# 操作系统概论

## Section 1

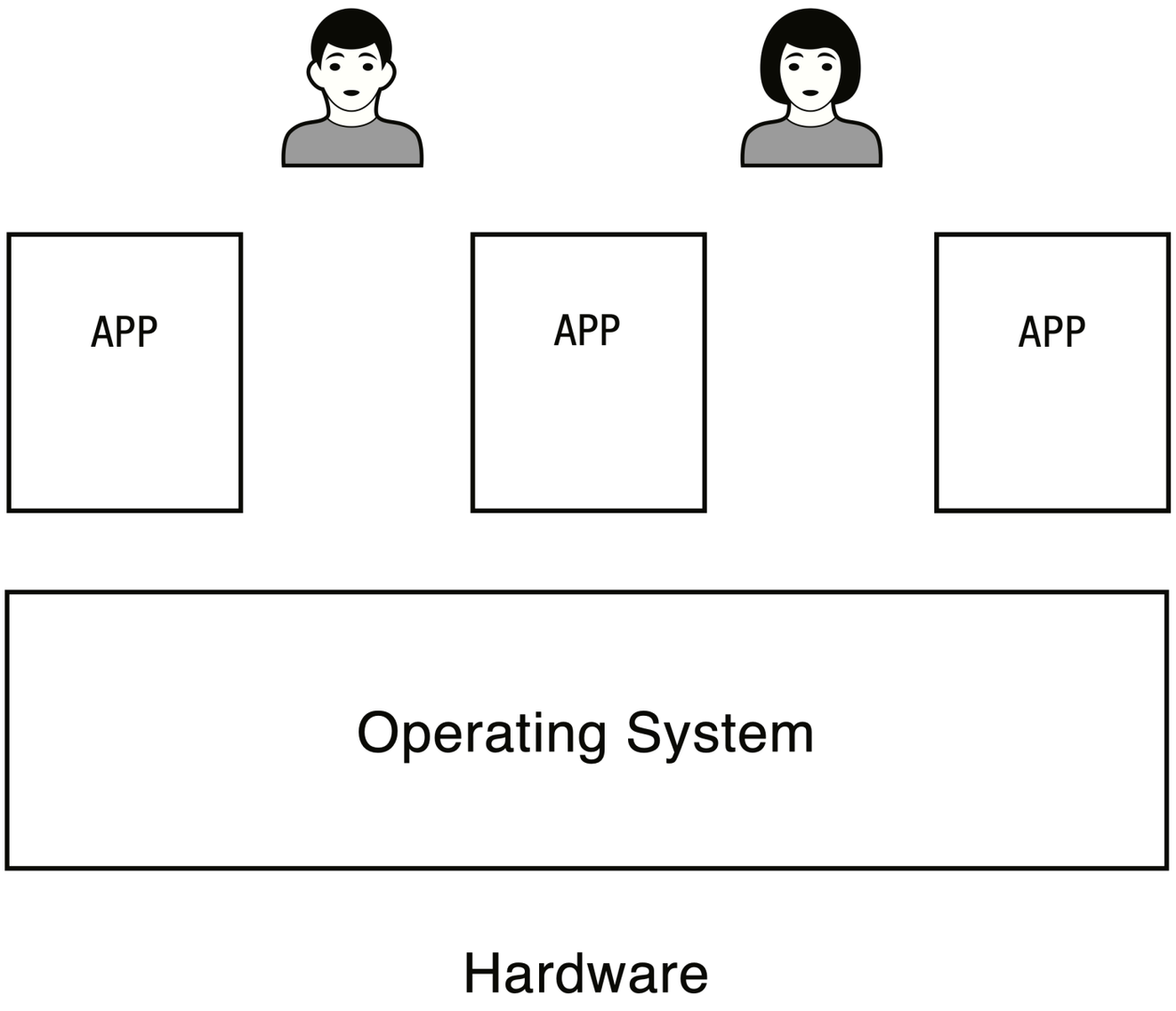
2025

# 什么是一个操作系统

## 作为应用启动器的操作系统

# 什么是一个操作系统

操作系统是位于计算机用户和硬件资源中间的一层软件系统，其目的在于对硬件进行管理和抽象，并为应用提供服务




应用程序因而能在更受限(防止损害)、更强大(克服硬件限制)和更有用(提供通用服务)的环境中运行 *(easy to use)*

为应用提供与设备无关的服务 *(device-independent)*

管理各种硬件设备 *(device-specific)*

# 软件和硬件的中间层

软件视角 (top-down view): 为应用程序提供硬件资源的易用抽象 (easy-to-use abstractions of physical hardwares)

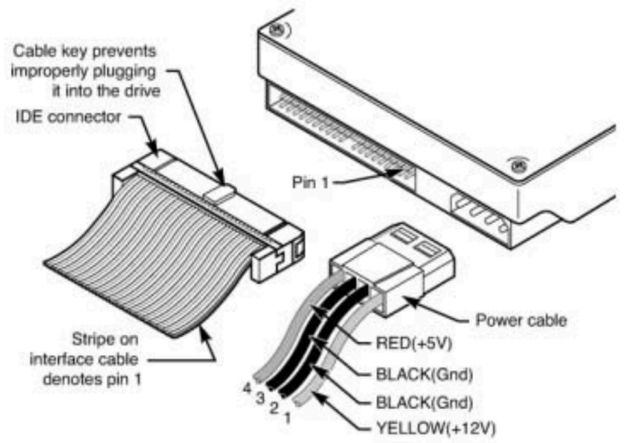
- 让软件的开发和运行 (计算机的使用) 变得更加容易
  - 如何在计算机中存储程序?
  - 如何让程序和外部设备交互?
  - 如何允许多个程序同时运行?
  - …

# 软件和硬件的中间层

软件视角 (top-down view): 为应用程序提供硬件资源的易用抽象 (easy-to-use abstractions of physical hardwares)

Hard Disk

没有操作系统，程序员将将直面底层硬件来编写程序




PATA (IDE) hard drive connectors

| Cmd Reg | Writes | Notes |
|---|---|---|
| Address | 7 0 | |
| 01F0 | Data | 16-bit accesses |
| 01F1 | Feature | Two 8-bit accesses |
| 01F2 | Sector Count | Two 8-bit accesses |
| 01F3 | LBA Low (31:24 then 7:0) | Two 8-bit accesses |
| 01F4 | LBA Middle (39:32 then 15:8) | Two 8-bit accesses |
| 01F5 | LBA High (47:40 then 23:16) | Two 8-bit accesses |
| 01F6 | Device | 8-bit access only |
| 01F7 | Command | 8-bit access only |
| Ctrl Reg | | |
| 03F6 | Device Control | 8-bit access only |

# 软件和硬件的中间层

软件视角 (top-down view): 为应用程序提供硬件资源的易用抽象 (easy-to-use abstractions of physical hardwares)

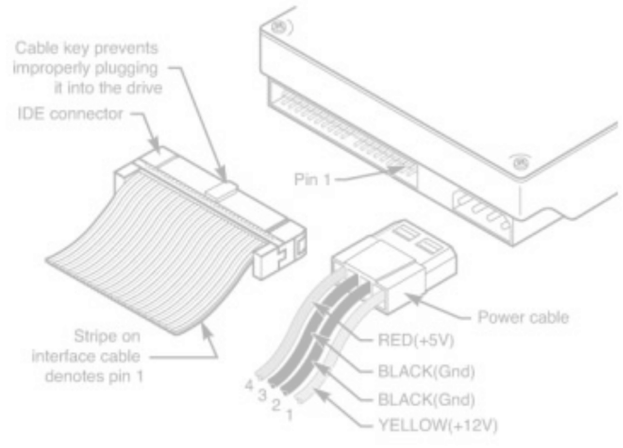

a.txt

```c
#include <unistd.h>

ssize_t read(int fd, void buf[.count], size_t count);
ssize_t write(int fd, const void buf[.count], size_t count);
```

允许程序员使用更加简单的概念

隐藏复杂、潜在不可靠的底层硬件细节




PATA (IDE) hard drive connectors

| Cmd Reg | Writes | Notes |
|---|---|---|
| Address | 7 0 | |
| 01F0 | Data | 16-bit accesses |
| 01F1 | Feature | Two 8-bit accesses |
| 01F2 | Sector Count | Two 8-bit accesses |
| 01F3 | LBA Low (31:24 then 7:0) | Two 8-bit accesses |
| 01F4 | LBA Middle (39:32 then 15:8) | Two 8-bit accesses |
| 01F5 | LBA High (47:40 then 23:16) | Two 8-bit accesses |
| 01F6 | Device | 8-bit access only |
| 01F7 | Command | 8-bit access only |
| Ctrl Reg | | |
| 03F6 | Device Control | 8-bit access only |

# 软件和硬件的中间层

硬件视角 (bottom-up view): 管理和分配计算机的硬件资源

- 硬件资源 (CPU、内存和磁盘等) 是有限的，计算机系统中的多个应用程序必须以合适的方式共享这些硬件资源
  - 允许多个程序同时运行 (sharing CPU)
  - 允许多个程序同时访问各自的指令和数据 (sharing memory)
  - 允许多个程序访问设备 (sharing disks)
  - 保护应用程序免受其他应用程序的影响并防止系统崩溃 (protection and isolation)
- 在资源的分配和使用过程中权衡需求、分离冲突、促进共享

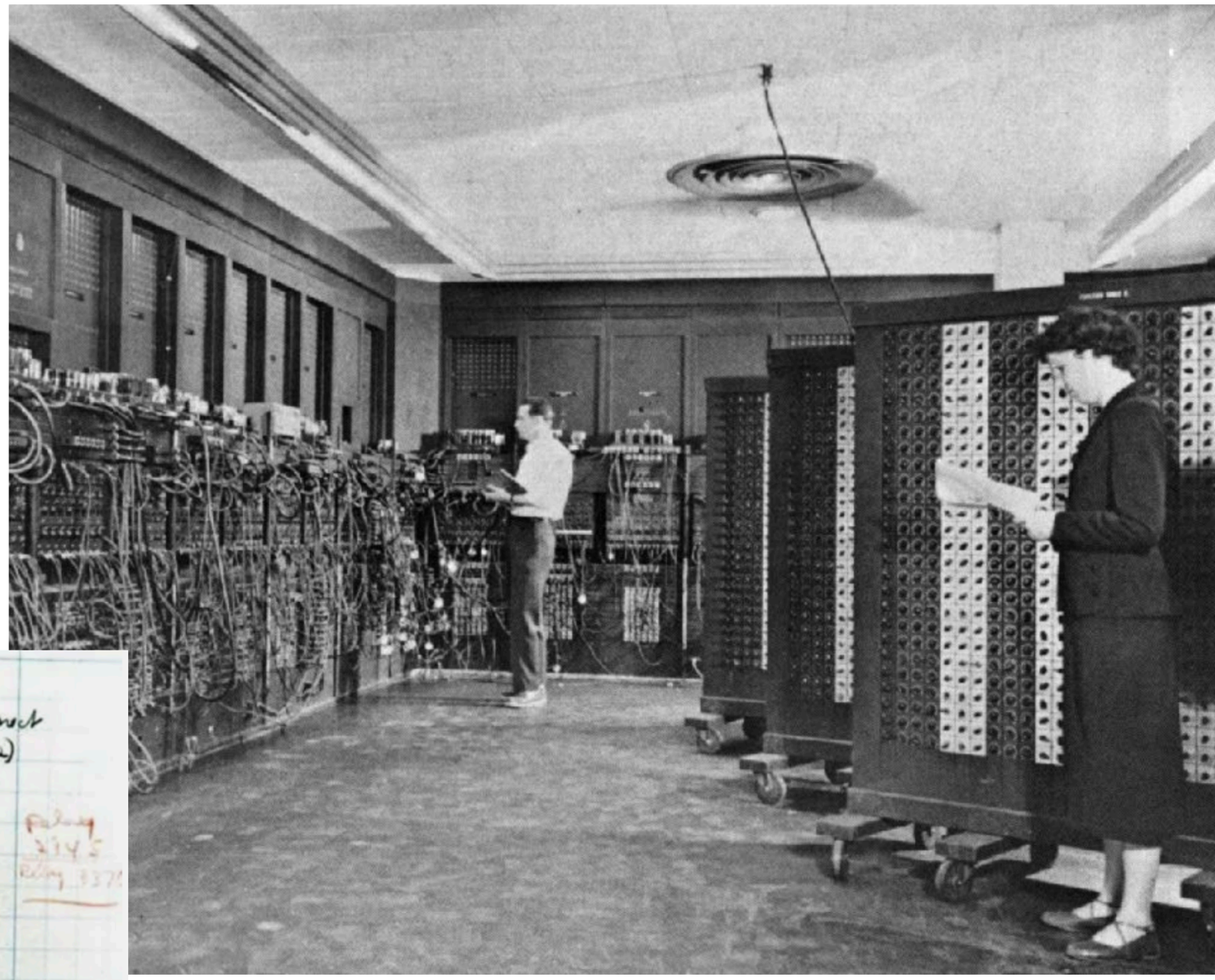
# 操作系统的发展

操作系统是目前人类开发的最为复杂的软件系统之一

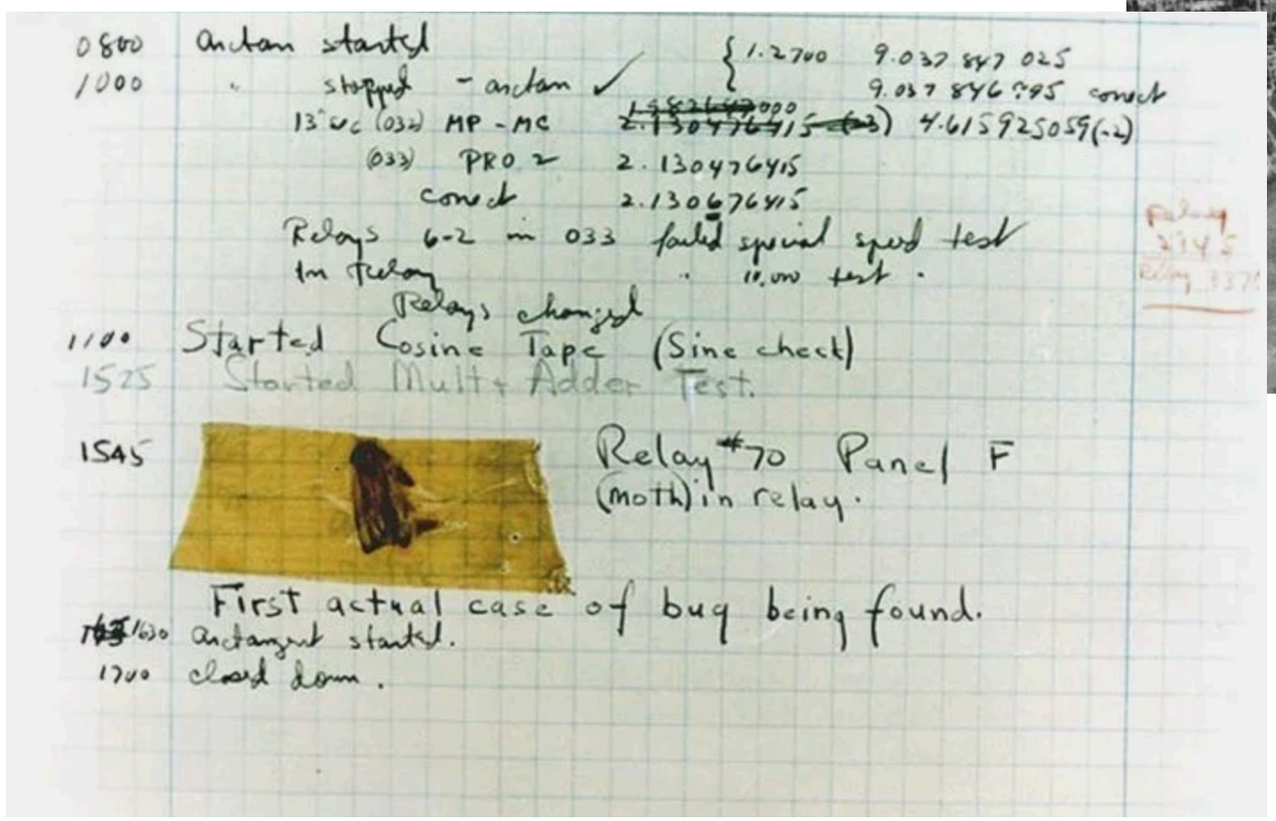
- 硬件的发展 ➔ 更复杂的软件
- 更复杂的软件 ➔ 硬件的发展

# 1945～1955: 真空管时代

- 只有用于计算的硬件
- 基于物理连线的编程、以及解决真正的“bugs”
- 没有操作系统，程序直接在硬件上运行



ENIAC (Electronic Numerical Integrator And Computer) 世界上第一台通用计算机



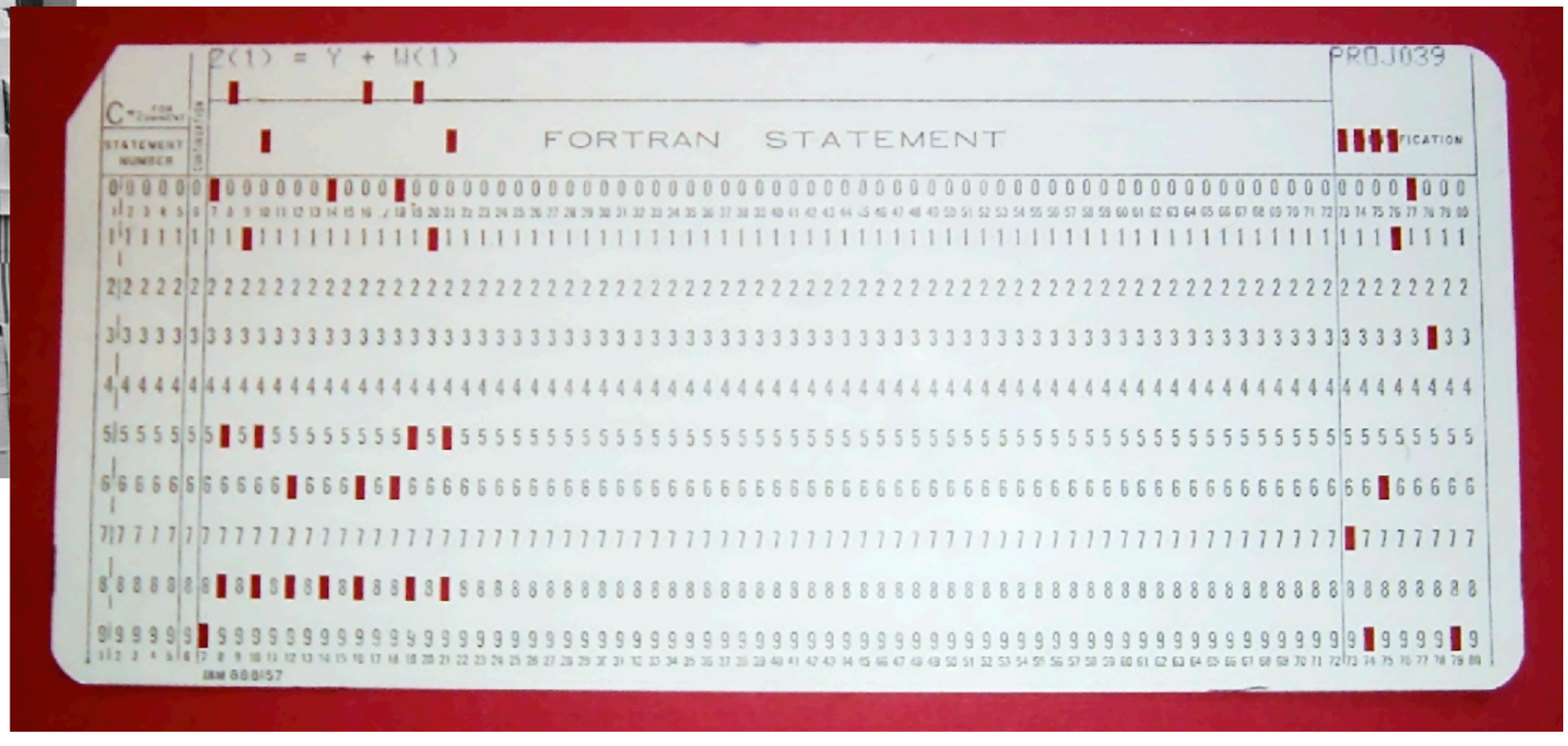
# 1955～1965: 晶体管时代

IBM 7094

- 更小、更快、更加可靠的逻辑门，更大的内存
- 更多用户想使用计算机

# 1955～1965: 晶体管时代

- 编程 (打孔) 开始变得相对容易
  - 一行代码、一张卡片
- 计算机非常昂贵，多用户轮流使用，并由操作员负责调度
- 开始需要管理计算机系统、并为用户提供更简单易用的系统模型




Fortran

# 1955～1965: 晶体管时代

- 操作系统 (that is, operating system jobs) 的概念开始形成
- 批处理系统 (batch system)
  - 维护一个任务队列，不断轮流加载和运行每一个任务
  - 用户提交任务，然后等待任务完成 (non-interactive use)
- 作为库函数的操作系统
  - 对硬件资源进行管理和抽象
  - 任务 (job)、文件 (file) 等概念开始出现

# 1965～1980: 集成电路时代

- 集成电路、总线的出现
  - 更快的 CPU
  - 更快/更大的内存
  - 更丰富的 I/O 设备
- 更多的高级编程语言
- 多用户希望同时使用计算机
  - 用户体验开始变得重要
  - 不仅是任务的完成时间，还有响应时间

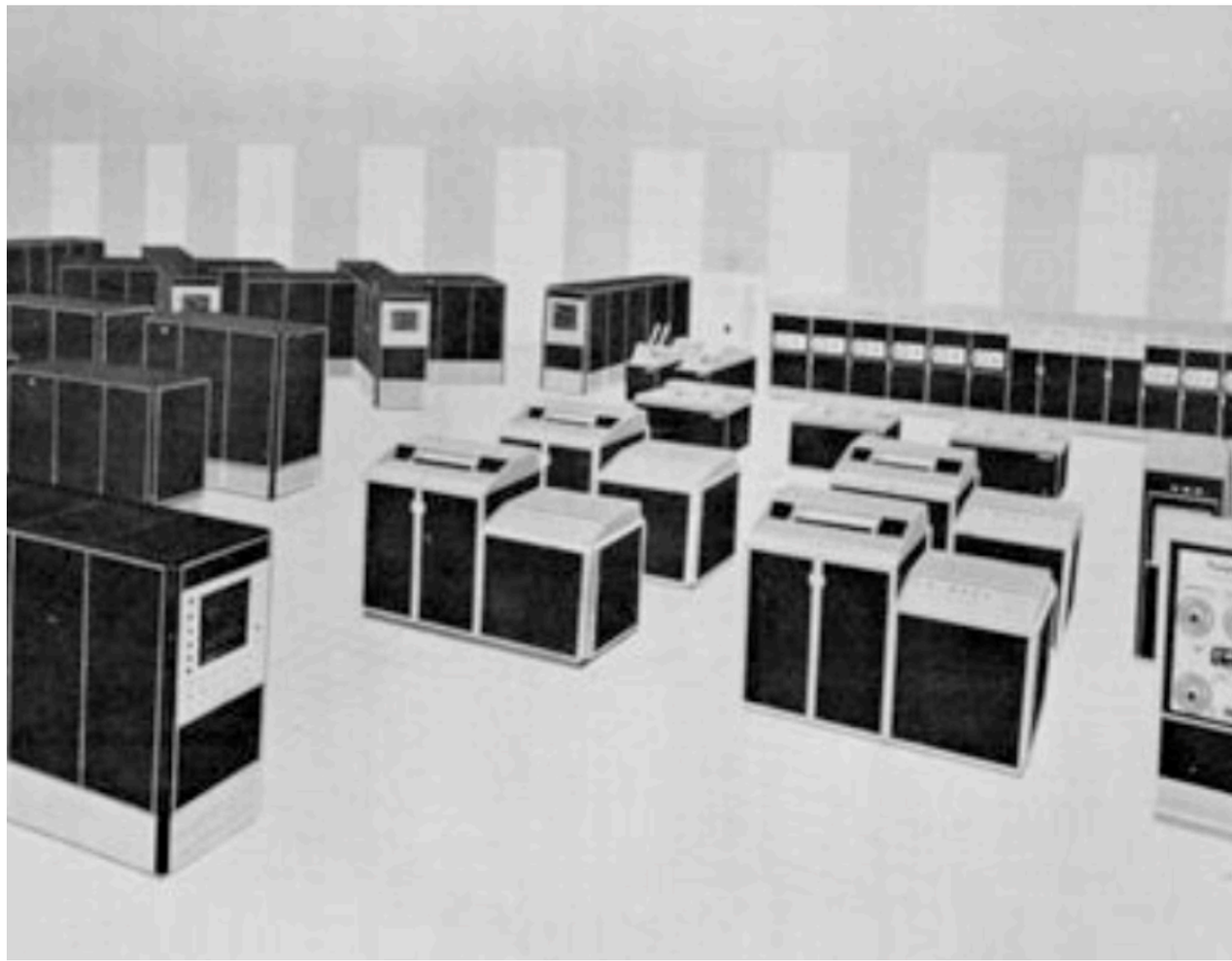

IBM System/360

# 1965～1980: 集成电路时代

- 多道程序设计 (multiprogramming)
  - 将多个程序同时载入内存，并在其中进行切换
  - 进程 (process) 的概念开始出现
  - 一个进程在执行 I/O 时可以将 CPU 让给另一个进程
- 分时系统 (time sharing)
  - 既然可以在应用程序之间切换，我们就可以通过定时切换来使得用户得到更快的响应 (interactive use)
  - 可以方便的支持多个用户同时使用某个计算机系统
- 对 CPU 和内存的保护、以及对并发的控制开始受到关注
  - 就需要防止不同程序之间的干扰

# 1965～1980: 集成电路时代

- Multics (Multiplexed Information and Computing Service) 分时操作系统
  - 1965s: MIT, Bell Labs 和 GE 提出了一个新概念 Computer Utility
  - 20 世纪末: 算力是一种服务 Cloud Computing
- Fernando J. Corbato 因组织和领导 Multics 的开发获得 1990 年图灵奖



“只需要一台超级计算机就可以为居住在波士顿的每个用户提供计算服务”

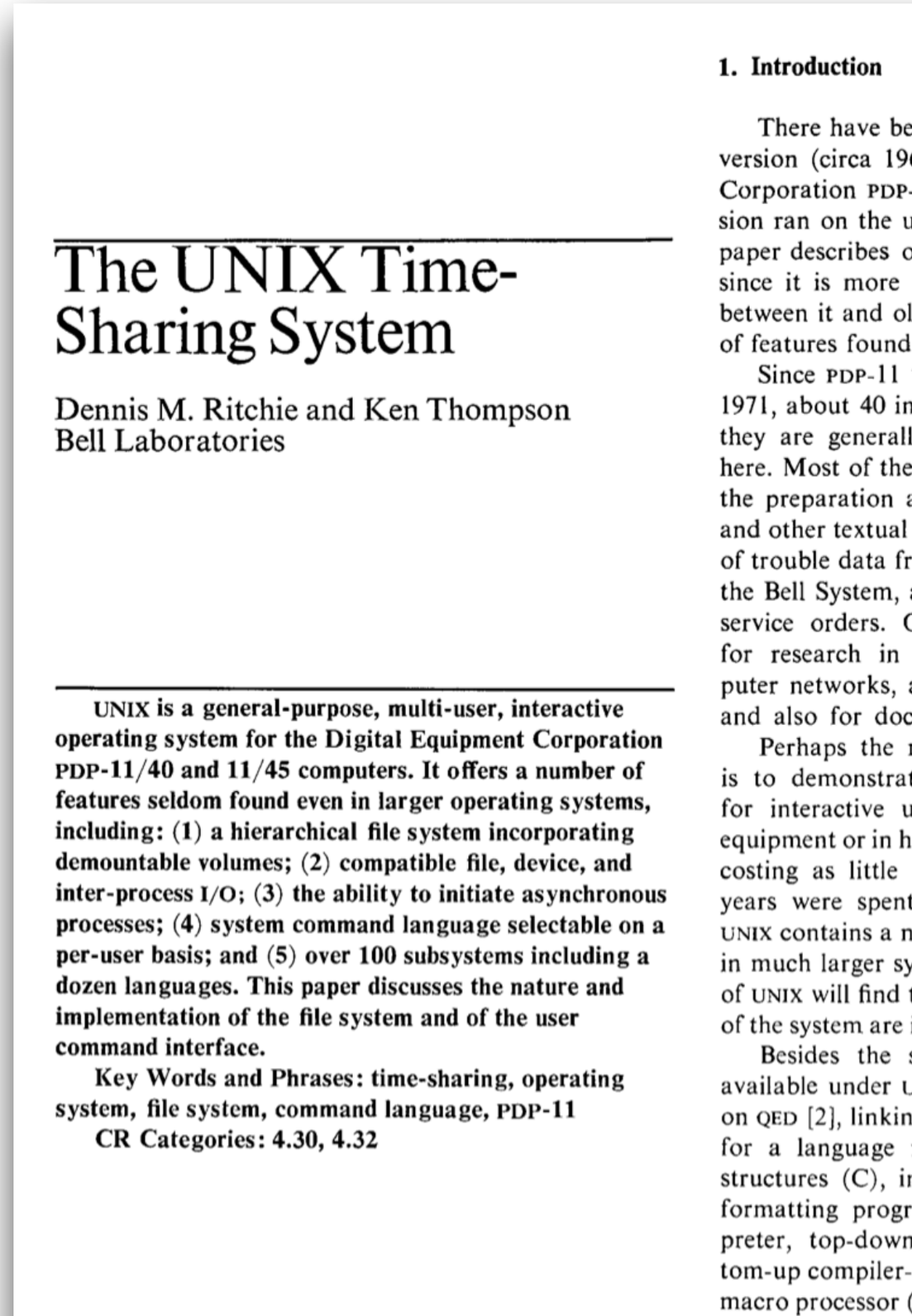
# 1965～1980: 集成电路时代

- 1970 年代 Unix 的诞生
  - 基于 Multics，Ken Thompson 和 Dennis Ritchie 在 Bell labs 开发了 Unix
  - 大部分基于 C 语言
  - 第一个可移植的操作系统
- 很多操作系统概念进一步完善，奠定了现代操作系统的形态
- Ken Thompson 和 Dennis Ritchie 因 Unix 操作系统和 C 语言获得 1983 年图灵奖

Unix on PDP-11

# 1965～1980: 集成电路时代

- Unix 具有高度模块化的设计，其设计理念影响深远
  - 操作系统应该提供简单的工具，其中每个工具应具有有限且明确定义的功能
- IEEE 基于 Unix 设计了 POSIX (Portable OS Interface for uniX) 标准
  - 规定了类 Unix 系统需要提供的标准 APIs
  - 如今大部分操作系统 (不只是类 Unix 系统) 都支持这一标准



The UNIX Time-Sharing System

Dennis M. Ritchie and Ken Thompson
Bell Laboratories

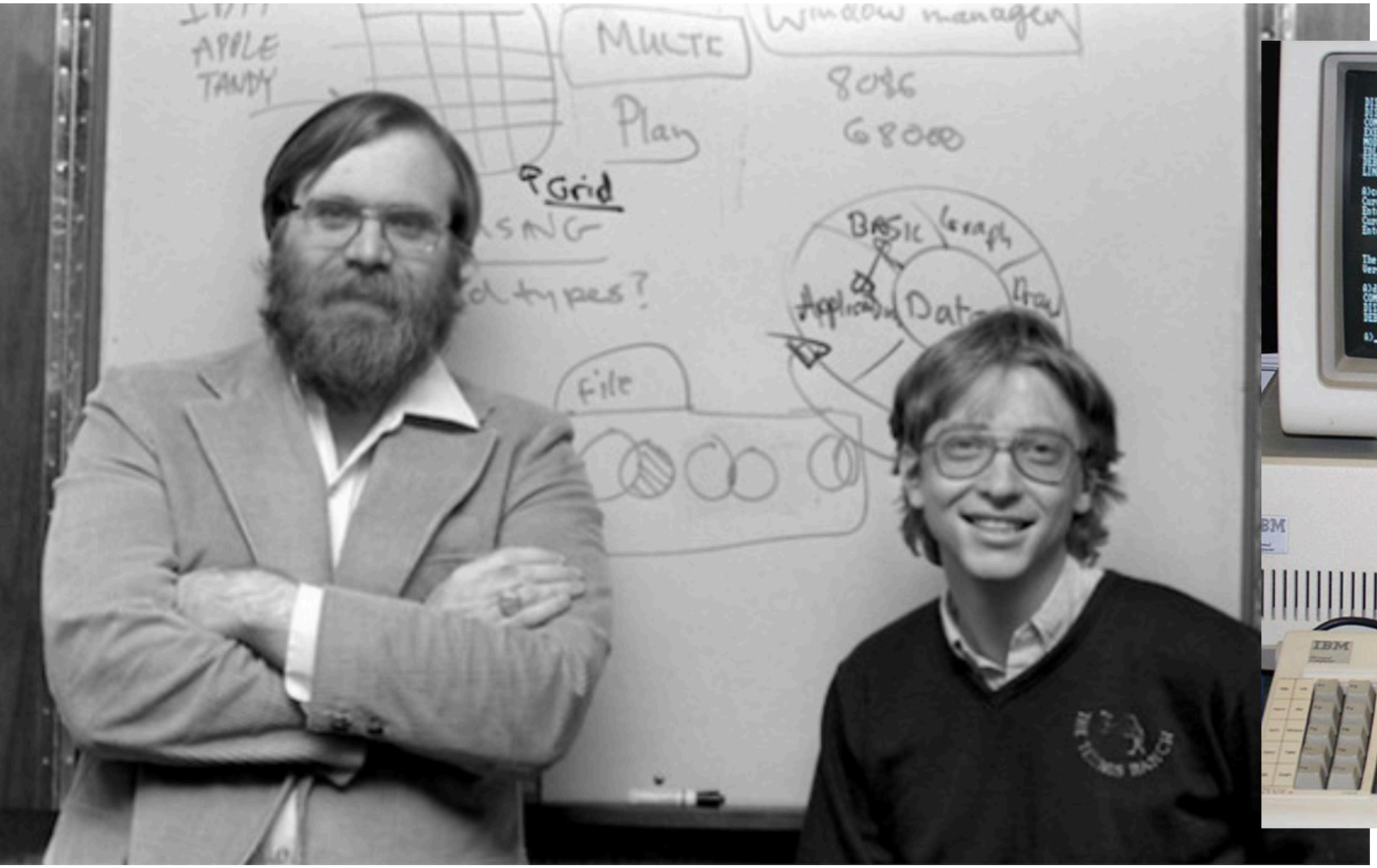
UNIX is a general-purpose, multi-user, interactive operating system for the Digital Equipment Corporation PDP-11/40 and 11/45 computers. It offers a number of features seldom found even in larger operating systems, including: (1) a hierarchical file system incorporating demountable volumes; (2) compatible file, device, and inter-process I/O; (3) the ability to initiate asynchronous processes; (4) system command language selectable on a per-user basis; and (5) over 100 subsystems including a dozen languages. This paper discusses the nature and implementation of the file system and of the user command interface.

Key Words and Phrases: time-sharing, operating system, file system, command language, PDP-11
CR Categories: 4.30, 4.32

1. Introduction

There have be
version (circa 19
Corporation PDP-
sion ran on the u
paper describes o
since it is more
between it and ol
of features found
Since PDP-11
1971, about 40 in
they are generall
here. Most of the
the preparation
and other textual
of trouble data fr
the Bell System,
service orders. C
for research in
puter networks,
and also for doc
Perhaps the
is to demonstrat
for interactive u
equipment or in h
costing as little
years were spent
UNIX contains a n
in much larger sy
of UNIX will find
of the system are
Besides the
available under U
on QED [2], linkin
for a language
structures (C), in
formatting progr
preter, top-down
tom-up compiler-
macro processor

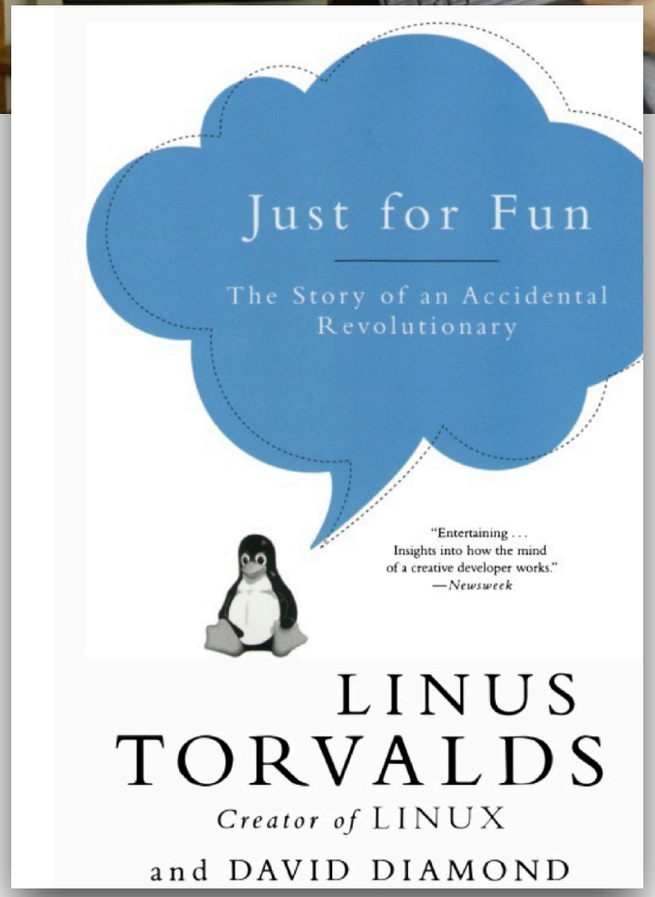
# 1980～Present: 个人电脑时代





- 大规模集成电路高速发展
- 每个桌面一台计算机 (IBM PC 和 MS-DOS)
  - 更便宜、更快捷、并且为大众设计

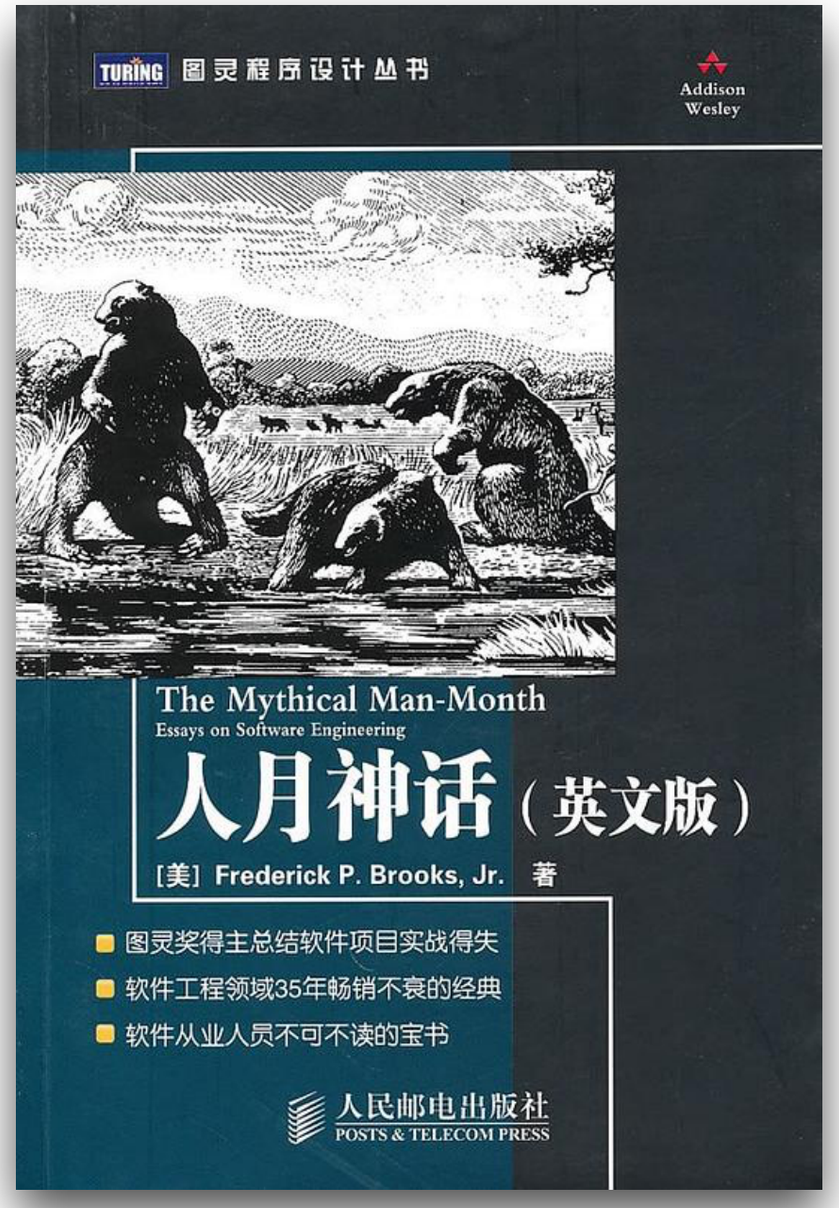
# 1980～Present: 个人电脑时代


- Unix 的简洁内核设计衍生出很多后续操作系统，与此同时很多大公司纷纷宣称对它拥有所有权并从中获利
- Linux (1991) 和开源软件
  - Linus 编写了自己的 Unix 版本，该版本大量借鉴了 Unix 的设计思想，但没有借鉴原始代码库，从而避免了版权问题
- 开源软件: Free to use

# 操作系统的发展

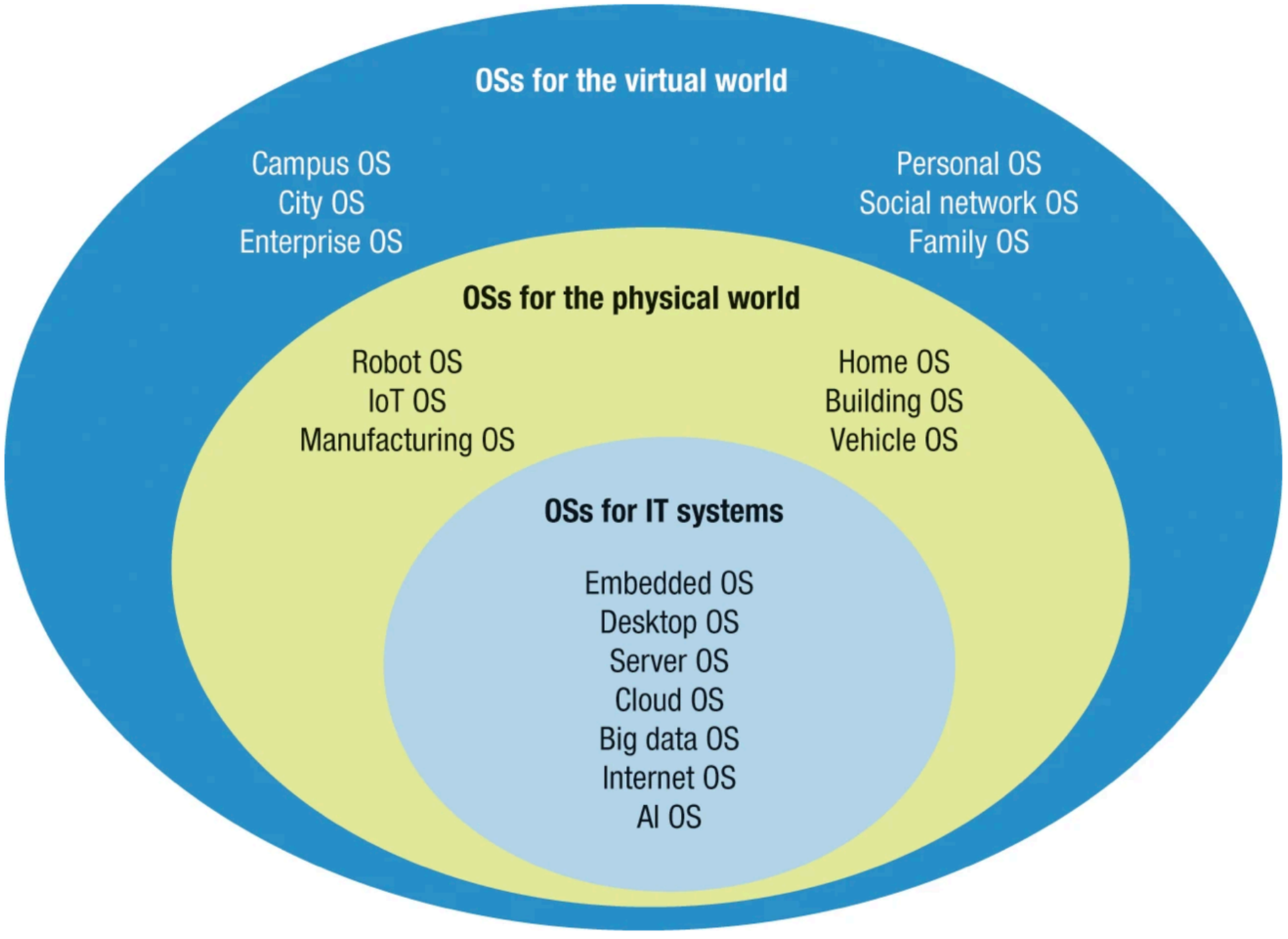
- 目前的商业操作系统已经不是一个人能够写出来的了
- 开发复杂系统中软件工程思维很重要
  - Frederick P. Brooks 在 1960s 主持和领导了 IBM System/360 系列计算机和操作系统的开发工作
  - 因在计算机结构、操作系统和软件工程方面里程碑式的贡献获得 1999 年图灵奖



# 如今的操作系统

- 更加高效 (复杂) 的硬件设备
  - 非对称多处理器 (symmetric multiprocessing)
  - 非均匀内存访问 (non-uniform memory access)
  - 从通用计算到领域计算 (GPU / TPU / NPU, …)
- 更加多样化和定制化的运行环境
  - PC 操作系统
  - 移动操作系统
  - 嵌入式操作系统
  - 物联网操作系统
  - …

# 泛在操作系统




Toward Ubiquitous Operating Systems: A Software-Defined Perspective. IEEE Computer, 2018

# 操作系统需要具备的能力

一个通用操作系统是如何支撑起我们耳熟能详的各种应用场景?

- **管理资源** (Sharing Resources)
  - 资源的分配和共享，应用程序的启动、切换、调度和销毁
  - 故障的隔绝和保护
- **构建幻象** (Masking Limitations)
  - 对物理硬件进行抽象以简化应用程序设计
  - 通过虚拟化克服有限硬件资源的限制
- **提供服务** (Providing Services)
  - 为应用程序提供一套常用和标准的 APIs

# 操作系统中的抽象

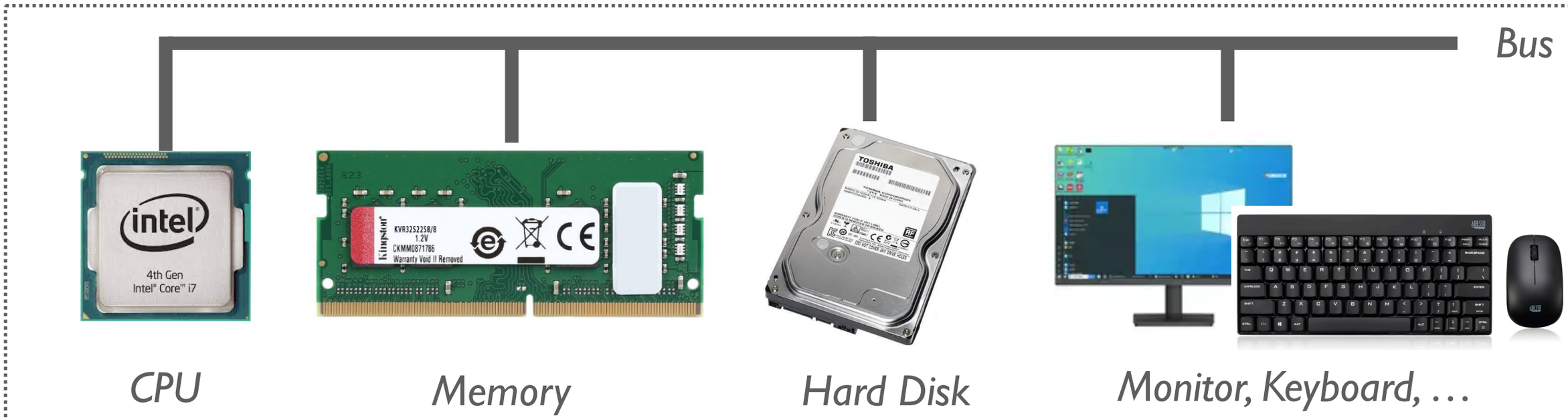
隐藏复杂对象不必要的属性、添加新能力、并以合适的方式进行组织

- 简化接口：形成复杂对象的一种简化表示形式 (Abstraction)
- 资源复用：将物理资源转化为多个虚拟资源实例 (Virtulization)
  - 为每个应用程序提供一个易用的 "virtual machine"
    - All alone: 资源的独占使用
    - All powerful: 潜在无限的资源
    - All expressive: 更强大的能力

# 操作系统中的抽象

**线程**
*Thread*

一个指令执行序列
(给每个应用程序一个 CPU)




计算机硬件

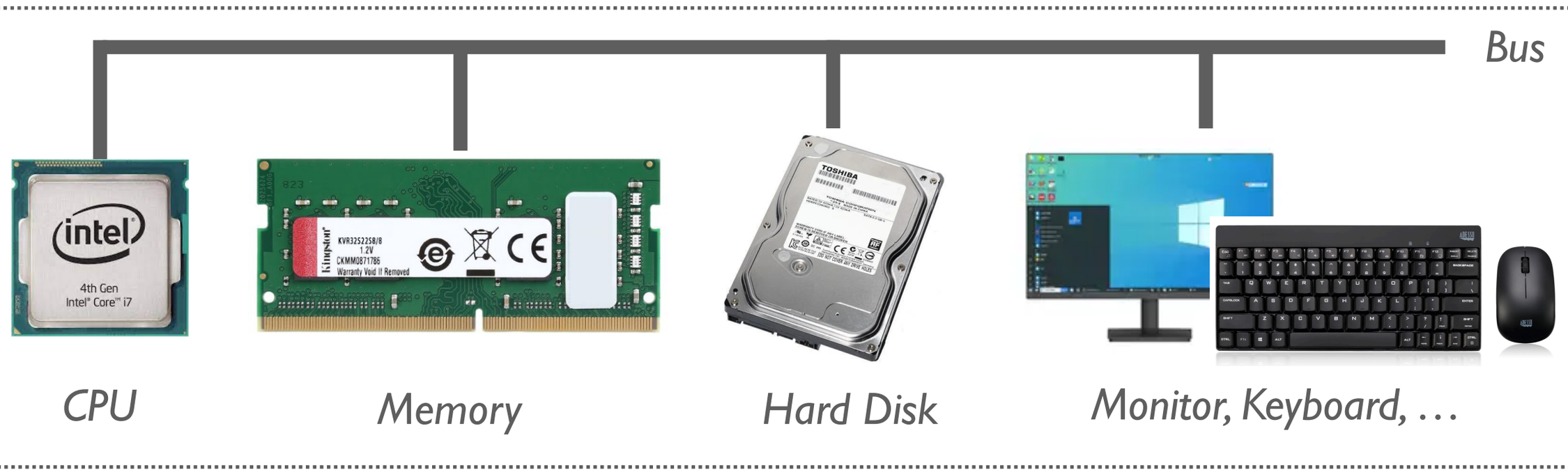
Reality: 多个执行序列在
有限个 CPU 上交替执行

# 操作系统中的抽象

**线程**
*Thread*

**地址空间**
*Address Space*

一个私有、连续、充足的寻址范围
(给每个应用程序一个 "无限大" 的内存条)




计算机硬件

Reality: 多个应用程序共享
一个有限大小的物理内存

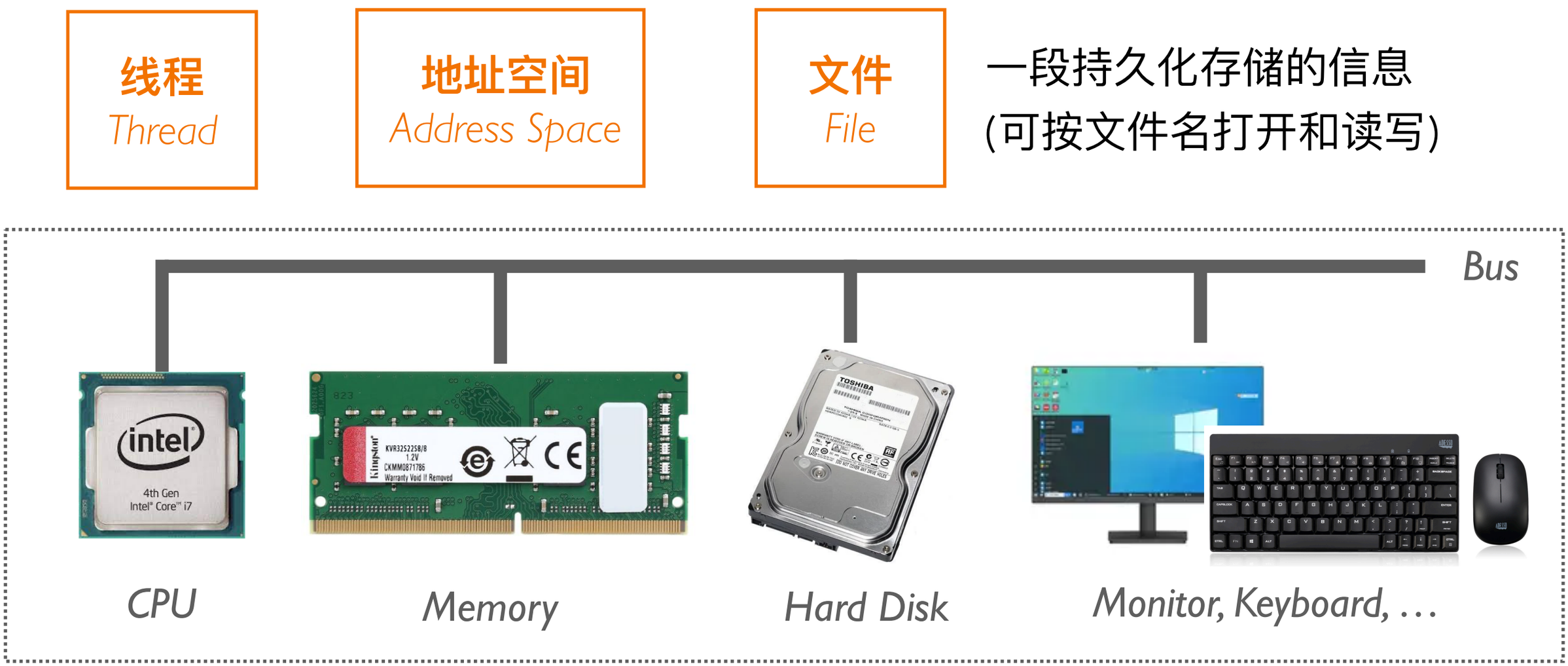
# 操作系统中的抽象

**线程**
*Thread*

**地址空间**
*Address Space*

**文件**
*File*

一段持久化存储的信息
(可按文件名打开和读写)




计算机硬件

# 操作系统中的抽象

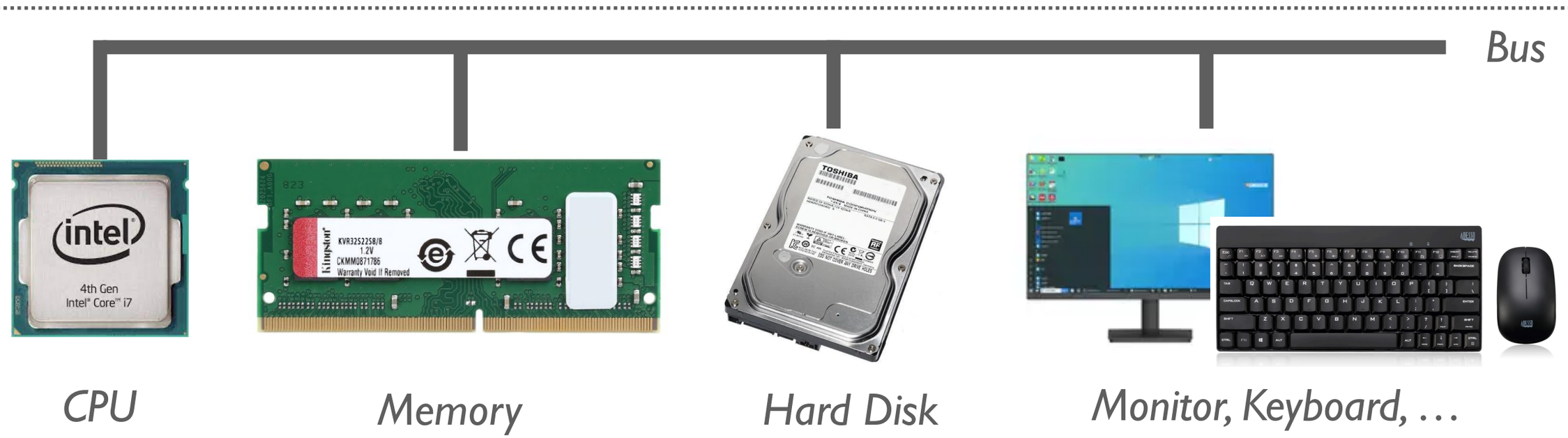
**线程**
*Thread*

**地址空间**
*Address Space*

**文件**
*File*

***I/O* 设备**
*I/O Devices*

一个可读写、控制的对象



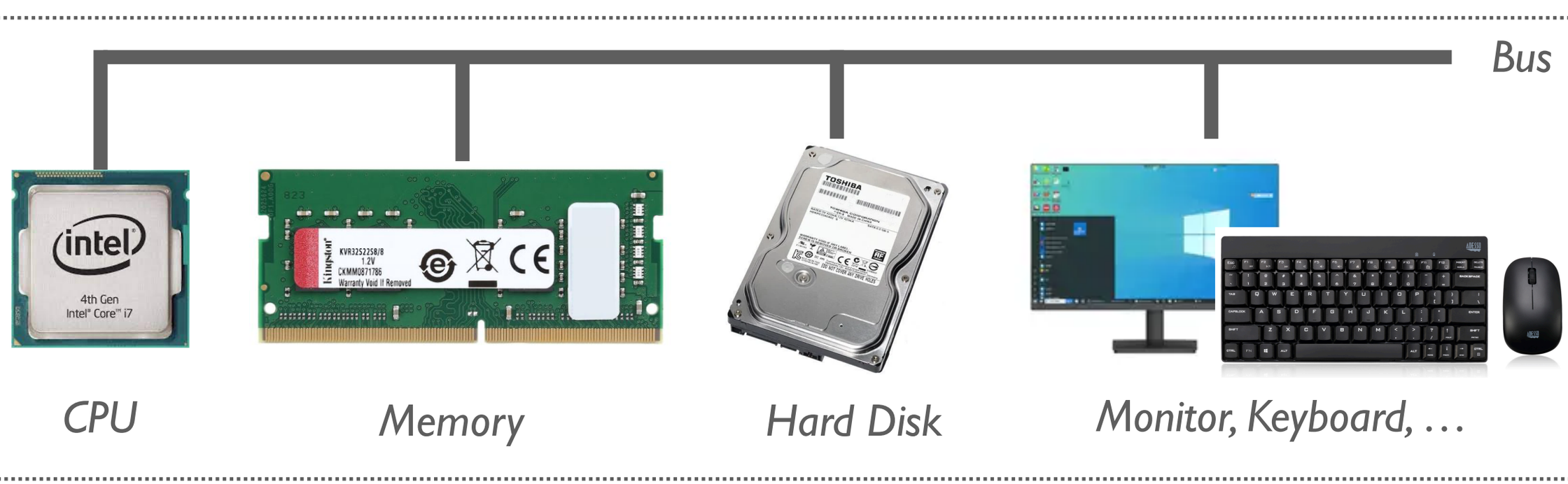

计算机硬件

# 操作系统中的抽象

应用程序看到的世界

**进程 (Process): 一个正在运行的应用程序**

| **线程** *Thread* | **地址空间** *Address Space* | **文件** *File* | ***I/O* 设备** *I/O Devices* |
|---|---|---|---|




*CPU* *Memory* *Hard Disk* *Monitor, Keyboard, …*

计算机硬件

# 操作系统中的抽象

**进程 (Process): 一个正在运行的应用程序**

**线程** *Thread* | **地址空间** *Address Space* | **文件** *File* | ***I/O* 设备** *I/O Devices*

**进程 (Process): 一个正在运行的应用程序**

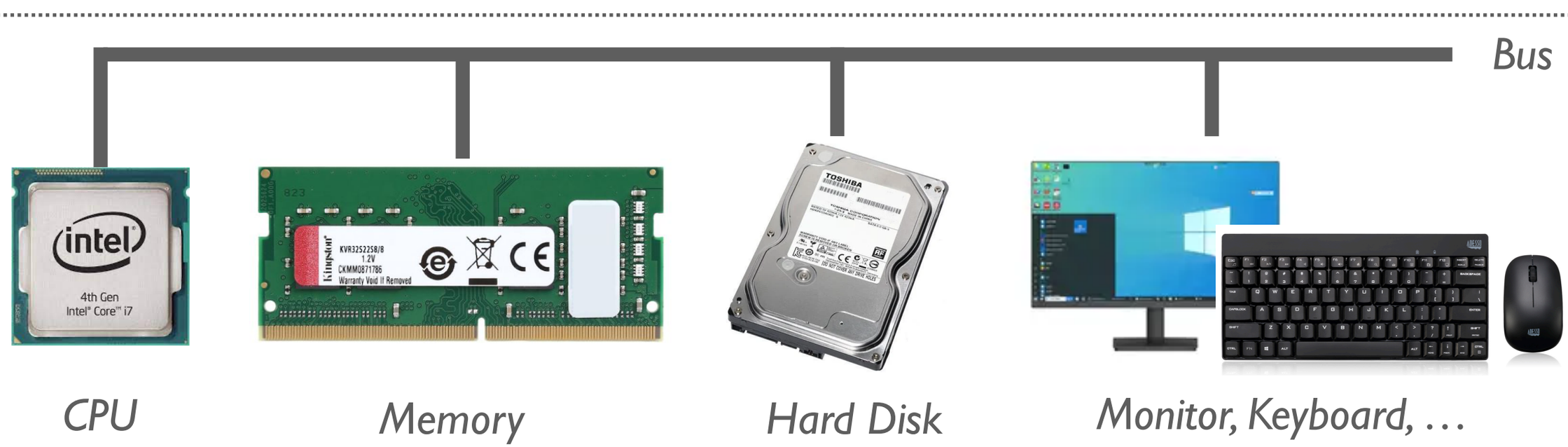
**线程** *Thread* | **地址空间** *Address Space* | **文件** *File* | ***I/O* 设备** *I/O Devices*

**进程 (Process): 一个正在运行的应用程序**

**线程** *Thread* | **地址空间** *Address Space* | **文件** *File* | ***I/O* 设备** *I/O Devices*

**进程 (Process): 一个正在运行的应用程序**

**线程** *Thread* | **地址空间** *Address Space* | **文件** *File* | ***I/O* 设备** *I/O Devices*




计算机硬件

# 操作系统中的保护

为了实现多个程序同时运行 (共享硬件资源)，需要确保一个程序的 Bugs 或恶意行为不会对整个系统造成影响

- 保护和隔绝是实现虚拟化的必然要求
  - 用户程序不能直接与硬件交互
  - 用户程序不能访问操作系统内核的数据结构
  - 用户程序的运行不应对操作系统或其它程序的运行产生影响
- 实现保护的同时需要保持功能 (functionality)、性能 (performance) 和控制 (control)
  - 通常是软件和硬件结合的方式

# 特权指令

通过特权指令 (Privileged Instructions) 确保操作系统的主导地位

- 在执行用户程序时，让操作系统模拟每一条指令的运行、还是直接在 CPU 上运行该程序?
- 控制和效率的平衡: 受限直接运行 (Limited Direct Execution)
  - 引入用户态和内核态 (Dual-Mode Operation)
  - 用户程序只能在受限的指令集上运行
  - 使用状态寄存器中若干位表示当前 CPU 的模式

# 特权指令

通过特权指令 (Privileged Instructions) 确保操作系统的主导地位

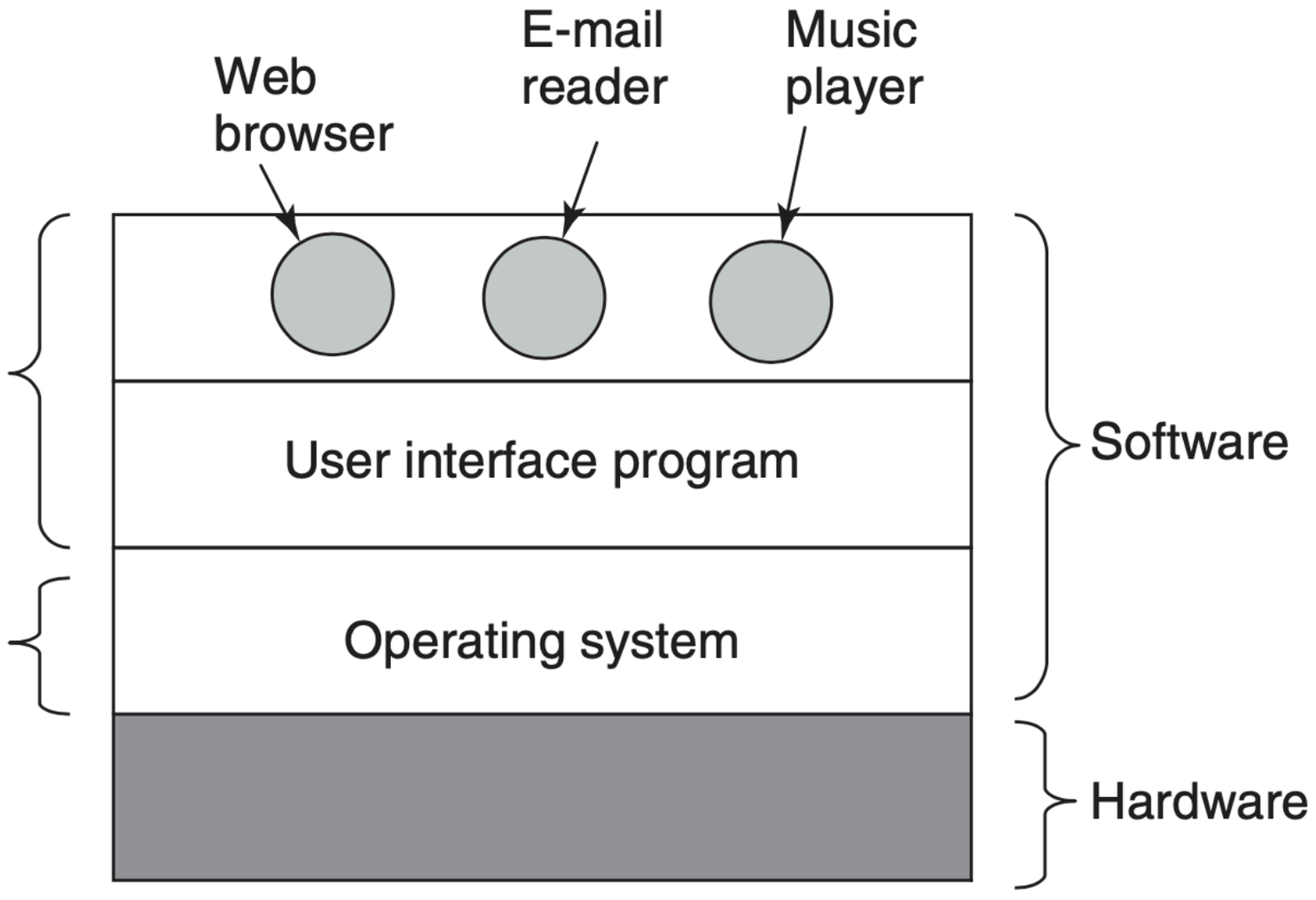
**用户态 User Mode**

CPU 检查每条指令以
验证它是否被允许执行

↓↑

**内核态 Kernel Mode**

指令以不受限制的权限执行

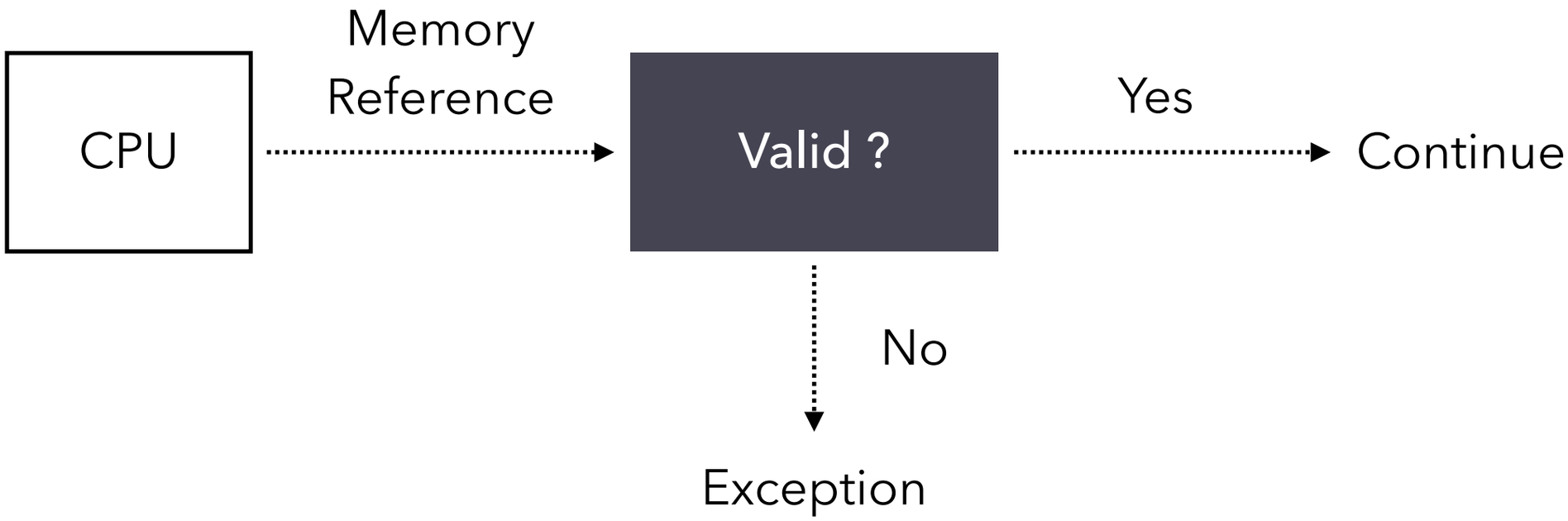



硬件提供特殊的指令来陷入内核 (*trap*)
和从内核中返回 (*return-from-trap*)

# 地址转换

通过地址转换 (Address Translation) 隔离不同用户进程的地址空间

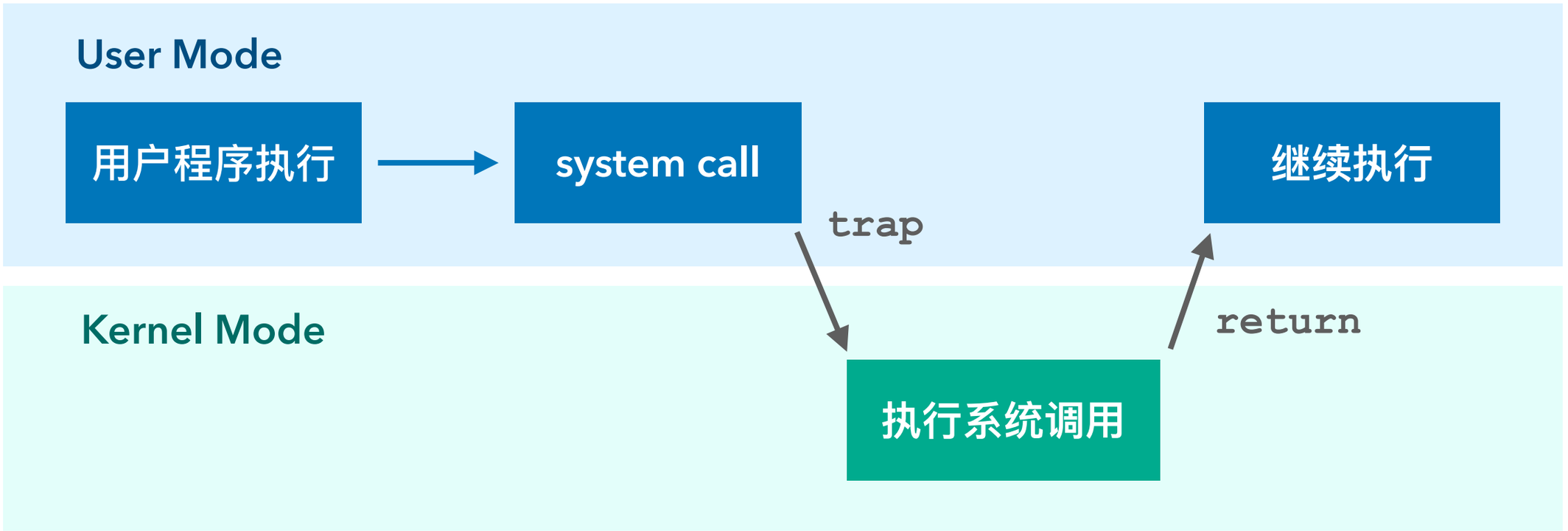
- 操作系统和应用程序的代码/数据都在同一个物理内存中
  - 需要防止不同程序的地址空间互相重叠和干扰
  - 需要防止应用程序意外或恶意地修改操作系统内核数据



# 系统调用

通过系统调用 (System Call) 向用户程序提供服务

- 用户程序只能通过操作系统向外暴露的接口 (以操作系统所允许的方式) 向操作系统请求服务
- 类似过程调用 (Procedure Call)，但会切换到内核态，且不能跳转到任意地址执行



# 回顾 Hello World

main.c

Run

```c
#include <stdio.h>

int main() {
    printf("Hello World\n");
    return 0;
}
```

Output

Clear

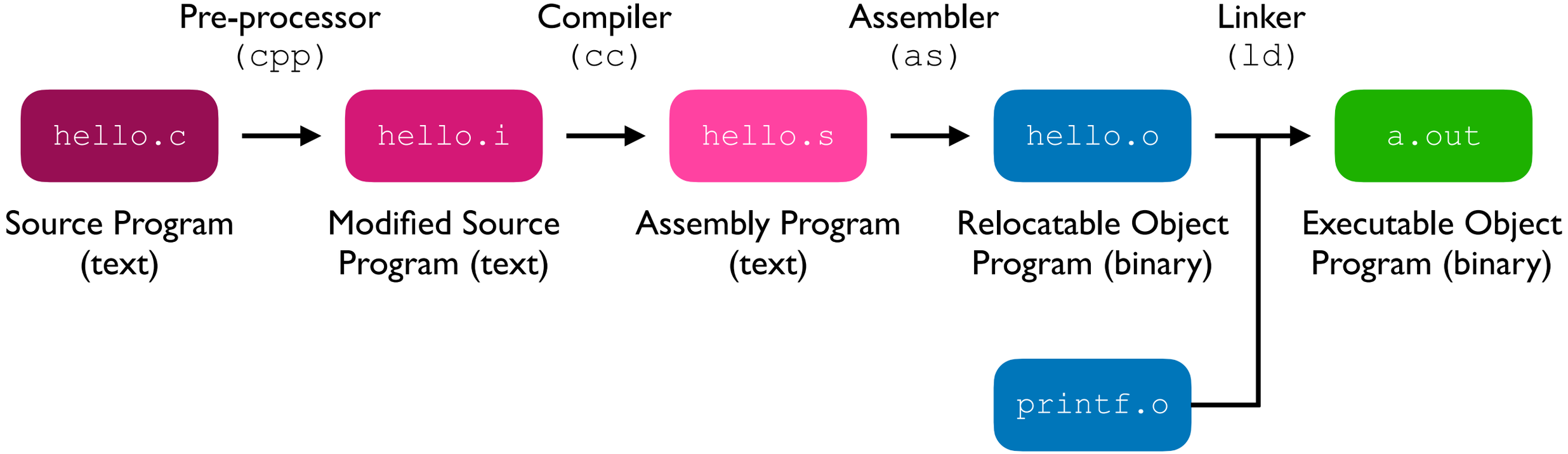
```
/tmp/1g0gu1dEUP.o
Hello World


=== Code Execution Successful ===
```

# 回顾 Hello World

编译阶段：text ➔ binary (存储在磁盘)

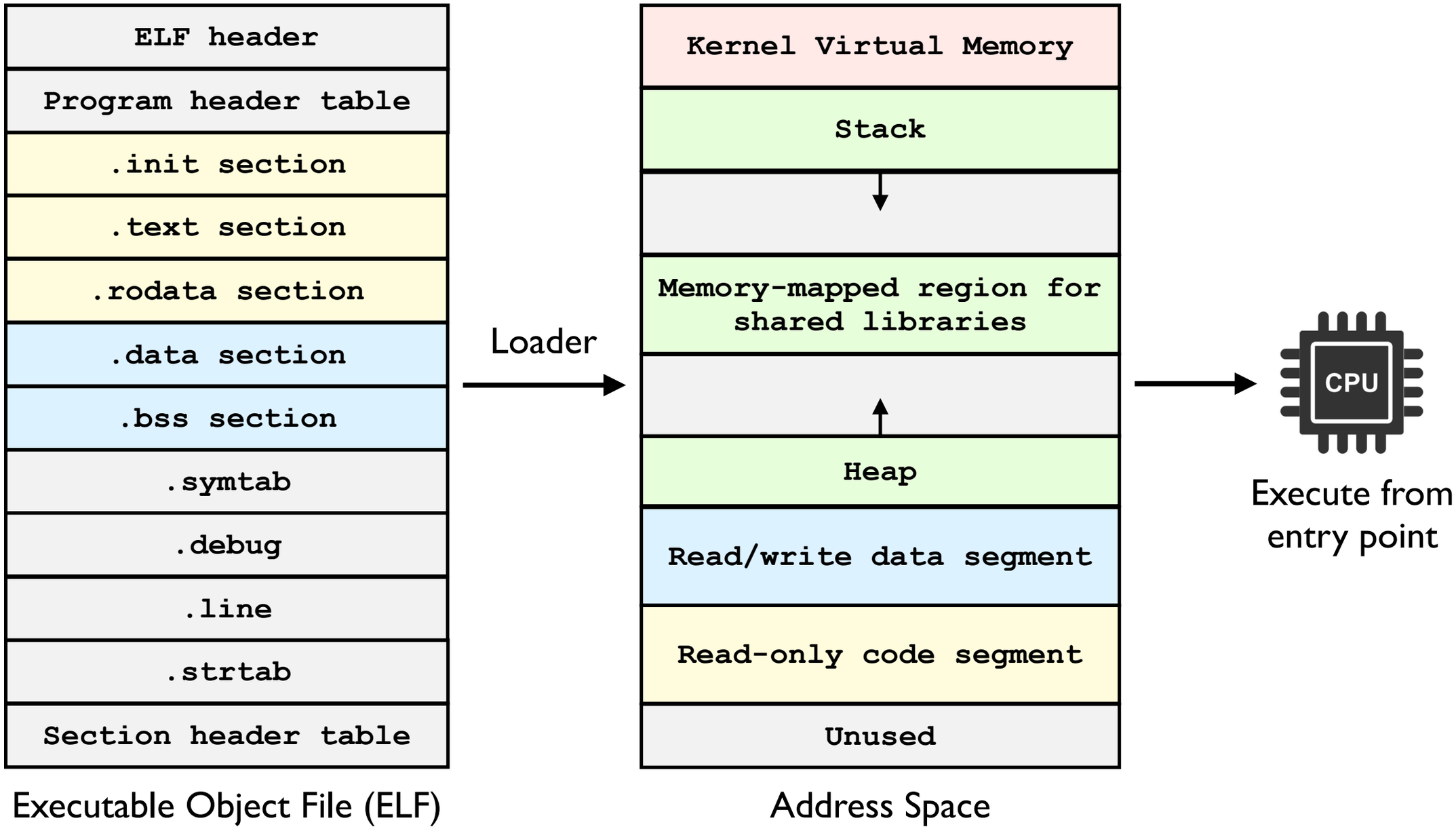
```
gcc hello.c
```



# 回顾 Hello World

加载和执行阶段：磁盘 → 内存 → CPU

`./a.out`



# 回顾 Hello World

- `gcc` 编译出来的 a.out 文件一点也不小
  - 可通过 `objdump -d` 查看 a.out 对应的汇编代码
  - 加上 `-static` 选项会链接 libc 产生更多的代码
- 可以尝试手动控制编译流程
  - 但 `ld` 不知道怎么链接 printf
    - 可通过 `--verbose` 查看 `gcc` 帮我们做的事情
  - 删除 printf 后可以链接
  - 但执行产生了 Segmentation Fault ❌

# 回顾 Hello World

- 如何让程序停下来?
  - 纯计算的指令 `mov` / `add` / `sub` / `call` /... 不行
  - 没有 “停机” 的指令
- 请操作系统来帮忙
  - 通过 `exit()` 系统调用结束当前进程 (回收资源)
  - 通过 `write()` 系统调用向 `stdout` 设备输出信息

# 回顾 Hello World

内核提供的系统调用接口 (platform-dependent)

- 操作系统为每个系统调用分配一个 ID
- 按约定的方式将参数放入指定的寄存器中，然后执行一条特殊的 `syscall` 指令

x86_64 (64-bit)

Compiled from Linux 4.14.0 headers.

| NR | syscall name | references | %rax | arg0 (%rdi) | arg1 (%rsi) | arg2 (%rdx) |
|---|---|---|---|---|---|---|
| 0 | read | man/ cs/ | 0x00 | unsigned int fd | char *buf | size_t count |
| 1 | write | man/ cs/ | 0x01 | unsigned int fd | const char *buf | size_t count |
| 2 | open | man/ cs/ | 0x02 | const char *filename | int flags | umode_t mode |
| 3 | close | man/ cs/ | 0x03 | unsigned int fd | - | - |

Linux System Call Table

https://www.chromium.org/chromium-os/developer-library/reference/linux-constants/syscalls/

# 回顾 Hello World

## 最小 Hello World 的实现

```
.section .data
msg:  .asciz "Hello World\n" ;

.section .text
.globl _start
```

```
_start:
  movq $1,   %rax   // write system call
  movq $1,   %rdi   // file descriptor
  movq $msg, %rsi   // pointer to message
  movq $12,  %rdx   // message length
  syscall

  movq $60,  %rax   // exit system call
  movq $0,   %rdi   // return number
  syscall
```

# 回顾 Hello World

```
int main() {
  printf("Hello World\n");
}
```

*app*

```
write(1, "Hello World\n", 12) {
  ...
  mov 1, rax     // syscall ID
  mov 1, rdi     // arg0
  mov msg, rsi   // arg1
  mov len, rdx   // arg2
  syscall
  ...
}
```

*libc*

程序通过系统调用号请求服务

```
sys_syscall:
  ...
  syscall_table[NR_write]
```

*trap handler*

操作系统内核维护一个 trap table，将系统调用号映射到实现系统调用的函数

```
sys_write() {
  ...
}
```

*kernel implementation*

# 系统调用

可移植的 POSIX 接口

- 通常通过 libc 来实现

**Process management**

| Call | Description |
|---|---|
| pid = fork( ) | Create a child process identical to the parent |
| pid = waitpid(pid, &statloc, options) | Wait for a child to terminate |
| s = execve(name, argv, environp) | Replace a process' core image |
| exit(status) | Terminate process execution and return status |

**File management**

| Call | Description |
|---|---|
| fd = open(file, how, ...) | Open a file for reading, writing, or both |
| s = close(fd) | Close an open file |
| n = read(fd, buffer, nbytes) | Read data from a file into a buffer |
| n = write(fd, buffer, nbytes) | Write data from a buffer into a file |
| position = lseek(fd, offset, whence) | Move the file pointer |
| s = stat(name, &buf) | Get a file's status information |

# 系统调用

可以通过 `strace` 来追踪系统调用

```
kuma@Surface-kuma:~/code/hello$ strace ./a.out
execve("./a.out", ["./a.out"], 0x7ffc5f215800 /* 23 vars */) = 0
brk(NULL)                               = 0x556d42d57000
arch_prctl(0x3001 /* ARCH_??? */, 0x7fff99577240) = -1 EINVAL (Invalid argument)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=42272, ...}) = 0
mmap(NULL, 42272, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fe279973000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\300A\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0", 32, 848) = 32
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\356\276]_K`\213\212S\354Dkc\230\33\272"..., 68, 880) = 68
fstat(3, {st_mode=S_IFREG|0755, st_size=2029592, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fe279971000
```

`strace` works by using the `ptrace` system call which causes the kernel to stop the program being traced each time it enters or exits the kernel via a system call. The tracing program (in this case, `strace`) can then inspect the state of the program by using `ptrace`.

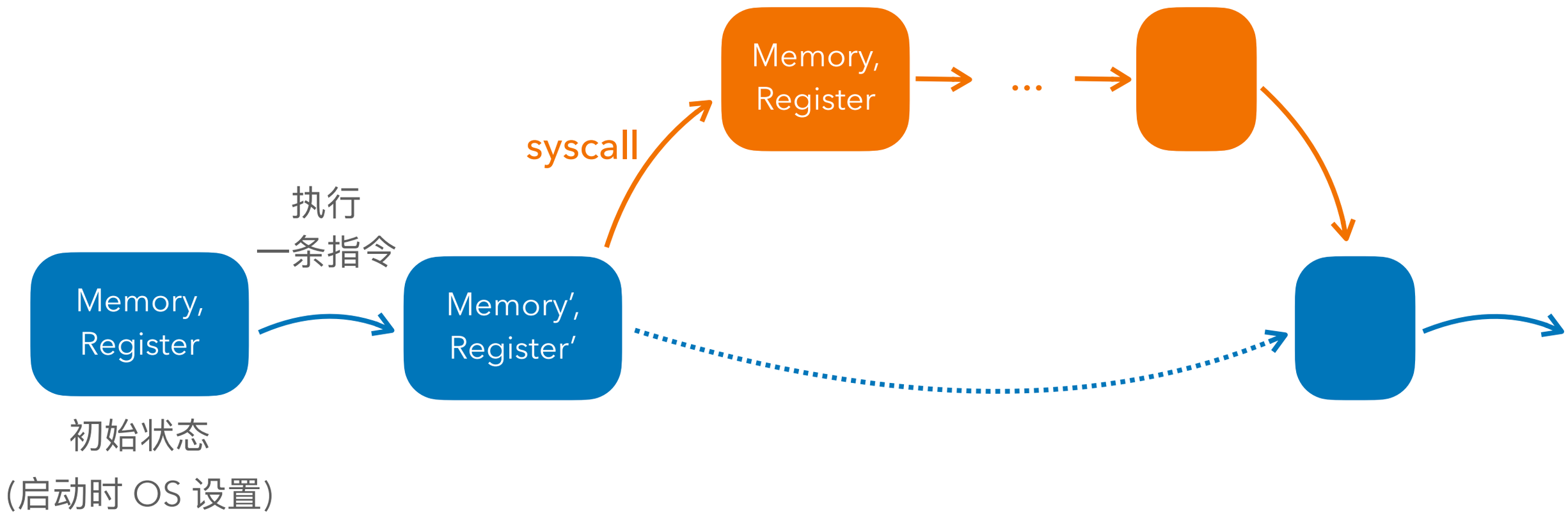
https://man7.org/linux/man-pages/man2/ptrace.2.html

# 系统调用

程序 = 计算 + 系统调用

操作系统的任务就是为应用程序提供
抽象对象及其相关操作

程序把控制权交给操作系统，
操作系统可以改变程序状态甚至终止程序



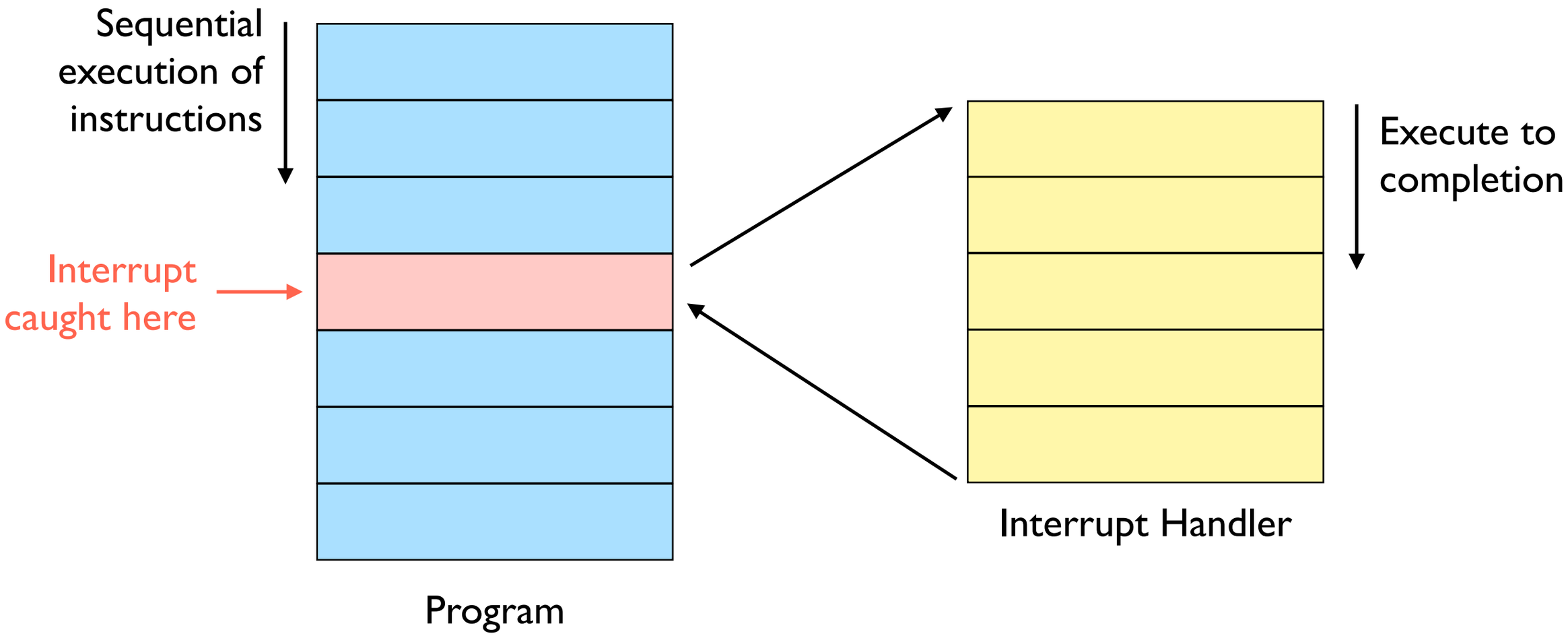
# 中断

通过中断 (Interrupt) 重新获取系统的控制权

- 如果用户程序不产生系统调用，例如 `while(1)`？
- 一种打断 CPU 正常执行的机制
  - I/O Device (硬件向 OS 报告): 已完成某项操作、或遇到错误
  - Program (软件向 OS 报告): 系统调用、执行异常 (exceptions)
  - Hardware Failure: 电源失效、内存数据错误等
  - Timer Interrupt: 时钟中断，可编程为在指定时间间隔 (e.g., every few milliseconds) 触发一次中断

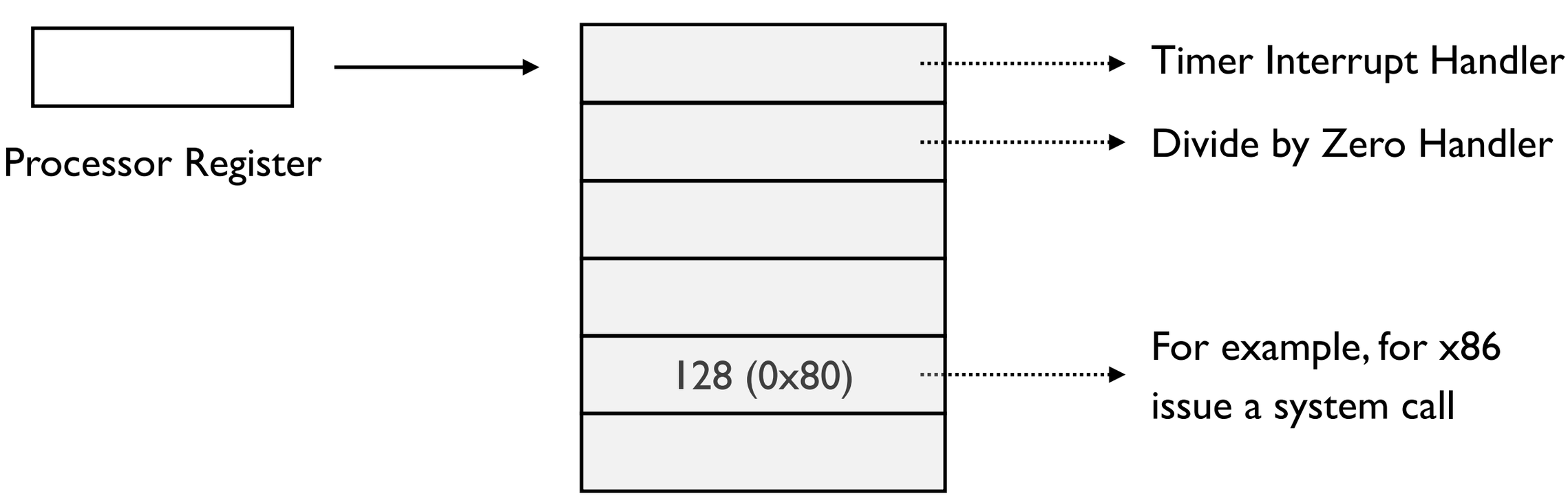
# 中断

- 当中断发生时，当前正在运行的进程将停止，操作系统中预先配置的一个特定中断处理程序 (Interrupt Handler) 将运行
- 无论当前正在干什么，都去执行一下操作系统的内核代码



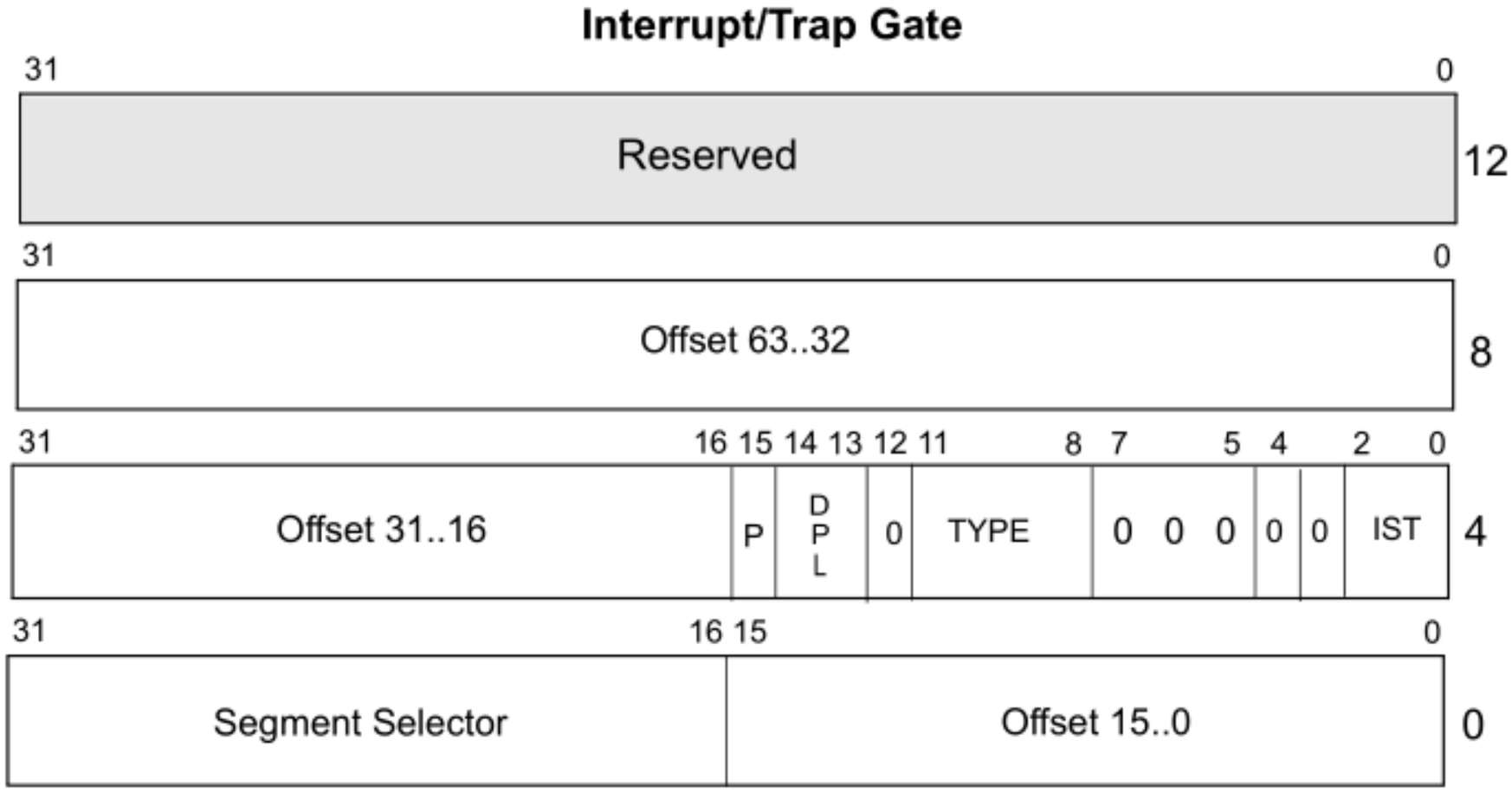
# 中断

- 如何知道要跳转到哪一个代码片段执行?
- 操作系统在启动时设置一个中断描述符表 (Interrupt Descriptor Table)，其中每一项记录内核中特定中断处理程序的指令位置



# 中断

## Case Study: x86




| | |
|---|---|
| DPL | Descriptor Privilege Level |
| Offset | Offset to procedure entry point |
| P | Segment Present flag |
| Selector | Segment Selector for destination code segment |
| IST | Interrupt Stack Table |

# 中断

## Case Study: x86

- 当中断产生时
  - 询问中断控制器获得中断号
  - 在跳转到操作系统代码之前，由硬件保存 Program Counter、Stack Pointer 和 Program Status Word 到堆栈
    - 使用 Kernel Stack 以确保可靠性和安全性
  - Interrupt Handler 保存其余寄存器信息
- 当中断处理结束时
  - 恢复其余寄存器信息
  - 执行 `iret` 指令恢复 PC、SP 和 PSW

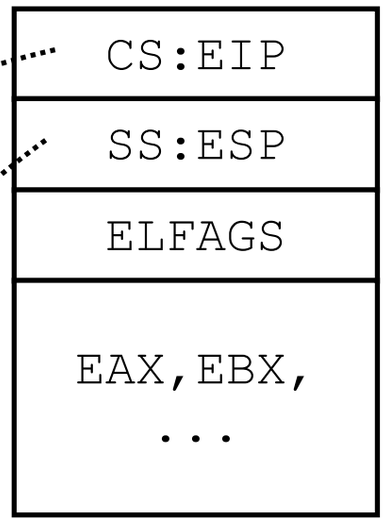
# 中断

## Case Study: x86

User-Level Process

```
foo() {
  while(...) {
    x = x + 1;
    Y = y - 2;
  }
}
```

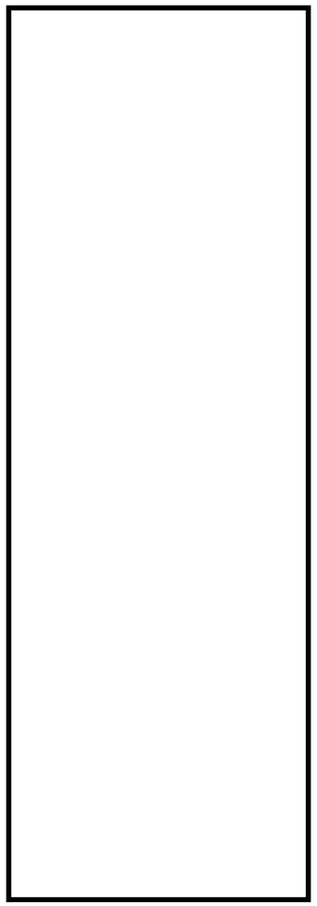
Registers




Kernel

```
handler() {
  pushad
  ...
}
```

Interrupt Stack



User Stack

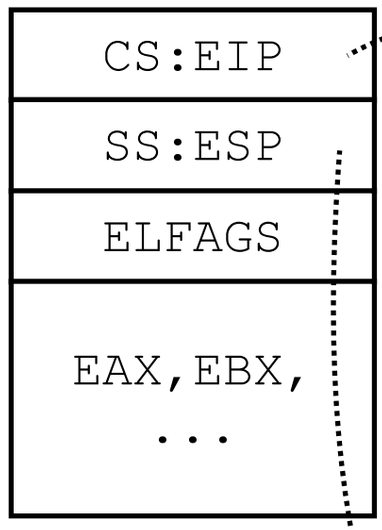
# 中断

## Case Study: x86

User-Level Process

```
foo() {
  while(...) {
    x = x + 1;
    Y = y - 2;
  }
}
```

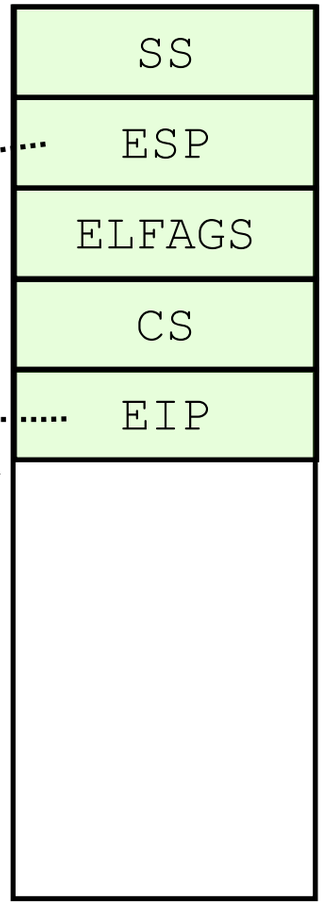
Registers



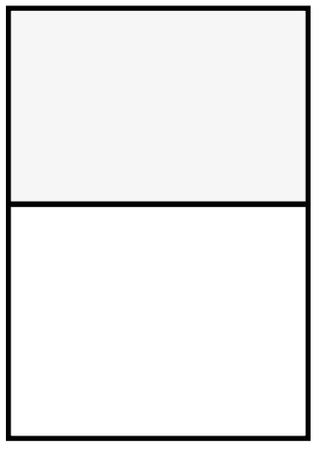

Kernel

```
handler() {
  pushad
  ...
}
```

Interrupt Stack



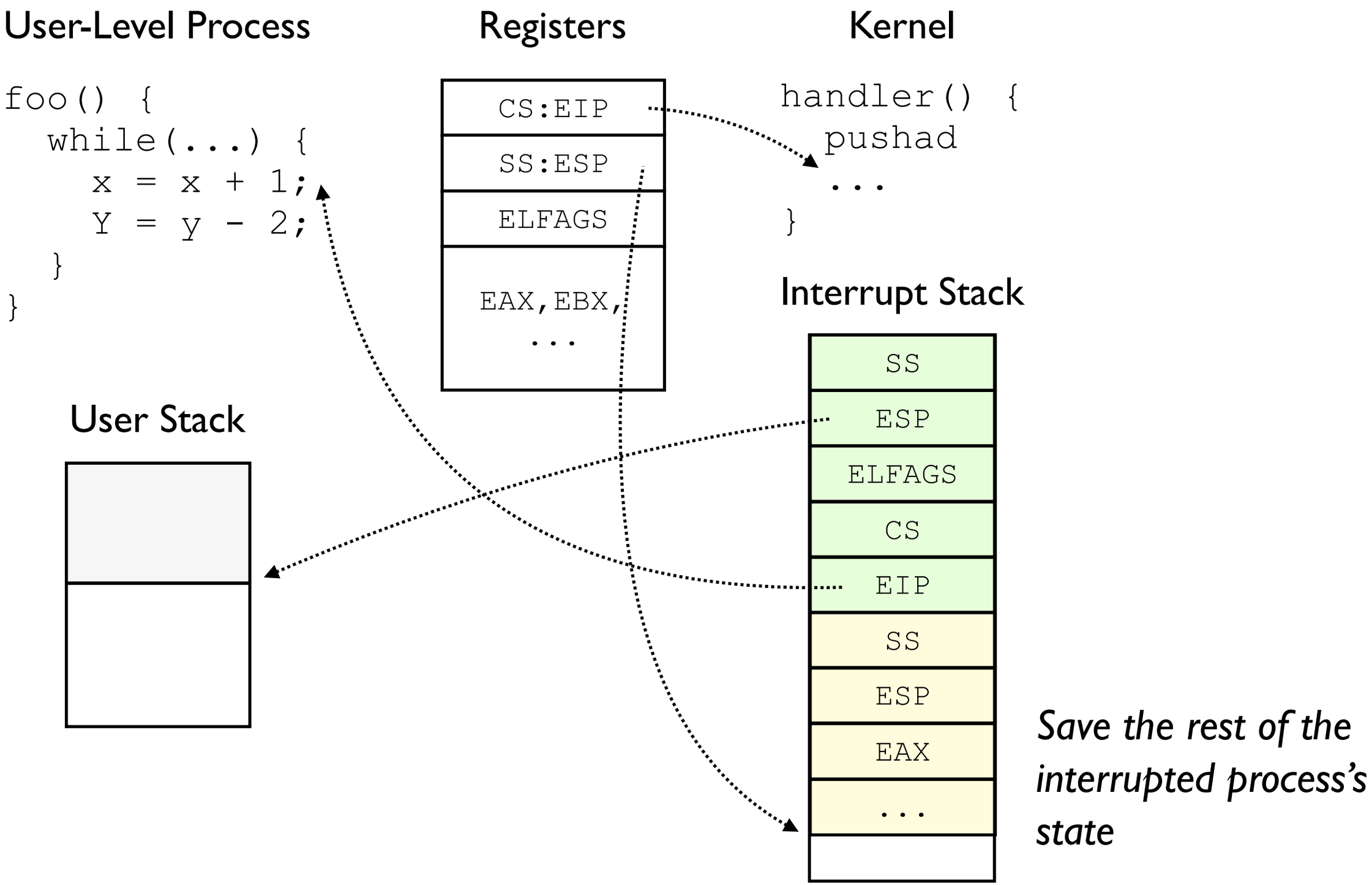

User Stack



*Save user context, and change the program counter/stack*

# 中断

## Case Study: x86




*Save the rest of the interrupted process's state*

# 中断

在处理中断时出现了另一个中断怎么办?

- 关中断 (Disable Interrupts)
  - 临时屏蔽中断，在中断开启后再传递给 CPU
- 将中断处理程序分为 Top Half 和 Bottom Half (Linux)
  - Top Half (屏蔽中断): 快速确认中断，保存必要的状态信息，并将当前中断需要处理的任务加到某个任务队列中
  - Bottom Half: 开启中断后，遍历并处理工作队列中的任务

# 启动操作系统

为了让 OS 这个程序能够正确启动，计算机硬件需要和程序员之间有个约定

- CPU Reset 后的状态 (寄存器值 PC) 是什么?
- 固件 (Firmware) 厂商自由处理这个地址上的值
- ROM 中的某段代码会执行
- 初始化硬件，管理硬件和系统配置
- 加载操作系统 (把存储设备上的某段代码加载到内存)

# 启动操作系统

例如，对于 x86 Family CPU

- `EIP = 0x0000fff0`
- `CR0 = 0x60000010`
  - CPU 处于实模式 (real mode)
  - 页表机制关闭
- `EFLAGS = 0x00000002`
  - 屏蔽中断



### 9.1.1 Processor State After Reset

Following power-up, The state of control register CR0 is 60000010H (see Figure 9-1). This places the processor is in real-address mode with paging disabled.

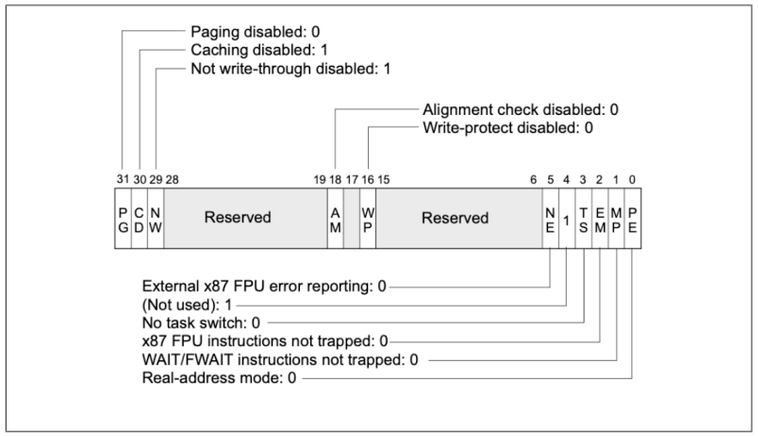



Figure 9-1. Contents of CR0 Register after Reset

The state of the flags and other registers following power-up for the Pentium 4, Pentium Pro, and Pentium processors are shown in Section 22.39, "Initial State of Pentium, Pentium Pro and Pentium 4 Processors" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

Table 9-1 shows processor states of IA-32 and Intel 64 processors with CPUID DisplayFamily signature of 06H at the following events: power-up, RESET, and INIT. In a few cases, the behavior of some registers behave slightly different across warm RESET, the variant cases are marked in Table 9-1 and described in more detail in Table 9-2.

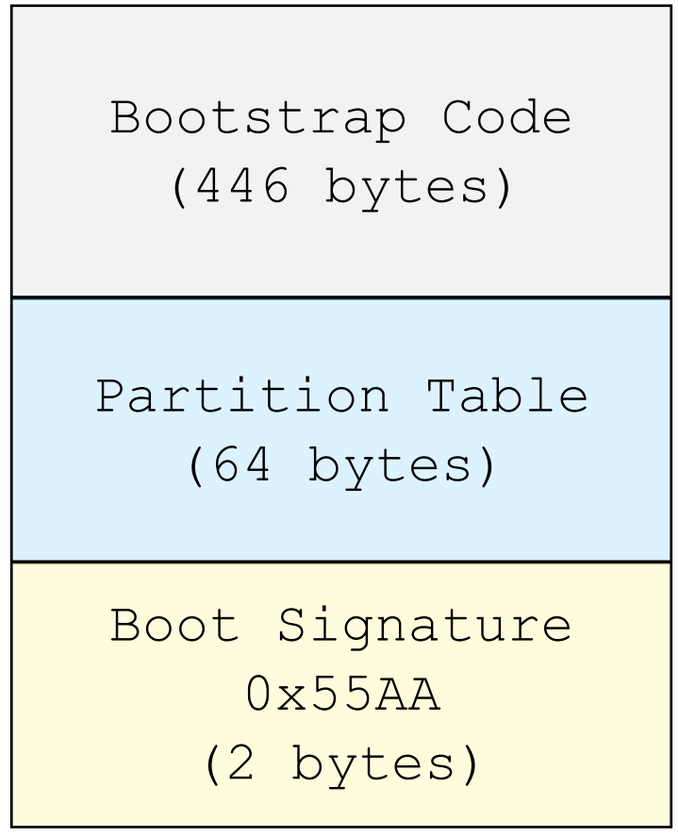
Table 9-1. IA-32 and Intel 64 Processor States Following Power-up, Reset, or INIT

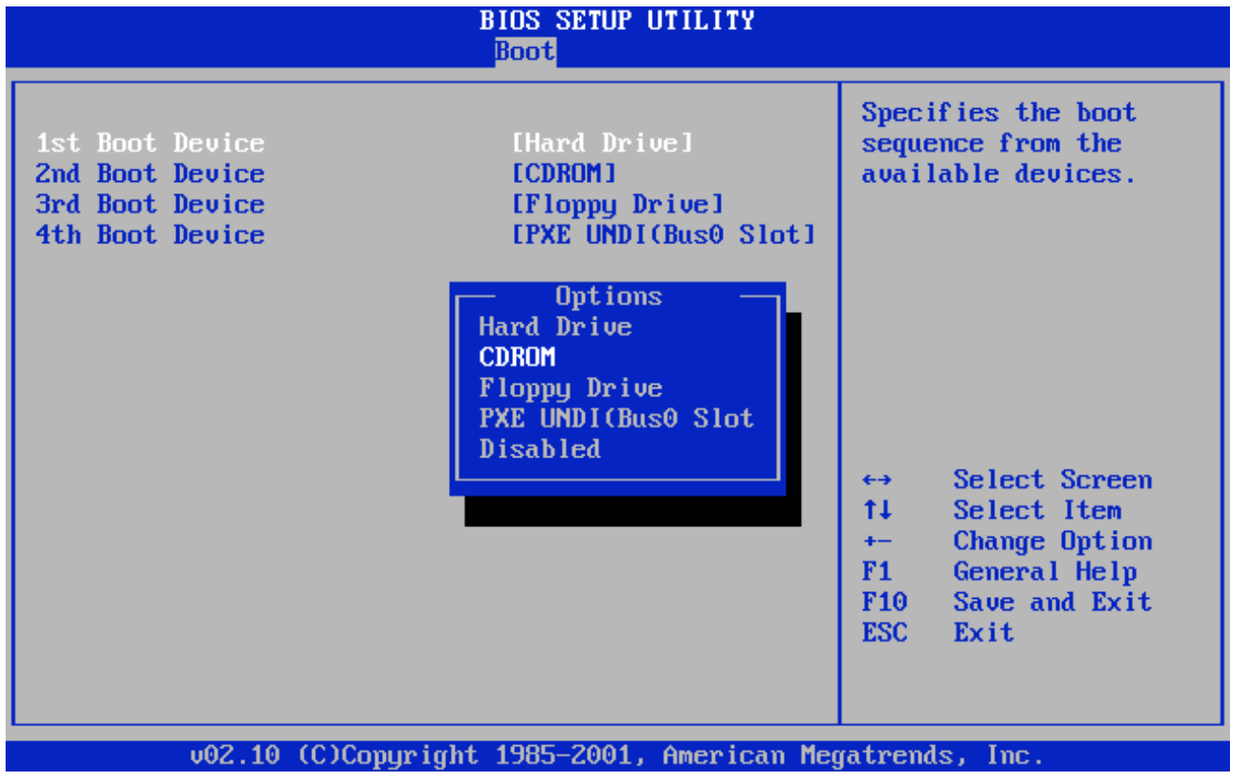
| Register | Power up | Reset | INIT |
|---|---|---|---|
| EFLAGS[1] | 00000002H | 00000002H | 00000002H |
| EIP | 0000FFF0H | 0000FFF0H | 0000FFF0H |
| CR0 | 60000010H[2] | 60000010H[2] | 60000010H[2] |
| CR2, CR3, CR4 | 00000000H | 00000000H | 00000000H |
| CS | Selector = F000H<br>Base = FFFF0000H<br>Limit = FFFFH<br>AR = Present, R/W, Accessed | Selector = F000H<br>Base = FFFF0000H<br>Limit = FFFFH<br>AR = Present, R/W, Accessed | Selector = F000H<br>Base = FFFF0000H<br>Limit = FFFFH<br>AR = Present, R/W, Accessed |
| SS, DS, ES, FS, GS | Selector = 0000H<br>Base = 00000000H<br>Limit = FFFFH<br>AR = Present, R/W, Accessed | Selector = 0000H<br>Base = 00000000H<br>Limit = FFFFH<br>AR = Present, R/W, Accessed | Selector = 0000H<br>Base = 00000000H<br>Limit = FFFFH<br>AR = Present, R/W, Accessed |
| EDX | 000n06xxH[3] | 000n06xxH[3] | 000n06xxH[3] |
| EAX | 0[4] | 0[4] | 0[4] |
| EBX, ECX, ESI, EDI, EBP, ESP | 00000000H | 00000000H | 00000000H |
| ST0 through ST7[5] | +0.0 | +0.0 | FINIT/FNINIT: Unchanged |

# 启动操作系统

## BIOS (Basic Input/Output System)

- 检查硬件并找到可引导设备 (boot devices)
- 把第一个可引导设备的第一个 512 字节 (Master Boot Record, MBR) 加载到内存 `0x7c00` 位置
  - CPU 处于 16 bit 模式
  - 规定 `CS:IP = 0x7c00`
- 虽然只有 446 bytes 代码，但控制权已回到程序员手中

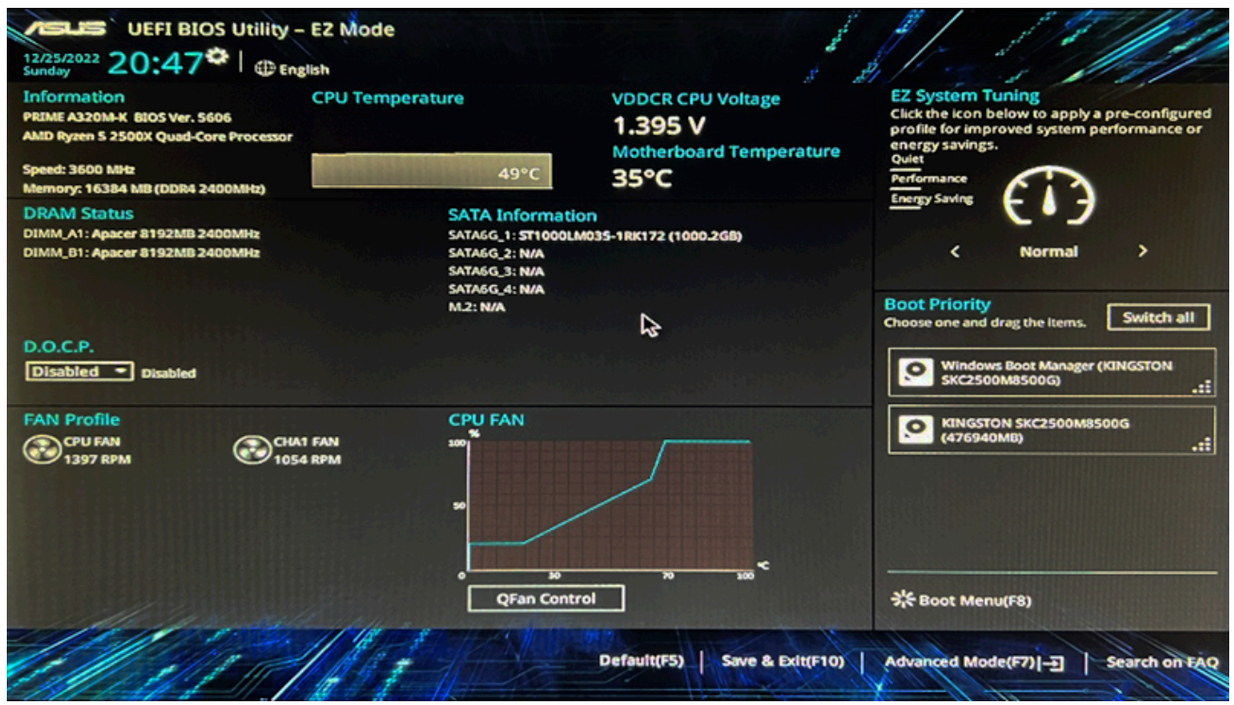


# 启动操作系统

## UEFI (Unified Extensible Firmware Interface)

- 相比 BIOS 支持更多 I/O 设备，更灵活高效、也更安全
- 磁盘按 GPT (GUID Partition Table) 方式格式化
- 预留一个 FAT32 文件系统分区
- UEFI 不依赖引导扇区，而是根据加载配置选项执行特定的代码，且能加载任意大小的 `.efi` 可执行文件

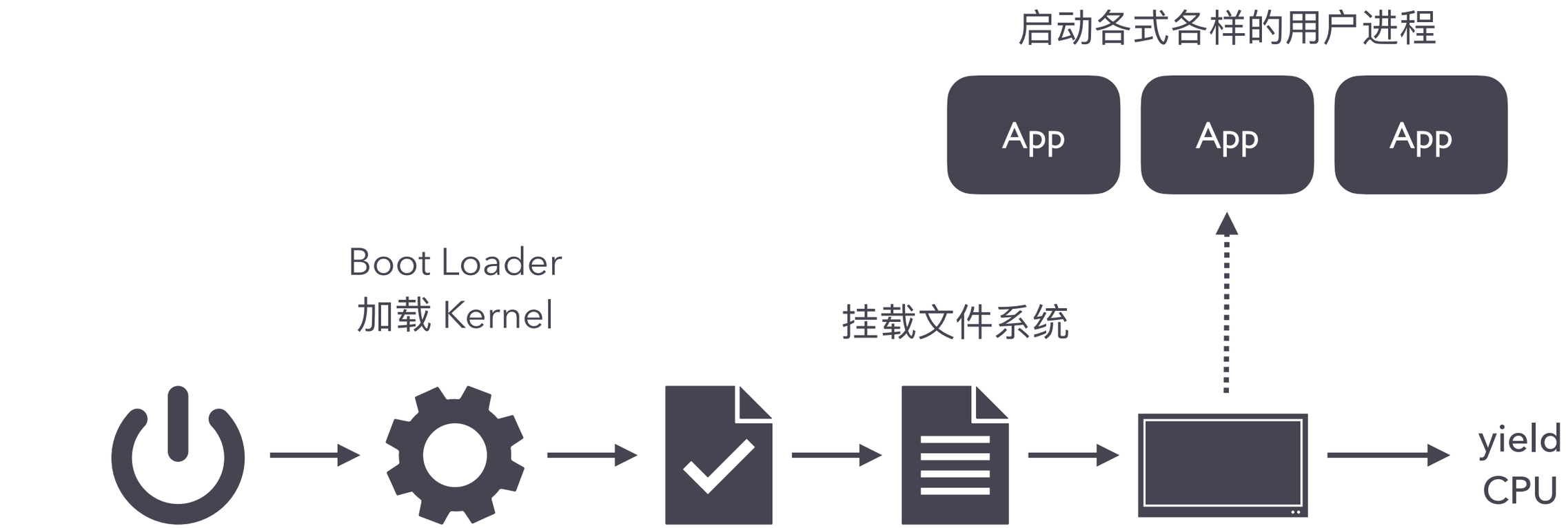


# 启动操作系统

从硬件视角看，操作系统就是一个 C 程序

- 我们已经可以让机器运行任意不超过 446 bytes 的指令序列 (boot loader)
- 利用些指令将磁盘上的一个 C 程序加载到内存，初始化 C 程序的执行环境，操作系统就可以从 `main` 开始运行了
- 在真实环境中，会加载第二级 boot loader，并由这个二级 boot loader 加载磁盘中的操作系统内核到内存中

# 操作系统的一生




CPU Reset 后
Firmware 加载
Boot Loader (MBR)

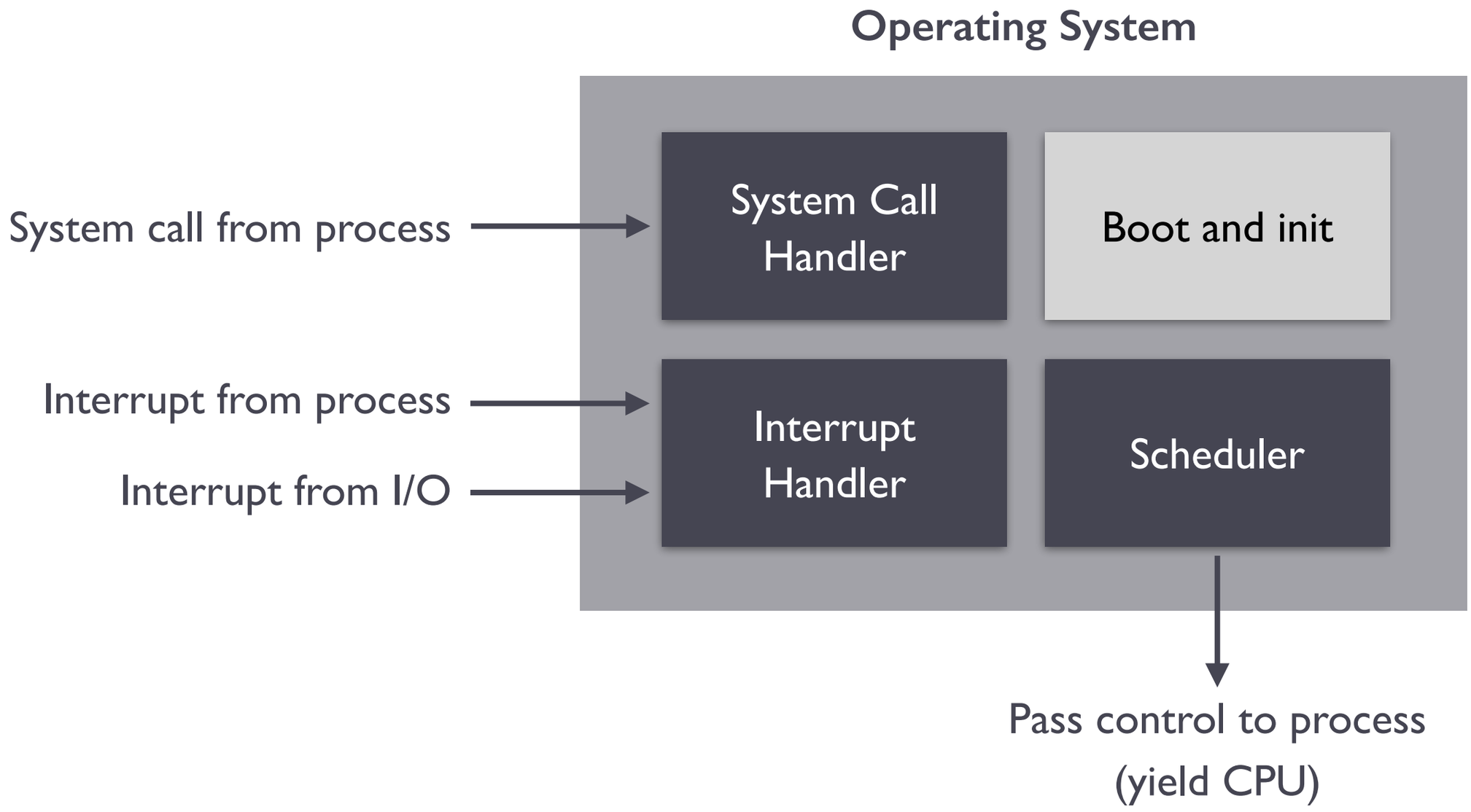
初始化和配置硬件
设置中断处理

启动第一个用
户进程

`init/systemd`
in Linux

# 操作系统的一生

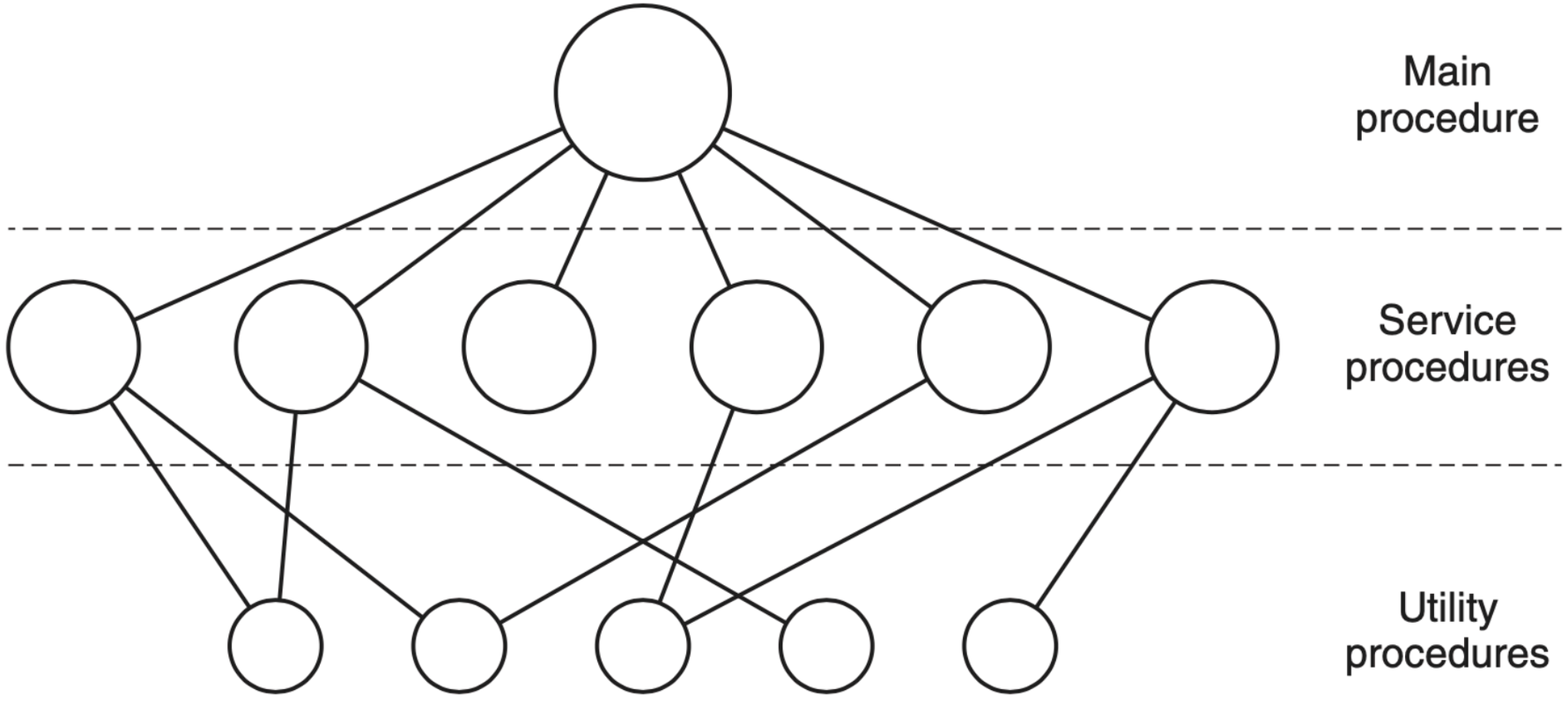
操作系统在初始化完成之后就变成了一个中断处理程序



# 操作系统的架构

## Monolithic Kernel (宏内核)

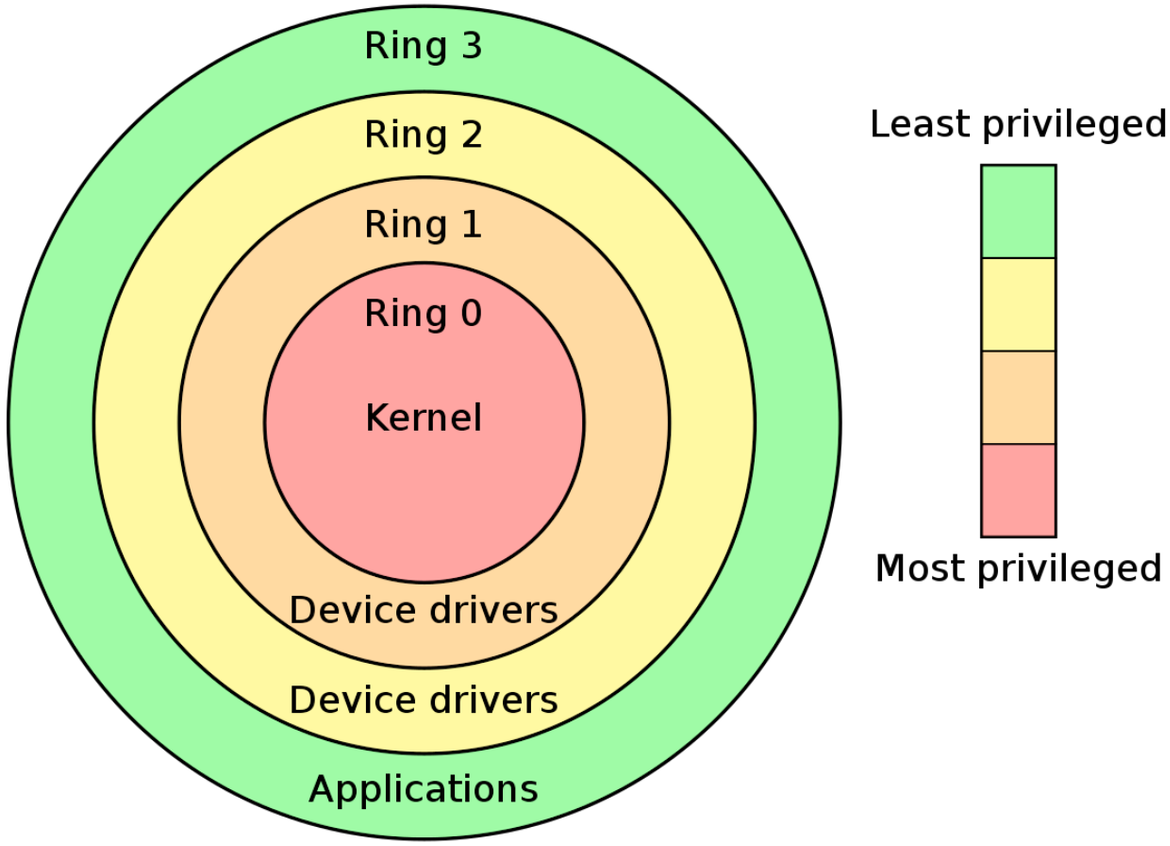
- 操作系统的所有模块都运行在内核态 (a single big program)
- 利用模块化 (e.g., loadable kernel module)、抽象 (e.g., everything is a file in Unix)、分层 (e.g., dual modes) 等控制不断增长的复杂度



# 操作系统的架构

## Monolithic Kernel (宏内核)

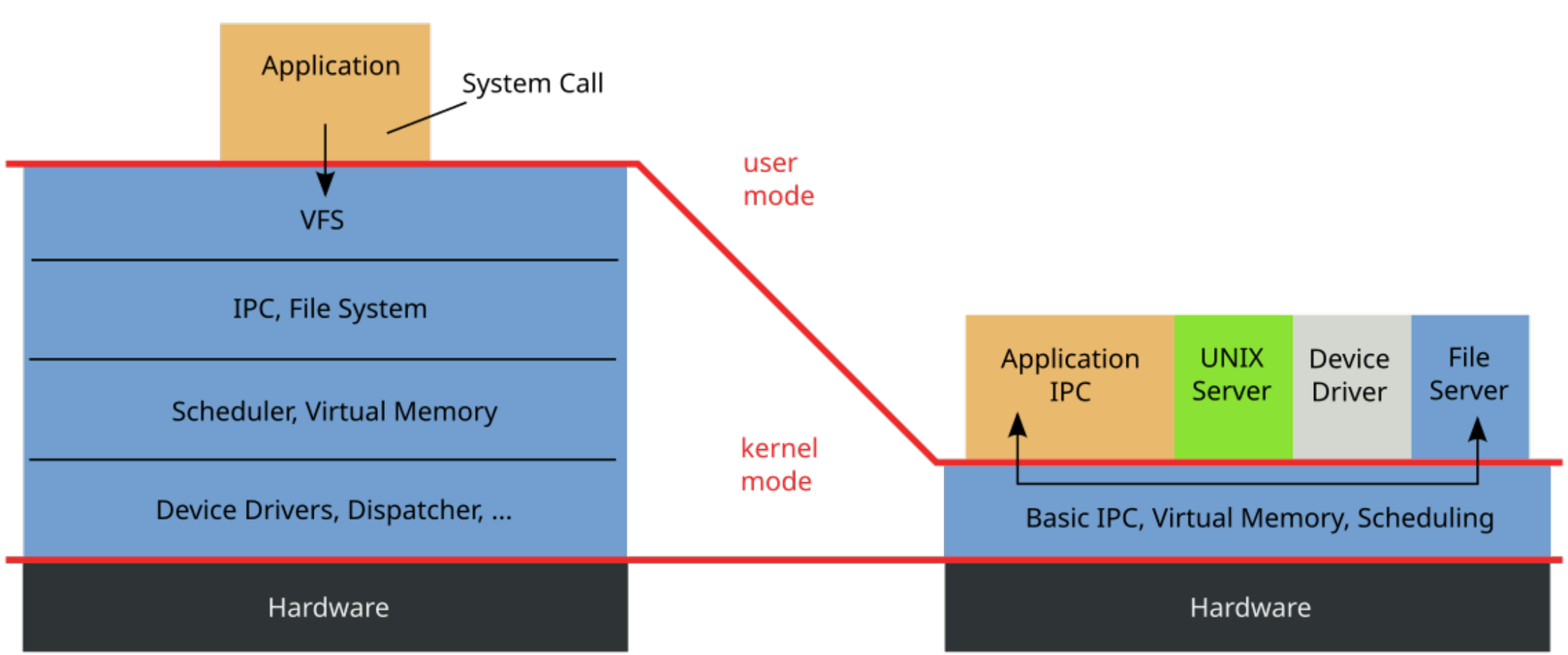
- 操作系统的所有模块都运行在内核态 (a single big program)
- 利用模块化 (e.g., loadable kernel module)、抽象 (e.g., everything is a file in Unix)、分层 (e.g., dual modes) 等控制不断增长的复杂度



# 操作系统的架构

## Microkernels (微内核)

- 运行在内核态模块的一个单点故障就可能导致整个系统奔溃
- 将操作系统核心功能模块化 (由运行在用户态的进程提供服务), 通过进程间通信方式实现交互




*Monolithic* *Microkernel*

# 操作系统的架构

## 机制 (Mechanism) 和策略 (Policy) 分离

- 类比 “接口所需完成的功能 vs. 接口的实现方式”
  - 机制 (如何做 / How to): 实现所需功能的底层方法或协议 (通常在内核中实现)
  - 策略 (做什么 / Which): 做出某种决策的算法 (可在用户态完成)

- 操作系统很多方面都有机制和策略分离的应用
  - CPU 调度
  - 缺页异常处理
  - …

# 操作系统的架构

## Tanenbaum–Torvalds Debate [1992]

"LINUX is obsolete"

Andrew S. Tanenbaum

Linus Torvalds

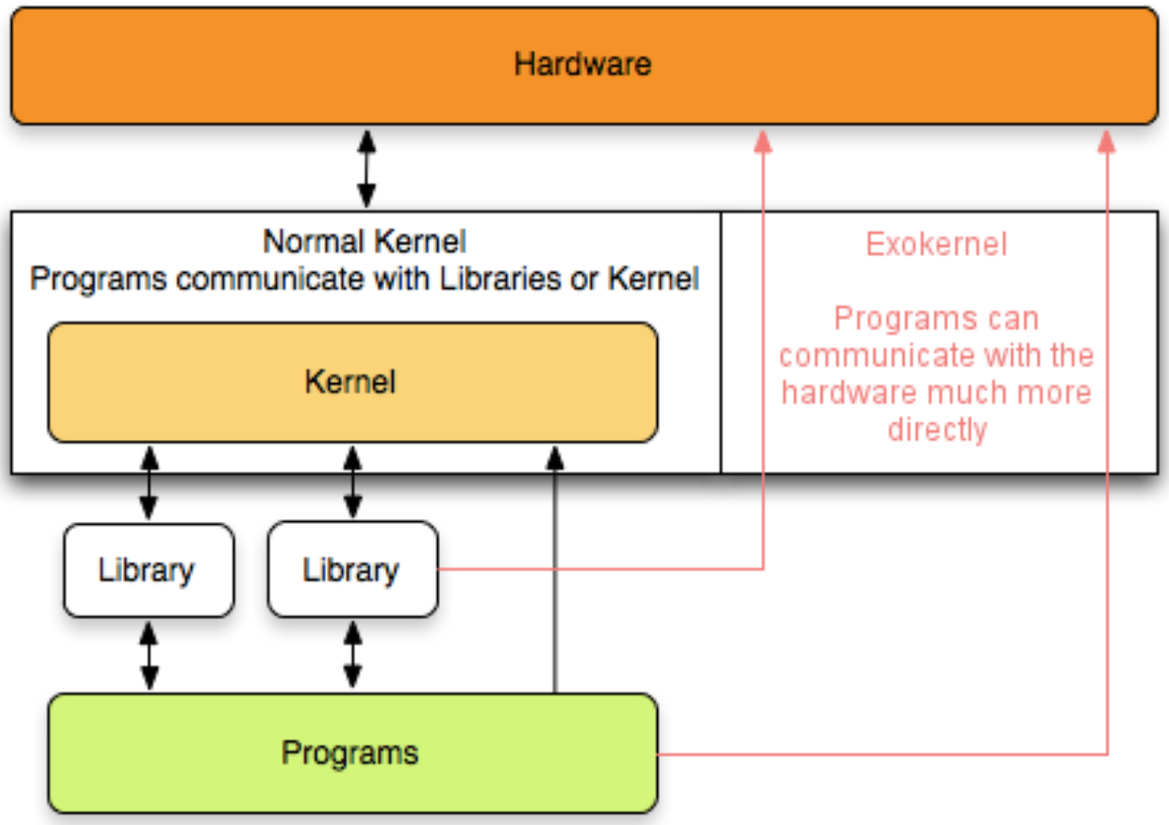
https://en.wikipedia.org/wiki/Tanenbaum–Torvalds_debate

https://www.oreilly.com/openbook/opensources/book/appa.html

# 操作系统的架构

## Exokernel (外核)

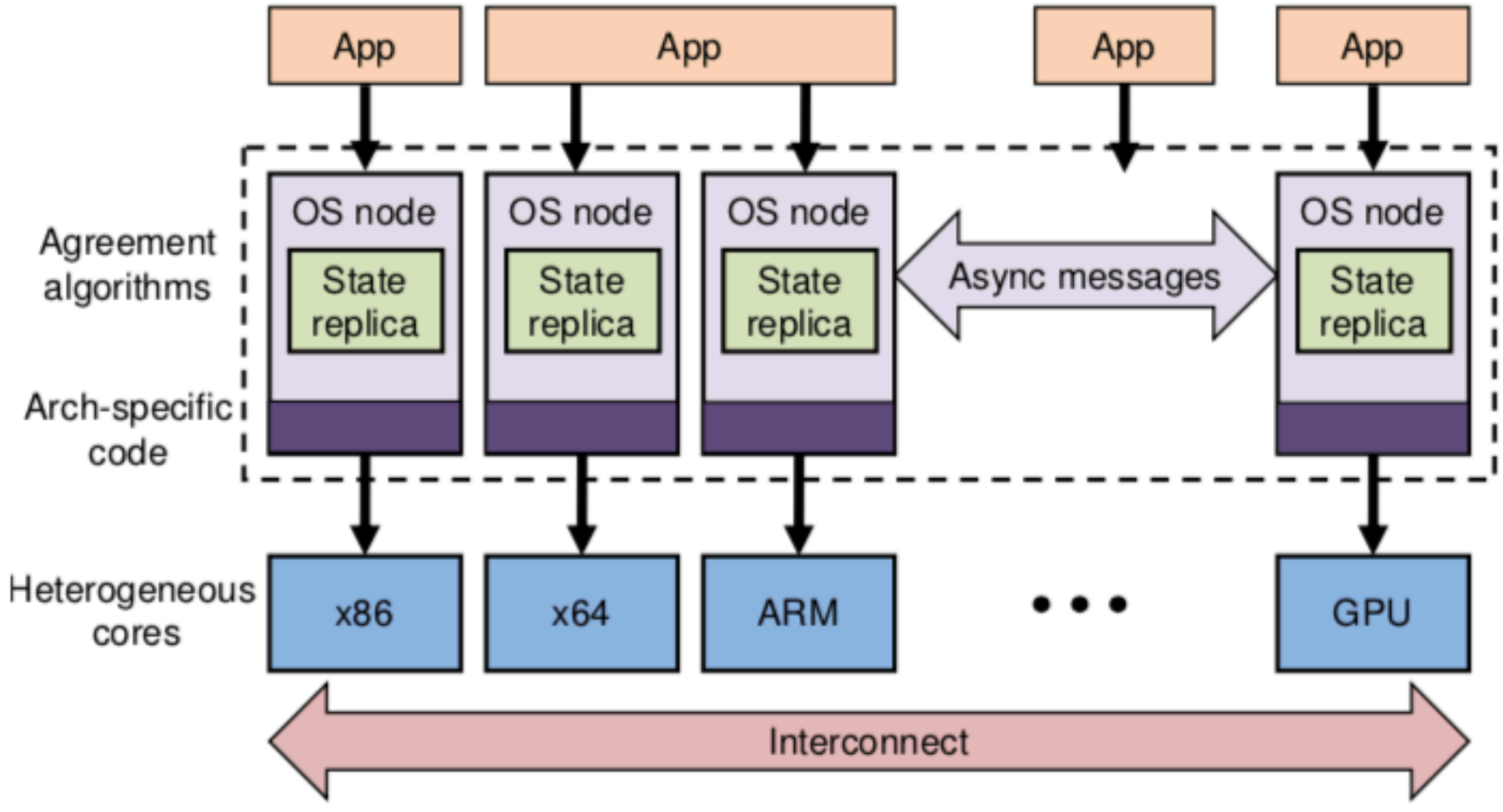
- 操作系统对共享资源的过度抽象可能会带来性能的损失：通用的抽象对一些特殊应用 (e..g, database) 来说也不是最优选择
- 库操作系统 (LibOS)：由应用程序控制对硬件资源的抽象，OS 只负责对硬件资源共享使用的支持



# 操作系统的架构

## Multikernel (多内核)

- 管理通过网络互联的成百上千个异构处理器核 (分布式系统)



# 总结

- 操作系统的定义和视角 
- 操作系统的发展
- 操作系统的核心概念 
  - 对硬件资源的抽象和虚拟化
  - 为应用程序提供服务
  - 本身就是一个运行在硬件上的 C 程序
- 操作系统的架构
  - 宏内核和微内核
  - 机制和策略分离