操作系统概论

Section 1

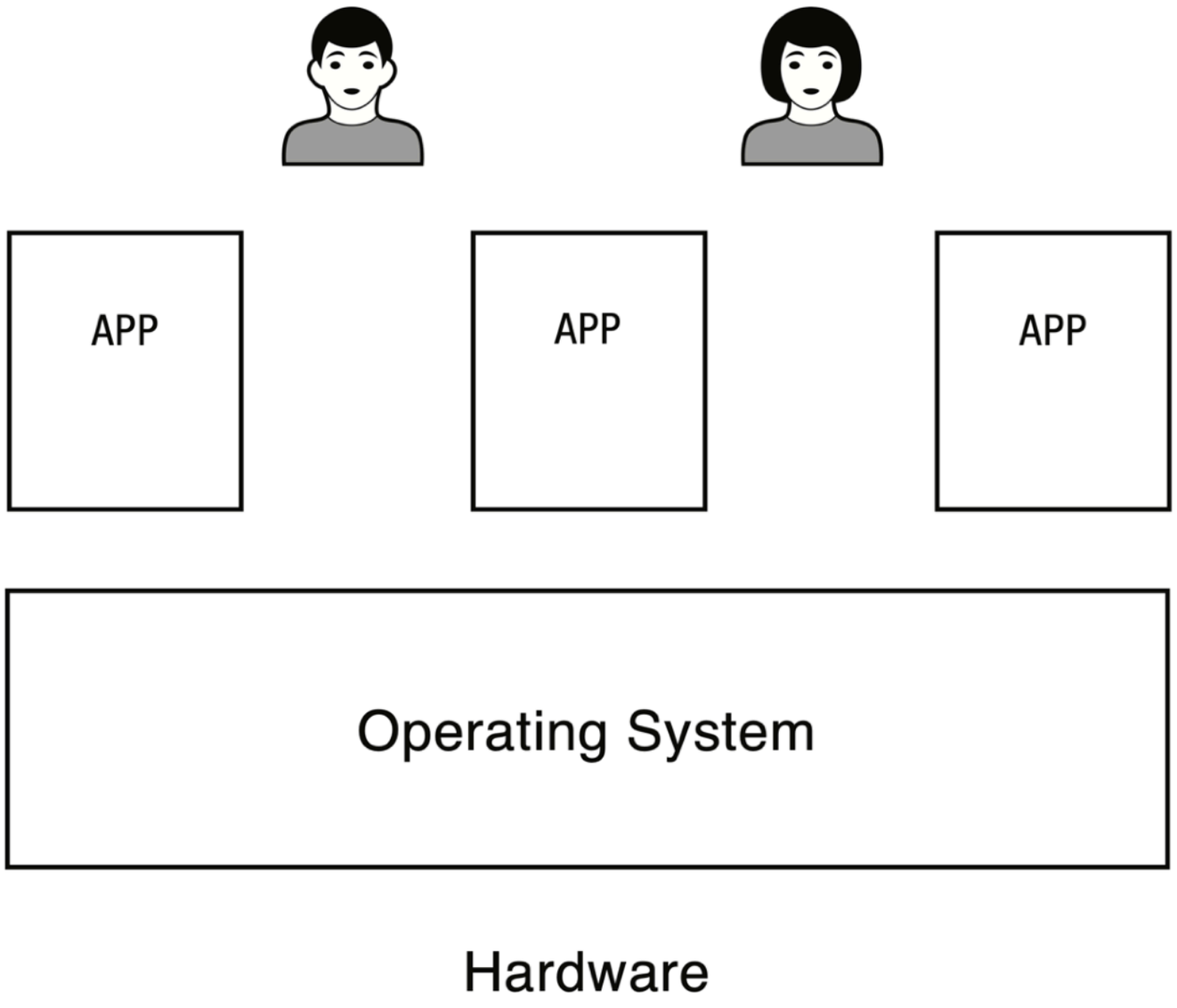
2026

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

服务软件: 为应用程序提供硬件资源的易用抽象

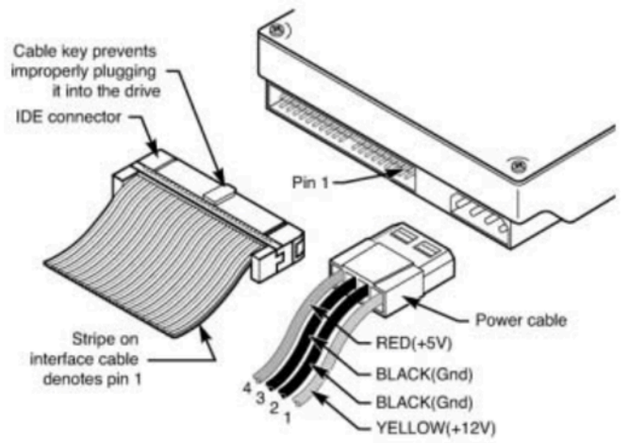
- 让软件的开发和运行 (计算机的使用) 变得更加容易
  - 如何在计算机中存储程序?
  - 如何让程序和外部设备交互?
  - 如何允许多个程序同时运行?
  - …

软件和硬件的中间层

服务软件: 为应用程序提供硬件资源的易用抽象

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

服务软件: 为应用程序提供硬件资源的易用抽象

a.txt

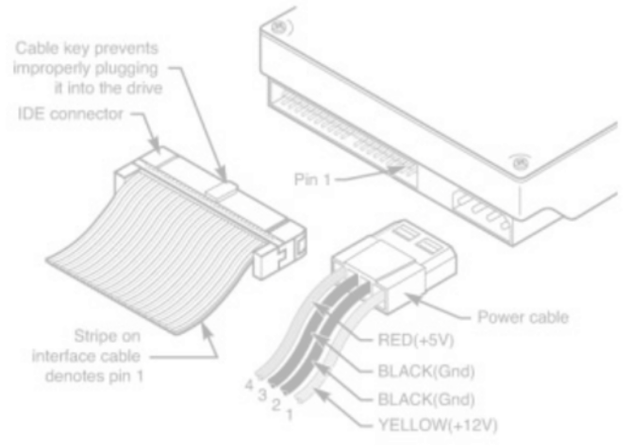
```c
#include <unistd.h>

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

Hard Disk



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

管理硬件: 对计算机硬件资源进行分配和管理

- 硬件资源 (CPU、内存和磁盘等) 是有限的，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

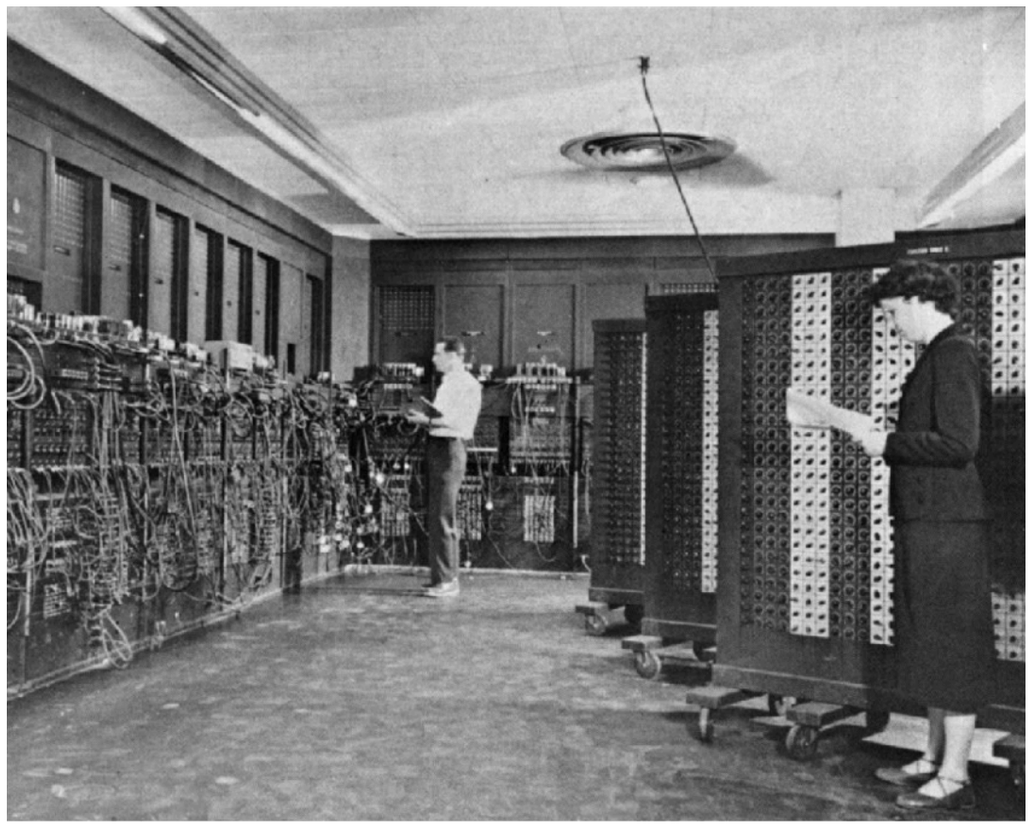
操作系统是目前人类开发的最为复杂的软件系统之一

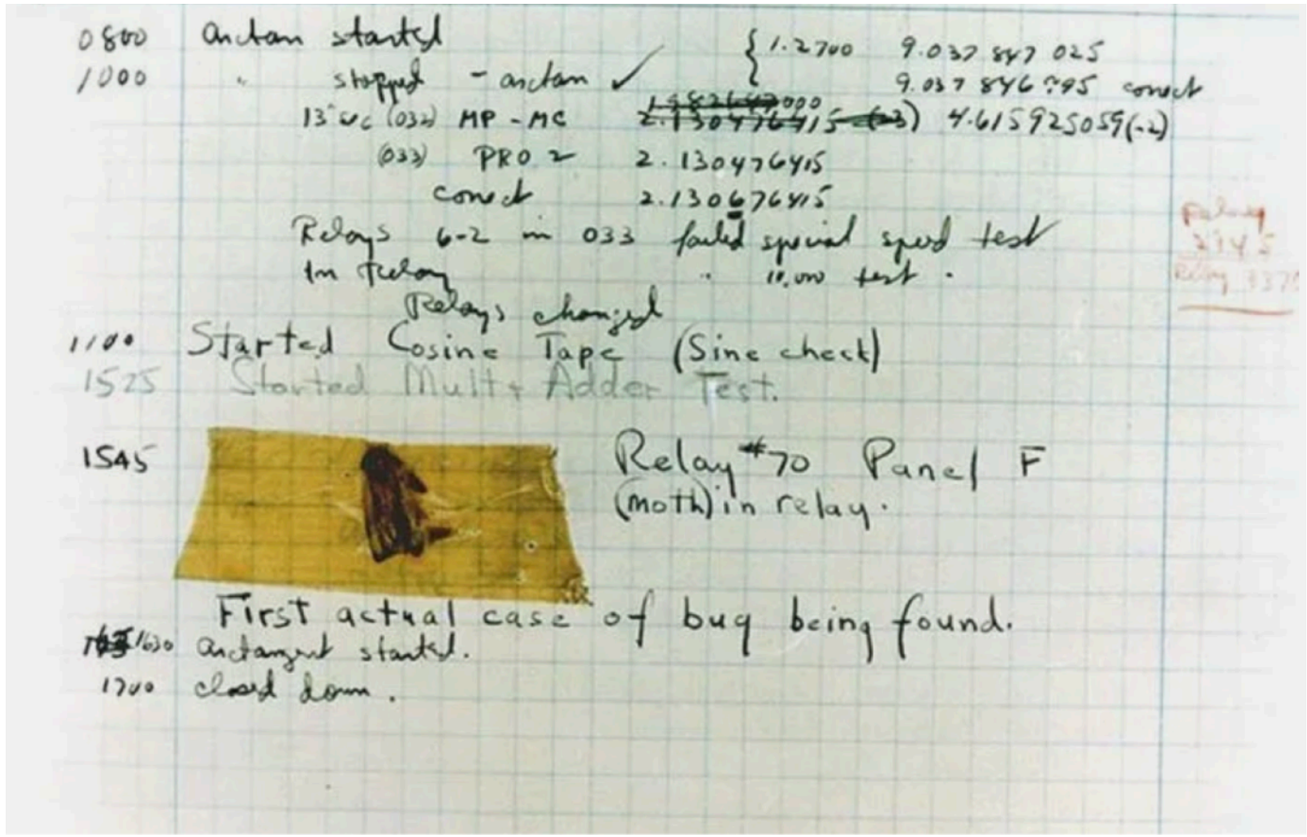
- 发展历程：硬件、软件以及操作系统
- 操作系统 = 管理 + 服务
  - 服务软件 ➡️ 单个应用的开发和运行效率最大化
  - 管理硬件 ➡️ 系统的资源整体利用率最大化

1945～1955: 真空管时代

没有操作系统，程序直接在硬件上运行

- 基于物理连线的编程、以及解决真正的 “bugs”



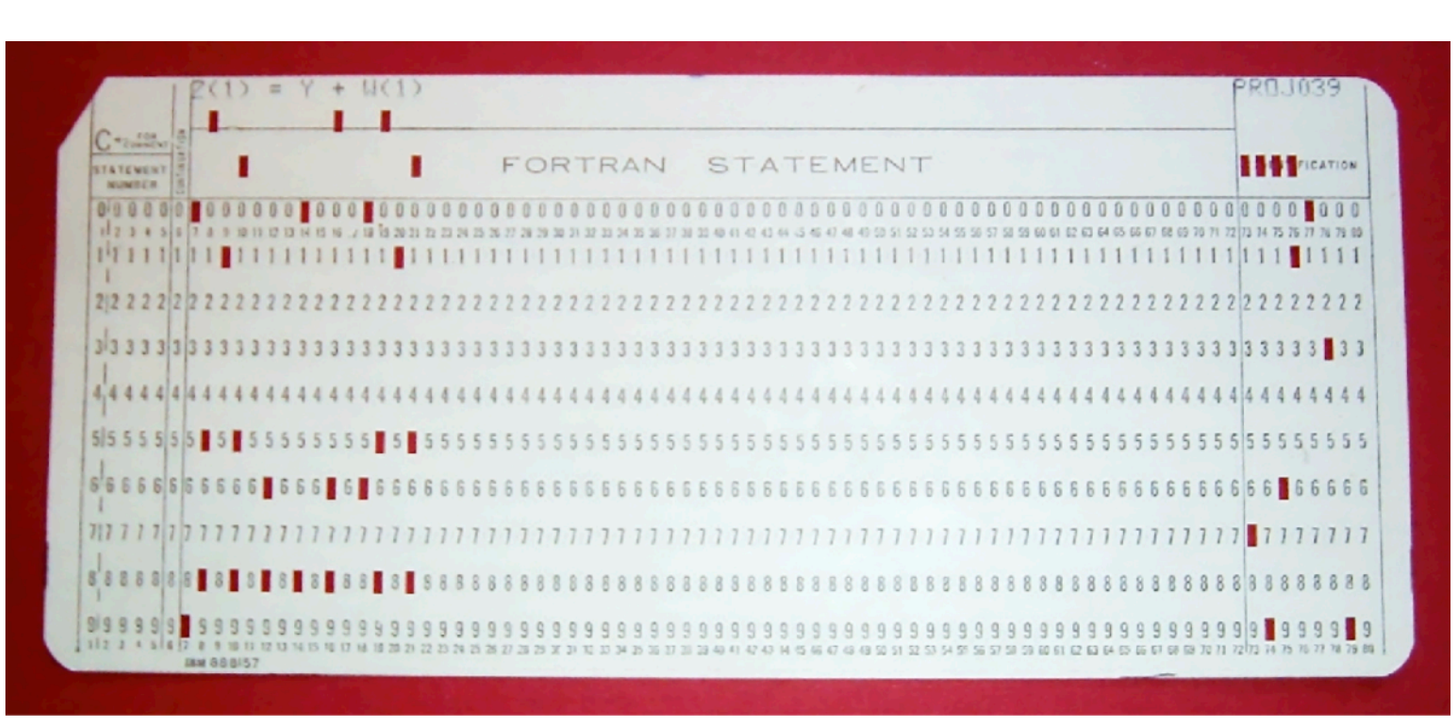


ENIAC (Electronic Numerical Integrator And Computer) 世界上第一台通用计算机

1955～1965: 晶体管时代

开始需要管理计算机系统：更快的硬件、更复杂的软件 (数值计算)

- 编程 (打孔) 开始变得相对容易
- 计算机仍非常昂贵，只能多用户轮流使用，并由操作员负责调度



Fortran (一行代码、一张卡片)

1955～1965: 晶体管时代

开始需要管理计算机系统：更快的硬件、更复杂的软件 (数值计算)

- 操作系统的概念开始形成 (operating system jobs)
- 处理器速度明显高于 I/O 速度，中断 (interrupt) 机制出现
- 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)、设备 (device) 等概念出现
- 管理程序排队运行 (batch processing)

1965～1980: 集成电路时代

集成电路的出现，更多的高级编程语言

- 更大的内存：可以同时载入多个应用程序 (多用户同时使用计算机)
- 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

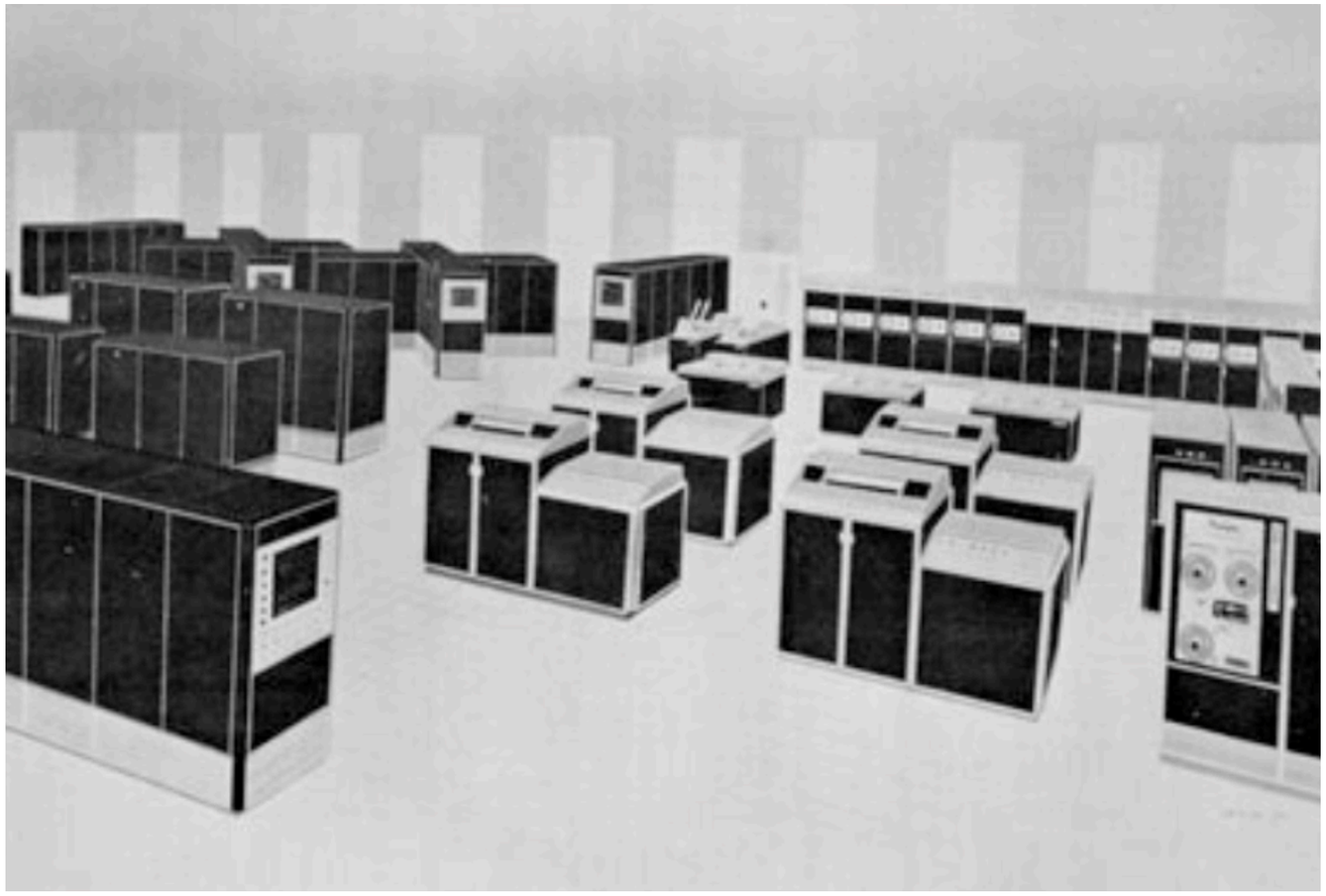
集成电路的出现，更多的高级编程语言

- 计算复杂度提升 ➔ 需要抽象层简化开发
  - 多道程序设计 (multiprogramming) 和进程 (process)
  - 一个进程在执行 I/O 时可以将 CPU 让给另一个进程
  - 需要防止不同进程之间的干扰 (保护、并发)
- 多任务需求 ➔ 需要资源调度和公平性
  - 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
  - 资源的分配、保护和管理需求

1965～1980: 集成电路时代

Multics (Multiplexed Information and Computing Service) 分时操作系统
[1965 @ MIT, Bell Labs and GE]

- 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

第一个可移植的操作系统 Unix [1970 @ Bell Labs]

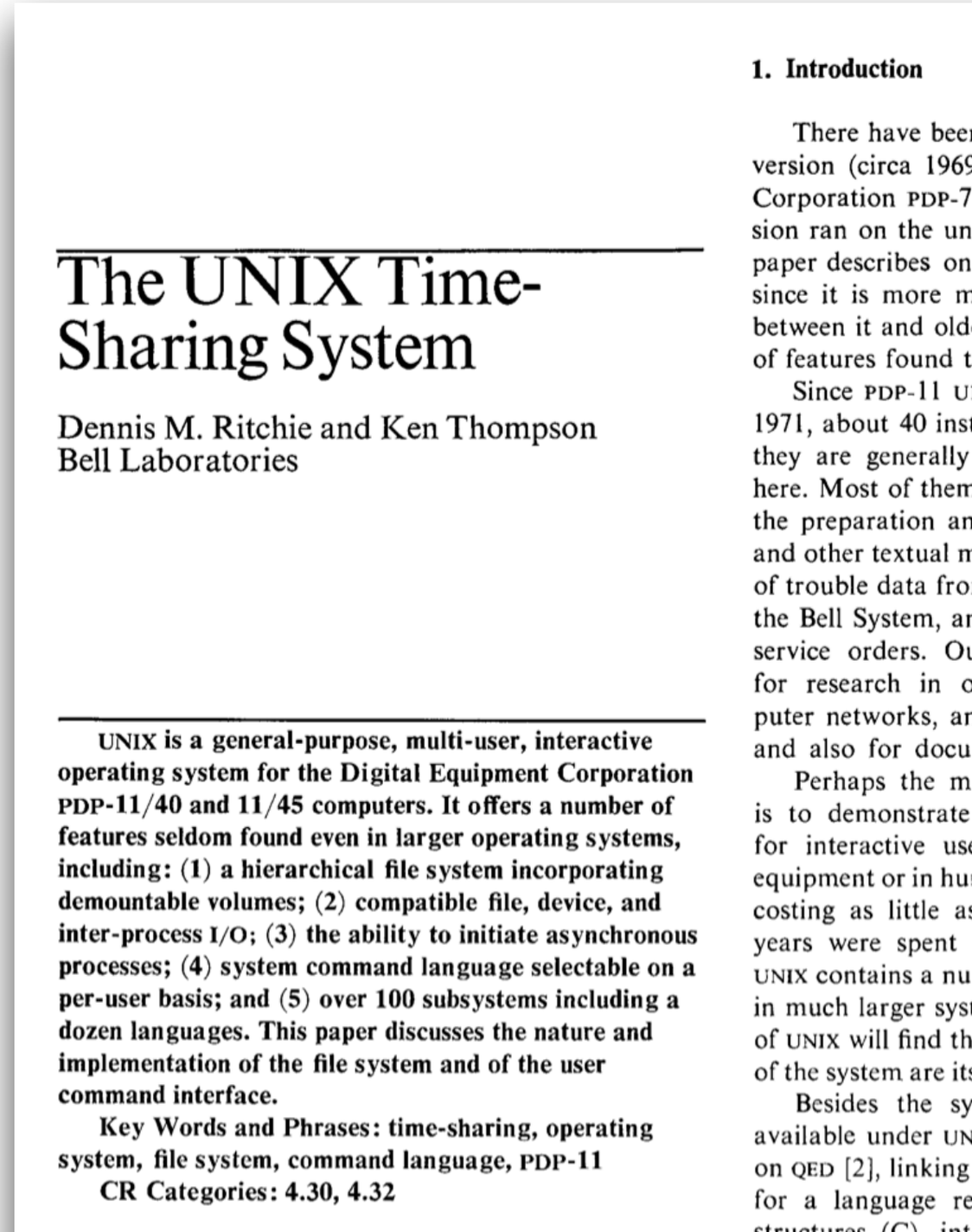
- 基于 Multics，大部分由 C 语言编写
- 很多操作系统概念进一步完善，奠定了现代操作系统的形态
- Ken Thompson 和 Dennis Ritchie 因 Unix 和 C 获得1983 年图灵奖

Unix on PDP-11

1965～1980: 集成电路时代

第一个可移植的操作系统 Unix [1970 @ Bell Labs]

- Unix 高度模块化的设计影响深远
  - 操作系统应该提供简单的工具，每个工具具有有限且明确的功能
- IEEE 基于 Unix 设计了 POSIX 标准 (Portable OS Interface for uniX)
  - 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

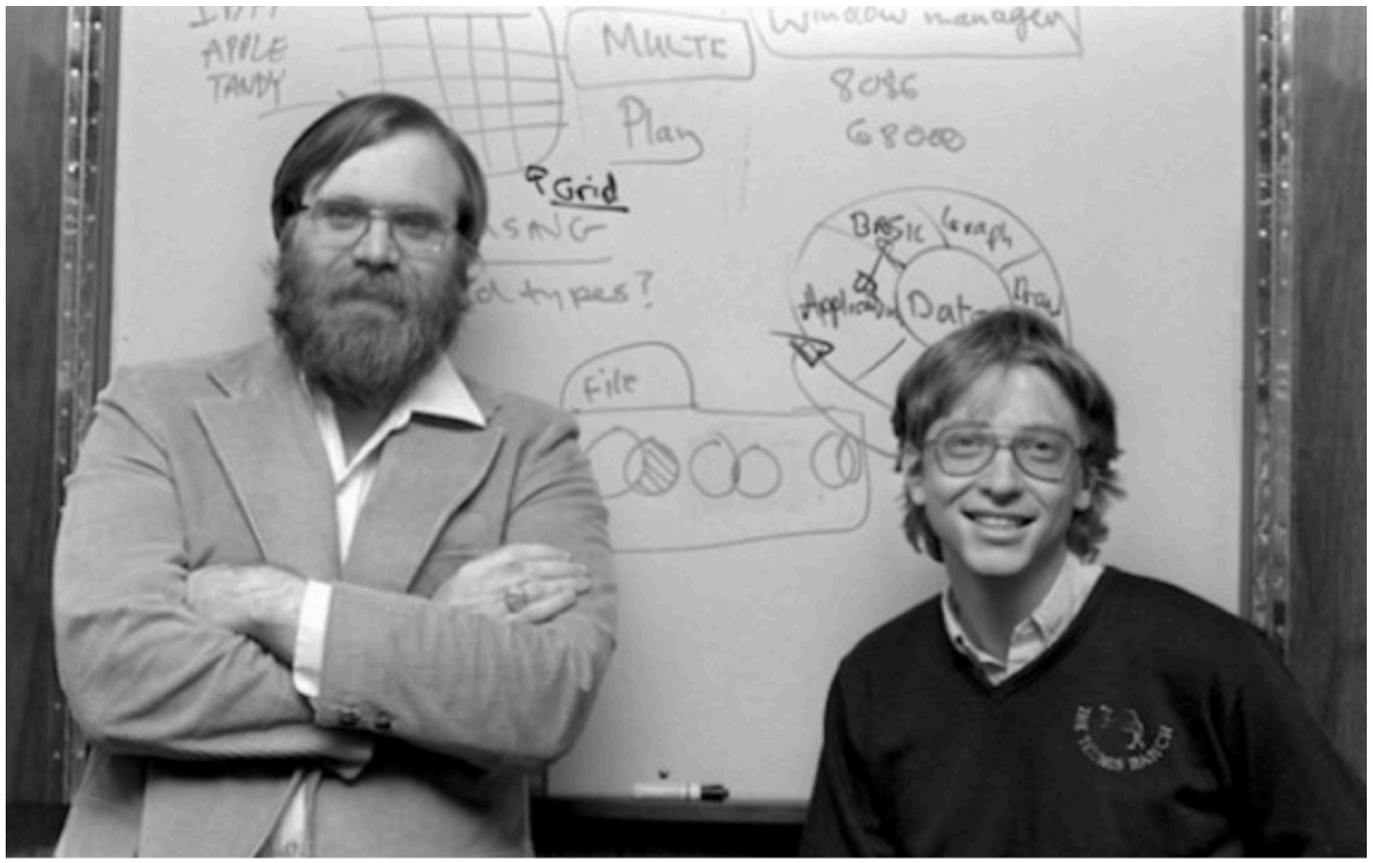
1. Introduction

There have bee
version (circa 196
Corporation PDP-7
sion ran on the un
paper describes on
since it is more m
between it and old
of features found t
Since PDP-11 U
1971, about 40 inst
they are generally
here. Most of them
the preparation an
and other textual m
of trouble data fro
the Bell System, ar
service orders. Ou
for research in c
puter networks, ar
and also for docu
Perhaps the m
is to demonstrate
for interactive use
equipment or in hu
costing as little as
years were spent
UNIX contains a nu
in much larger syst
of UNIX will find th
of the system are its
Besides the sy
available under UN
on QED [2], linking
for a language re

1980～Present: 个人电脑时代

每个桌面一台计算机

- 大规模集成电路高速发展
- 更便宜、更快捷、并且为大众设计



IBM PC 和 MS-DOS

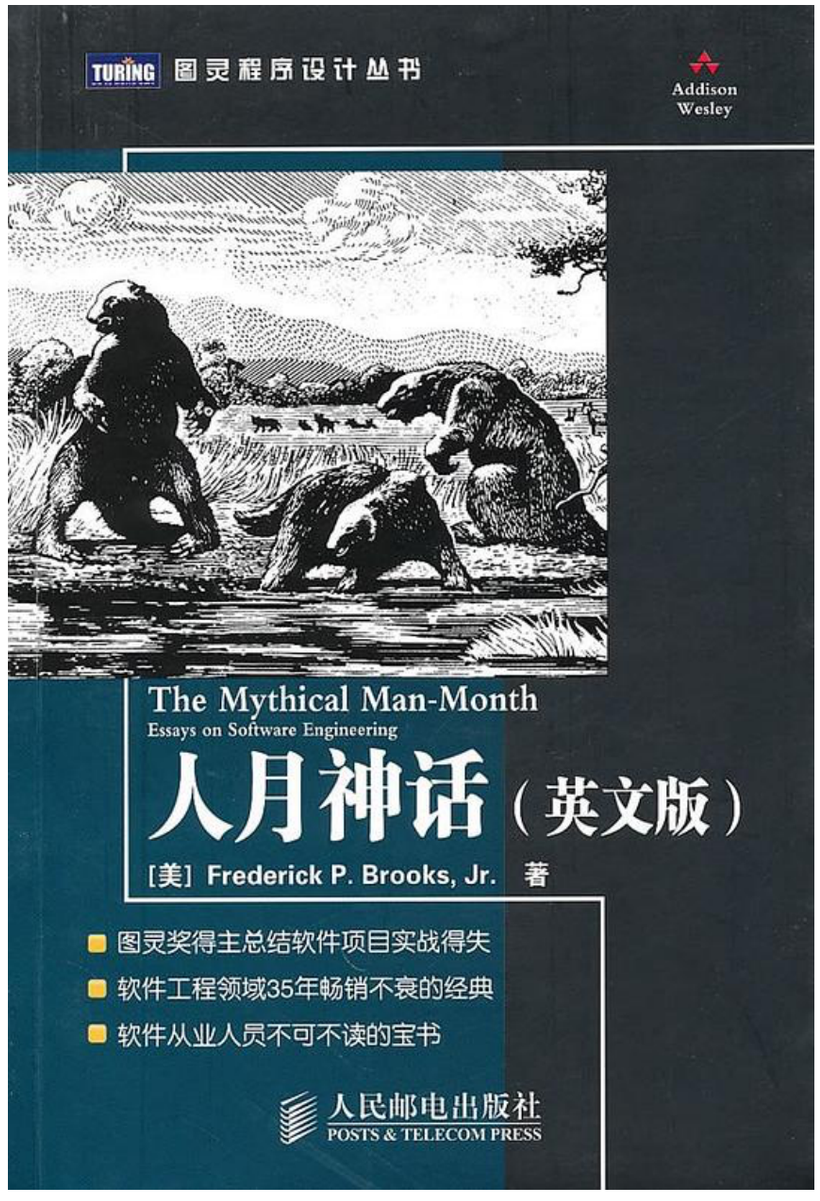
1980～Present: 个人电脑时代

Linux (1991) 和开源软件 (free to use)

- Unix 的简洁内核设计衍生出很多后续操作系统，很多大公司纷纷宣称对它拥有所有权并从中获利
- Linus 编写了自己的 Unix 版本，大量借鉴了 Unix 的设计思想但没有借鉴原始代码库

操作系统的发展

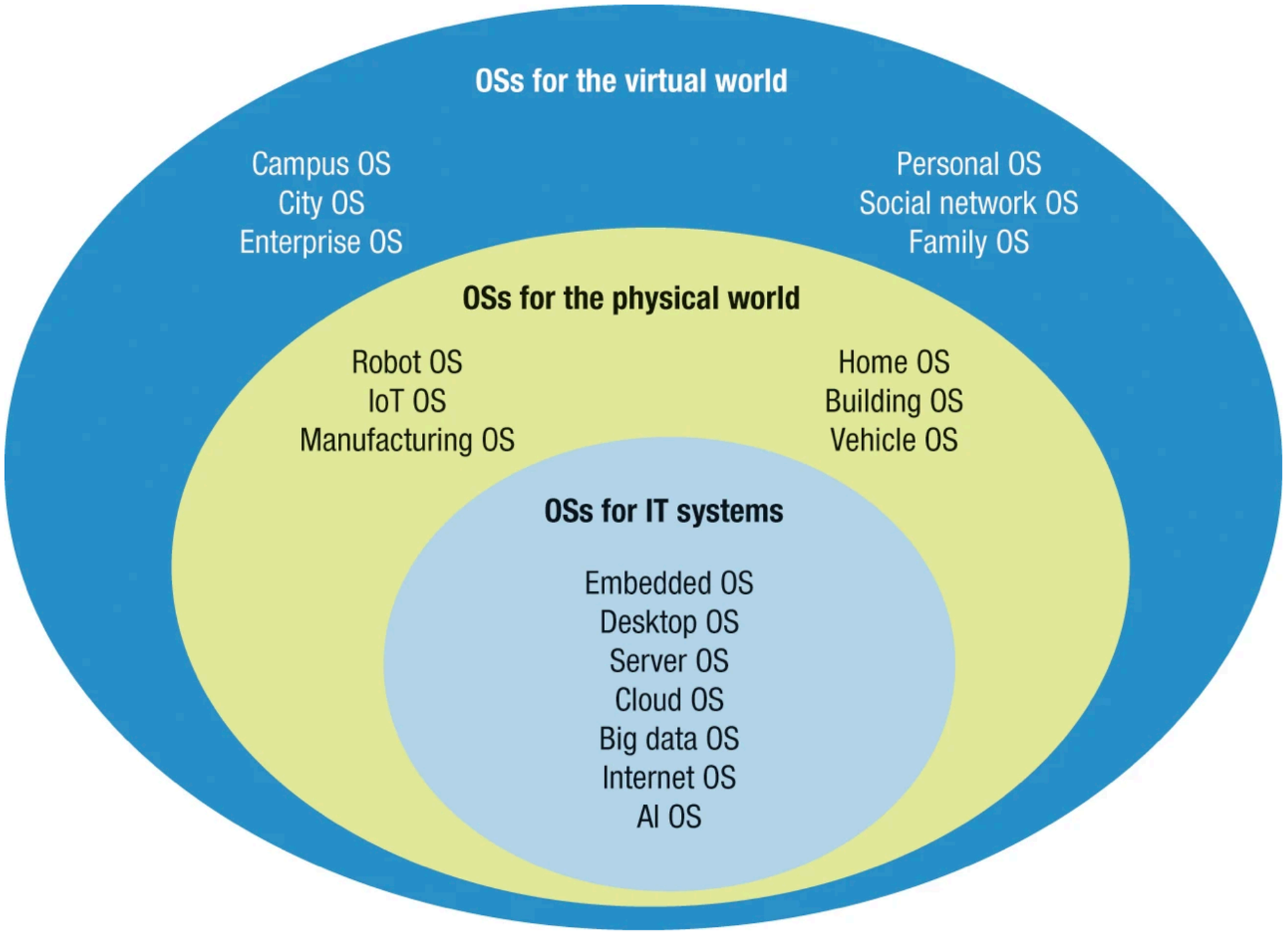
- 目前的商业操作系统已经不是一个人能够写出来的了
- 开发复杂系统中软件工程思维很重要
  - Frederick P. Brooks 在 1960s 主持和领导了 IBM System/360 系列计算机和操作系统的开发工作
  - 因在计算机结构、操作系统和软件工程方面里程碑式的贡献获得 1999 年图灵奖



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 (asymmetric 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

一个通用操作系统如何支撑起我们耳熟能详的各种应用场景

- **管理资源** (Sharing Resources)
  - 应用程序的启动、切换、调度和销毁
  - 资源的分配和共享、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简洁易用的 APIs

操作系统中的抽象

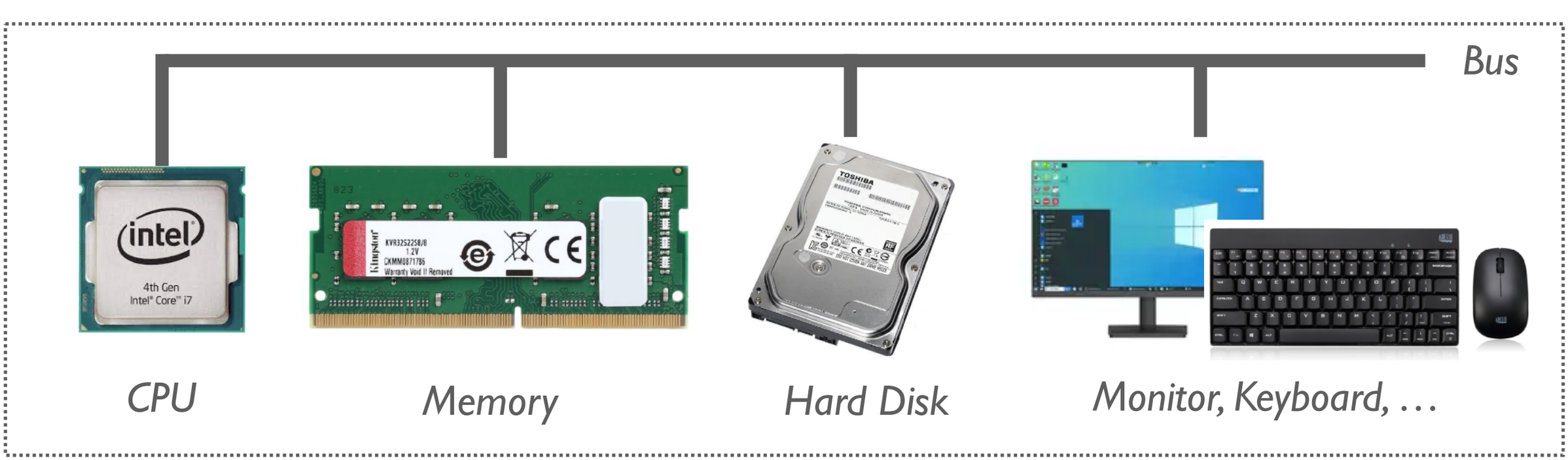
通过抽象 (Abstraction) 形成复杂对象的一种简化表示形式

- 隐藏复杂的底层物理实现细节，提供简洁易用的逻辑接口
  - 设计一系列对象 (object)
  - 为不同对象提供相关的操作 (operation)

操作系统中的抽象

线程
Thread

—个指令执行序列



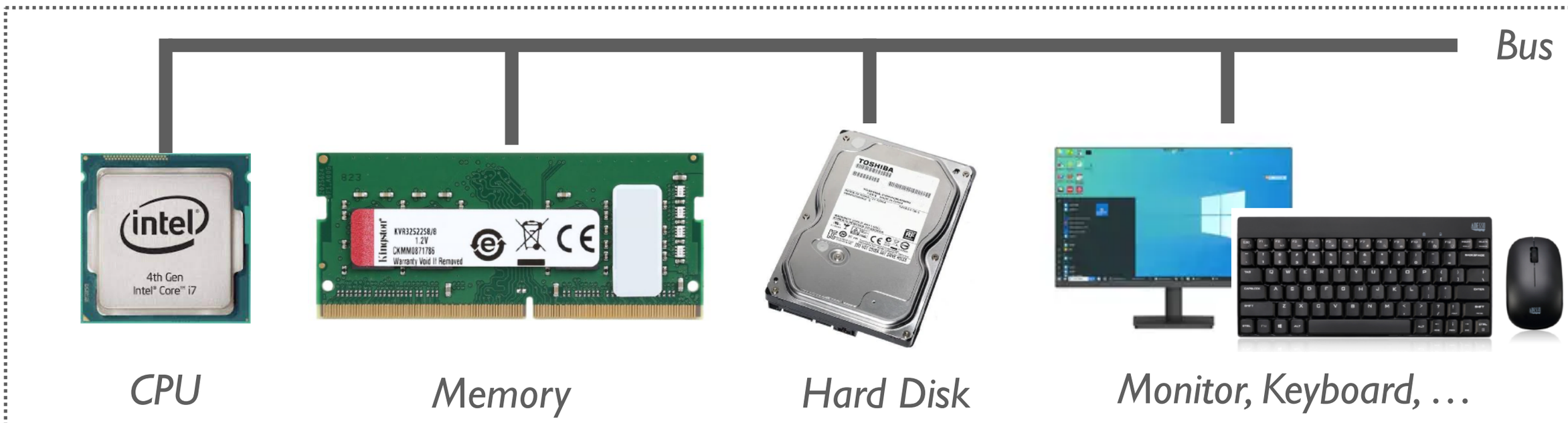
计算机硬件

操作系统中的抽象

线程
Thread

地址空间
Address Space

一个连续且充足的寻址范围



计算机硬件

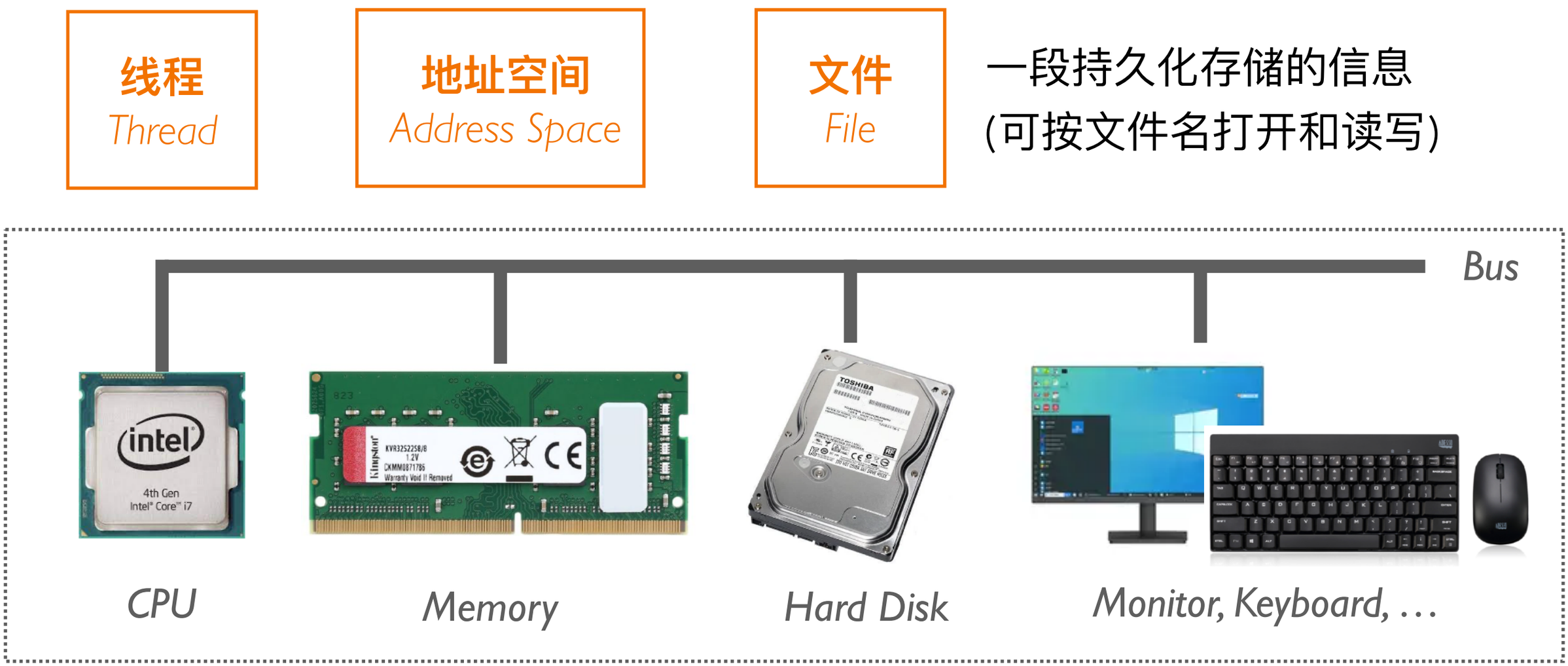
操作系统中的抽象

线程
Thread

地址空间
Address Space

文件
File

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

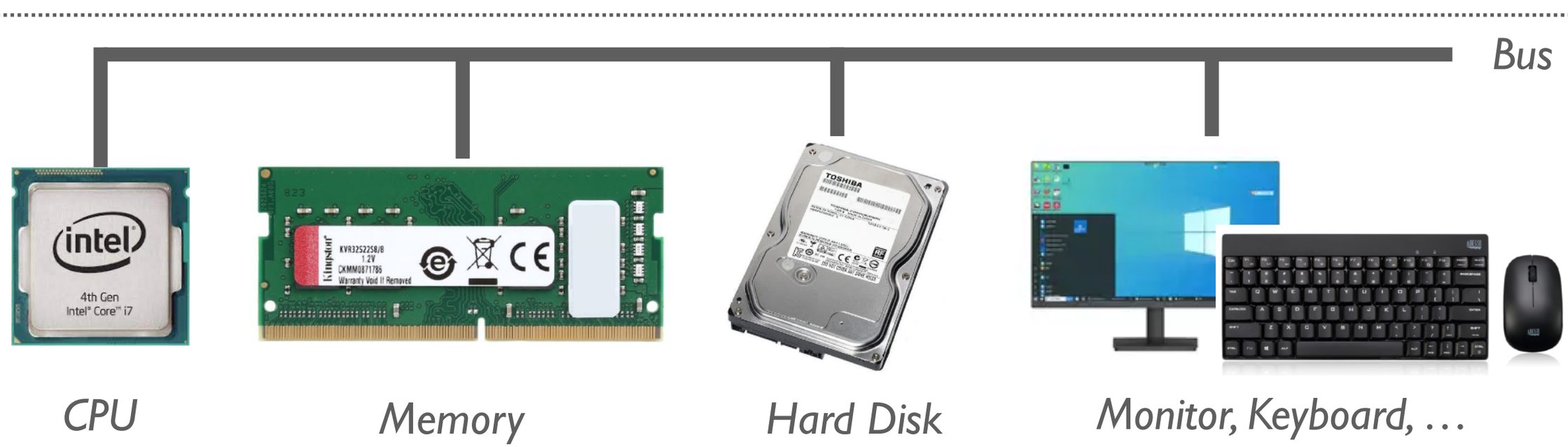
线程
Thread

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



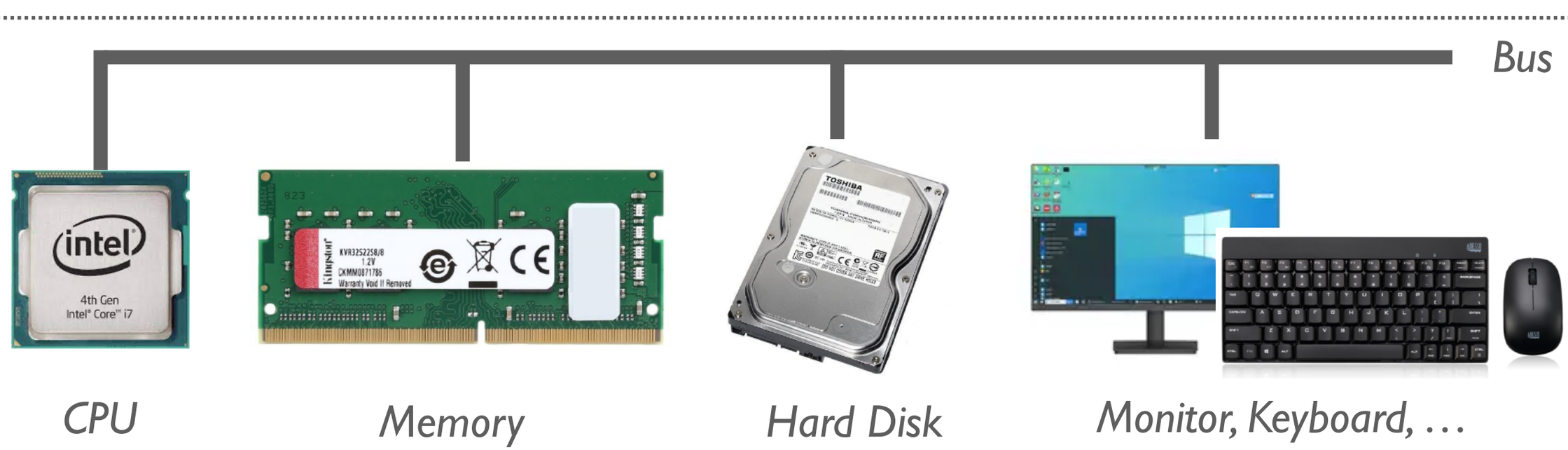
计算机硬件

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

操作系统中的虚拟化

通过虚拟化 (Virtulization) 将有限物理资源转化为多个虚拟资源实例

- 不断创建对象，为每个应用程序提供一个 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
- 需要协调抽象对象和物理现实之间的 "映射" 关系

操作系统中的虚拟化

App 1

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*

App 2

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*

App 3

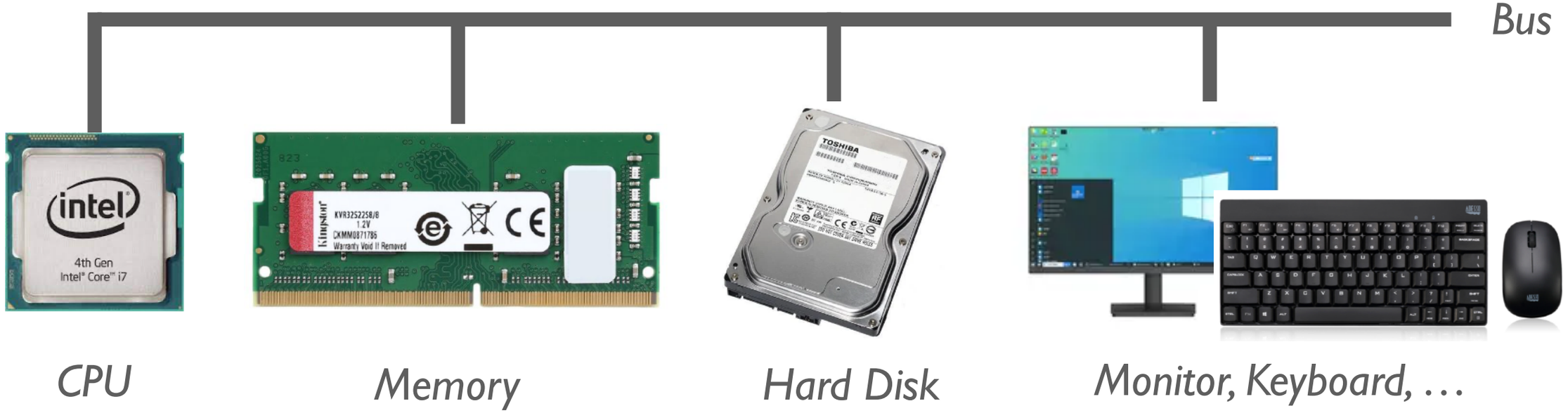
进程 (Process): 一个正在运行的应用程序

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

App 4

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

多个执行序列在有限个 CPU 上交替执行

多个地址空间共享一个有限大小的物理内存

操作系统中的保护

通过保护 (Protection) 防止物理资源共享情况下的相互干扰

- 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
  - 用户程序不能执行任意指令 (直接与硬件交互)
  - 用户程序不能修改其它程序或操作系统内核的数据
- 实现虚拟化的必然要求
- 还需要考虑控制和效率之间的平衡

操作系统中的保护

对 CPU 的保护 ➡️ 区分用户态 (User Mode) 和内核态 (Kernel Mode)

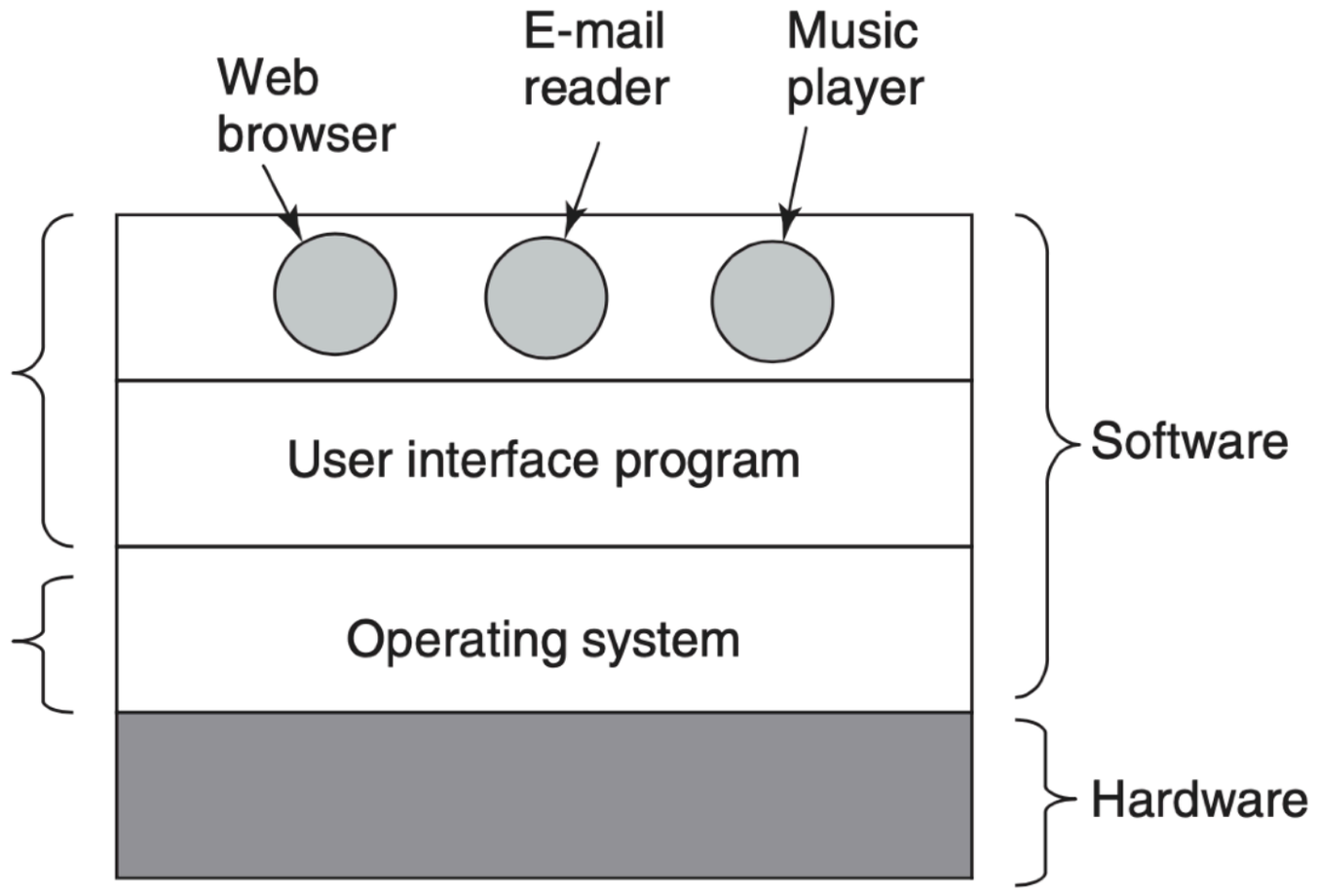
User Mode

在受限指令集下执行

↓↑

Kernel Mode

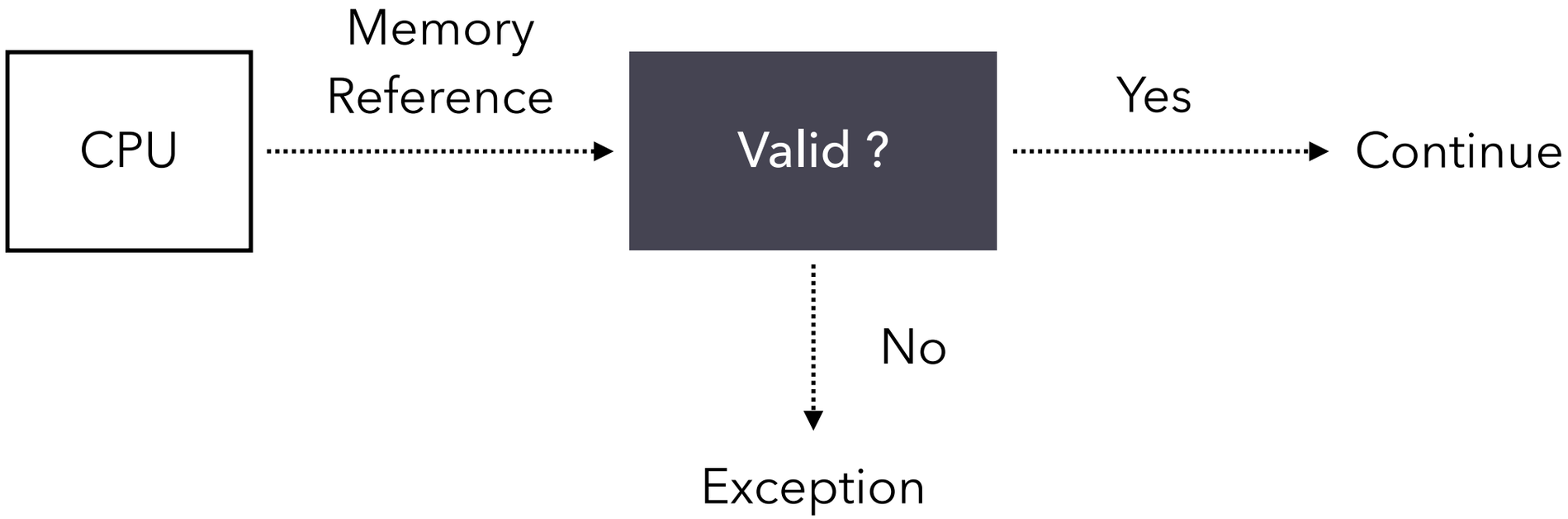
可执行与硬件交互、系统管理等相关的特权指令



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*)

操作系统中的保护

对物理内存的保护 ➡️ 构建虚拟地址 (Virtual Address) 向物理地址 (Physical Address) 的转换



向应用程序提供服务

在用户程序受限运行的情况下，操作系统需要通过特定的接口向用户程序提供服务

main.c

Run

```c
#include <stdio.h>

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

```

Output

Clear

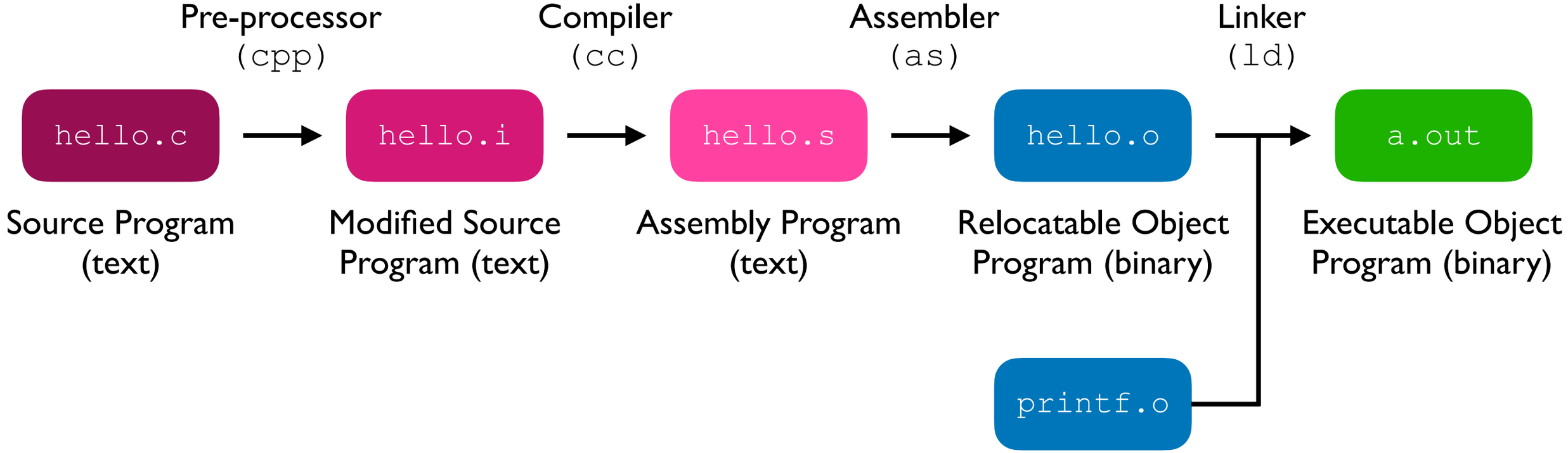
```
/tmp/1g0gu1dEUP.o
Hello World


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

回顾 Hello World

编译阶段：text ➔ binary (存储在磁盘)

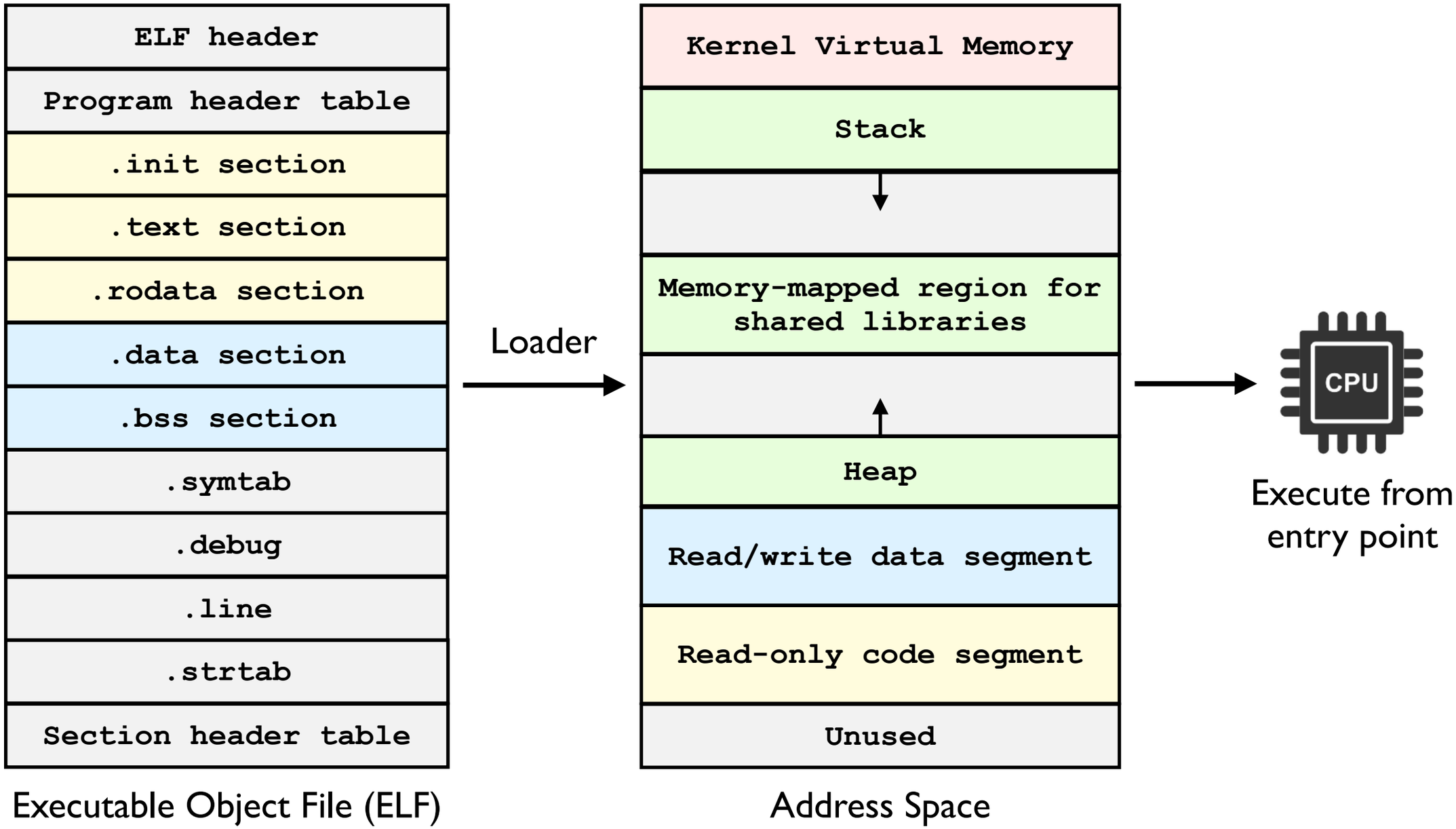
```
gcc hello.c
```



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 有很多内容

- 可通过 `objdump -d` 查看 a.out 对应的汇编代码
- 加上 `-static` 静态链接 libc 会产生更多的代码

可以尝试手动控制编译流程

- 预处理 `gcc -E hello.c > hello.i`
- 编译为汇编代码 `gcc -S hello.i -o hello.s`
- 汇编为目标文件 `gcc -c hello.c -o hello.o`
- 运行 `ld hello.o` 进行链接
  - 产生一个 error，找不到 puts (不知道怎么链接 printf)
  - 删除 printf 后可以链接，但执行产生了 Segmentation Fault ❌

回顾 Hello World

```
Disassembly of section .text:

0000000000401000 <main>:
  401000:       f3 0f 1e fa                 endbr64
  401004:       55                          push   %rbp
  401005:       48 89 e5                    mov    %rsp,%rbp
  401008:       b8 00 00 00 00              mov    $0x0,%eax
  40100d:       5d                          pop    %rbp
  40100e:       c3                          ret
```

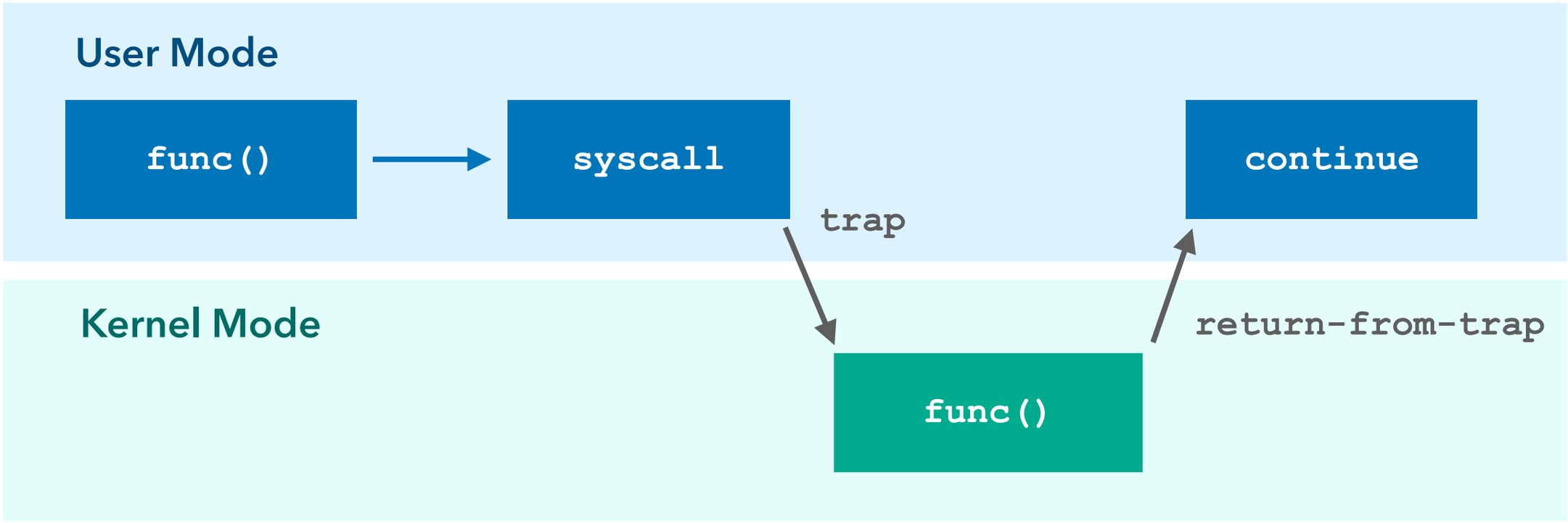
如何让程序停下来?

- 程序内部的计算指令不行 (mov / add / push / call / ret 等)
- 必须借助程序外部 (即操作系统) 的帮助

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
- 将系统 (程序自身状态) 的控制权交给操作系统



系统调用

系统调用接口 (与硬件平台相关)

- 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，然后执行一条特殊指令 (`syscall` in x86-64)

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/

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
```

最小 Hello World 的实现

```
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

用户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` 来追踪程序运行过程中产生的系统调用

```
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` 工具的实现主要利用了 `ptrace()` 系统调用，其能使操作系统内核在被追踪程序每次进入和退出内核时暂停程序的运行，从而可以查看和打印被追踪程序的状态 (各种 CPU 寄存器的值)

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

操作系统上的程序

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 (API)

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