238P: Operating Systems

Lecture 1: Introduction

Anton Burtsev
September, 2018

Class details

- Graduate
  - 75 students
- Instructor: Anton Burtsev
- Meeting time: 3:30pm-4:50pm (Tue/Thu)
  - Discussions: 4:00pm-4:50pm (Fri)
    - Regular discussion sections
    - Feel free to stop by my office with questions (DBH 3066)
- 2 TAs
- Web page
  - https://www.ics.uci.edu/~aburtsev/238P/

More details

- 4-5 homeworks
  - Implement a shell
  - Explain whats on the stack
  - Implement a system call
  - Change file system layout
- Midterm
- Final
- Grades are curved
  - Homework: 60%, midterm exam: 15%, final exam: 25% of your grade.
  - You can submit late homework 3 days after the deadline for 60% of your grade

This course

- Inspired by
  - MIT 6.828: Operating System Engineering
    https://pdos.csail.mit.edu/6.828/2016/
  - Adapted for undergraduate students
- We will use xv6
  - Relatively simple OS kernel (only 9K lines of code)
  - Reasonably complete UNIX kernel
  - https://pdos.csail.mit.edu/6.828/2016/xv6.html
- xv6 comes with a book
  - https://pdos.csail.mit.edu/6.828/2016/xv6/book-rev9.pdf
- And source code printout
  - https://pdos.csail.mit.edu/6.828/2016/xv6/xv6-rev9.pdf

Another Book

“Operating Systems: Three Easy Pieces” (OSTEP) Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau

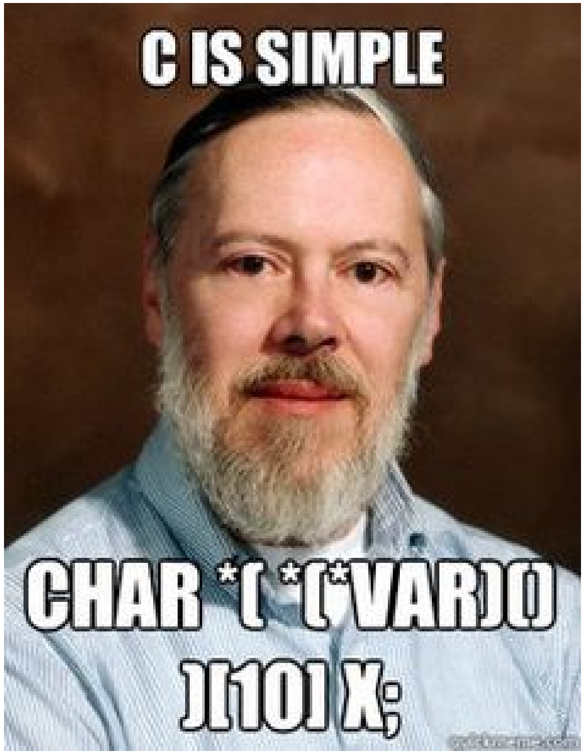
- Free online version http://pages.cs.wisc.edu/~remzi/OSTEP/

Course organization

- Lectures
  - High level concepts and abstractions
- Reading
  - Xv6 book + source code
  - Bits of OSTEP book
- Homeworks
  - Coding real parts of the xv6 kernel
- Design riddles
  - Understanding design tradeoffs, explaining parts of xv6

Prerequisites

- Solid C coding skills
  - Xv6 is written in C
  - You need to read, code and debug
  - All homeworks are in C
  - Many questions will require explaining xv6 code
- Be able to work and code in Linux/UNIX
- Some assembly skills



How to succeed?

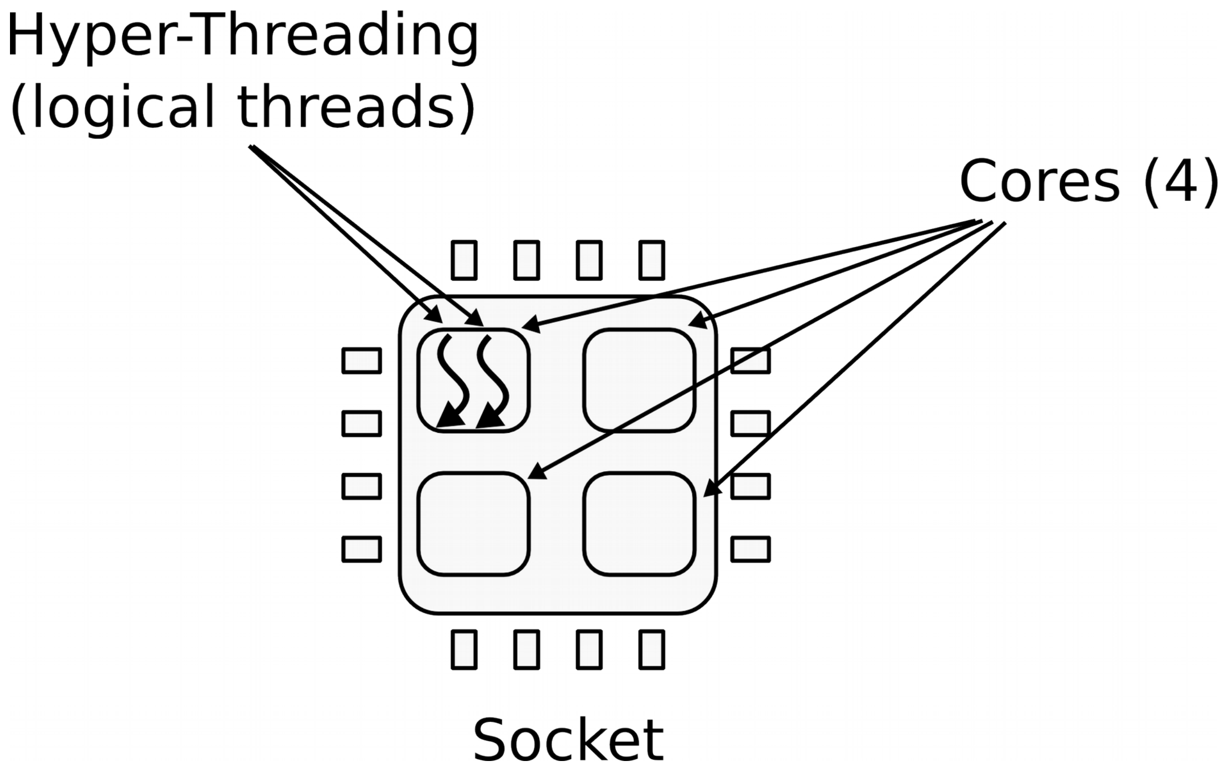
- Read the source

What is an operating system?

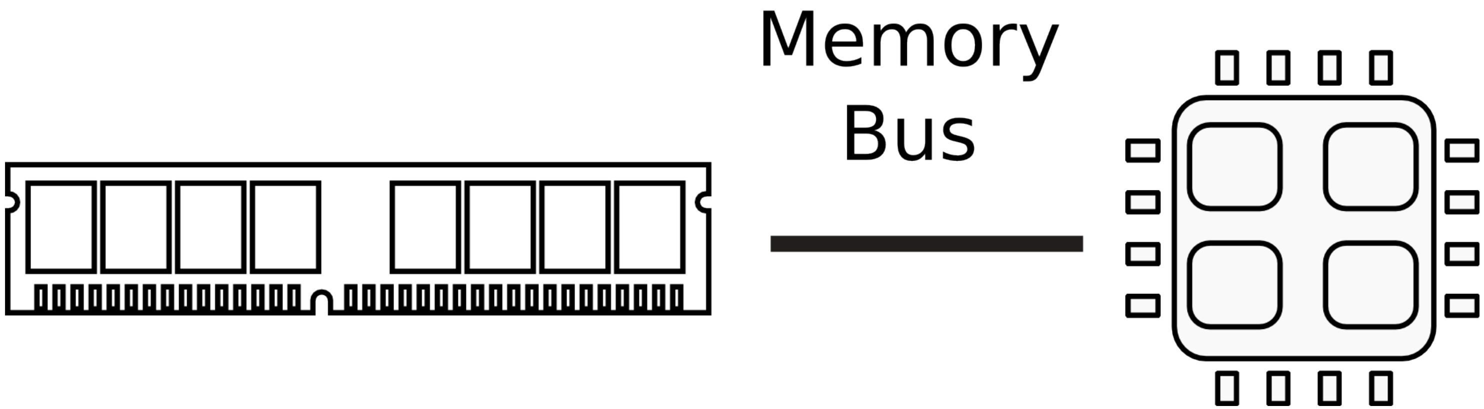
PC Hardware

CPU

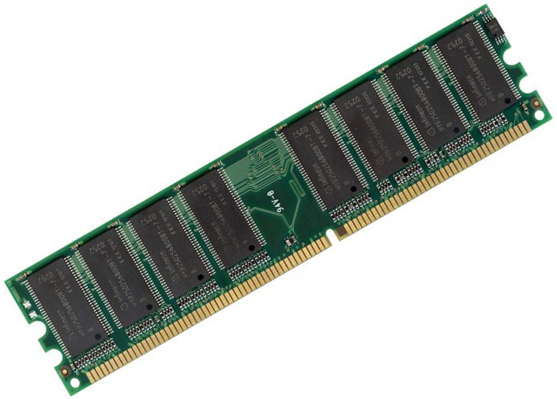
- 1 CPU socket
  - 4 cores
  - 2 logical (HT) threads each



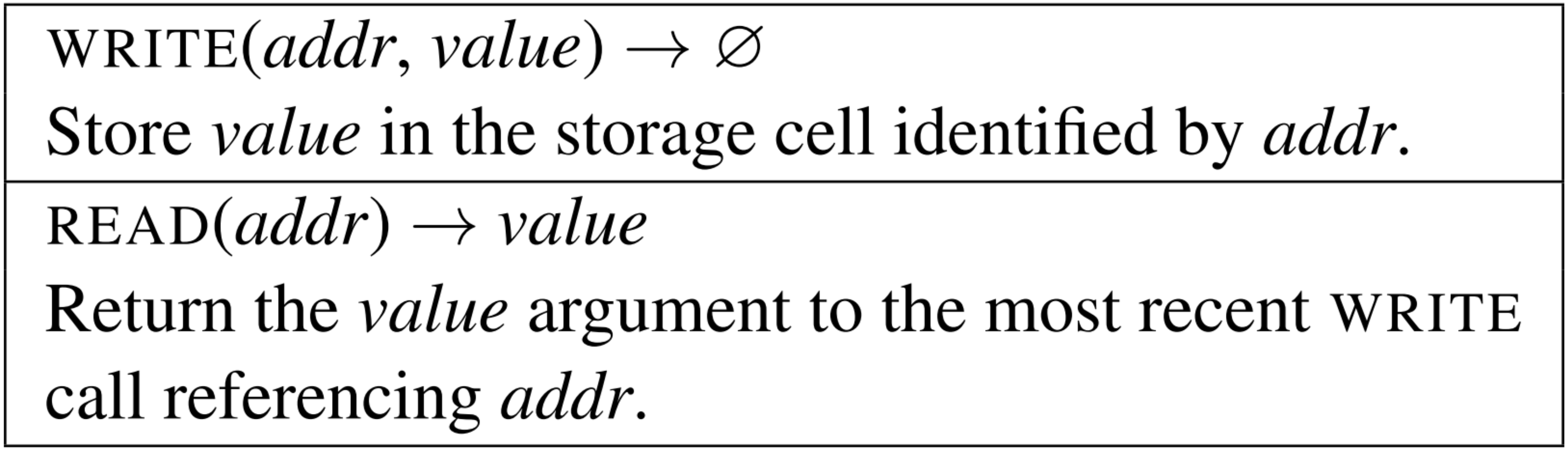
Memory



Memory
Bus



Memory abstraction



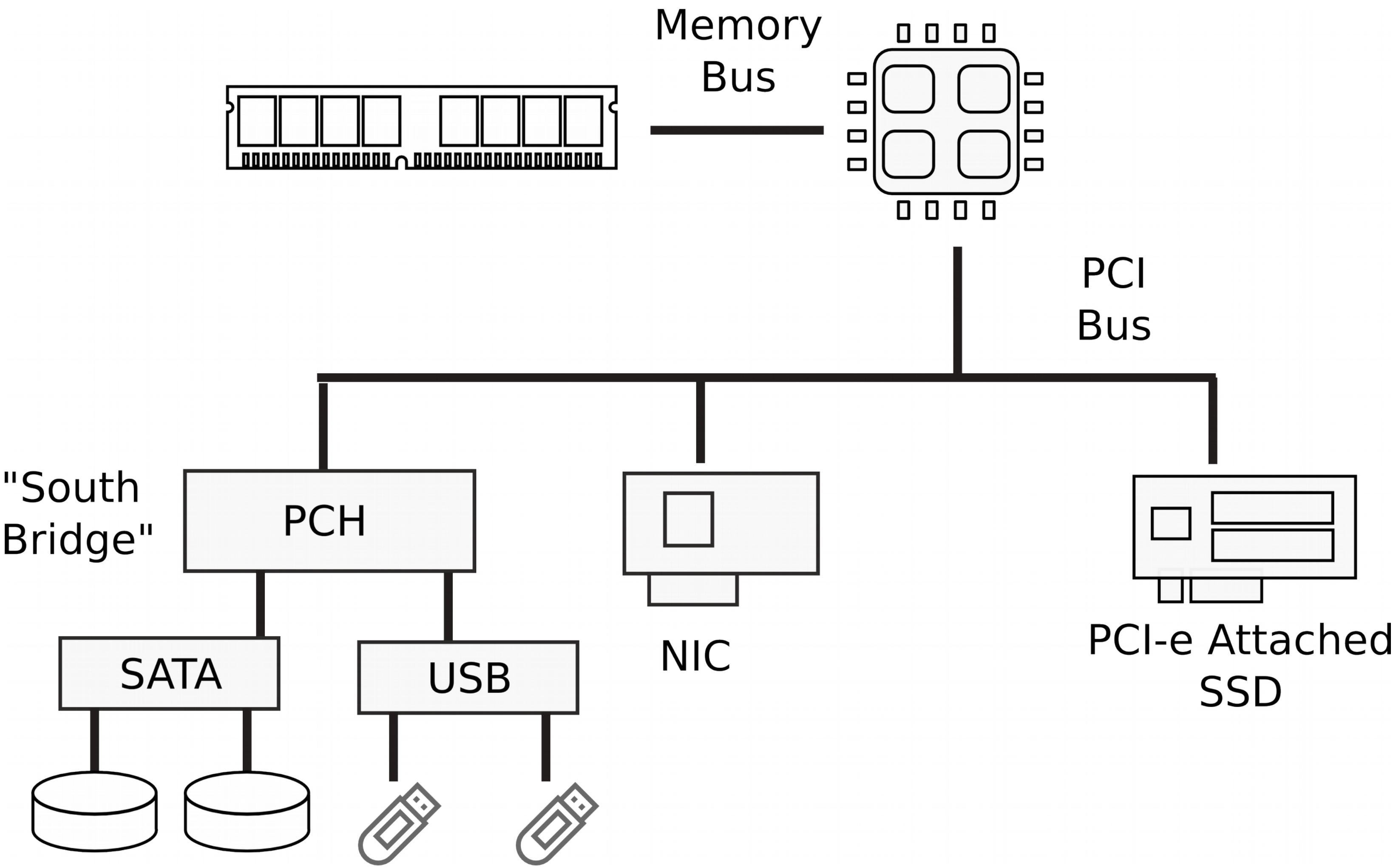
WRITE(*addr*, *value*) $\rightarrow \varnothing$

Store *value* in the storage cell identified by *addr*.

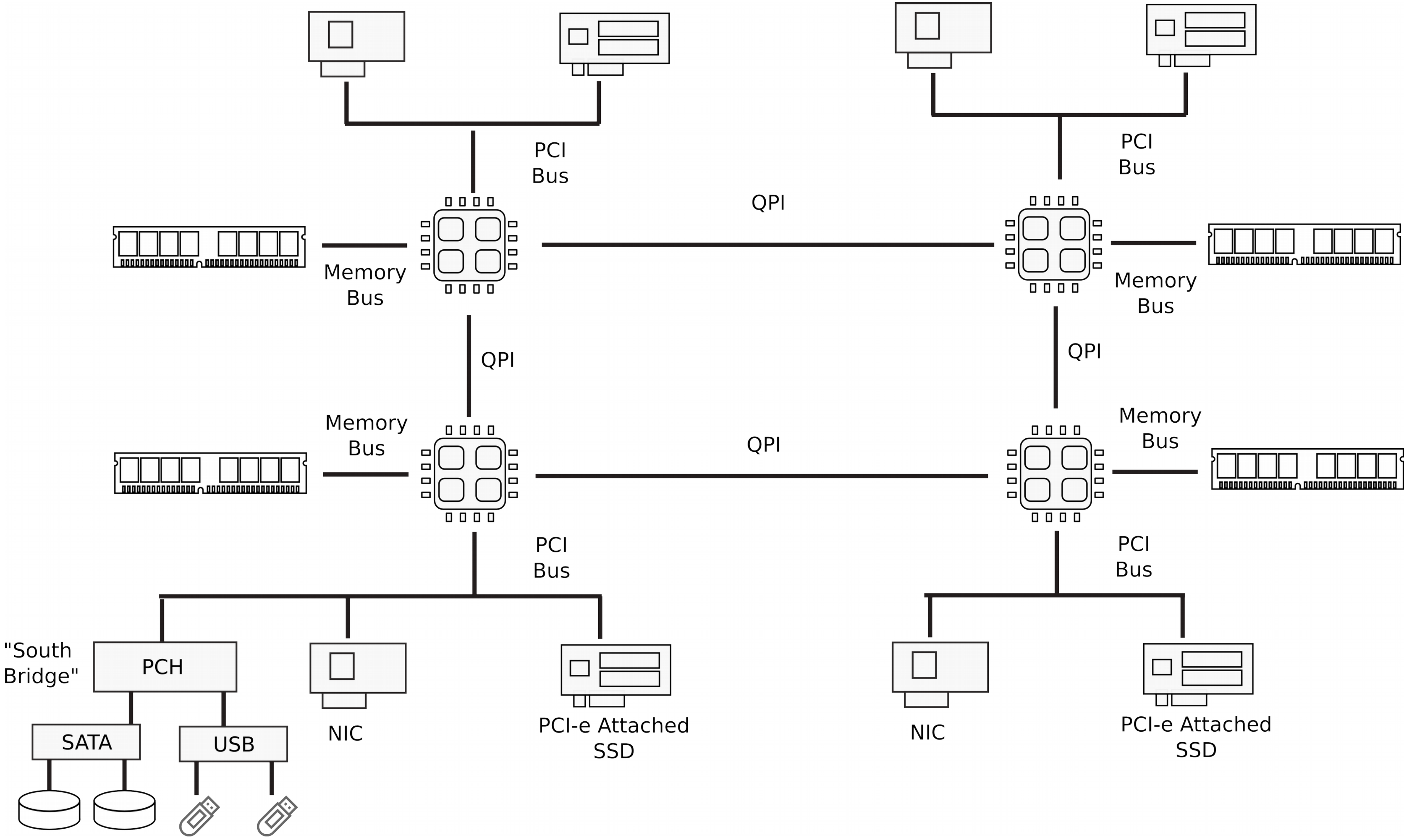
READ(*addr*) $\rightarrow$ *value*

Return the *value* argument to the most recent WRITE call referencing *addr*.

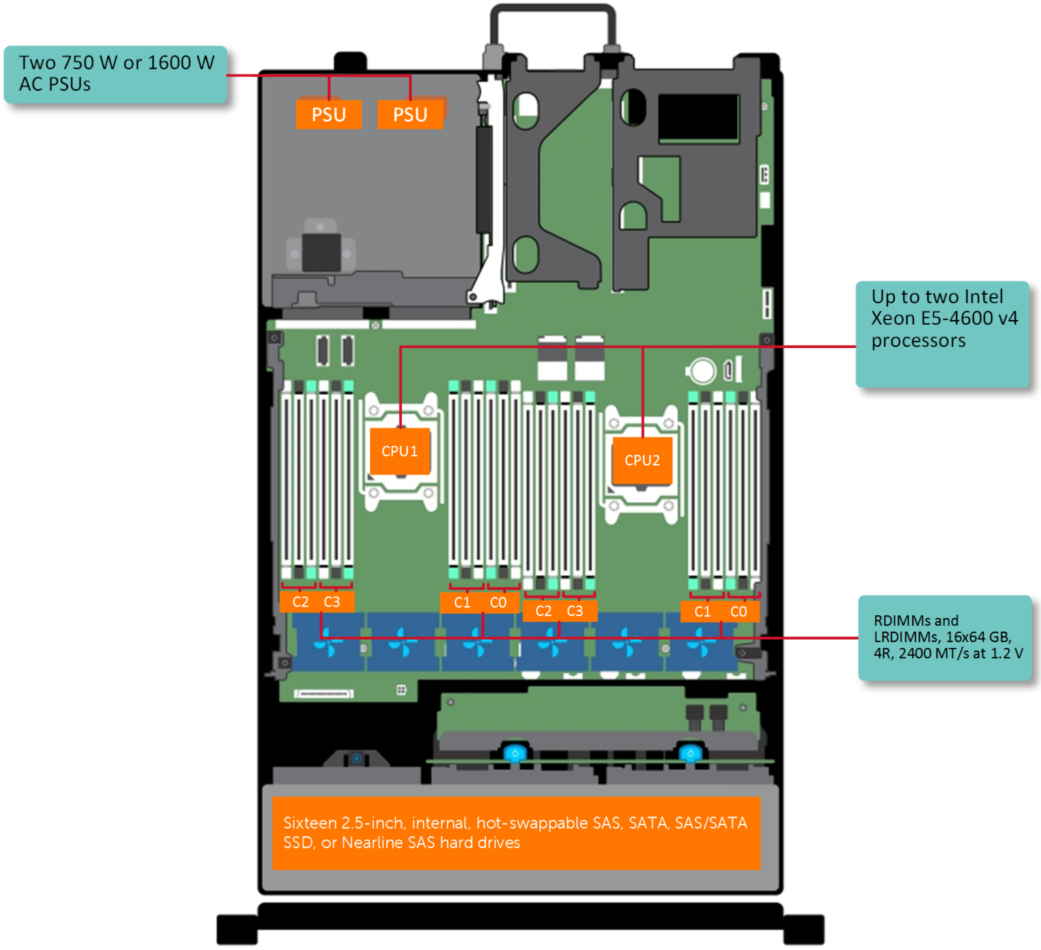
I/O Devices


Memory
Bus
PCI
Bus
"South
Bridge"
PCH
NIC
PCI-e Attached
SSD
SATA
USB

Multi-socket machines



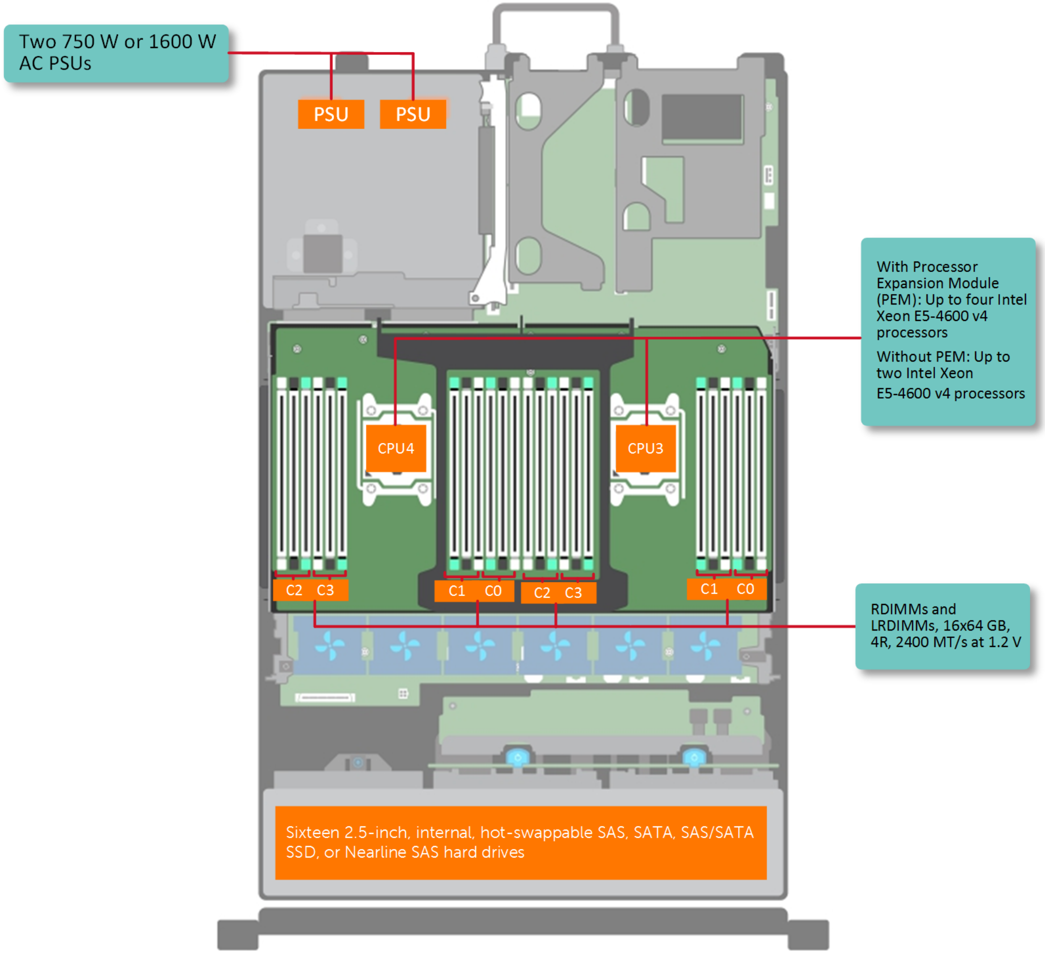
Dell R830 4-socket server


Two 750 W or 1600 W AC PSUs

Up to two Intel Xeon E5-4600 v4 processors

RDIMMs and LRDIMMs, 16x64 GB, 4R, 2400 MT/s at 1.2 V

Sixteen 2.5-inch, internal, hot-swappable SAS, SATA, SAS/SATA SSD, or Nearline SAS hard drives


Two 750 W or 1600 W AC PSUs

With Processor Expansion Module (PEM): Up to four Intel Xeon E5-4600 v4 processors

Without PEM: Up to two Intel Xeon E5-4600 v4 processors

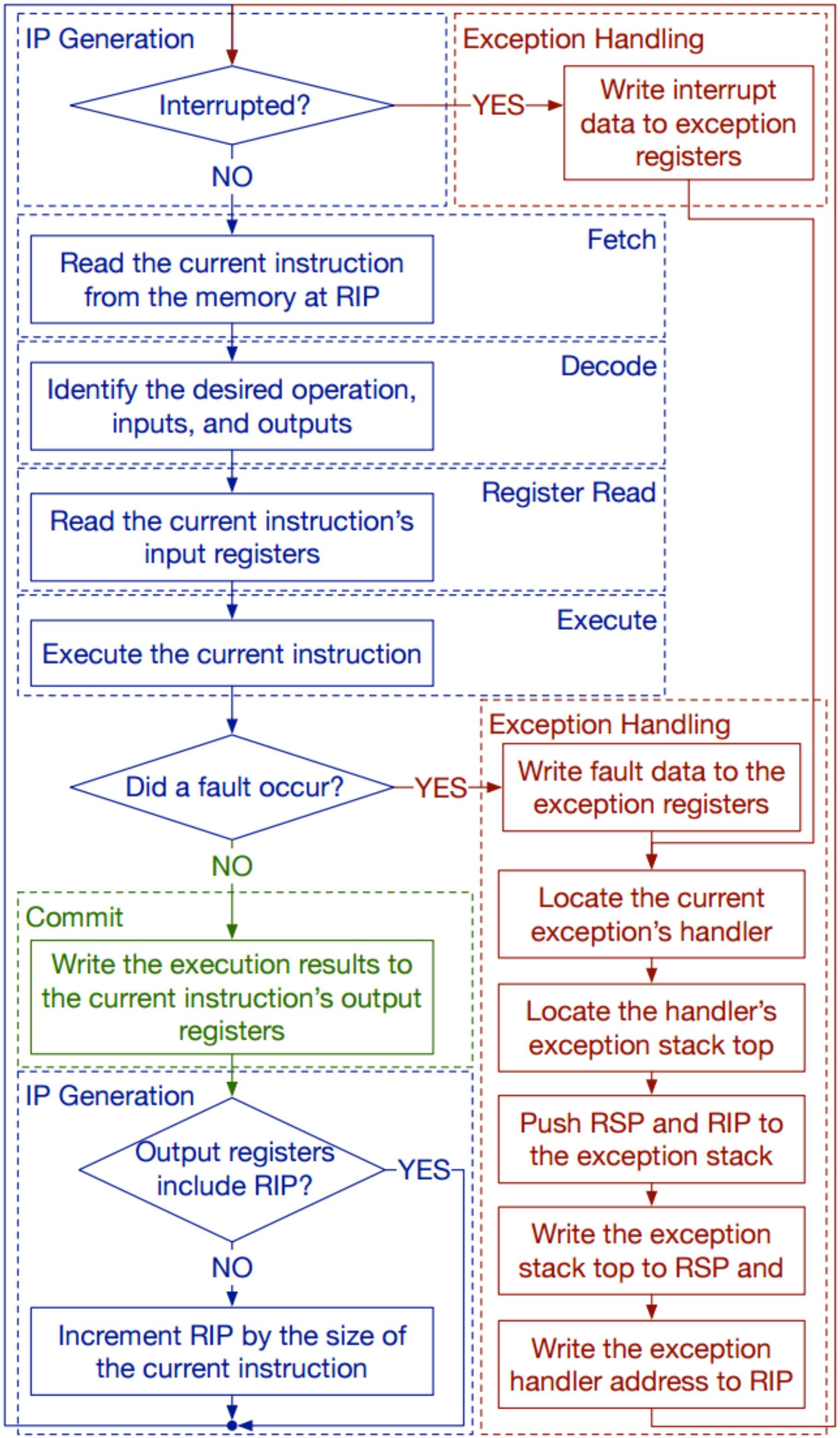
RDIMMs and LRDIMMs, 16x64 GB, 4R, 2400 MT/s at 1.2 V

Sixteen 2.5-inch, internal, hot-swappable SAS, SATA, SAS/SATA SSD, or Nearline SAS hard drives

Dell Poweredge R830 System Server with 2 sockets on the main floor and 2 sockets on the expansion

http://www.dell.com/support/manuals/us/en/19/poweredge-r830/r830_om/supported-configurations-for-the-poweredge-r830-system?guid=guid-01303b2b-f884-4435-b4e2-57bec2ce225a&lang=en-us

What does CPU do internally?

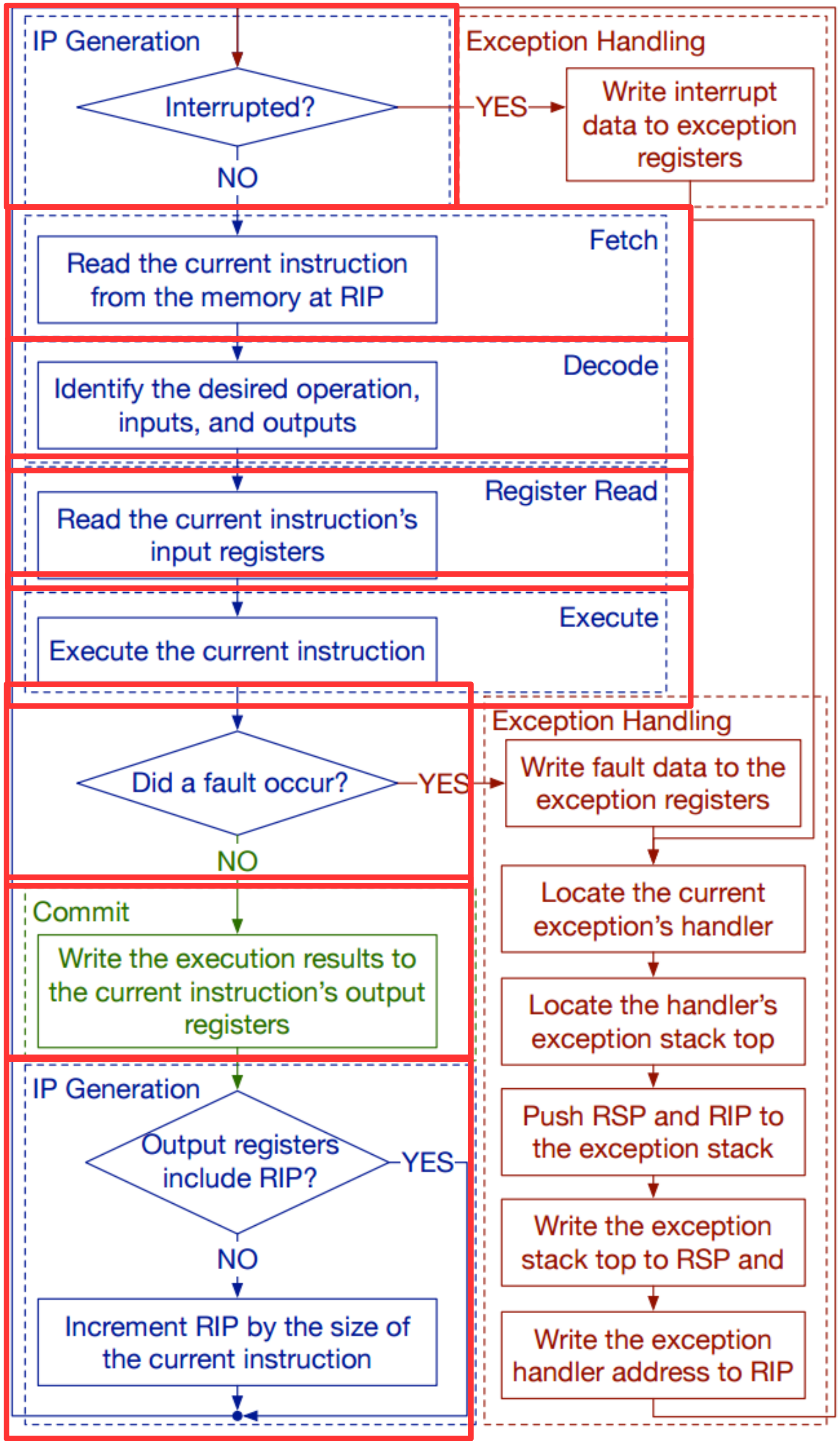

IP Generation
Interrupted?
YES
NO
Exception Handling
Write interrupt data to exception registers
Fetch
Read the current instruction from the memory at RIP
Decode
Identify the desired operation, inputs, and outputs
Register Read
Read the current instruction's input registers
Execute
Execute the current instruction
Did a fault occur?
YES
NO
Exception Handling
Write fault data to the exception registers
Locate the current exception's handler
Locate the handler's exception stack top
Push RSP and RIP to the exception stack
Write the exception stack top to RSP and
Write the exception handler address to RIP
Commit
Write the execution results to the current instruction's output registers
IP Generation
Output registers include RIP?
YES
NO
Increment RIP by the size of the current instruction

CPU execution loop

- CPU repeatedly reads instructions from memory
- Executes them
- Example

```
ADD EDX, EAX, EBX

// EDX = EAX + EBX
```



IP Generation

Interrupted?

YES

NO

Exception Handling

Write interrupt data to exception registers

Fetch

Read the current instruction from the memory at RIP

Decode

Identify the desired operation, inputs, and outputs

Register Read

Read the current instruction's input registers

Execute

Execute the current instruction

Did a fault occur?

YES

NO

Exception Handling

Write fault data to the exception registers

Locate the current exception's handler

Locate the handler's exception stack top

Push RSP and RIP to the exception stack

Write the exception stack top to RSP and

Write the exception handler address to RIP

Commit

Write the execution results to the current instruction's output registers

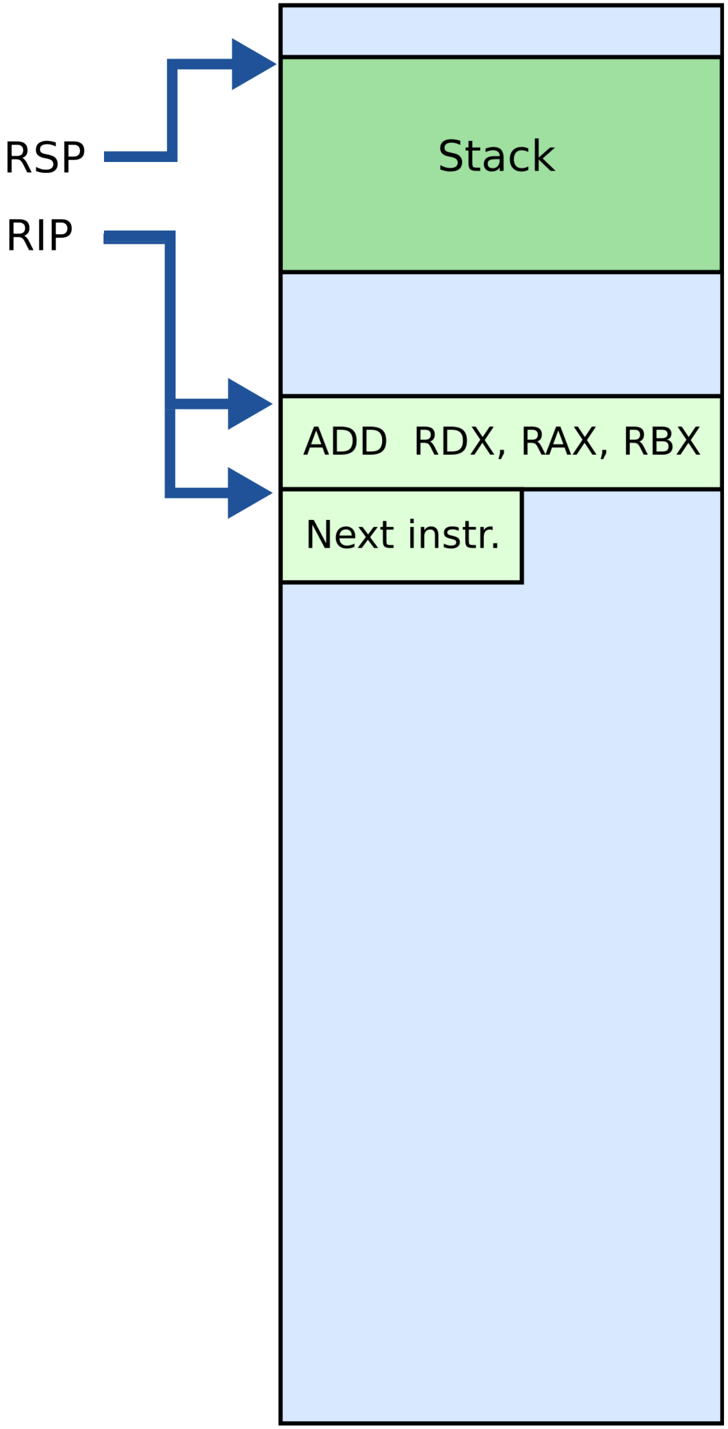
IP Generation

Output registers include RIP?

YES

NO

Increment RIP by the size of the current instruction



RSP

RIP

Stack

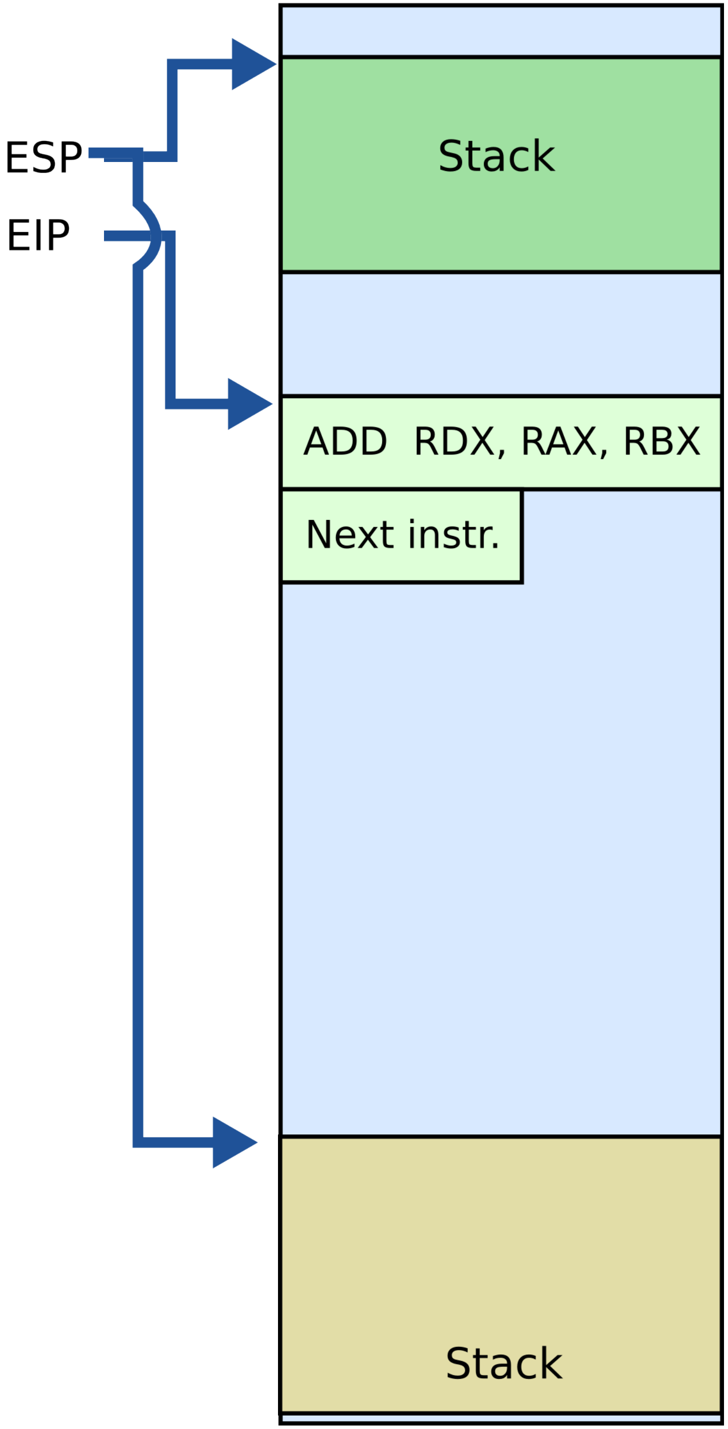
ADD RDX, RAX, RBX

Next instr.

What is stack?

Stack

- It's just a region of memory
  - Pointed by a special register ESP
- You can change ESP
  - Get a new stack



Why do we need stack?

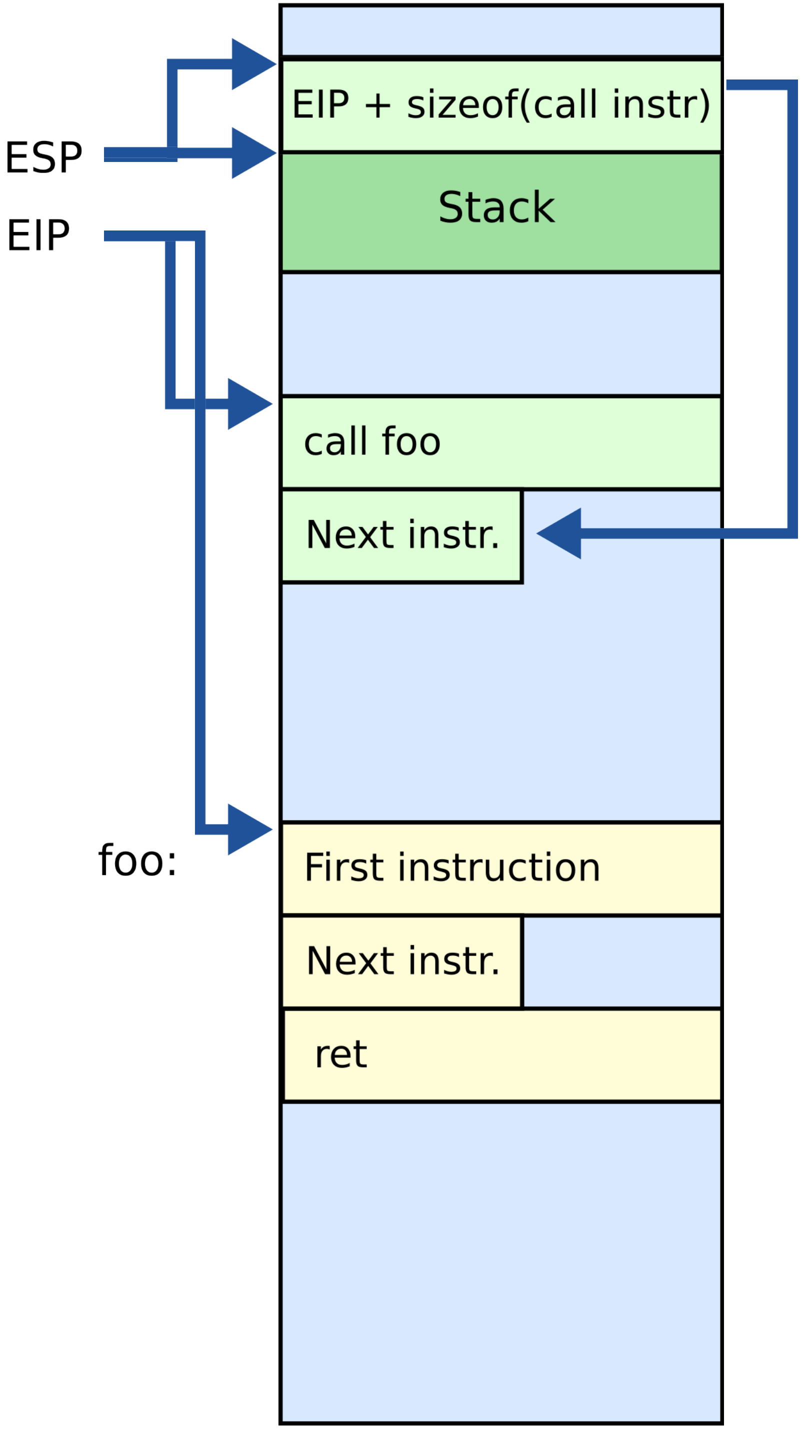
Calling functions

```
// some code...
foo();
// more code..
```

- Stack contains information for how to return from a subroutine
  - i.e., foo()

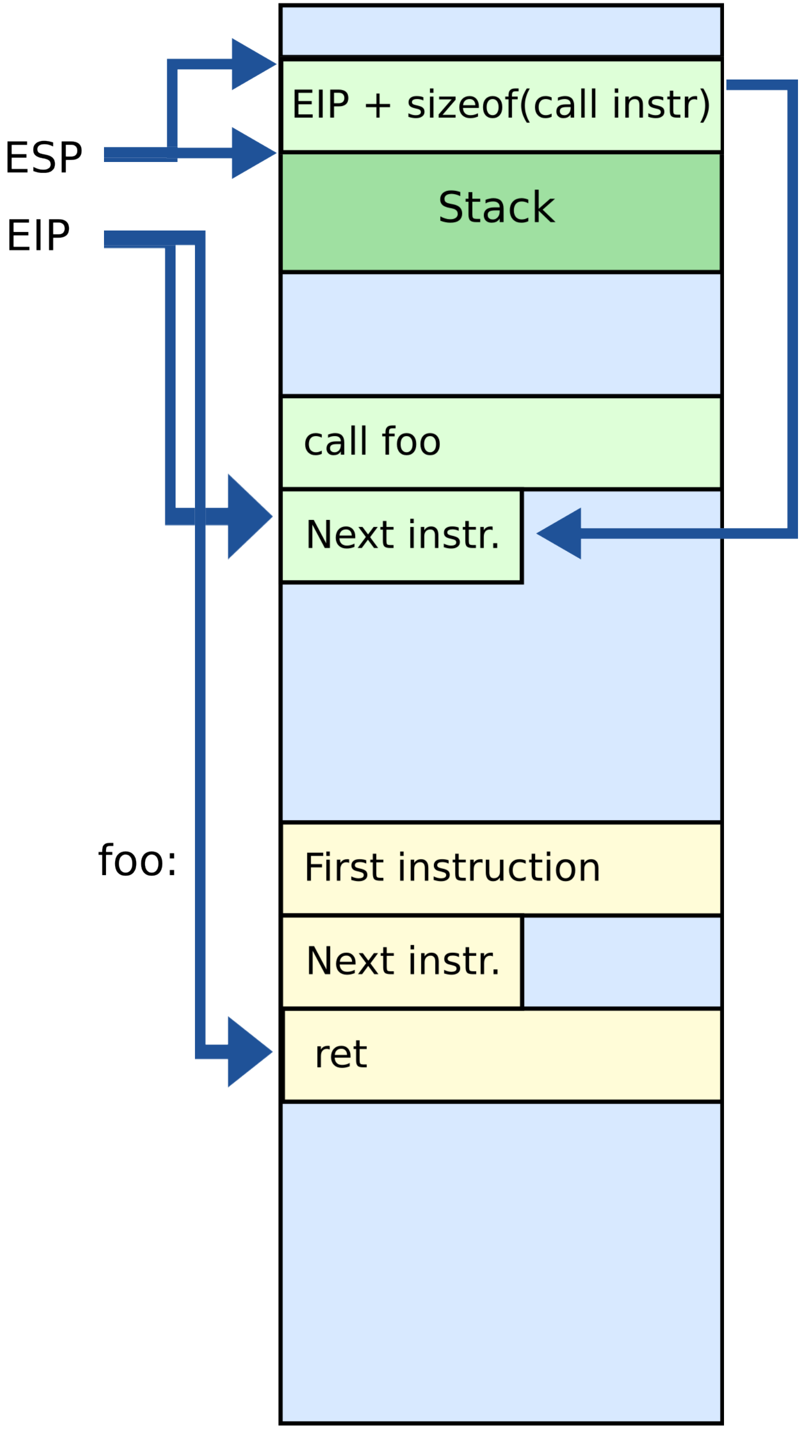
Stack

- Main purpose:
  - Store the return address for the current procedure
  - Caller pushes return address on the stack
  - Callee pops it and jumps



Stack

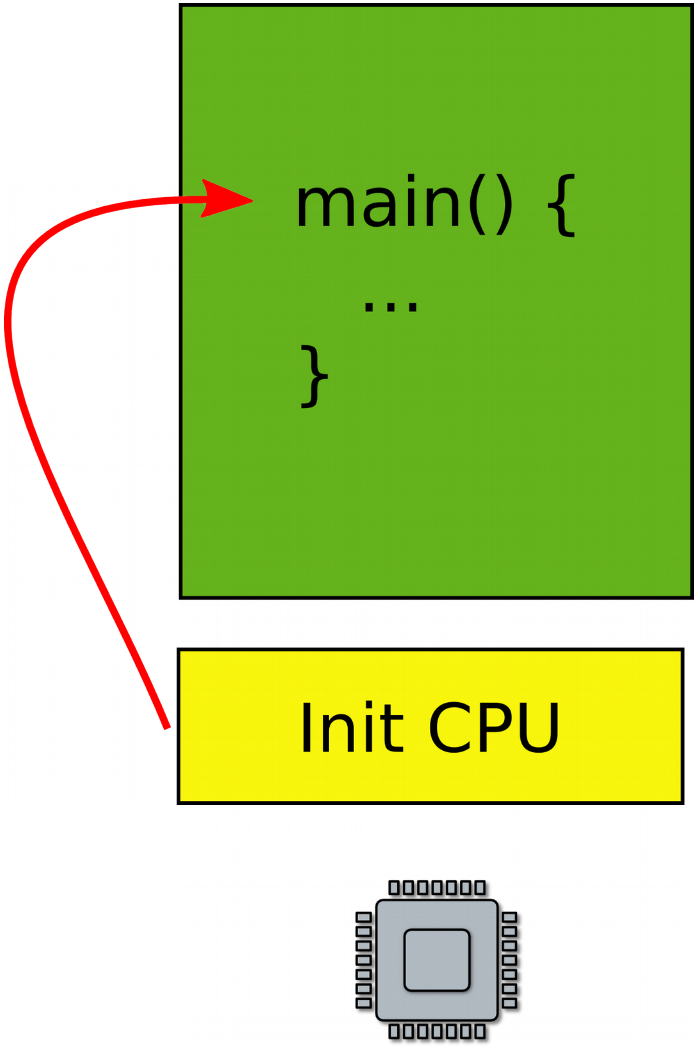
- Main purpose:
  - Store the return address for the current procedure
  - Caller pushes return address on the stack
  - Callee pops it and jumps



Simple observation

- Hardware executes instructions one by one

Goal: Run your code on a piece of hardware



- Read CPU manual
- A tiny boot layer
  - Initialize CPU
  - Jump to the entry point of your program
    - main()
  - **This can be the beginning of your OS!**

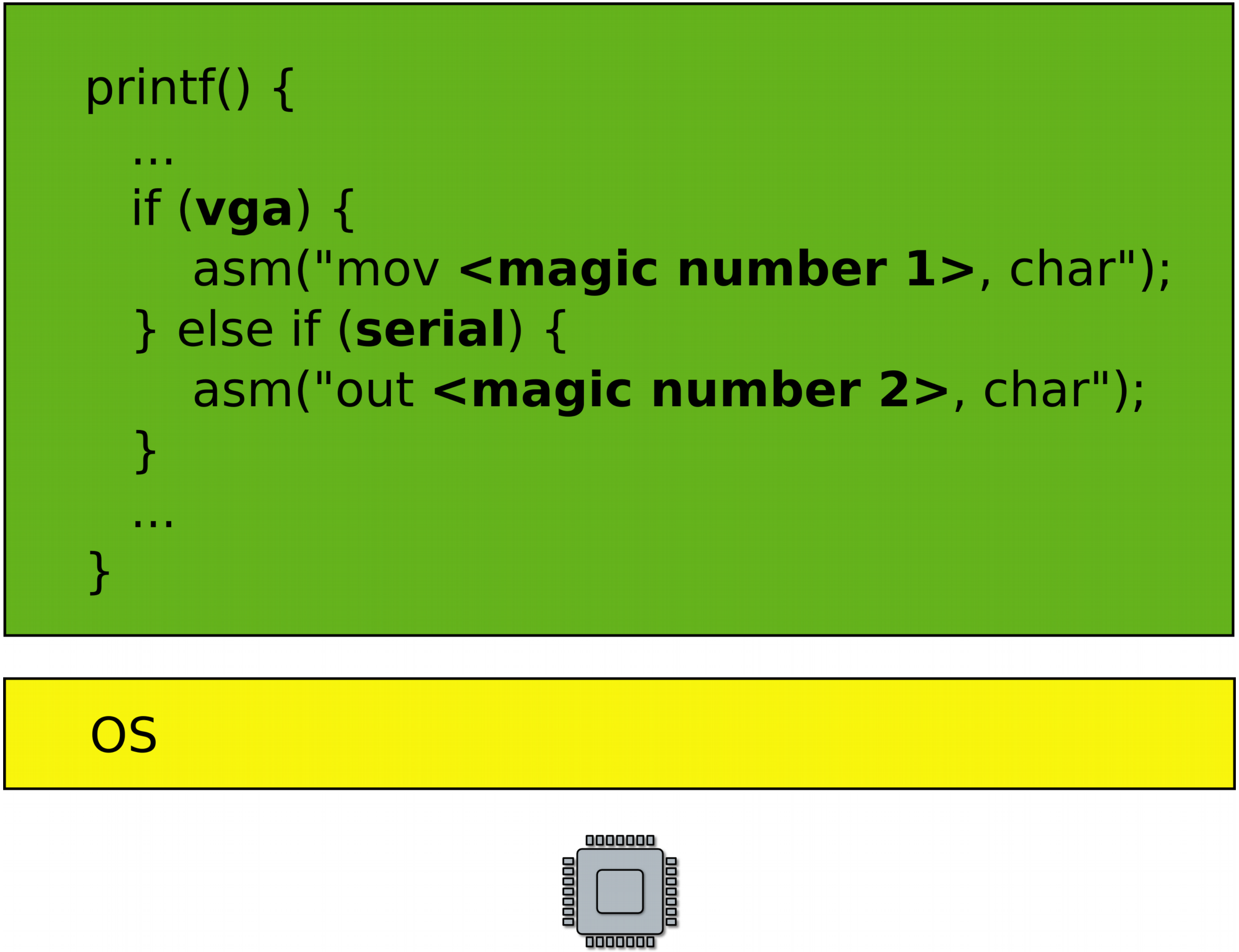
How do you learn a new programming language?

Hello world

```
printf("Hello world\n");
```

Print out a string

- On the screen or serial line

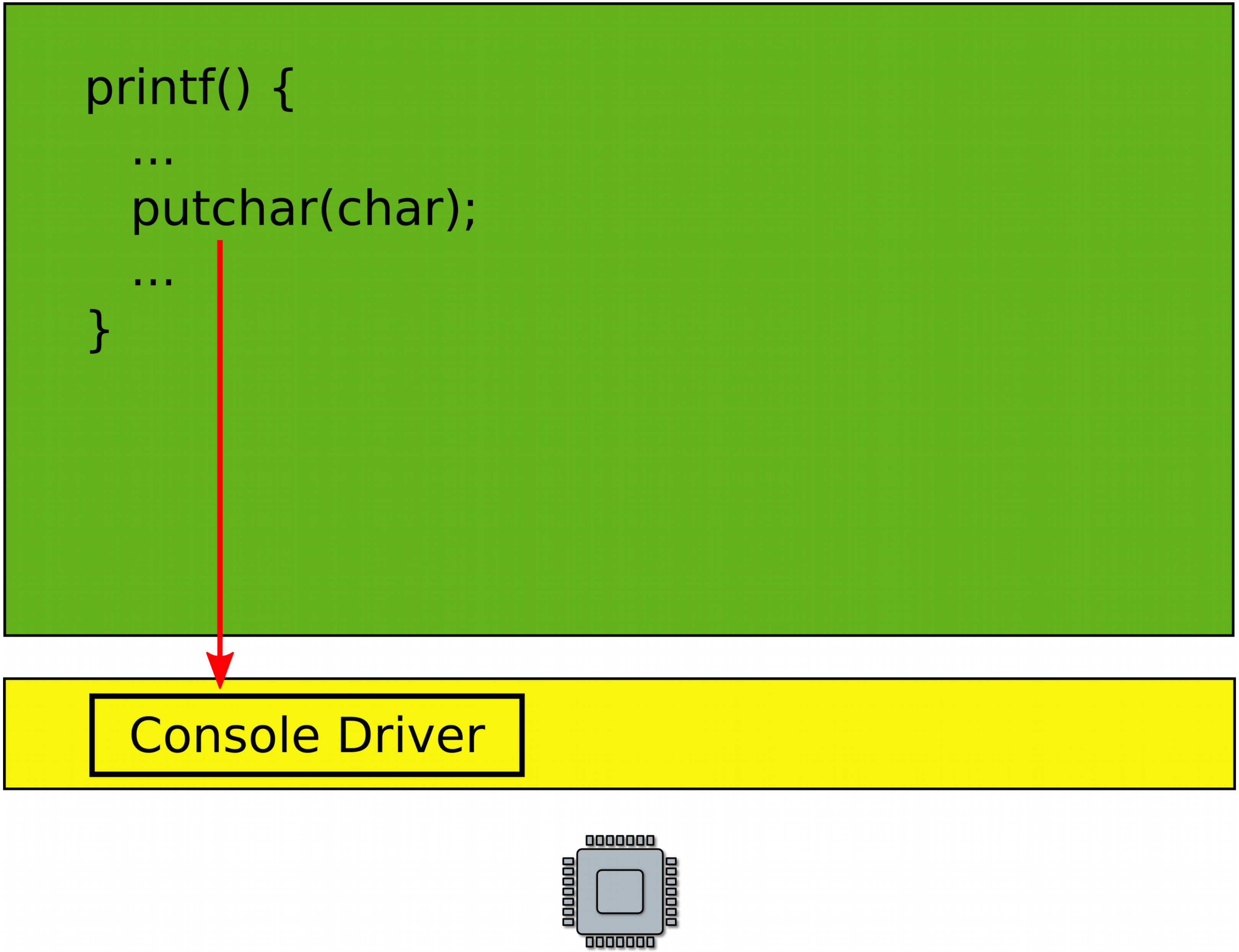


```
printf() {
    ...
    if (vga) {
        asm("mov <magic number 1>, char");
    } else if (serial) {
        asm("out <magic number 2>, char");
    }
    ...
}
```

OS

A more general interface

- First device driver



```
printf() {
   ...
   putchar(char);
   ...
}
```

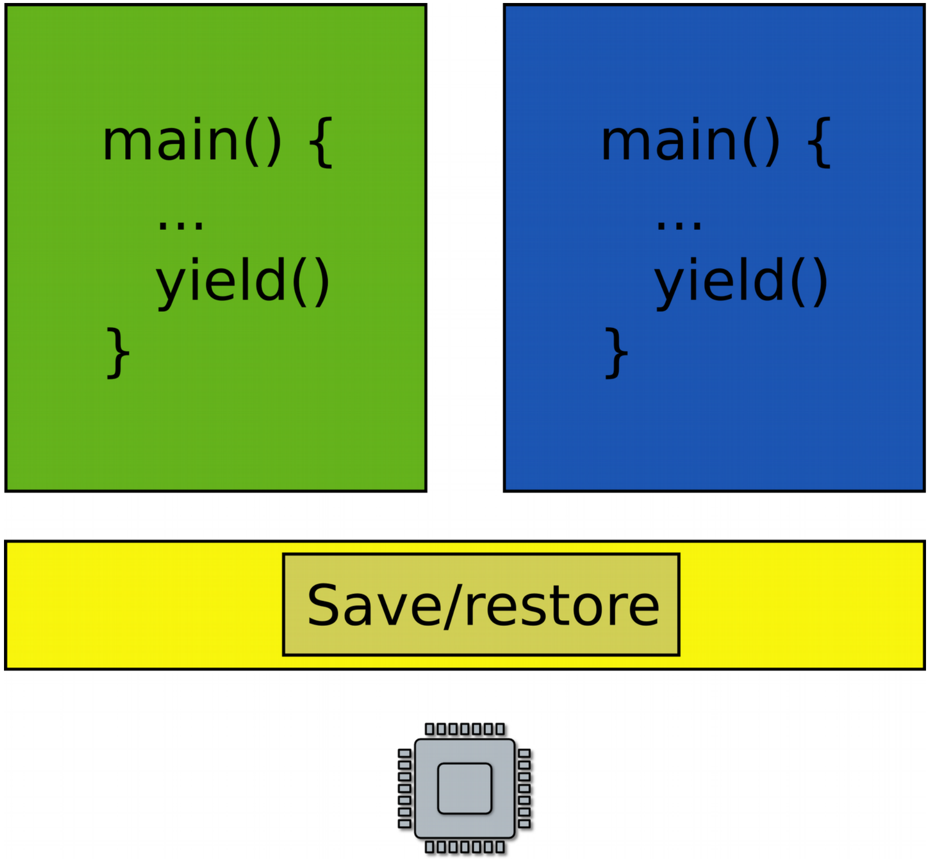
Console Driver

Device drivers

- Abstract hardware
  - Provide high-level interface
  - Hide minor differences
  - Implement some optimizations
    - Batch requests
- Examples
  - Console, disk, network interface
  - ...virtually any piece of hardware you know

OS is like a library that provides a collection of useful functions

Goal: Want to run two programs



```
main() {
   ...
   yield()
}
```

```
main() {
   ...
   yield()
}
```

Save/restore

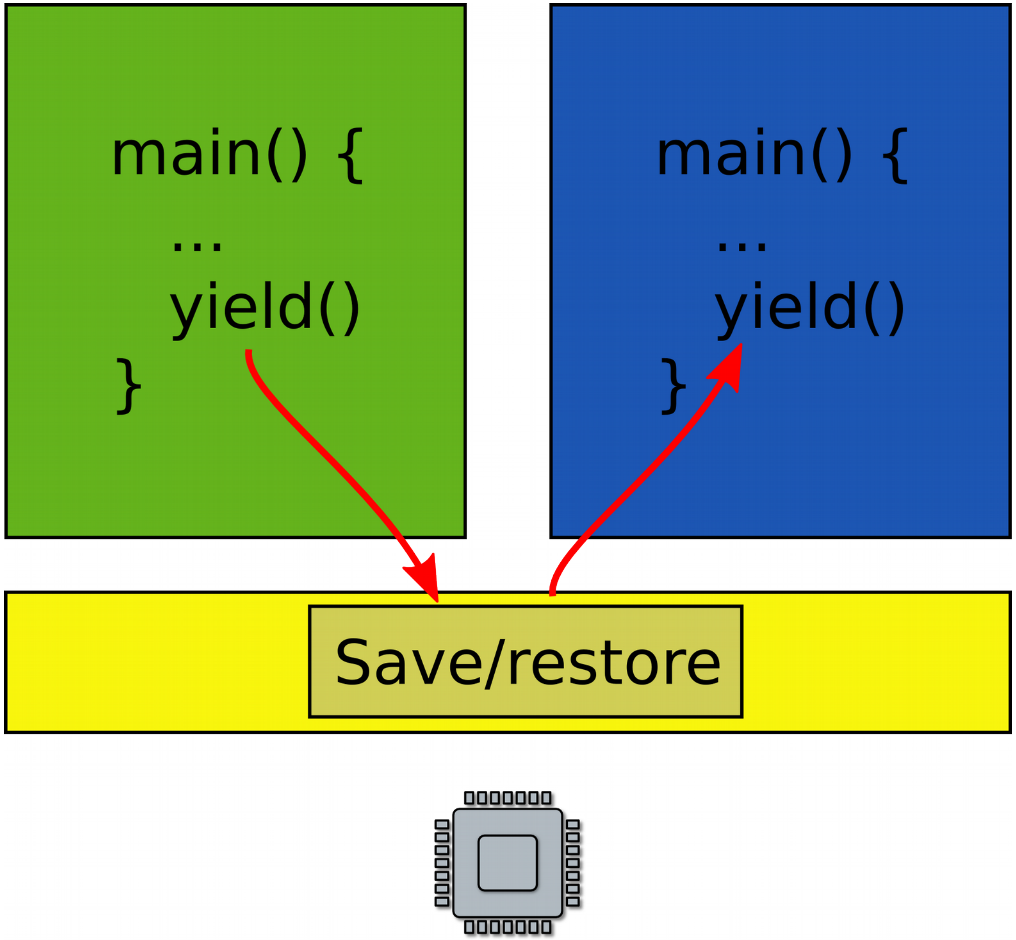
- What does it meanings?
    - Only one CPU
- Run one, then run another one

Very much like car sharing

Time sharing

- Programs use CPU in turns
  - One program runs
  - Then OS takes control
  - Launches another program
  - Then another program runs
  - OS takes control again
  - ...

Goal: Want to run two programs



- Exit into the kernel periodically
- Context switch
  - Save state of one program
  - Restore state of another program

What is this state?

State of the program

- Roughly it’s
  - Registers
  - Memory

  - Plus some state (data structures) in the kernel associated with the program
    - Information about files opened by the program, i.e. file descriptors
    - Information about network flows
    - Information about address space, loaded libraries, communication channels to other programs, etc.

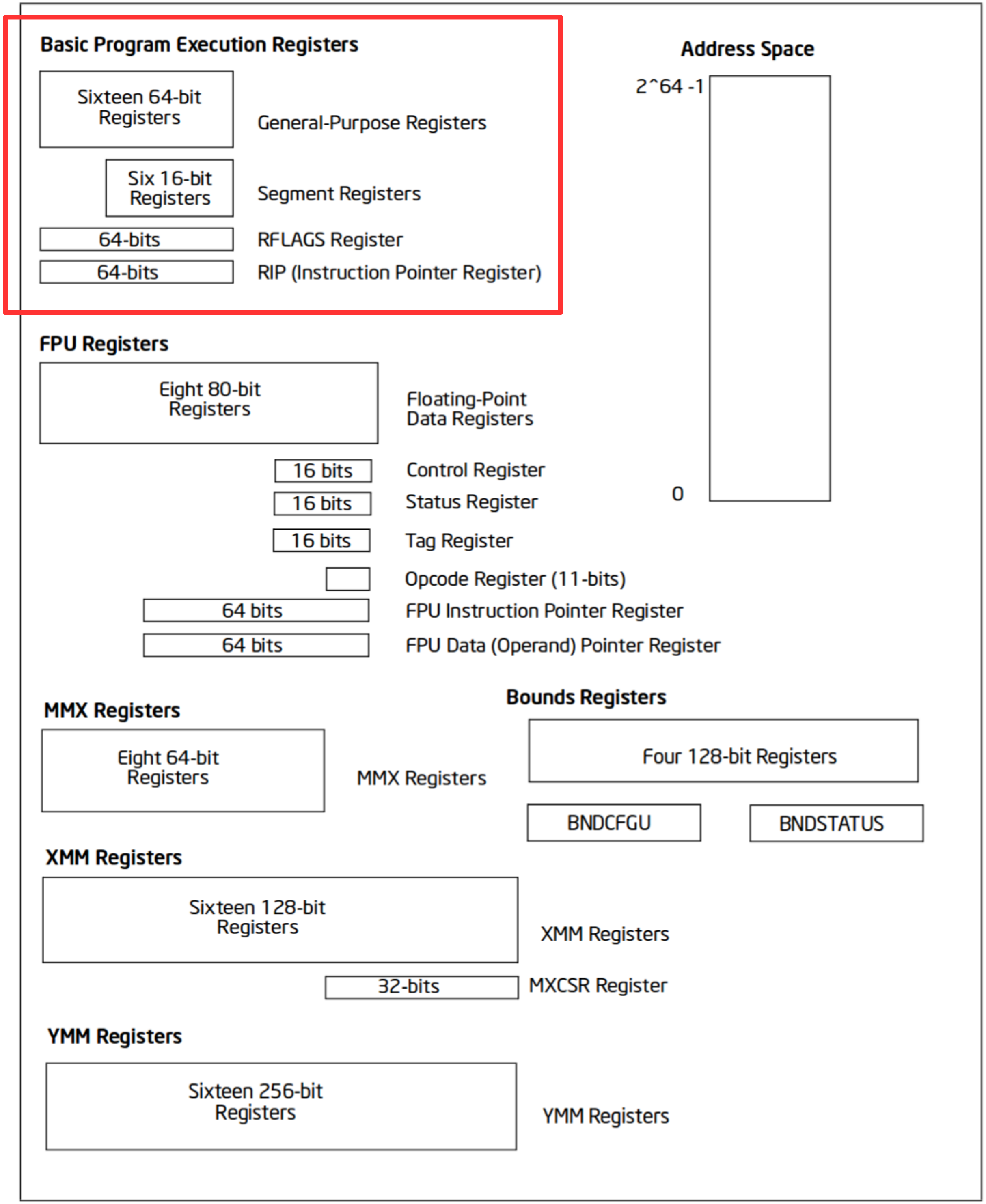
Saving and restoring state

- Note that you do not really have to save/restore in-kernel state on the context switch
  - It’s in the kernel already, i.e., in some part of the memory where kernel keeps its data structures
  - You only have to switch from using one to using another
    - i.e., instead of using the file descriptor table (can be as simple as array) for program X start using at file descriptor table for program Y

Saving and restoring state

- All you have to save are internal structures of the CPU, i.e.
  - Registers
  - Note CPU has more registers then just
    - General registers, i.e., EAX, EBX, …
      - 8 general registers in x86 32bit mode
      - 16 general registers in x86 64bit mode

Intel x86 64bit Execution Environment



Basic Program Execution Registers

Sixteen 64-bit Registers — General-Purpose Registers

Six 16-bit Registers — Segment Registers

64-bits — RFLAGS Register

64-bits — RIP (Instruction Pointer Register)

Address Space

2^64 -1

0

FPU Registers

Eight 80-bit Registers — Floating-Point Data Registers

16 bits — Control Register

16 bits — Status Register

16 bits — Tag Register

Opcode Register (11-bits)

64 bits — FPU Instruction Pointer Register

64 bits — FPU Data (Operand) Pointer Register

MMX Registers

Eight 64-bit Registers — MMX Registers

Bounds Registers

Four 128-bit Registers

BNDCFGU

BNDSTATUS

XMM Registers

Sixteen 128-bit Registers — XMM Registers

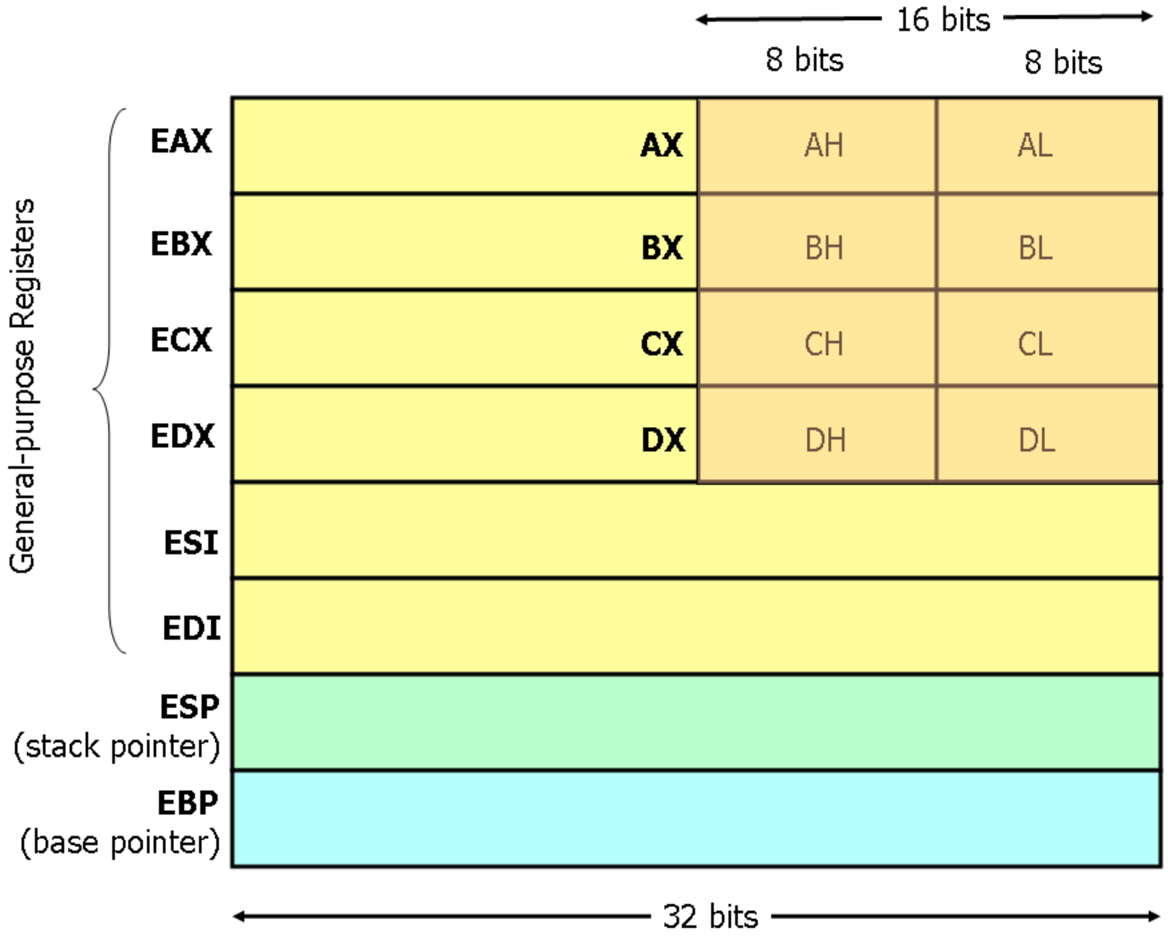
32-bits — MXCSR Register

YMM Registers

Sixteen 256-bit Registers — YMM Registers

https://software.intel.com/sites/default/files/managed/a4/60/253665-sdm-vol-1.pdf

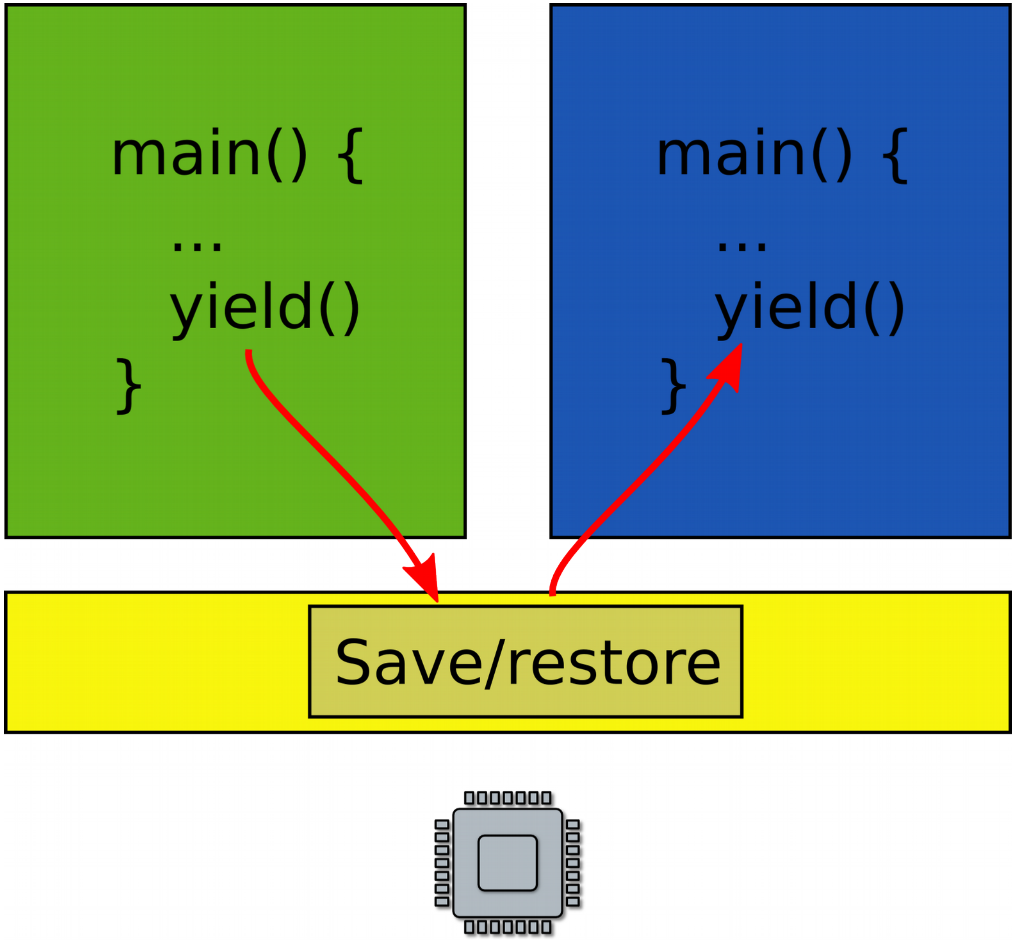
General registers



More registers...

- This is a bit misleading…
- CPU also has registers that describe state of
  - Segments
  - Page tables
  - Interrupt tables
  - Etc.
- If they don’t change you don’t have to save/restore them

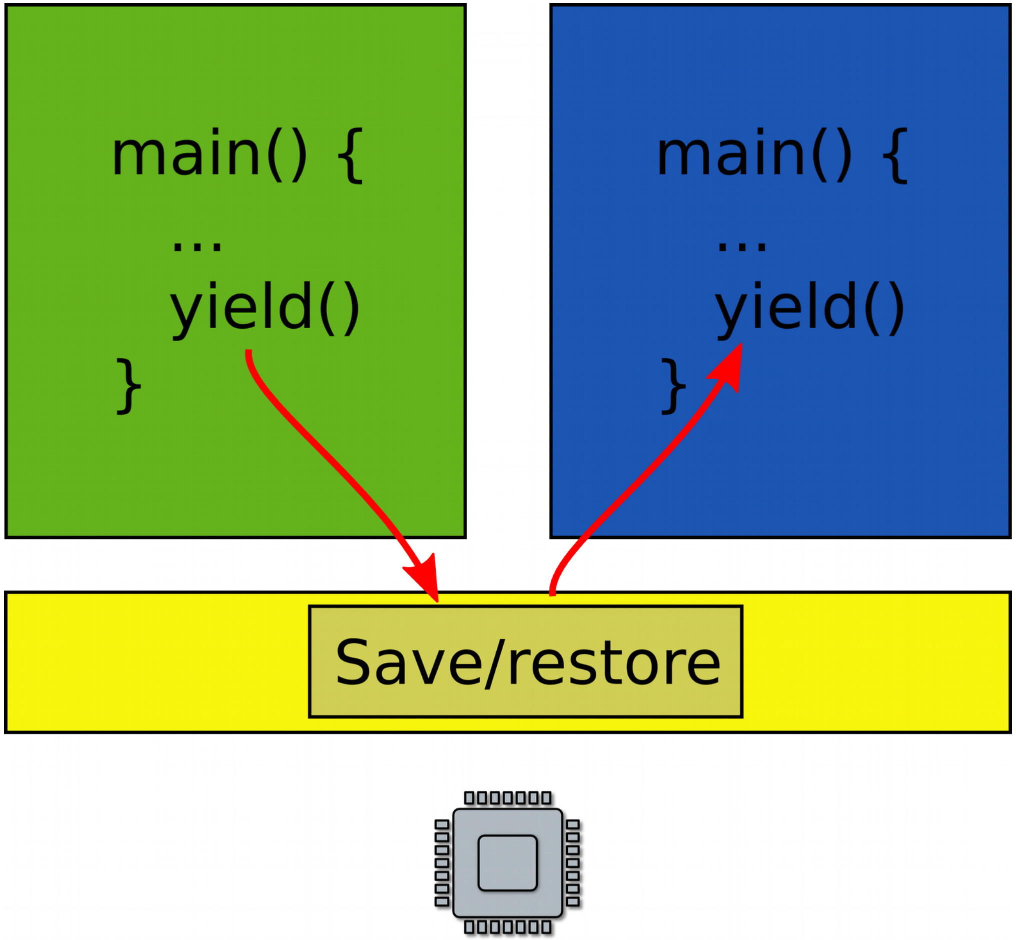
But anyway… if you want to run two programs



- Exit into the kernel periodically
- Context switch
  - Save state of one program
  - Restore state of another program

What about memory?

- Two programs, one memory?



Time-share memory

- Well you can copy in and out the state of the program into a region of memory where it can run
    - Similar to time-sharing the CPU

Time-share memory

- Well you can copy in and out the state of the program into a region of memory where it can run
    - Similar to time-sharing the CPU
- What do you think is wrong with this approach?

Time-share memory

- Well you can copy in and out the state of the program into a region of memory where it can run
  - Similar to time-sharing the CPU

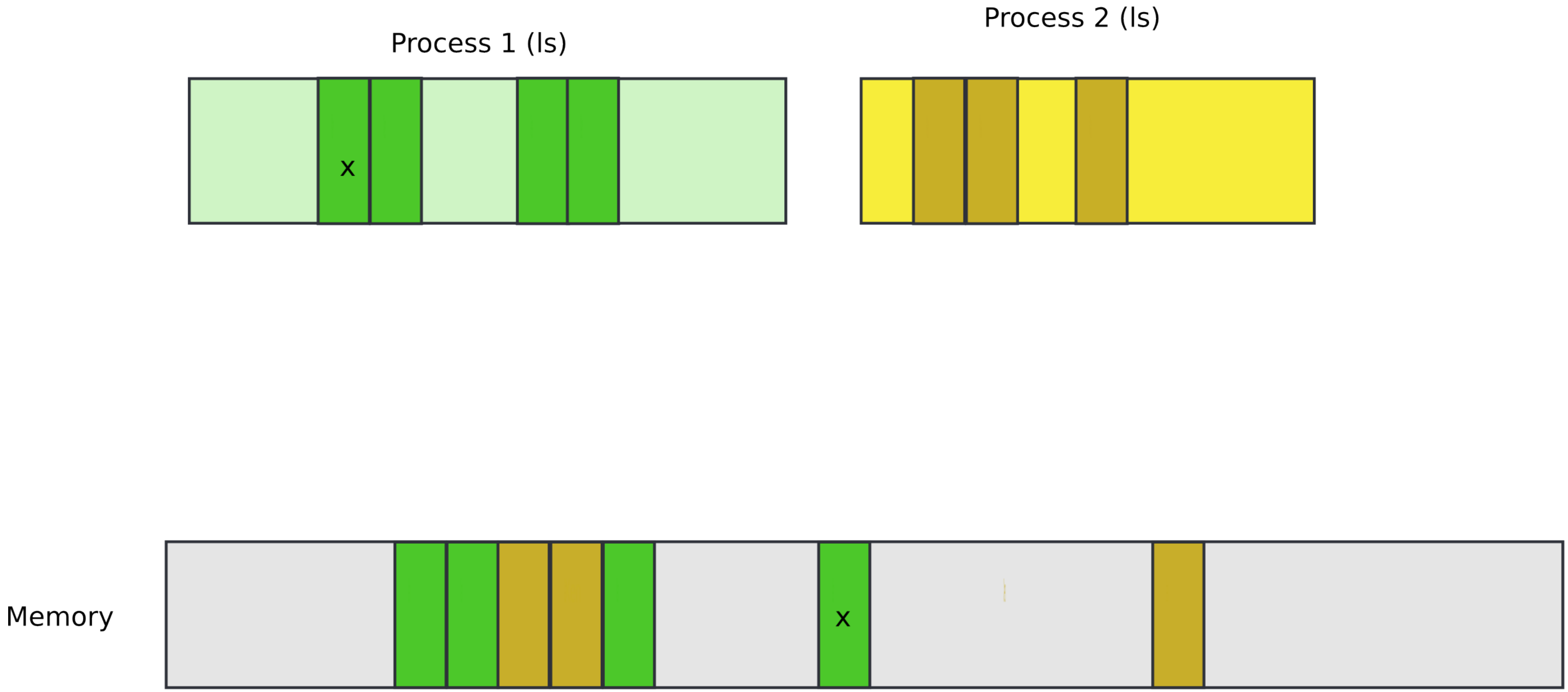
- What do you think is wrong with this approach?
  - Unlike registers the state of the program in memory can be large
  - Takes time to copy it in and out

Space sharing: virtual address spaces

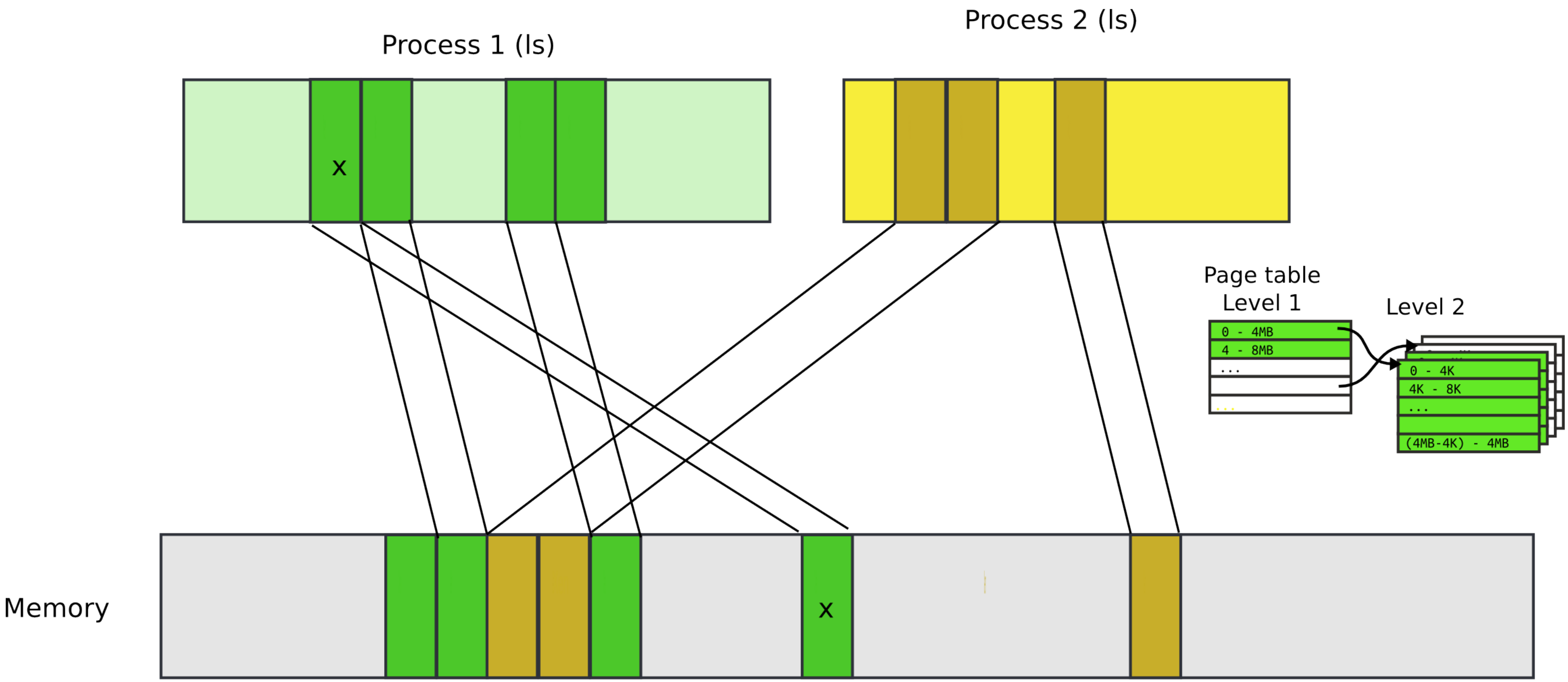
- Illusion of a private memory for each application
  - Keep a description of an address space
  - In one of the registers

- OS maintains description of address spaces
  - Switches between them

Address spaces and paging



Address spaces and paging

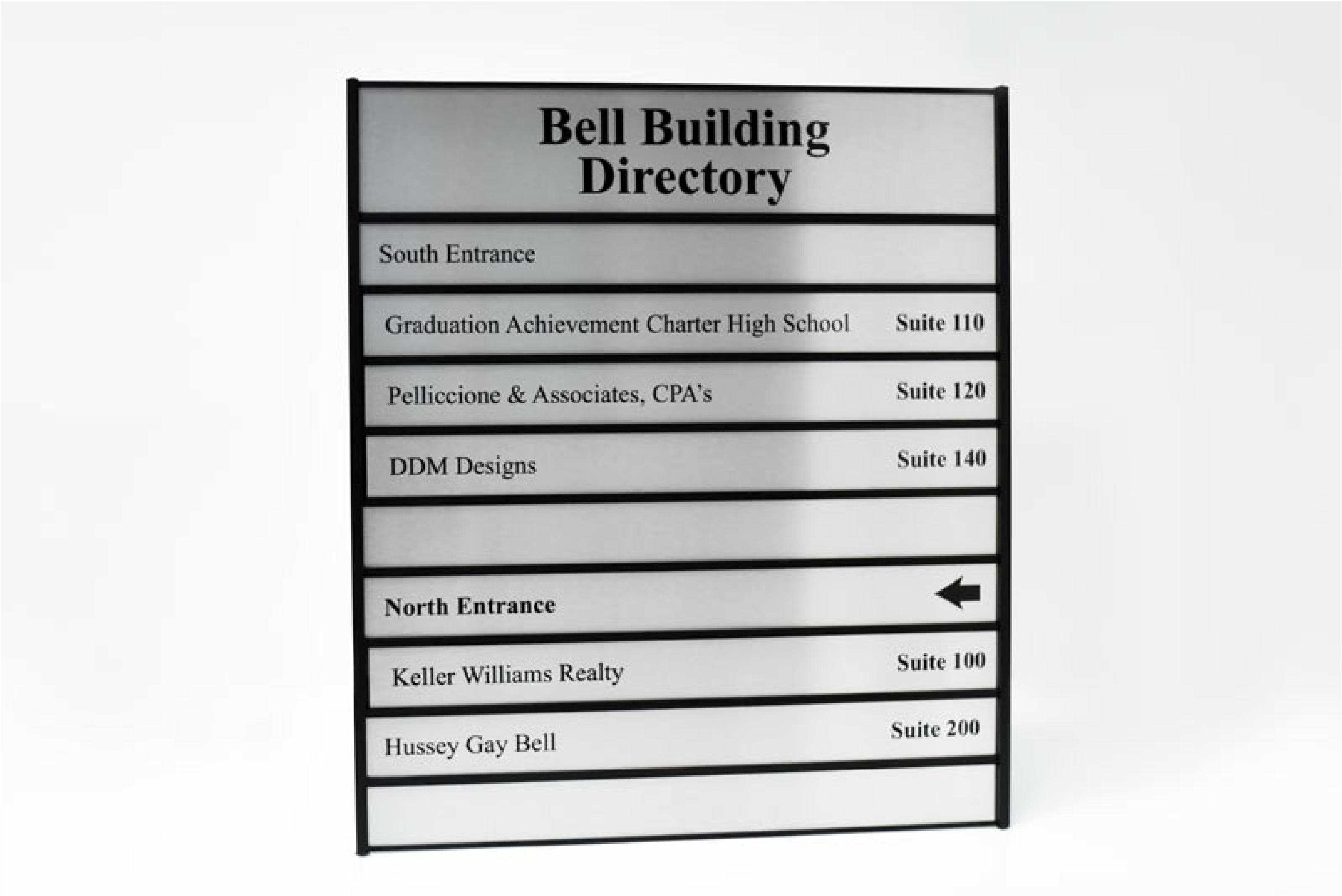


Paging idea

- Break up memory into 4096-byte chunks called pages
  - Modern hardware supports 2MB, 4MB, and 1GB pages
- Independently control mapping for each page of linear address space

Notice the main difference: time-sharing vs space sharing

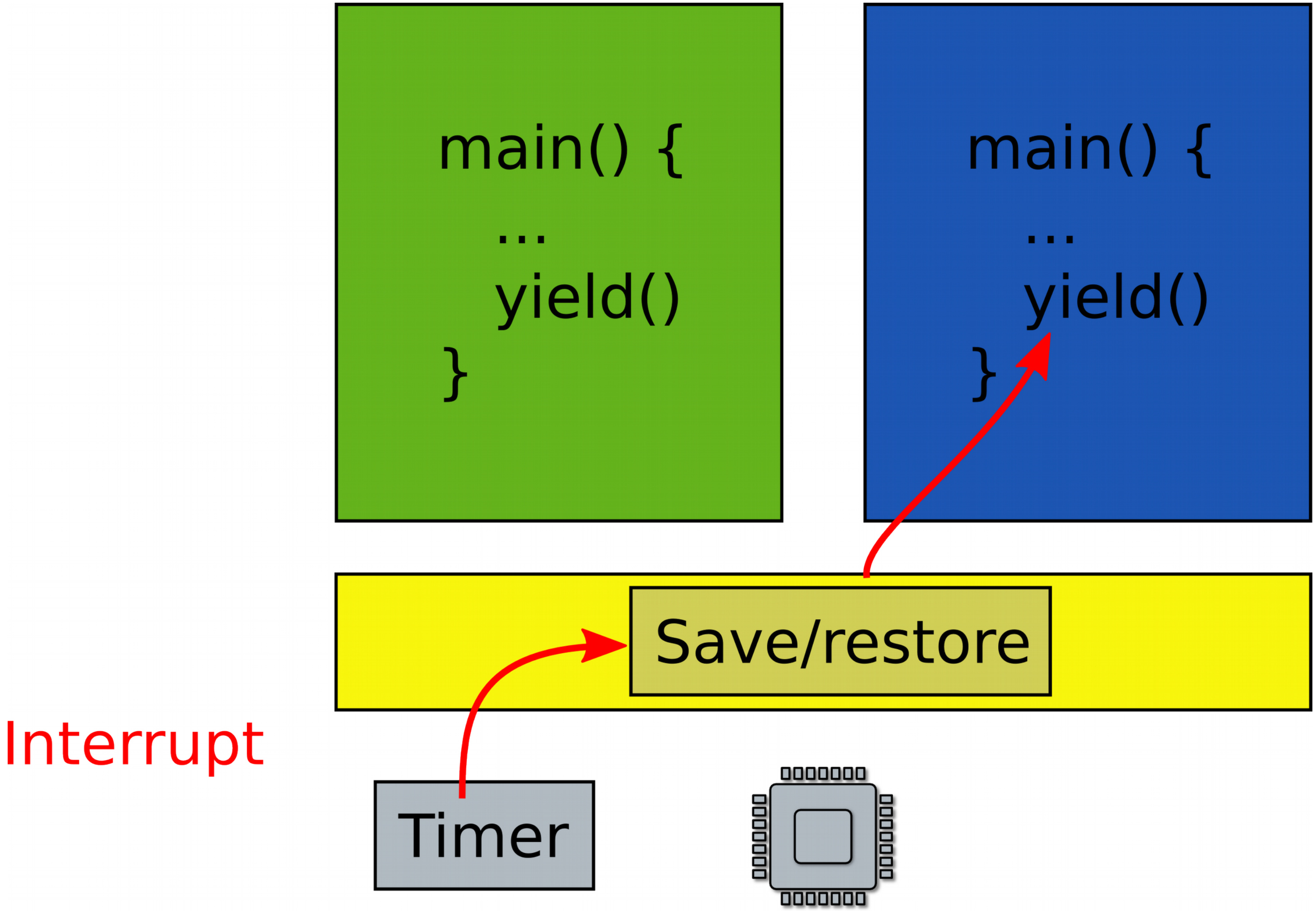
Space sharing is like renting a some rooms in an office building


Bell Building Directory
South Entrance
Graduation Achievement Charter High School Suite 110
Pelliccione & Associates, CPA's Suite 120
DDM Designs Suite 140
North Entrance
Keller Williams Realty Suite 100
Hussey Gay Bell Suite 200

Staying in control

Staying in control

- What if one program fails to release the CPU?
- It will run forever. Need a way to preempt it. How?



```
main() {
    ...
    yield()
}
```

```
main() {
    ...
    yield()
}
```

Save/restore

Interrupt

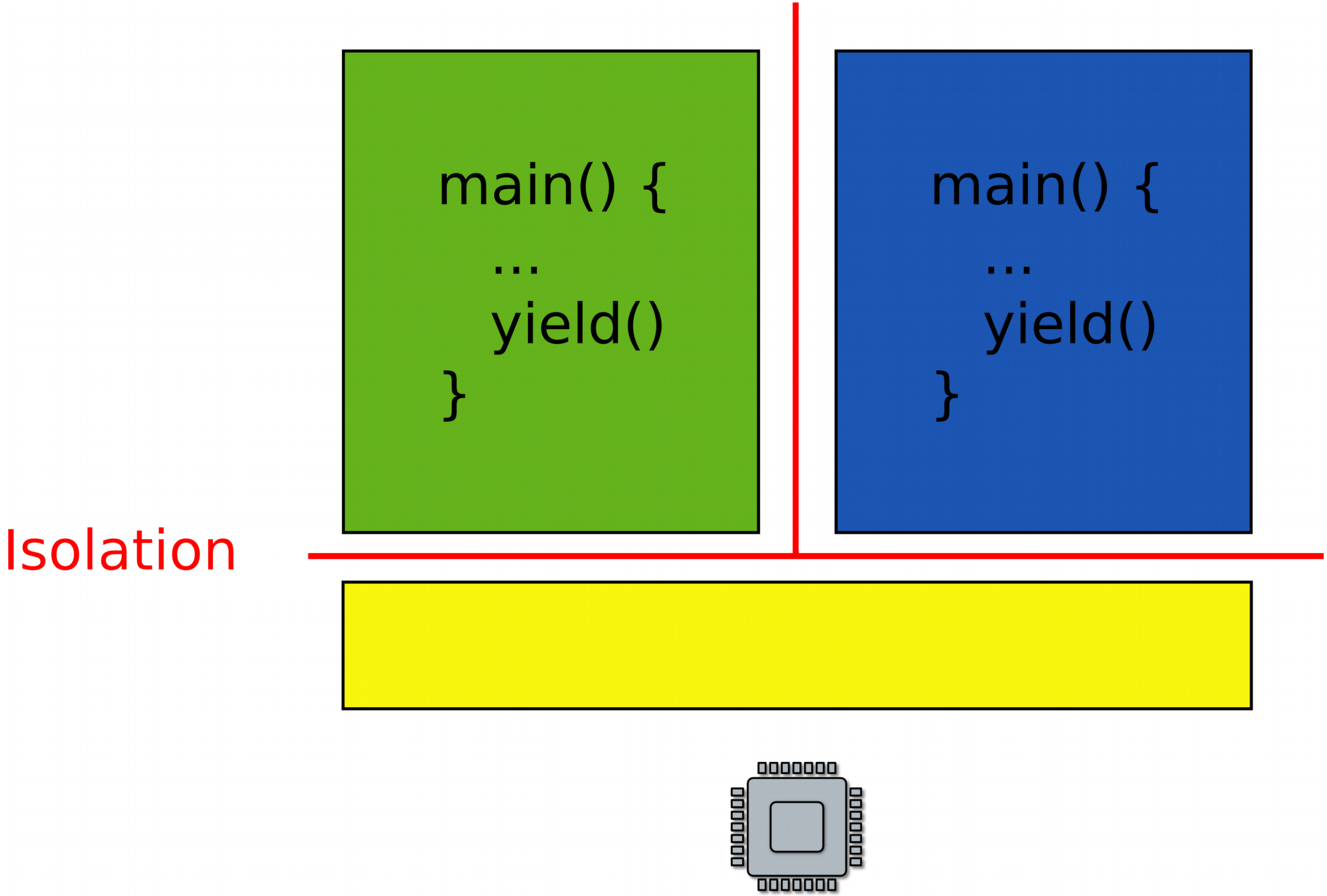
Timer

Scheduling

- Pick which application to run next
  - And for how long
- Illusion of a private CPU for each task
  - Frequent context switching

Isolation

- What if one faulty program corrupts the kernel?
- Or other programs?

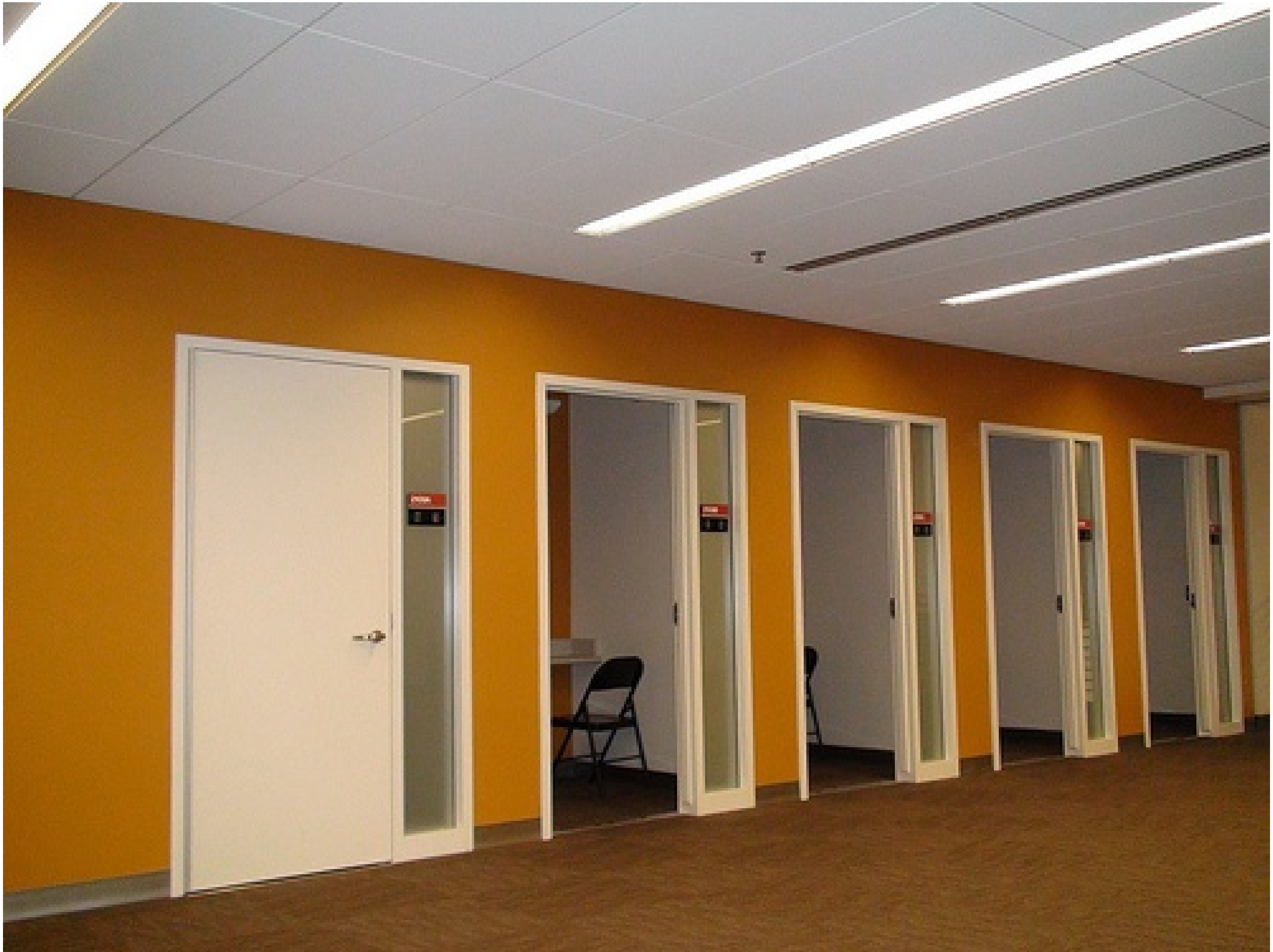


```
main() {
   ...
   yield()
}
```

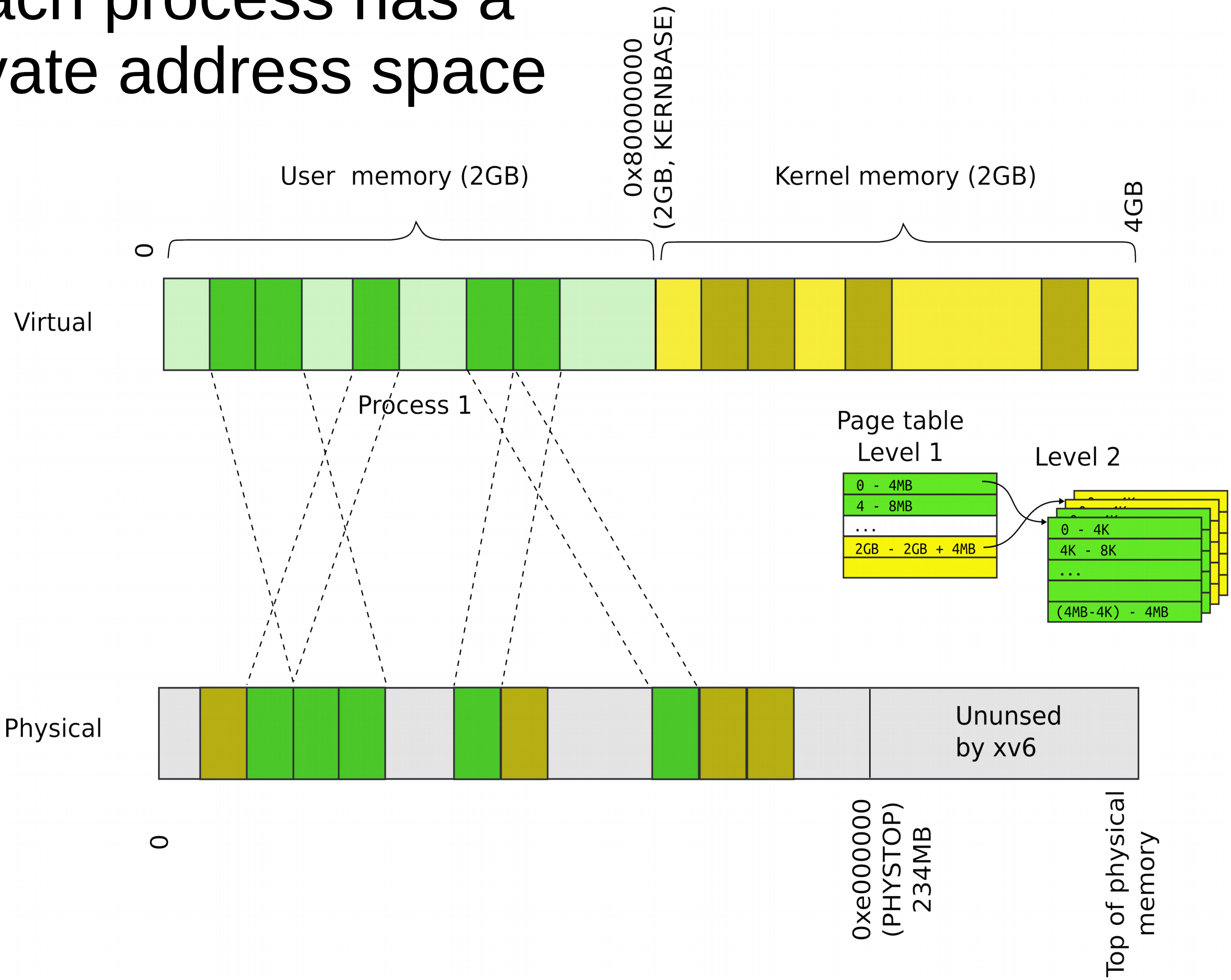
```
main() {
   ...
   yield()
}
```

No isolation: open space office

Isolated rooms

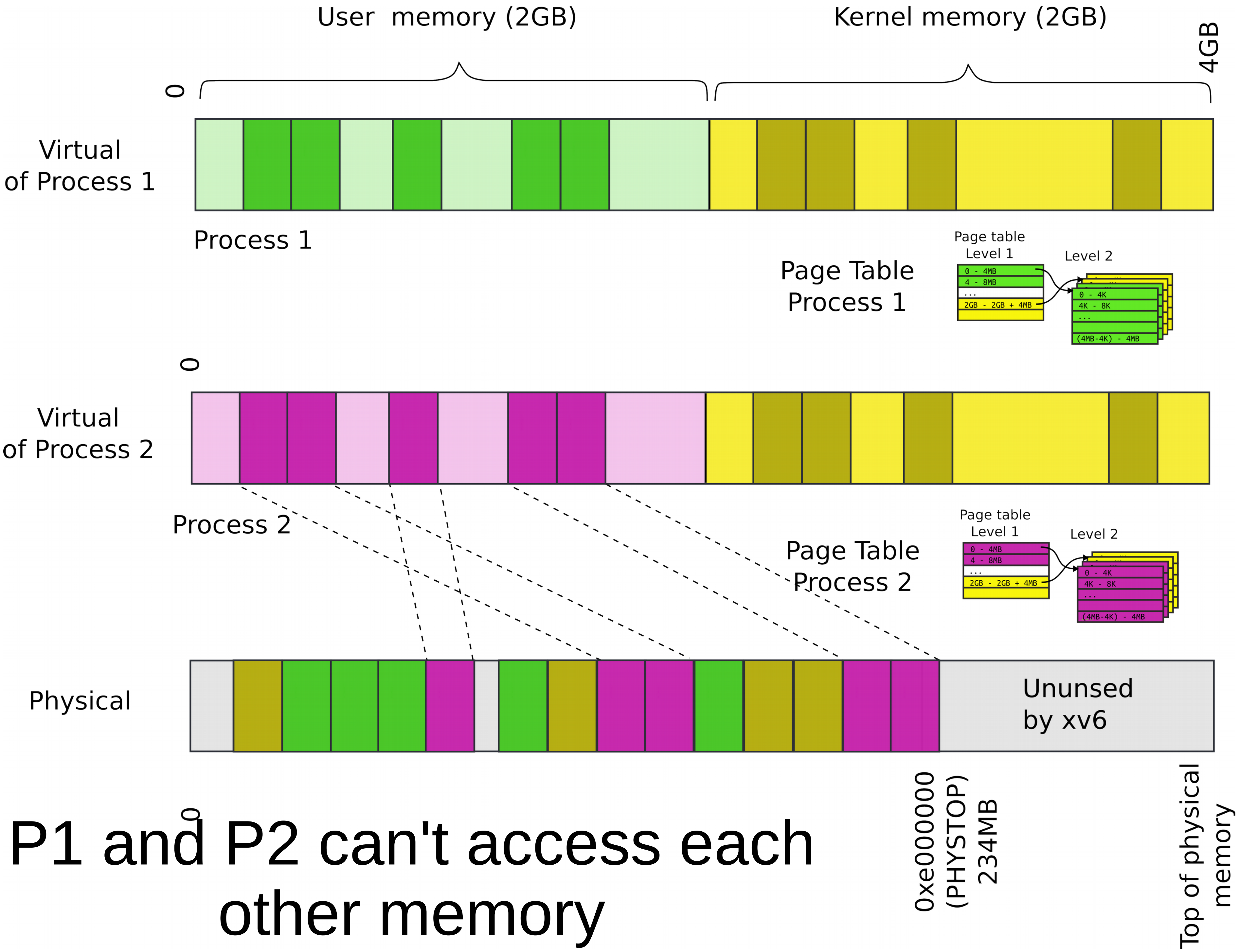


Each process has a private address space



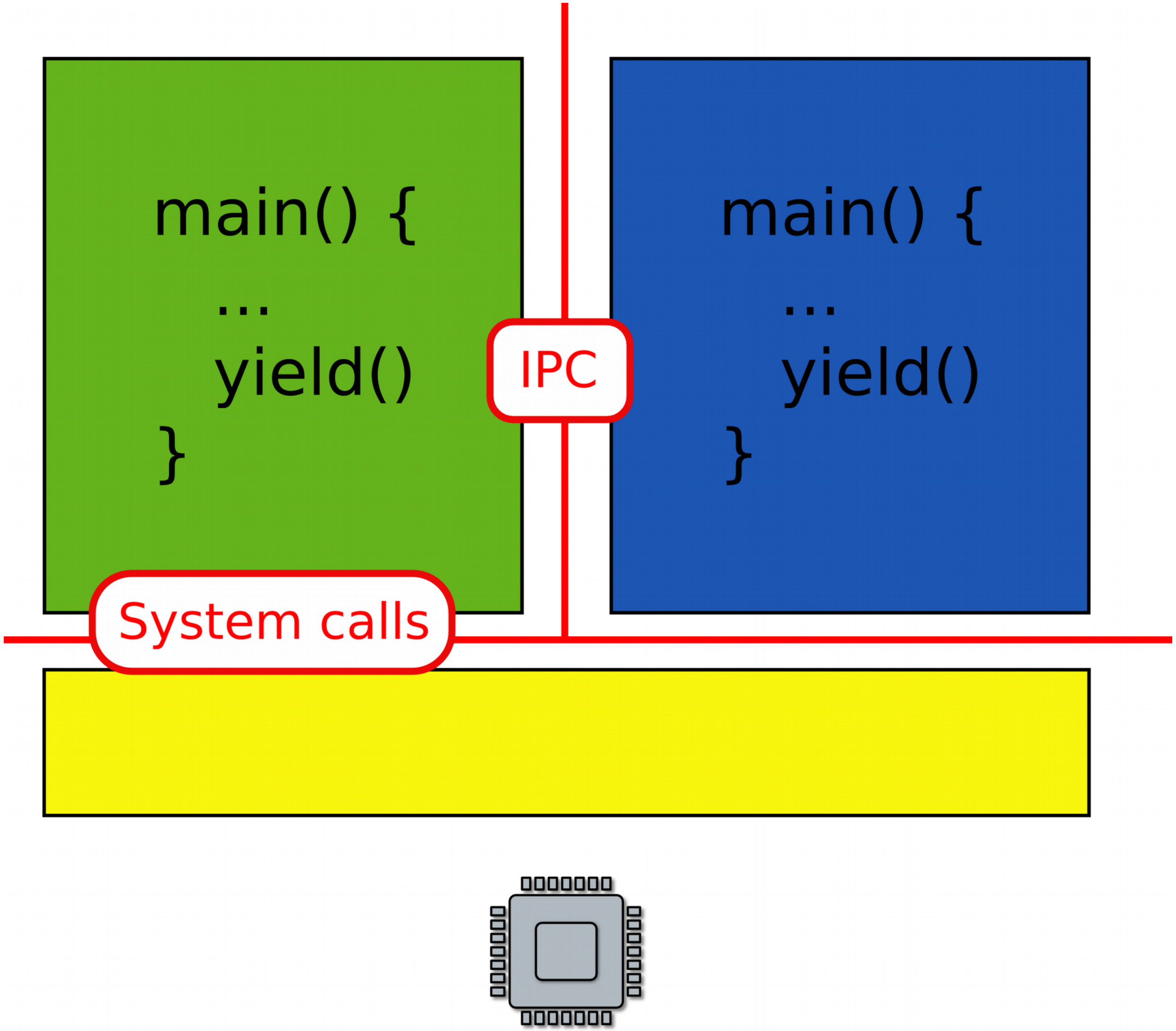
Each process maps the kernel

- It's not strictly required
  - But convenient for system calls
  - No need to change the page table when process enters the kernel with a system call
  - **Things are much faster!**



P1 and P2 can't access each other memory

- What about communication?
- Can we invoke a function in a kernel?



Files and network

- What if you want to save some data to a file?

- What if you want to save some data?
- Permanent storage
  - E.g., disks
- But disks are just arrays of blocks
  - wrtie(block_number, block_data)
- Files
  - High level abstraction for saving data
  - fd = open(“contacts.txt”);
  - fpritnf(fd, “Name:%s\n”, name);

Remember our console driver

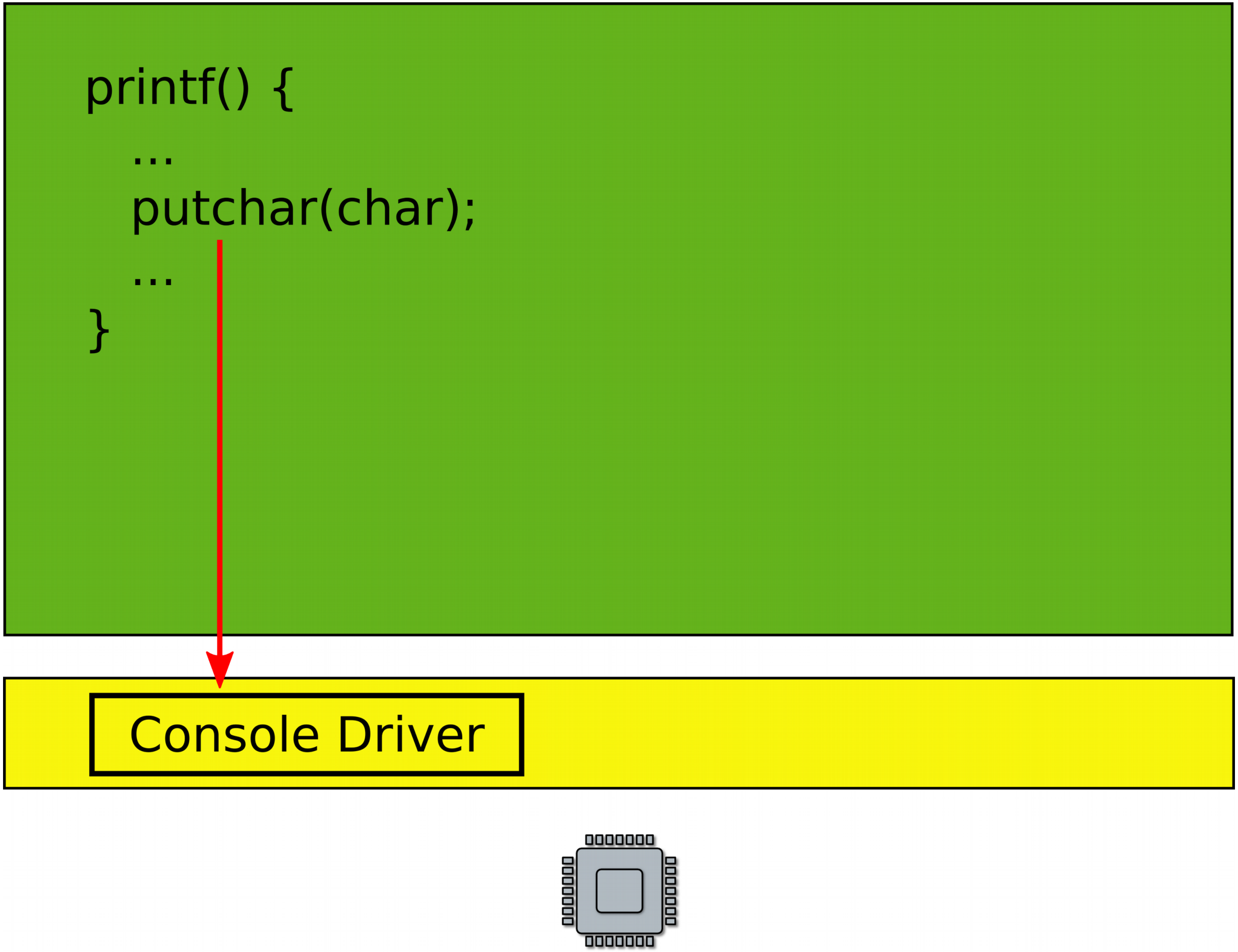
- Print a string on the screen or serial line

```
printf() {
    ...
    if (vga) {
        asm("mov <magic number 1>, char");
    } else if (serial) {
        asm("out <magic number 2>, char");
    }
    ...
}
```

OS

A more general interface

- First device driver



```
printf() {
   ...
   putchar(char);
   ...
}
```

Console Driver

- File system and block device provide similar abstractions
- Permanent storage
  - E.g., disks
- But disks are just arrays of blocks
  - wrtie(block_number, block_data)
- Files
  - High level abstraction for saving data
  - fd = open(“contacts.txt”);
  - fpritnf(fd, “Name:%s\n”, name);

File system and block layer

System calls	File descriptors
Pathnames	Recursive lookup
Directories	Directory inodes
Files	Inodes and block allocator
Transactions	Logging
Blocks	Buffer cache

- Reliable storage on top of raw disc blocks
- Disks are just arrays of blocks

```
wrtie(block_number, block_data)
```

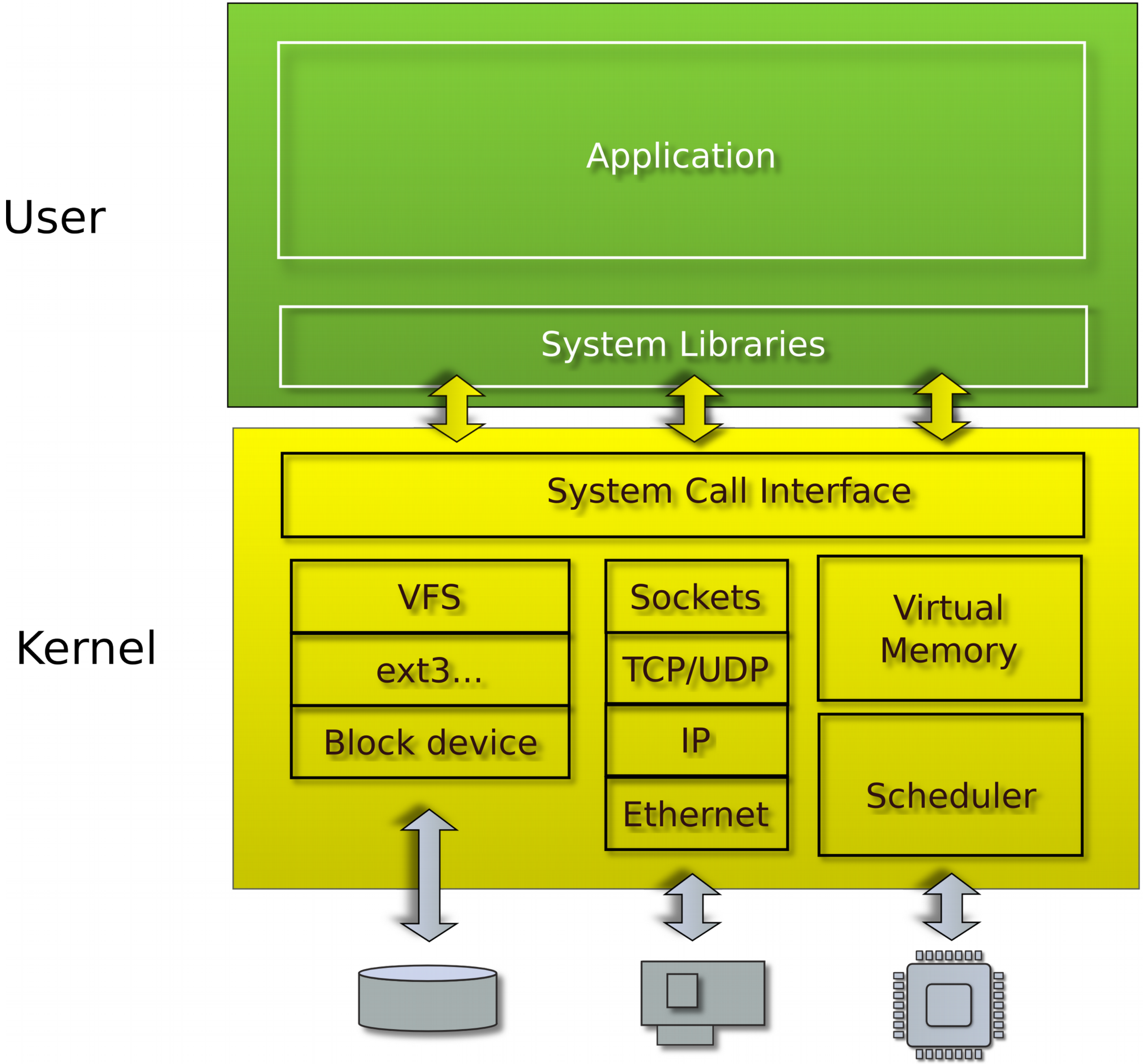
- Human readable names (files)
  - High level abstraction for saving data

```
fd = open("contacts.txt");
fpritnf(fd, "Name:%s\n",
name);
```

What if you want to send data over the network?

- Similar idea
  - Send/receive Ethernet packets (Level 2)
  - Two low level
- Sockets
  - High level abstraction for sending data

- Linux/Windows/Mac



User

Application

System Libraries

Kernel

System Call Interface

VFS

ext3...

Block device

Sockets

TCP/UDP

IP

Ethernet

Virtual Memory

Scheduler

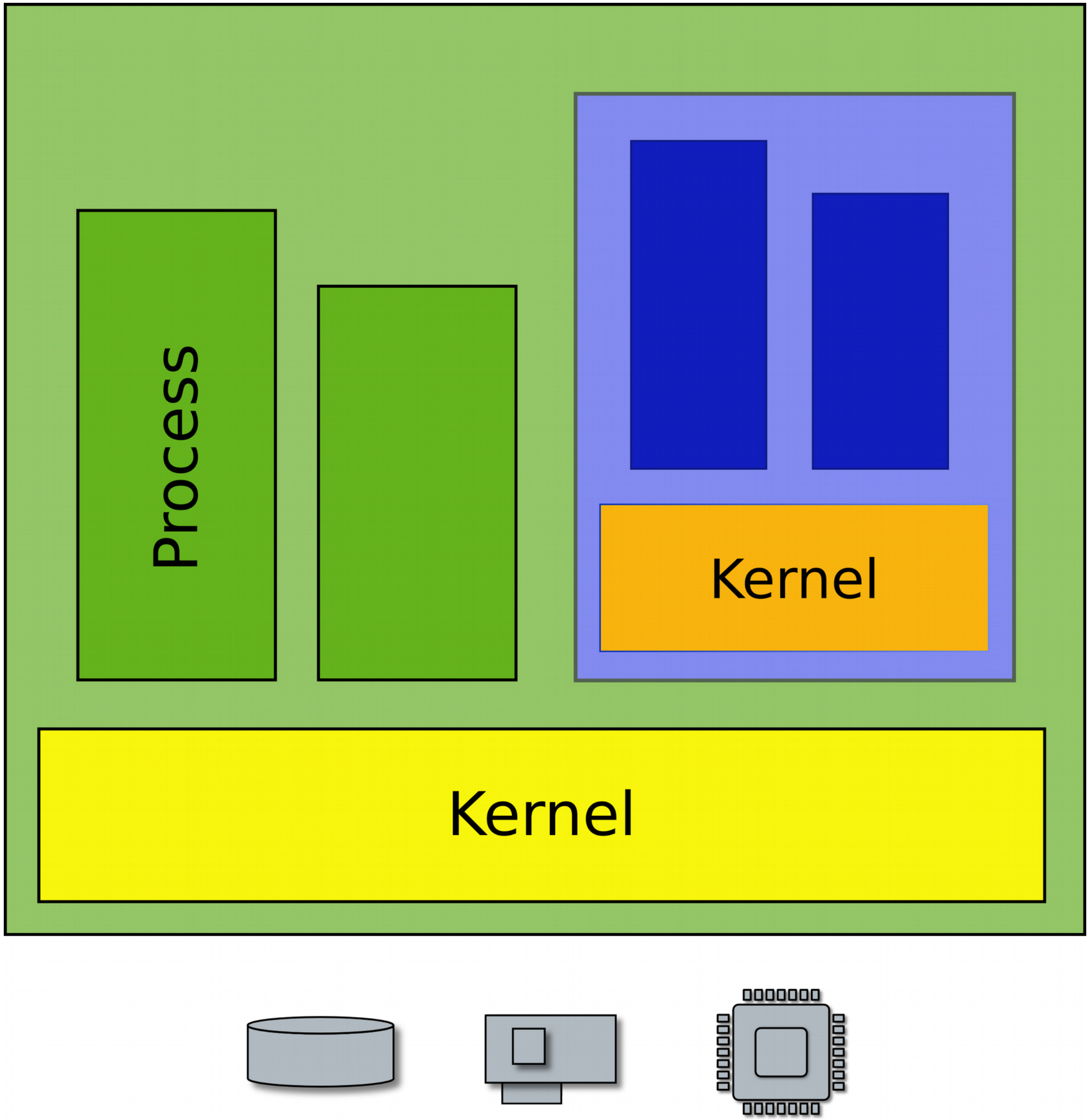
Recap

- Run multiple programs
  - Each has illusion of a private memory and CPU
    - Context switching
    - Isolation and protection
  - Management of resources
    - Scheduling (management of CPU)
    - Memory management (management of physical memory)
- High-level abstractions for I/O
  - File systems
    - Multiple files, concurrent I/O requests
    - Consistency, caching
  - Network protocols
    - Multiple virtual network connections

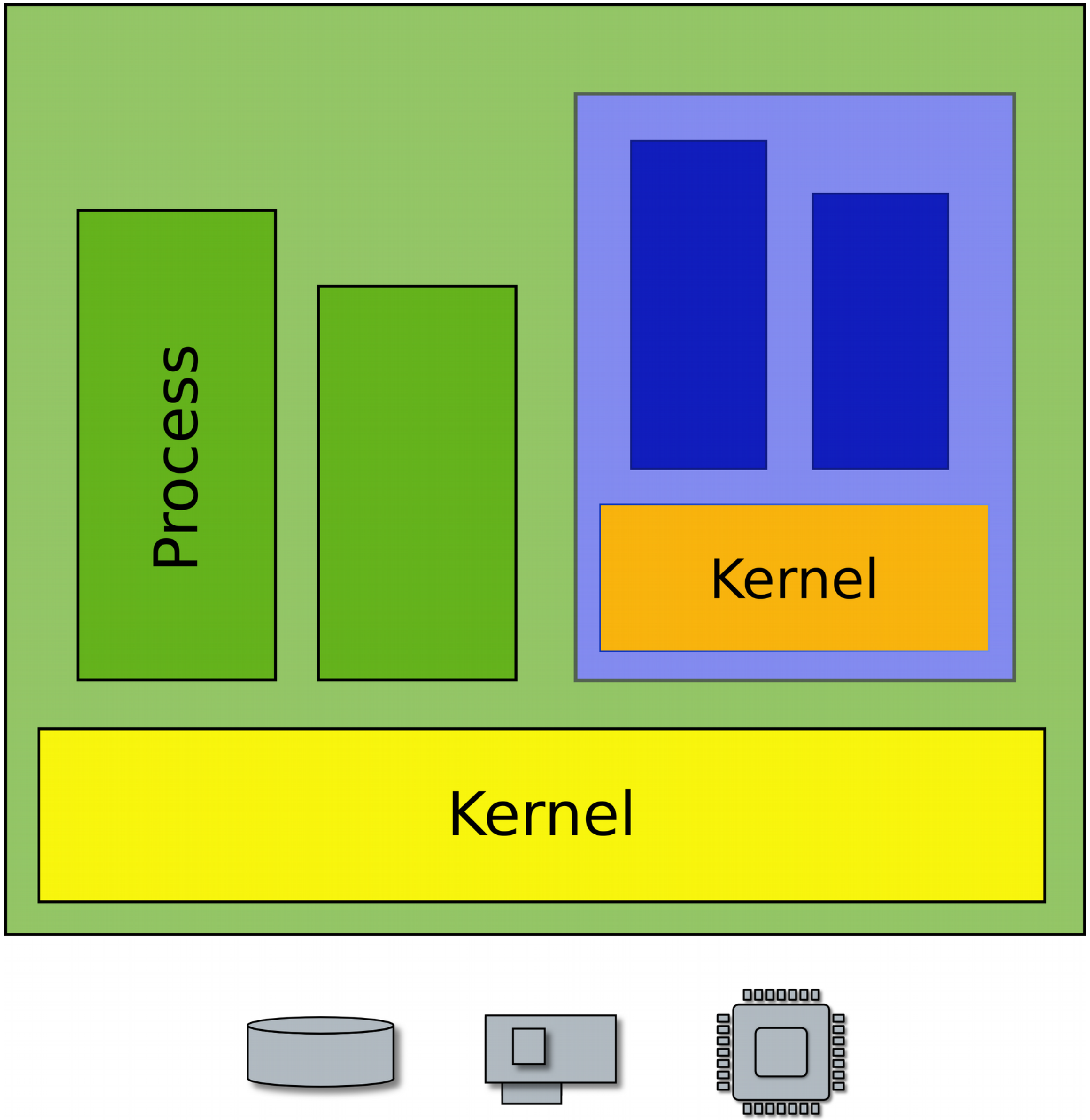
Questions?

Virtualization

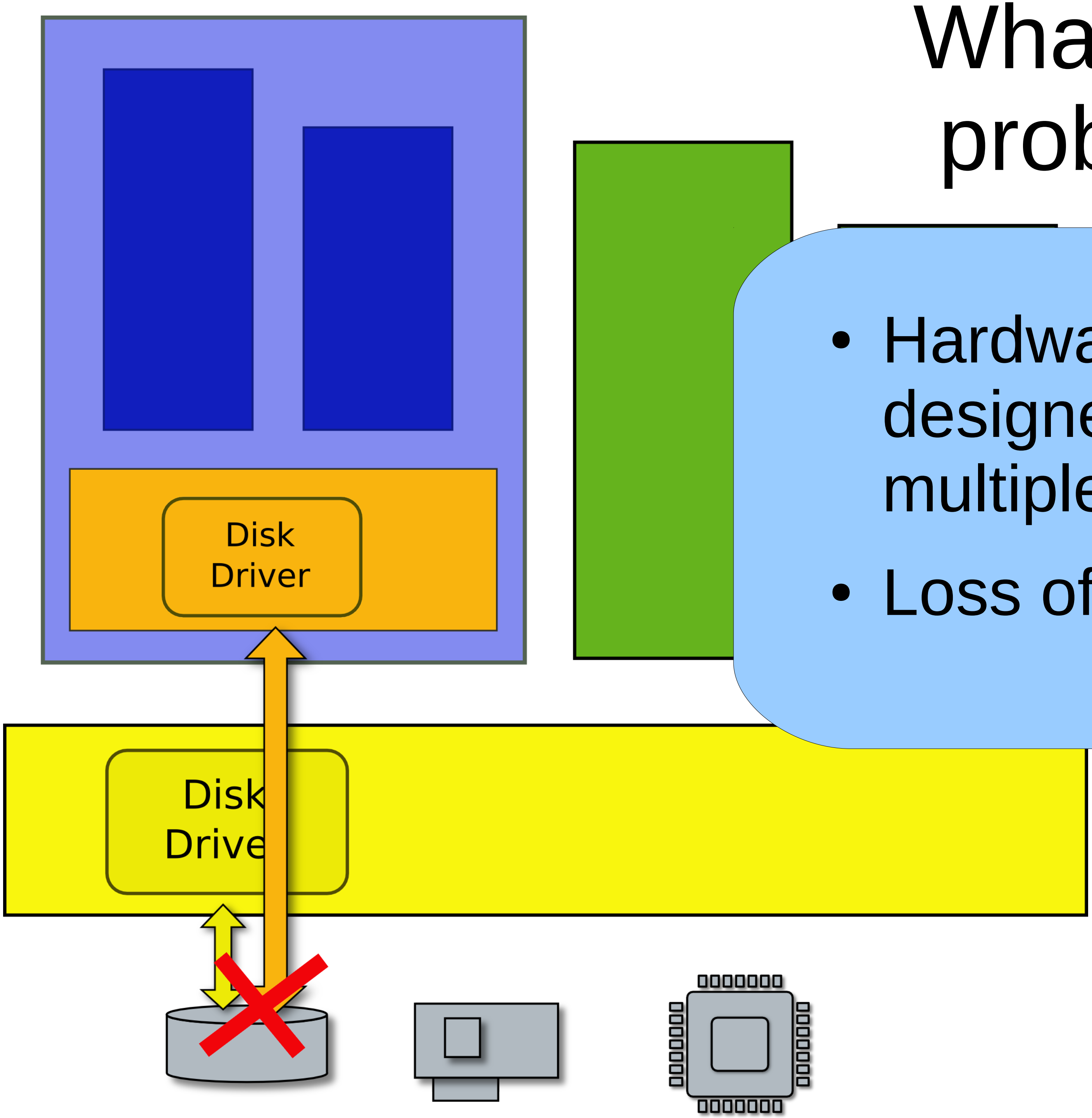
- Want to run a Windows application on Linux?



- Want to run a Windows application on Linux?

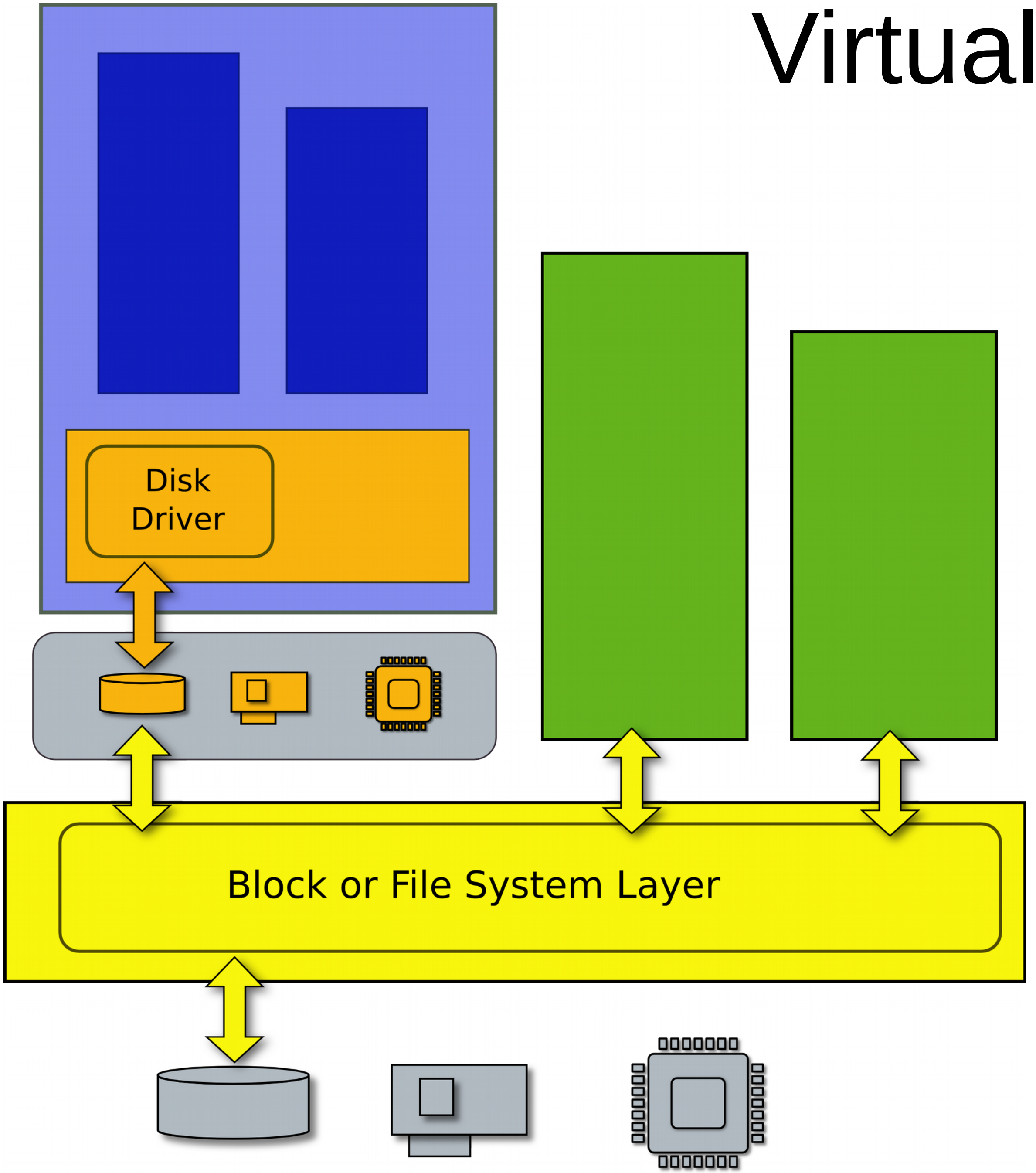


What is the problem?



- Hardware is not designed to be multiplexed
- Loss of isolation

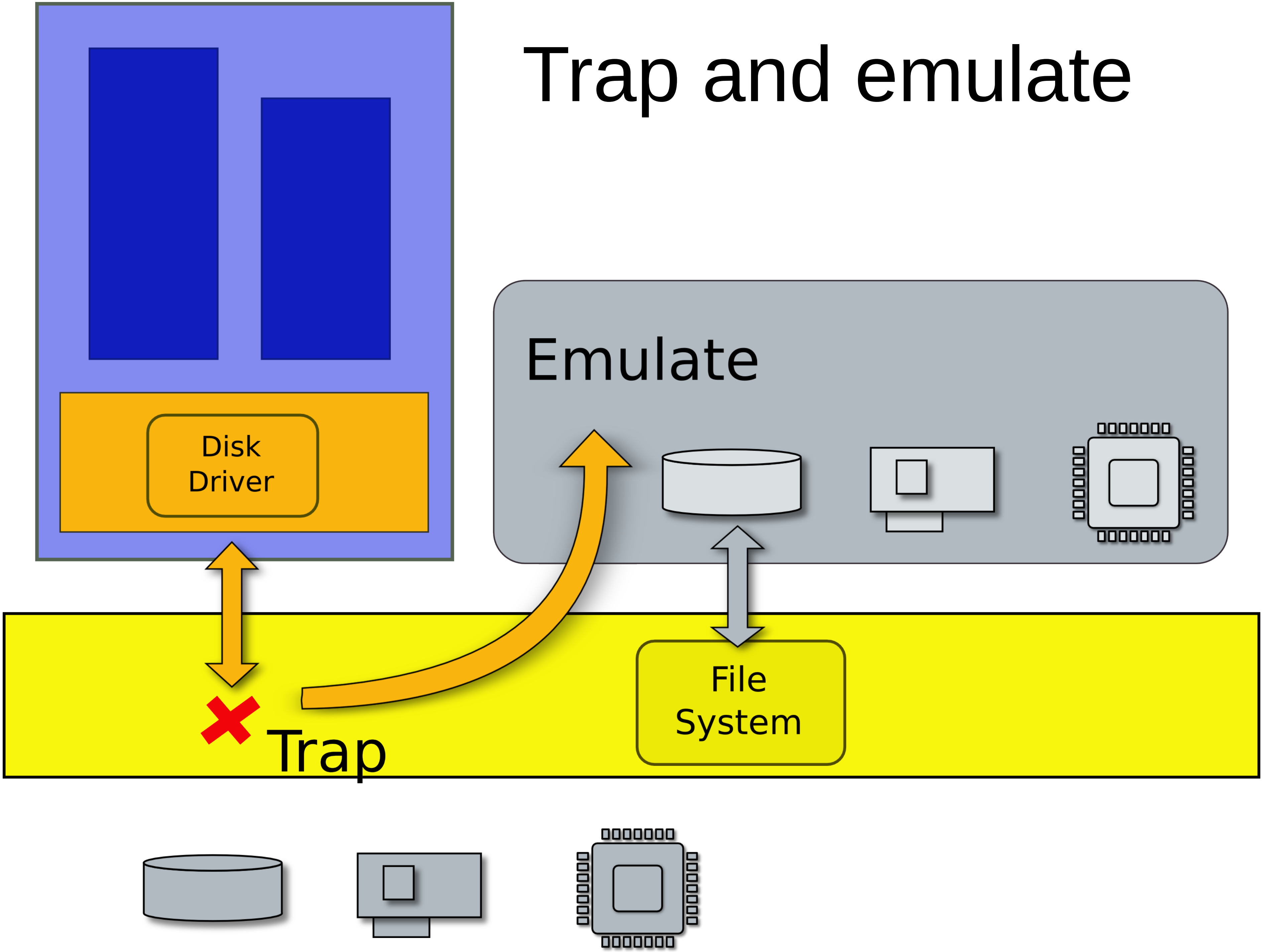
Virtual machine



Efficient duplicate of a real machine

- Compatibility
- Performance
- Isolation

Trap and emulate


Disk
Driver
Emulate
Trap
File
System