FDU 计算机组成与体系结构 0. An Introduction

本文参考以下教材:

- Computer Systems: A Programmer's Perspective (R. Bryant & D. O'Hallaron)
- 深入理解计算机系统 (R. Bryant & D. O'Hallaron) 龚奕利 & 贺莲 (译) 第 1 章
- 计算机组成与系统结构 (袁春风、唐杰、杨若瑜、李俊) 第 1 章

欢迎批评指正!

0.0 Introduction

计算机系统由**硬件** (hardware) 和**系统软件** (systems software) 构成,
它们共同工作来运行**应用程序** (application program).

考虑 `hello` 程序:

```c
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

我们通过跟踪 `hello` 程序的**生命周期**: 从被创建开始，到在系统上运行，输出信息，最后终止,
来简要地介绍一些逐步出现的关键概念.

0.1 位序列 (Sequence of bits) + 上下文 (Context) = 信息 (Information)

`hello` 程序的生命周期始于一个名为 `hello.c` 的**源文件** (source file)
它实际上是一个由 `0,1` 组成的位序列 (sequence of bits),
每 8 个连续的**位** (bit) 被组织成一个**字节** (byte),
源文件 `hello` 的文本字符就是以字节为单位进行存储的.
现代计算机系统中，文本字符最常用的编码方式为 **ASCII 编码**
(ASCII, American Standard Code for Information Interchange, 美国信息交换标准代码)
即用单字节的无符号整数值 $(0 \sim 255)$ 来编码每个文本字符
(但实际上只用了前 7 位 $(0 \sim 127)$ 编码了 128 个常用字符，参见 ASCII 码对照表)
源文件 `hello.c` 的 ASCII 码如图所示:

#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	<sp>	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	<sp>	<sp>	<sp>	<sp>	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	<sp>	w	o	r	l	d	\	n	"	)	;	\n	}
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	125

图 1.9　hello.c 源程序文件的表示

源文件 `hello.c` 是以**字节序列** (a sequence of bytes) 的方式存储在**文本文件** (text file) 中的.
(区别于**二进制文件** (binary file))
源文件 `hello.c` 的表示方法说明了一个基本思想:
位序列 (sequence of bits) + **上下文** (context) = **信息** (information)
系统中的所有**数据对象** (data objects) 都是以**位序列**的形式存储的,
区分不同数据对象的唯一方法是我们读取这些数据对象时的**上下文** (即**解读方式**)
例如一个位序列可能被解读为整数、浮点数或者机器指令等等.

我们需要重点学习整数、浮点数的机器表示方式,
它们与实际的整数、实数是不同的，它们是真值的有限近似值.

0.2 源文件被翻译为可执行目标文件

我们人类很容易读懂源文件 `hello.c` 中的语句,
这类语言 (例如 C 语言) 称为**高级语言** (High-level Language),
或者更贴切地，**人类友好语言** (Human-Friendly Language)
高级语言是抽象程度较高的编程语言，它们隐藏了机器语言的复杂性,
更接近人类的自然语言或数学符号，从而使得编程和理解代码变得更加容易.

然而，为了在计算机系统上运行 `hello` 程序,
每条 C 语句都需要被翻译为一系列机器指令，以对计算机硬件进行精细控制,
最终得到二进制文件 `hello`，称为**可执行目标文件** (executable object file).
这个翻译过程可分为四个阶段:

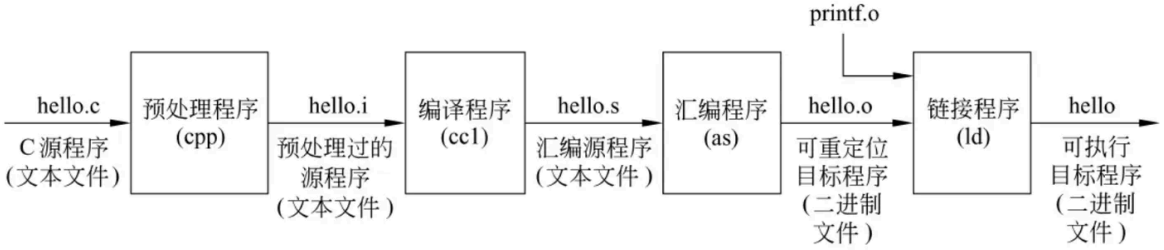


图 1.10　hello.c 源程序文件到可执行目标文件的转换过程

- **① 预处理阶段 (Preprocessing Phase):**
  预处理器 (preprocessor) `cpp` 根据**预处理指令** (preprocessor directive) (例如 #include 和 #define)
  修改源文件得到**预处理器输出文件** (preprocessor output file)
  其中 `#include <stdio.h>` 指令告诉预处理器 `cpp` 读取头文件 `stdio.h` 的内容,

并将其直接插入源文件 `hello.c` 得到预处理器输出文件 `hello.i`
它仍是文本文件.

- **② 编译阶段 (Compilation Phase):**
  编译器 (compiler) `ccl` 将预处理器输出文件 `hello.i` 翻译为 `hello.s`,
  称为**汇编语言文件** (assembly language file)，它仍是文本文件.
  汇编语言是非常有用的，因为它为不同高级语言的不同编译器提供了通用的输出语言.
  汇编代码比源代码更接近机器码，但仍保持一定程度的可读性，使得人类能够理解和修改.

- **③ 汇编阶段 (Assembly Phase):**
  汇编器 (assembler) `as` 将汇编语言文件 `hello.s` 翻译成机器码,
  但这些机器码还没有被完全链接成一个可执行程序，储存在 `hello.o` 中,
  称为**目标文件** (object file)，它是一个二进制文件.

- **④ 链接阶段 (Linking Phase):**
  链接器 (linker) `ld` 将一个或多个目标文件与所需的库合并，生成一个**可执行目标文件**.
  其中 `printf("hello, world\n");` 调用了 `printf` 函数,
  链接器需要将标准 C 库中的目标文件 `printf.o` 以某种方式合并到目标文件 `hello.o` 中,
  得到可执行目标文件 `hello`，它是一个二进制文件，可被加载到内存中由系统执行.

不同层次语言之间的等价转换如图所示:

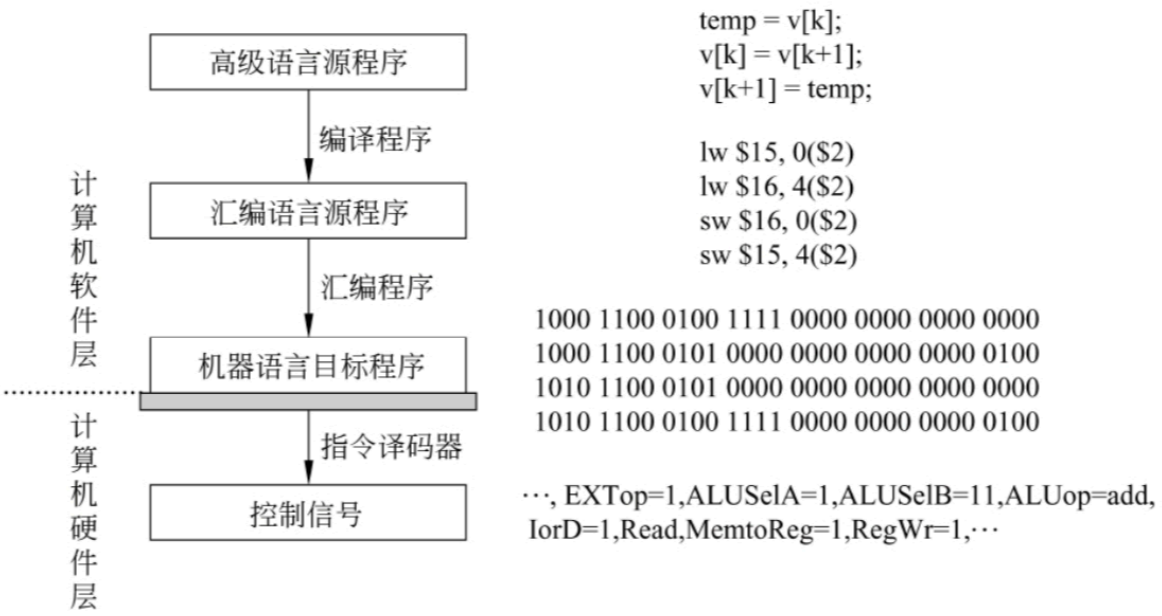
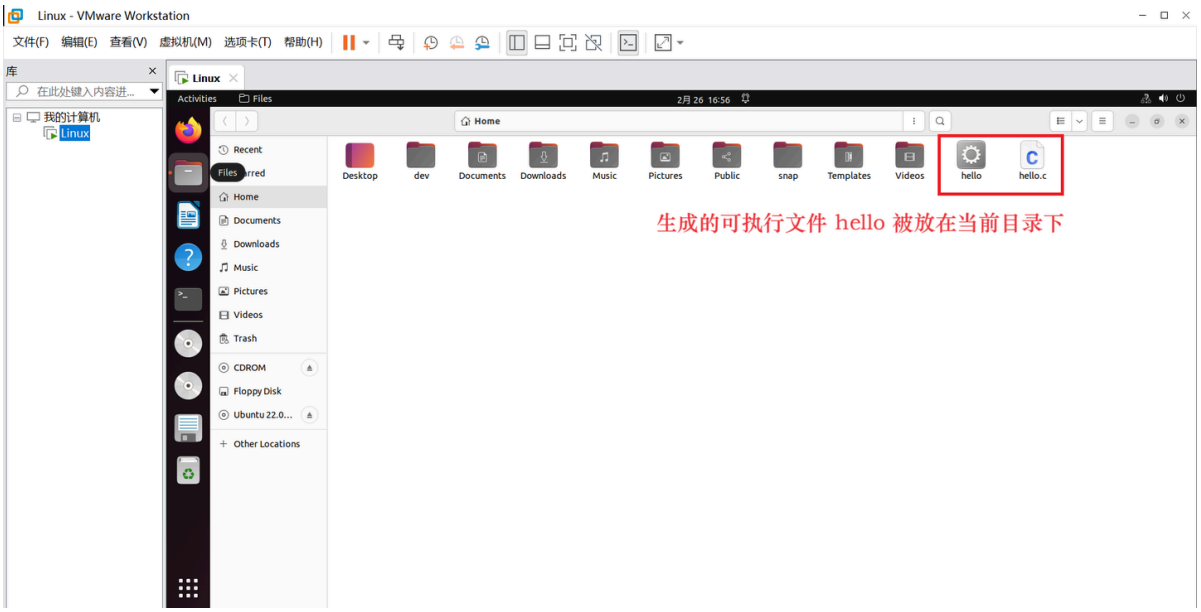


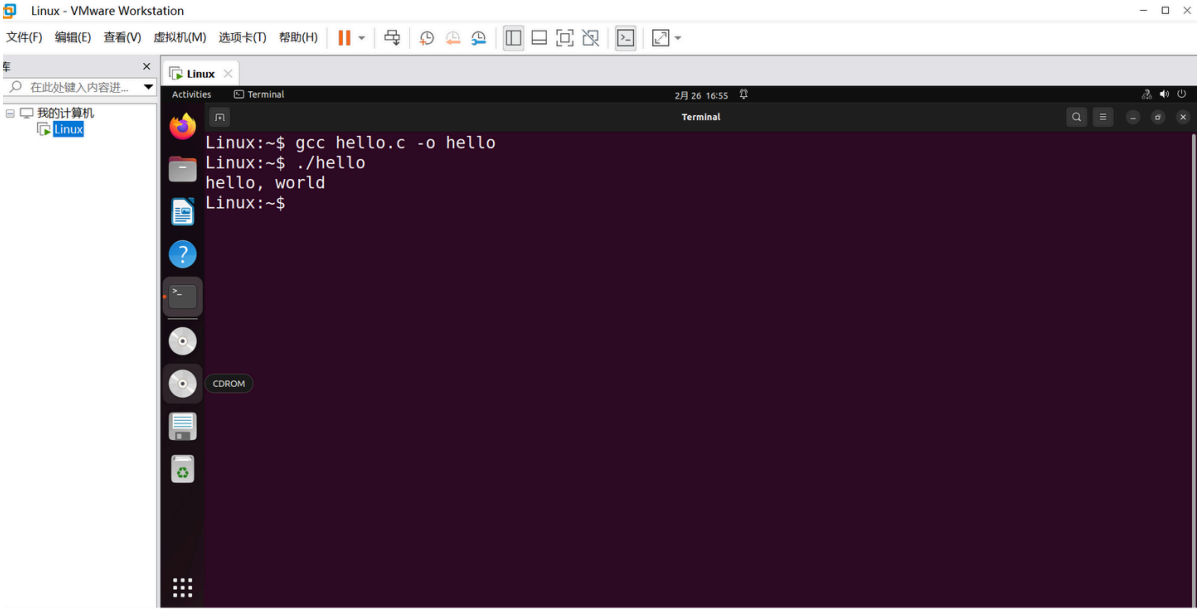
图 1.12　不同层次语言之间的等价转换

我们不妨实际操作一下:
在 Linux 系统的 shell 中 (shell即命令行解释器，它是用户与操作系统之间的接口),
我们可以通过输入 `gcc hello.c -o hello` 指令,
来调用 `GCC` 编译器将源文件 `hello.c` 编译为可执行文件 `hello`，放在当前目录中.
如图所示:



0.3 处理器读取并解释内存中的指令

此时，源文件 `hello.c` 已经被编译系统翻译成了可执行文件 `hello`
我们继续在 Linux 系统的 shell 中输入 `./hello` 指令，
系统便会加载并运行 `hello` 程序，然后等待程序终止.
`hello` 程序会在屏幕上输出它的消息 `hello, world`，如图所示:



0.3.1 计算机系统的硬件组成

为了理解运行 `hello` 程序时发生了什么，我们需要了解计算机系统的硬件组成: (如图所示)

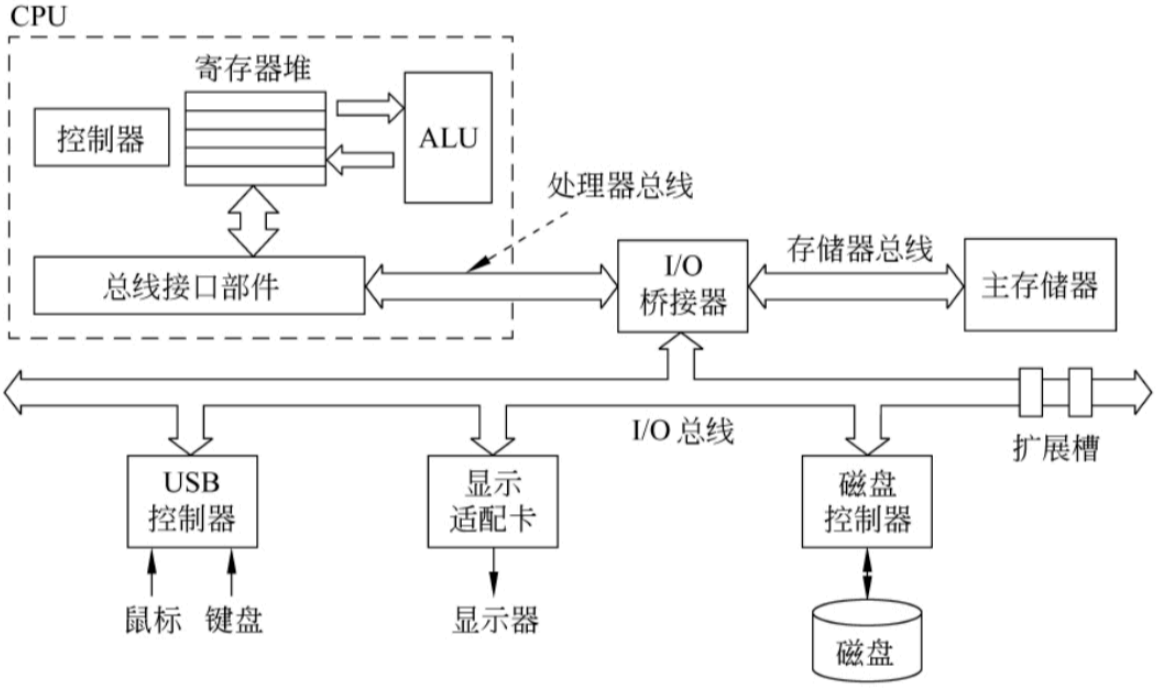


图 1.2　一个典型计算机系统的硬件组成

- **(Ⅰ) 总线 (bus):**
  总线负责在各个部件之间传递信息，根据功能分为三类: **地址总线**、**数据总线**、**控制总线**.
  总线通常被设计成单次传输固定长度的**字节块** (chunks of bytes)，也就是**字** (word).
  字长 (word size)，即字中的字节数，是一个基本的系统参数，
  现代的大多数计算机的字长要么是 4 字节 (32 位)，要么是 8 字节 (64 位)，
  本门课程里，我们不对字长做任何固定的假设，具体情况具体分析.
- **(Ⅱ) I/O 设备:**
  I/O 设备 (输入/输出设备) 是计算机系统与外部世界的信息交换通道.
  我们的示例系统包含了四个 I/O 设备: 键盘、鼠标、显示器、磁盘
  最开始，可执行文件 `hello` 就存放在磁盘上.
  这门课程里，我们需要重点学习磁盘是如何进行输入/输出的.
- **(Ⅲ) 主存 (MM, Main Memory, 主存储器):**
  主存用于存储和提供执行中程序的指令和数据，
  它可以直接与处理器交互 (这点区别于磁盘等辅助存储器).
  主存通常由一组**动态随机存取存储器** (DRAM, Dynamic Random Access Memory) 构成，
  即通过**电容**存储信息 (需要定期刷新来保持数据，断电后会损失所有数据)，
  且支持**随机访问** (意味着处理器可以在几乎相同的时间内访问主存中的任意位置)
  它是一个线性的字节数组，每个字节都有唯一的**地址** (数组索引)
- **(Ⅳ) 处理器 (CPU, Central Processing Unit, 中央处理单元):**
  处理器是解释、执行主存中指令的单元.
  它至少由 7 部分构成:
  - **① 程序计数寄存器 (PC, Program Counter):**
    一种专用寄存器，用于存储下一条要执行的指令的内存地址.
  - **② 控制器 (CU, Control Unit, CU):**
    负责从内存中取出指令，解码并执行它们.
    (根据解码得到的控制码，控制其他组件的操作以及数据在 CPU 内部的流动)
  - **③ 算术逻辑单元 (ALU, Arithmetic Logic Unit):**
    负责执行所有的算术运算 (e.g. 加、减、乘、除) 和逻辑运算 (e.g. 与、或、非、异或)

- **④ 指令寄存器 (IR, Instruction Register):**
  一种专用寄存器，用于存储当前正在被执行的指令.
- **⑤ 寄存器堆 (Register Files):**
  一系列通用寄存器，用于临时存储指令、数据和运算结果.
- **⑥ 缓存 (Cache):**
  一种高速存储器，用于临时存储主存中频繁访问的数据和指令 (我们后续会重点学习 Cache)
- **⑦ 总线接口单元 (BIU, Bus Interface Unit):**
  负责 CPU 与其他部件 (e.g. 主存、I/O 设备) 之间的通信，
  管理着三类总线: 地址总线、数据总线、控制总线.

0.3.2 hello 程序的执行过程 (不可避免地有所简化)

(Ⅰ) 从键盘上读取用户指令 `./hello`:
最开始的时候，shell 正等待着我们输入一个指令.
当我们通过键盘输入字符串 `./hello` 时，shell 将它们逐一读入寄存器，然后存放在主存中.

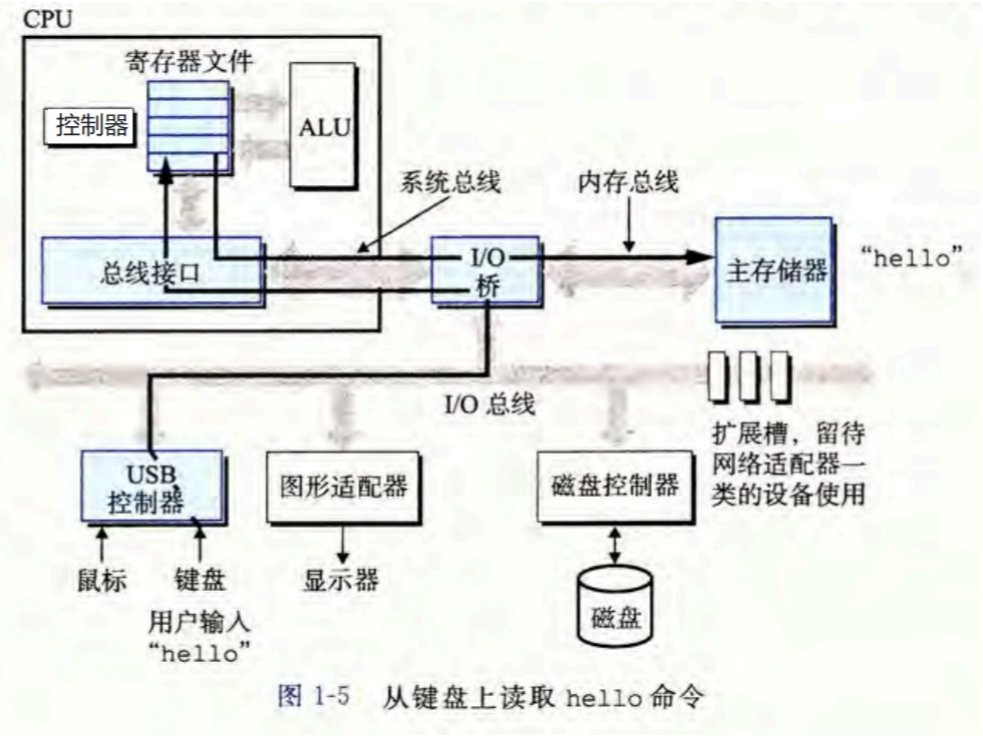


图 1-5 从键盘上读取 hello 命令

(Ⅱ) 将可执行文件 `hello` 从磁盘加载到主存:
当我们敲下回车键时，shell 就知道我们已经结束了指令的输入.
然而 shell 发现该指令的第一个单词不是内置的 shell 命令，
于是 shell 会假设这是一个可执行文件的名字，并在当前目录中寻找名为 `hello` 的可执行文件.
不出所料，shell 成功找到了 `hello`，
并执行一系列指令来将 `hello` 中的二进制代码和数据从磁盘加载到主存 (如图所示).

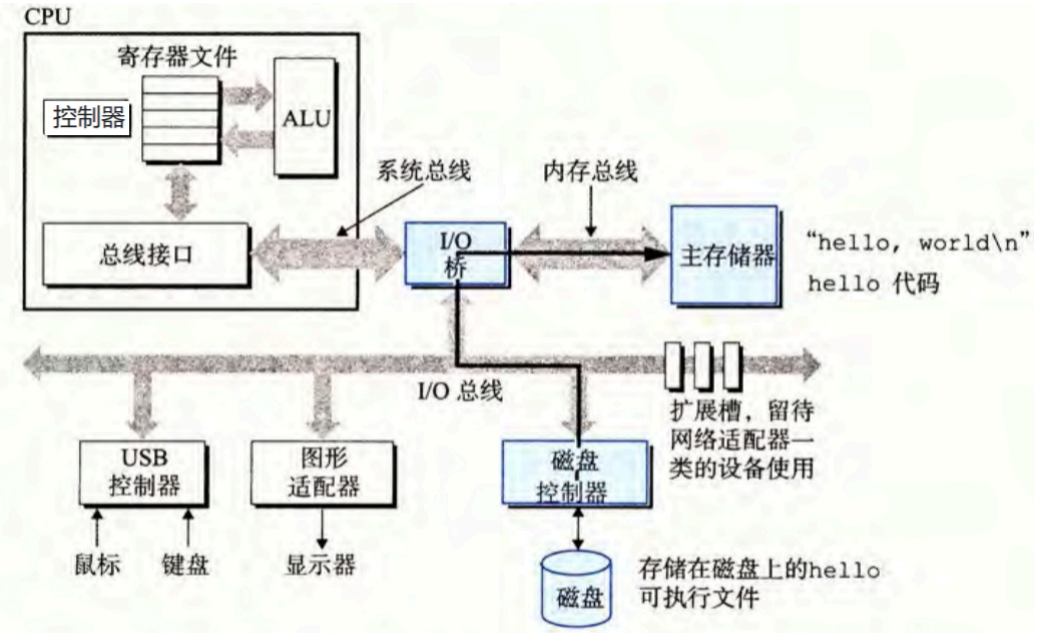


图 1-6　从磁盘加载可执行文件到主存

(III) 执行 `hello` 程序，输出字符串 `"hello, world"` 到显示屏:

一旦可执行文件 `hello` 中的二进制代码和数据都被加载到了主存，
CPU 就开始执行 `hello` 程序中的机器指令，
这些指令将字符串 "hello, world" 从主存复制到寄存器堆中，
再从寄存器堆复制到图形适配器，最终显示在屏幕上 (如图所示).

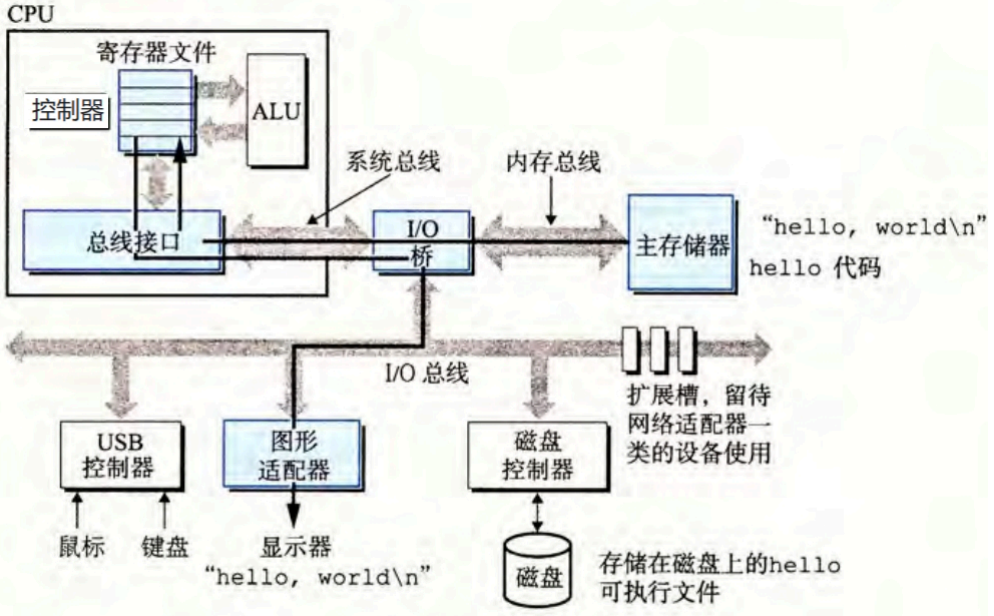


图 1-7　将输出字符串从存储器写到显示器

上述分析中，我们只停留在处理器的**指令集架构** (instruction set architecture) 上，
它描述的是 `hello` 程序中每条二进制指令的效果.
然而我们并没有讨论处理器具体是如何实现 `hello` 程序的每条二进制指令的，
实际上，其机制相当复杂，我们称之为处理器的**微体系结构** (micro-architecture)
后续课程中，我们先讨论指令集架构，再讨论微体系结构.

0.4 计算机系统的性能评价

0.4.1 计算机系统的性能指标

吞吐率 (Throughput) 和**执行时间** (Execution Time) 是衡量计算机系统性能的两个互补指标，
它们从不同的维度衡量系统性能:

- **① 吞吐率:**
  即系统在单位时间内能够完成的最大工作量，通常用于衡量系统的并行处理能力.
  在多任务和并发执行环境中，提高吞吐率是提升系统性能的关键.
- **② 执行时间:**
  即完成单个任务 (无论是在串行还是并行环境下执行) 所需的时间，它是一个更通用的性能指标.
  在需要快速响应的应用中，减少执行时间是提升系统性能的关键.

某些情况下，提高吞吐率可能会增加单个任务的执行时间，因为系统需要在多个任务间分配资源;
反过来，优化执行时间也可能会牺牲并行处理能力，从而降低系统的吞吐率.
若不考虑应用背景而直接比较计算机性能，则通常用执行时间来衡量.

考虑某一用户程序的执行时间:
它可分为 CPU 时间和其他时间 (等待 I/O 操作完成的时间，或 CPU 执行其他用户程序的时间)
而 CPU 时间又分为:

- **① 用户 CPU 时间:**
  真正用于该用户程序的 CPU 时间，它可用于衡量 CPU 性能;
- **② 系统 CPU 时间:**
  为执行该用户程序而需要 CPU 运行操作系统程序的时间;

0.4.2 CPU 时间的计算

- **(Ⅰ) 时钟周期 (clock cycle) & 时钟频率 (clock rate):**
  (这门课程里，我们只考虑同步 CPU，其设计和实现相对简单; 不考虑异步 CPU)
  CPU 执行一条机器指令的过程可以被分为若干步骤，
  每个步骤都要由相应的控制信号进行控制，
  这些控制信号何时发出、作用时间多长，都要由相应的全局定时信号进行同步.
  因此，CPU 必须能够产生同步的全局定时信号，即**主脉冲信号**，其宽度称为**时钟周期**.
  时钟频率，又称**主频** (主脉冲频率)，它是时钟周期的倒数.
- **(Ⅱ) 指令的时钟周期数 (CPI, Cycles Per Instruction):**
  - **① 单周期 CPU:**
    单周期 CPU 的设计理念是让每条机器指令在一个时钟周期内完成所有的操作，
    包括指令的取出 (Fetch)、解码 (Decode)、执行 (Execute)、访问内存 (Memory Access)，以及写回 (Write back) 过程.
    这种设计模式简化了控制逻辑，使得不同指令的执行时间是一致的，而付出的代价是:
    - (i) 单周期 CPU 的时钟周期必须匹配运行最慢的指令，
      这导致运行较快的指令在时钟周期的大部分时间内等待，从而降低了整体性能;
    - (ii) 单周期 CPU 可能会导致不必要的能耗，
      因为其所有组件在每个时钟周期都处于激活状态，即使某些机器指令只需要其中的一部分组件.

- **② 多周期 CPU:**
  多周期 CPU 采用了不同的设计理念，允许机器指令的操作分散到多个时钟周期中完成，它通过减少指令的平均时钟周期数来提高性能，但这也使得控制逻辑变得更加复杂.

综合(Ⅰ)(Ⅱ)可知:
假设某程序中有 n 种不同类型的指令，
对应的指令条数为 $N_1, \ldots, N_n$，
时钟周期数分别为 $\mathrm{CPI}_1, \ldots, \mathrm{CPI}_n$
我们记**指令总数**为 $N = \sum_{i=1}^{n} N_i$
记**指令的平均时钟周期数**为 $\overline{\mathrm{CPI}} = \frac{\sum_{i=1}^{n} N_i \cdot \mathrm{CPI}_i}{\sum_{i=1}^{n} N_i} = \frac{1}{N} \sum_{i=1}^{n} N_i \cdot \mathrm{CPI}_i$
则 CPU 时间的计算公式如下:

$$\begin{aligned} \mathrm{CPU}\text{时间} &= \text{程序总时钟周期数} \times \text{时钟周期} \\ &= \left(\sum_{i=1}^{n} N_i \cdot \mathrm{CPI}_i \right) \times \text{时钟周期} \\ &= N \times \overline{\mathrm{CPI}} \times \text{时钟周期} \end{aligned}$$

我们定义 $\text{指令的平均执行时间} = \overline{\mathrm{CPI}} \times \text{时钟周期}$
相应地，我们定义:

$$\begin{aligned} \text{指令的平均执行速度} &= \frac{1}{\text{指令的平均执行时间}} \\ &= \frac{1}{\overline{\mathrm{CPI}} \times \text{时钟周期}} \end{aligned}$$

计量单位为 **MIPS** (Million Instructions Per Second, 百万指令每秒)
这个单位通常用于评估 CPU 执行固定指令集的能力，尤其是在执行整数运算方面.
类似的一个计量单位为 **MFLOPS** (Million FLOating-point operation Per Second)，
衡量的是计算机执行浮点运算能力.

0.4.3 Amdahl 定律

Amdahl 定律的主要思想是:
当我们改进系统的某个部件时，其对系统整体性能的影响取决于该部件的**重要程度**和**加速比**.

假设系统某部件 (不妨想象为 CPU) 执行时间占比为 $\alpha \in (0, 1)$ (刻画重要程度)，
改进前后的加速比为 $k > 1$ (可以简单理解为并行了 $k > 1$ 个 CPU)，
若改进前系统执行某程序所需的时间为 T_{old}，
则改进后系统的执行时间为:

$$\begin{aligned} T_{\text{new}} &= \frac{\alpha T_{\text{old}}}{k} + (1 - \alpha) T_{\text{old}} \\ &= T_{\text{old}} \left[1 - \left(1 - \frac{1}{k} \right) \alpha \right] \end{aligned}$$

因此整个系统的**加速比** (Speedup) 为 $S = \frac{T_{\text{old}}}{T_{\text{new}}} = \frac{1}{1 - (1 - \frac{1}{k})\alpha}$
显然，当 $k \to +\infty$ 时，加速比的极限 $S_\infty = \frac{1}{1-\alpha}$

Amdahl 定律的核心结论是:
程序的加速比受到其串行部分的极限制约.
即使并行的 CPU 数量无限增加，整个系统的加速比也会趋向于一个固定的上限.
这意味着，当我们增加并行 CPU 的数量时，性能提升的边际效益会逐渐减少.

同时，Amdahl 定律启发我们寻找更高效的并行算法，
专注于减少串行代码段，以提高程序的并行部分比例，从而实现更大的性能提升.

以 $\begin{cases}\alpha = 60\% \\ k = 3\end{cases}$ 的情况为例:
此时系统的加速比为 $S = \frac{1}{1-(1-\frac{1}{3})\cdot 0.6} \approx 1.67$ 倍.
虽然我们对系统的一个重要部件 (占比 60%) 做出了重大改进 (加速比为 3 倍),
但我们获得的系统加速比 (1.67 倍) 却明显小于 3 倍，
即使是极限情况的系统加速比也只有 2.5 倍.
说明想要显著加速整个系统，必须提升全系统中相当大部分组件的性能 (60% 都可能不太够).

0.4.4 Moore 定律

Moore (Intel 的创始人之一) 于1965年提出:
集成电路上可容纳的晶体管数量大约每两年 (后来修正为 18 个月) 翻一番，而成本相对固定.
这一观察被广泛解释为单位价格的 CPU 的性能每 18 个月翻倍.
尽管随着技术接近物理极限，维持 Moore 定律所预测的增长速度变得越来越困难，
但 Moore 定律仍然激励着人们不断探索，以实现更高的计算性能和效率.

THE END