COMP 4735: Operating Systems

Lecture 5: System Calls

Rob Neilson

rneilson@bcit.ca

Reading

- The following sections should be read before next Monday
  - it would also help to read this before your lab this week
  - Textbook Sections: 1.5, 1.6, 1.7 (Theory Part)
    10.2 (Case Study Part)
- Yes, there will be a quiz next Monday in lecture on the above sections
  - A sample quiz will be posted on webct on Friday
  - This quiz (Quiz 2) covers material from all of the sections listed above

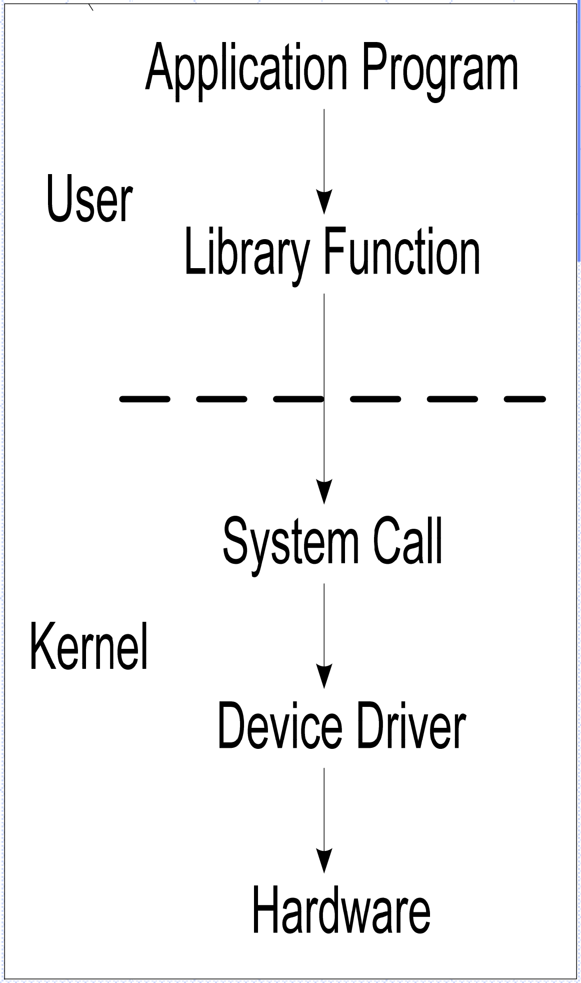
Agenda

Key concepts for this lesson:

- System Calls
- Processor Modes
- User Space vs Kernel Space
- Traps and Signals
- OS Modularization
- Monolithic vs Micro-kernel Design
- Unix Architecture

System Calls and Library Functions

- ***Library functions*** (eg: printf) are what you use when you write a program (in your role as application programmer)
  - in C these are referenced by including a header (.h) file, and linked by with your object file to create an executable image
  - in Java these are referenced by using the import statement to identify the classes you want to use
- ***System calls*** are what the people who write the library functions use
  - a system call is a program that runs in the kernel
  - the system call interacts with the hardware via device drivers



System Calls and Library Functions - 2

- Consider a C program "Hello World", compiled, linked, and saved in file `a.out`
- When you execute this program from a shell on a unix-like system, the following happens:

1. you type `a.out` and press ENTER
2. the shell forks a new process, and loads the executable image from `a.out`, using the `exec()` system call
3. the new process starts executing `a.out` from the beginning of its entry point (ie: `main()`)
4. the library function `printf(...)` is called by `main()`
5. `printf` does some formatting, and calls another library function named `write()`
6. the library function `write()` traps to the kernel, and invokes the system call *`write()`*
7. *`write()`* uses the device driver for the default terminal to send the bytes for "Hello World" to the console

How does write() work?

System Calls: Steps 1-4 (refer to next diagram)

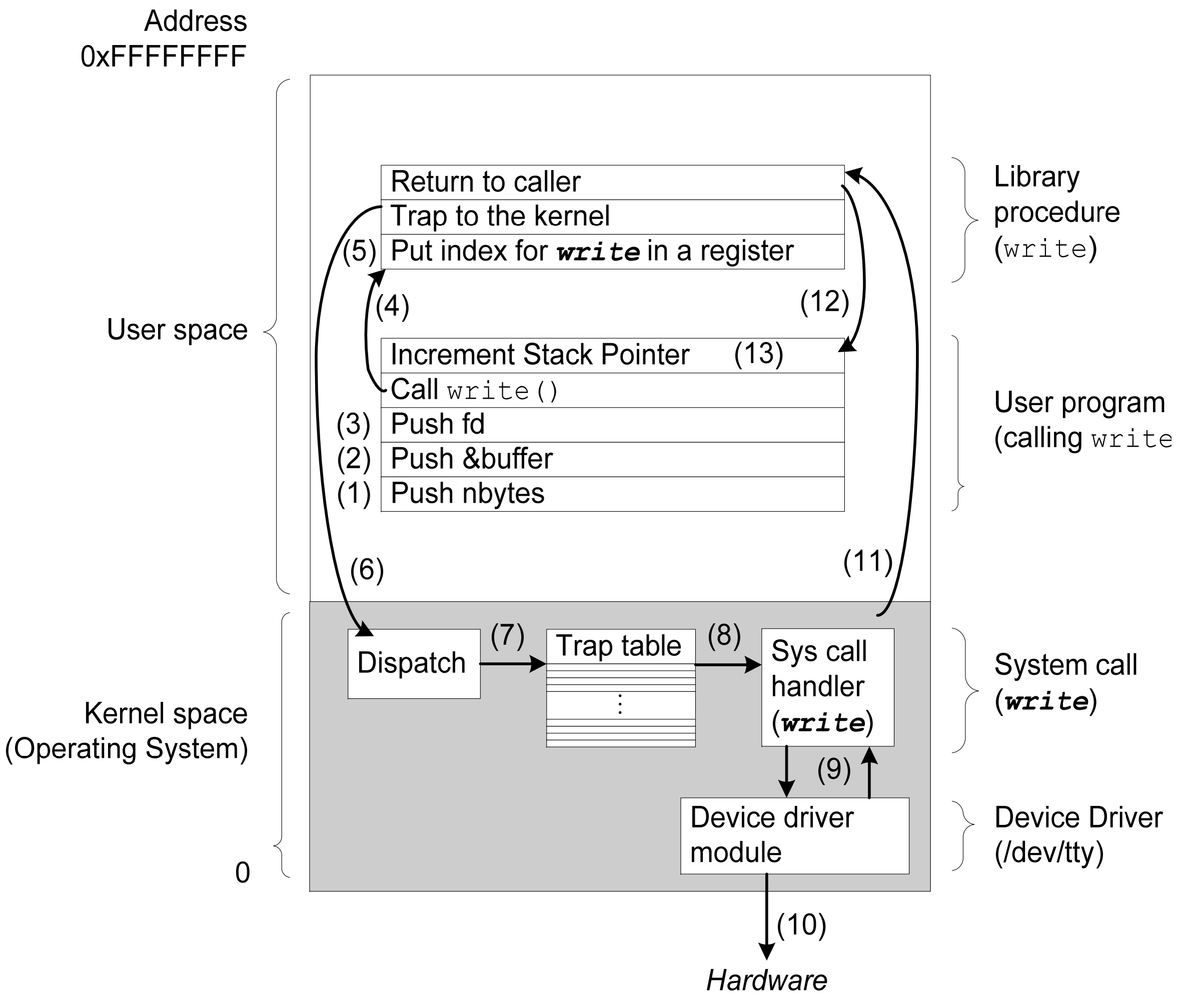
- First, the library function for **`printf`** is called. This library function calls another library function, **`write(2)`**, which in turn invokes the necessary system calls to complete the operation.
- Our diagram (next slide) starts as the **`write(2)`** library function is about to be called.
- **`write(2)`** has the following prototype
  - **`ssize_t write(int`** ***`fd`*****`, const void *`*****`buf`*****`, size_t`** ***`nbytes`*****`);`**

The parameters for write are pushed onto the stack in reverse order:

1) push the size of the string to be output
2) push the address of the string
3) push the file descriptor (default fd == 1 means stdout, or /dev/tty)
4) the user library function is called (ie: the program branches to the instructions in the library that implement **`write(2)`**)

Making a System Call (Unix) -123

Looking at this operation in detail….



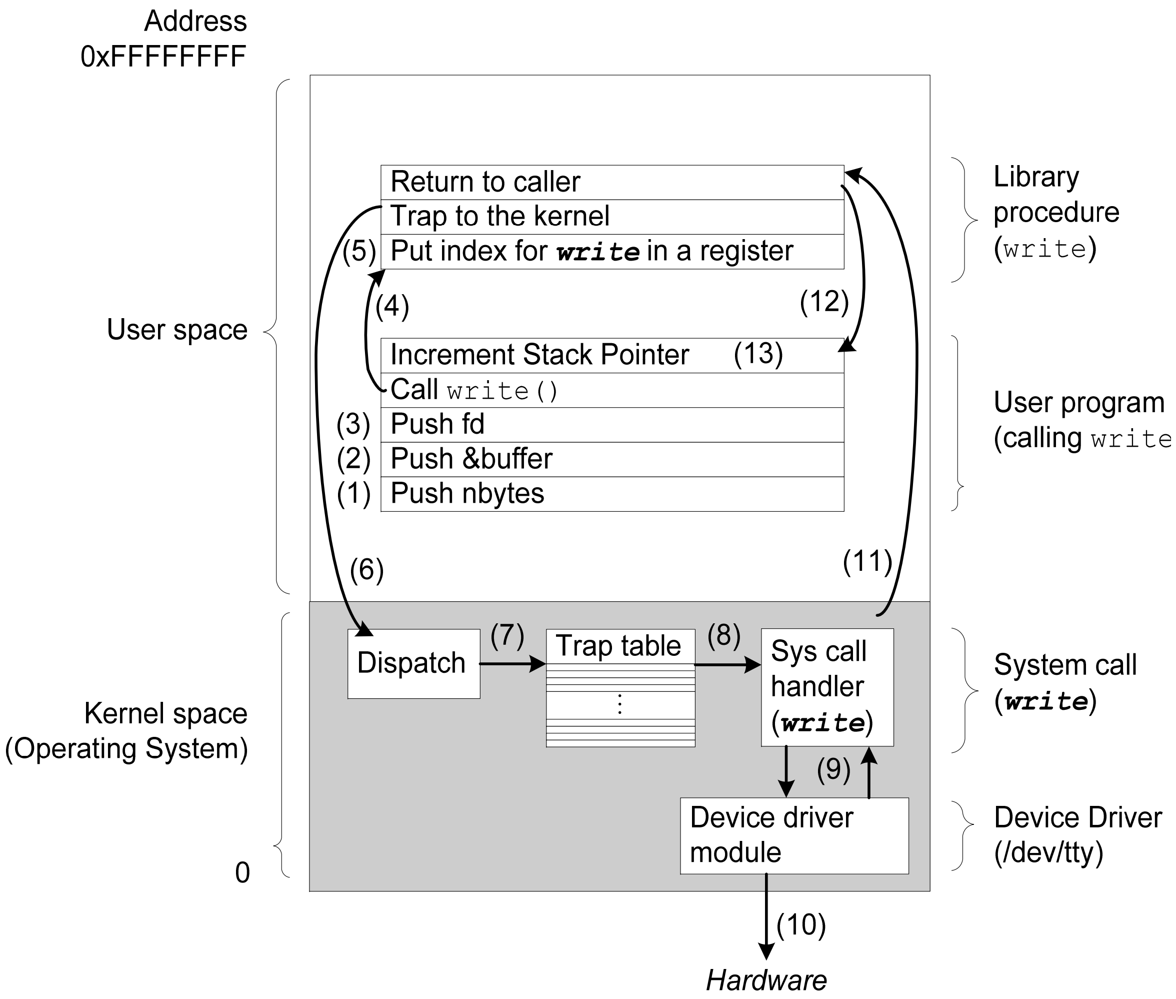
steps 1/2/3:

push the parameters onto the processes stack in reverse order

(the compiler included this code for you when it compiled the function call

Making a System Call (Unix) -4

Looking at this operation in detail….



steps 1/2/3:

push the parameters onto the processes stack in reverse order

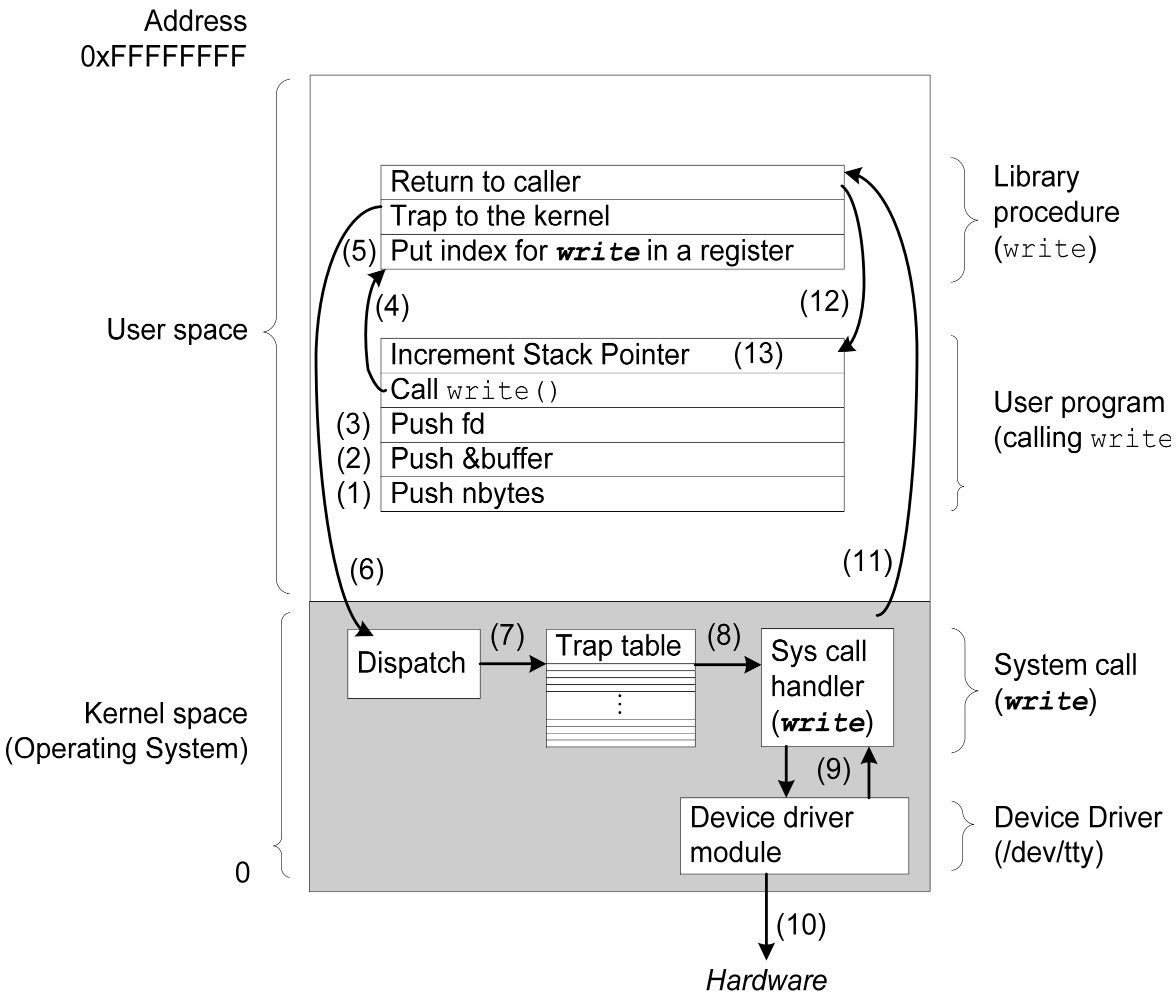
(the compiler included this code for you when it compiled the function call

step 4:

the user library function is called

Making a System Call (Unix) -5

Looking at this operation in detail….



steps 1/2/3:

push the parameters onto the processes stack in reverse order

(the compiler included this code for you when it compiled the function call

step 4:

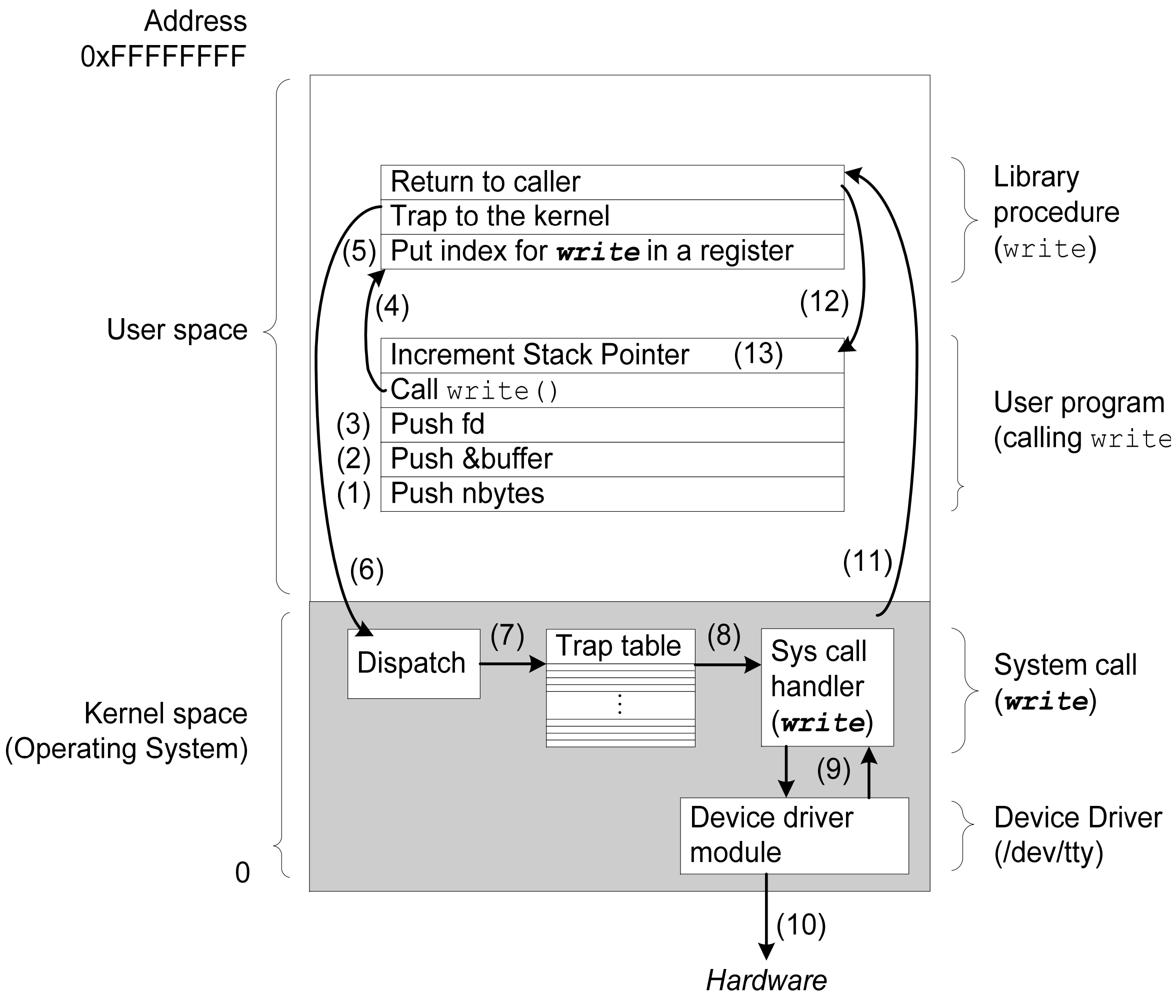
the user library function is called

step 5:

the write(2) library function puts a trap table index for the *write system call* in a place that the OS expects it (such as a register)

Making a System Call (Unix) -6

Looking at this operation in detail….



step 4:

the user library function is called

step 5:

the write(2) library function puts a trap table index for the *write system call* in a place that the OS expects it (such as a register)

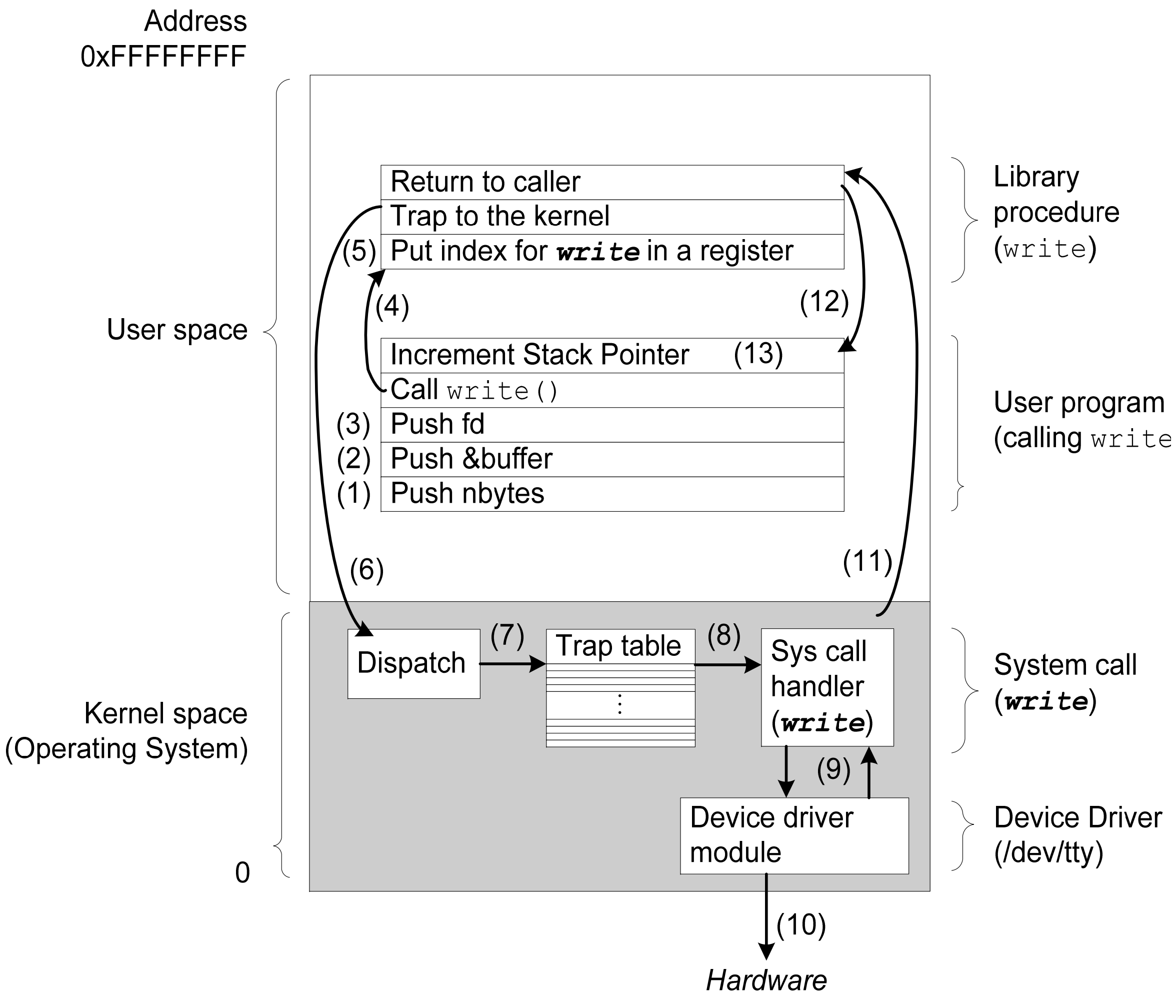
step 6:

the library function executes a TRAP instruction which causes the OS to

- change from *user mode* to *kernel mode* (more on this later)
- start executing code at a fixed entry point in the kernel

Making a System Call (Unix) -7

Looking at this operation in detail….



step 6:

the library function executes a TRAP instruction which causes the OS to

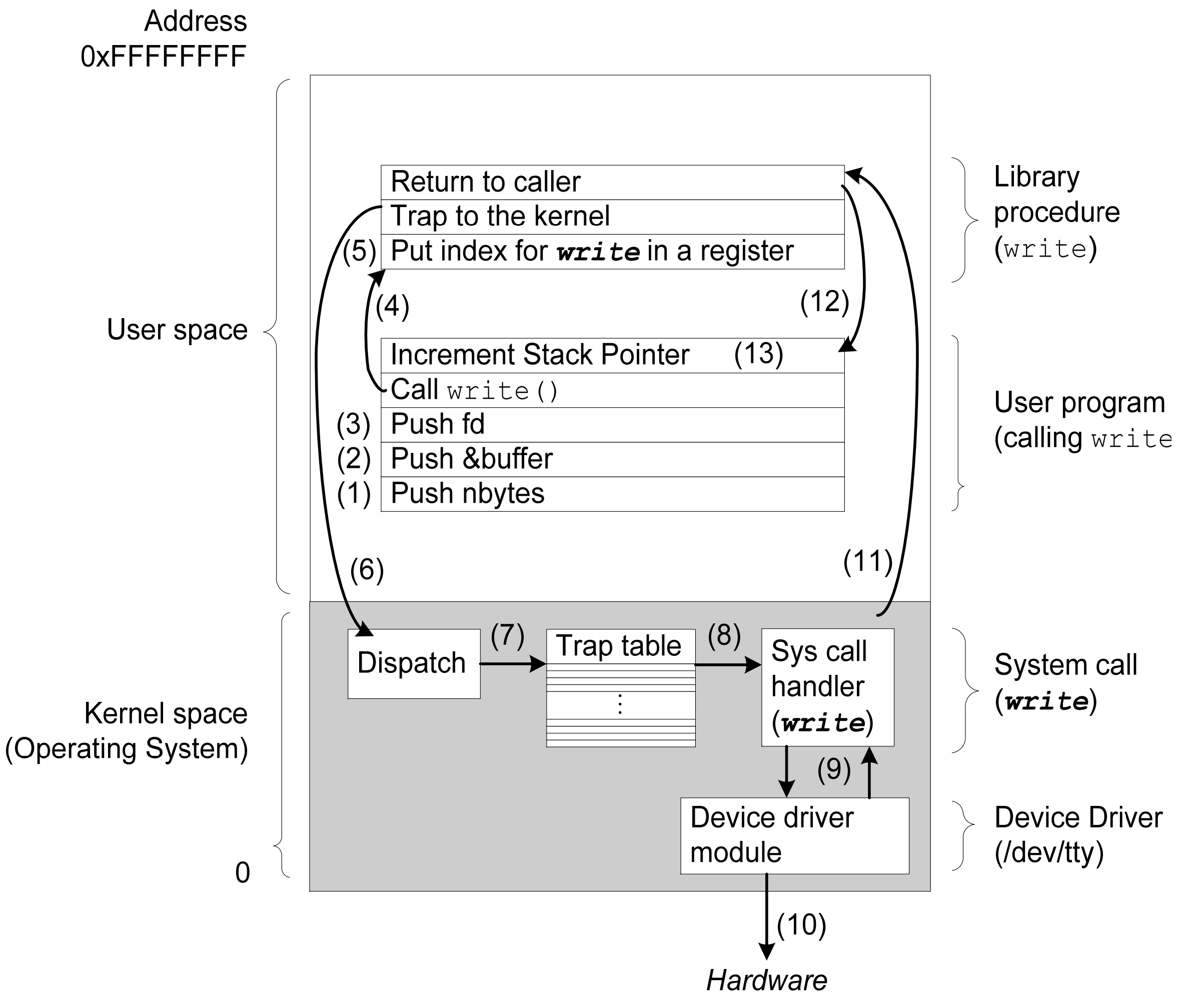
- change from *user mode* to *kernel mode* (more on this later)
- start executing code at a fixed entry point in the kernel

step 7:

the kernel code reads the trap table index from the register, and looks up the address of the system call (**write**) handler in the trap table

Making a System Call (Unix) -8

Looking at this operation in detail….



step 7:

the kernel code reads the trap table index from the register, and looks up the address of the system call (***write***) handler in the trap table

step 8:

the kernel branches to the address of the system call (***write***) handler ...

... the code for the *write* system call handler starts executing

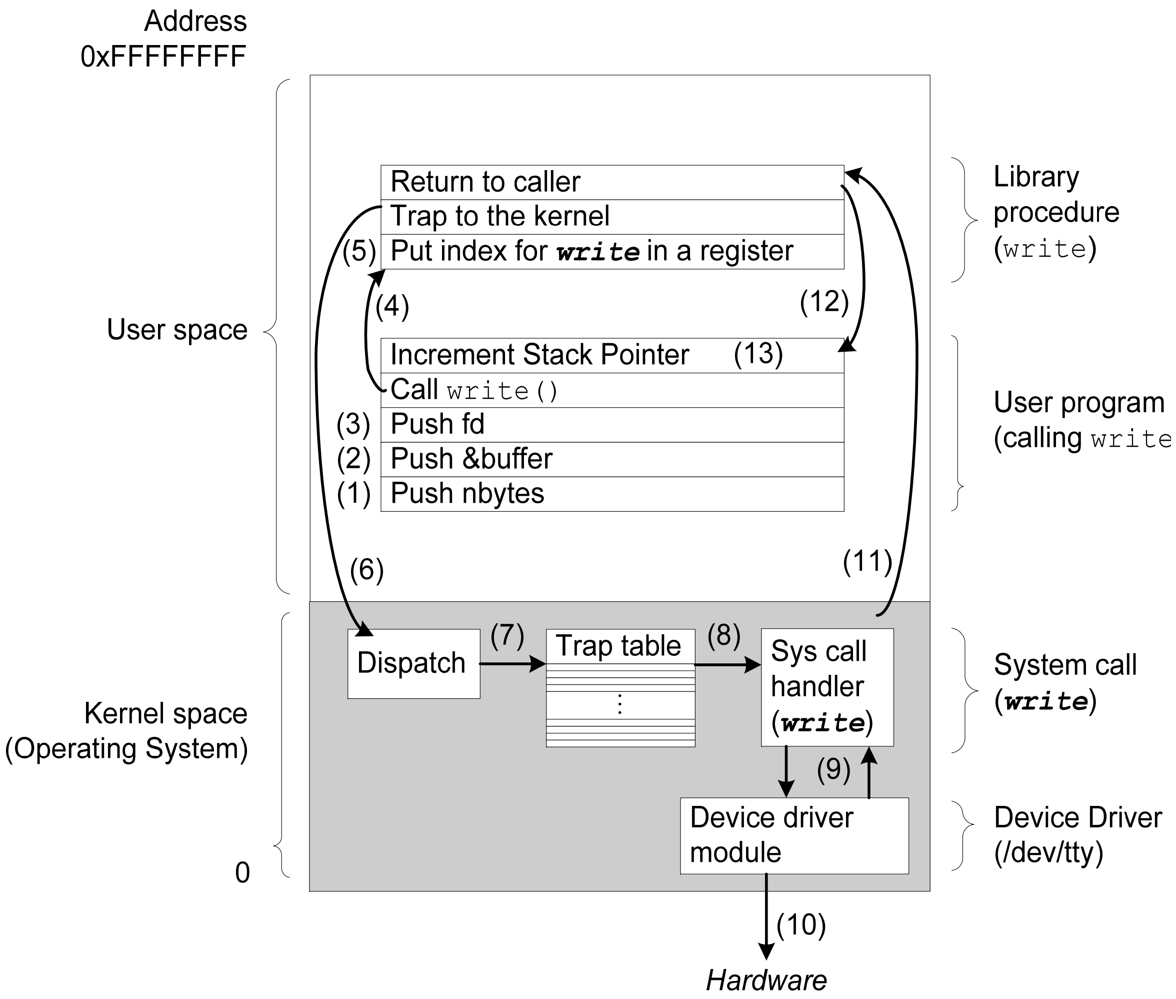
Note:

all this is happening in the context of the user process ...

... we have not changed process context at all

Making a System Call (Unix) -9

Looking at this operation in detail….



step 8:

the kernel branches to the address of the system call (**`write`**) handler ...

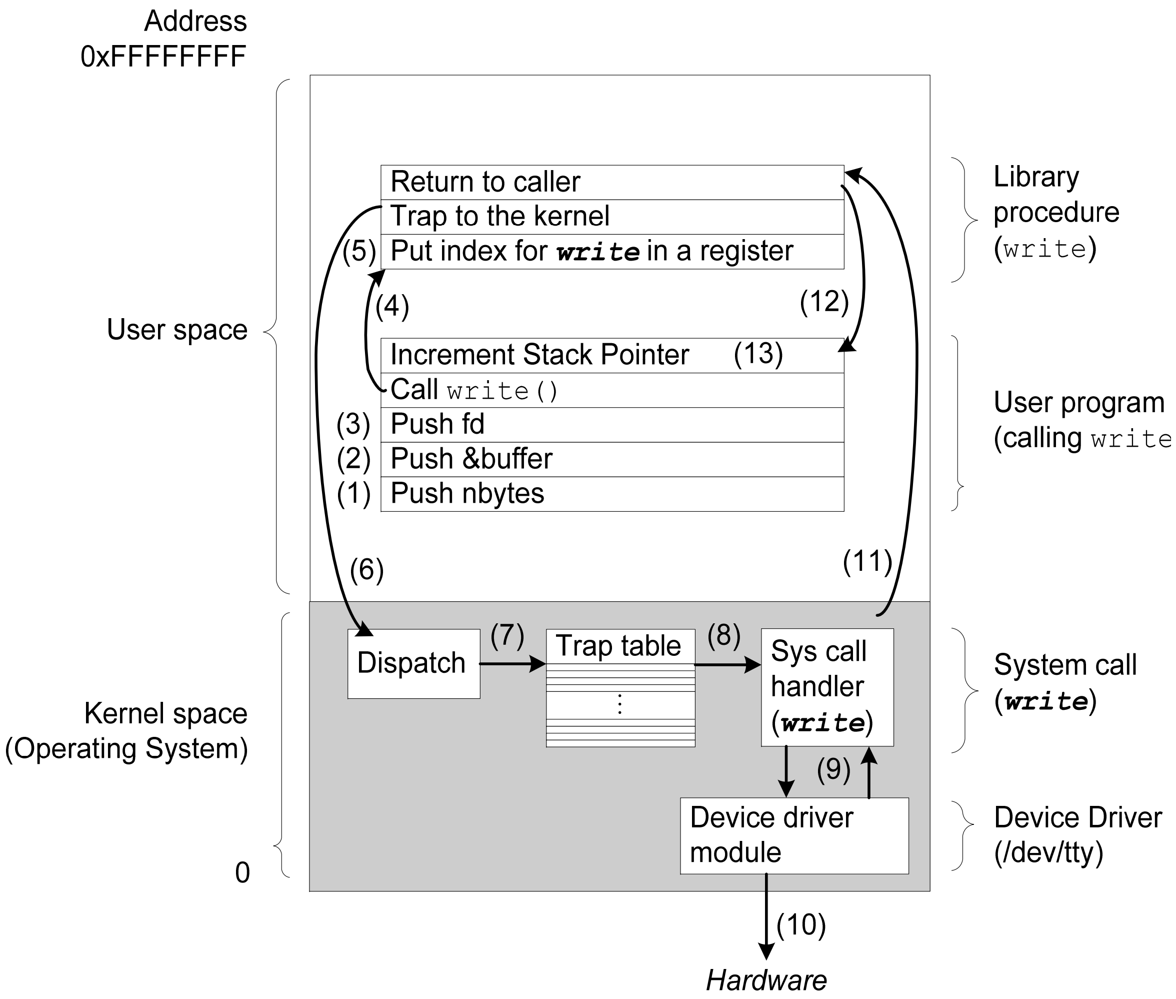
... the code for the *write* system call handler starts executing

step 9:

the code for the system call **`write`** retrieves the parameters from the stack and jumps to the appropriate device driver code for the device being written to

Making a System Call (Unix) -10

Looking at this operation in detail….



step 8:

the kernel branches to the address of the system call (**write**) handler ...

... the code for the *write* system call handler starts executing

step 9:

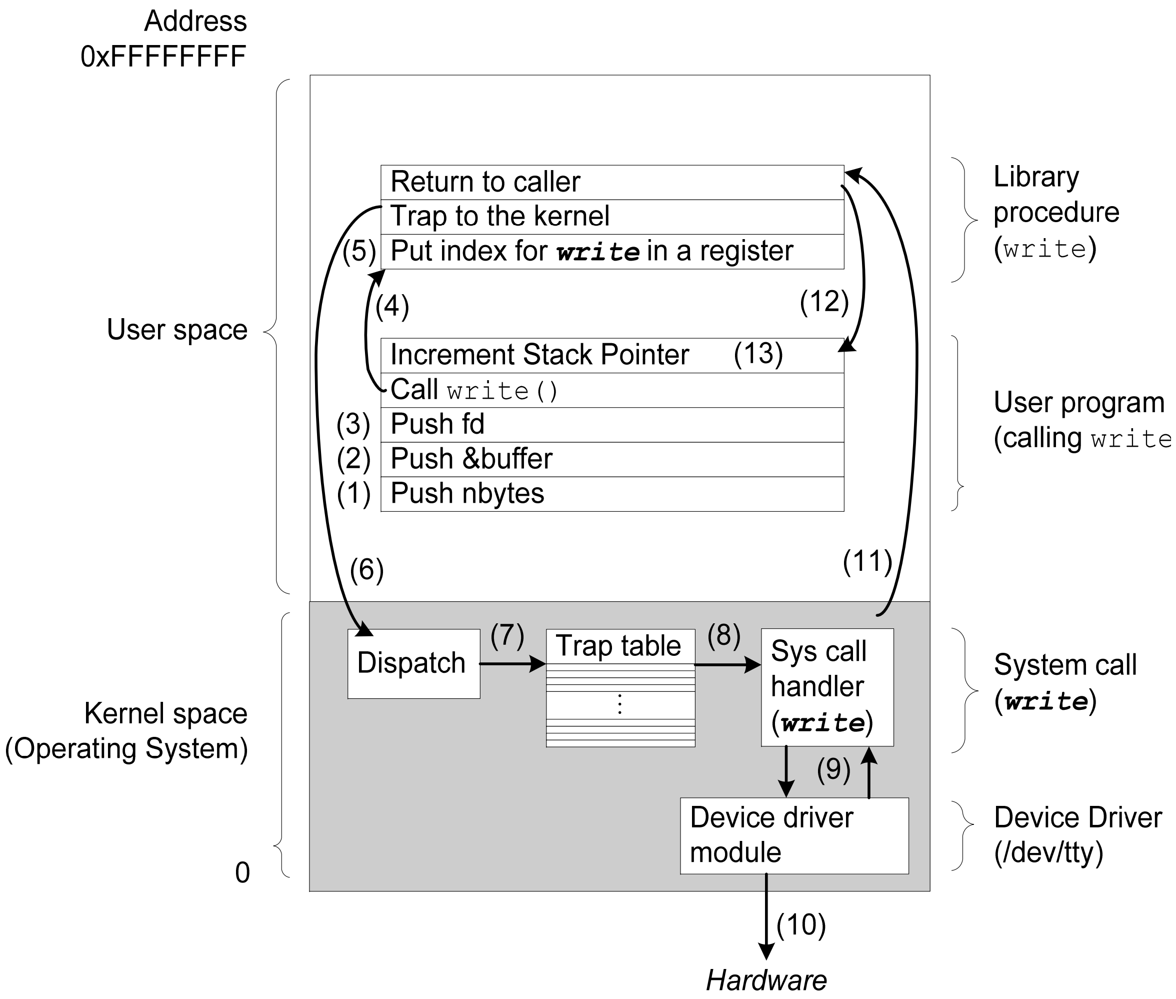
the code for the system call **write** retrieves the parameters from the stack and jumps to the appropriate device driver code for the device being written to

step 10:

the device driver code sends the output string (bytes) to the hardware device

Making a System Call (Unix) -11

Looking at this operation in detail….



step 9:

the code for the system call **write** retrieves the parameters from the stack and jumps to the appropriate device driver code for the device being written to

step 10:

the device driver code sends the output string (bytes) to the hardware device

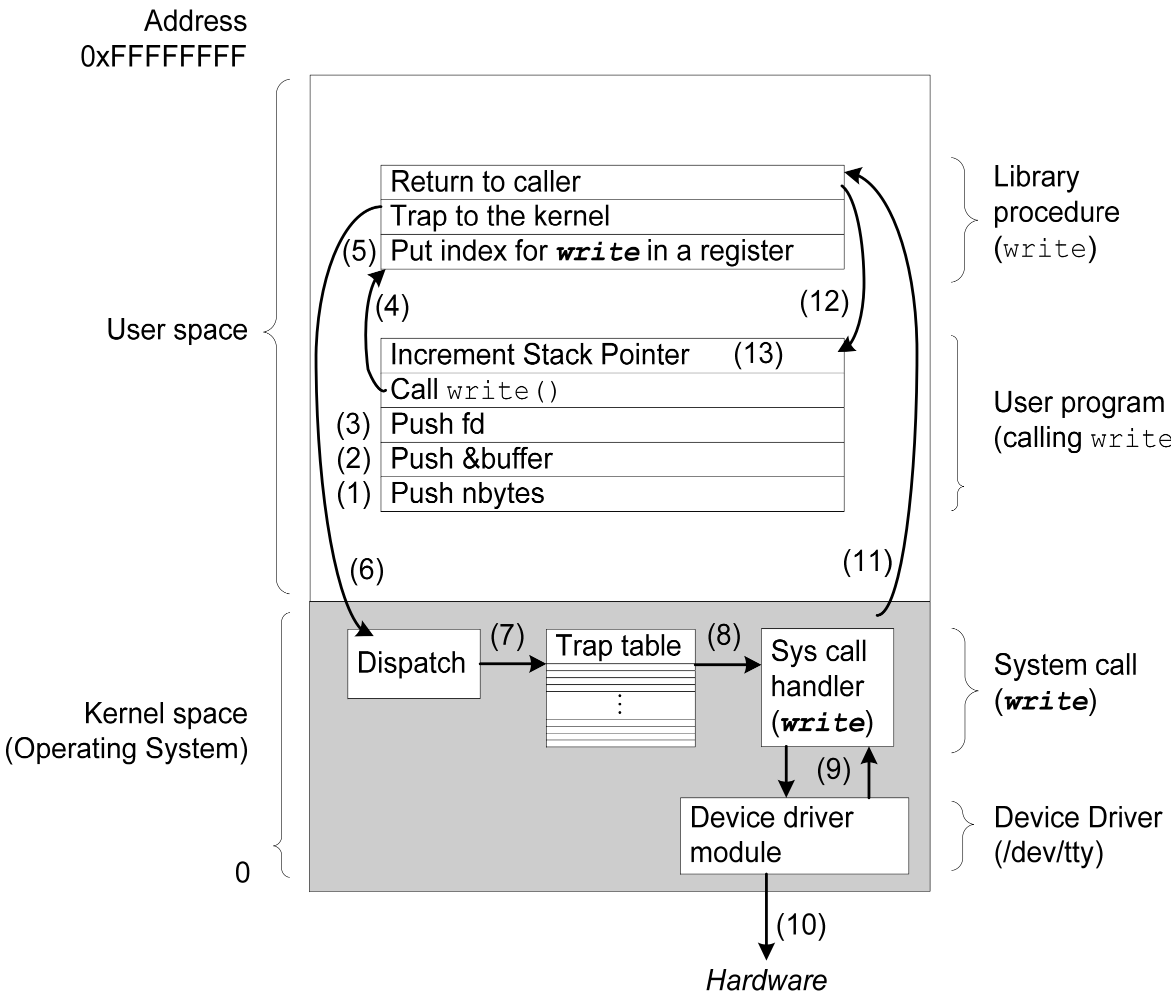
step 11:

when the system call is finished:

- the processor is switched back into user mode
- control is passed back to the library procedure
- execution continues from the instruction immediately following the TRAP instruction in write(2)

Making a System Call (Unix) -12

Looking at this operation in detail….



step 11:

when the system call is finished:

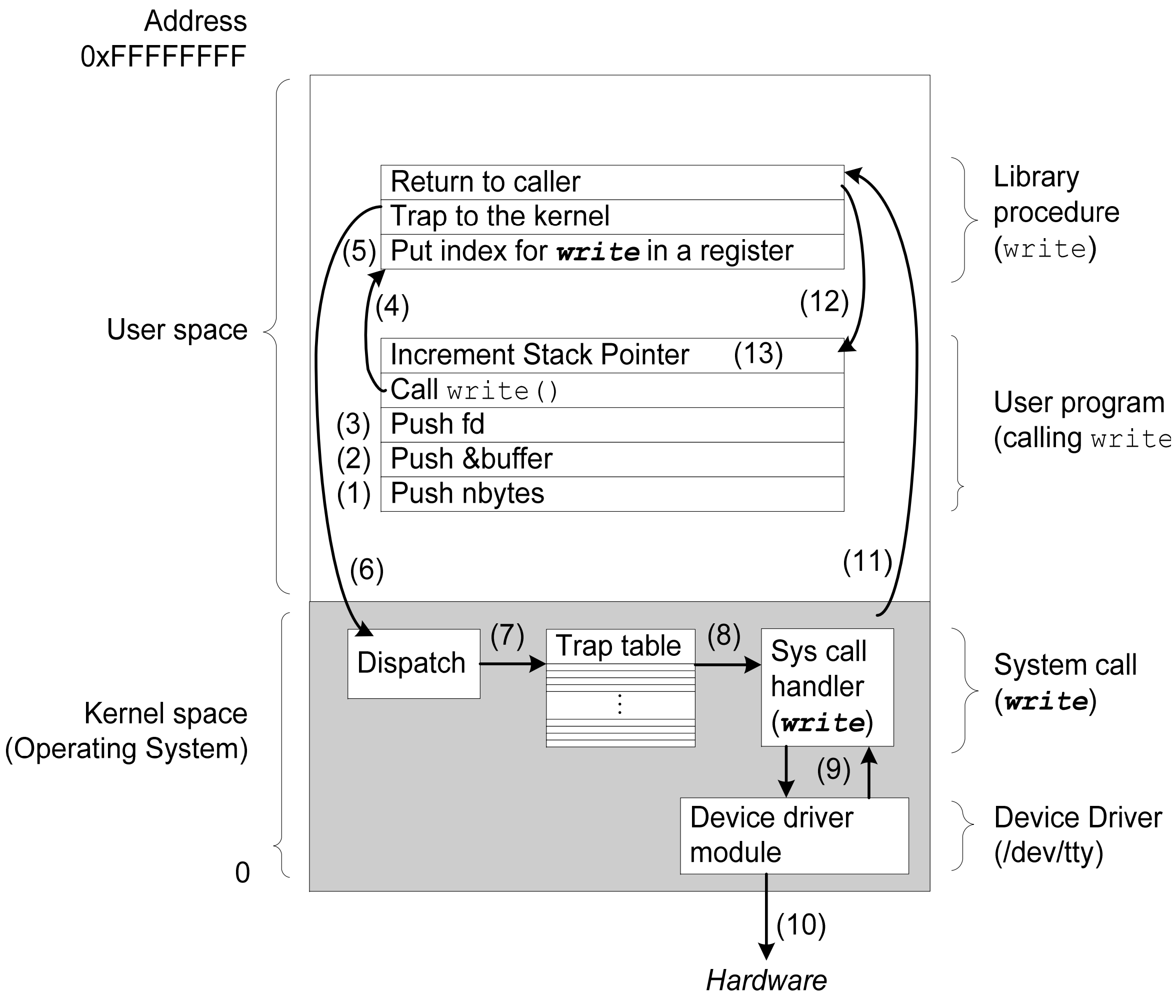
- the processor is switched back into user mode
- control is passed back to the library procedure
- execution continues from the instruction immediately following the TRAP instruction in write(2)

step 12:

the library procedure **`write`** returns to the user program that called it (in our example, **`write`** would return to **`printf`** which would return to **`a.out`**)

Making a System Call (Unix) -13

Looking at this operation in detail….



step 11:

when the system call is finished:

- the processor is switched back into user mode
- control is passed back to the library procedure,
- execution continues from the instruction immediately following the TRAP instruction in write(2)

step 12:

the library procedure **`write`** returns to the user program that called it (in our example, **`write`** would return to **`printf`** which would return to **`a.out`**)

step 13:

to finish, the user program has to clean up the stack, just as it does after any procedure call

System Calls and Multi-processing

Notes for the preceding example:

1. the processor can at any time switch and run a different process
   - this could happen when the process is in user mode or when it is in kernel mode
   - if this happens when the system call is blocked on IO, or busy performing IO, other processes will not have access to the device
     - ie: the device will be marked as “busy” and the kernel will not let another process write to it

2. the process may block when it is executing the driver or system call kernel code
   - in this case the OS will pass control to another process that is waiting to run
   - an example of when a system call might block is a read(), such as from a keyboard – when it is waiting for you to type something

Processor Modes

- most processors allow for a variety of ***processor modes***
- processor modes give different levels of access to the CPU instructions
- processor mode is controlled by a ***mode bit*** (2 bits?) in the PSW register that can be set only by the OS
- the OS keeps track of which processes are running in which mode, and sets the bit as required when
  - the process context is switched
  - a process attempts to execute kernel code, such as when it invokes a system call via the TRAP instruction
- current OS's use two modes
  - ***user mode***: the CPU runs in user mode when executing non-kernel code
  - ***supervisor mode***: the CPU runs in supervisor mode when kernel code is executing

User Space

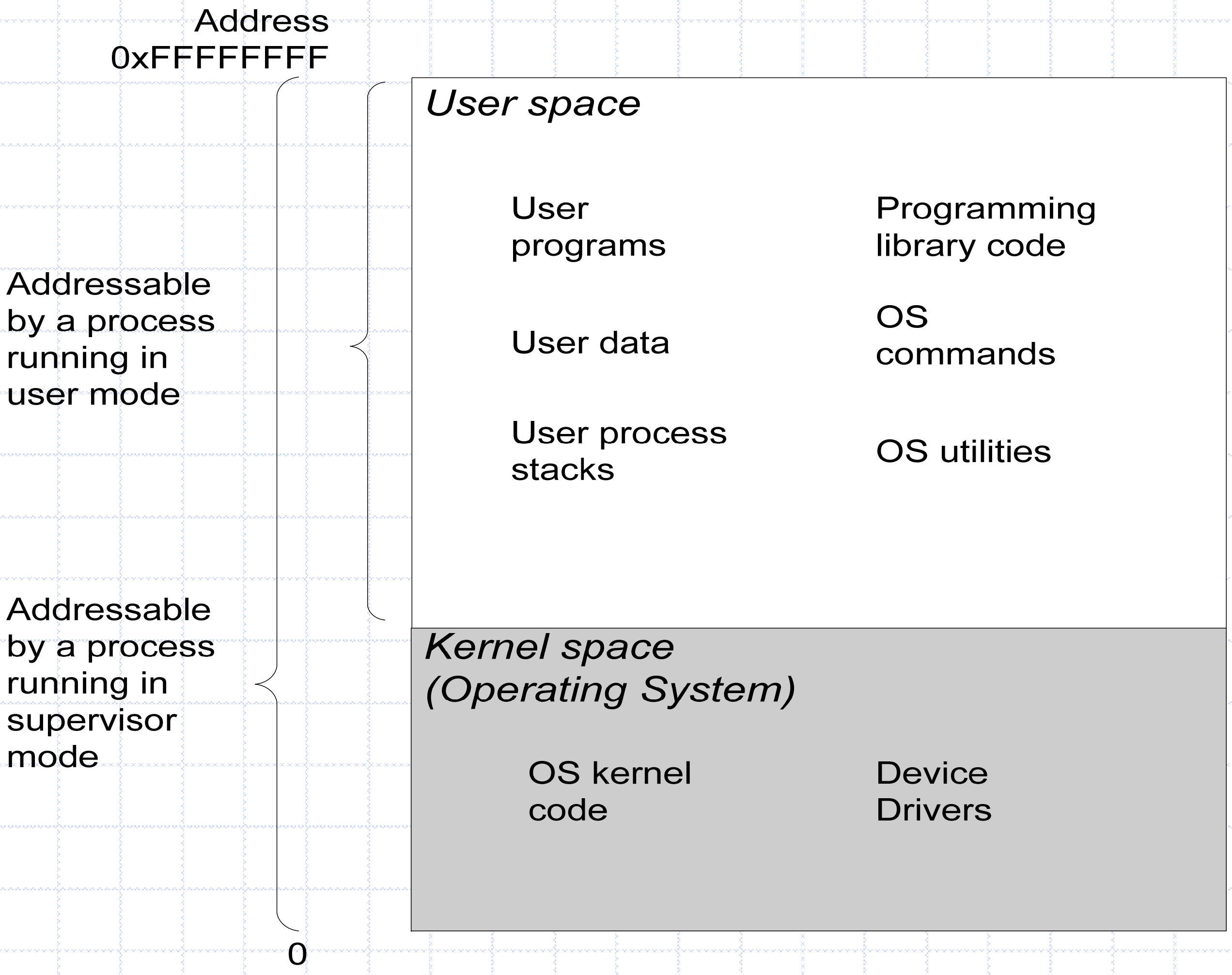
- recall in the System Call example we mentioned that the process changed from User mode to Kernel mode when the TRAP instruction was executed
  - *this was done via the setting of the processor mode bit*
- prior to the setting of this bit, we say that the process was executing in user space, which means:
  - all references were to a portion of memory reserved for user programs
  - only instructions and addresses in this user accessible memory can be referenced
    - this means that kernel code cannot be run when in user mode, as it is loaded in a different area of memory, known as kernel space (or system space)

Kernel Space

- when a process is running in kernel mode, it can
  - access any kernel space memory
    - which is where OS code, device drivers etc are stored
  - write to kernel space memory
    - which means it can stomp all over the OS code

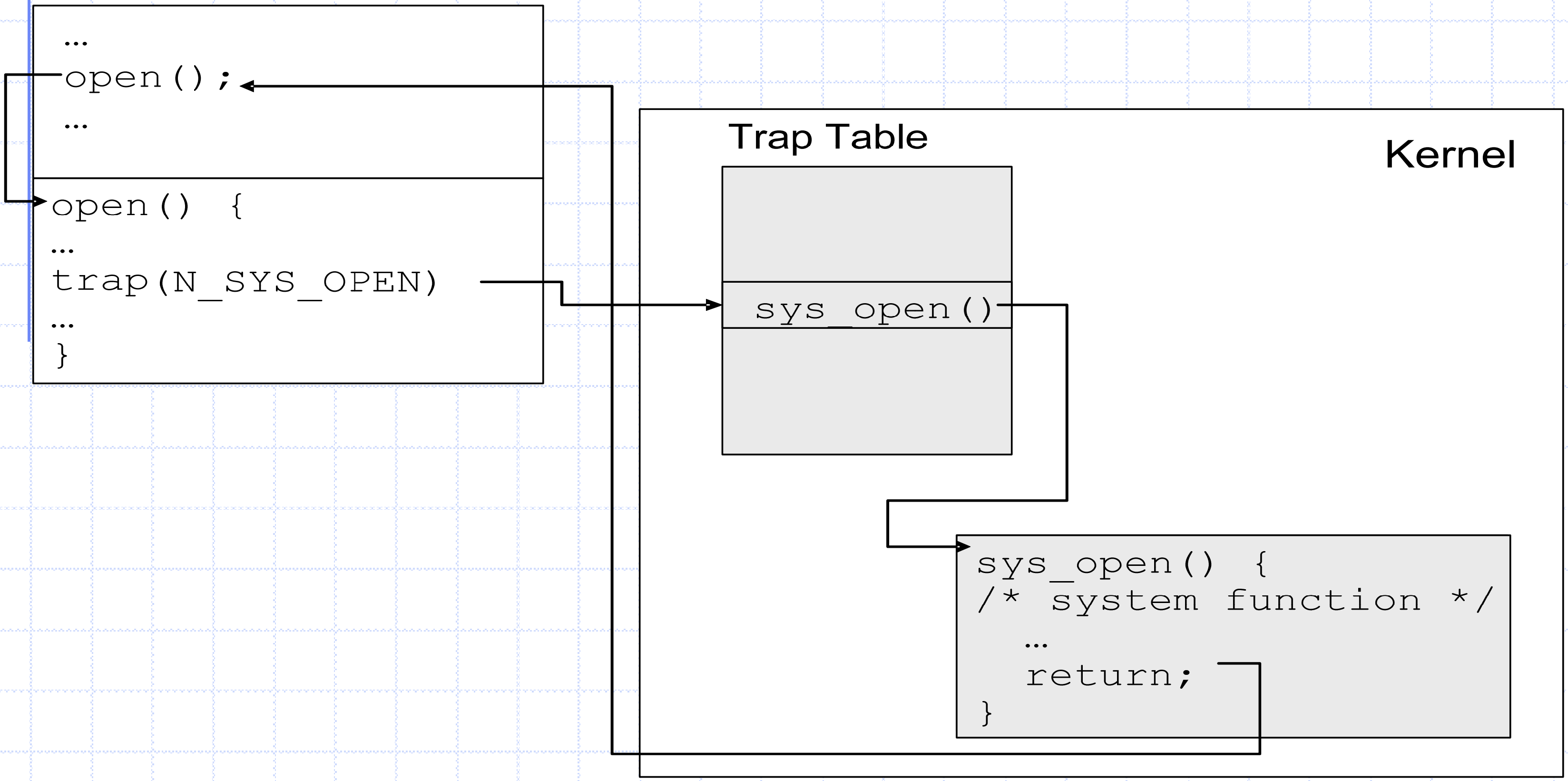
- since we don’t want user programs to mess with kernel memory, we make the following rules (which are enforced by the OS)
  - do not allow any user code to be executed while the processor is in supervisor mode
    - only instructions from the kernel memory space will be executed
  - do not allow any kernel instructions to be executed while in user mode
  - do not allow user mode programs to reference kernel memory

Address Space



TRAP Instruction

- TRAP can be thought of as procedure calls via a jump table
  - the kernel stays protected because the OS will only jump to registered system call addresses

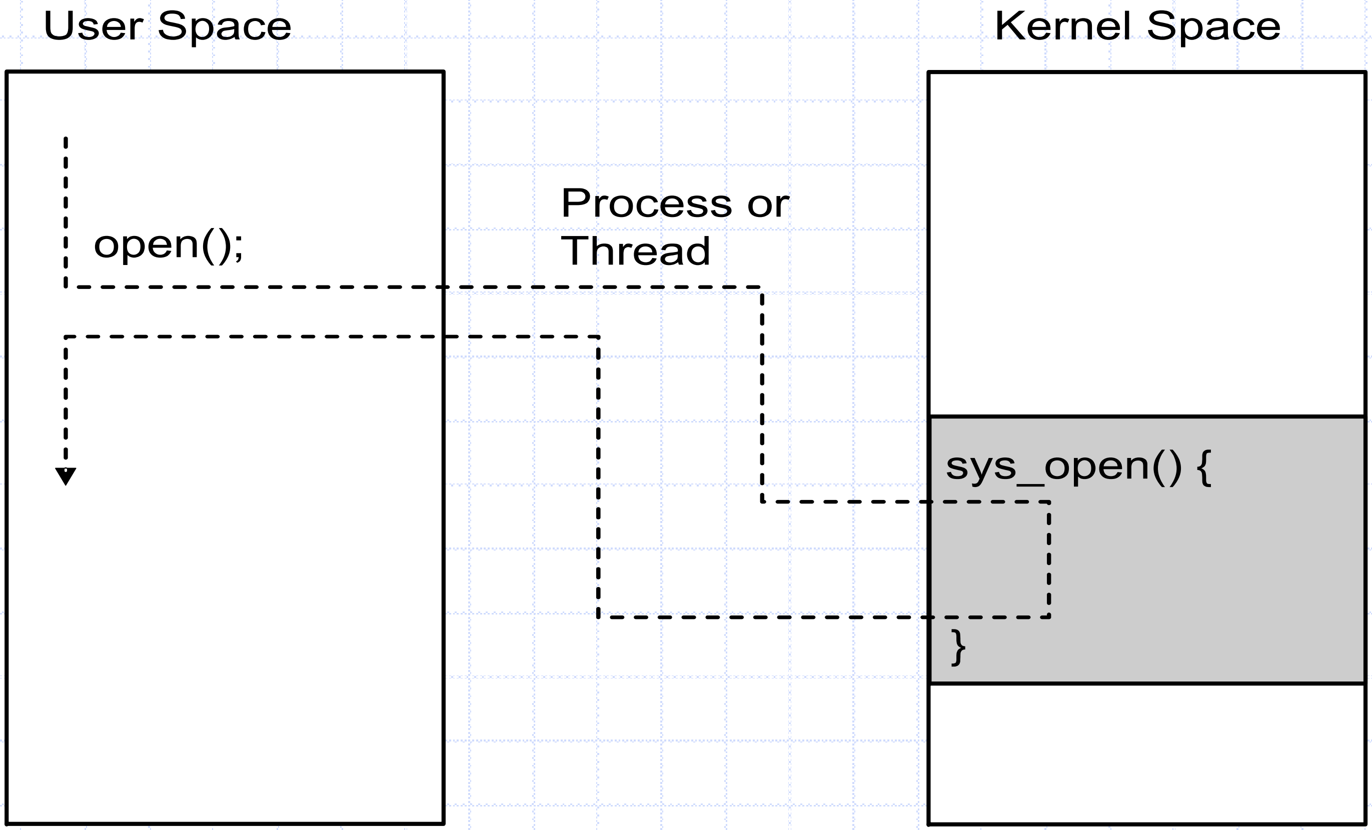


```
…
open();
…
```

```
open() {
…
trap(N_SYS_OPEN)
…
}
```

```
sys_open() {
/* system function */
   …
   return;
}
```

Process Executing a System Call



OS Modularization

OS Modularization

Design trade-off: extensibility vs efficiency

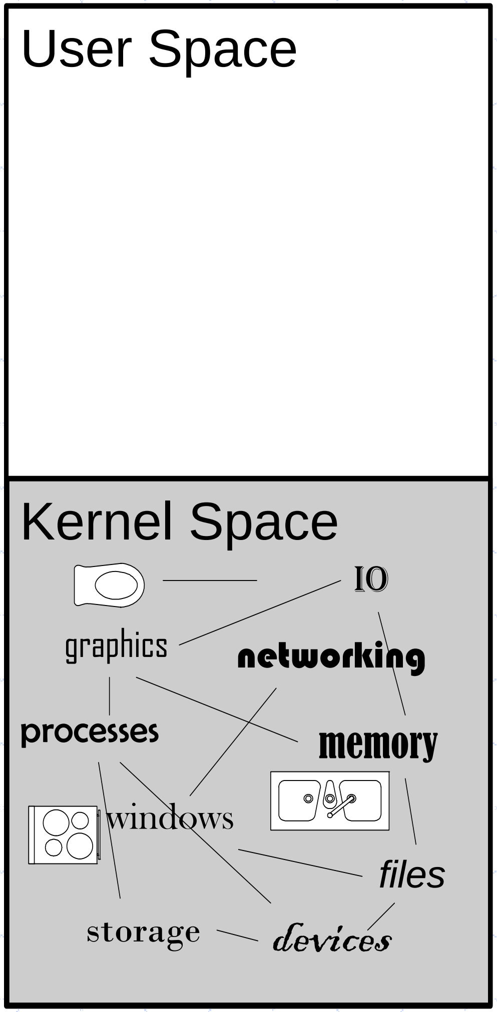
- Extensibility:
  - the more modular it is, the easier it is to add/change functionality
- Efficiency:
  - modularity implies interfaces
  - interfaces necessitate function calls, context switching etc (you should read this as "overhead")
- *Commercial OS offerings are biased towards adding changing functionality*
- *Experimental or research OS's are biased towards performance (and simplicity)*

OS Design Strategies

- There are two popular OS design strategies for OS kernels ...
  - monolithic
    - put everything you need in the kernel
  - micro-kernel
    - put only stuff that is necessary in the kernel

Monolithic kernels

- *all (or most) OS functionality is combined and run in the kernel*
  - leads to ***very complicated*** intertwined code
  - as functionality is added, the ***kernel gets larger*** and slower
  - to add something (eg: driver) you typically ***recompile and link*** the entire kernel
  - most variations of unix and linux employ a monolithic kernel design

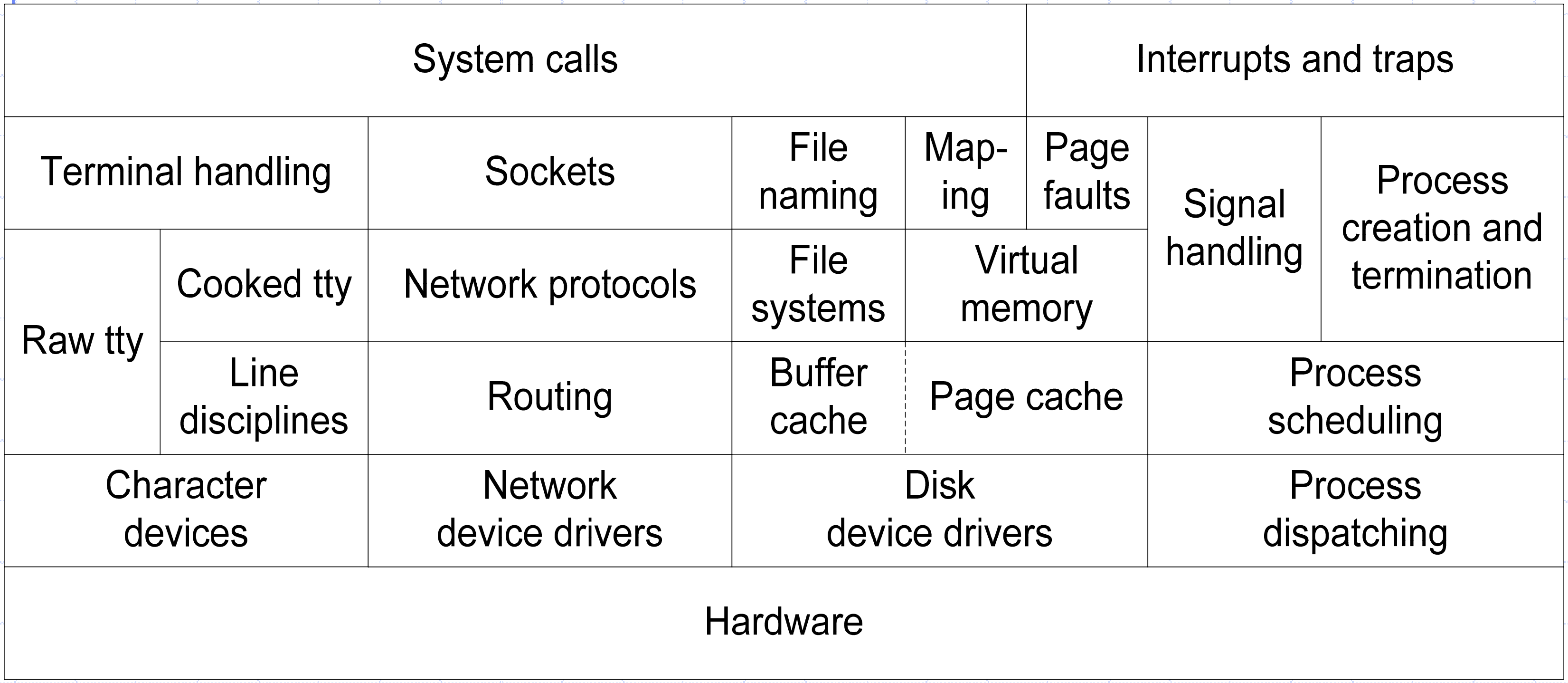


Structure of Monolithic Kernels

- ***A main program*** that invokes the requested service procedure
- ***A set of service procedures*** that carry out the system calls
- ***A set of utility procedures*** that help the service procedures

... of course this is grossly over-simplified ...

Typical Unix OS Organization



System calls

Interrupts and traps

Terminal handling

Sockets

File naming

Map-ing

Page faults

Signal handling

Process creation and termination

Raw tty

Cooked tty

Network protocols

File systems

Virtual memory

Line disciplines

Routing

Buffer cache

Page cache

Process scheduling

Character devices

Network device drivers

Disk device drivers

Process dispatching

Hardware

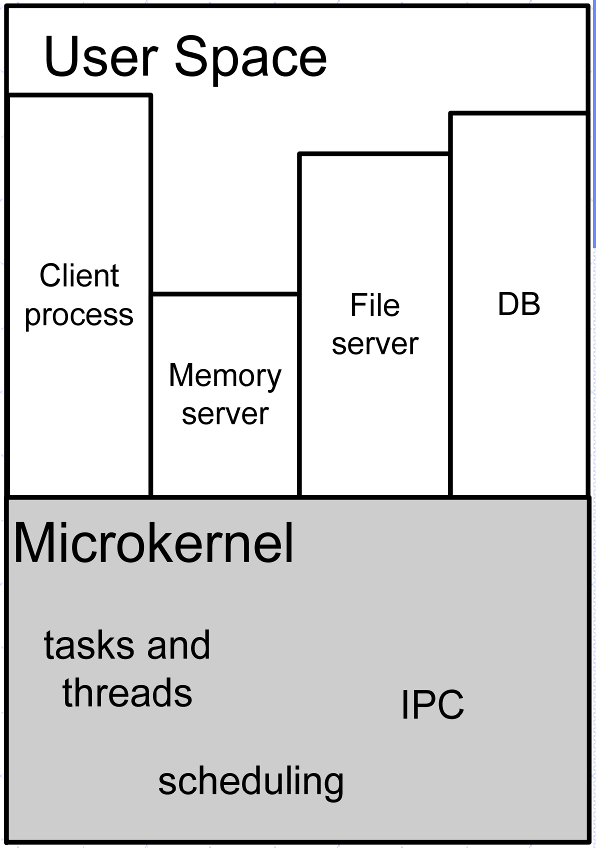
Structure of the 4.4BSD Kernel

Micro-kernels

- *try to keep the kernel as small as possible*
  - only ***basic core services*** are provided by the ***kernel***
    - eg: process and memory management
  - anything that can be ***run in user space*** is run in user space
  - typically implemented using ***messaging*** (client/server) interface between user level OS processes and kernel

Problem:

- *requires extra context switching or function calls, which slows things down*



The Minix OS Organization (micro-kernel)

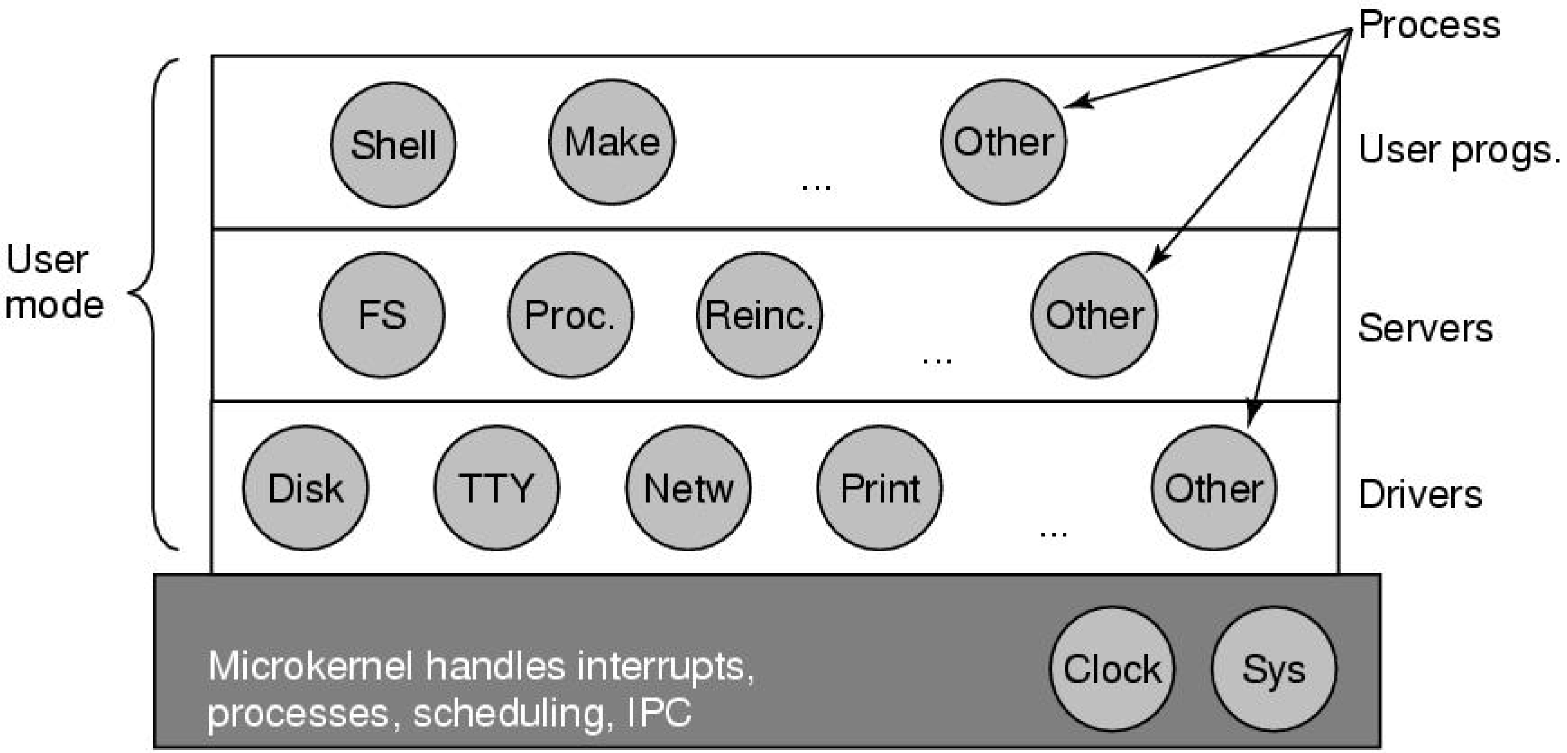
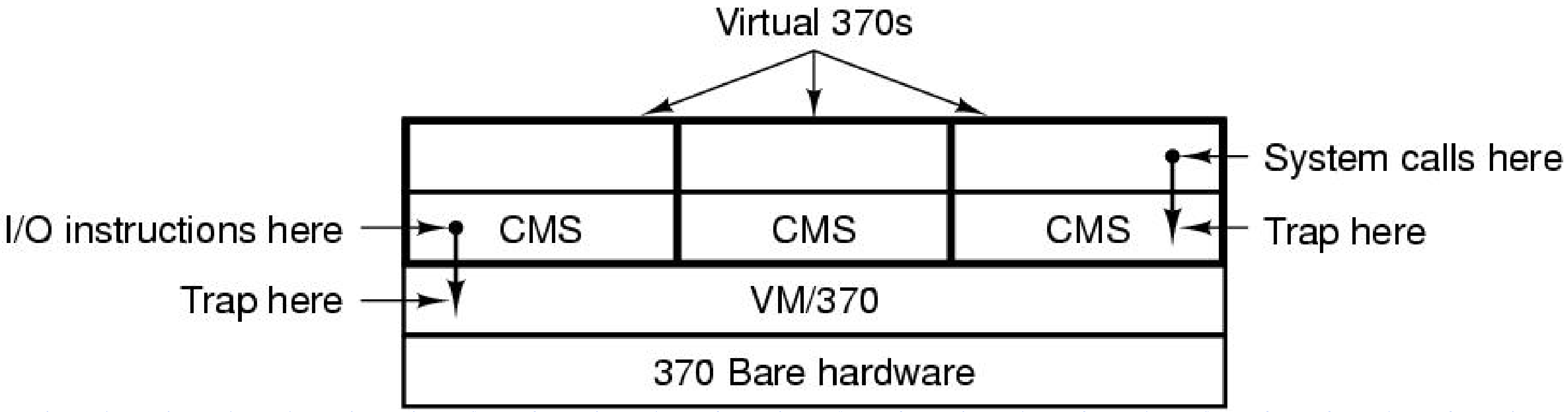


Figure 1-26. Structure of the MINIX 3 system

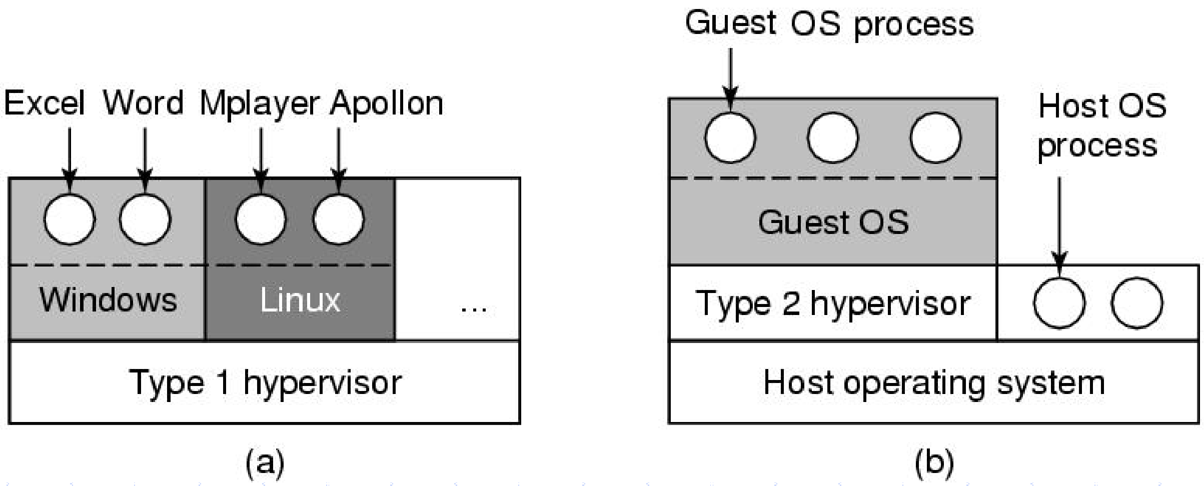
Note: drivers are in user-space - but they need to interact with devices (I/O) via the kernel

The Virtual Machine Concept (VM/CMS)



- developed by IBM in late 1970's
- idea is to duplicate the hardware in software, and create multiple instances
- each instance can run any OS it wants - even different OS's because the copy of the hardware is complete (all registers, processor modes, etc)
- allowed different OS's to execute at the same time on the same machine

Modern Virtual Machines



- Type 1: the hardware emulator (hypervisor) runs directly on the hardware
  - requires specific hardware support such as the ability for the higher level OS code to execute privileged instructions
  - very expensive to develop as it needs to be done for each hw platform
- Type 2: (used by VMware) - the hypervisor runs on top of an existing OS, allowing the existing OS to abstract the hardware
  - system calls from the guest OS are replaced (in the binary OS code) with corresponding hypervisor calls

This is the end for now ...