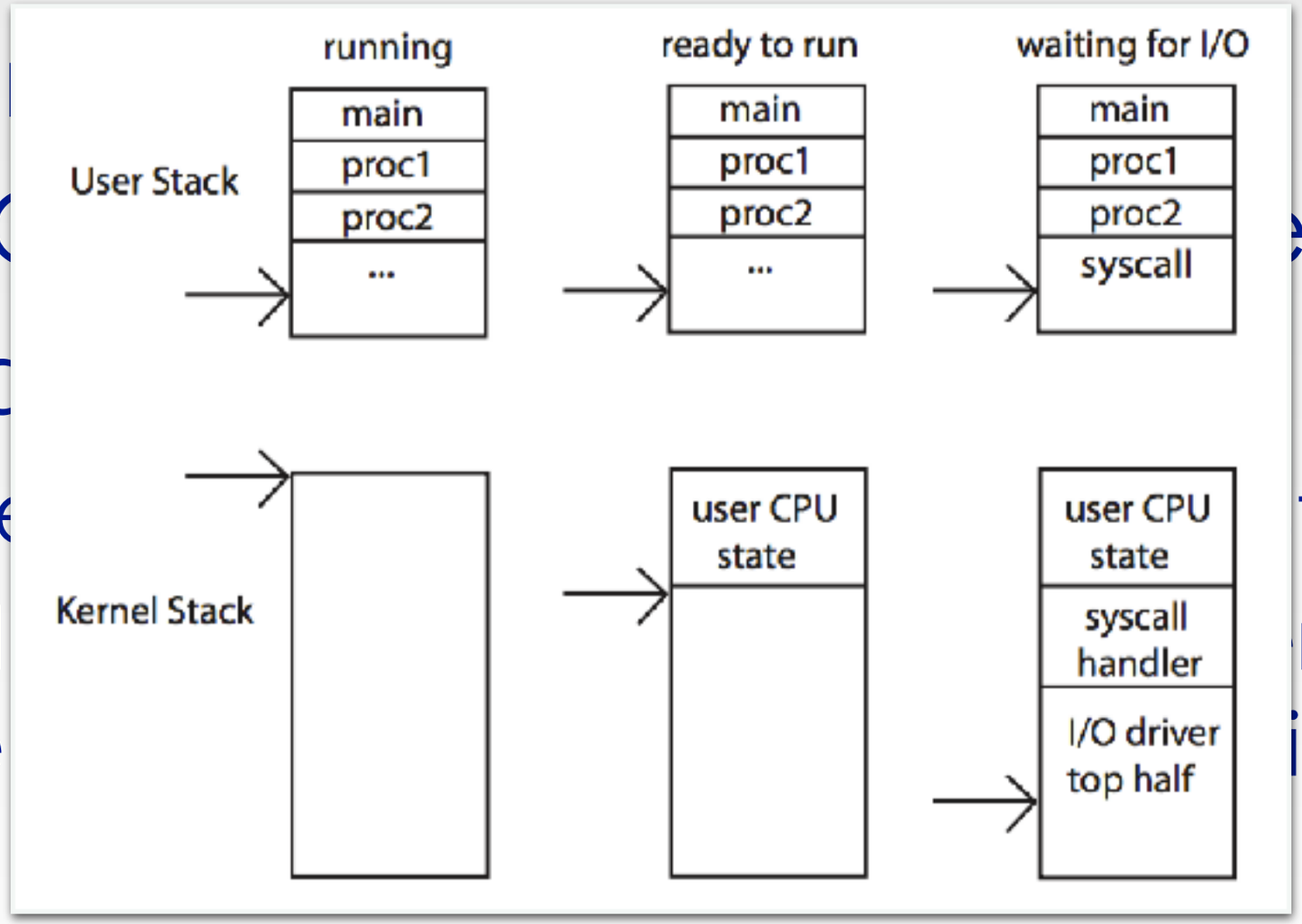
- Important Aspects
  - Separate kernel stack
  - Controlled transfer into kernel (e.g., syscall table)
- Carefully constructed kernel code packs up the user process state and sets it aside
  - Details depend on the machine architecture
- Should be impossible for buggy or malicious user program to cause the kernel to corrupt itself

Need for Separate Kernel Stacks

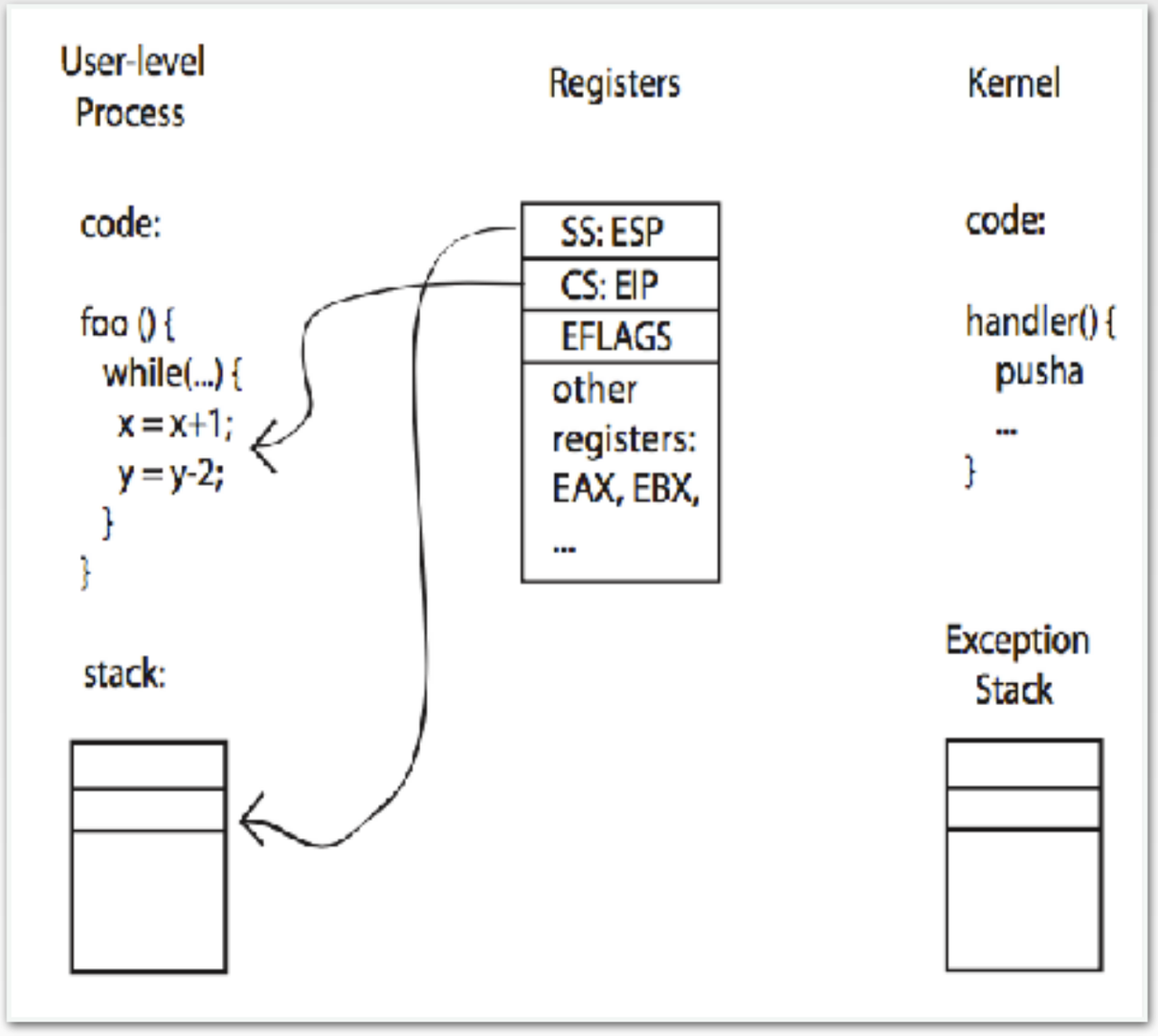
- Kernel needs space to work
- Can NOT put anything on the user stack
- Two-stack model
  - OS thread has interrupt stack + user stack
  - syscall handler copies user args to kernel space before invoking specific function

Need for Separate Kernel Stacks

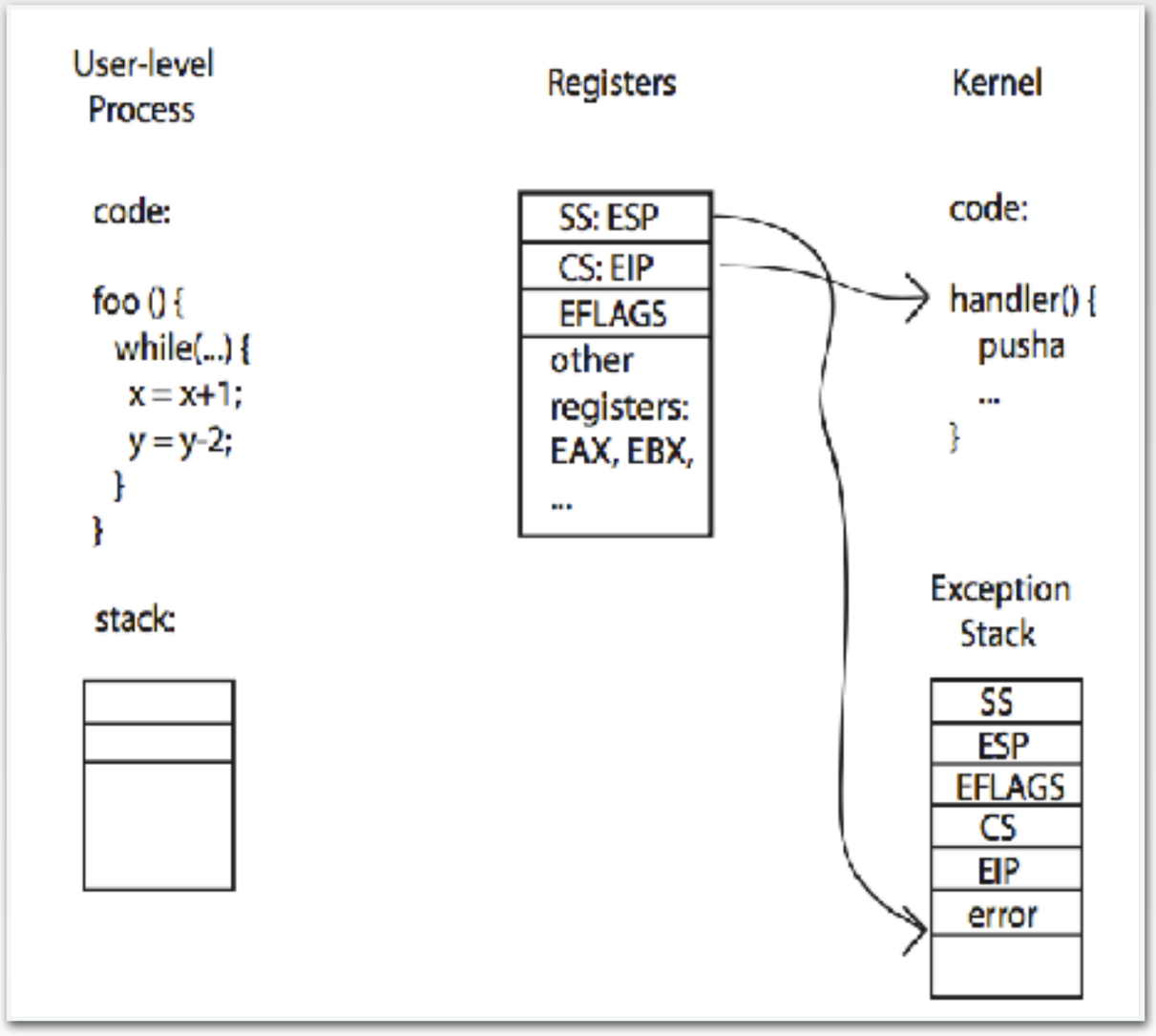
- Kernel
- Can NO er stack
- Two-sta
  - OS thre tack
  - syscall rnel space ion



Before



During



Kernel System Call Handler

- Vector through well-defined syscall entry points
  - Table mapping system call number to handle
- Locate arguments
  - In registers or on user stack
- Copy arguments
  - from user memory into kernel memory
  - protect kernel from malicious code evading checked
- Validate arguments
  - protect kernel from errors in user code
- Copy results back
  - into user memory

Hardware support: Interrupt Control

- Not be visible to the user process
  - Occurs between instructions, restarted transparently
  - No change to process state
- Interrupt Handler invoked with interrupts ‘disabled’
  - Re-enabled upon completion
  - Non-blocking
  - Pack up in a queue
  - Pass off to an OS thread for hard work
    - wake up an exiting

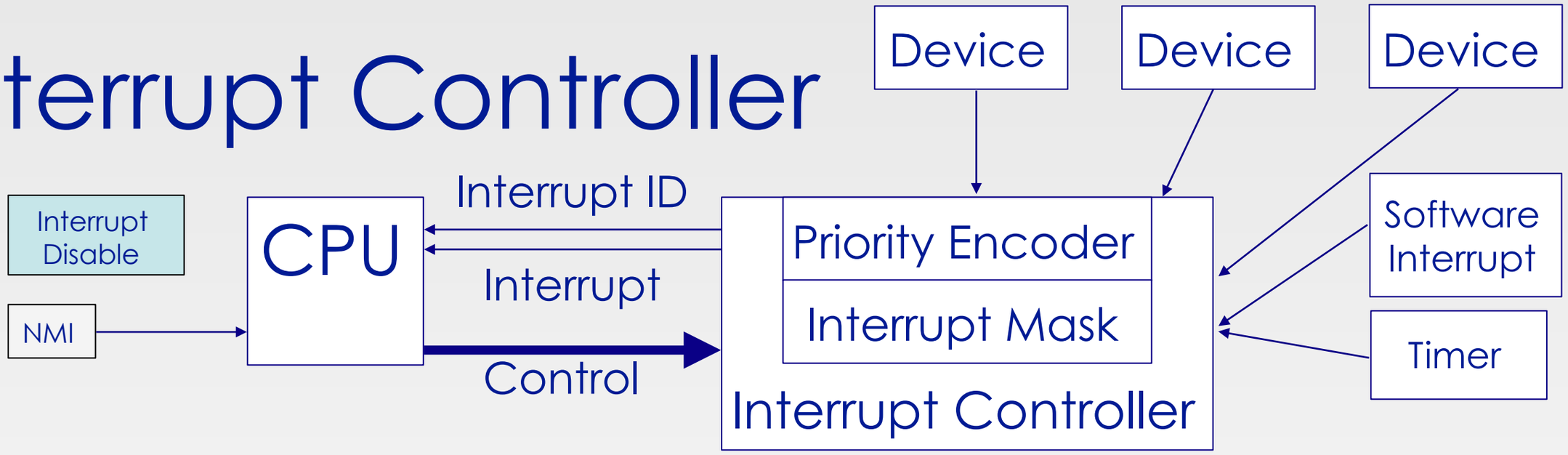
Interrupt Control (cont.)

- OS kernel may enable/disable interrupts
  - On x86
    - CLI: disable interrupts
    - STI: enable interrupts
  - Atomic section when select next process/ thread to run
  - Atomic return from interrupt or syscall

Interrupt Control (cont.)

- HW may have multiple levels of interrupt
  - Maskoff (disable) certain interrupts
    - e.g., lower priority
  - Certain Non-Maskable-Interrupt (NMI)
    - e.g., kernel segmentation fault

Interrupt Controller



- Interrupts invoked with interrupt lines form devices
- Interrupt controller chooses interrupt requests to honor
  - Mask enable/disable interrupts
  - Priority encoder Set/Clear by software
  - Interrupt identity specified with ID line
- CPU can disable all interrupts with internal flag
- NMI line can NOT be disabled

How to take interrupts safely?

- Interrupt vector
  - Limited number of entry points into kernel
- Kernel interrupt stack
  - Handler works regardless of state of user code
- Interrupt masking
  - Handler is non-blocking

Take interrupts safely (cont.)

- Atomic transfer of control
  - Single-instruction-like to change
    - PC
    - SP
    - Memory protection
    - Kernel/User mode
- Transparent restartable execution
  - User program does NOT know interrupt occurred

Process Creation

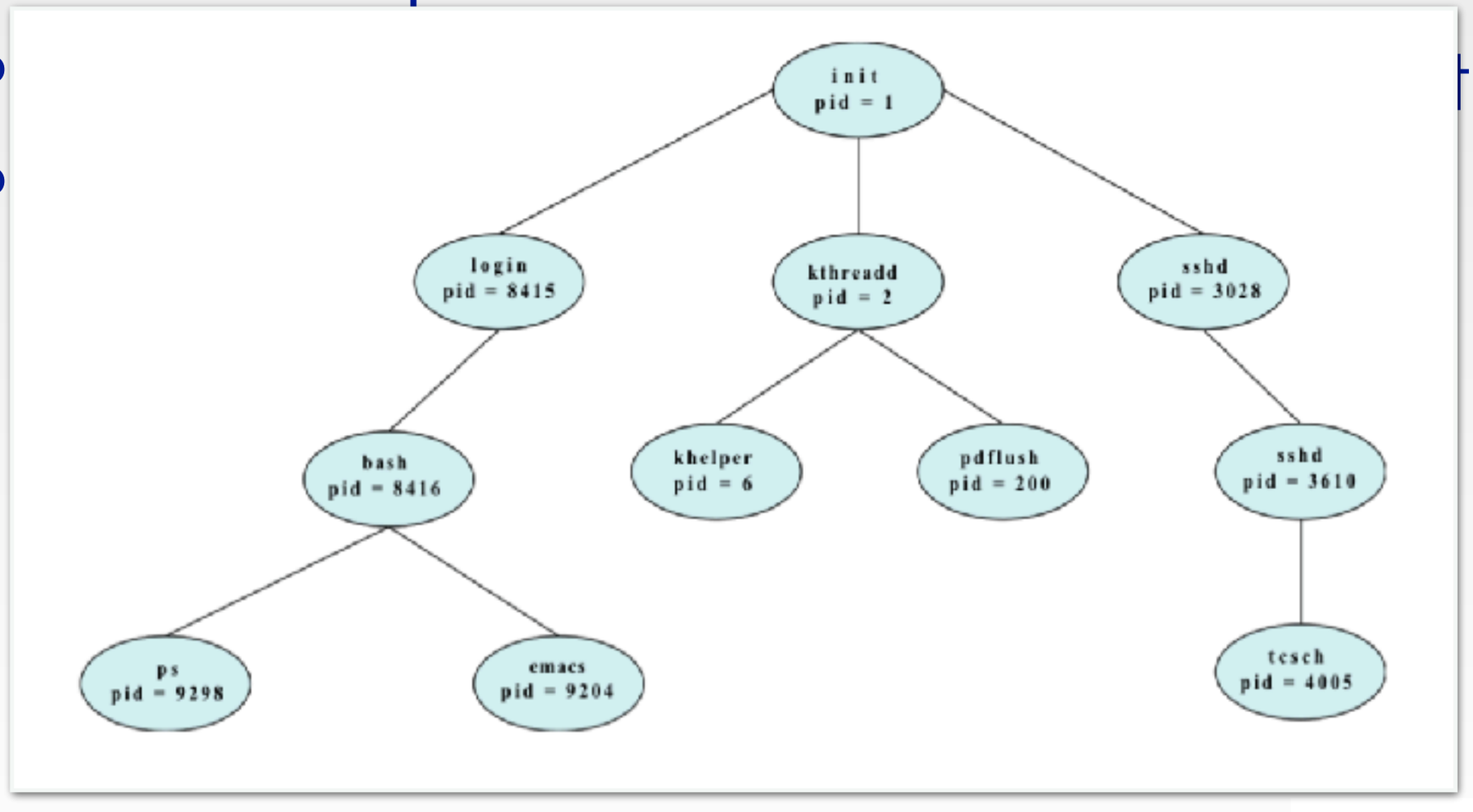
- Parent process create children processes, which create other processes
- A tree of processes
- PID: process identifier
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent & child share no resources

Process Creation (cont.)

- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (cont.)

- Execution options
  - P
  - P



Fork

- Can a process create a process?
- Yes!
- Recall: PID- unique identity of process
- `fork()` creates a copy of current process with a new PID
- `exec()` used after a `fork()` to replace the process' memory space with a new program

Fork (cont.)

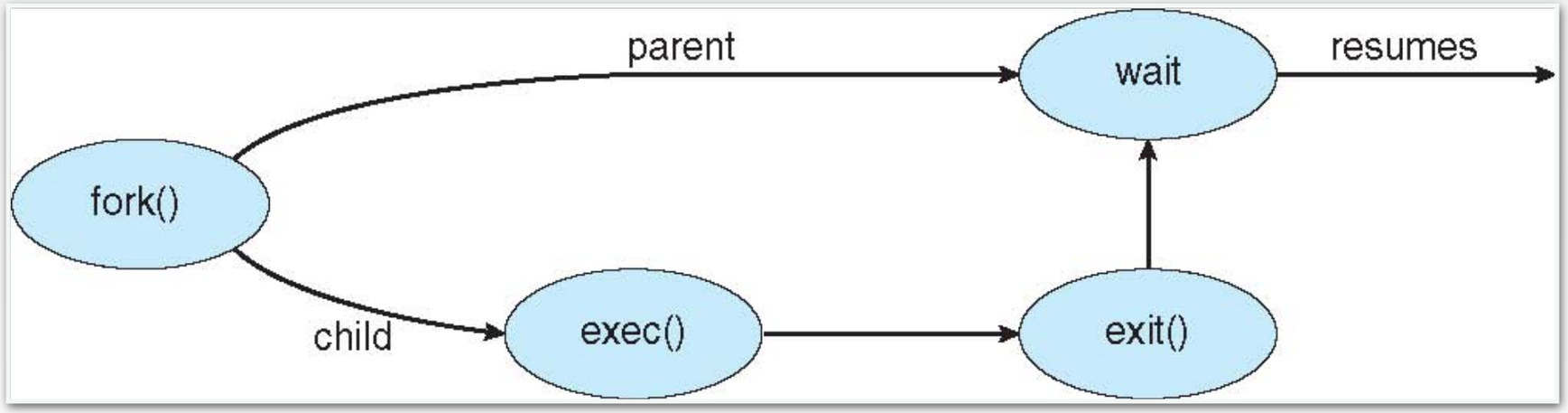
- Return value from Fork(): integer
  - >0
    - Running in *Parent* process
    - Return value is *PID* of new child
  - =0
    - Running in new *Child* process
  - <0
    - Error! Must be handled somehow
    - Running in original process

Fork (cont.)

- Return value from Fork(): integer
  - >0
    - Running in *Parent* process
  - =0
    - Running in *Child* process
  - <0
    - Error! Must be handled somehow
    - Running in original process

All state of original process is duplicated in both Parent and Child process
- Memory
- File Descriptors, etc…

Fork (cont.)



Fork(cont.)

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

fork1.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 1024
int main(int argc, char *argv[])
{
  char buf[BUFSIZE];
  size_t readlen, writelen, slen;
  pid_t cpid, mypid;
  pid_t pid = getpid();          /* get current processes PID */
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {                    /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  }  else if (cpid == 0) {          /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
    exit(1);
  }
  exit(0);
}
```

Process Termination

- `exit()`: process asks OS to delete it
  - Return status data from child to parent (via `wait()`)
  - OS de-allocate process' resources
- `abort()`: process terminate the execution of children process

Process Termination (cont.)

- If a process terminates, all its children must also be terminated
- `wait()`: return status information and the pid of the terminated process
  - `pid = wait(&status);`
- If no parent waiting, process is a zombie
- if parent terminated without `wait()`, process is an orphan

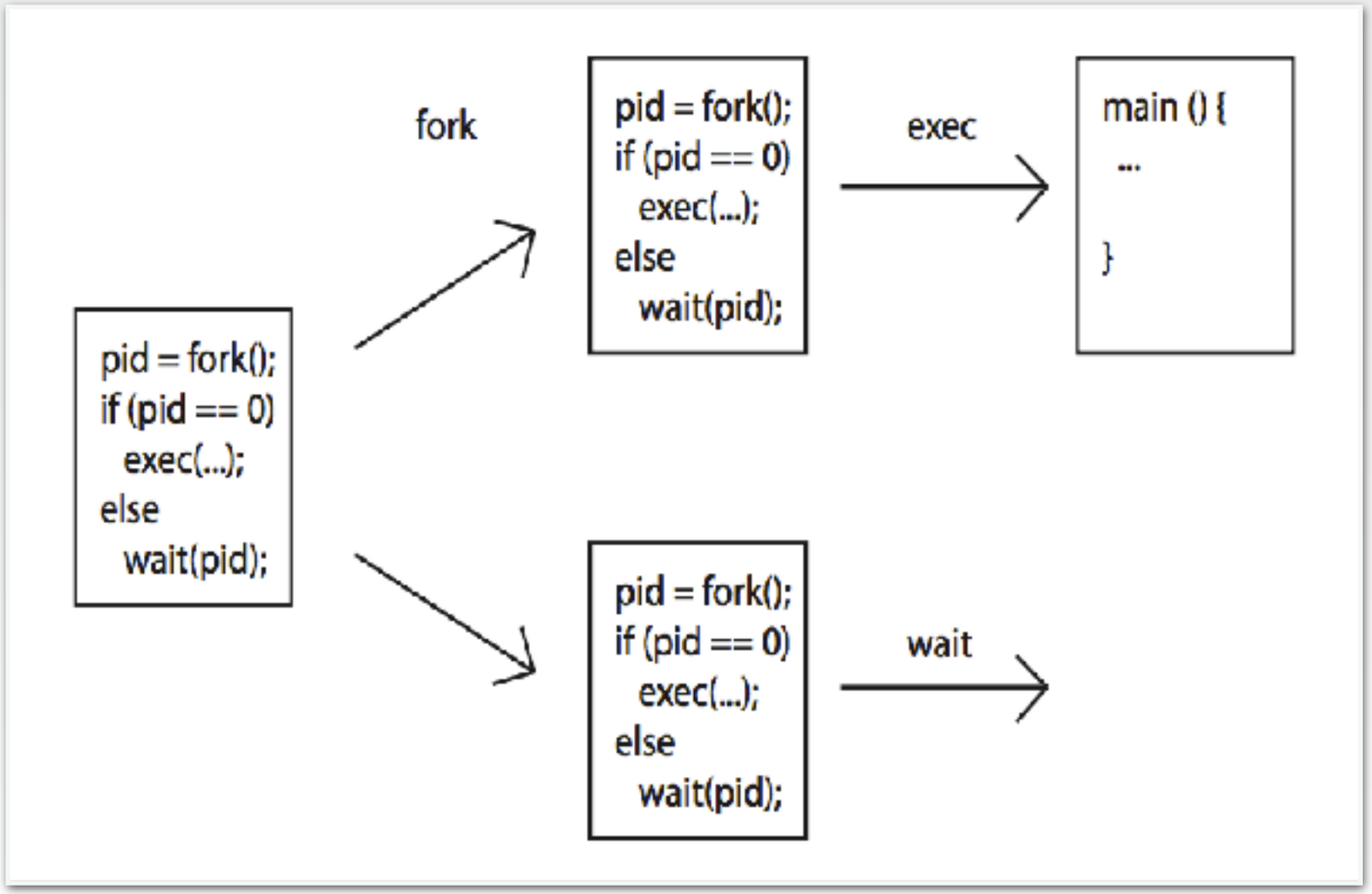
Unix Process Management

- `fork`
  - system call to *create* a copy of the current process, and start it running
- `exec`
  - system call to *change* the program being run by the current process
- `wait`
  - system call to *wait* for a process to finish
- `signal`
  - system call to send a notification to another process
- UNIX man pages for details

fork2.c

```
int status;
…
cpid = fork();
if (cpid > 0) {                    /* Parent Process */
  mypid = getpid();
  printf("[%d] parent of [%d]\n", mypid, cpid);
  tcpid = wait(&status);
  printf("[%d] bye %d(%d)\n", mypid, tcpid, status);
}  else if (cpid == 0) {           /* Child Process */
  mypid = getpid();
  printf("[%d] child\n", mypid);
}
…
```

Unix Process Management



Process Races: fork3.c

```
int i;
cpid = fork();
if (cpid > 0) {
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
    for (i=0; i<100; i++) {
      printf("[%d] parent: %d\n", mypid, i);
      //       sleep(1);
    }
} else if (cpid == 0) {
    mypid = getpid();
    printf("[%d] child\n", mypid);
    for (i=0; i>-100; i--) {
      printf("[%d] child: %d\n", mypid, i);
      //       sleep(1);
    }
}
```

Process Races: fork3.c

```
[4574] parent of [4575]
[4574] parent: 0
[4574] parent: 1
[4574] parent: 2
[4574] parent: 3
[4574] parent: 4
[4574] parent: 5
[4574] parent: 6
[4574] parent: 7
[4574] parent: 8
[4574] parent: 9
[4574] parent: 10
[4574] parent: 11
[4574] parent: 12
[4574] parent: 13
[4574] parent: 14
[4574] parent: 15
[4574] parent: 16
[4574] parent: 17
[4574] parent: 18
[4574] parent: 19
[4575] child
[4575] child: 0
[4575] child: -1
[4575] child: -2
[4575] child: -3
[4575] child: -4
[4575] child: -5
[4575] child: -6
[4575] child: -7
[4575] child: -8
[4575] child: -9
[4575] child: -10
[4575] child: -11
[4575] child: -12
[4575] child: -13
[4575] child: -14
[4575] child: -15
[4575] child: -16
[4575] child: -17
[4575] child: -18
[4575] child: -19
```

```
int i;
cpid = fork();
if (cpid > 0) {
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
    for (i=0; i<100; i++) {
      printf("[%d] parent: %d\n", mypid, i);
      //       sleep(1);
    }
} else if (cpid == 0) {
    mypid = getpid();
    printf("[%d] child\n", mypid);
    for (i=0; i>-100; i--) {
      printf("[%d] child: %d\n", mypid, i);
      //       sleep(1);
    }
}
```

Process Races: fork3.c

```
int i;
cpid = fork();
if (cpid > 0) {
    mypid = getpid();
    printf("[%d] parent of [%d]\n",
    for (i=0; i<100; i++) {
      printf("[%d] parent: %d\n", r
      //      sleep(1);
    }
} else if (cpid == 0) {
   mypid = getpid();
   printf("[%d] child\n", mypid);
   for (i=0; i>-100; i--) {
     printf("[%d] child: %d\n", my
     //      sleep(1);
   }
}
```

```
[4574] parent of [4575]
[4574] parent: 0
[4574] parent: 1
[4574] parent: 2
[4574] parent: 3
[4574] parent: 4
[4574] parent: 5
[4574] parent: 6
[4574] parent: 7
[4574] parent: 8
[4574] parent: 9
[4574] parent: 10
[4574] parent: 11
[4574] parent: 12
[4574] parent: 13
[4574] parent: 14
[4574] parent: 15
[4574] parent: 16
[4574] parent: 17
[4574] parent: 18
[4574] parent: 19
[4575] child
[4575] child: 0
[4575] child: -1
[4575] child: -2
[4575] child: -3
[4575] child: -4
[4575] child: -5
[4575] child: -6
[4575] child: -7
[4575] child: -8
[4575] child: -9
[4575] child: -10
[4575] child: -11
[4575] child: -12
[4575] child: -13
[4575] child: -14
[4575] child: -15
[4575] child: -16
[4575] child: -17
[4575] child: -18
[4575] child: -19
```

```
[4615] parent of [4616]
[4615] parent: 0
[4616] child
[4616] child: 0
[4616] child: -1
[4615] parent: 1
[4615] parent: 2
[4616] child: -2
[4615] parent: 3
[4616] child: -3
[4616] child: -4
[4615] parent: 4
[4616] child: -5
[4615] parent: 5
[4616] child: -6
[4615] parent: 6
[4616] child: -7
[4615] parent: 7
[4616] child: -8
[4615] parent: 8
[4616] child: -9
[4615] parent: 9
[4616] child: -10
[4615] parent: 10
[4616] child: -11
[4615] parent: 11
[4616] child: -12
[4615] parent: 12
[4616] child: -13
[4615] parent: 13
[4616] child: -14
[4615] parent: 14
[4616] child: -15
[4615] parent: 15
[4616] child: -16
[4615] parent: 16
[4616] child: -17
[4615] parent: 17
[4616] child: -18
[4615] parent: 18
[4616] child: -19
[4615] parent: 19
```

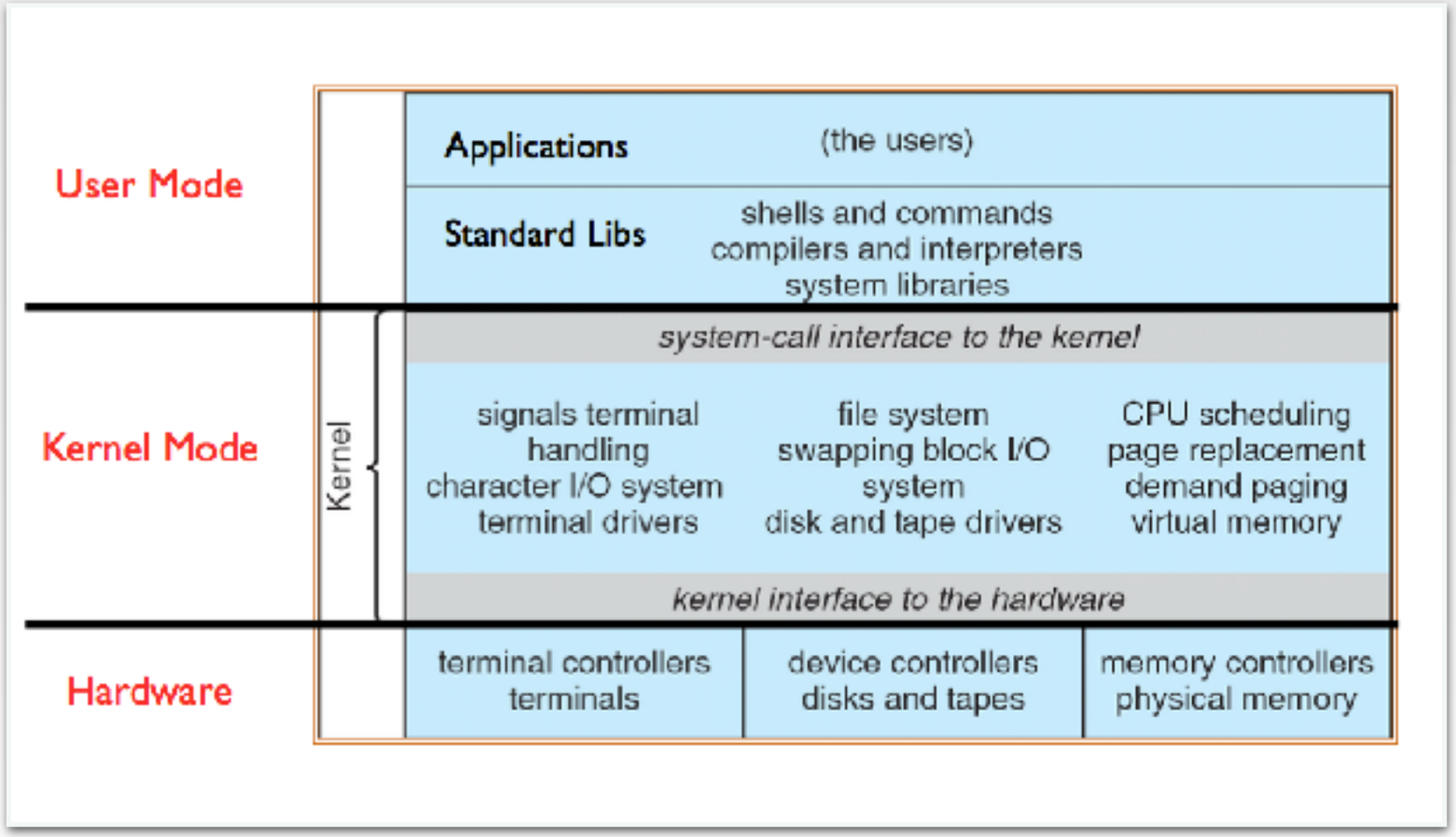
To sum up

- If `fork()` returns a negative value
  - creation of a child process was un-successful
- `fork()` returns a zero to the newly created child process
- `fork()` returns a positive value (child process PID) to the parent

To Sum Up (cont.)

- UNIX makes an EXACT copy of the parent's address space and give it to the child
- Hence, parent and child precesses have separate address space

Recall: UNIX System Structure

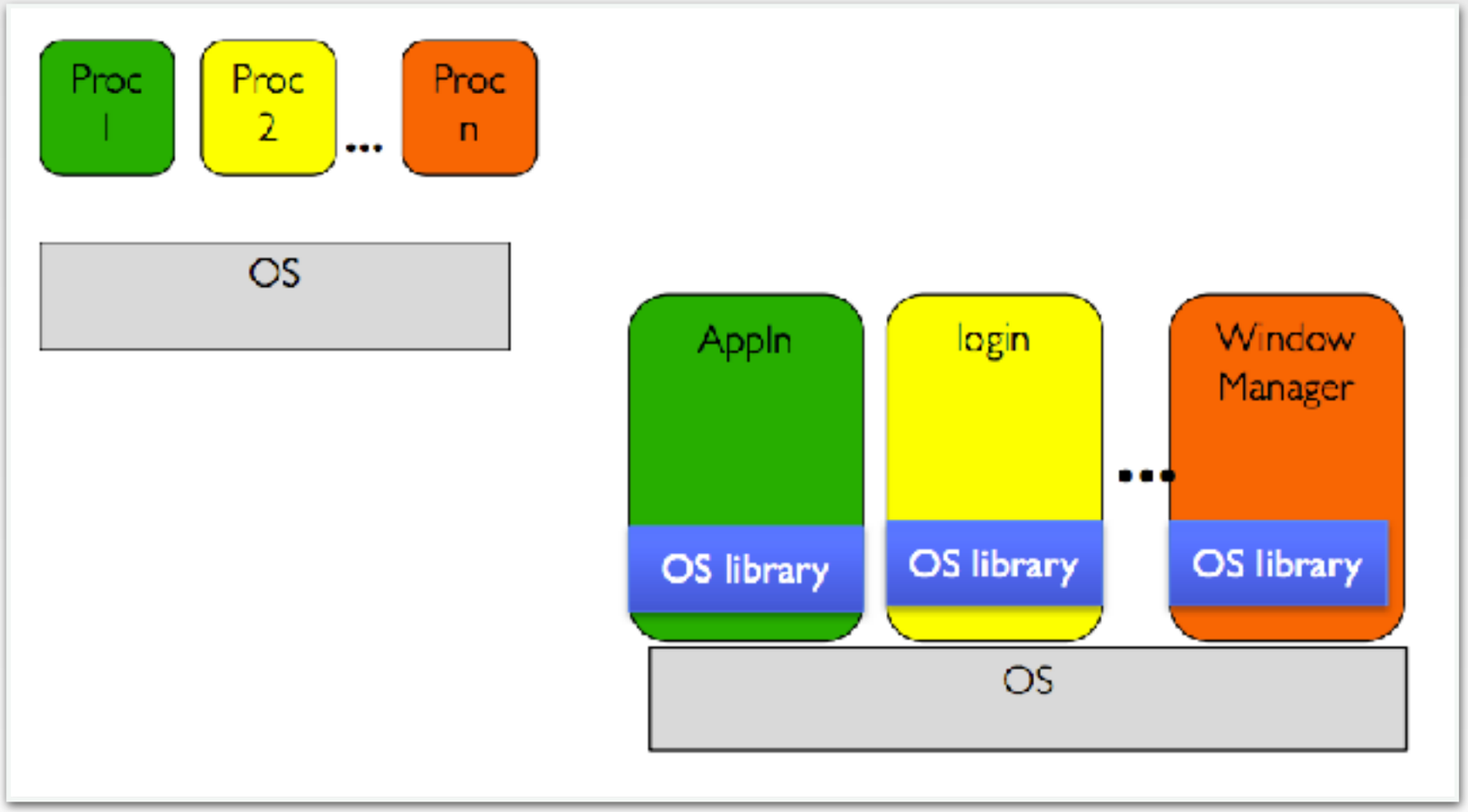


How does the kernel provides service?

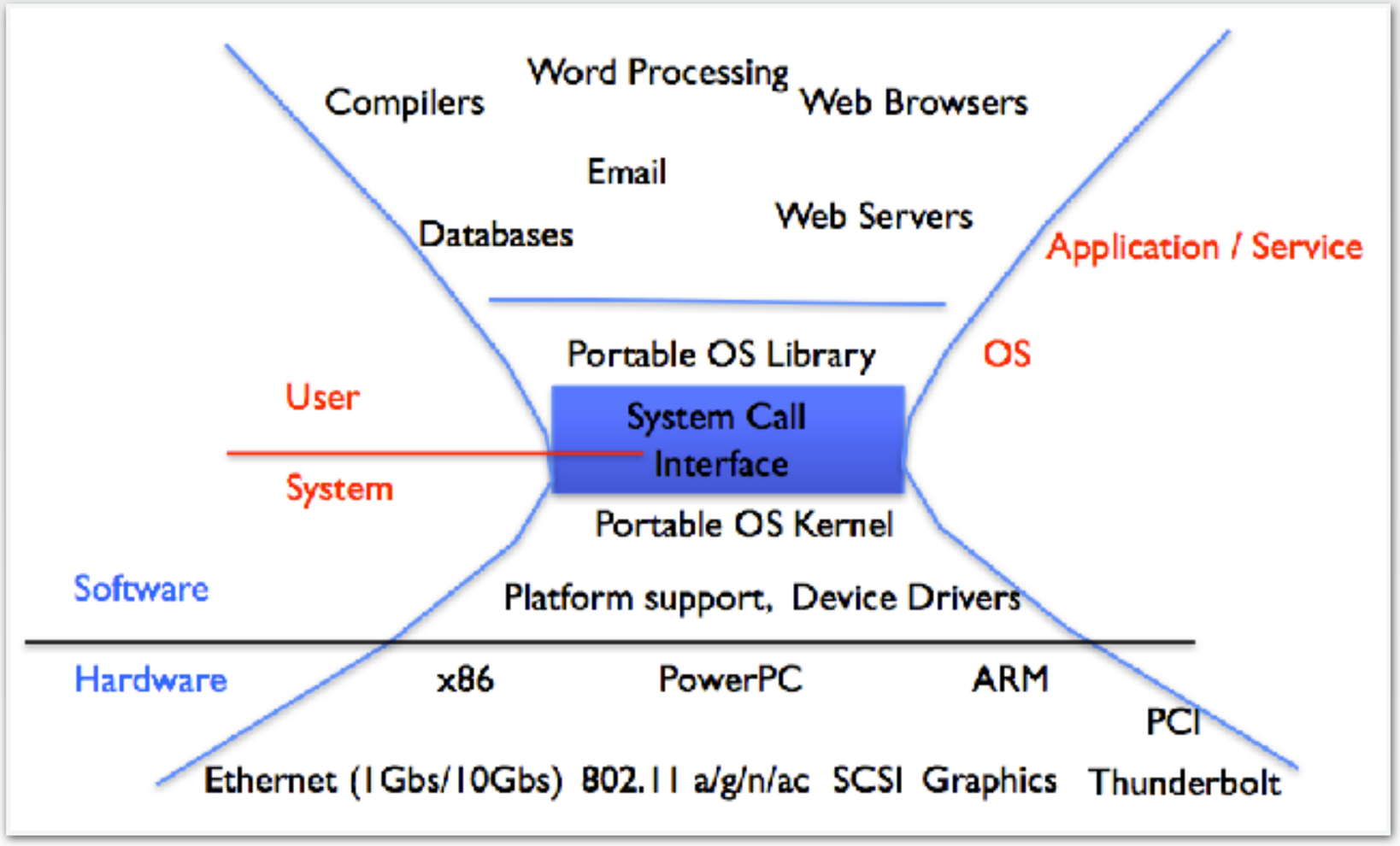
- You said applications request services from the OS via `syscall` but…
- I've been writing all sort of useful applications and I never ever saw a "`syscall`"…

- Right!
- syscall was buried in the programming language runtime library (e.g., `libc.a`)

OS Run-Time Library



A Kind of Narrow Waist



Conclusion

- 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 process resources
- Interrupts
  - HW mechanism for regaining control from user
  - Notification that events have occurred
  - User-level equivalent: Signals
- Native control of Process
  - `fork`