

# 嵌入式系统设计与实例开发

—基于ARM微处理器与实时操作系统

## 第二讲 基本概念及设计方法

华北电力大学

控制与计算机工程学院

李东江副教授

# 本节提要

1

嵌入式系统硬件基础

2

嵌入式系统软件基础

3

嵌入式操作系统

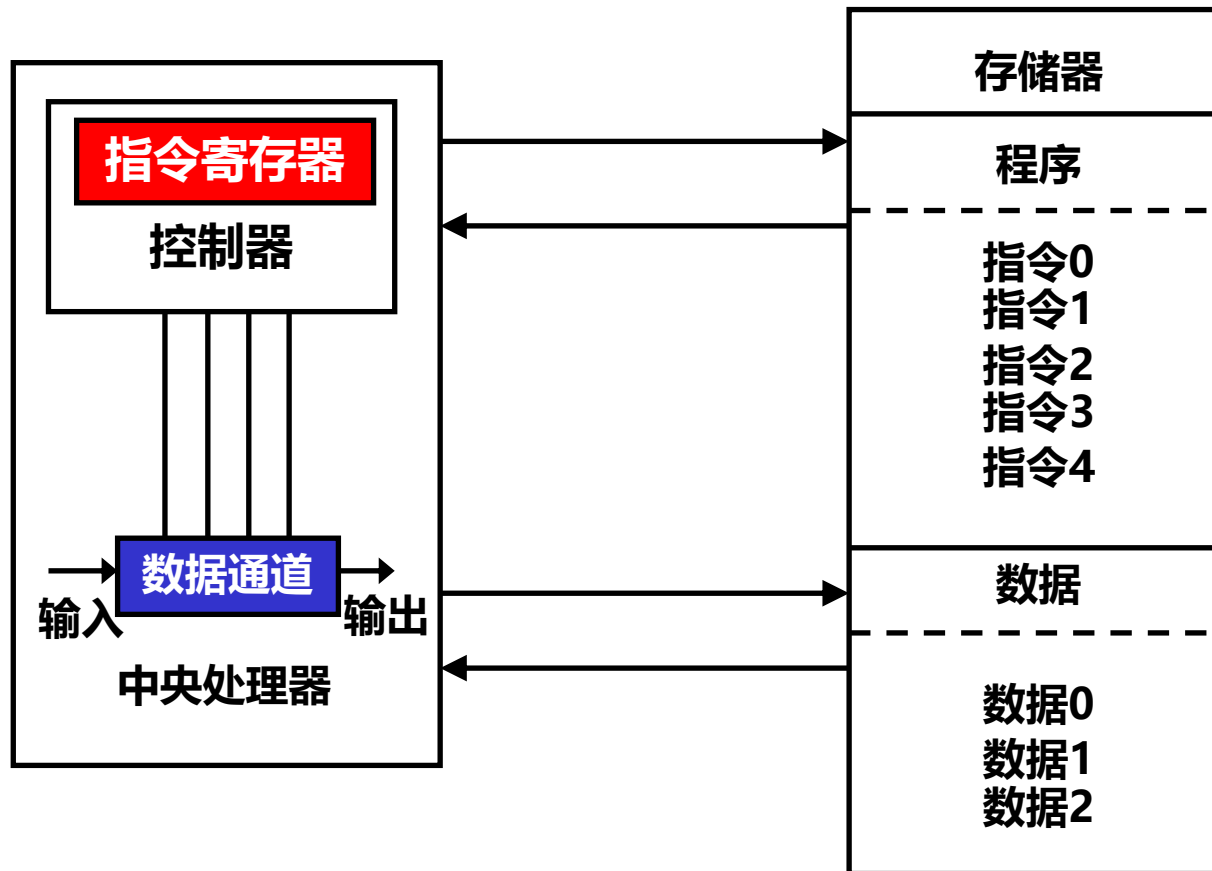
4

嵌入式系统设计方法

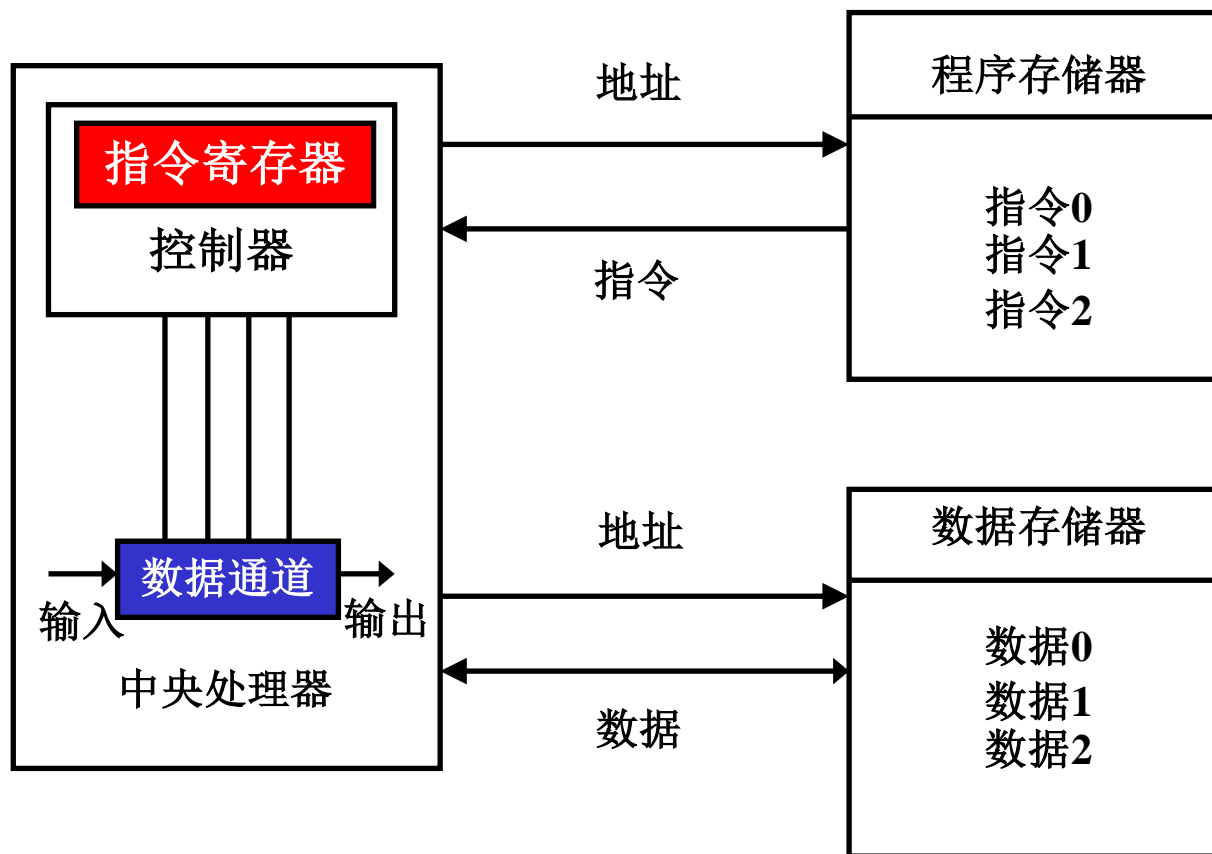
# 嵌入式系统硬件基础

- 冯·诺依曼体系结构和哈佛体系结构
- CISC与RISC
- IP 核
- 流水线
- 存储器系统

# 冯·诺依曼体系结构模型



# 哈佛体系结构



# CISC和RISC

**CISC: 复杂指令集 (Complex Instruction Set Computer)**

**具有大量的指令和寻址方式**

**8/2原则: 80%的程序只使用20%的指令**

**大多数程序只使用少量的指令就能够运行。**

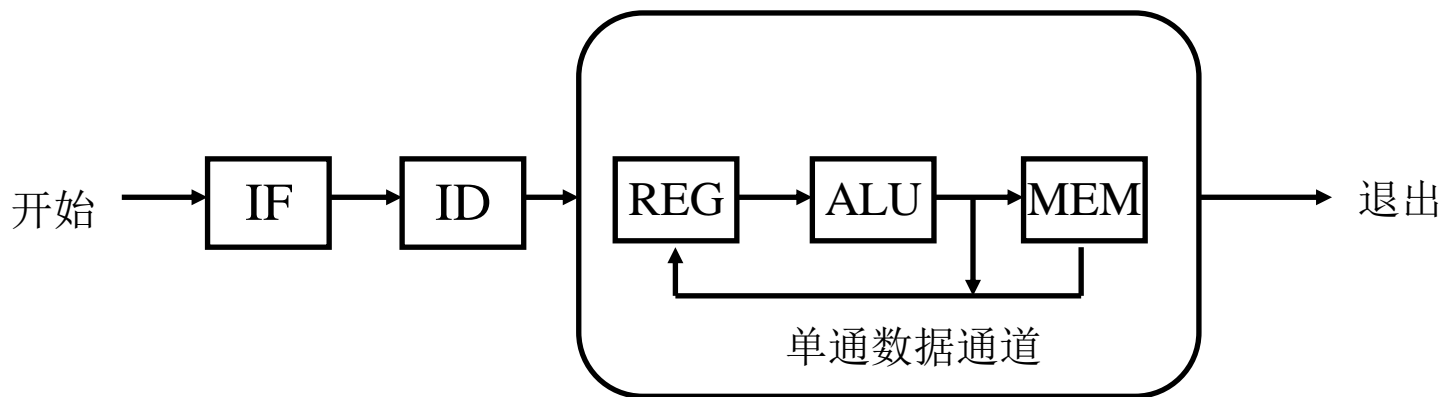
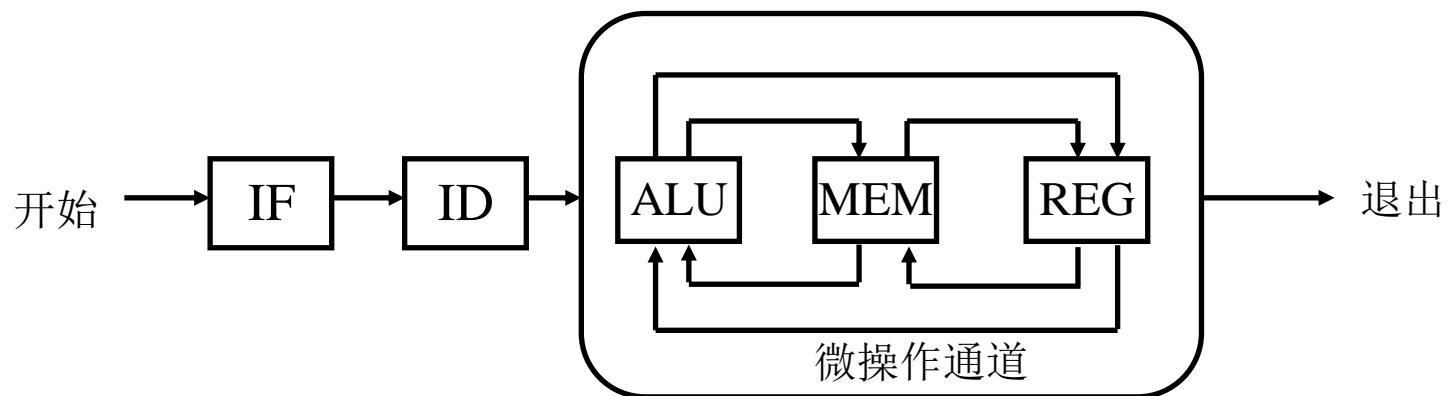
**RISC: 精简指令集 (Reduced Instruction Set Computer)**

**在通道中只包含最有用的指令**

**确保数据通道快速执行每一条指令**

**使CPU硬件结构设计变得更为简单**

# CISC与RISC的数据通道



# CISC的背景和特点

- **背景:存储资源紧缺, 强调编译优化**
- **增强指令功能, 设置一些功能复杂的指令, 把一些原来由软件实现的、常用的功能改用硬件的 (微程序) 指令系统来实现**
- **为节省存储空间, 强调高代码密度, 指令格式不固定, 指令可长可短, 操作数可多可少**
- **寻址方式复杂多样, 操作数可来自寄存器, 也可来自存储器**
- **采用微程序控制, 执行每条指令均需完成一个微指令序列 (微程序)**
- **$CPI > 5$ , 指令越复杂, CPI越大。**



# CISC的主要缺点

- 指令使用频度不均衡。
  - 高频度使用的指令占据了绝大部分的执行时间，扩充的复杂指令往往是低频度指令。
- 大量复杂指令的控制逻辑不规整，不适于VLSI工艺
  - VLSI的出现，使单芯片处理机希望采用规整的硬联逻辑实现，而不希望用微程序，因为微程序的使用反而制约了速度提高。(微码的存控速度比CPU慢5-10倍)。
- 软硬功能分配
  - 复杂指令增加硬件的复杂度，使指令执行周期大大加长，直接访存次数增多，降低了CPU性能。
- 不利于先进指令级并行技术的采用
  - 流水线技术

# RISC基本设计思想

- 减小CPI:  $\text{CPUtime} = \text{Instr\_Count} * \text{CPI} * \text{Clock\_cycle}$
- 精简指令集：保留最基本的,去掉复杂、使用频度不高的指令
- 采用Load/Store结构，有助于减少指令格式，统一存储器访问方式
- 采用硬接线控制代替微程序控制

# 典型的高性能RISC处理器

- **SUN公司的SPARC(1987)**
- **MIPS公司的SGI:MIPS(1986)**
- **HP公司的PA-RISC,**
- **IBM, Motorola公司的PowerPC**
- **DEC、Compac公司的Alpha AXP**
- **IBM的RS6000(1990)第一台Superscalar RISC机**

# CISC与RISC的对比

类别	CISC	RISC
指令系统	指令数量很多	较少，通常少于100
执行时间	有些指令执行时间很长，如整块的存储器内容拷贝；或将多个寄存器的内容拷贝到存贮器	没有较长执行时间的指令
编码长度	编码长度可变，1-15字节	编码长度固定，通常为4个字节
寻址方式	寻址方式多样	简单寻址
操作	可以对存储器和寄存器进行算术和逻辑操作	只能对寄存器对行算术和逻辑操作，Load/Store体系结构
编译	难以用优化编译器生成高效的目标代码程序	采用优化编译技术，生成高效的目标代码程序

# 小实验1

```
for (i = 0; i < 10000; ++i)  
    /* 各种算术运算操作 */
```

**实验平台：桌面Intel Pentium4，带硬件浮点支持**

Operator	Time	Operator	Time
+ (int)	1	+ (double)	5
* (int)	5	* (double)	5
/ (int)	12	/ (double)	10
<<(int)	2	sin	48

## 小实验2

**实验平台：400MHz Intel PXA250 Xscale(ARM)处理器**

<b>Operator</b>	<b>Time</b>	<b>Operator</b>	<b>Time</b>
<b>+ (int)</b>	<b>1</b>	<b>+ (double)</b>	<b>140</b>
<b>* (int)</b>	<b>1</b>	<b>* (double)</b>	<b>110</b>
<b>/ (int)</b>	<b>7</b>	<b>/ (double)</b>	<b>220</b>
<b>&lt;&lt;(int)</b>	<b>1</b>	<b>sin</b>	<b>3300</b>

# 知识产权核(IP核, intellectual property)

- **知识产权(IP) 电路或核是设计好并经过验证的集成电路功能单元**
- **IP复用意味着设计代价降低（时间，价格）**
- **IP核的类别：**
  - **微处理器微处理器: ARM, PowerPC;**
  - **存储器存储器: RAM, memory controller;**
  - **外设: PCI, DMA controller;**
  - **多媒体处理: MPEG/JPEG ;**
  - **encoder/decoder ;**
  - **数字信号处理器(DSP)**
  - **通信: Ethernet controller, router,**

# IP核的种类

- **Soft Cores(“code”)(软核)**
  - HDL语言描述
  - 灵活度高, 可修改
  - 与工艺独立, 可根据具体的加工工艺重新综合;
  - IP很难保护
- **Firm cores(“code+structure”)(固核)**
  - 逻辑综合后的描述
  - 与工艺相关
- **Hard cores(“physical”)(硬核)**
  - 物理综合后的描述
  - 准备流片
  - 包含工艺相关的布局和时序信息
  - IP很容易保护
  - 多数的处理器和存储器



# IP核的商业模型

## 三种模式

### 一、计者提供设计和工具的许可证

- DSP Group (Pine and Oak Cores), 3Soft, ARM
- 提供包括HDL在内的模拟模型, 工具或仿真器
- 使用者负责设计制造

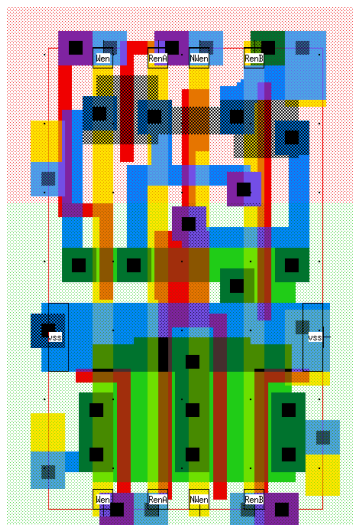
### 二、核厂商设计并制造集成电路芯片

- TI, Motorola, Lucent
- VLSI, SSI, Cirrus, Adaptec

### 三、核厂商卖核, 负责为客户设计并制造芯片

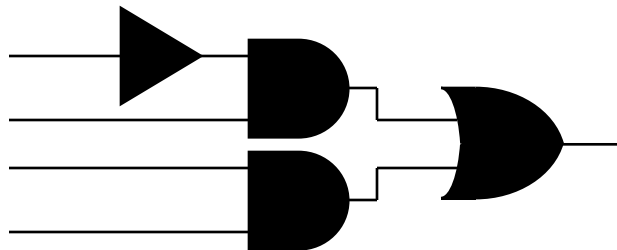
- LSI logic, TI, Lucent

# ARM的IP核



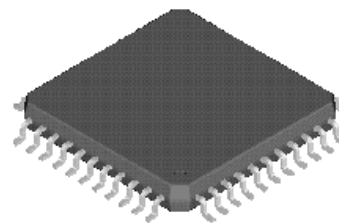
固化宏单元（硬核）

ARM920T  
ARM7TDMI  
ARM720T  
ARM1022E



可综合内核（软核）

ARM926EJ-S  
ARM7TDMI-S  
ARM1026EJ-S



测试芯片  
**ARM10200E**

# 流水线技术

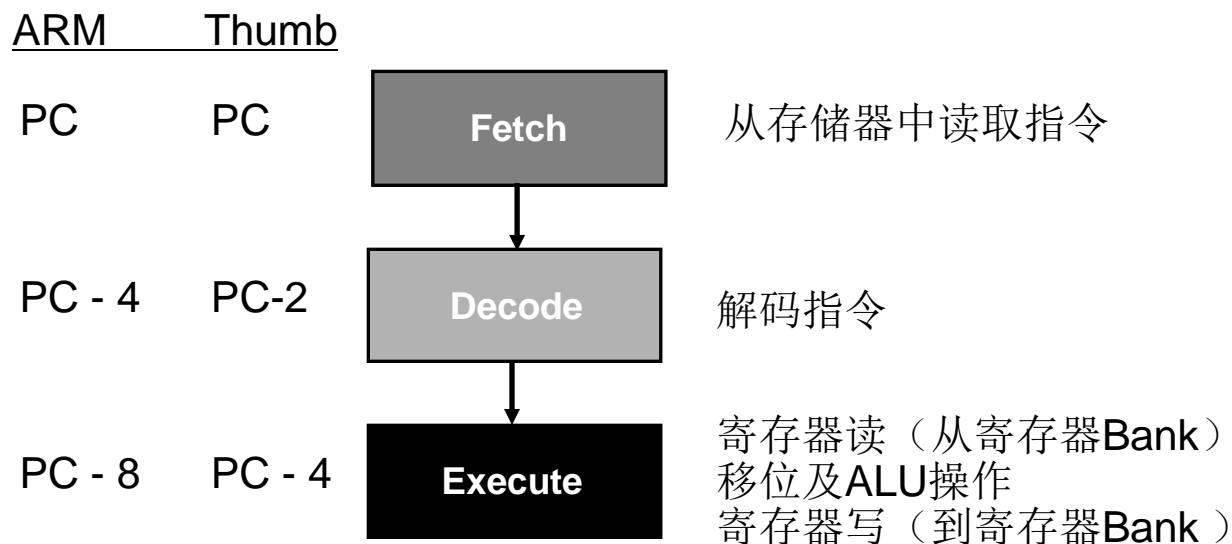
## 流水线技术：几个指令可以并行执行

- 提高了CPU的运行效率
- 内部信息流要求通畅流动



# 指令流水线—以ARM为例

- 为增加处理器指令流的速度，ARM7 系列使用3级流水线。
  - 允许多个操作同时处理，比逐条指令执行要快。



- PC指向正被取指的指令，而非正在执行的指令

# 最佳流水线

周期		1	2	3	4	5	6
操作							
ADD		Fetch	Decode	Execute			
SUB			Fetch	Decode	Execute		
MOV				Fetch	Decode	Execute	
AND					Fetch	Decode	Execute
ORR						Fetch	Decode
EOR							Fetch
CMP							
RSB							

- 该例中用6个时钟周期执行了6条指令
- 所有的操作都在寄存器中（单周期执行）
- 指令周期数 (CPI) = 1

# LDR 流水线举例

周期		1	2	3	4	5	6
操作							
ADD	Fetch	Decode	Execute				
SUB		Fetch	Decode	Execute			
LDR			Fetch	Decode	Execute	Data	Writeback
MOV				Fetch	Decode		Execute
AND					Fetch		Decode
ORR							Fetch

- 该例中，用6周期执行了4条指令
- 指令周期数 (CPI) = 1.5

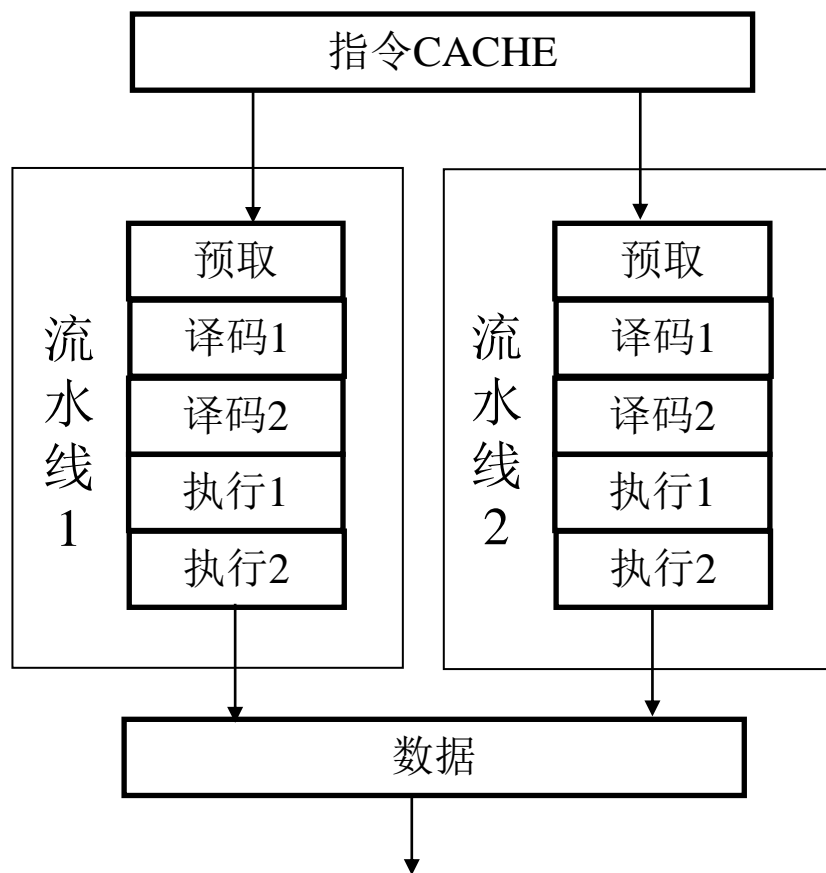
# 分支流水线举例

周期		1	2	3	4	5		
地址	操作							
0x8000	BL	Fetch	Decode	Execute	Linkret	Adjust		
0x8004	X		Fetch	Decode				
0x8008	XX			Fetch				
0x8FEC	ADD				Fetch	Decode	Execute	
0x8FF0	SUB					Fetch	Decode	Execute
0x8FF4	MOV						Fetch	Decode
								Fetch

- 流水线被阻断
- 注意:内核运行在ARM状态

# 超标量执行

超标量执行：超标量CPU采用多条流水线结构





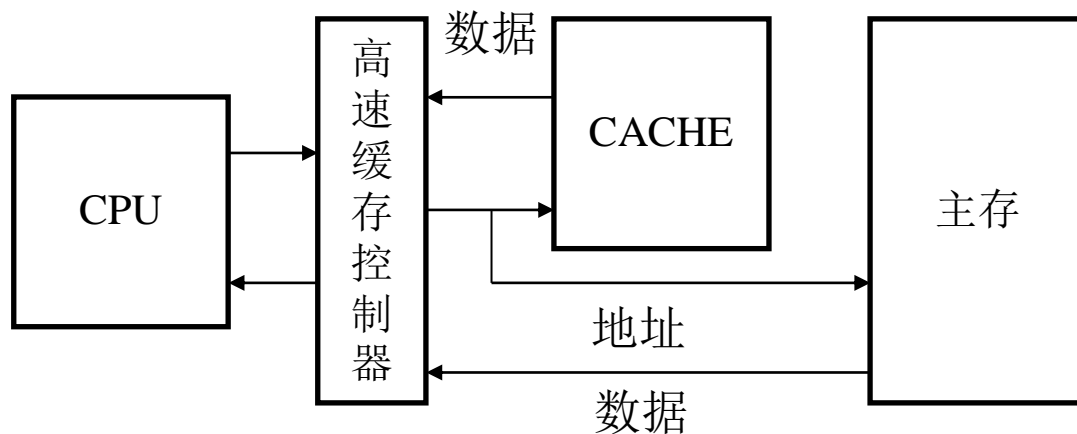
# 高速缓存（CACHE）

## 1、为什么采用高速缓存

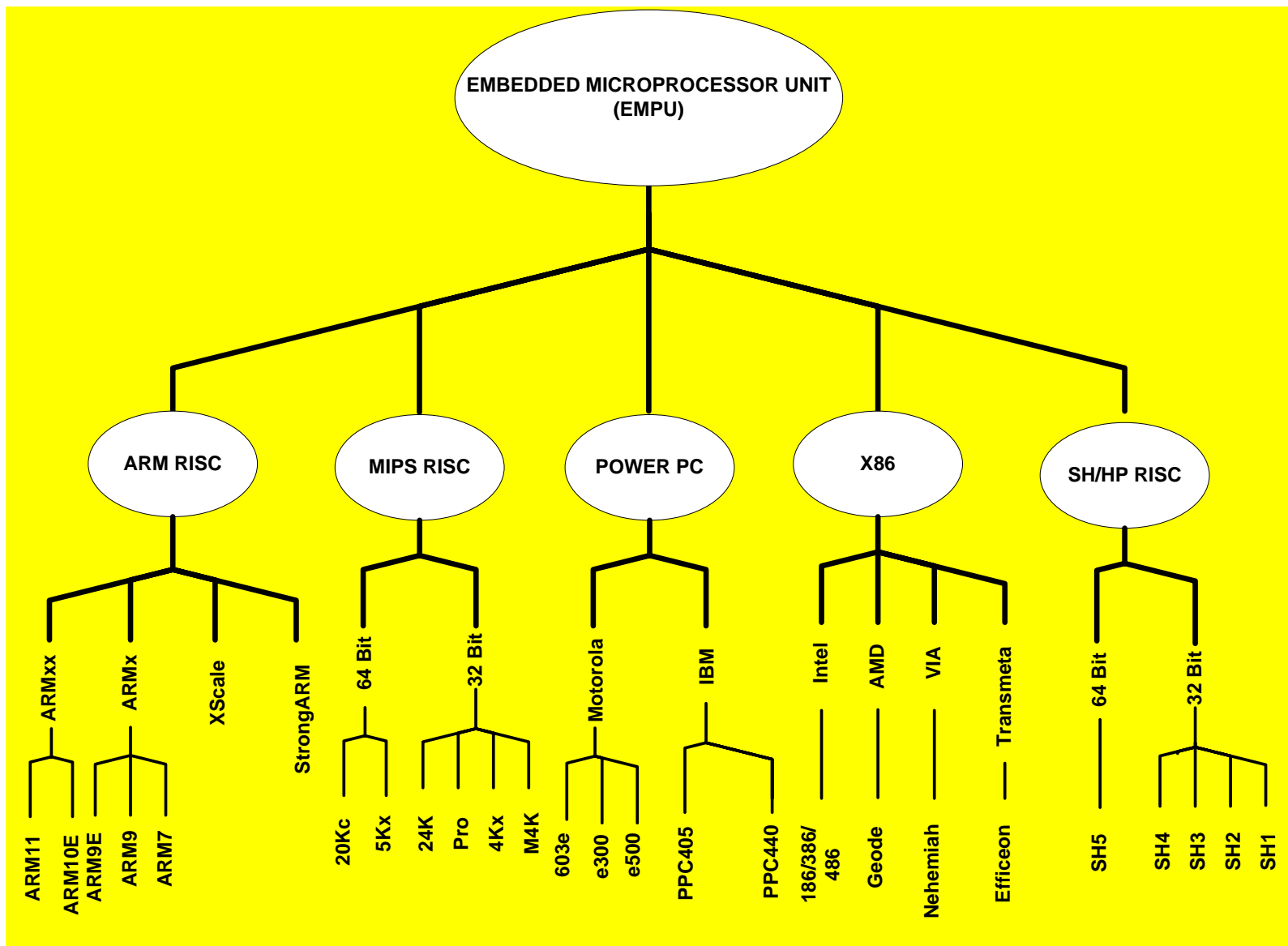
微处理器的时钟频率比内存速度提高快得多，高速缓存可以提高内存的平均性能。

## 2、高速缓存的工作原理

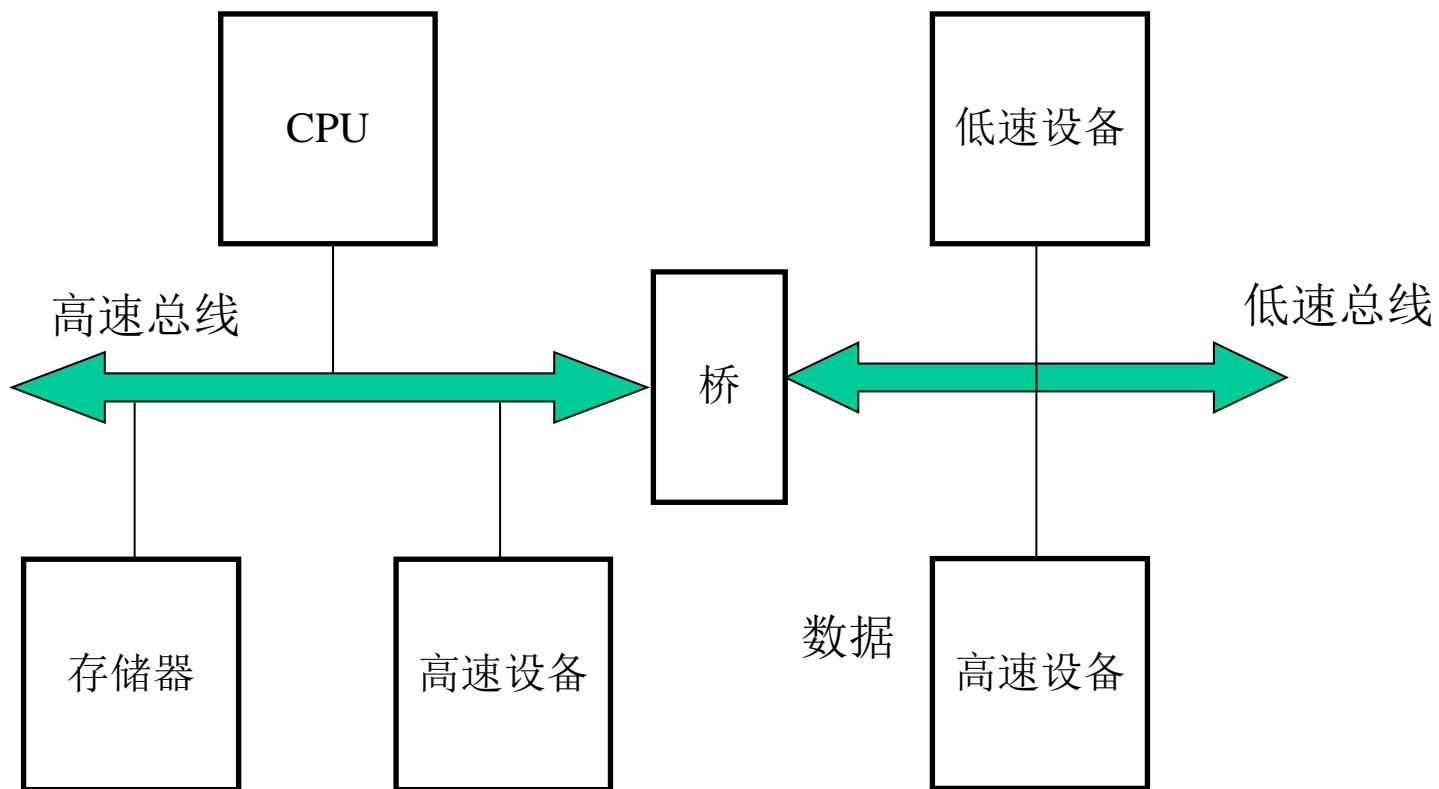
高速缓存是一种小型、快速的存储器，它保存部分主存内容的拷贝。



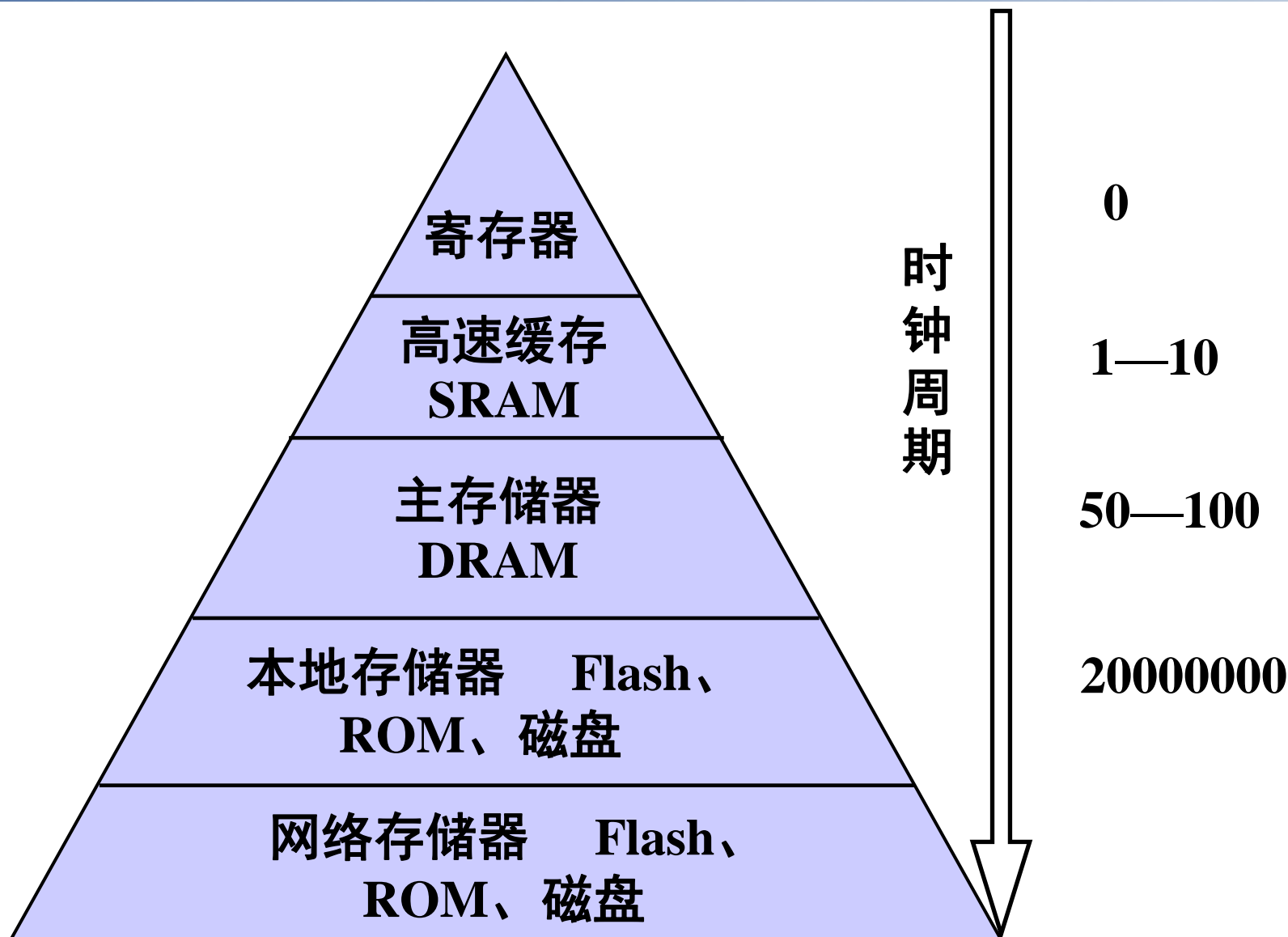
# 常见的嵌入式处理器结构



# 总线和总线桥



# 存储器系统的层次结构



# 存储器系统

**RAM：随机存取存储器， SRAM：静态随机存储器，  
DRAM：动态随机存储器**

- 1) SRAM比DRAM快
- 2) SRAM比DRAM耗电多
- 3) DRAM存储密度比SRAM高得多
- 4) DRAM需要周期性刷新

**ROM：只读存储器**

**FLASH：闪存**

# NOR技术

- **NOR技术闪速存储器是最早出现的Flash Memory，目前仍是多数供应商支持的技术架构，它源于传统的EPROM器件。**
- **与其它Flash Memory技术相比，具有可靠性高、随机读取速度快的优势，但擦除和写的速度较NAND慢。**
- **在擦除和编程操作较少而直接执行代码的场合，尤其是代码（指令）存储的应用中广泛使用。**
- **由于NOR技术Flash Memory的擦除和编程速度较慢，而块尺寸又较大，因此擦除和编程操作所花费的时间很长，在纯数据存储和文件存储的应用中，NOR技术显得力不从心。**

# NAND技术

- **NAND技术 Flash Memory具有以下特点：**
  - **以页为单位进行读和编程操作，1页为256或512字节；以块为单位进行擦除操作，1块为4K、8K或16K字节。具有快编程和快擦除的功能，其块擦除时间是2ms；而NOR技术的块擦除时间达到几百ms。**
  - **数据、地址采用同一总线，实现串行读取。随机读取速度慢且不能按字节随机编程。**
  - **芯片尺寸小，引脚少，是位成本(bit cost)最低的固态存储器，突破了每兆字节0.1元的价格限制。**
  - **芯片包含有失效块，其数目最大可达到3~35块（取决于存储器密度）。失效块不会影响有效块的性能，但设计者需要将失效块在地址映射表中屏蔽起来。**
- **基于NAND的存储器可以取代硬盘或其它块设备。**

# 输入输出接口

- I/O
- A/D、D/A
- 键盘
- LCD
- 存储器接口
- 设备接口



# 本节提要

1

嵌入式系统硬件基础

2

嵌入式系统软件基础

3

嵌入式操作系统

4

嵌入式系统设计方法

# 嵌入式软件体系结构

- **无操作系统的情形**

- ✓ 在嵌入式系统的发展初期，由于硬件的配置比较低，对于是否有系统软件的支持，要求还不是很强烈。在那个阶段，嵌入式软件的设计主要是以应用为核心，应用软件直接建立在硬件上，没有专门的操作系统。

- **有操作系统的情形**

# 无操作系统的情形

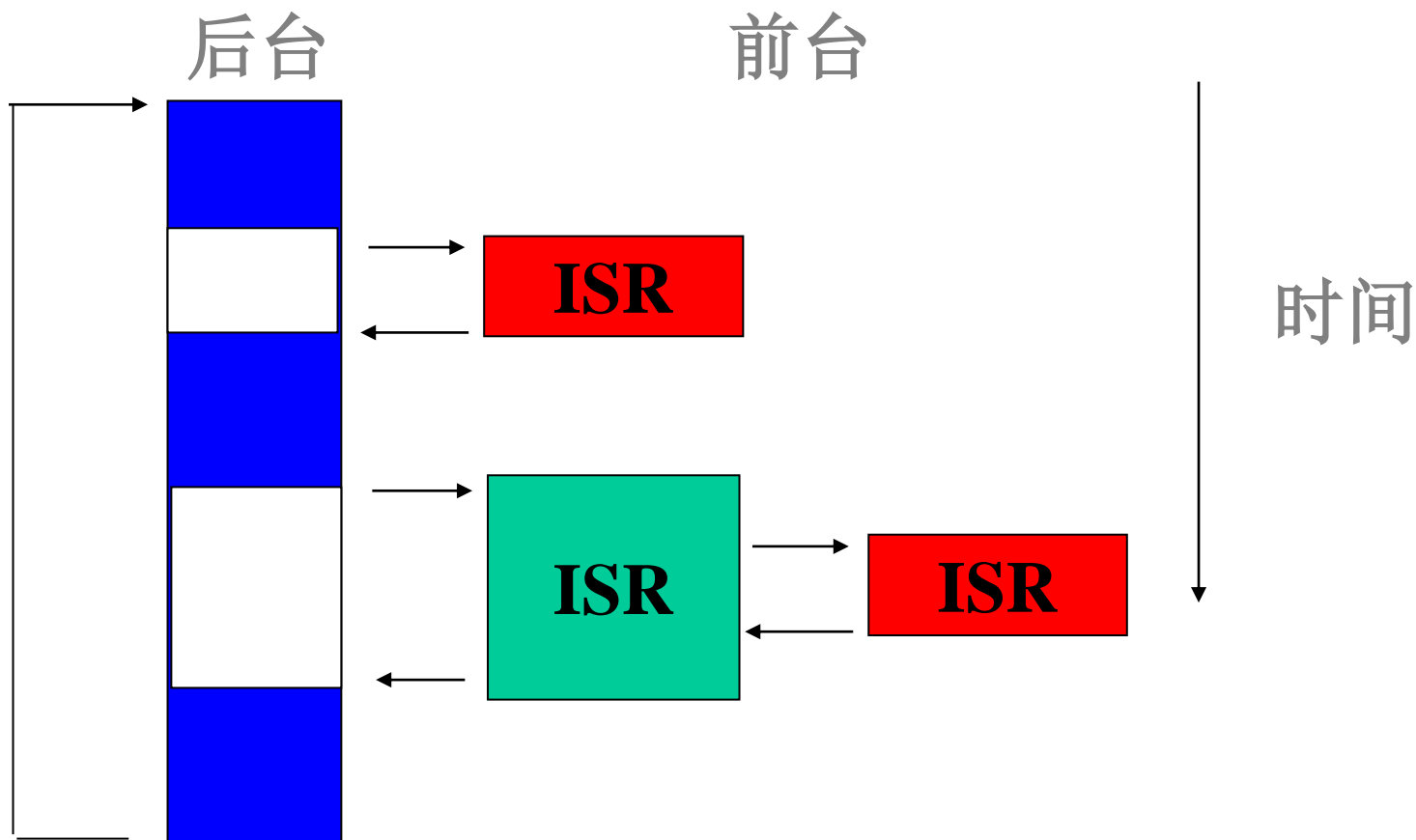
- 循环轮询系统：（**Polling Loop**）
  - 最简单的软件结构，程序依次检查系统的每个输入条件，一旦条件成立就进行相应的处理。

```
Initialize();  
while(1){  
    if(condition_1) action_1();  
    if(condition_2) action_2();  
    .....  
    if(condition_n) action_n();  
}
```

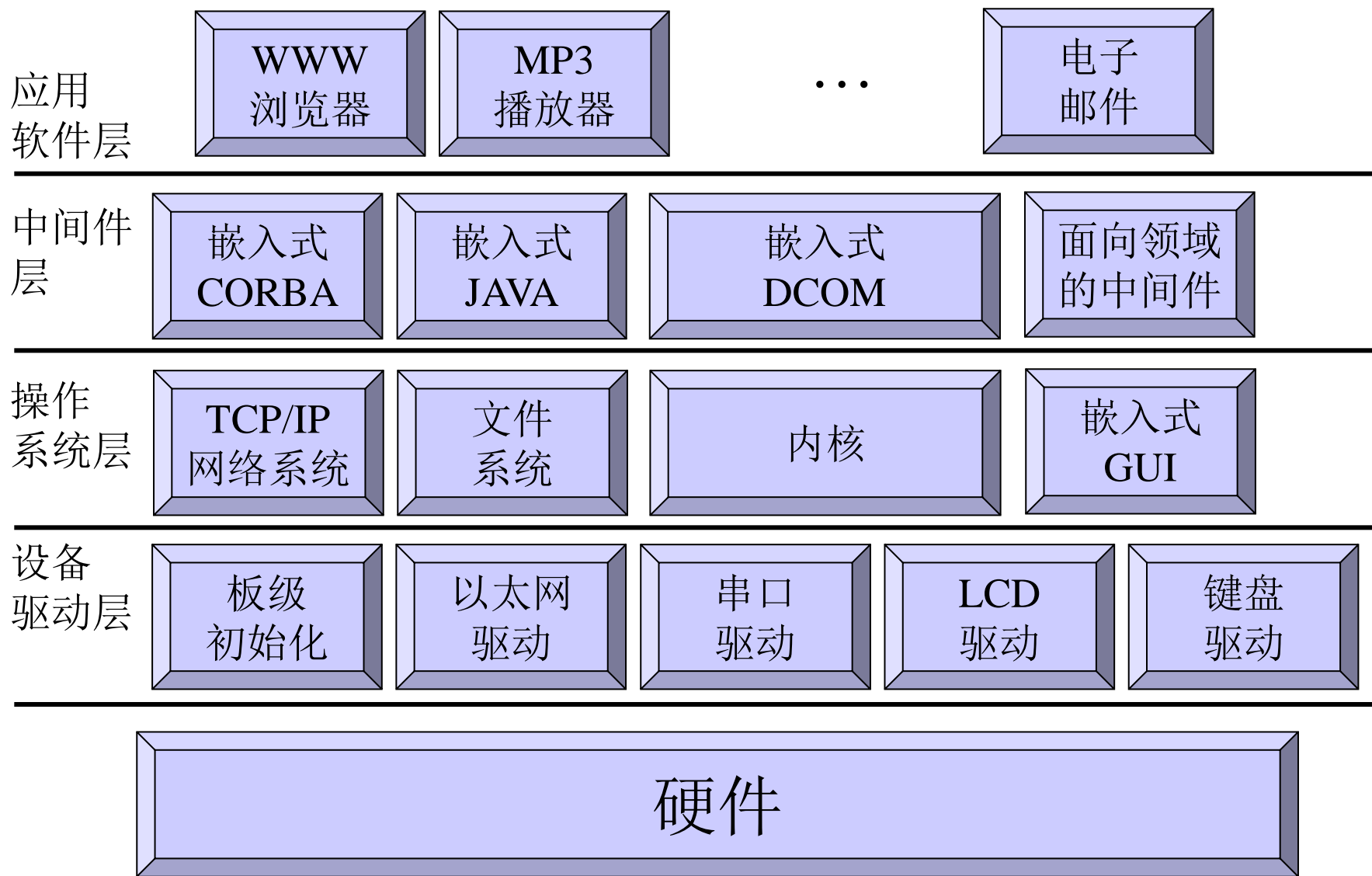
- 事件驱动系统：（**Event-Driven system**）
  - 事件驱动系统是能对外部事件直接响应的系统。它包括前台、实时多任务、多处理器等，是嵌入式实时系统的主要形式。
  - 应用程序是一个无限循环，循环中调用相应的函数完成相应操作，这部分可以看成后台行为（**background**）。中断服务程序处理异步事件，这部分可看成前台行为（**foreground**）。
  - 后台也可以叫做任务级，前台也叫中断级。

- 例如，很多基于微处理器的产品采用前后台系统设计，如微波炉、电话机、玩具等。从省电的角度出发，平时微处理器处在停机状态，所有的事都靠中断服务来完成。

## 前后台系统（后台循环、前台中断）



# 有操作系统的情形



# 设备驱动程序

- 为什么要有设备驱动程序？

嵌入式硬件设备本身无法工作，需要软件来驱动，如初始化、控制、数据读写等。

- 什么是设备驱动程序？

直接与硬件打交道、对硬件进行控制和管理  
的软件。

- 在一个嵌入式系统中，设备驱动程序是**必不可少**的。



# 设备驱动程序的主要功能

- **硬件启动 (Startup) : 在开机上电或重启的时候, 对硬件进行初始化;**
- **硬件关闭 (Shutdown) : 把硬件配置成关机状态;**
- **硬件停用 (Disable) : 暂停使用硬件;**
- **硬件启用 (Enable) : 重新启用硬件;**
- **硬件读操作 (Read) : 从硬件中读取数据;**
- **硬件写操作 (Write) : 往硬件中写数据;**
- **.....**

# 嵌入式操作系统

- **嵌入式操作系统包括嵌入式内核、嵌入式TCP/IP网络系统、嵌入式文件系统、嵌入式GUI系统和电源管理等部分；**
- **嵌入式内核是基础和核心，其他部分要根据嵌入式系统的需要来确定。**

# 嵌入式中间件

- **中间件 (Middleware) : 在OS内核、设备驱动程序和应用软件之外的所有系统软件;**
- **中间件的基本思路: 把原本属于应用软件层的一些通用的功能模块抽取出来, 形成独立的一层软件, 从而为运行在其上的各个应用软件提供一个灵活、安全、移植性好、相互通信、协同工作的平台;**
- **优点: 实现软件的可重用, 降低应用软件的复杂性, 降低开发成本。**

# 嵌入式C程序设计

**“which of the following programming languages have you used for embedded systems in the last 12 months”**

<b>C</b>	<b>81%</b>
<b>Assembly</b>	<b>70%</b>
<b>C++</b>	<b>39%</b>
<b>Visual Basic</b>	<b>16%</b>
<b>Java</b>	<b>7%</b>

**Source: “ESP: A 10-year retrospective”, Embedded Systems Programming, November, 1998**

# 嵌入式软件的目标

- 函数必须正确;
- 源代码简洁、可读性好、可维护;
- 实时性要求较高的代码能够运行得足够快;
- 目标代码小且高效。
- 总之，要优化对以下三种资源的使用：
  - 执行时间;
  - 存储空间;
  - 开发/维护时间。

# 数据类型与运算符

## (1) 宏定义

宏定义：用一个指定的标识符来代表一个字符串。

**#define**    标识符    字符串

如：**#define PI 3.1415926**，其作用是指定用标识符**PI**来代替“3.1415926”这个字符串，在编译预处理时，将程序中出现的所有**PI**都用“3.1415926”代替。

宏定义的基本思想是：一次定义，多次使用。其优点是：

- 可以用简短的标识符来代替长的数据，减少需要输入的字符数；
- 用易于理解的标识符来代替那些不太好记的具体数据，便于程序的理解和维护；
- 有利于程序的修改和升级，当这个数据需要修改时，只需改动宏定义之处即可。

不用  
此法

```
if ( myMoney > 80.0 )  
{  
    myShoes ++;  
    myMoney = myMoney - 80.0;  
}
```

一次  
定义

```
#define COST_OF_SHOES 80.0  
if(myMoney > COST_OF_SHOES)  
{
```

多次  
使用

```
    myShoes ++;  
    myMoney = myMoney - COST_OF_SHOES;  
}
```



## (2) **const**常量

- 常量数据：整数（12）、字符（‘a’）、字符串（“hello”）和实数（3.14）等；
- 以变量的形式来定义的一个量，并且通过使用关键字**const**，来表明这个变量的值不能被改变。如：**const int x = 1**。

### (3) 算术运算

整数的算术运算

带有硬件支持的浮点运算

用软件来实现的浮点运算

最快

较慢

非常慢

$+$  ,  $-$

$\times$

$\div$

$\sin$ ,  $\log$ ,  $\sqrt{\phantom{x}}$ , etc

快

慢

结论:

- **尽量使用整数 (char、short、int和long) 的加法和减法;**
- **如果没有硬件支持, 尽量避免使用乘法;**
- **尽量避免使用除法;**
- **如果没有硬件支持, 尽量避免使用浮点数;**
- **数学库函数使用得越少越好。**

## (4) 位运算

**C语言有很多位操作运算符：**

<b>&amp;</b>	<b>与操作；</b>
<b> </b>	<b>或操作；</b>
<b>^</b>	<b>异或操作；</b>
<b>~</b>	<b>取反操作；</b>
<b>&gt;&gt;</b>	<b>右移操作；</b>
<b>&lt;&lt;</b>	<b>左移操作。</b>

**a |= 0x4**

**// 把第2位设置为1**

**b &= ~0x4**

**// 把第2位设置为0**

**c &= ~(1 << 3)**

**// 把第3位设置为0**

**d ^= (1 << 5)**

**// 把第5位反转**

**e >>= 2**

**// 把 e 除以4**

# 分支语句

```
if (a == 1)
    ant();
else if (a == 2)
    bar();
else if (a == 3)
    cee();
else if (a == 4)
    due();
else if (a == 5)
    eat();
else if (a == 6)
    foo();
```

```
switch (a)
{
    case 1: ant(); break;
    case 2: bar(); break;
    case 3: cee(); break;
    case 4: due(); break;
    case 5: eat(); break;
    case 6: foo(); break;
}
```

*Any Differences?*

结论:

- 假设a的取值个数为n, 对于if-then-else语句, 时间复杂度为 $O(n)$ , 而对于switch语句, 时间复杂度为 $O(1)$ ;
- 如果n的值较小, 两种语句均可;
- 如果n的值较大, 则switch语句更佳。

# 函数

函数原型

→ 声明该函数

```
main ( )
```

```
{
```

```
.....
```

函数调用

→ 使用该函数

```
.....
```

```
}
```

函数定义

→ 定义一个函数

函数的使用模式

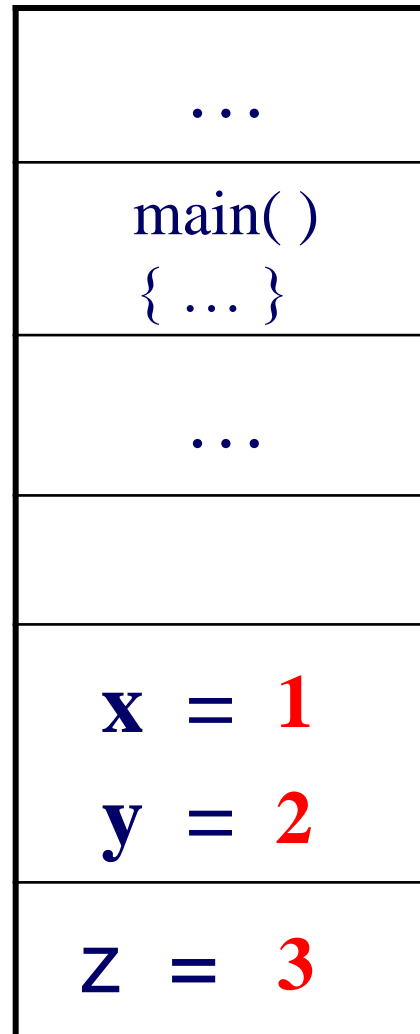


# 内存分布状况



# 主函数的执行过程

```
int z;  
void main( )  
{  
    int x, y;  
  
    x = 1;  
    y = 2;  
    z = x + y;  
}
```



程序

栈帧(main)

全局变量区域

# 控制流与数据流

**控制流：** 程序当前执行位置的流向；

**数据流：** 函数调用发生及结束时，数据在函数之间流转的过程。

# 函数调用过程

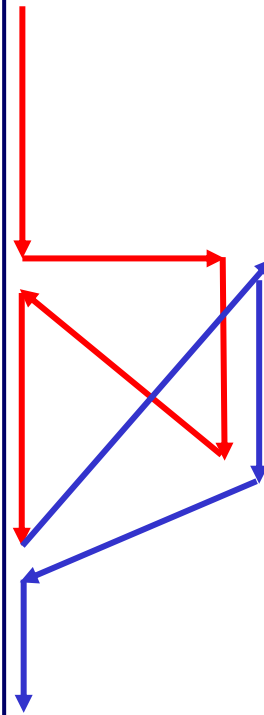
## 当一个函数被调用时：

1. 在内存的栈空间当中为其分配一个栈帧，用来存放该函数的形参和局部变量；
2. 把实参变量的值复制到相应的形参变量；
3. 控制转移到该函数的起始位置；
4. 该函数开始执行；
5. 控制流和返回值返回到函数调用点。

# 控制流的变化

```
void main( )  
{  
    double  x, y, z;  
  
    y = 6.0;  
    x = Area( y / 3.0 );  
    ...  
  
    z = 3.4 * Area(7.88);  
    ...  
}
```

```
/* 给定半径，计算一个圆的面积 */  
double Area(double r)  
{  
    return(3.14 * r * r);  
}
```



# 一个简单的例子

```
int Times2(int value);
main ( )
{
    int number;
    printf("请输入一个整数: " );
    scanf("%d", &number);
    printf("该数的两倍是: %d", Times2(number));
}

int Times2(int value)
{
    return(2 * value);
}
```

**main**

**number**

**3**

```

int Times2(int value);
main ( )
{
    int number;
    printf("请输入一个整数: " );
    scanf("%d", &number);
    printf("该数的两倍是: %d", Times2(number));
}

int Times2(int value)
{
    return(2 * value);
}

```

**main**

**number**

**3**

**Times2**

**value**

**Times2**也得到一个栈帧，  
它的参数看成局部变量

```
int Times2(int value);
main ( )
{
    int number;
    printf("请输入一个整数: " );
    scanf("%d", &number);
    printf("该数的两倍是: %d", Times2(number));
}
→ int Times2(int value)
{
    return(2 * value);
}
```

**main**

**number**

3

**Times2**

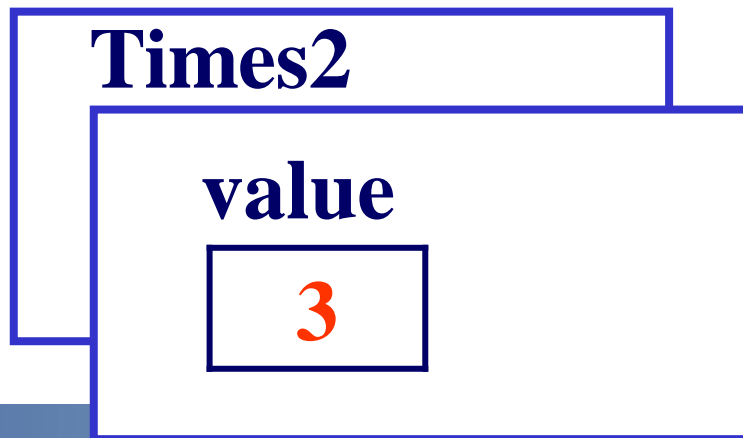
**value**

3

“值传递”，把实参的值  
传给形参。



```
int Times2(int value);
main ( )
{
    int number;
    printf("请输入一个整数: " );
    scanf("%d", &number);
    printf("该数的两倍是: %d", Times2(number));
}
→ int Times2(int value)
{
    return(2 * value);
}
```



把**Times2**的栈帧叠在主函数的栈帧之上，说明在执行**Times2**函数时，主函数中的变量是不可见的。

```
int Times2(int value);
main ( )
{
    int number;
    printf("请输入一个整数: " );
    scanf("%d", &number);
    printf("该数的两倍是: %d", Times2(number));
}

int Times2(int value)
{
    return(2 * value);
}
```

6

**main**

**number**

3

**Times2函数的返回值被放在函数的调用位置上，然后，分配给Times2函数的堆栈区域被释放。**

# 变量的存储与作用域

```
/* 全局变量，固定地址，其他源文件可见 */
int global_static;
/* 静态全局变量，固定地址，但只在本文件中可见 */
static int file_static;
/* 函数参数：位于栈帧当中，动态创建，动态释放 */
int foo(int auto_param)
{
    /*静态局部变量，固定地址，只在本函数中可见 */
    static int func_static;
    /* 普通局部变量，位于栈帧当中，只在本函数可见 */
    int auto_i, auto_a[10];
    /* 动态申请的内存空间，位于堆当中 */
    double *auto_d = malloc(sizeof(double)*5);
    return auto_i;
}
```

# 可重入函数-1

可以被一个以上的任务调用，而不必担心数据的破坏。  
可重入型函数任何时候都可以被中断，一段时间以后又可以运行，而相应数据不会丢失。可重入型函数只使用局部变量，即变量保存在CPU寄存器或栈中。

一个不可重入型函数的例子

```
int temp;  
void swap(int *x, int *y)  
{  
    temp = *x;  
    *x = *y;  
    *y = temp;  
}
```

## 可重入函数-2

### 一个可重入型函数的例子

```
void swap(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

# 不可重入函数被中断破坏

## LOW PRIORITY TASK

```
while (1) {  
    x = 1;  
    y = 2;  
  
    swap(&x, &y);  
    {  
        Temp = *x;  
        *x = *y;  
        *y = Temp;  
    }  
    OSTimeDly(1);  
}
```

Temp == 1

(1)

Temp == 3!

## HIGH PRIORITY TASK

```
while (1) {  
    z = 3;  
    t = 4;  
  
    swap(&z, &t);  
    {  
        Temp = *z;  
        *z = *t;  
        *t = Temp;  
    }  
    OSTimeDly(1);  
}
```

(3)

(4)

Temp == 3

OSIntExit()

ISR

O.S.

O.S.

Temp == 3

# 本节提要

1

嵌入式系统硬件基础

2

嵌入式系统软件基础

3

嵌入式操作系统

4

嵌入式系统设计方法

# 嵌入式操作系统概述

**An Embedded Operating System (EOS)**

**is**

**an Operating System (OS) in an Embedded System environment.**



# Being an OS means...

## ■ 系统软硬件资源的管理者：

- ☺ 进程管理
- ☺ 存储管理
- ☺ I/O设备管理
- ☺ 文件管理

## Being an EOS means...

- 完成某一项或有限项功能，非通用型；
- 在性能和实时性方面可能有严格限制；
- 能源、成本和可靠性通常是影响设计的重要因素；
- 占有资源少，适合在有限存储空间运行；
- 系统功能可针对需求进行裁剪、调整，以便满足最终产品的设计要求。

# 按响应时间分类...

## ■ 嵌入式实时操作系统

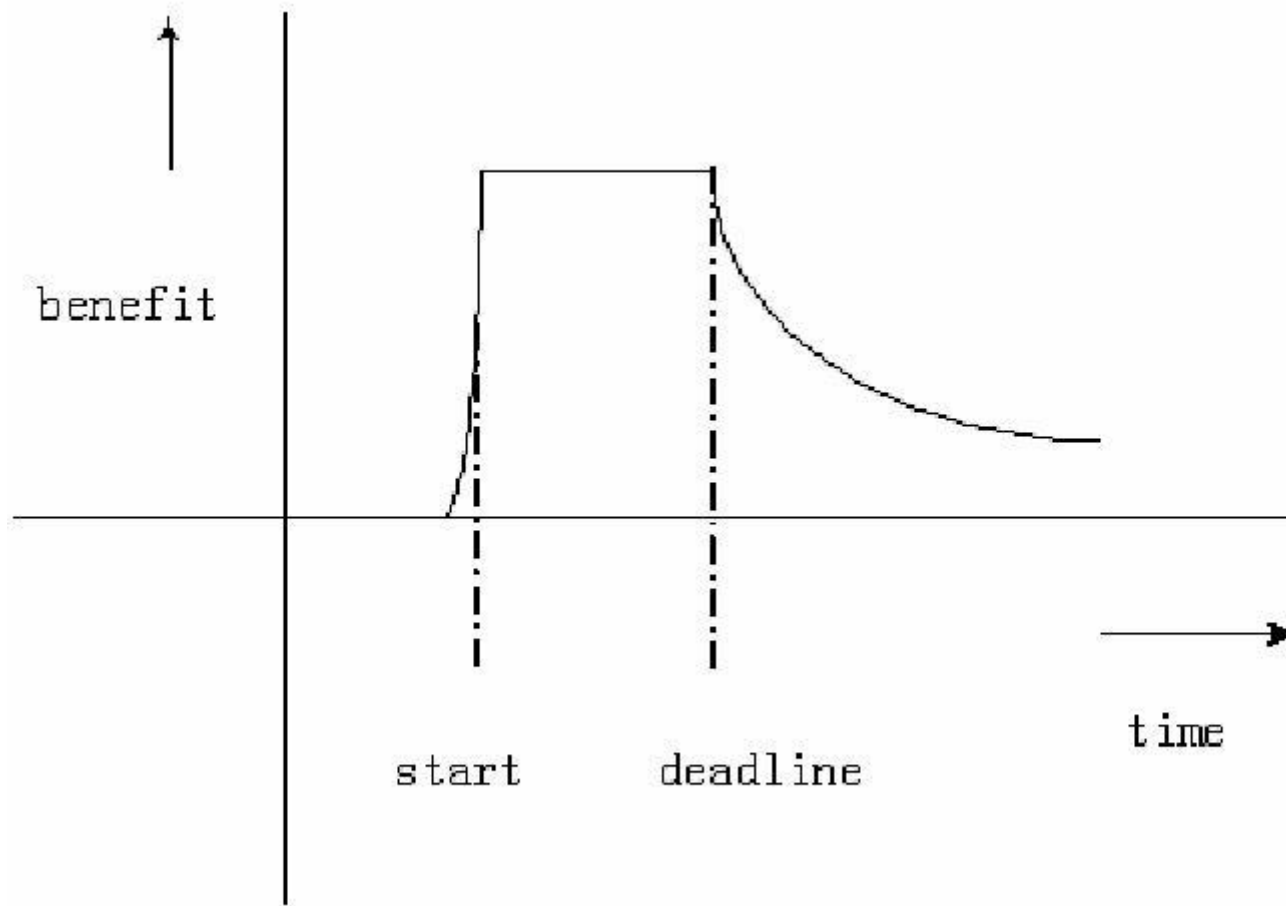
**当事件/请求发生时，相应的任务应该在规定的时间内完成；**

## ■ 分时操作系统

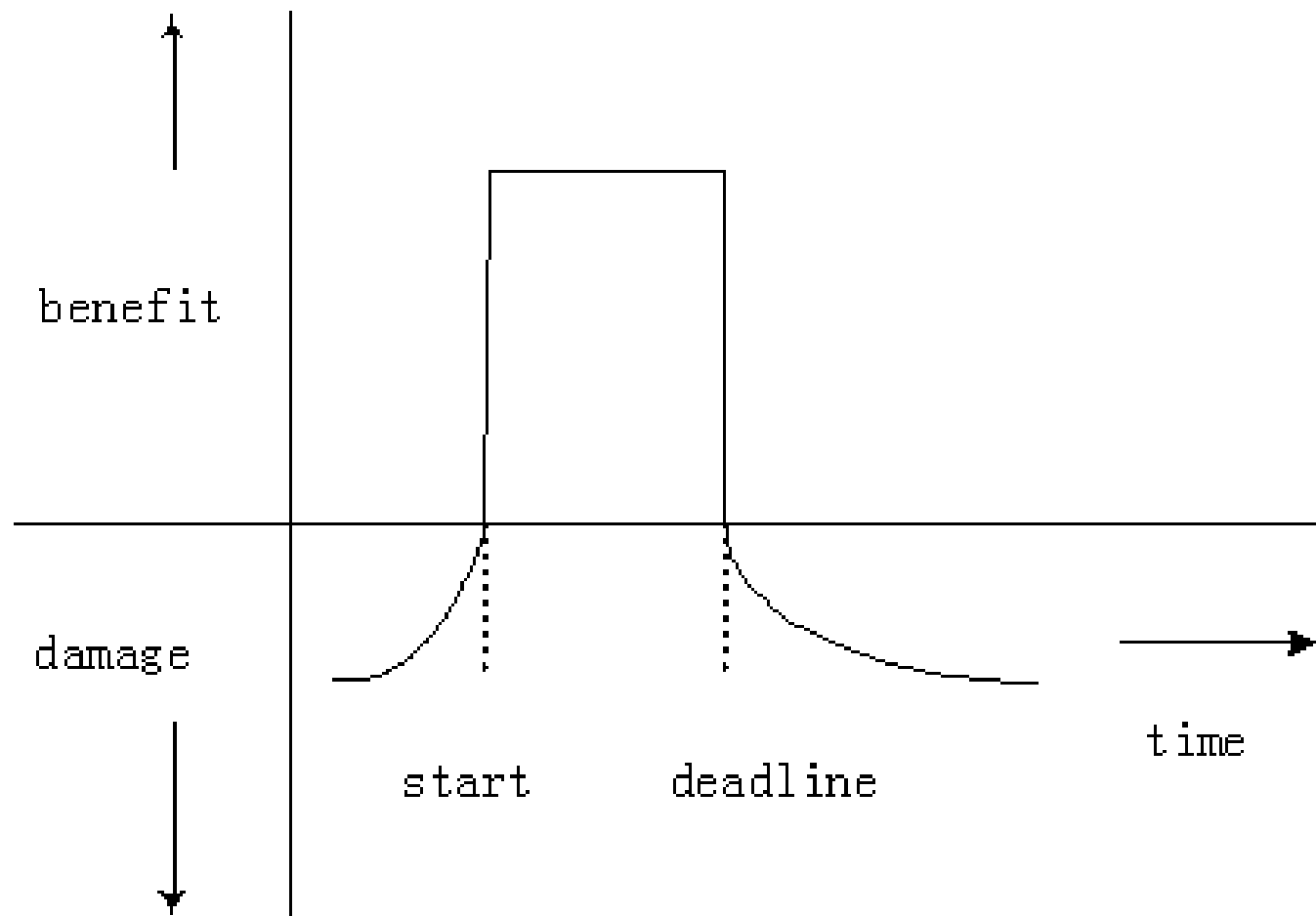
**基于公平性原则，各个进程分享处理器，获得大致相同的运行时间。当一个进程在进行I/O操作时，交出处理器，让其他进程运行。**

## ■ 实时操作系统(RTOS)

- A real-time operating system (RTOS) is an **operating system** whose correctness includes its **response time** as well as its functional correctness.
- **hard real time** and **soft real time**



**soft real time**



**hard real time**

# 按软件结构分类...

- 单体结构 (**Monolithic Structure**)

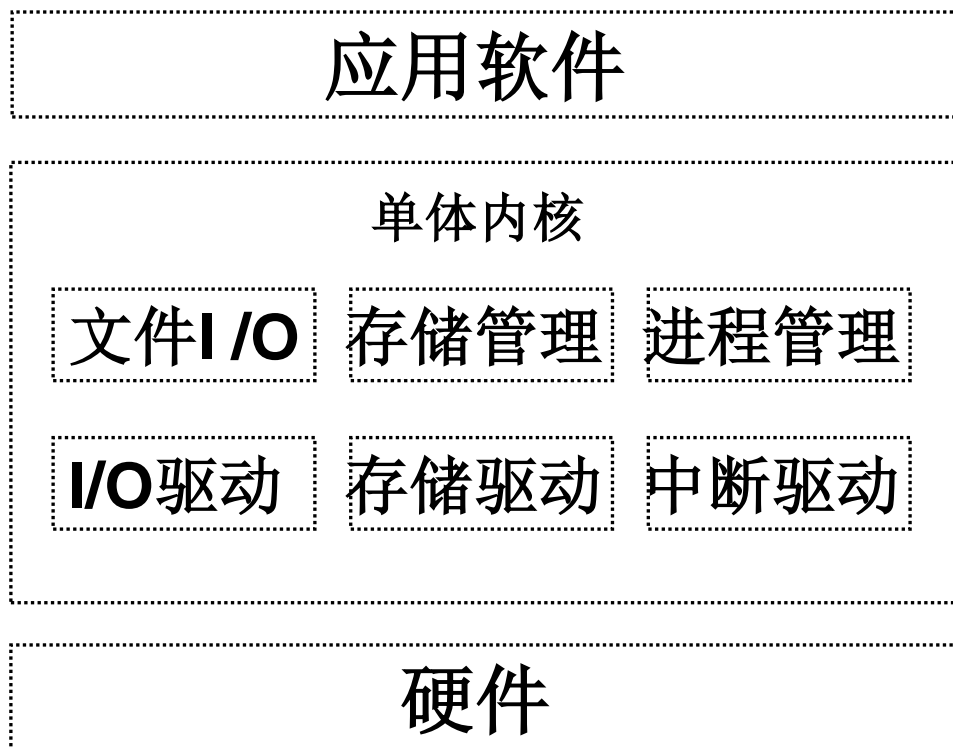
- 分层结构 (**Layered Structure**)

**Out of date...**

- 微内核结构 (**Microkernel Model**)

# 单体结构

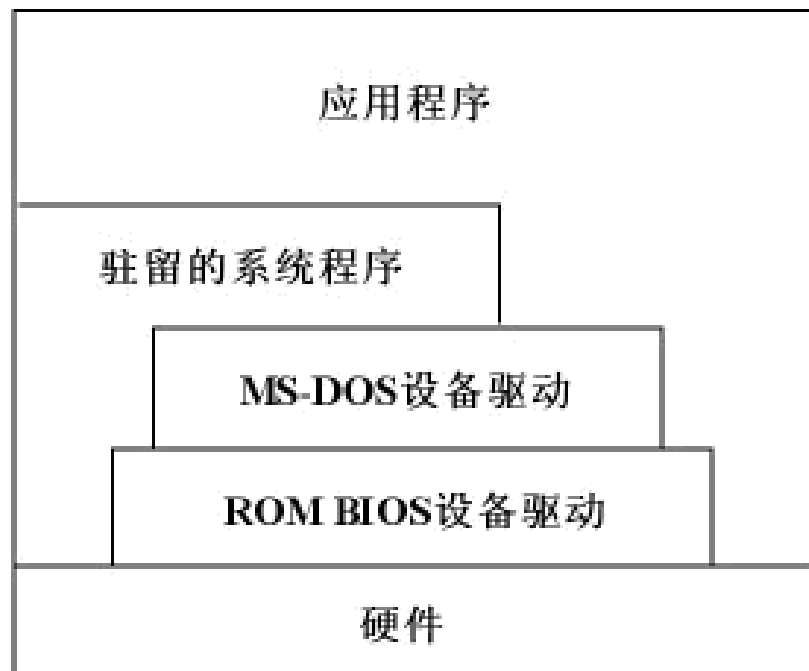
- 最常用的组织结构；
- 整个系统只有一个可执行文件，包含所有的操作系统组件；
- 系统的结构就是无结构，由一组函数组成，相互之间可以随意地调用。





# 分层结构

- 在分层结构（**layered**）中，一个操作系统被划分为若干个层次（**0..N**），各个层次之间的调用关系是单向的，即某一层上的代码只能调用比它低层的代码。
- 这种结构要求在每个层次上都要提供一组**API**接口函数，这就会带来额外的开销



# 微内核结构

- 操作系统内核只包含最少的功能，如存储管理和进程管理；
- 其他的操作系统组件以中间件的形式存在于内核之外；
- 设备驱动程序完全从内核中剥离，独立成为一层。



# 常见的嵌入式操作系统

❖ **VxWorks**



❖ **Embedded Linux**



❖ **uC/OS-II** (重点)

❖ **WinCE**



❖ **PalmOS**

❖ ...



# 多道程序技术

为了提高计算机系统中各种资源的利用率，现代操作系统广泛采用多道程序技术（**multi-programming**），使多个程序同时在系统中存在并运行。

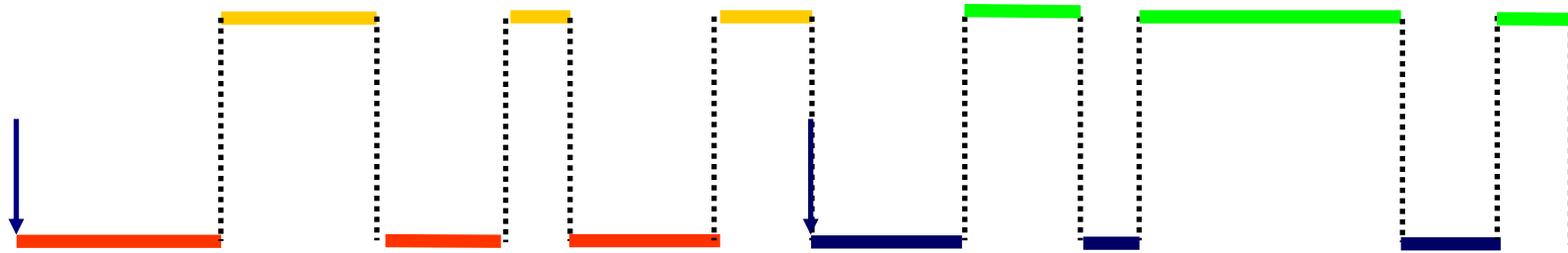
单道程序:

作业甲 (红黄)

作业乙 (蓝绿)

CPU

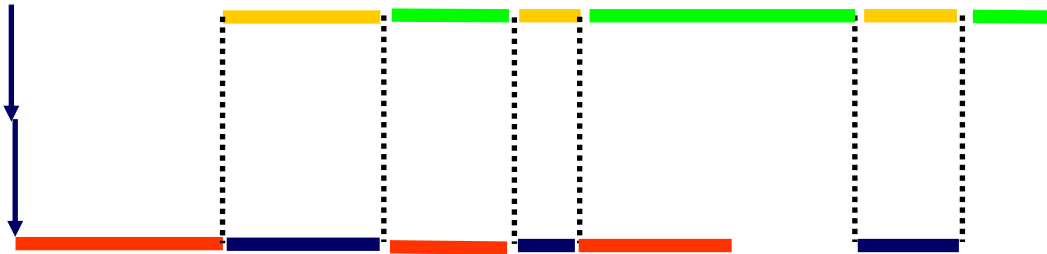
I/O

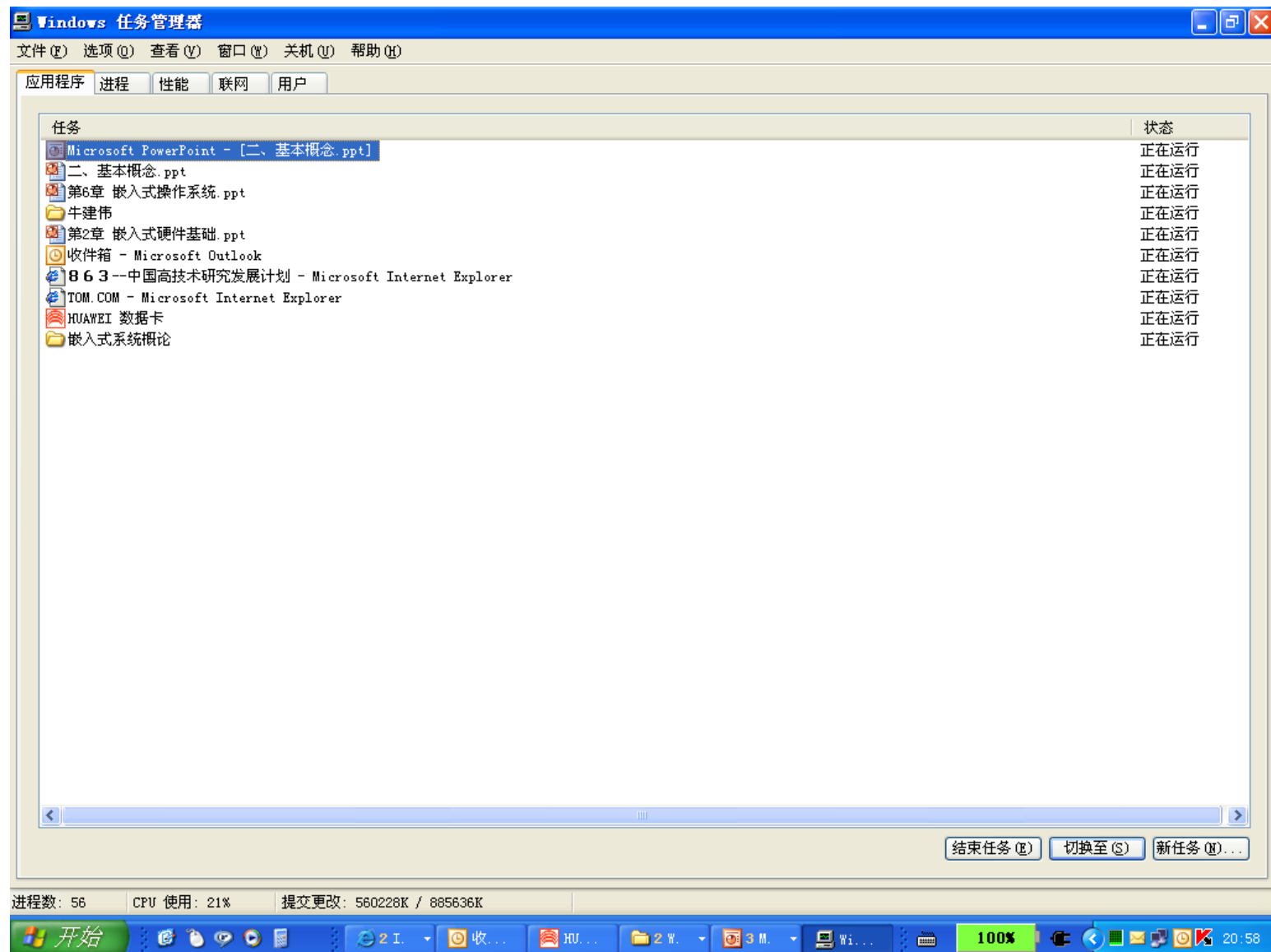


多道程序:

CPU

I/O





# 进程、线程和任务

在多道程序系统中，各个程序之间是并发执行的，共享系统资源。CPU需要在各个运行的程序之间来回地切换，这样的话，要想描述这些多道的并发活动过程就变得很困难。为此，操作系统设计者提出了**进程**的概念。

# 什么是进程？

**A process = a program in execution**

一个进程应该包括：

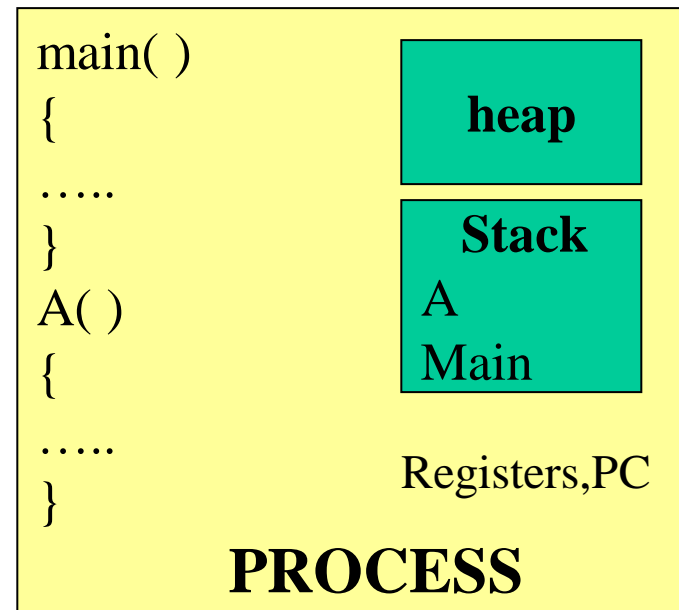
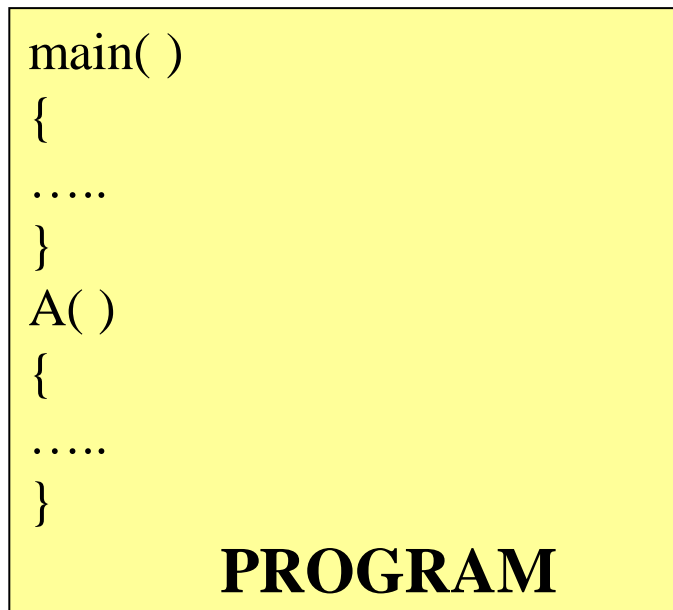
- 程序的代码；
- 程序的数据；
- PC中的值，用来指示下一条将运行的指令；
- 一组通用的寄存器的当前值，堆、栈；
- 一组系统资源（如打开的文件）

总之，进程包含了正在运行的一个程序的所有状态信息。



# Process $\neq$ Program

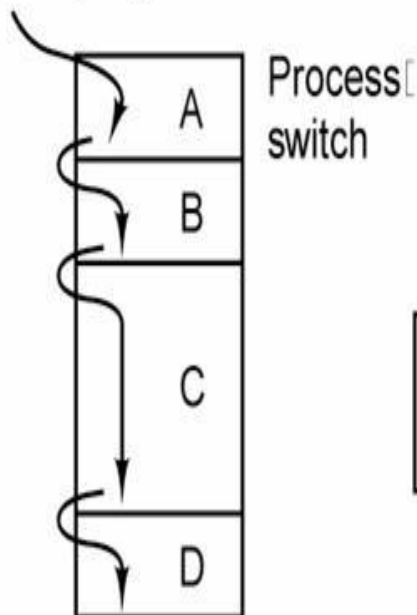
- A program is C statements or commands  
静态的;
- A process is program + running context  
动态的.



# 进程的特性

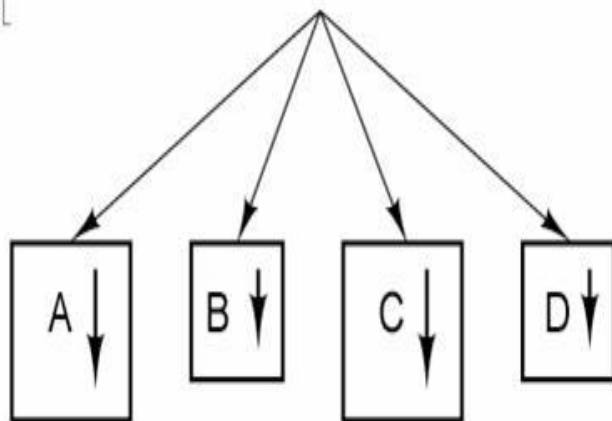
- **动态性：** 程序的运行状态在变，PC、寄存器、堆和栈等；
- **独立性：** 是一个独立的实体，是计算机系统资源的使用单位。每个进程都有“自己”的PC和内部状态，运行时独立于其他的进程（逻辑PC和物理PC）；
- **并发性：** 从宏观上看各进程是同时独立运行的

One program counter

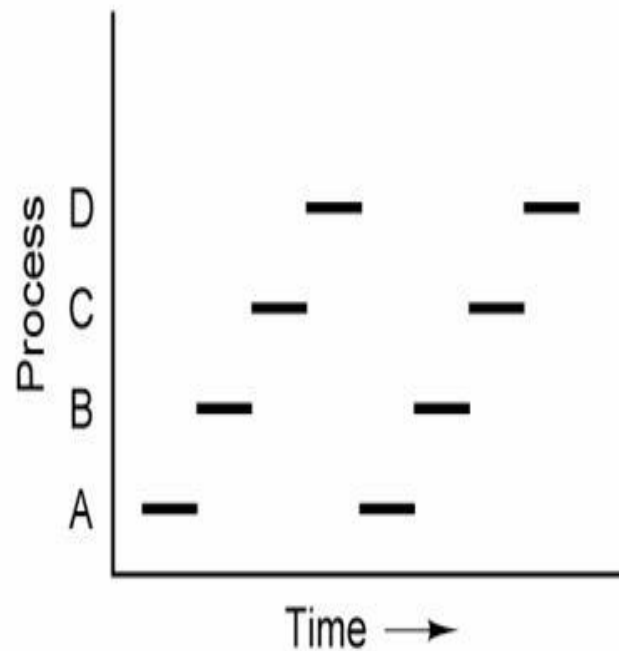


(a)

Four program counters



(b)



(c)

四个进程在并发地运行

# 什么是线程？

自从60年代提出进程概念以来，在操作系统中一直都是以进程作为独立运行的基本单位，直到80年代中期，人们又提出了更小的能独立运行的基本单位——线程。

## Why线程？

**【案例】** 编写一个MP3播放软件。核心功能模块有三个：（1）从MP3音频文件当中读取数据；（2）对数据进行解压缩；（3）把解压缩后的音频数据播放出来。

# 单进程的实现方法

```
main( )  
{  
    while(TRUE)  
    {  
        Read( );  
        Decompress( );  
        Play( );  
    }  
}  
Read( ) { ... }  
Decompress( ) { ... }  
Play( ) { ... }
```

I/O

CPU

问题:

- 播放出来的声音能否连贯?
- 各个函数之间不是并发执行, 影响资源的使用效率;

# 多进程的实现方法

程序1

```
main( )  
{  
    while(TRUE)  
    {  
        Read( );  
    }  
}  
Read( ) { ... }
```

程序2

```
main( )  
{  
    while(TRUE)  
    {  
        Decompress( );  
    }  
}  
Decompress( ) { ... }
```

程序3

```
main( )  
{  
    while(TRUE)  
    {  
        Play( );  
    }  
}  
Play( ) { ... }
```

问题：进程之间如何通信，共享数据？

# 怎么办？

需要提出一种新的实体，满足以下特性：

- （1）实体之间可以并发地执行；
- （2）实体之间共享相同的地址空间；

这种实体就是：线程（Thread）



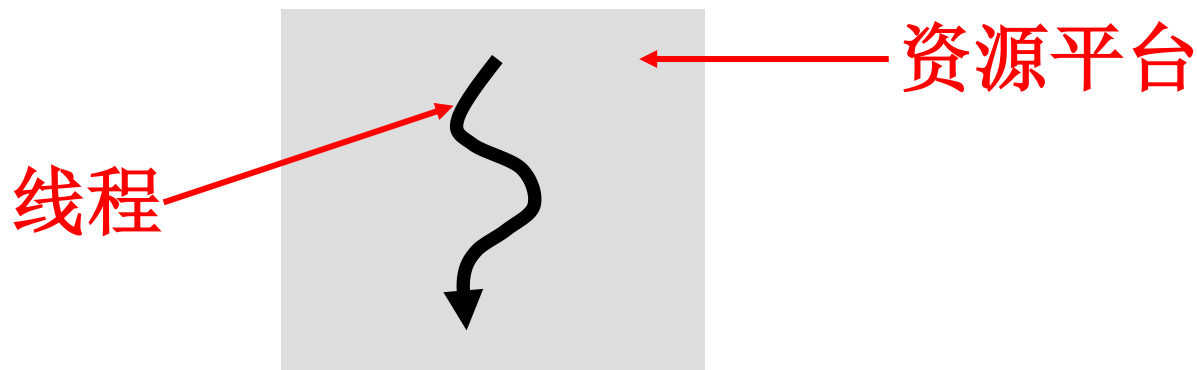
# 什么是线程？

## Thread:

- A sequential execution stream within a process;
- A **thread** of execution;
- 进程当中的一条执行流程。

从两个方面来理解进程：

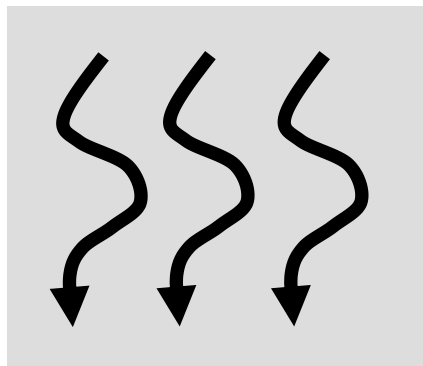
- 从资源组合的角度：进程把一组相关的资源组合起来，构成了一个资源平台（环境），包括地址空间（代码段、数据段）、打开的文件等各种资源；
- 从运行的角度：代码在这个资源平台上的一条执行流程（线程）。



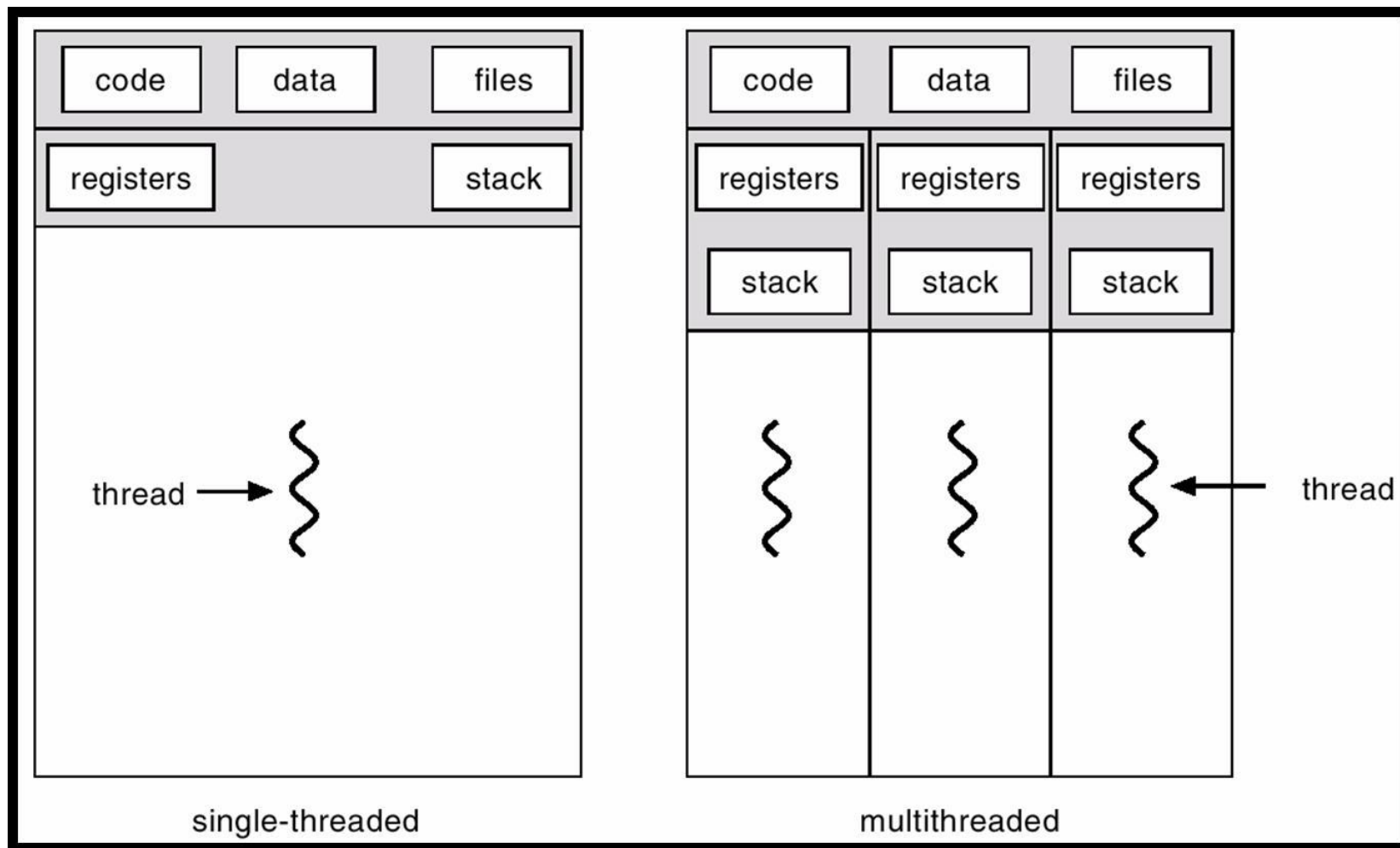
# 进程 = 线程 + 资源平台

优点：

- 一个进程中可以同时存在多个线程；
- 各个线程之间可以并发地执行；
- 各个线程之间可以共享地址空间。



# 线程所需的资源



(本图摘自Silberschatz, Galvin and Gagne: “Operating System Concepts”)

# 什么是任务？

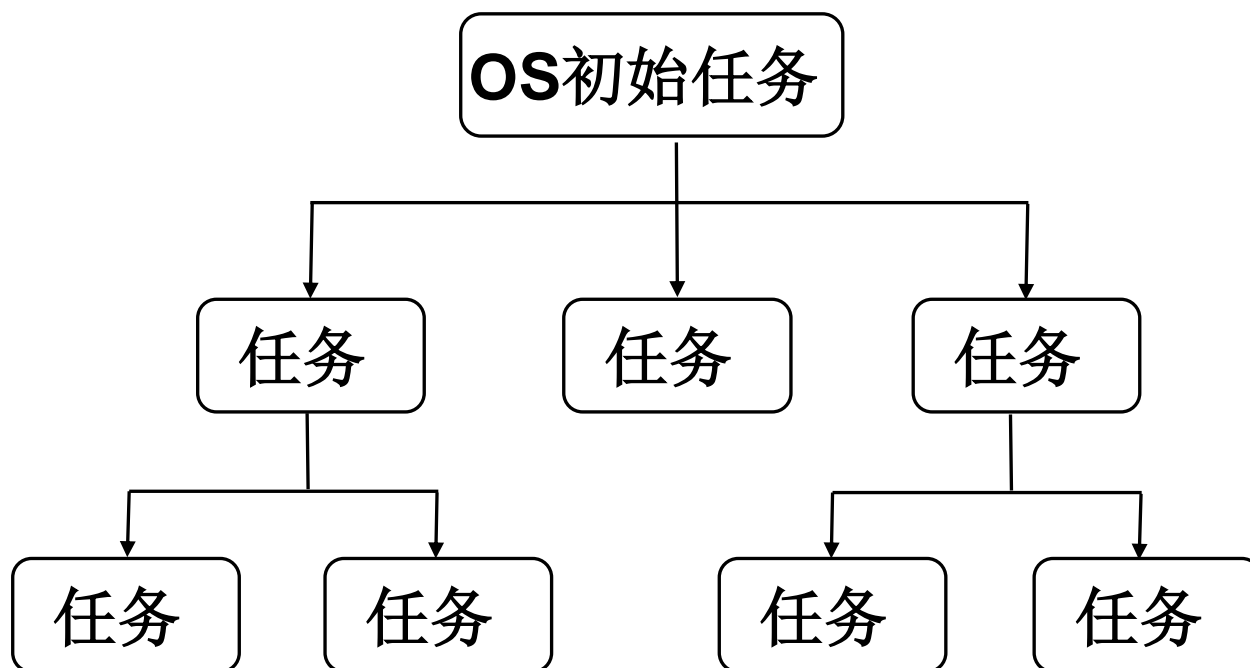
在许多嵌入式操作系统当中，一般把能够独立运行的实体称为“任务”

（Task），那么这里所说的任务到底是进程还是线程呢？

# 任务的实现

- ✦ 在多道程序（多任务）的嵌入式操作系统中，任务之间的结构为层状结构，存在着父子关系；
- ✦ 当嵌入式内核刚刚启动时，只有一个任务存在，然后由该任务派生出所有其他的任务。

# 任务的层次结构



# 任务的创建

- ✦ 在嵌入式操作系统当中，任务的创建主要有两种模型：**fork/exec**和**spawn**；
- ✦ **fork/exec**：符合**IEEE/ISO POSIX 1003.1**标准，先用**fork**系统调用创建与父任务完全相同的一份内存空间，然后再用**exec**系统调用来移除父任务的内容，并调入子任务的程序代码。优点：允许继承；
- ✦ **spawn**：直接为子任务创建一个全新的地址空间，并装入其程序代码。



# 任务的描述

问题：如果让你来设计OS当中的任务机制，那么你将如何来描述一个任务？

描述任务的数据结构：**任务控制块**  
(**Task Control Block, TCB**)。

系统为每个任务都维护了一个TCB，用来保存与该任务有关的所有信息。

# 任务控制块的内容

- 任务**ID**、任务的状态、任务的优先级；
- **CPU**上下文信息：通用寄存器的值、**PC**寄存器的值、程序状态字、栈指针的值；
- 如果在该**OS**中，任务描述的是进程，则还应包括其他的一些内容，如段表地址、页表地址等存储管理方面的信息；根目录、文件描述字等文件管理方面的信息。

系统用**TCB**来描述任务的基本情况以及运行变化的过程，**TCB**是任务存在的唯一标志。

任务的创建：为该任务生成一个**TCB**；

任务的终止：回收它的**TCB**；

任务的组织管理：通过对**TCB**的组织管理来实现。

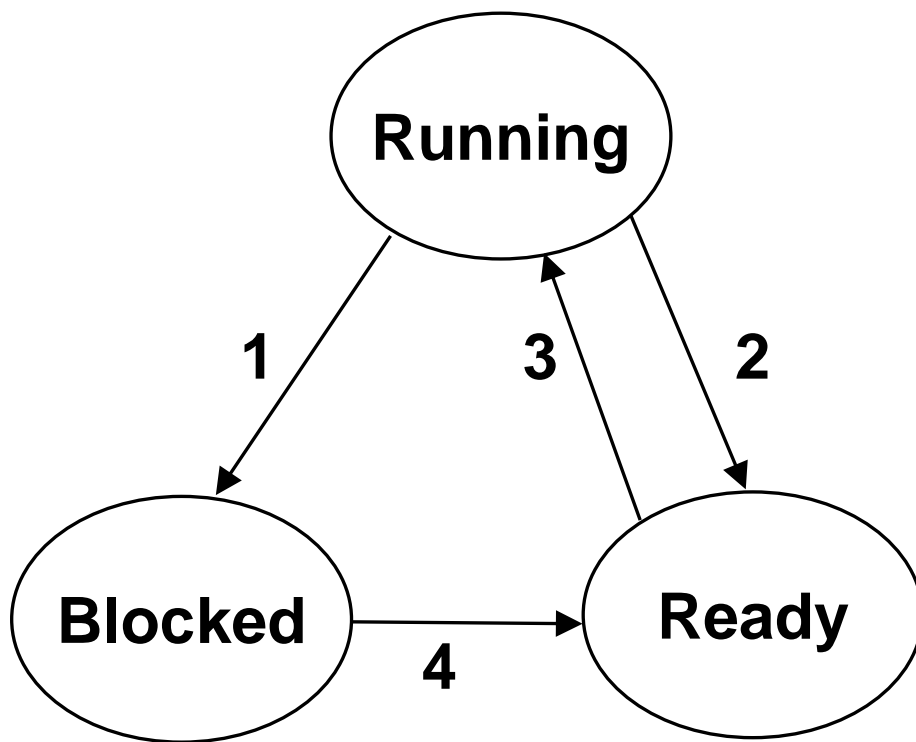
# 任务的状态

任务的三种基本状态：

任务在生命结束前处于且仅处于三种基本状态之一  
不同系统设置的任务状态数目不同。

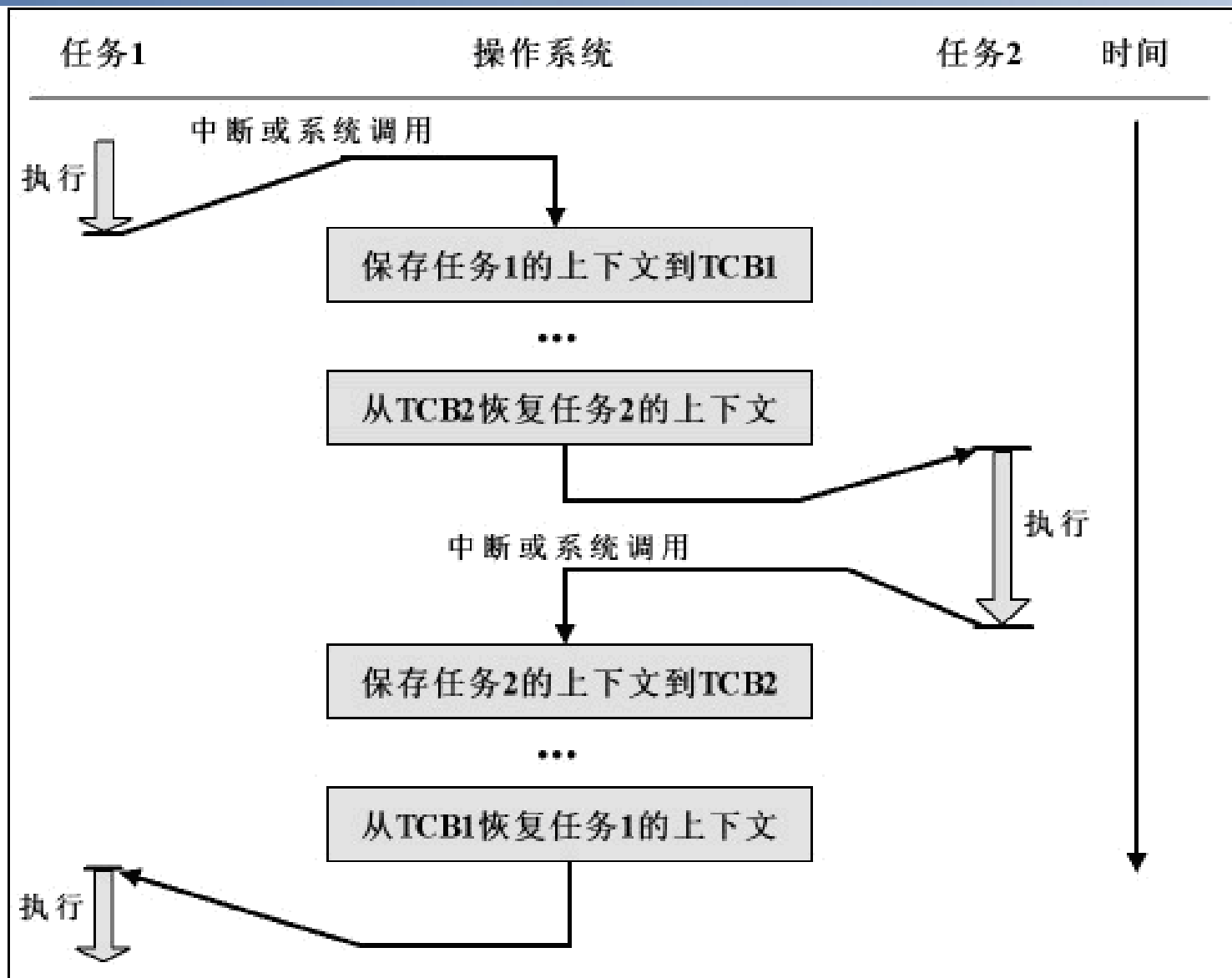
- **运行状态（Running）**：任务占有CPU，并在CPU上运行。处于此状态的任务数目小于等于CPU的数目；
- **就绪状态（Ready）**：任务已经具备运行条件，但由于CPU忙暂时不能运行，只要分得CPU即可执行；
- **阻塞/等待状态（Blocked/Waiting）**：任务因等待某种事件的发生而暂时不能运行（如I/O操作或任务同步），此时即使CPU空闲，该任务也不能运行。

# 任务的状态及其转换



1. 任务由于I/O操作被阻塞；
2. 调度器选择了另一个任务；
3. 调度器选中该任务
4. 任务的I/O操作完成了。

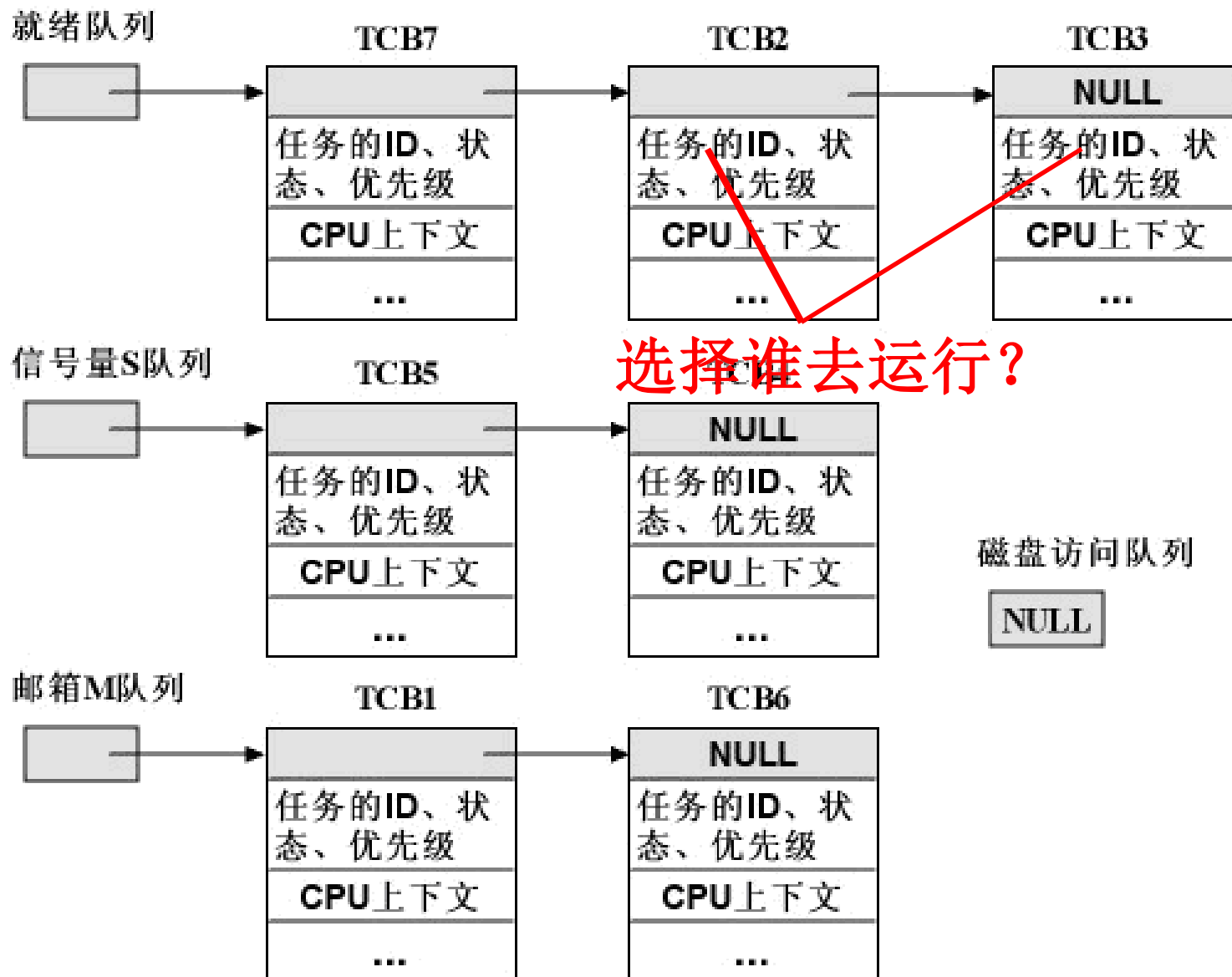
# 两个任务的状态转换过程



# 状态队列

- 由操作系统来维护一组队列，用来表示系统当中所有任务的当前状态；
- 不同的状态分别用不同的队列来表示（运行队列、就绪队列、各种类型的阻塞队列）；
- 每个任务的TCB都根据它的状态加入到相应的队列当中，当一个任务的状态发生变化时，它的TCB从一个状态队列中脱离出来，加入到另外一个队列。

# 就绪队列和各种I/O队列







# 任务的调度

- ✦ 在操作系统中，负责去做这个选择的那部分程序，称为**调度程序或调度器 (scheduler)**;
- ✦ 调度程序在决策过程中所采用的算法，称为是**调度算法**;
- ✦ 调度程序是**CPU资源的管理者**。

# 何时进行调度？

1. 当一个新的任务被创建时，是执行新任务还是继续执行父任务？
2. 当一个任务运行完毕时；
3. 当一个任务由于I/O、信号量或其他某个原因被阻塞时；
4. 当一个I/O中断发生时，表明某个I/O操作已经完成，而等待该I/O操作的任务转入就绪状态；
5. 在分时系统中，当一个时钟中断发生时。

# 两种调度方式

-  **不可抢占（non-preemptive）调度方式：**一个进程若被选中就一直运行下去，直到它被阻塞（I/O，或正在等待其他进程），或主动地交出CPU。以上的情形1—3均可发生调度；
-  **可抢占（preemptive）调度方式：**当一个进程在运行时，调度程序可以打断它。以上的情形1—5均可发生调度，另外，在其他一些情形下，如就绪队列中有新进程的优先级高于当前正运行的进程，也可能立即进行调度。

# 嵌入式调度算法的评价指标

- ◎ **响应时间**（response time）：调度器为一个就绪任务进行上下文切换的时间，以及任务在就绪队列中等待的时间；
- ◎ **周转时间**（turnaround time）：一个任务从提交到完成所经历的时间；
- ◎ **调度开销**（overhead）：调度算法在执行时所需要的时间和空间开销；
- ◎ **公平**（fairness）：大致相当的两个进程所得到的CPU时间也应大致相同的，防止**饥饿**(starvation)；
- ◎ **均衡**：尽可能使整个系统的各部分（CPU、I/O）都忙起来，提高系统资源的使用效率；
- ◎ **吞吐量**（Throughput）：单位时间内完成的任务数。

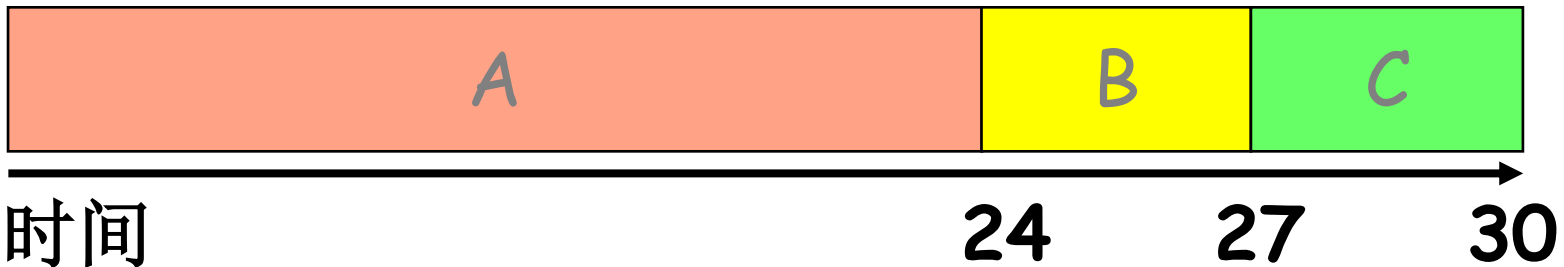
# 先来先服务调度算法

- ⊙ **先来先服务**（**First Come First Served, FCFS; First In First Out, FIFO**）：按照任务到达就绪队列的先后次序进行调度；
- ⊙ **不可抢占方式**：当前任务占用**CPU**，直到执行完或被阻塞，才让出**CPU**给另外一个任务；
- ⊙ 在任务被唤醒后（如**I/O**完成），并不立即恢复执行，而是放在就绪队列的末尾；
- ⊙ **优点**：简单、公平，易于理解也易于实现。  
现实生活中应用广泛：排队。

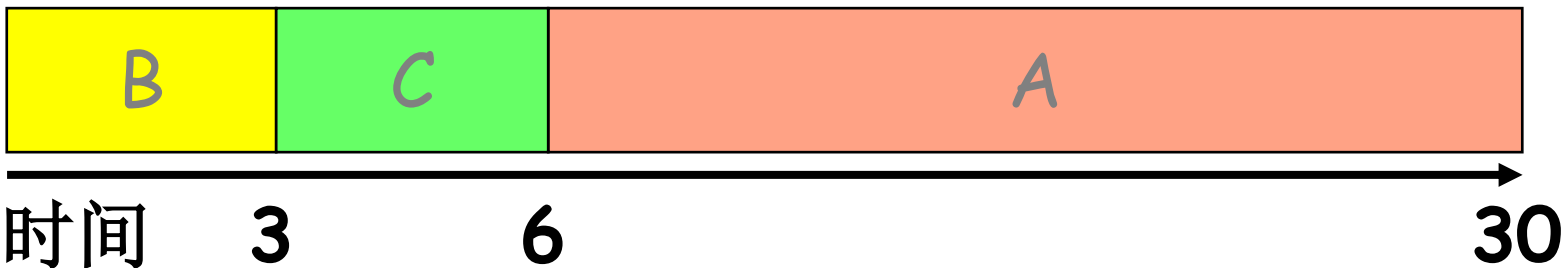
# FCFS算法的问题

平均周转时间取决于各任务到达的顺序，若短任务位于长任务之后，将增大平均周转时间。

例如，三个任务A、B、C的运行时间为24、3、3



$$\text{平均周转时间} = (24 + 27 + 30) / 3 = 27$$



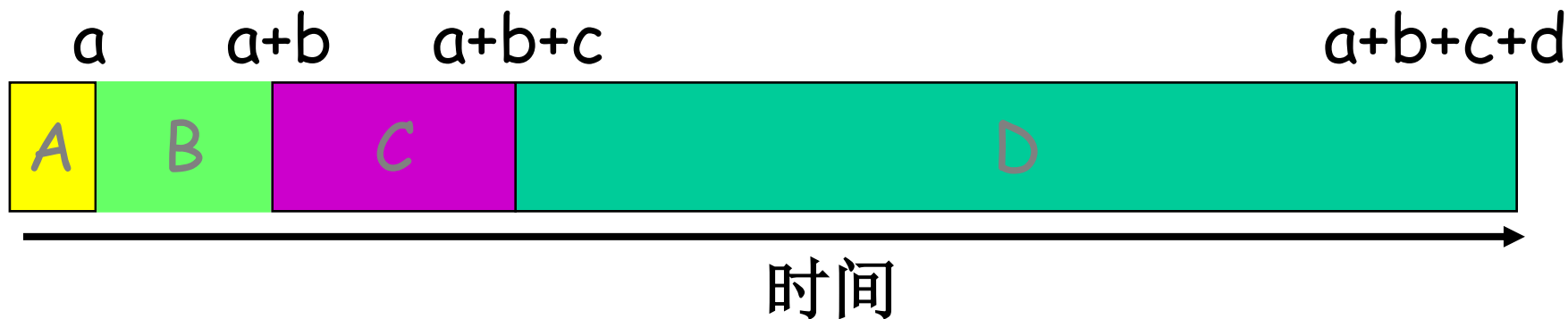
$$\text{平均周转时间} = (3 + 6 + 30) / 3 = 13$$

# 短作业优先调度算法

- ◎ **短作业优先**（**Shortest Job First, SJF**），设计目标是改进**FCFS**算法，减少平均周转时间；
- ◎ **SJF**算法要求任务在开始执行时预计执行时间，对预计执行时间短的任务优先分派处理器；
- ◎ 两种实现方案：
  - 不可抢占方式：当前任务在运行时不会被打断，只有运行完毕或阻塞时，才让出**CPU**；
  - 可抢占方式：如果一个新的短任务到来，其运行时间小于当前正在运行任务的剩余时间，则抢占**CPU**运行，称为**SRTF**（**Shortest Remaining Time First**）。

◎ 可以证明：对于一组同时到达的任务，采用SJF算法将得到一个最小的平均周转时间。

例如，考察4个任务A、B、C、D，其运行时间分别为a、b、c、d

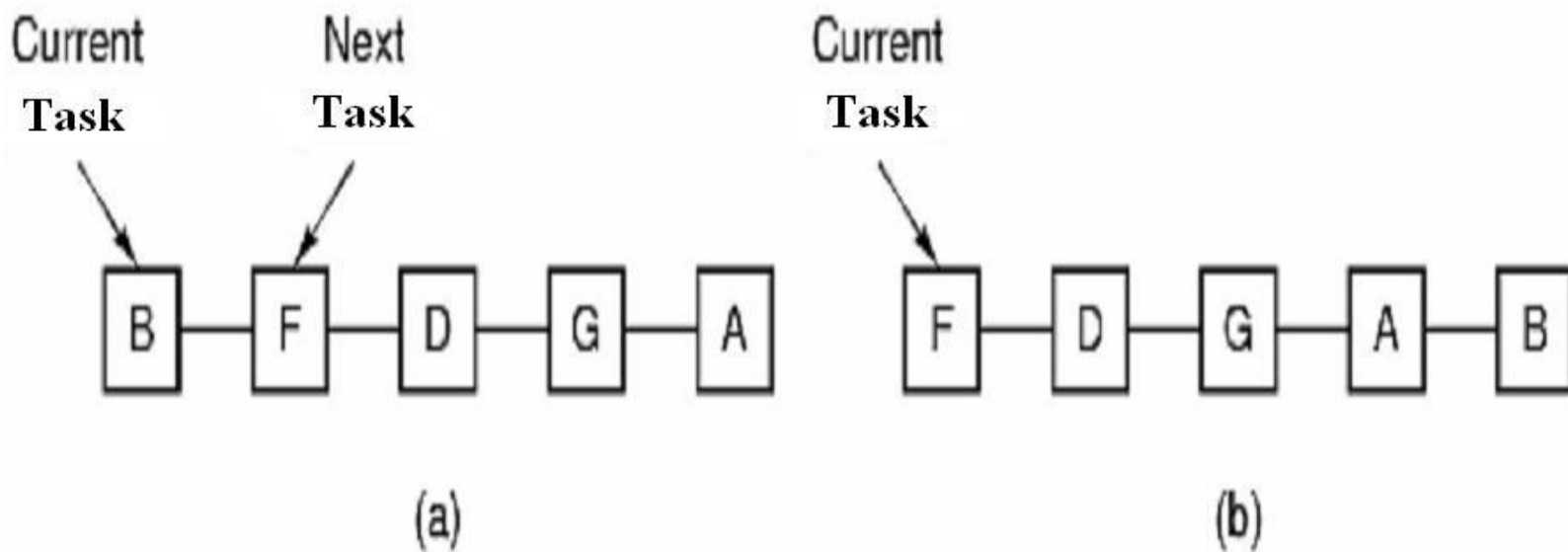


A、B、C、D的周转时间分别为a、a+b、a+b+c和a+b+c+d，因此平均周转时间为： $(4a+3b+2c+d)/4$   
显然，当 $a \leq b \leq c \leq d$ 时，平均周转时间最小。



# 时间片轮转调度算法

- ⊙ 在**时间片轮转算法**（**Round-Robin, RR**）中，将所有就绪任务按照**FCFS**原则，排成一个队列；
- ⊙ 每次调度时将处理器分派给队首任务，让其执行一小段**CPU时间**（**时间片, time slice**）；
- ⊙ 在一个时间片结束时，如果任务还没有执行完的话，将发生时钟中断，在时钟中断中，调度程序将暂停当前任务的执行，并将其送到就绪队列的末尾，然后执行当前的队首任务；
- ⊙ 如果一个任务在它的时间片用完之前就已结束或被阻塞，那么立即让出**CPU**。



开始时，任务**B**位于队列之首，因此被调度执行。当它的时间片用完后，就把它送到就绪队列的末尾。同时，任务**F**成为新的队首，被调度运行。



Round robin, too....

# 时间片轮转法的特点

## ◎ 优点:

- ⇒ **公平性**: 各个就绪任务平均地分配CPU的使用时间。假设有 $n$ 个就绪任务, 时间片大小为 $q$ , 那么每个任务将得到 $1/n$ 的CPU时间;
- ⇒ **活动性**: 每个任务最多等待 $(n-1)q$ 时间就能够再次得到CPU去运行;

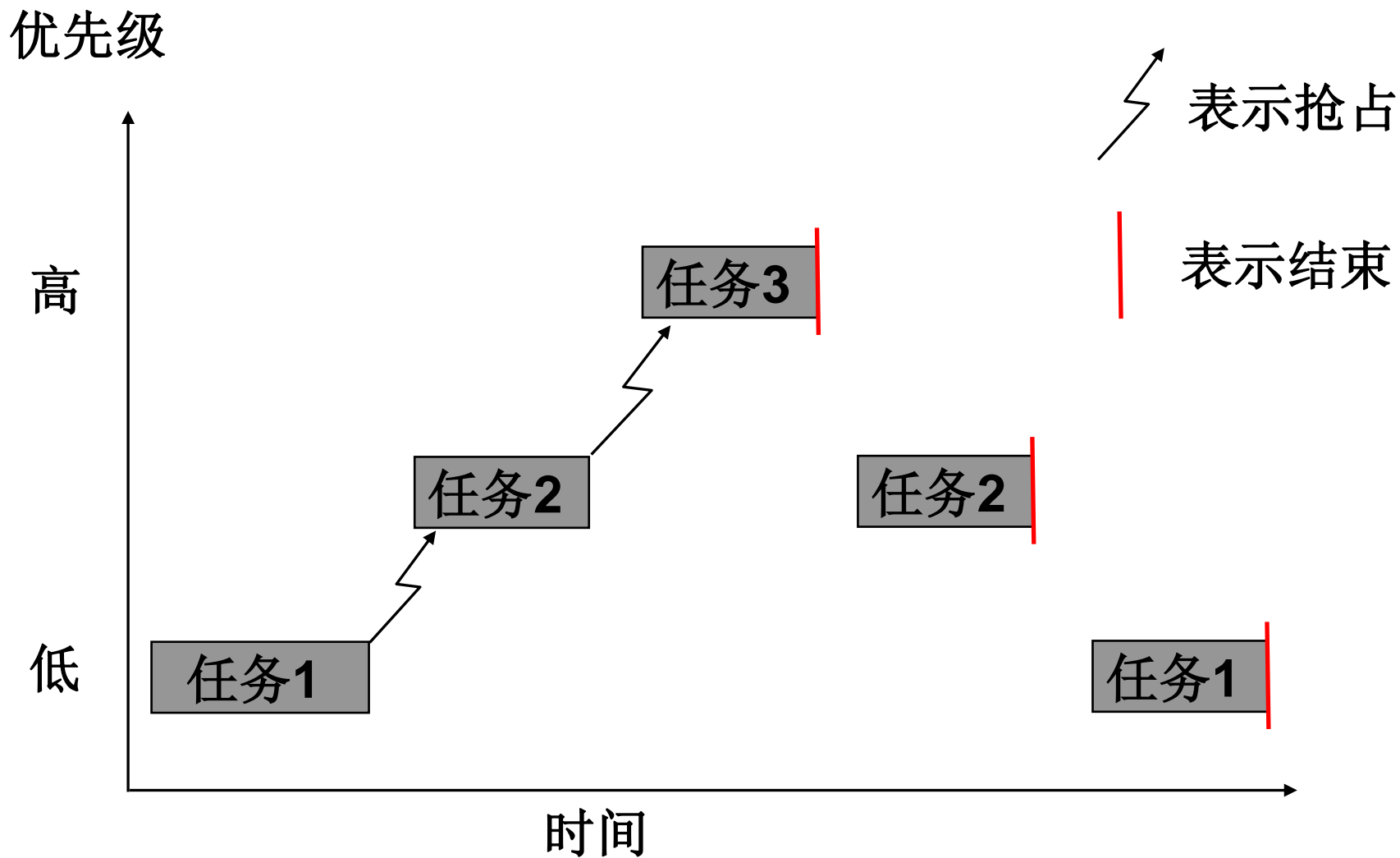
## ◎ 缺点: **$q$ 的大小难以确定** (一般在20—50ms)。

- ⇒  **$q$ 太大**: 退化为FCFS算法, 进程在一个时间片内都执行完, 响应时间长。如 $q=100\text{ms}$ ;
- ⇒  **$q$ 太小**: 每个任务都需要更多的时间片才能处理完, 任务切换次数增加, 增大系统开销。如 $q=4\text{ms}$

# 优先级调度算法

- ❏ 轮转法有一个缺省的前提，即各任务同等重要；
- ❏ “人人生而平等”？恐怕不太现实！同样，并不是每个任务都同等重要，怎么办？分等级！
- ❏ **优先级算法（Priority Scheduling）**：给每个任务设置一个优先级，然后在所有就绪任务中选择优先级最高的那个任务去运行；
- ❏ **SJF**就是一个优先级算法，每个任务的优先级是它的**CPU**运行时间（时间越短，优先级越高）；
- ❏ 分为**可抢占**和**不可抢占**两种方式；各任务优先级的确定方式可分为**静态**和**动态**两种。

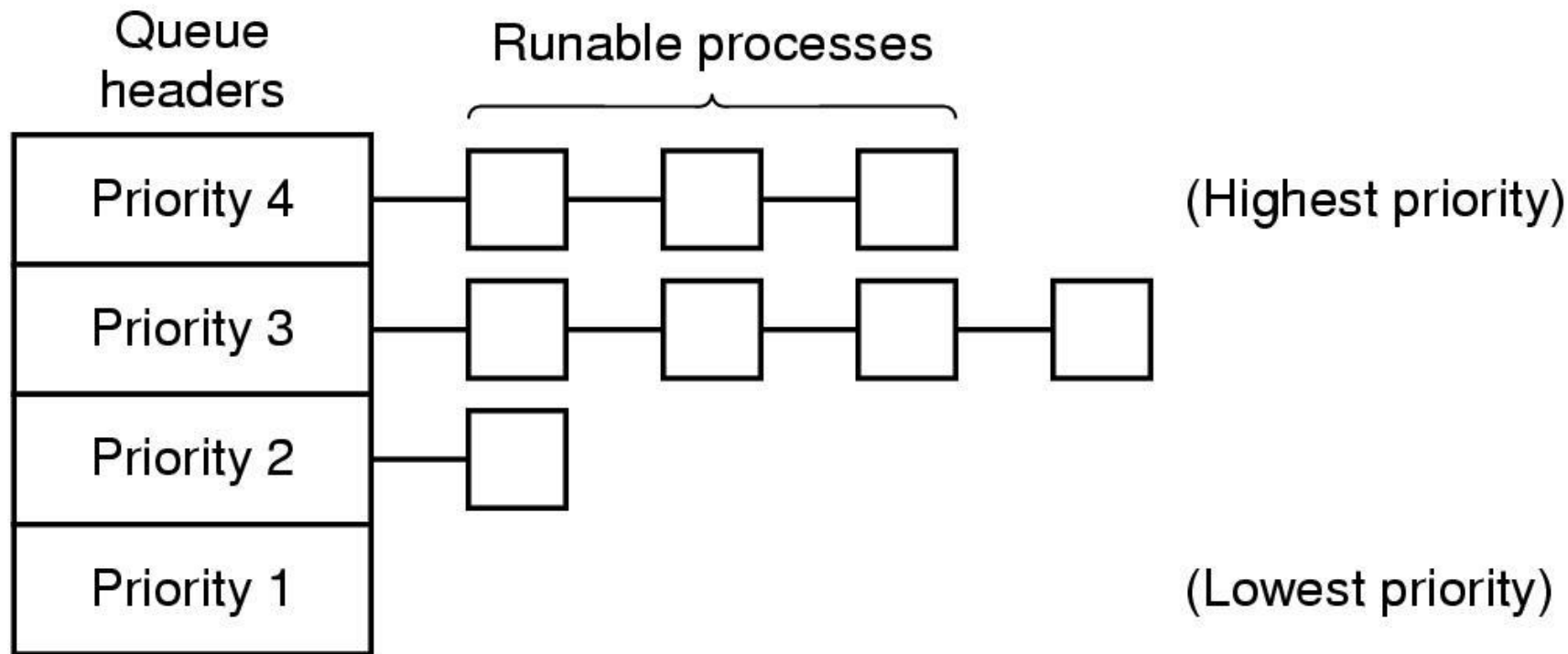
# 可抢占方式





- **静态优先级方式**：指在创建任务时即确定任务的优先级，并保持不变到任务运行结束。
  - **缺点**：如果一直有高优先级的任务出现，则它们一直占用着CPU，而低优先级的任务“饥饿”。
- **动态优先级方式**：指在创建任务时赋予给进程的优先级，在任务运行过程中可以动态改变，以便获得更好的调度性能。
  - **为防“饥饿”，根据任务的等待时间调整优先级。在就绪队列中，等待时间延长则优先级提高，从而使优先级较低的任务在等待足够的时间后，其优先级提高到可被调度执行。**

# 优先级类别



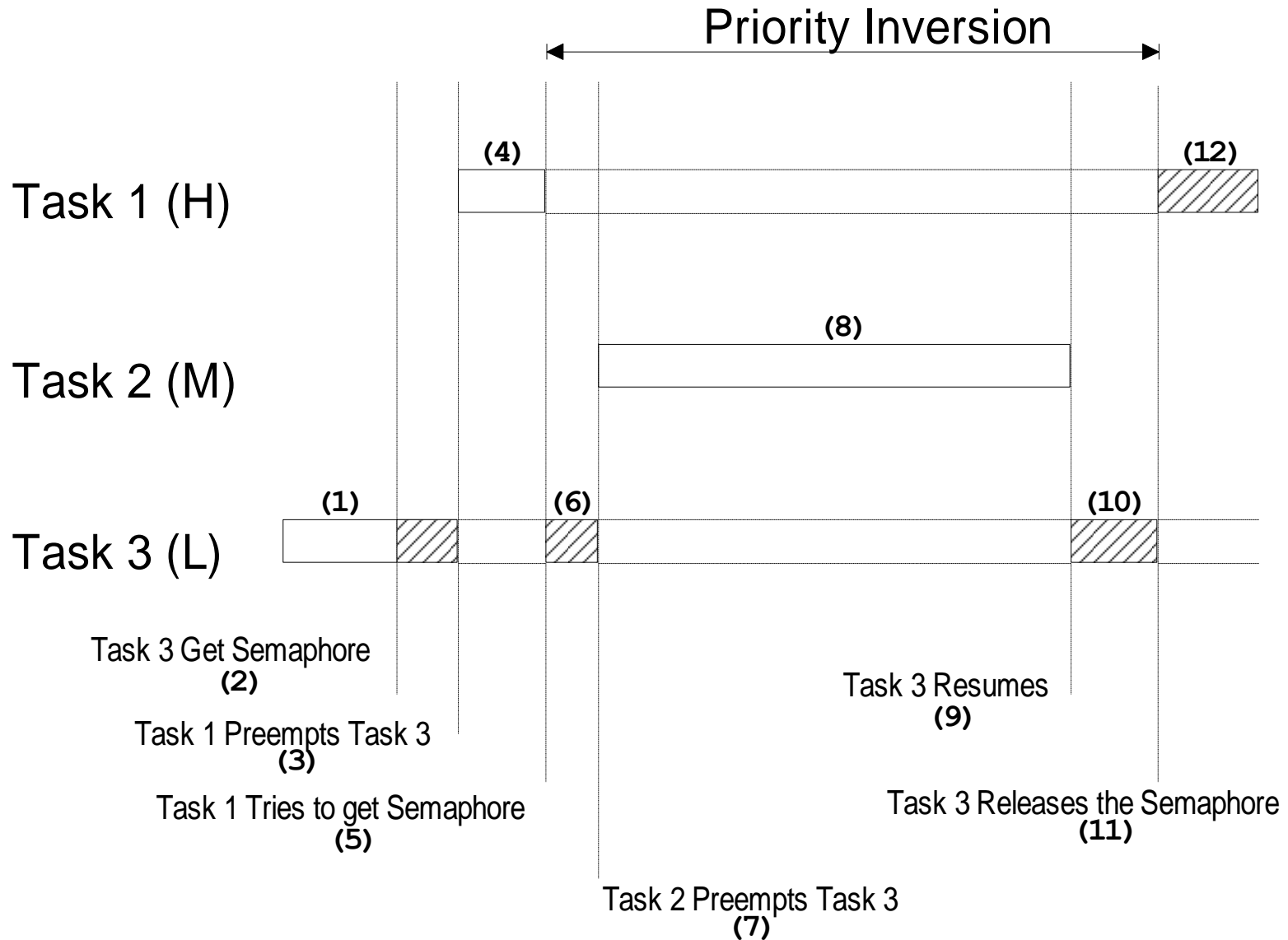
**可以把进程按照不同的优先级别分组，然后在不同级别之间使用优先级算法，而在同一级别的各个进程之间使用时间片轮转法。**



# 实时系统调度

- ✦ 对于**RTOS**调度器来说，公平性并不重要。例如：如果老师给你布置了五个家庭作业，其中有一个作业必须在一个小时内完成，显然先做那个作业；
- ✦ **RTOS**调度器的主要目标就是要使得每个任务都必须在其最终时间期限(**deadline**)之前完成。

# 优先级反转



- ✦ 大多数**RTOS**调度器都采用基于优先级的可抢占调度算法;
- ✦ 但在具体实现上:
  - **如何设定各个任务的优先级?**
  - **优先级是静态设置的还是动态可变的?**
  - **算法的性能如何, 能否满足实时要求?**

# 任务模型（周期性任务）

- **启动时间**  $r(i, j)$ : 第 $i$ 个任务的第 $j$ 次执行的开始时间;
- **时间期限Deadline**,  $D(i)$ : 第 $i$ 个任务所允许的最大响应时间（从任务启动到运行结束所需的时间）;
- **周期Period**,  $P(i)$ : 第 $i$ 个任务的连续两次运行之间的最小时间间隔;
- **执行时间Execution Time**,  $E(i)$ : 对于第 $i$ 个任务, 当它所需要的资源都已具备时, 它的执行所需要的最长时间。

# 单调速率调度算法

- ✦ 单调速率调度(Rate Monotonic Scheduling, RMS) 算法;
- ✦ RMS是一种静态优先级调度算法, 也是最常用的一种确定任务优先级的算法;
- ✦ 思路: 单位时间内任务被执行的次数越多, 优先级越高。即任务的周期越短, 优先级越高。

例如:	周期	优先级
	10	1 (最高)
	12	2
	15	3
	20	4 (最低)

- ✦ **RMS算法是一种最优算法**：如果存在一种基于静态优先级的调度顺序，使得每个任务都能够在期限内完成，那么RMS算法总能找到一种可行的调度方案；
- ✦ 对于一组任务，这种调度方案不一定存在；
- ✦ 处理器的利用率： $U = \sum_i \frac{E_i}{P_i}$ 
  - 如果 $U > 1$ ，则RMS调度方案不存在（处理器不可能一天工作25小时）；
  - 如果 $U < n(2^{1/n}-1)$ ， $n$ 为任务的个数，则RMS调度方案一定存在。

# 最早期限优先调度算法

- ✦ 最早期限优先(Earliest Deadline First, EDF) 调度算法:
- ✦ EDF是一种动态优先级调度算法，也是性能最好的一种调度算法；
- ✦ 基本思路：对时间期限最近的任务，分配最高的优先级。

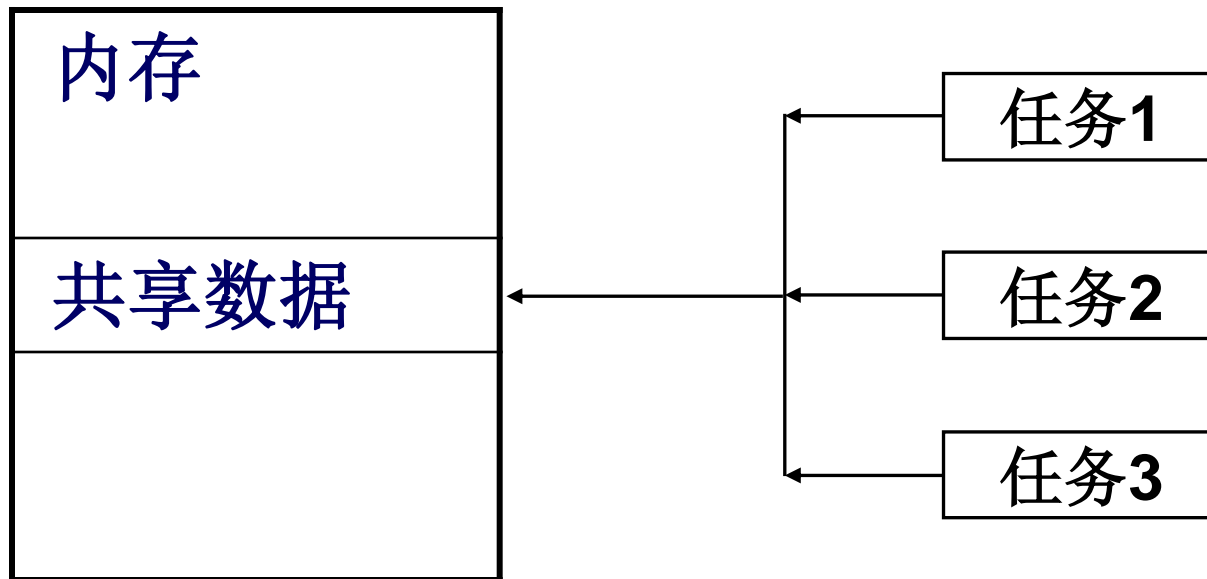
# 任务间通信

- **任务间通信（Intertask Communication）**  
：任务之间为了协调工作，需要相互交换数据和控制信息；
- **任务间通信的方式：**
  - 共享内存（shared memory）；
  - 消息传递（message passing）；
  - 管道（pipe）；
  - 信号（signal）。



# 共享内存

- ◎ 各个任务共享它们地址空间当中的某些部分，即共享内存。在此区域，可以任意读写和使用任意的数据结构（缓冲区）；
- ◎ 一组任务向共享内存中写数据，另一组任务从中读数据，通过此方式实现它们之间的信息交换。



# 消息传递

- **消息**：由若干数据位组成；
- **消息传递**：任务之间通过发送和接收消息来交换信息；
- 消息机制由OS来维护，包括定义寻址方式、认证协议、消息的大小等。一般提供两个操作：
  - **send()**，发送一条消息；
  - **receive()**，接收一条消息。
- 如果两个任务P和Q想要进行通信，它们需要
  - 在两者之间建立一个通信链路；
  - 使用**send()**和**receive()**交换信息。

# 直接通信

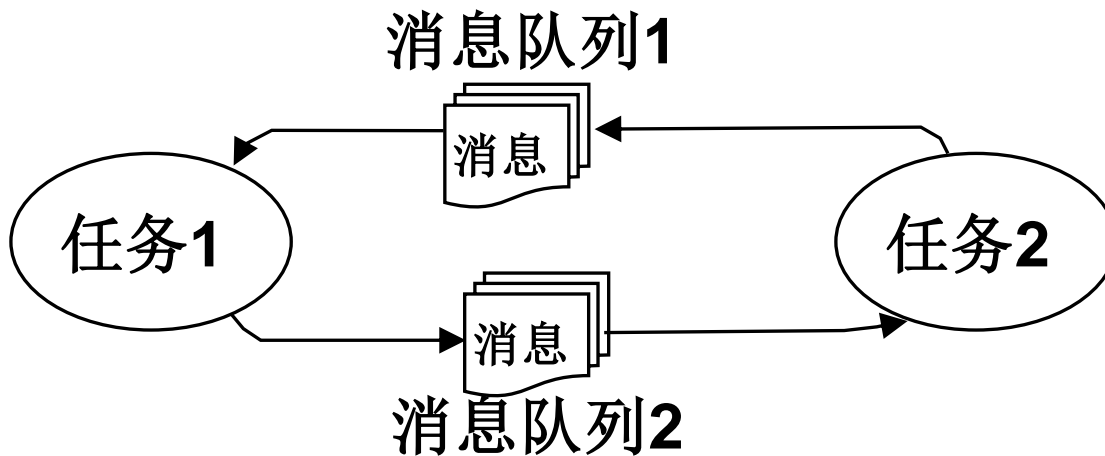
- **直接通信**：通信双方必须指明与之通信的对象。
  - **send(P, message)**：发送一条消息给任务P；
  - **receive(Q, message)**：从任务Q那里接收一条消息。如果没有收到消息，可以阻塞起来等待消息的到来，也可以立即返回；
- 通信双方之间存在一条通信链路
  - **通信链路是自动建立的，由OS来维护；**
  - **每条链路只涉及两个任务，每对任务之间仅存在一条链路；**
  - **通信链路可以是单向或双向的。**

# 间接通信1

- **间接通信**：通信时不必指明发送或接收的对象，而是通过共享的**邮箱**来发送和接收消息。
  - **send(A, message)**：发送一条消息给邮箱A；
  - **receive(A, message)**：从邮箱A接收一条消息。
- **间接通信的特点**
  - **对于一对任务，只有当它们共享一个公共邮箱时才能进行通信；**
  - **一个邮箱可以被多个任务访问，每对任务也可以使用多个邮箱来通信；**
  - **可以是单向或双向的。**

## 间接通信2

- 邮箱只能存放单条消息，其状态只有两种：空或满。另一种间接通信机制是：消息队列。
- 消息队列与邮箱类似，但可以同时存放若干条消息，提供了一种任务间缓冲通信的方法。



# 管道 (pipe)

- 管道通信由**UNIX**首创，由于其有效性，后来的一些系统相继引入了管道技术；
- 管道通信以文件系统为基础，所谓管道即连接两个任务之间的一个打开的共享文件，专用于任务之间的数据通信；
- 发送任务从管道的一端写入数据流，接收任务从管道的另一端按先进先出的顺序读出数据流；
- 管道的读写操作即为文件操作**fwrite/fread**，数据流的长度和格式没有限制。

# 信号 (signal)

- 信号（异步信号）是任务的一个标识，表明某个异步事件已经发生了，该事件可能来自于外部（如其他的任务、硬件或定时器），也可能来自于内部（如执行指令出错）；
- 信号机制用于任务与任务之间、任务与中断服务程序ISR之间的异步操作，异步信号被任务（或ISR）用来通知其他任务某个事件的出现；
- 当任务收到一个信号后，将暂停执行其自身的代码，转而去运行相应的信号处理程序。

# 信号与中断

- 相同点
  - 具有中断性;
  - 有相应的处理程序;
  - 可以屏蔽响应;
- 不同点
  - 中断由硬件或特定的指令产生，信号由系统调用产生;
  - 中断触发后，硬件会根据中断向量找到相应的处理程序去执行；信号通过发送信号的系统调用触发，但系统不一定马上对它进行处理；
  - 中断处理程序是在系统内核的上下文中运行，是全局的；而信号处理程序是在相关任务的上下文中运行，是任务的一个组成部分。



# 同步与互斥

- ☹️ 多数操作系统（包括分时和实时）都是多任务系统，允许多个任务同时运行；
- ☹️ 当两个或多个任务在访问共享资源（如共享内存）的时候，如何确保它们不会相互妨碍 —— 任务互斥问题；
- ☹️ 当两个或多个任务之间存在着某种依存关系时，如何来调整它们的运行次序 —— 任务同步问题。

## 竞争条件（race condition）：

两个或多个任务对同一共享数据同时进行读写操作，而最后的结果是不可预测的，它取决于各个任务的具体运行情况。

## 解决之道：

在同一时刻，只允许一个任务访问该共享数据，即如果当前已有一个任务正在使用该数据，那么其他任务暂时不能访问。这就是**互斥**的概念。

# 竞争条件问题的抽象描述

把一个任务在运行过程中所做的事情分为两类：

- 任务内部的计算或其他的一些事情，肯定不会导致竞争条件的出现；
- 对共享内存或共享文件的访问，可能会导致竞争条件的出现。我们把完成这类事情的那段代码称为“**临界区**”（Critical Region），把需要互斥访问的共享资源称为“**临界资源**”。

如果我们能设计出某种方法，使得任何两个任务都不会同时出现在临界区中，就可以避免竞争条件的出现。

# 关闭中断的解决方案

当一个任务进入临界区后，关闭所有的中断；当它退出临界区时，再打开中断。

- ❖ 把关闭中断的权力授予用户任务，是很不聪明的。而且假设系统有多个CPU，则此法无效；
- ❖ 对内核任务而言，是一种方便有效的办法。

结论：适用于内核任务，但不适用于用户任务，不能作为一种普遍适用的互斥实现方法。

# 基于繁忙等待的解决方案

可以采用各种基于**繁忙等待**(busy waiting)的策略，基本思路是：当一个任务想要进入它的临界区时，首先检查一下是否允许它进入，若允许，就直接进入了；若不允许，就在那里循环地等待，一直等到允许它进入。

```
while (TestAndSet (lock));  
临界区  
lock = FALSE;  
非临界区
```

——浪费CPU时间

基于繁忙等待策略的方法

# 信号量

- 1965年由著名的荷兰计算机科学家Dijkstra提出，其基本思路是用一种新的变量类型（semaphore）来记录当前可用资源的数量。
- 有两种实现方式：1) semaphore的取值必须大于或等于0。0表示当前已没有空闲资源，而正数表示当前空闲资源的数量；2) semaphore的取值可正可负，负数的绝对值表示正在等待进入临界区的任务个数。
- 信号量是由操作系统来维护的，任务只能通过初始化和两个标准原语（P、V原语）来访问。初始化可指定一个非负整数，即空闲资源总数。

- **P、V原语**作为操作系统内核代码的一部分，是一种不可分割的原子操作（**atomic action**），在其运行时，不会被时钟中断所打断；
- **P、V原语**包含有任务的阻塞和唤醒机制，因此在任务等待进入临界区时不会浪费**CPU**时间；
- **P原语**：**P**是荷兰语**Proberen**（测试）的首字母。申请一个空闲资源（把信号量减1），若成功，则退出；若失败，则该任务被阻塞；
- **V原语**：**V**是荷兰语**Verhogen**（增加）的首字母。释放一个被占用的资源（把信号量加1），如果发现**有被阻塞的任务**，则选择一个唤醒之。

# 信号量和P、V原语的实现

## 信号量结构体类型的定义

```
typedef struct  
{  
    int count;                // 计数变量  
    struct TCB *queue; // 进程等待队列  
} semaphore;
```



## P原语：申请一个资源

```
P( semaphore S)
{
    --S.count;           //表示申请一个资源;
    if (S.count < 0)     //表示没有空闲资源;
    {
        该任务进入等待队列S.queue末尾;
        阻塞该任务;
    }
}
```

## V原语：释放一个资源

```
V( semaphore S)  
{  
    ++S.count;           //表示释放一个资源;  
    if (S.count <= 0) //表示有进程被阻塞;  
    {  
        从等待队列S.queue中取出一个进程;  
        把该进程改为就绪状态, 插入就绪队列  
    }  
}
```

- **Windows 2000**

- **CreateSemaphore**（创建信号量）
- **WaitForSingleObject**（P操作）
- **ReleaseSemaphore**（V操作）

- **uCOS**

- **osSemCreate**（创建信号量）
- **osSemPend**（P操作）
- **osSemPost**（V操作）

# 利用信号量来实现进程互斥

```
semaphore mutex;  
mutex.count = 1;    // N = 1
```

```
P(mutex);  
临界区  
V(mutex);  
非临界区
```

# 任务同步

任务间的**同步**是指多个任务中发生的事件存在某种时序关系，因此在各个任务之间必须协同合作，相互配合，使各个任务按一定的速度执行，以共同完成某一项工作。

## 【例子】 司机与售票员

```
while(上班时间)
{
    发动汽车;
    正常运行;
    到站停车;
}
```

公交车司机

```
while(上班时间)
{
    关闭车门;
    售票;
    打开车门;
}
```

售票员

只有关闭车门以后，才能启动汽车；只有停车以后，才能打开车门。

# 基于信号量的任务同步

```
semaphore S_DoorClose;    // 初始为0  
semaphore S_Stop;        // 初始为0
```


```
while(上班时间)  
{  
    P(S_DoorClose);  
    发动汽车;  
    正常运行;  
    到站停车;  
    V(S_Stop);  
}
```

先关门  
后开车



```
while(上班时间)  
{  
    关闭车门;  
    V(S_DoorClose);  
    售票;  
    P(S_Stop);  
    打开车门;  
}
```

先停车  
后开门



# 内存布局

<b>.text</b>	<b>.data</b>	<b>.bss</b>	<b>堆空间</b>	<b>栈空间</b>
--------------	--------------	-------------	------------	------------

- ☯ **.text**: 代码段，包含操作系统和应用程序的所有代码；
- ☯ **.data**: 数据段，存放了操作系统和应用程序当中所有带有初始值的全局变量；
- ☯ **.bss**: **bss**段，存放了操作系统和应用程序当中所有未带初始值的全局变量；
- ☯ **堆空间**: 动态分配的内存空间，**malloc/free**；
- ☯ **栈空间**: 保存运行上下文以及函数调用时的局部变量和行参。



- ☯ 对于.text、.data和.bss段，它们的大小在编译时即可确定，称为静态段；
- ☯ 对于堆和栈，它们的大小在编译时不能确定，而且在运行时会发生变化，称为动态段（有些嵌入式系统以静态数组的方式来实现任务的栈空间，其大小是固定的）；
- ☯ 问题：堆管理器如何实现？

## I/O设备的类型

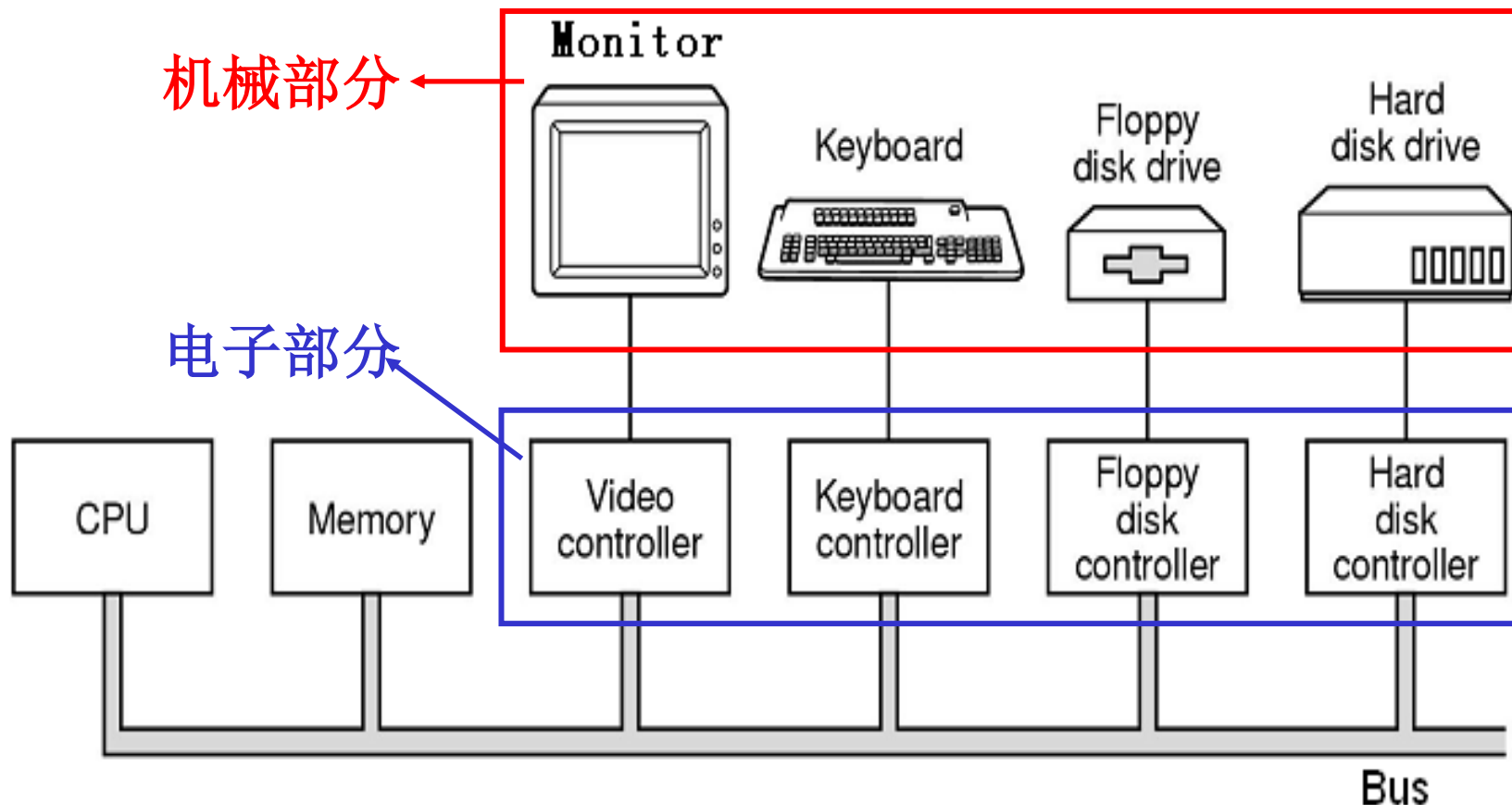
### ★按交互方向分类:

- 输入设备：键盘、鼠标、扫描仪；
- 输出设备：显示器、打印机；
- 输入/输出：磁盘、网卡。

### ★按数据组织分类:

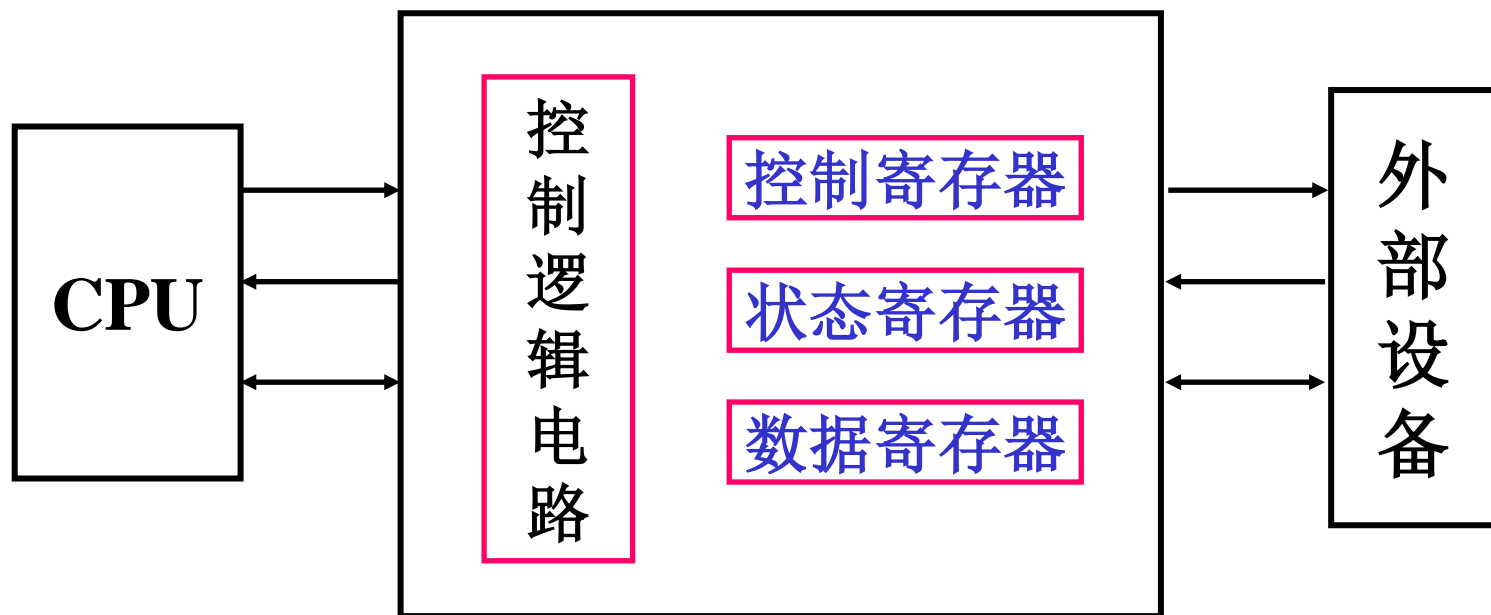
- 块设备：以数据块来作为信息的存储和传输单位，每个数据块都有一个地址，如磁盘；
- 字符设备：以字符来作为信息的存储和传输单位，如打印机。

# 设备控制器



一个I/O单元由两部分组成：**机械部分**和**电子部分**。机械部分即为I/O设备本身；电子部分称为：**设备控制器**或**适配器**，它的功能是完成设备与主机间的连接和通讯。

# I/O地址



每个设备控制器都有一些寄存器用来与CPU通信。通过往这些寄存器中写入不同的值，OS能命令该设备去执行发送数据、接收数据、打开、关闭等操作；OS也能通过读取这些寄存器的值来了解设备的当前状态。

此外，许多设备还有一个数据缓冲区供OS读写。

现在的问题是：**CPU**如何与设备控制器当中的寄存器以及数据缓冲区来进行通信？  
因为这不是普通的内存访问！

方法有三种：

- **I/O独立编址；**
- **内存映像编址；**
- **混合编址。**

# I/O独立编址

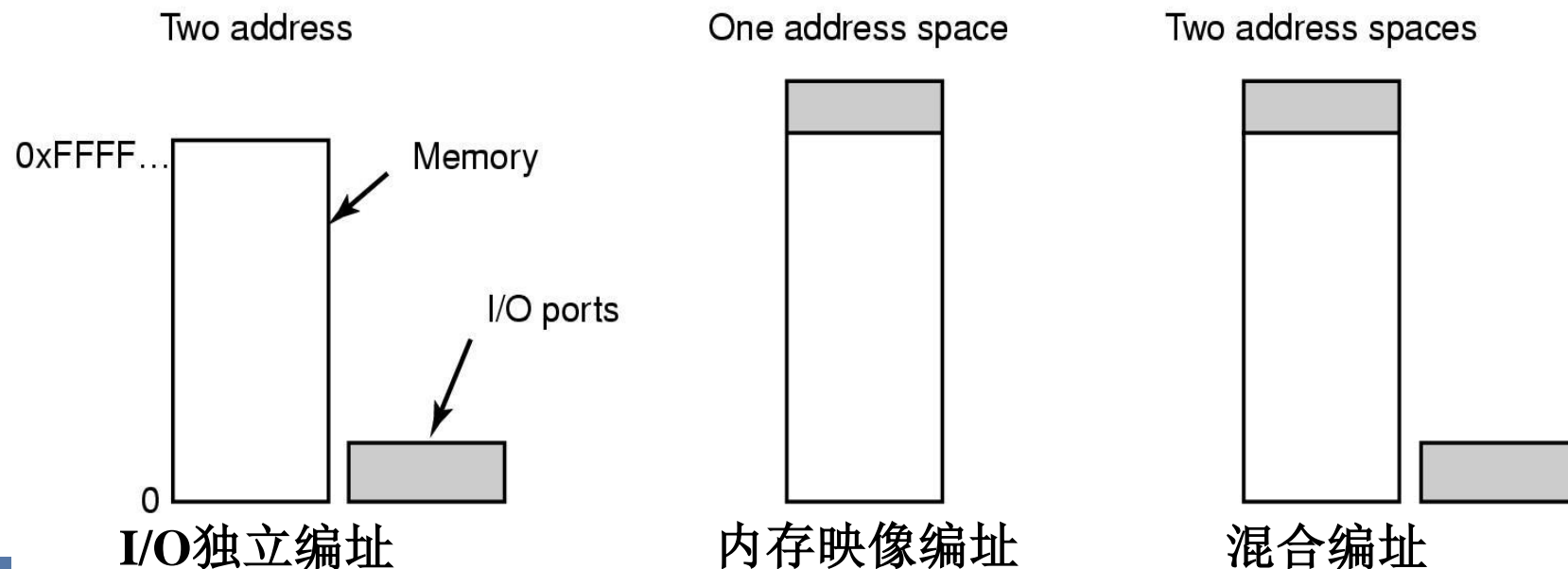
- ◆ 基本思路：给控制器中的每一个寄存器分配一个唯一的**I/O端口**（I/O port）编号，称为**I/O端口地址**，然后用专门的I/O指令对端口进行操作；
- ◆ 这些端口地址所构成的地址空间是完全独立的，与内存的地址空间没有关系。例如：  
**IN R0 [4]** 表示读入I/O端口地址为4的内容；  
**MOV R0 [4]** 表示读入内存地址为4的内容；
- ◆ 优点：I/O设备不占用内存地址空间，而且程序设计时，易于区分是对内存操作还是对I/O端口操作。
- ◆ 例子：8086/8088，给I/O端口分配的地址空间64K，0000H~FFFFH，只有IN和OUT指令进行读写操作。

# 内存映像编址

- ◆ 基本思路：把所有控制器当中的每一个寄存器都映射为一个内存单元，专门用于I/O操作（功能上），对这些单元的读写操作即为普通的内存访问操作。
- ◆ 端口地址空间与内存的地址空间统一编址，前者是后者的一部分，一般位于后者的顶端部分。
- ◆ 优点：
  - ☞ 编程方便，无需专门的I/O指令(C vs. 汇编)；
  - ☞ 对普通的内存单元可进行的所有操作指令均可作用于I/O端口，如TEST指令；
  - ☞ 无须专门的保护机制来防止用户进程执行I/O。

# 混合编址

- ◆ 基本思路：对于设备控制器中的寄存器，采用独立编址的方法；而对于设备的数据缓冲区，采用内存映像编址的方法。
- ◆ 例如：Pentium，把内存地址空间640K~1M保留作为设备的数据缓冲区，另外，还有一个独立的I/O端口地址空间，从0到64K。





到目前为止，已经介绍了I/O设备的类型、设备的控制器、I/O的端口地址。现在的问题是：根据已有的这些知识，现在能否开始使用这些I/O设备，完成相应的输入输出功能呢？若能，如何来使用？

答案是能！

方法：程序循环检测I/O(Programmed I/O)。

- 程序循环检测方式(Programmed I/O);
- 中断驱动方式(Interrupt-driven I/O);
- 直接内存访问方式(DMA, Direct Memory Access);

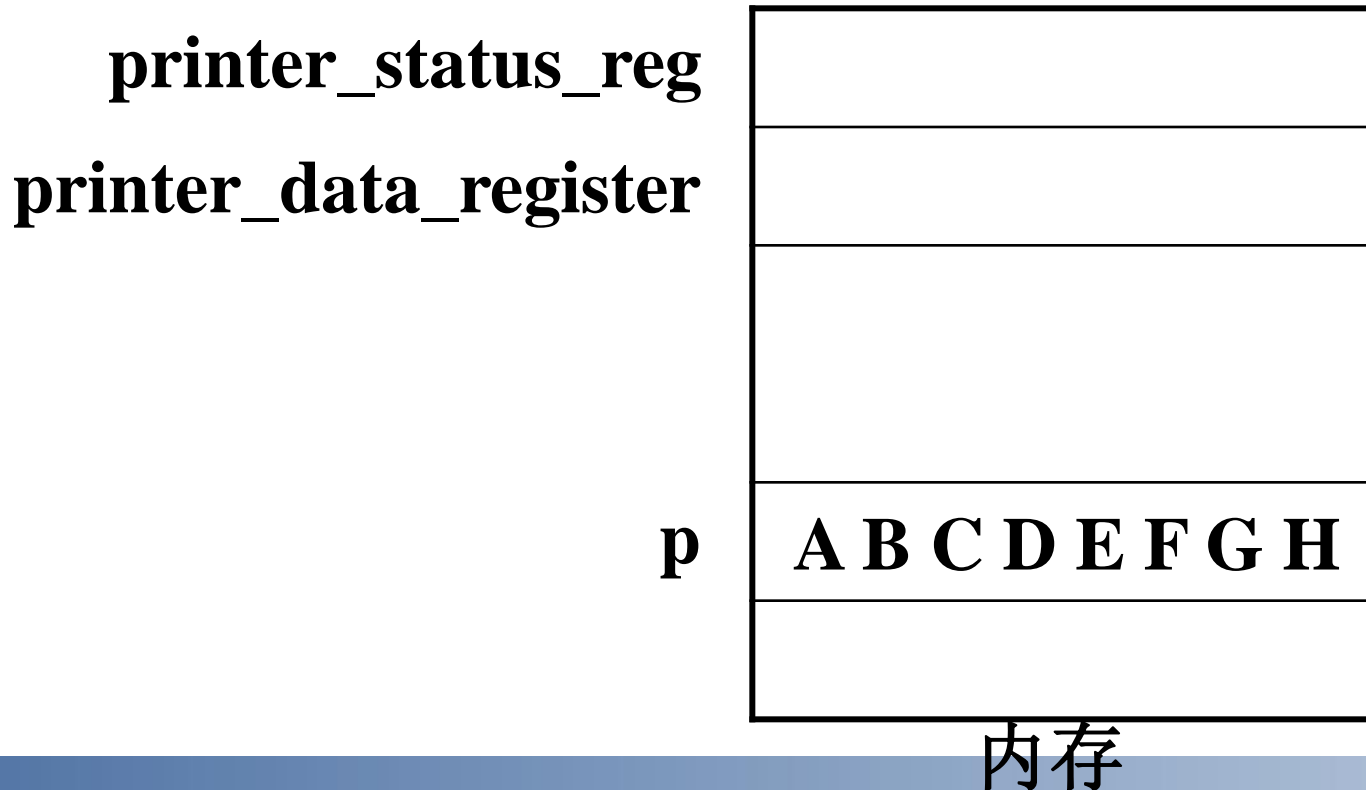
# 程序循环检测方式

- ◆ 基本思路：在程序（设备驱动程序）中通过不断地检测I/O设备的当前状态，来控制I/O操作的完成。具体来说，在进行I/O操作之前，要循环地检测设备是否就绪；在I/O操作进行之中，要循环地检测设备是否已完成；在I/O操作完成之后，还要把输入的数据保存到内存（输入操作）。从硬件来说，控制I/O的所有工作均由CPU来完成。
- ◆ 也称为**繁忙等待方式**（busy waiting）或**轮询方式**（polling）。
- ◆ 缺点：在进行I/O操作时，一直占用CPU时间。

## 一个例子

已知I/O地址采用**内存映像编址**的方式，现在需要在打印机上打印一个字符串“ABCDEFGH”。

基本思路：把这8个字符逐个送到打印机设备的I/O端口地址（内存地址）。



## 程序循环检测方式

```
for (i = 0; i < count; i++)  
{  
    while(*printer_status_reg != READY);  
    *printer_data_register = p[i];  
}
```

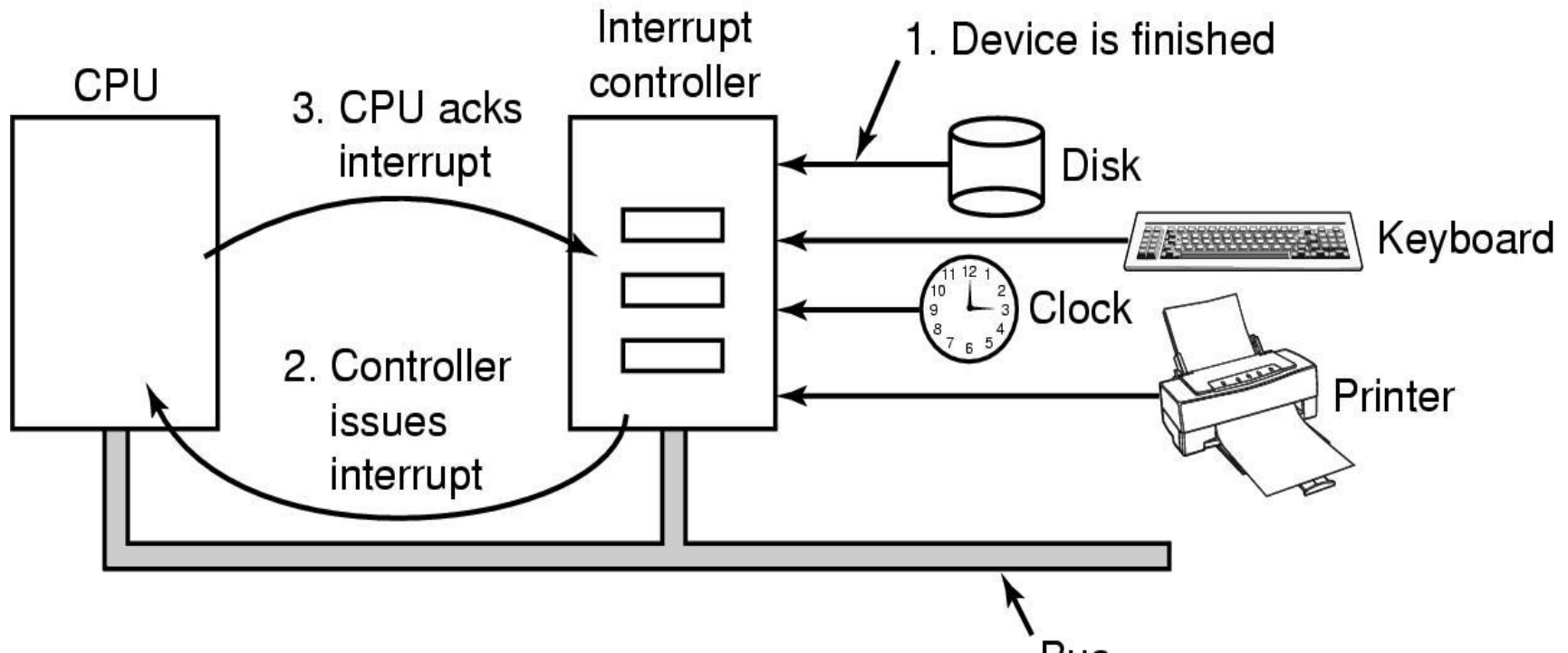
逐个打印每一个字符：先判断打印机是否空闲，若不空闲则循环等待。然后把第 *i* 个字符复制给打印机的数据寄存器（内存单元）。

# 中断驱动方式

循环检测的控制方法占用了太多的CPU时间，可能会造成CPU时间的浪费。例如：假设打印机的打印速度为100字符/秒，在循环检测方式下，当一个字符被写入到打印机的数据寄存器中后，CPU需要等待10毫秒才能写入下一个字符。

一种解决的办法：**中断驱动**的控制方式。

# 中断机制



在硬件一级，当一个I/O设备完成任务时，它的控制器会通过总线向中断控制器发出一个信号，如果中断控制器接受了该信号，就把标明该设备的一个编号放在地址线上，并向CPU发出一个中断信号。CPU就中断当前工作，并以该编号为索引去访问中断向量表，取出中断处理程序的起始地址，并在该程序运行后向中断控制器发出确认信号。

# 中断驱动方式

## 用户进程

```
strcpy(buffer, "ABCDEFGH");  
print(buffer, strlen(buffer));
```

## 系统调用函数print

```
copy_from_user(buffer, p, count);  
enable_interrupts( );  
while(*printer_status_reg != READY);  
*printer_data_register = p[0];  
scheduler( );
```



## 中断处理程序

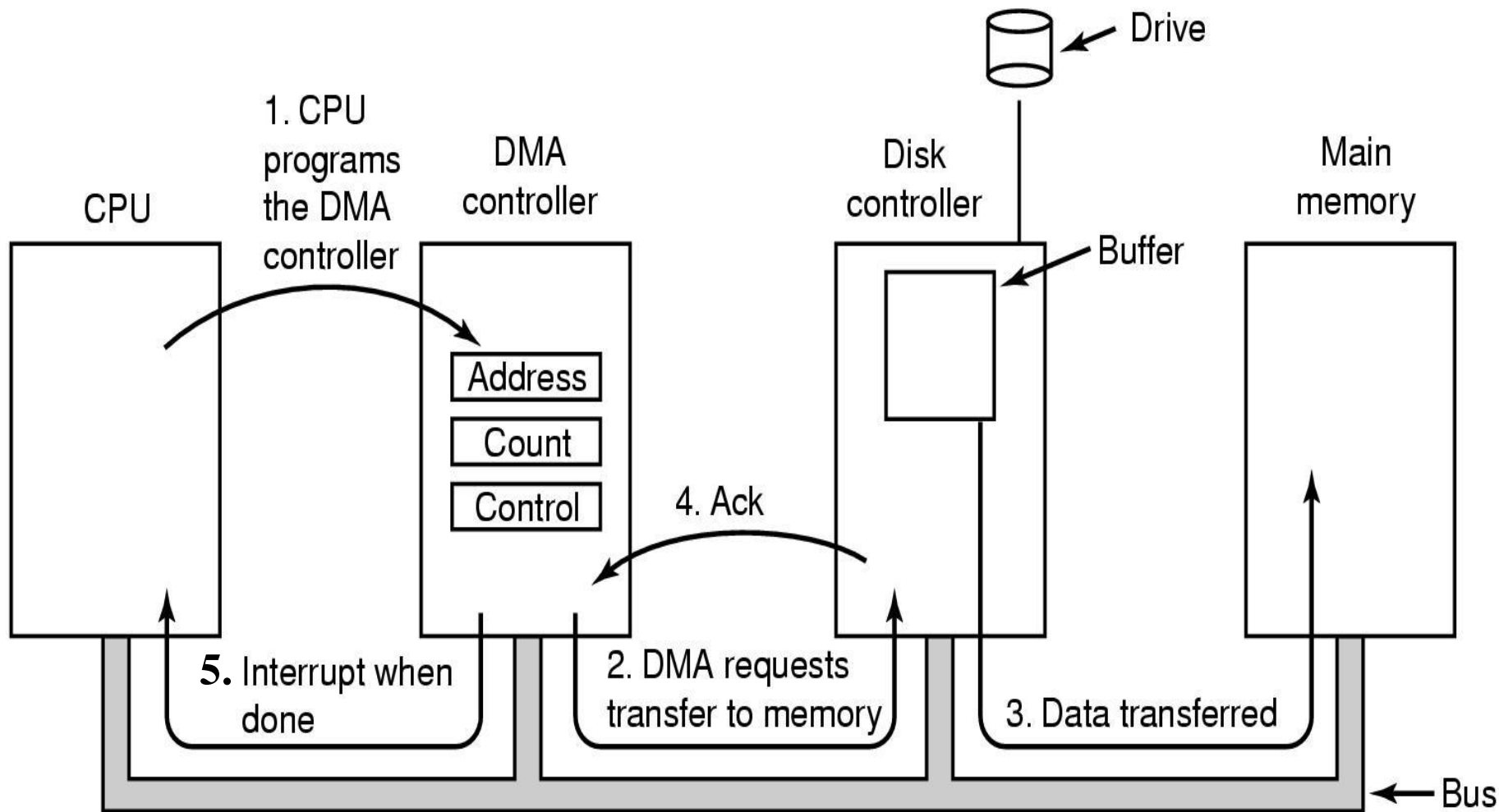
```
if(count == 0)
{
    unblock_user( );
}
else
{
    *printer_data_register = p[i];
    count --; i++;
}
acknowledge_intereupt( );
return_from_interrupt( );
```

中断驱动方式的基本思路是：用户进程通过系统调用函数来发起I/O操作，并在发起后阻塞该进程，调度其他的进程使用CPU。在I/O操作完成时，设备向CPU发出中断，然后在中断处理程序中做进一步的处理。在中断驱动方式下，数据的每次读写还是通过CPU来完成，但是当I/O设备在进行数据处理时，CPU不必等待，可以继续执行其他的进程。

# 直接内存访问方式

- ◆ 要使用直接内存访问（**Direct Memory Access, DMA**）的控制方式，首先在硬件上要有一个**DMA控制器**。该控制器可集成在设备控制器中，也可集成在主板上。
- ◆ **DMA**控制器可以直接去访问系统总线，它能代替CPU去指挥I/O设备与内存之间的数据传送。
- ◆ **DMA**控制器包含了一些寄存器，可被CPU来读或写。包括：一个内存地址寄存器、一个字节计数器，以及一个或多个控制寄存器（指明了I/O设备的端口地址、数据传送方向、传送单位，以及每一次传送的字节数）。

# DMA工作原理



(本图摘自Andrew S. Tanenbaum: “Modern Operating Systems”)

# 本节提要

1

嵌入式系统硬件基础

2

嵌入式系统软件基础

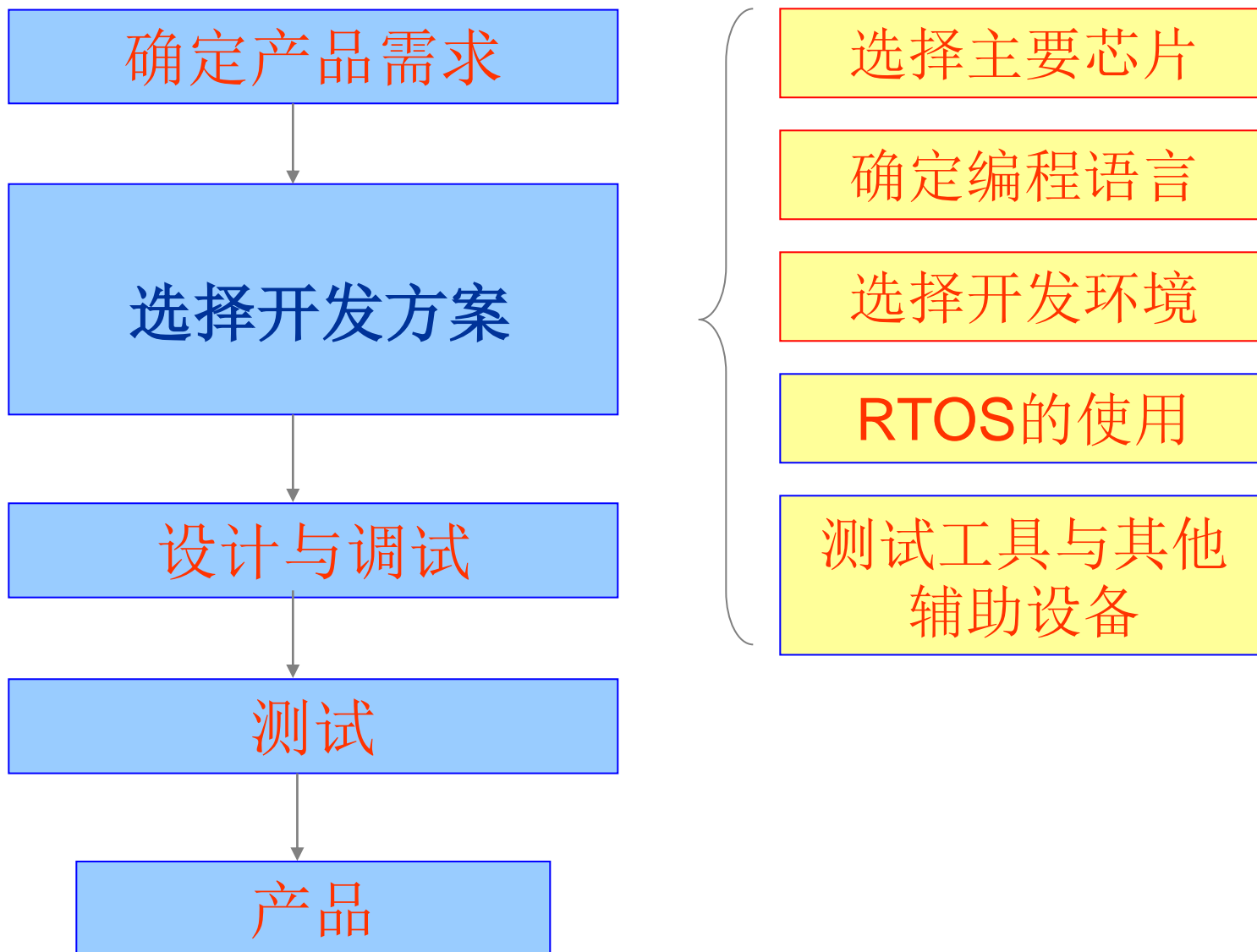
3

嵌入式操作系统

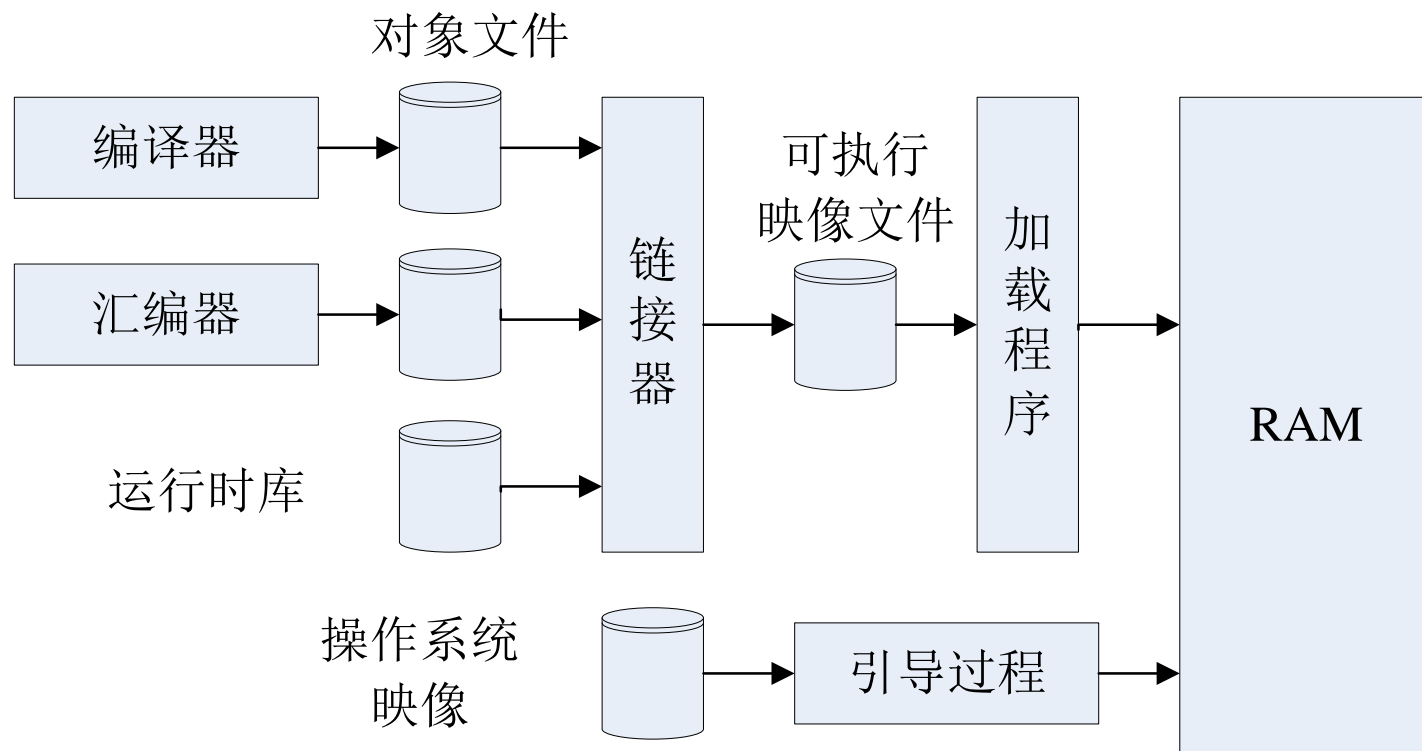
4

嵌入式系统设计方法

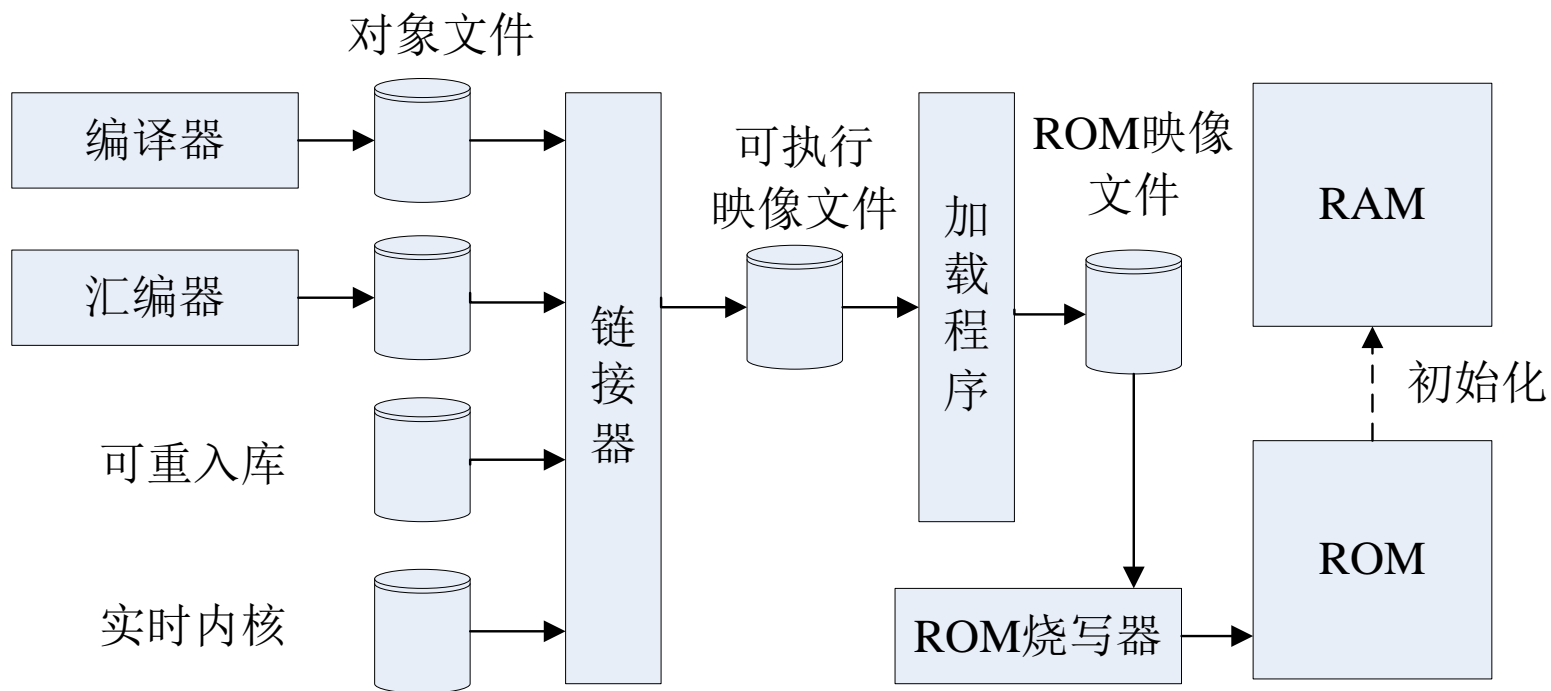
# 嵌入式系统的开发——流程



# 桌面应用程序的编译和加载过程

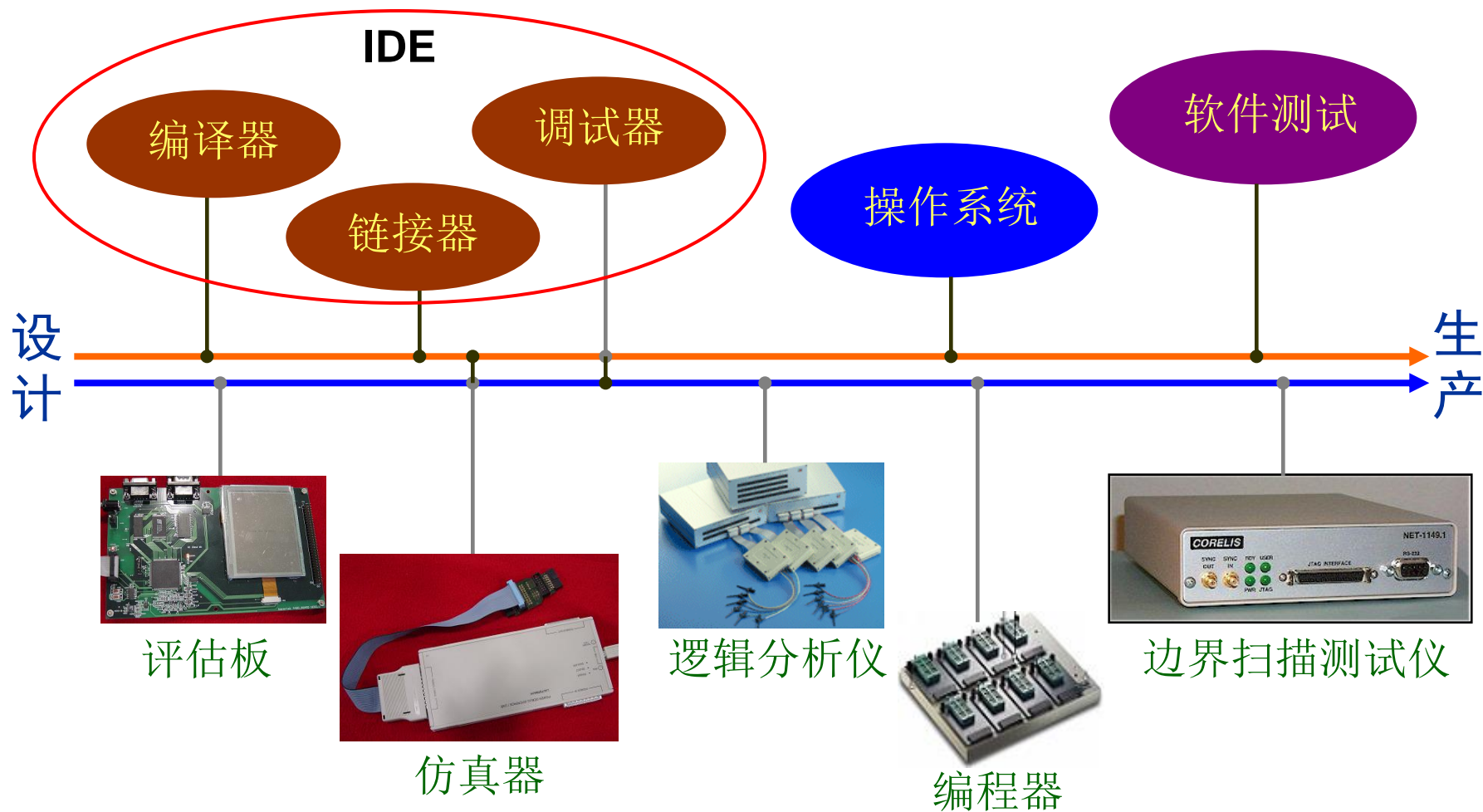


# 嵌入式应用程序的编译和加载过程





# 嵌入式系统的开发——设计与调试



# 开发环境

- 什么是嵌入式开发环境:

编译器/汇编器/链接定位器

调试器/仿真器

主机 (Host) 及其工作平台

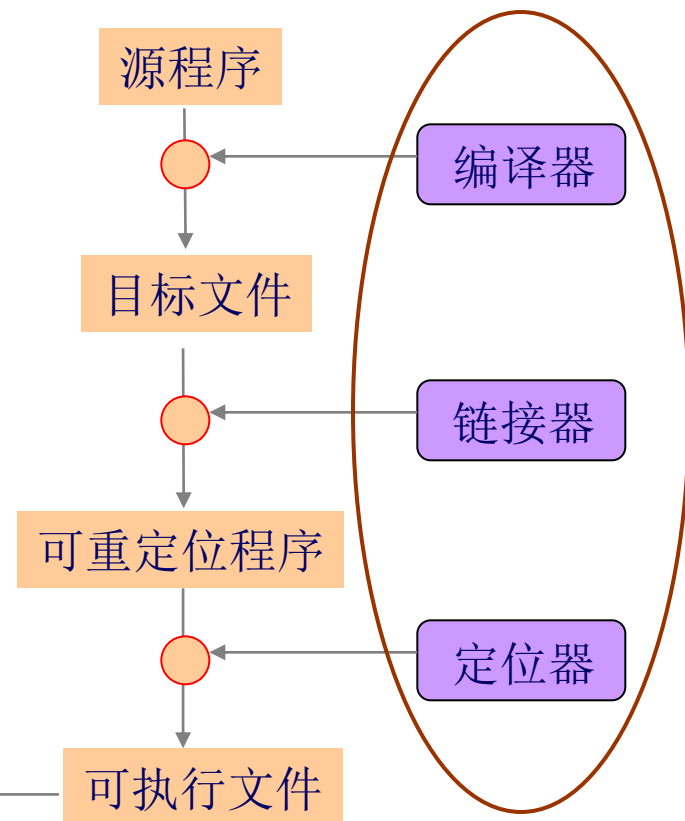
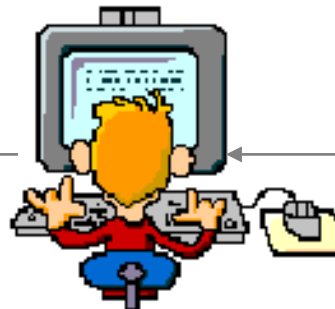
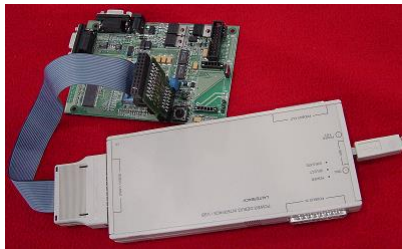
实时操作系统 (可选)

目标评估系统 (可选)

测试工具 (软件/硬件/协议等, 可选)

其他辅助设备 (可选)

## 典型的开发环境



# ARM的编译器（1）

- ADS1.2

- ARM公司出品
- IDE环境，包括
  - ARM/Thumb汇编器： **armasm**
  - ANSI C 编译器 - **armcc** 和 **tcc**
  - ISO / Embedded C++ 编译器 - **armcpp** and **tcpp**
  - 链接器 – **armlink**
  - Windows 集成开发环境 – **CodeWarrior**
  - 格式转换器 – **fromelf**
  - 库管理器 - **armar**
  - 调试器
    - 模拟调试器： **ARMulator**
    - JTAG调试： **AXD**（与**Multi-ICE**配合）
- 支持所有ARM内核，最新版本： **RealView2.0**



# ARM的编译器（2）

- EW-ARM
  - 瑞典IAR公司出品
    - 著名的嵌入式工具提供商，以提供编译器/协议栈/统一建模工具著称
    - 主要产品：Embedded Workbench（EW）、Make APP、Visual State等
  - EW-ARM：针对ARM的集成开发环境：
    - C/C++编译器
    - C-SPY 模拟调试器
    - ROM-Monitor
    - 多种级别代码优化方法，满足用户在速度、文件大小方面的要求
    - 内建ARM特性优化器
    - 支持多种断点模式
    - 支持Nucleus, VxWorks等RTOS
- Greenhills
- GNU

# 嵌入式系统的调试（1）

## 嵌入式系统的调试有四种基本方法：

- 模拟调试 (Simulator)
- 软件调试 (Debugger)
- BDM/JTAG调试 (BDM/JTAG Debugger)
- 全仿真调试 (Emulator)

# 嵌入式系统的调试（2）

- **模拟调试 (Simulator)**

调试工具和待调试的嵌入式软件都在主机上运行，由主机提供一个模拟的目标运行环境，可以进行语法和逻辑上的调试。

- **优点：**简单方便，不需要目标板，成本低
- **缺点：**功能非常有限，无法实时调试

大多数调试工具都提供Simulator功能



# 嵌入式系统的调试（3）

- **软件调试 (Debugger)**

主机和目标板通过某种接口（通常是串口）连接，主机上提供调试界面，待调试软件下载到目标板上运行。

这种方式的先决条件是要在Host和Target之间建立起通信联系（目标板上称为监控程序Monitor）

- **优点：**纯软件，价格较低，简单，软件调试能力较强
- **缺点：**需要事先烧制Monitor（往往需多次试验才能成功）且目标板工作正常，功能有限，特别是硬件调试能力较差。

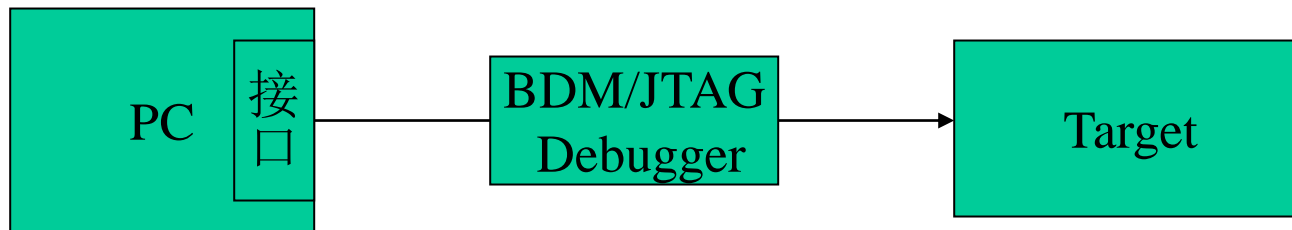


# 嵌入式系统的调试（4）

## ● BDM/JTAG调试

这种方式有一个硬件调试体。该硬件调试体与目标板通过BDM、JTAG等调试接口相连，与主机通过串口、并口、网口或USB口相连。待调试软件通过BDM/JTAG调试器下载到目标板上运行。

- **优点：**方便、简单，无须制作Monitor，软硬件均可调试
- **缺点：**需要目标板，且目标板工作基本正常（至少MCU工作正常），仅适用于有调试接口的芯片





# 嵌入式系统的调试（5）

- 全仿真调试 (Emulator)

这种方式用仿真器完全取代目标板上的MCU，因而目标系统对开发者来说完全是透明的、可控的。仿真器与目标板通过仿真头连接，与主机有串口、并口、网口或USB口等连接方式。由于仿真器自成体系，调试时既可以连接目标板，也可以不连接目标板 (Stand alone) 。

- **优点：**功能非常强大，软硬件均可做到完全实时在线调试
- **缺点：**价格昂贵。

# ARM的调试方式

- 模拟调试
  - SDT2.52: ARMulator
  - ADS1.2: ARMulator
  - Trace32: Simulator
  - EW-ARM: C-spy
- 软件调试
  - ADS1.2: Angel（串口）
  - SDT2.52 Angel（串口）
- JTAG调试
  - ARM: Multi-ICE, 简易型仿真器
  - Trace32-ICD for ARM
  - Hitex: Tanto for ARM
- 全仿真调试
  - Trace32-FIRE/ICE

# ARM调试工具

- Multi-ICE
  - ARM公司出品
  - 与ADS配套使用
  - 支持不同的ARM内核
  - 另有Multi-trace模块可选

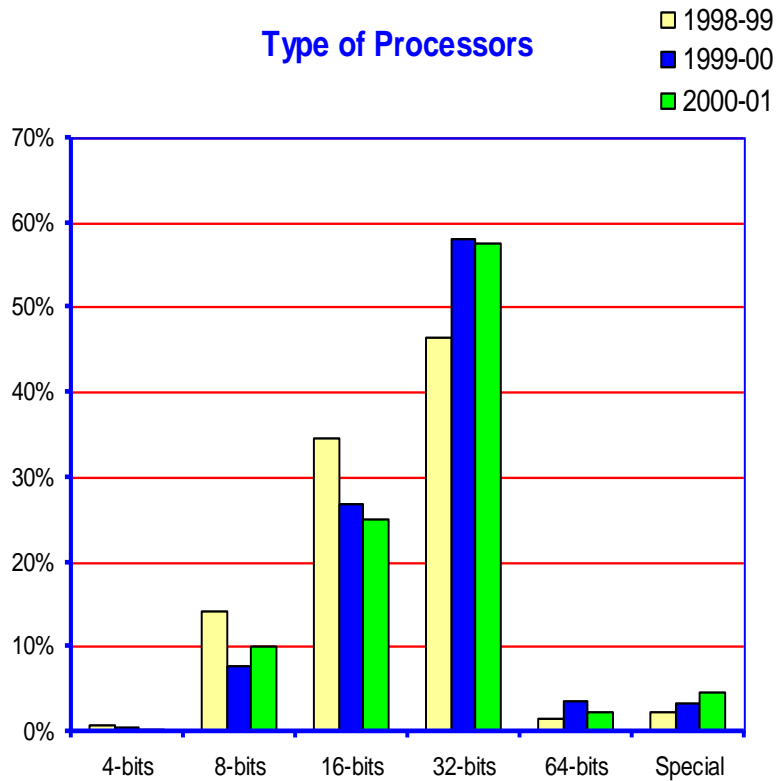


# 选择实时操作系统RTOS

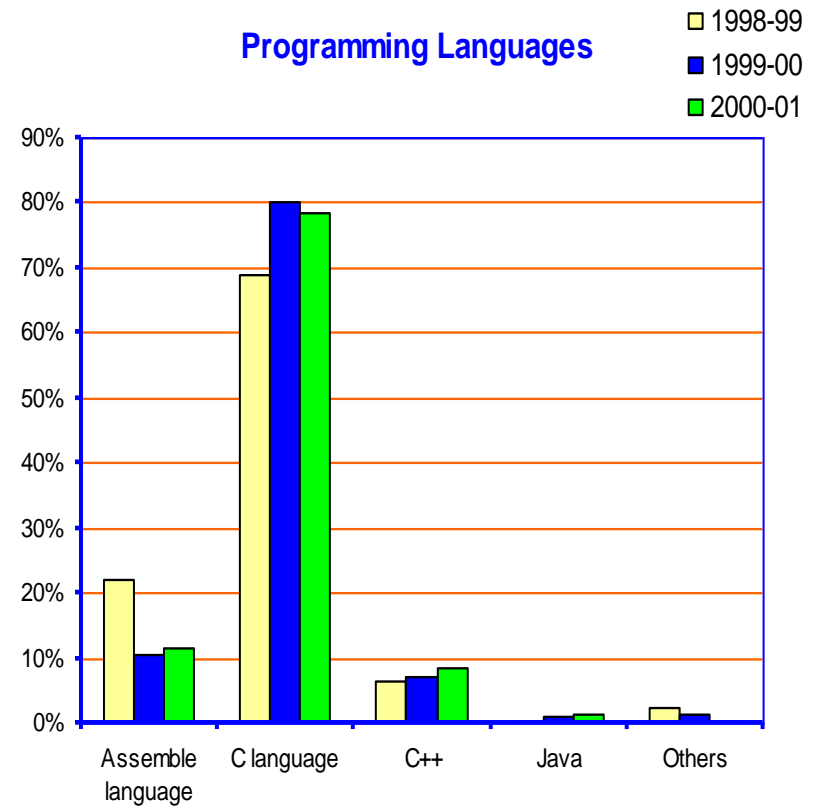
- 对于复杂的嵌入式系统应考虑使用**RTOS**
- **RTOS**的作用：
  - **提供API（应用编程接口）**：操作系统为应用程序员提供可供调用的API，允许程序员致力于应用程序的开发
  - **简化系统设计**：实时嵌入式系统比非实时系统更难设计. 使用实时多任务的内核能简化系统设计，可将复杂的应用程序分为几个不同的任务，由内核去对他们协调处理
- 实验平台如下支持**ARM**的实时操作系统：
  - **uC/OS**
  - **Linux**

# 嵌入式系统编程语言

Type of Processors



Programming Languages



# 嵌入式系统的测试

- 新技术，新方法

- 使用边界扫描测试技术可以有效地解决这些问题！
- 边界扫描来源于IEEE Std 1149.1,是由联合测试行动小组 (Joint Test Action Group)制定的一种测试逻辑，所以又称JTAG标准。
- JTAG作为集成电路的一部分，可以完成以下功能：

*测试器件间的相互连线；*

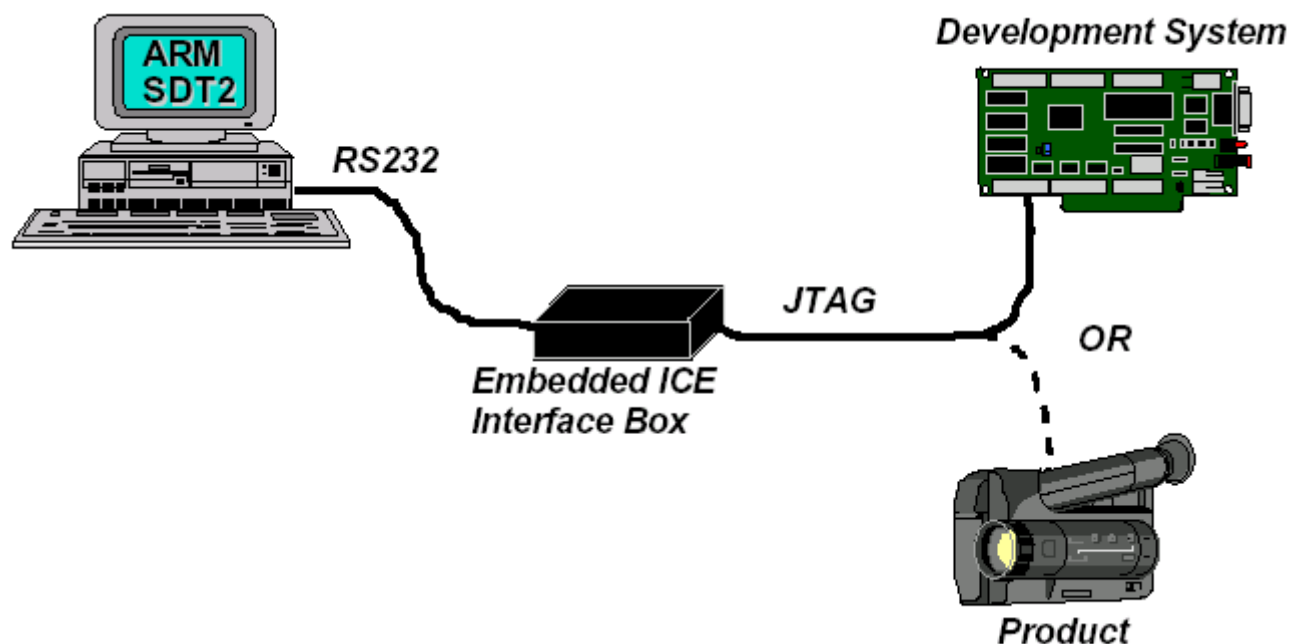
*测试集成电路本身；*

*在线编程CPLD、FPGA、FLASH；*

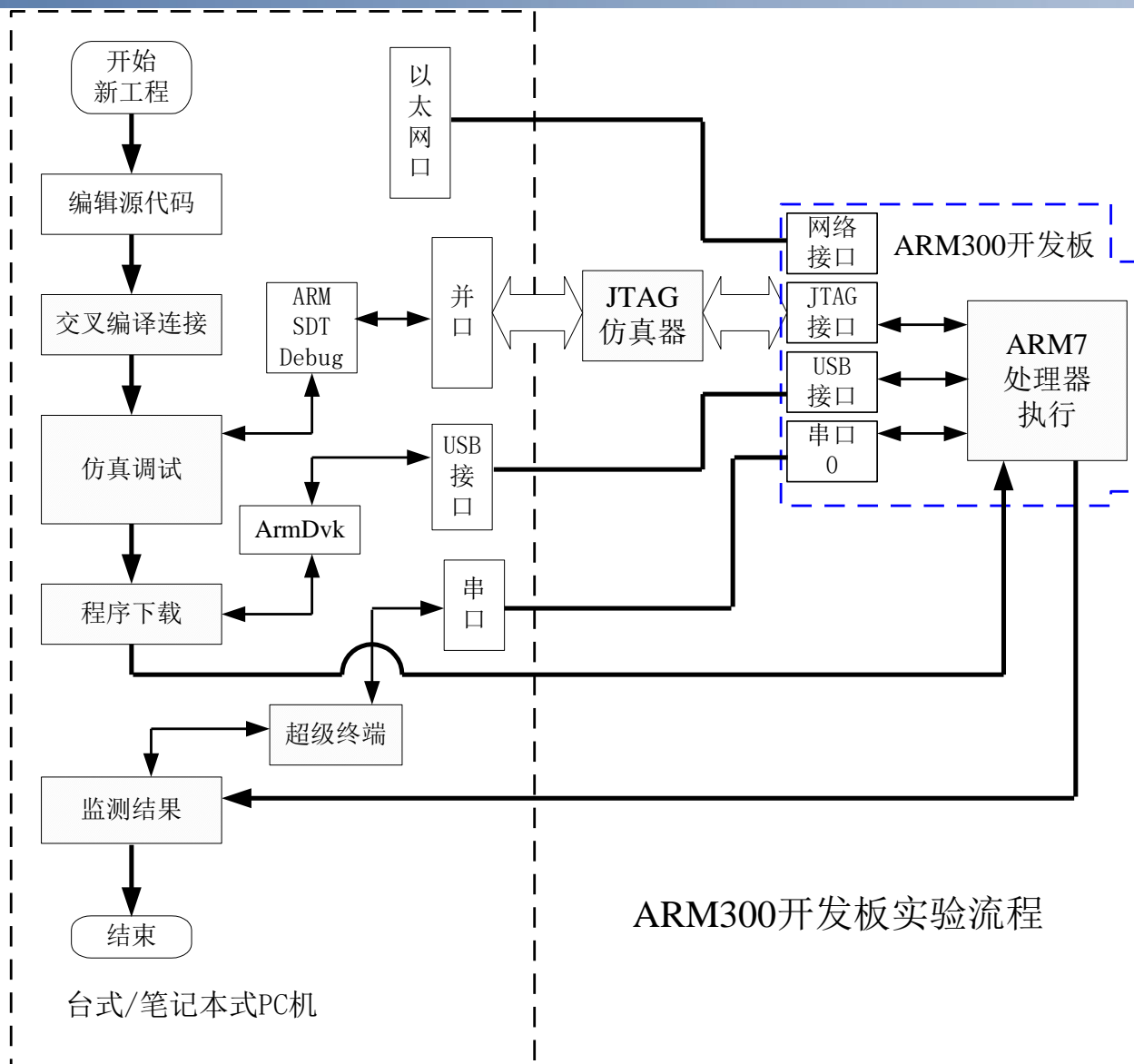
*JTAG仿真调试*



# 嵌入式开发与开发环境



# 嵌入式软件开发流程





谢谢各位