Device driver & Kernel models

- Device driver
- Kernel models
- Answer the Guide Question 2

- Device driver

Device

- a hardware to enhance the capability of computers
  - keyboard
  - network adapter
  - hard disk
- To make good use of them, some way must exist to control them

Direct access of device

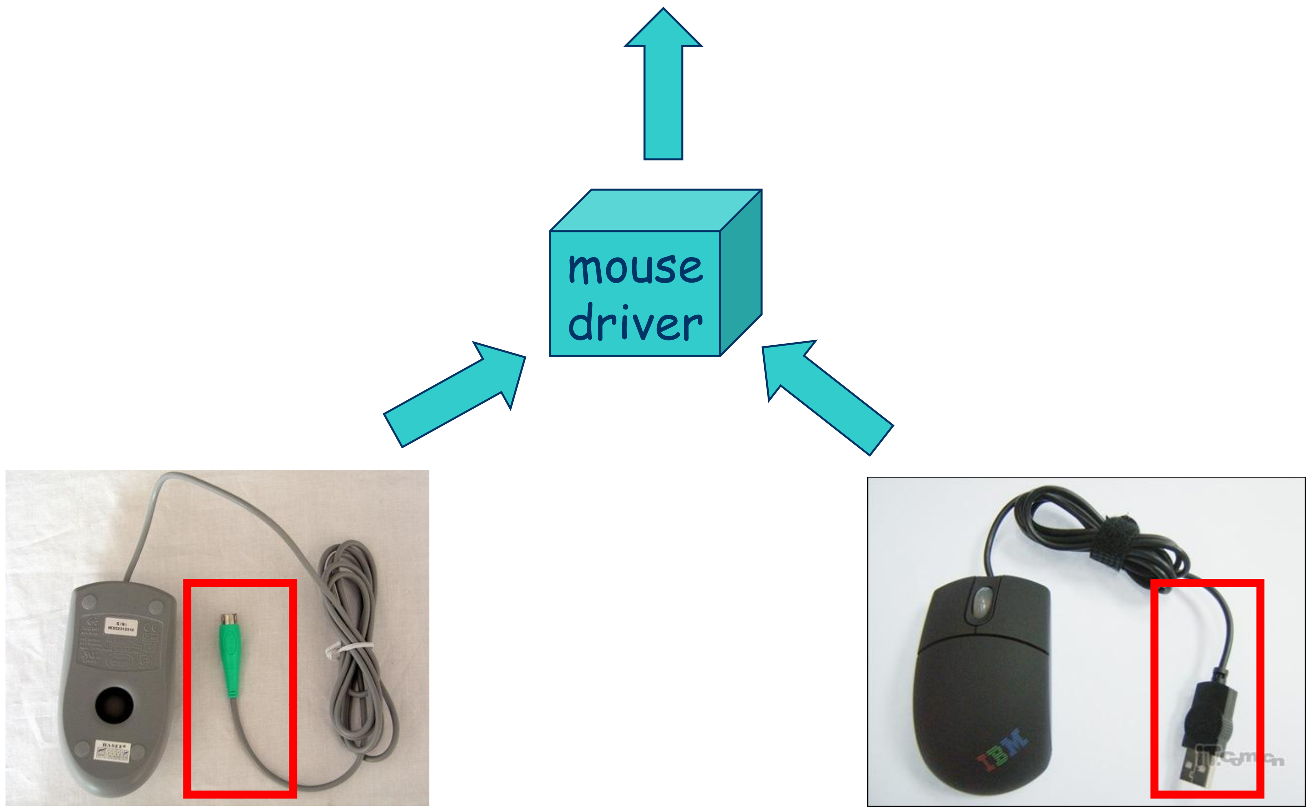
- use “magic numbers”
  - “ugly” interface
  - tedious

outb $0x21, $0x11

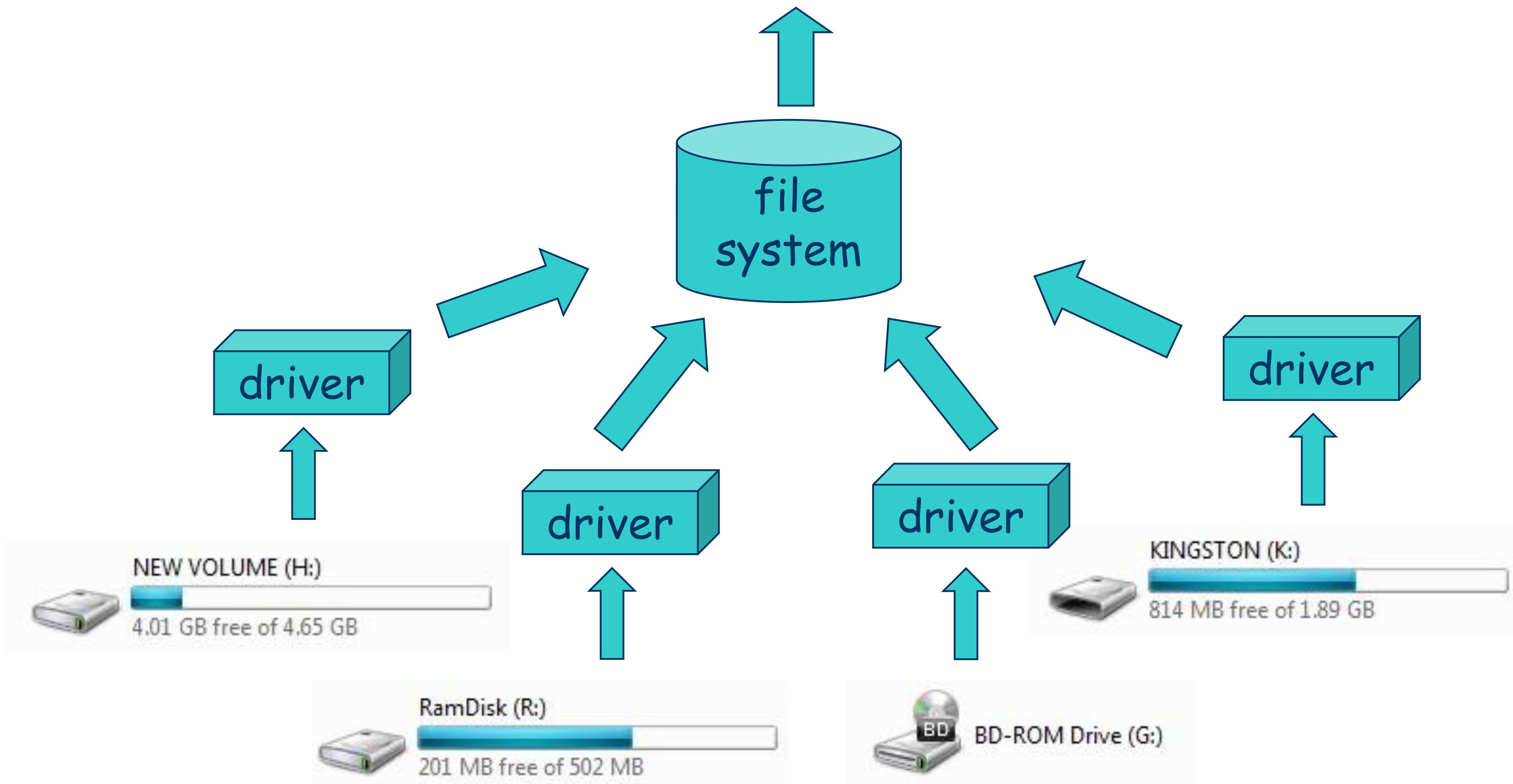
Device driver

- a program to access the device directly
- an abstraction of the device
  - shade hardware details
    - physical interface
    - data transfer
    - …
  - provide beautiful interface

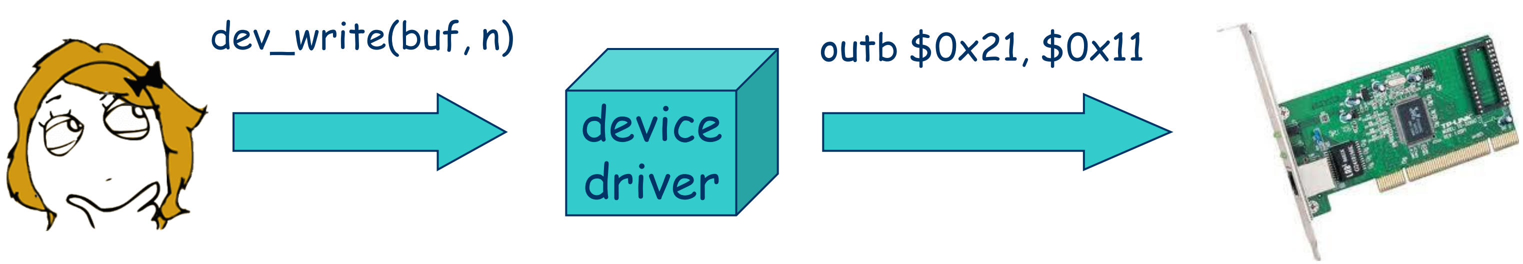
Shade hardware details



Shade hardware details (cont.)



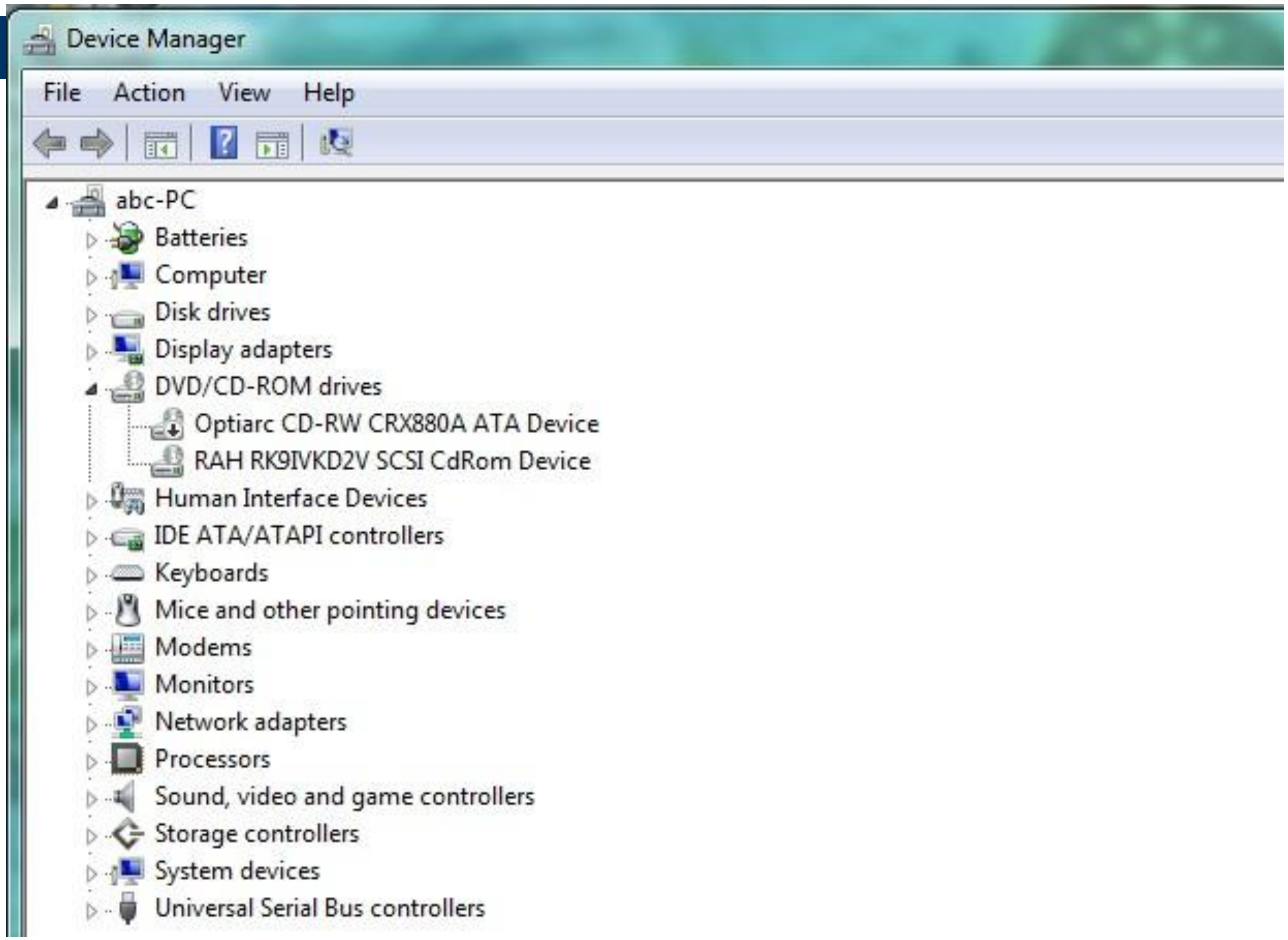
Beautiful interface



Management

- user process should not communicate with drivers directly
  - malicious programs may do something harmful
- OS should manage different drivers
  - identify them
  - watch their working states
  - provide more beautiful interface to user process
  - restart them if they crash
  - …

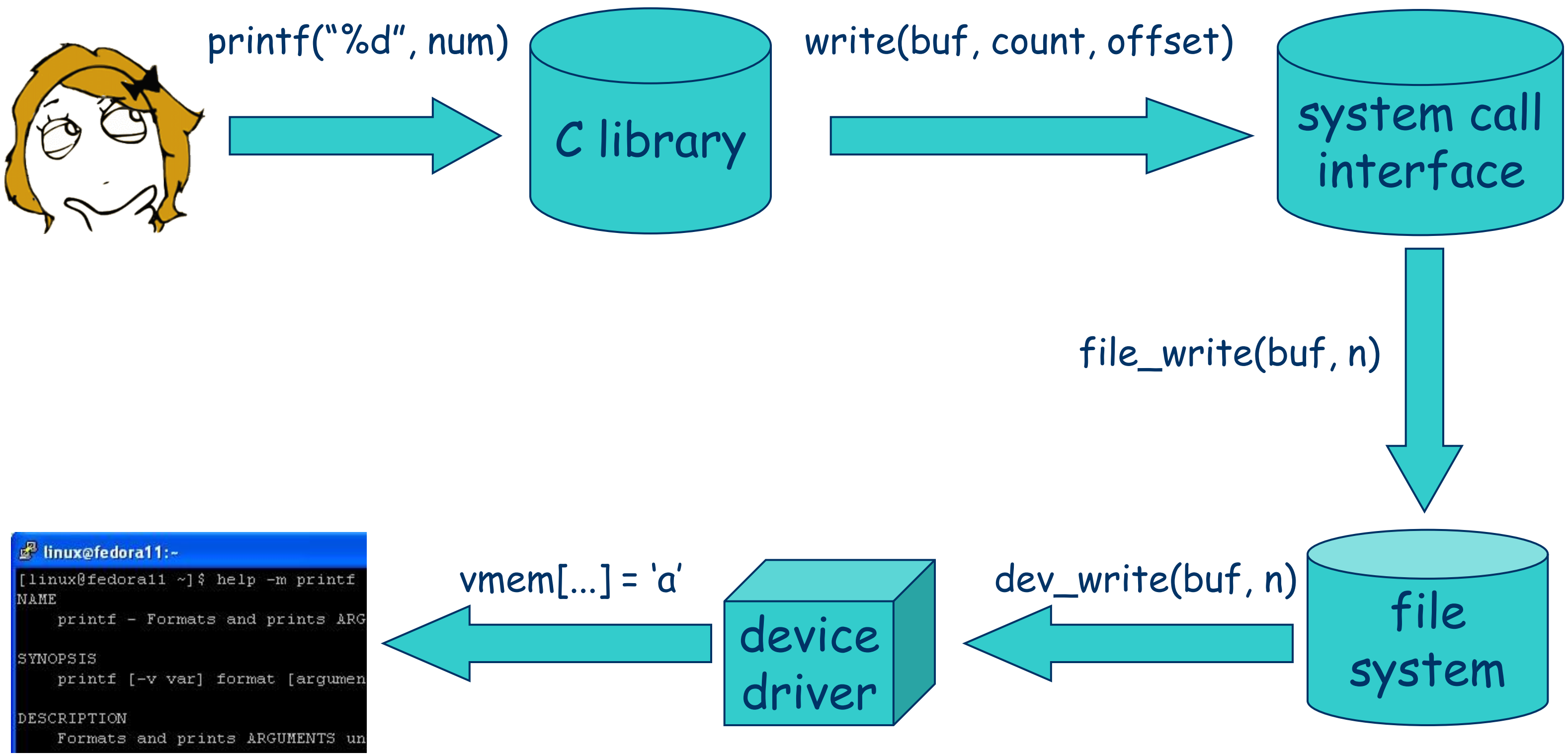
Management (cont.)



Hardware abstraction layer

- provide unified interfaces to communicate with drivers
  - device register
  - dev_read() & dev_write()
- very easy to use
  - dev_write("hda", reqst_pid, buf, offset, size);
  - Why reqst_pid?

More beautiful interface



Virtual device

- not associated with any physical device
  - null, zero, random
  - HAL makes them transparent to upper layers

```
[528][1: ~]$ head -c 10 < /dev/random
�IO��▯���[529][1: ~]$ printf "Hello world\n"
Hello world
[530][1: ~]$ printf "Hello world\n" > /dev/null
[531][1: ~]$ █
```

Device drivers in Nanos

- Timer
  - rtc, intel 8253 pit
- TTY
  - terminal, input from keyboard, output to screen
- IDE
  - disk read/write
- in Nanos, they all run as kernel threads
  - communicate by message passing

Device drivers in Nanos (cont.)

- initialize
  - register interrupt handler
  - register device
  - initialize other components
- work like a server
  - receive request
  - process request
  - send reply

```
while (TRUE) {
    receive(ANY, &m);
    if (m.src == MSG_HARD_INTR) {
        ...
    } else {
        switch (m.type) {
            case DEV_READ:
                ...
                break;
            case DEV_WRITE:
                ...
                break;
        }
    }
}
```

Interrupt handler in Nanos

- top half
  - interrupt was wrapped into a message
  - notify the coming of interrupt to driver
  - goal: return from interrupt as soon as possible
    - why?

```
void
send_keymsg(void) {
    Msg m;
    m.type = MSG_TTY_GETKEY;
    m.src = MSG_HARD_INTR;
    send(TTY, &m);
}
```

Interrupt handler in Nanos (cont.)

- bottom half
  - handle the interrupt
  - read/write the hardware
  - perform other works
  - send reply

```
while (TRUE) {
    receive(ANY, &m);
    if (m.src == MSG_HARD_INTR) {
        ...
    } else {
        switch (m.type) {
            case DEV_READ:
                ...
                break;
            case DEV_WRITE:
                ...
                break;
        }
    }
}
```

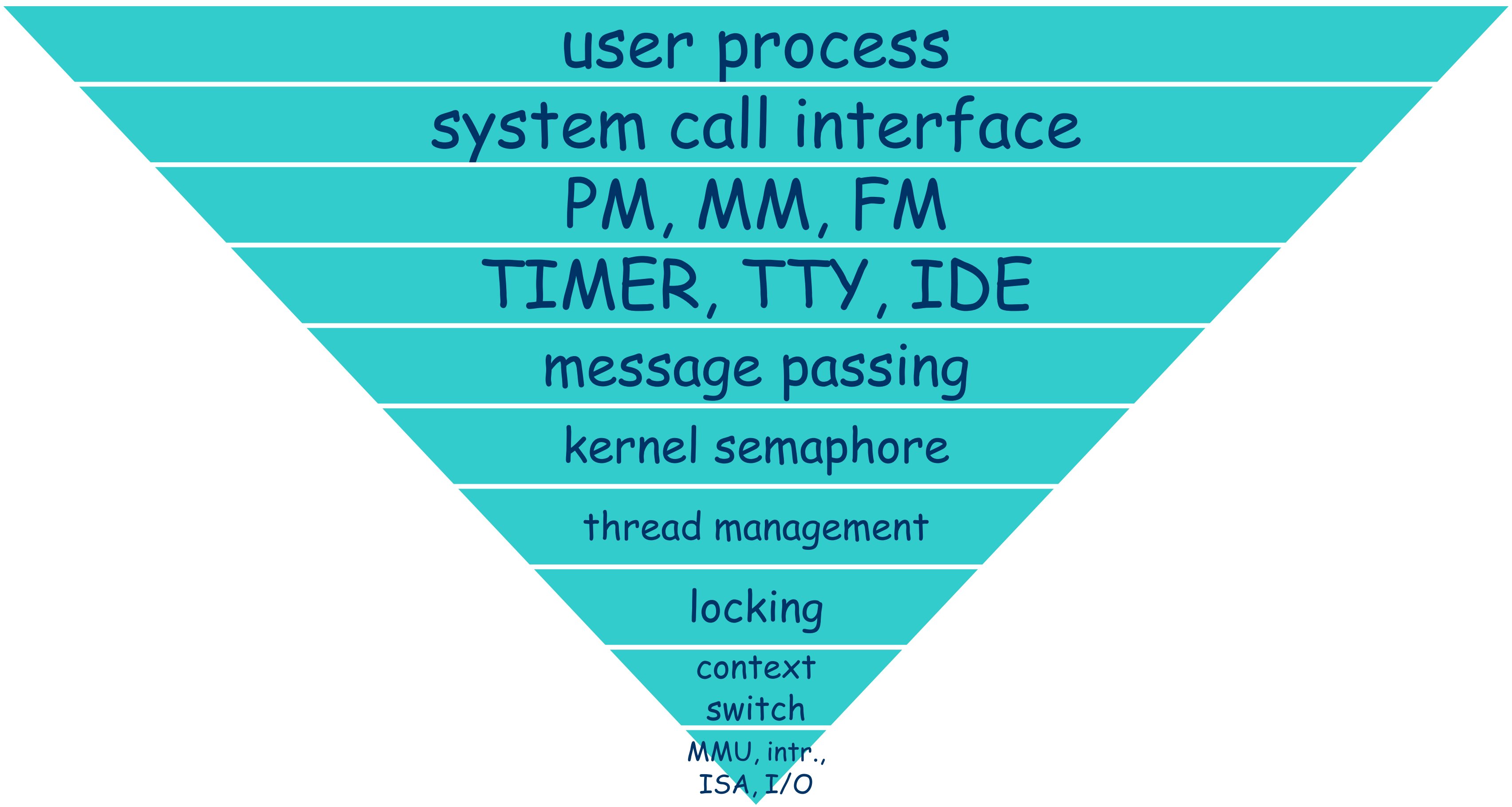
in Nanos, bottom half is performed in driver threads

Read the code for more details

- You must make it clear how to communicate with drivers.
- It is easy to understand how the drivers work, even if you skip the magic numbers.
  - If you are interested with the magic numbers, search them on the Internet.

- Kernel models

Where are we?


user process
system call interface
PM, MM, FM
TIMER, TTY, IDE
message passing
kernel semaphore
thread management
locking
context switch
MMU, intr., ISA, I/O

Where are we? (cont.)

- We have already built a kernel!
  - system initialization
  - interrupt management
  - context switching
  - thread management & thread scheduling
  - lockingkernel semaphore
- Depending how remaining components of an OS are organized, kernel models are devided into two types.
  - monolithic kernel
  - micro kernel

- Kernel models
  - monolithic kernel

Monolithic kernel

- All remaining components are intergrated in the kernel.
  - device management
  - device drivers
  - process management
  - memory management
  - file management
  - network management
  - ...

More bugs, less security

- They share the same address space with the whole kernel.
  - use function calls as interfaces
- Potentially, one component can access others' resources.
  - A buggy component may harm others.
  - A malicious component will easily destroy the kernel.

System calls handling

- System calls are directly handled by kernel.
- Take write() as an example
  - write() ➔ int $0x80 ➔ sys_write() ➔ file->f_op->write() ➔ dev_write() ➔ disk_write() ➔ function returns ➔ iret
- For one system call request, only one execution flow is involved.
- Any problem?

Nested interrupts

- In a monolithic kernel system, handling one system call may cost pleaty of time.
- Interrupts should be enabled during handling a system call
  - to garuntee in-time reponse to I/O or other processes
- This causes nested interrupts!
  - write() ➔ int $0x80 ➔ sys_write() ➔ file->f_op->write() ➔ timer interrupt!

Handling nested interrupts

- All accessing to global resources during the interrupt context should be very careful.
  - data race
  - disable interrupts to lock
    - Can semaphore be used? Why?

- Is that all OK?

More critical regions

- interrupts = context switch!
- What if two processes issue write() concurrently?
  - All global resources accessed during handling write() are cirtical regions!
  - lock() and unlock() are everywhere.
- Is that all OK?

SMP

- lock by disabling interrupts → lock by atomic instructions
- But what if a critical region is in an interrupt context?
  - atomic instructions + disabling interrupts
- other locking issues
  - overhead
  - granularity
    - element lock, row lock, table lock, ..., giant lock
  - deadlock

Why use monolithic kernel?

- Efficiency!
- When handling a system call
  - no unnecessary context switch
  - no unnecessary data copying
  - higher concurrency respecting to one system call

- Comparing there features with micro kernel, you will have a better understanding.

- Kernel models
  - micro kernel

Micro kernel

- All remaining components run as seperate processes.
  - device management → device manager
  - device drivers → different driver processes
  - process management → process manager
  - memory management → memory manager
  - file management → file manager
  - network management → network manager
  - ...

C/S architecture

- process = client / server
- message = request / reply
- message passing = client / server interface

- C/S architecture is conceptual.
  - For the view of kernel, they are all processes.
  - But this helps you to modularize the design.

Less bugs, more security

- They have saperate address spaces.
  - use message passing as interfaces
- One component cannot access others' resources.
  - A buggy component will not harm others.
    - They will be restarted by kernel.
  - A malicious component will not destroy the kernel.
  - Illegal operations are captured by CPU exceptions.

System calls handling

- System calls are handled by different processes.
- Take write() as an example
  - process A: write() ➔ int $0x80 ➔ sys_write() ➔ send(FM, m) ➔ receive(FM, m) ➔ sleep()
  - FM: receive(ANY, m) ➔ dev_write() ➔ send(IDE, m) ➔ receive(IDE, m) ➔ sleep()
  - IDE: receive(ANY, m) ➔ disk_write() ➔send(FM, ok)
  - FM: send(A, ok)
  - process A: iret
- For one system call request, several execution flows are involved.
- Any problem?

Cross-process data copying

- The address of buffer given by process A is invalid to IDE.
  - Why?
- How to solve?
  - Given VAs and the PCBs of two processes, translate VAs into PAs, then copy the data to the right place.
  - Why this sounds?
  - How to implement the address translation?

Decoupling

- receive() = sleep() = context switch
- What if two processes issue write() concurrently?

- Surprisingly, no extra effort is needed to maintain lots of critical regions.
  - This benefits from message passing.
  - Why?

Drawbacks

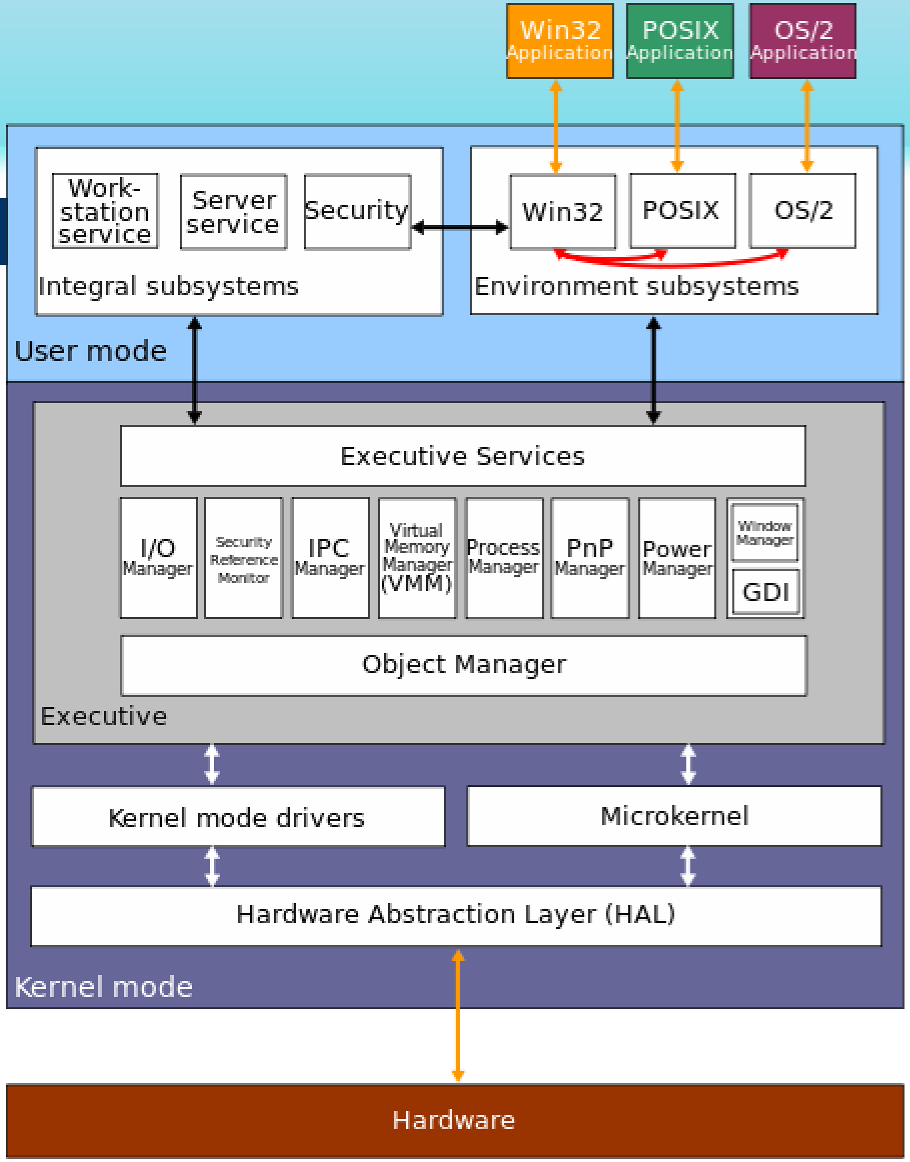
- less efficiency
  - message passing = send() + context switch + receive()
  - more overhead than function calls
- "redundant" data-copying
- less concurrency for a specific system call
  - Different system calls associated with different servers can still be handled concurrently.
- In many-core systems, these features will greatly reduce parallelism.
  - Servers become bottlenecks.

- Kernel models
  - hybrid kernel & Nanos

Hybrid kernel

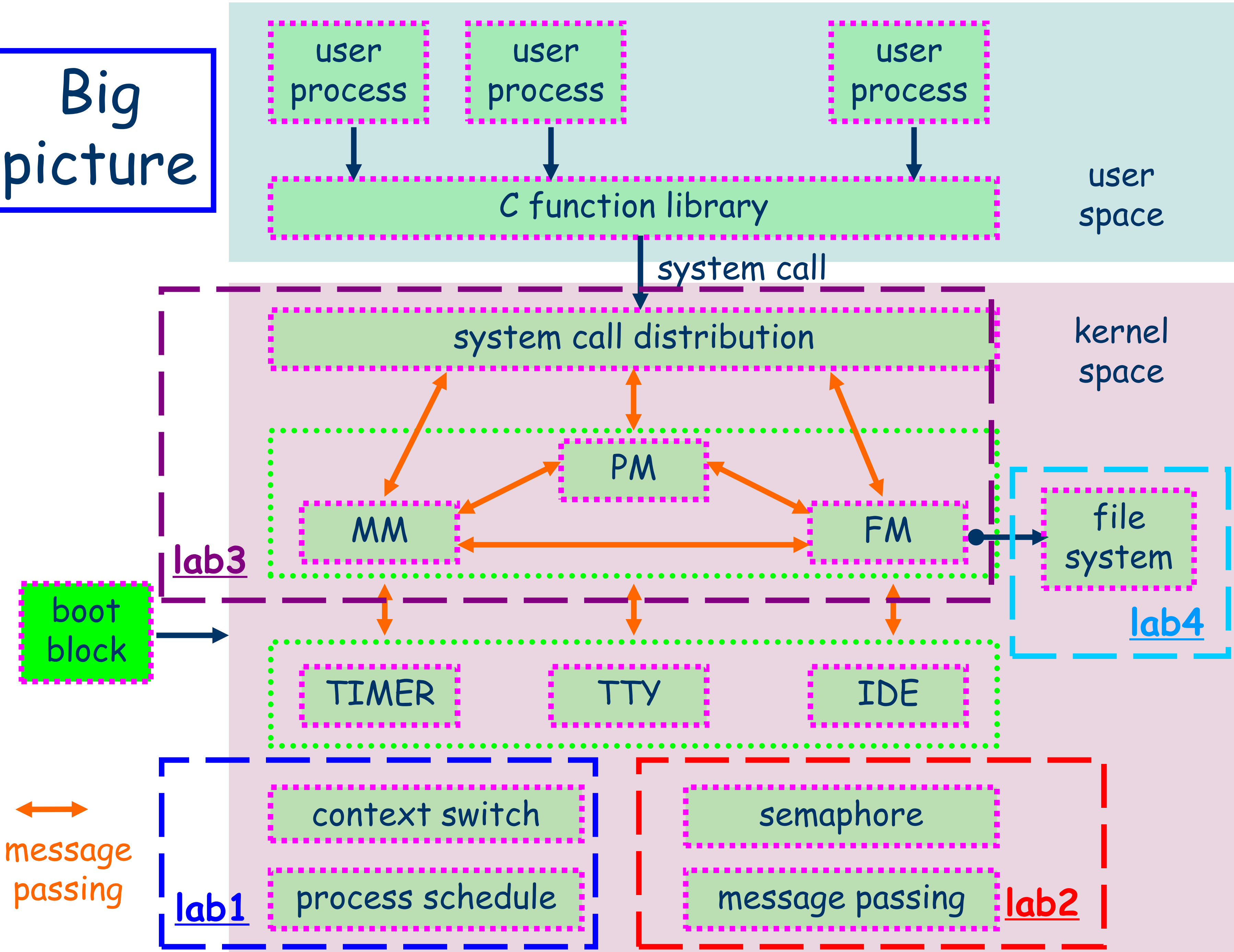
- put drivers and servers back into kernel space
  - they share the same address space with kernel
    - the same as monilithic kernel
    - more easy to debug
  - avoid some redunant message passing
    - drivers can access PCBs directly
- still use message passing as interfaces
  - keep the design clear
- but less security
  - and this is not the issue of OSlab

NT kernel



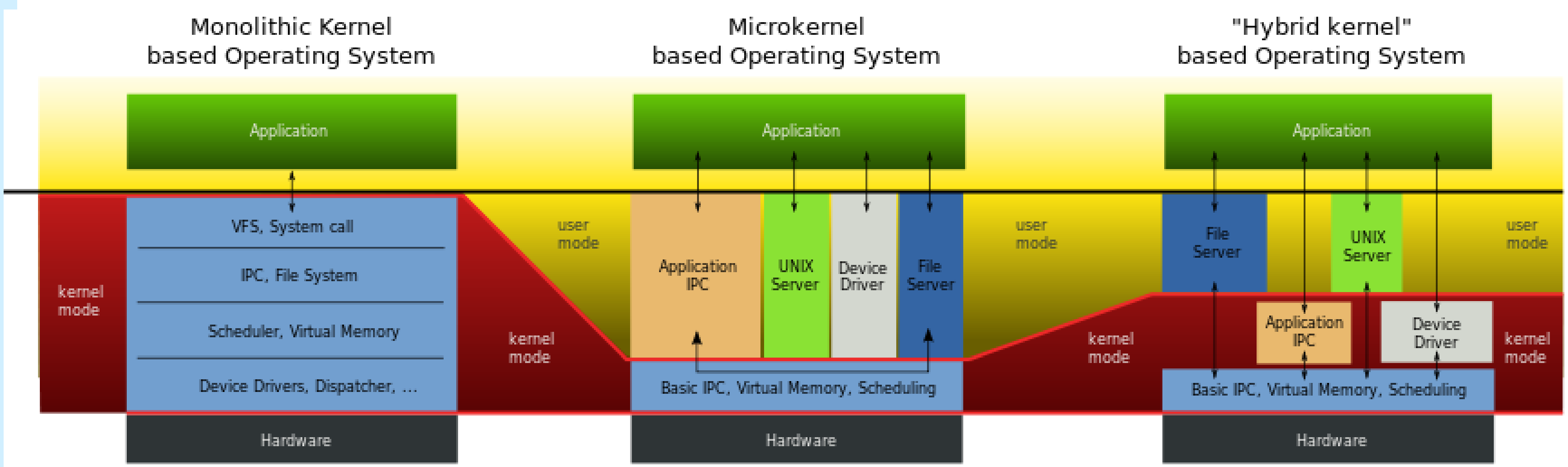
Win32 Application
POSIX Application
OS/2 Application
Work-station service
Server service
Security
Win32
POSIX
OS/2
Integral subsystems
Environment subsystems
User mode
Executive Services
I/O Manager
Security Reference Monitor
IPC Manager
Virtual Memory Manager (VMM)
Process Manager
PnP Manager
Power Manager
Window Manager
GDI
Object Manager
Executive
Kernel mode drivers
Microkernel
Hardware Abstraction Layer (HAL)
Kernel mode
Hardware

Big picture



user process
user process
user process
C function library
user space
system call
system call distribution
kernel space
PM
MM
FM
file system
lab3
lab4
boot block
TIMER
TTY
IDE
context switch
process schedule
lab1
semaphore
message passing
lab2
message passing

Summary
http://en.wikipedia.org/wiki/Monolithic_kernel

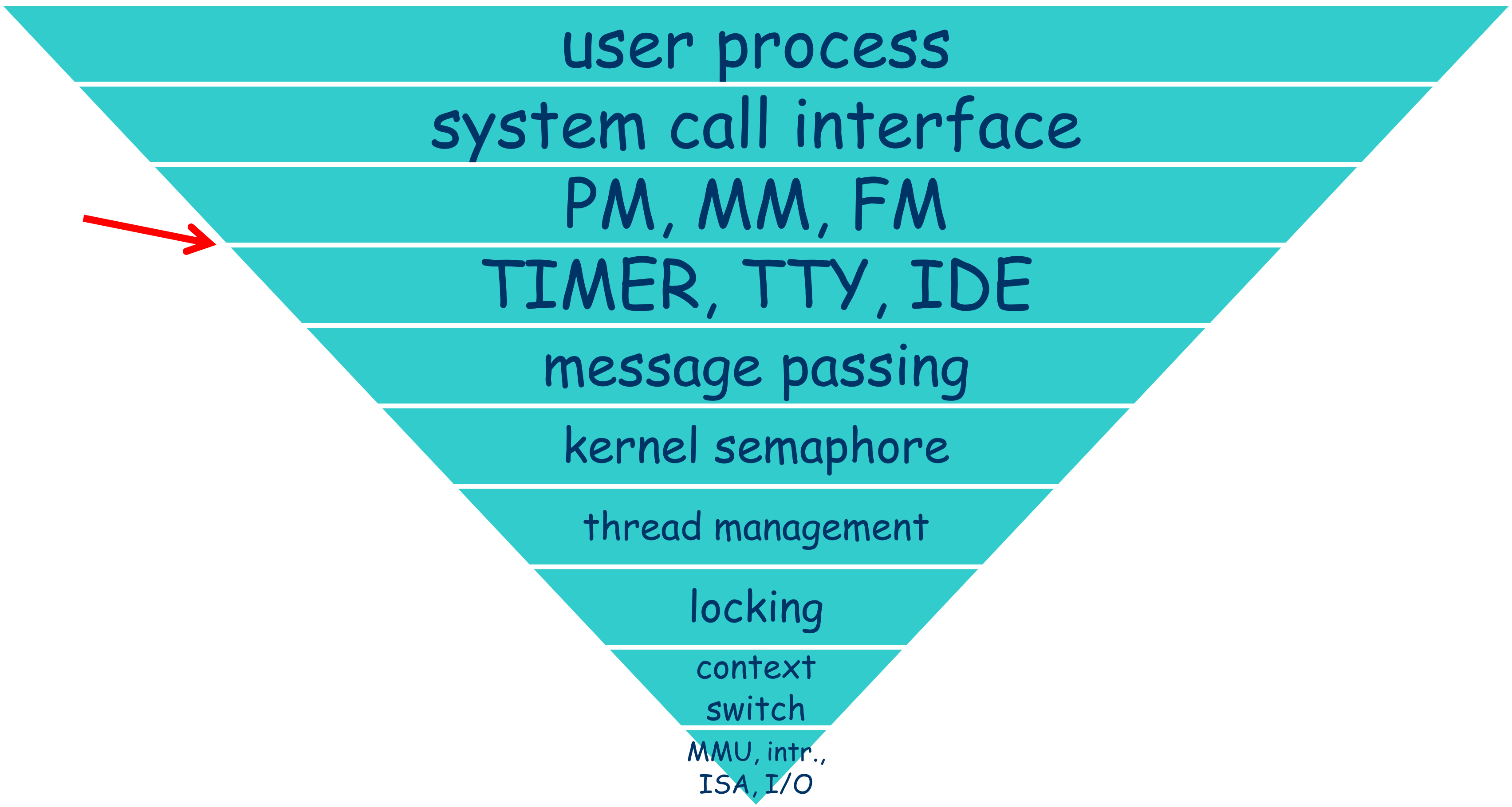


- Answer the Guide Question 2

Guide Question 2

- After you issue command to run a “hello world” program, what happen to the OS exactly?
- Now you can answer some part of this question!

Starting from here



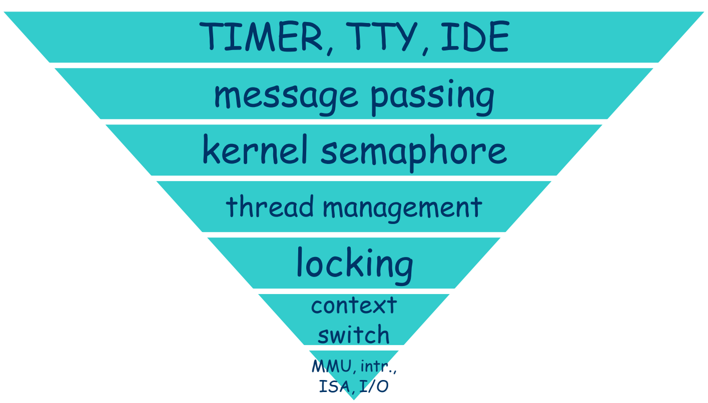
Similar code

```
void simulation() {
        char str[] = “Hello World!\n”;
        dev_write(“tty1”, current->pid, str, 0, strlen(str));
}
```

- Try to answer this:
  - How is this string printed on the screen?

Tracing

- thread A: dev_write("tty1") → send(TTY) →receive(TTY) →P() →sleep() →lock()→int $0x80→context switch
- TTY: iret→unlock()→ret from sleep()→ret from P()→ret from receive(ANY) → vmem[...] = 'H' ... → send(A) → receive(ANY) →P() →sleep() →lock()→int $0x80→context switch
- thread A: iret → unlock() → ret from sleep() → ret from P() → ret from receive(TTY) → ret from dev_write("tty1")



Next

- But you still do not know what happen before dev_write("tty1").
  - How does the "hello world" process come?
  - How is the "Hello World!\n" string transferred from user process to TTY?

- This is solved in Lab3!
  - You will get a decent OS after Lab3!
- Have fun!

劳动节快乐!