嵌入式系统
An Introduction to Embedded System

第六课 开源嵌入式RTOS内核分析

教师：蔡铭 cm@zju.edu.cn

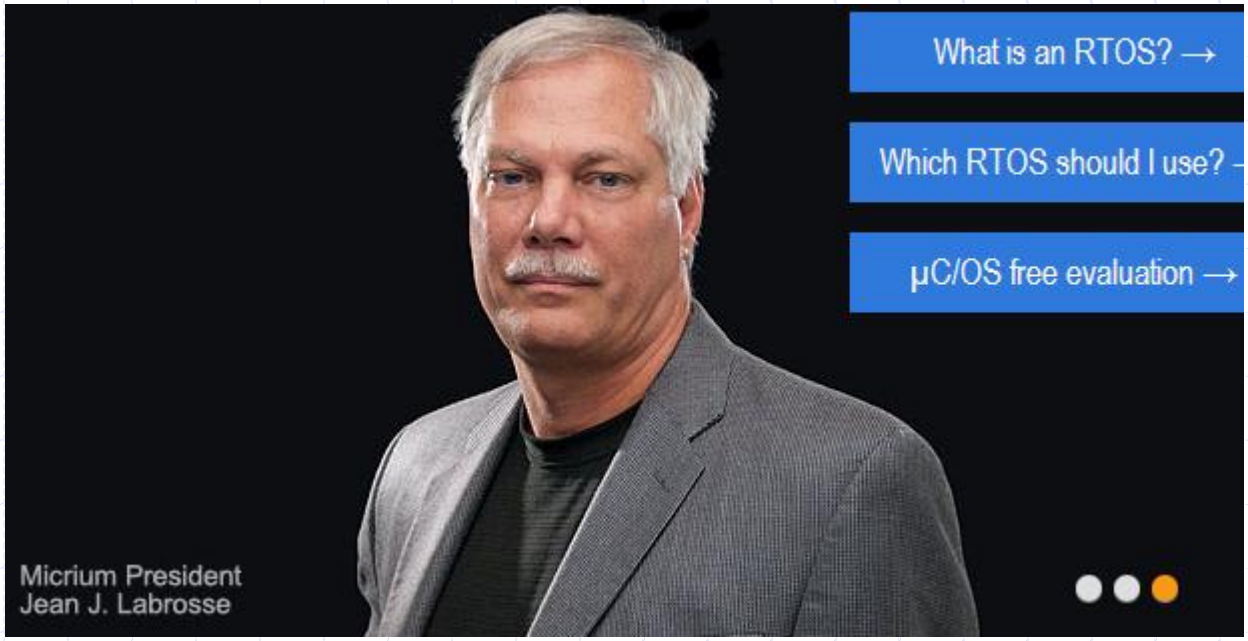
浙江大学计算机学院人工智能研究所
航天科技—浙江大学基础软件研发中心

课程大纲

- μC/OSII操作系统简介
- ARM-Linux操作系统简介
- 操作系统内核移植简介

µC/OS简介（1/2）

- µC/OS全称：micro Control OS，意为“微控制器操作系统”
- µC/OS由嵌入式领域研究人员Jean J.Labrosse于1992年在《嵌入式系统编程》杂志的5月、6月刊上刊登的文章连载，并把µC/OS的源码发布在该杂志的BBS上，被广泛下载和应用。
- Jean J.Labrosse于1999年建立了Micrium公司，提供高质量的嵌入式软件和解决方案，出售µC/OS-II及其他软件的商用许可证。



μC/OS简介（2/2）

- μC/OS是一种基于优先级抢占式、可移植、可裁剪的多任务实时操作系统，其特点为短小、精悍。
- μC/OS绝大部分源码是用ANSI C写的，与硬件相关的那部分汇编代码被压缩至最低限度，使得系统移植性强。
- μC/OS经裁剪最小可达2KB，最小数据RAM需求10KB。
- μC/OS可以在8位～64位，超过40种不同架构的微处理器上运行，在世界范围内得到广泛应用，包括：手机、路由器、集线器、不间断电源、飞行器、医疗设备及工业控制。

μC/OSII代码树结构

与处理器无关的文件

source
|-- OS_CORE.C
|-- OS_DBG_R.C
|-- OS_FLAG.C
|-- OS_MBOX.C
|-- OS_MEM.C
|-- OS_METEX.C
|-- OS_Q.C
|-- OS_SEM.C
|-- OS_TASK.C
|-- OS_TIME.C
|-- uCOS_II.C
`-- uCOS_II.H

与应用相关的文件

编译器\处理器
|-- OS_CFG_R.H
`-- INCLUDES.H

与处理器有关的文件

处理器\编译器
|-- OS_CPU.H
|-- OS_CPU_A.ASM
`-- OS_CPU_C.C

μC/OSII核心代码（X86结构）统计

序号	模块名称		开发语言	代码行数
1	任务管理和调度		ANSI C, x86 asm	983
2	任务间同步与通讯	邮箱	ANSI C	366
		消息队列	ANSI C	519
		互斥信号量	ANSI C	440
		计数信号量	ANSI C	349
		事件标志	ANSI C	700
		公共部分	ANSI C	189
3	内存管理		ANSI C	257
4	时钟和中断		ANSI C	931
总计			4,734	

μC/OSII系统功能

序号	模块名称	功能	支持情况
1	任务管理和调度	任务调度方式	优先级抢占（不支持时间片轮转）
		内核抢占	否
		优先级数量	64
		任务数量	64（其中，8个系统任务）
2	同步和通信	同步	信号量、事件标志组
		通信	消息队列、邮箱
		优先级反转	支持
		有限等待	支持
		递归申请	不支持
3	内存管理	MMU	不支持
		内核数据结构	静态分配
		用户内存使用	固定大小的内存分配
4	中断管理	中断服务程序	不支持ISR自动插桩

μC/OSII内核结构—临界区保护

- 为了处理临界区，代码需要关中断，处理完毕后再开中断。使μC/OSⅡ能够避免同时有其它任务或中断服务进入临界段代码。
- μC/OSⅡ定义两个宏(macros)来关中断和开中断：
  - ✓ OS_ENTER_CRITICAL()
  - ✓ OS_EXIT_CRITICAL()。

OS_CPU.H (x86)

```
#define OS_ENTER_CRITICAL() asm CLI
#define OS_EXIT_CRITICAL()  asm STI
```

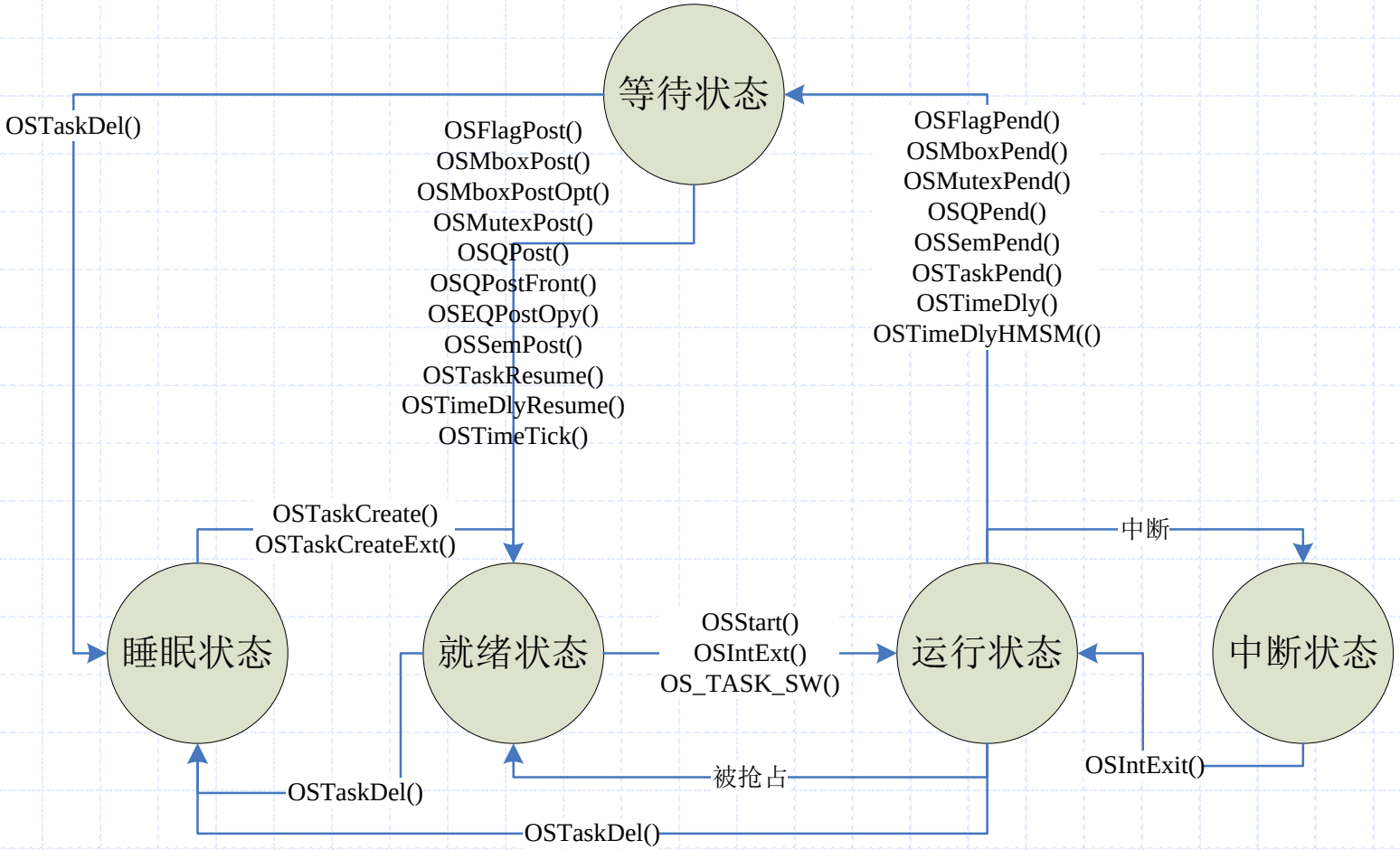
μC/OSII任务函数

- μC/OSII的任务看起来像其它C的函数一样，有函数返回类型，有形式参数变量，但是任务是绝不会返回的，故返回参数必须定义成void。
- 任务完成以后，任务可以自我删除，任务代码并非真的删除了，只是μC/OSⅡ不再调度这个任务。

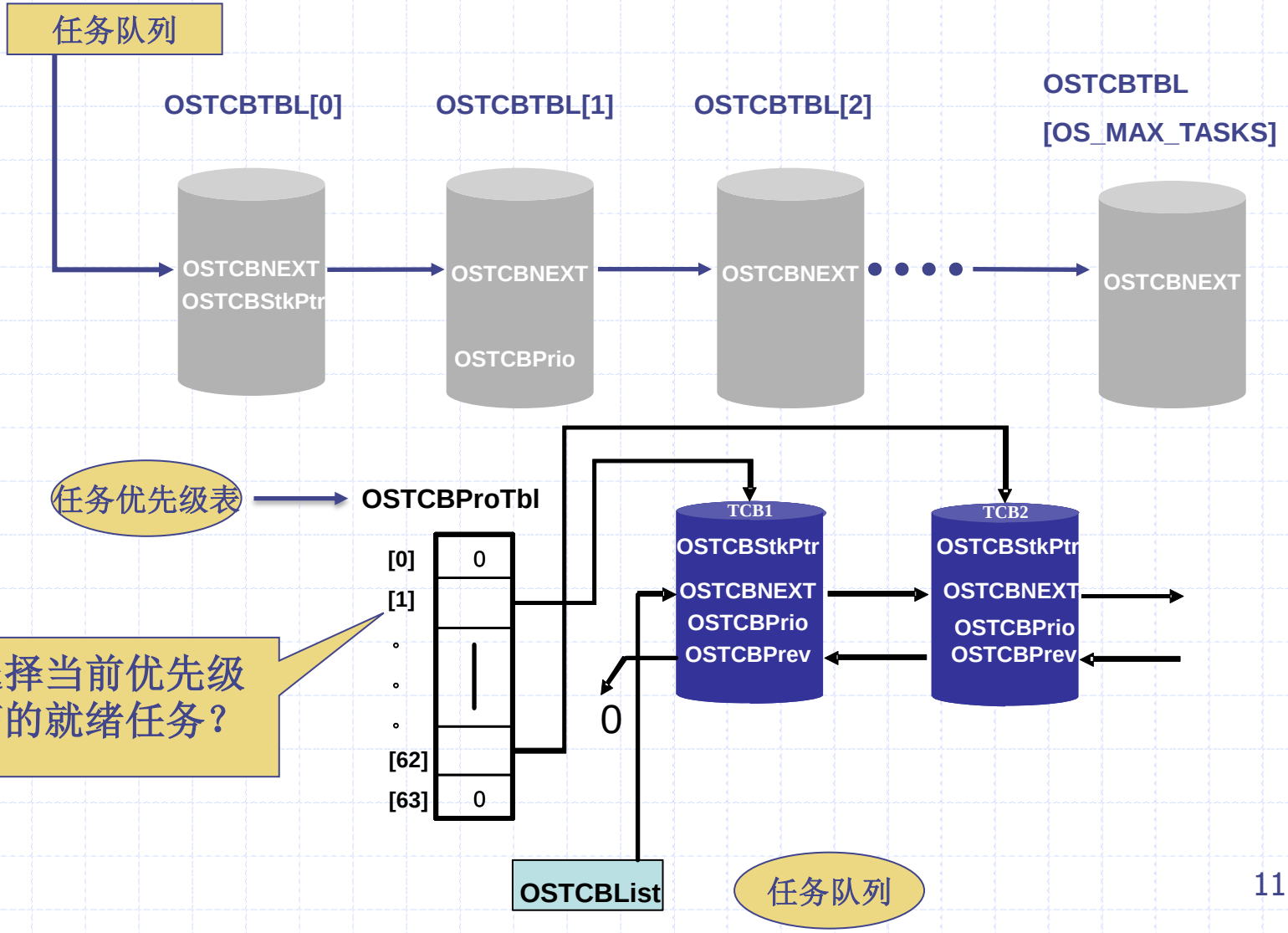
任务代码：任务完成后，可自我删除

```
void YourTask (void *pdata)
{
    /* 用户代码 */
    ……
    OSTaskDel(OS_PRIO_SELF);
}
```

μC/OSII任务状态转换图



μC/OSII—任务控制块（TCB）管理



μC/OSII—独特的就绪表结构（1/2）

- 就绪表（Ready List）
  - ✓两个变量

OSRdyGrp　　OSRdyTbl [ ]

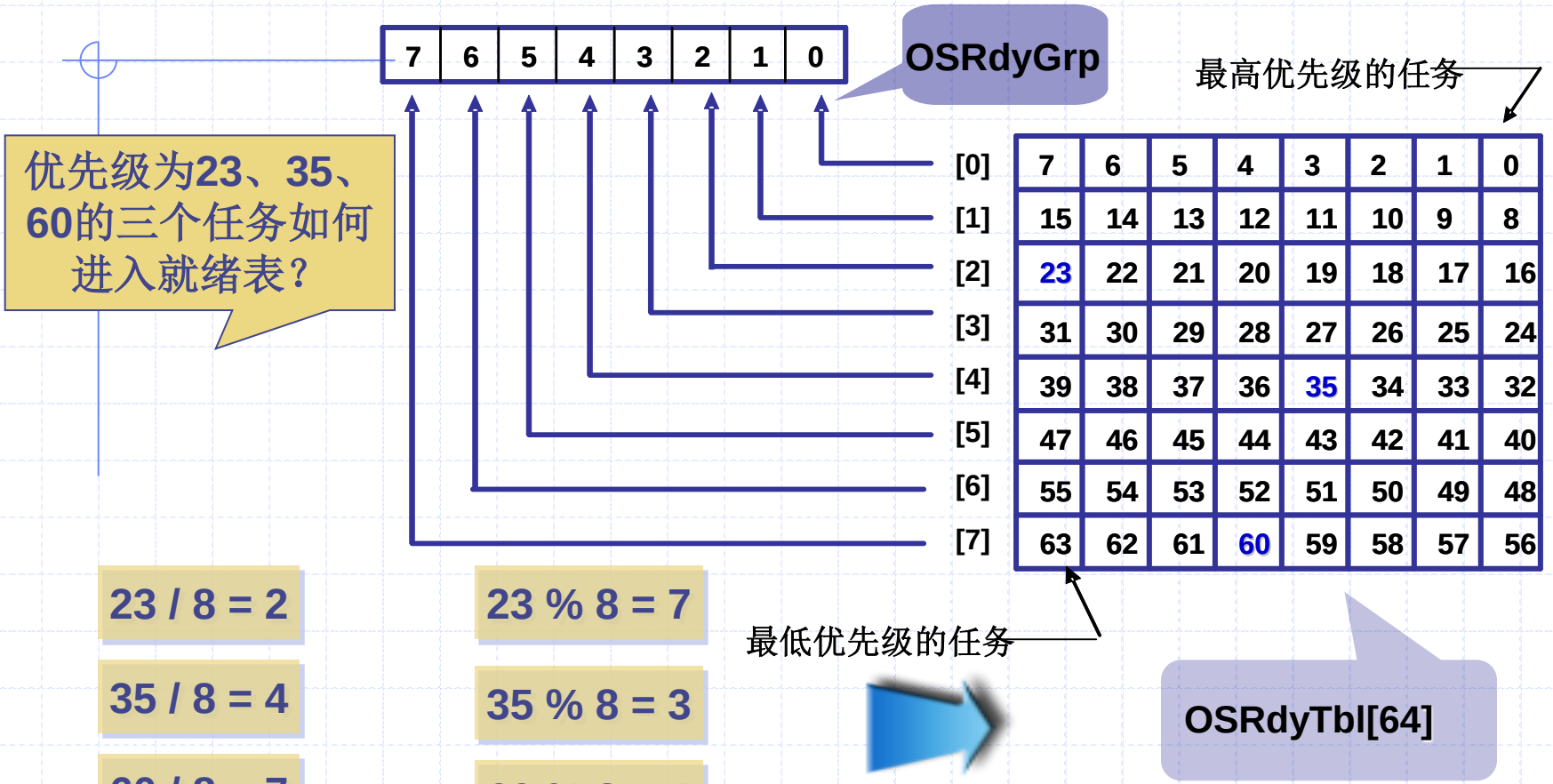
  - ✓两张映射表

OSMapTbl[ ]

OSUnMapTbl[ ]

Index	BitMask（Binary）
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

μC/OSII—独特的就绪表结构（2/2）



μC/OSII—就绪表的操作算法

使任务进入就绪态

```
OSRdyGrp |= OSMapTbl [ prio >> 3 ];
OSRdyTbl [prio >> 3]  |=
                    OSMapTbl [prio & 0x07];
```

从就绪表中删除一个任务

```
if ( ( OSRdyTbl [prio >> 3]  &=
~OSMapTbl [prio & 0x07] ) == 0)
    OSRdyGrp &= ~OSMapTbl [prio >> 3];
```

找出进入就绪态优先级最高的任务

```
y    = OSUnMapTbl [OSRdyGrp];
x    = OSUnMapTbl [ OSRdyTbl [y] ];
prio = ( y << 3 ) + x;
```

μC/OSII－关于优先级表OSUnMapTbl

- OSUnMapTbl（优先级表）

```
INT8U const OSUnMapTbl[256] = {
    0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x00 to 0x0F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x10 to 0x1F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x20 to 0x2F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x30 to 0x3F */
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x40 to 0x4F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x50 to 0x5F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x60 to 0x6F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x70 to 0x7F */
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x80 to 0x8F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x90 to 0x9F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0xA0 to 0xAF */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0xB0 to 0xBF */
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0xC0 to 0xCF */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0xD0 to 0xDF */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0xE0 to 0xEF */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0     /* 0xF0 to 0xFF */
};
```

- 如果OSRdyGrp 的值为二进制01101000(0x68)，查OSUnMapTbl[0x68] ＝ 3，它相应于OSRdyGrp 中的第3位
- 如果OSRdyTbl[3]的值是二进制11100100(0xe4),则OSUnMapTbl[OSRdyTbc[3]] ＝ 2
- 任务的优先级 Prio＝3×8+2＝26

μC/OSII—任务调度器实例

```
void OSSched (void)
{
    INT8U y;

    OS_ENTER_CRITICAL();
    if ((OSLockNesting | OSIntNesting) == 0)
    {
        y = OSUnMapTbl [OSRdyGrp];
        OSPrioHighRdy =  (INT8U)((y << 3) +
                                 OSUnMapTbl [OSRdyTbl [y] ]);
        if (OSPrioHighRdy != OSPrioCur)
        {
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++;
            OS_TASK_SW();
        }
     }
    OS_EXIT_CRITICAL();
}
```

进入临界区

获取最高优先级

任务切换

退出临界区

课程大纲

- μC/OSII操作系统简介
- ARM-Linux操作系统简介
- 操作系统内核移植简介

ARM-Linux内存管理

- 存储管理是一个很大的范畴
- 存储管理机制的实现和具体的CPU以及MMU的结构关系非常紧密
- 操作系统内核的复杂性相当程度上来自内存管理，对整个系统的结构有着根本性的深远影响

内存管理和MMU

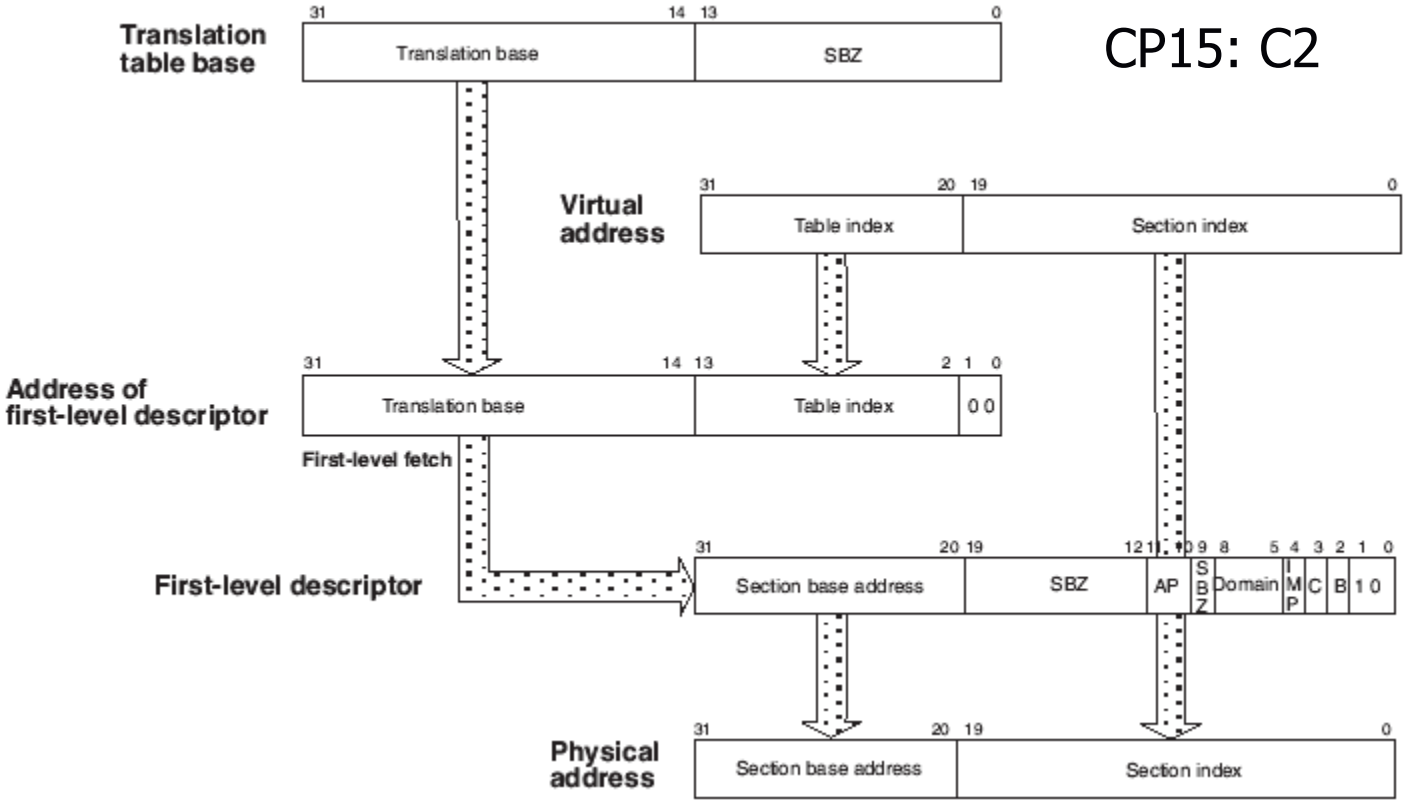
- MMU—“内存管理单元”，其主要作用是两个方面：
  - 地址映射
  - 对地址访问的保护和限制
- MMU可以内置在芯片中，也可以作为协处理器
  - ARM中的协处理器CP15（系统控制协处理器）

ARM存储管理机制

- ARM系统结构中，内存地址映射支持两种：
  - 单层的按“段（section）”映射，段大小为1M
  - 二层的按“页面（page）”映射
    - ✓Tiny pages：1K
    - ✓Small pages：4K
    - ✓Large pages：64K

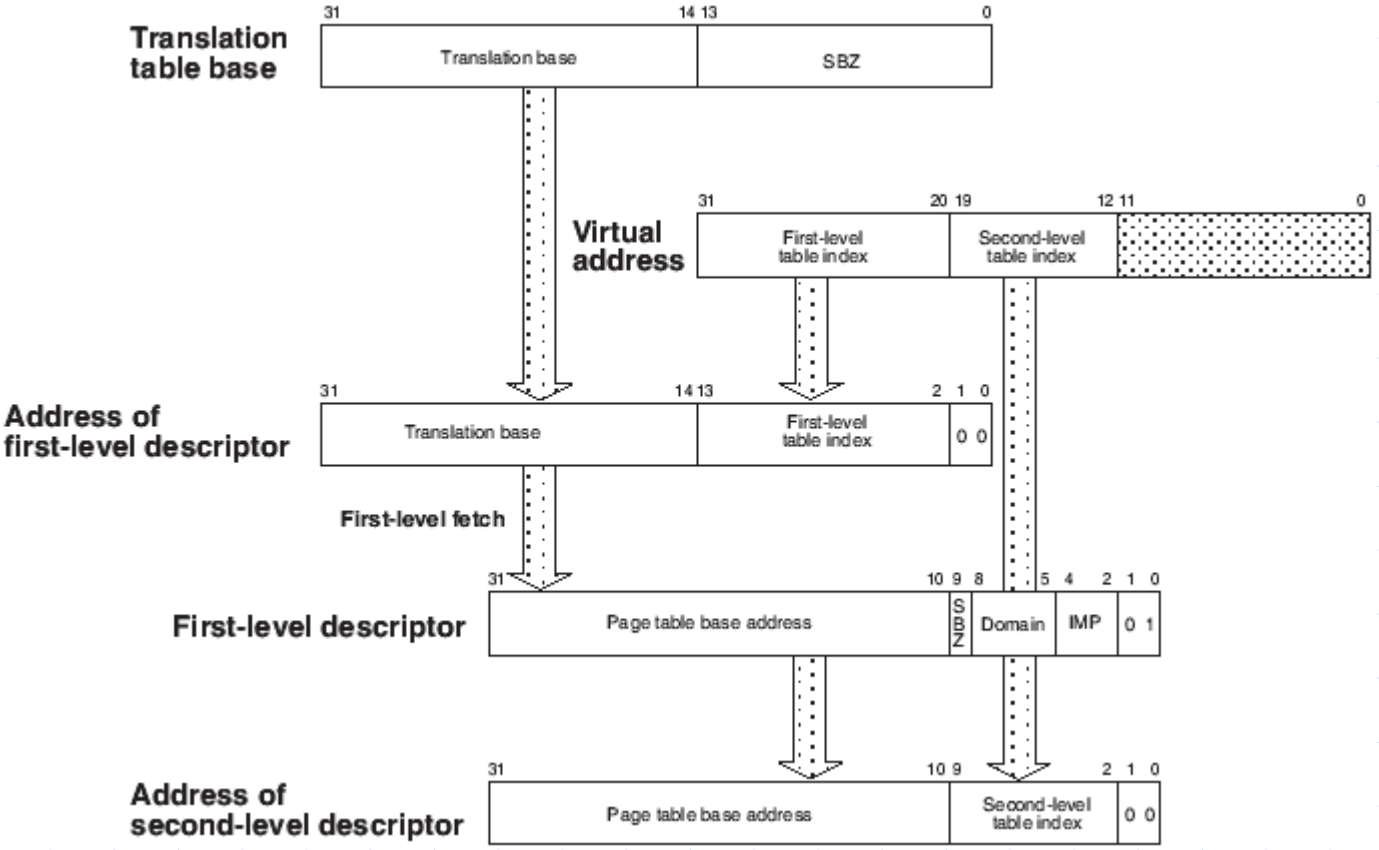
ARM存储管理机制

- 按“段”进行地址映射方式：



ARM存储管理机制

- 采用页面映射，“段映射表”就成了“首层页面映射表”，映射的过程如下(以页面大小＝4KB为例)：



ARM存储管理机制

- 凡是支持虚存的CPU必须为有关的映射表提供高速缓存，使地址映射的过程在不访问内存的前提下完成，用于这个目的高速缓存称为地址转换表（TLB）
- 高速缓存
- ARM系统结构中配备了两个地址映射TLB和两个高速缓存

ARM存储管理机制

- ARM处理器中，MMU是作为协处理器CP15的一部分实现的
- MMU相关的最主要的寄存器有三个：
  - 控制寄存器，控制MMU的开关、高速缓存的开关、写缓冲区的开关等（CP15：C1）
  - 地址转换表基地址寄存器（CP15：C2）
  - 域访问控制寄存器（CP15：C3）

ARM-Linux存储机制的建立

- ARM-Linux内核将4GB虚拟地址空间分为两个部分，高端的1G为系统空间，低端的3G为用户空间。
- ARM将I/O也放在内存地址空间中，所以系统空间的一部分虚拟地址不是映射到物理内存，而是映射到一些I/O设备地址。

ARM-Linux进程的虚存空间

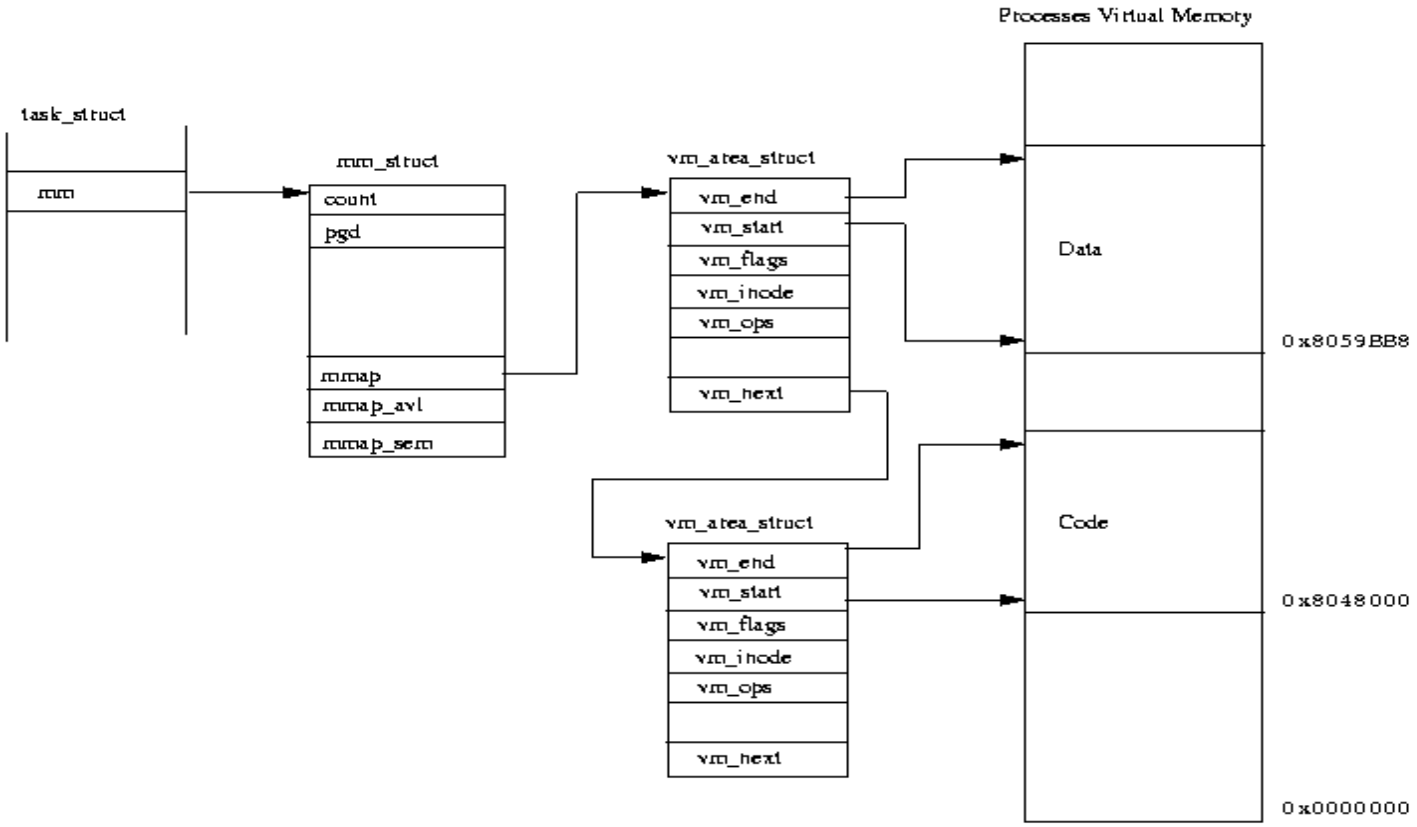
- Linux虚拟内存的实现需要6种机制的支持：
  - 地址映射机制
  - 内存分配回收机制
  - 缓存和刷新机制
  - 请求页机制
  - 交换机制
  - 内存共享机制

ARM-Linux进程的虚存空间

- 系统中的每个进程都各有自己的首层映射表，这就是它的空间，没有独立的空间的就只是线程而不是进程
- Linux内核需要管理所有的虚拟内存地址，每个进程虚拟内存中的内容在其task_struct结构中指向的vm_area_struct结构中描述

ARM-Linux进程的虚存空间

- task_struct结构分析图：



ARM-Linux进程的虚存空间

- 由于那些虚拟内存区域来源各不相同，Linux使用vm_area_struct中指向一组虚拟内存处理过程的指针来抽象此接口
- 为进程创建新的虚拟内存区域或处理页面不在物理内存中的情况下，Linux内核重复使用进程的vm_area_struct数据结构集合
- 当进程请求分配虚拟内存时，Linux并不直接分配物理内存

ARM的中断向量表

地址	异常类型	进入时的模式	进入时I的状态	进入时F的状态
0x0000 0000	复位	管理	禁止	禁止
0x0000 0004	未定义指令	未定义	I	F
0x0000 0008	软件中断(SWI)	管理	禁止	F
0x0000 000C	预取中止(指令)	中止	I	F
0x0000 0010	数据中止	中止	I	F
0x0000 0014	保留	保留	—	—
0x0000 0018	IRQ	中断	禁止	F
0x0000 001C	FIQ	快中断	禁止	禁止

注：表中的I和F表示不对CPSR的该位有影响。

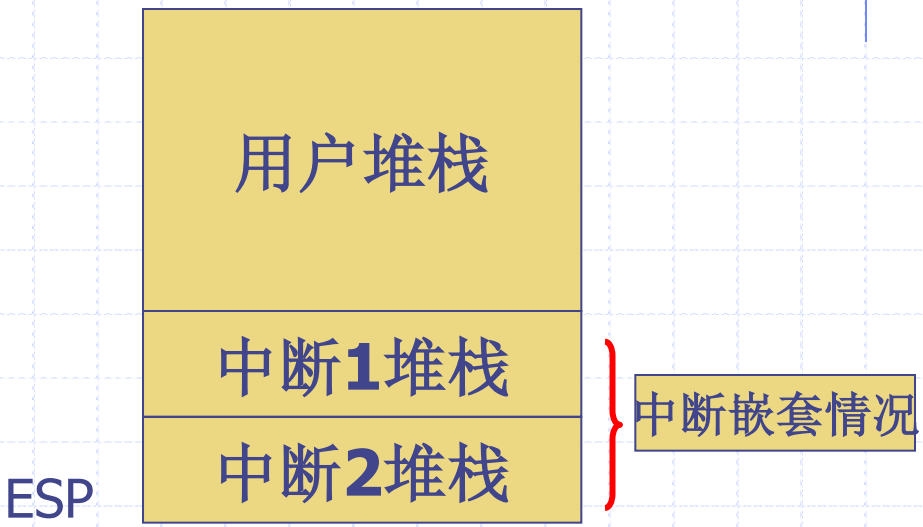
U－boot的中断向量表设置

```
_start:

b       reset
ldr     pc, _undefined_instruction
ldr     pc, _software_interrupt
ldr     pc, _prefetch_abort
ldr     pc, _data_abort
ldr     pc, _not_used
ldr     pc, _irq
ldr     pc, _fiq
```

ARM的中断堆栈设置

ARM的独立中断堆栈模式



X86任务、中断共享堆栈模式

ARM-Linux的外部中断响应和处理

- ARM是将中断控制器集成在CPU内部的，由外设产生的中断请求都由芯片上的中断控制器汇总成一个IRQ中断请求
- 中断控制器向CPU提供一个中断请求寄存器和一个中断控制寄存器
- GPIO是一个通用的可编程的I/O接口，其接口寄存器中的每一位都可以分别在程序的控制下设置用于输入或者输出

ARM-Linux的中断响应和处理

- 在Linux中，每一个中断控制器都由struct hw_interrut_type数据结构表示：

```
struct hw_interrupt_type {
const char * typename;
unsigned int (*startup)(unsigned int irq);
void (*shutdown)(unsigned int irq);
void (*enable)(unsigned int irq);
void (*ack)(unsigned int irq);
void (*end)(unsigned int irq);
void (*set_affinity)(unsiged int irq,unsigned long mask);
};
```

ARM-Linux的中断响应和处理

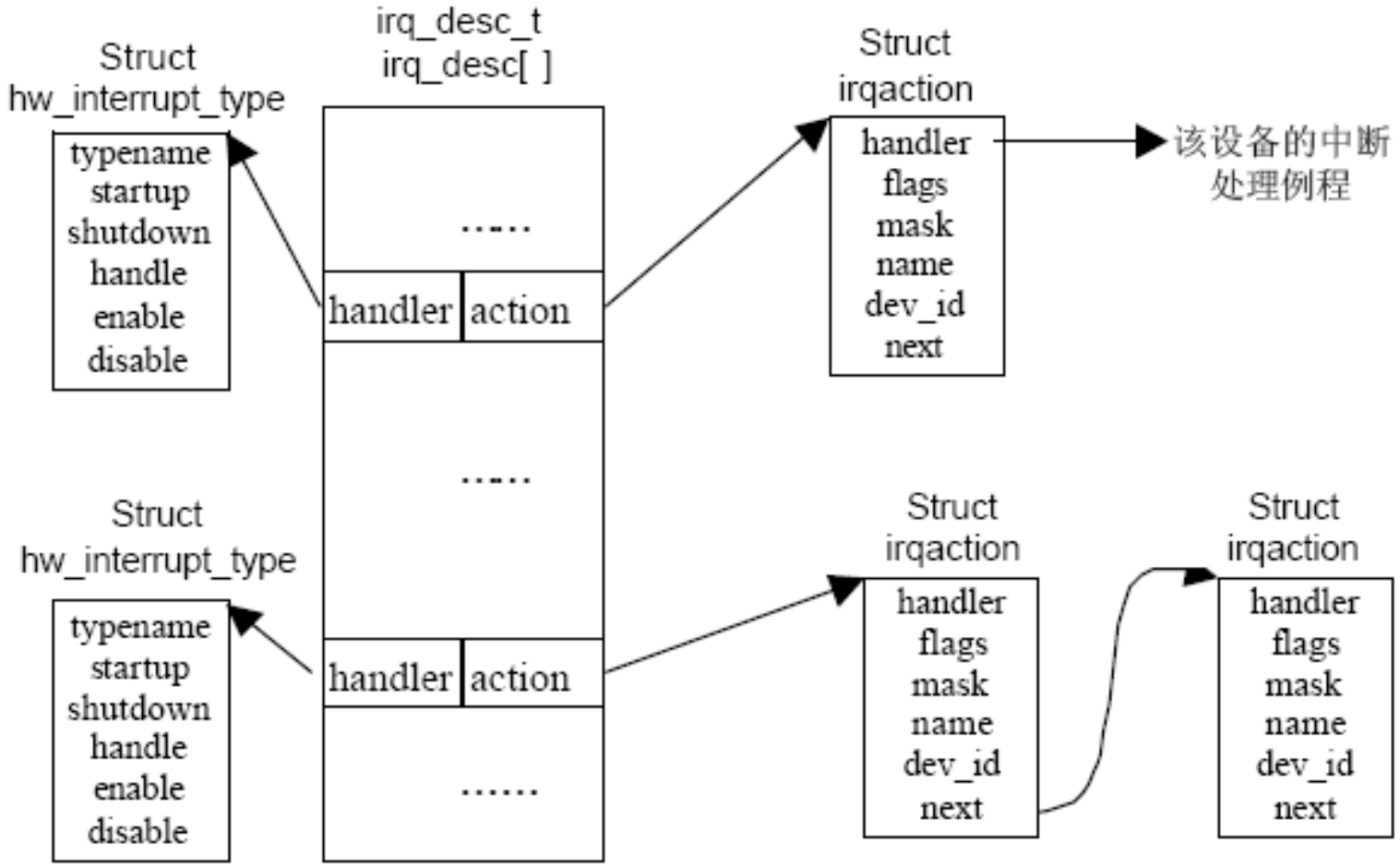
- 每一个中断请求线都有一个struct irqdesc 数据结构表示：

```
typedef struct {
unsigned int status;   /* IRQ status */
hw_irq_controller *handler;
struct irqaction *action;  /*IRQ action list */
unsigned int depth;    /* nested irq disables */
spinlock_t lock;
}_cacheline_aligned irq_desc_t;
```

- 具体中断处理程序则在数据结构 struct irqaction

ARM-Linux的中断响应和处理

- 三个数据结构的相互关系如图：



ARM-Linux的中断响应和处理

- ARM—Linux的中断初始化
  - 执行函数trap_init（）
    - 安装中断向量表到虚拟地址0处：

```
.LCvectors:   swi     SYS_ERROR0
              b       __real_stubs_start + (vector_undefinstr - __stubs_start)
              ldr     pc, __real_stubs_start + (.LCvswi - __stubs_start)
              b       __real_stubs_start + (vector_prefetch - __stubs_start)
              b       __real_stubs_start + (vector_data - __stubs_start)
              b       __real_stubs_start + (vector_addrexcptn - __stubs_start)
              b       __real_stubs_start + (vector_IRQ - __stubs_start)
              b       __real_stubs_start + (vector_FIQ - __stubs_start)
```

ARM-Linux的中断响应和处理

- 接着搬运中断响应程序代码到0x200处：

```
__stubs_start:
vector_IRQ:
          ...
vector_data:
...
vector_prefetch:
          ...
vector_undefinestr:
          ...
vector_FIQ:
          ...
vector_addrexcptn:
          ...
.LCvswi: .word    vector_swi
.LCsirq: .word    __temp_irq
.LCsund: .word    __temp_und
.LCsabt: .word    __temp_abt
__stubs_end:
```

ARM-Linux的中断响应和处理

- trap_init()函数执行完了以后，再执行init_IRQ()。
- 通过函数init_IRQ()建立上述提及的3个数据结构及其相互联系的框架。

ARM-Linux系统调用

- X86有INT 0x80
- ARM处理器有自陷指令SWI
- CPU遇到自陷指令后，跳转到内核态
- 操作系统首先保存当前运行信息，然后根据系统调用号查找相应的函数去执行
- 执行完了以后恢复原先保存的运行信息返回

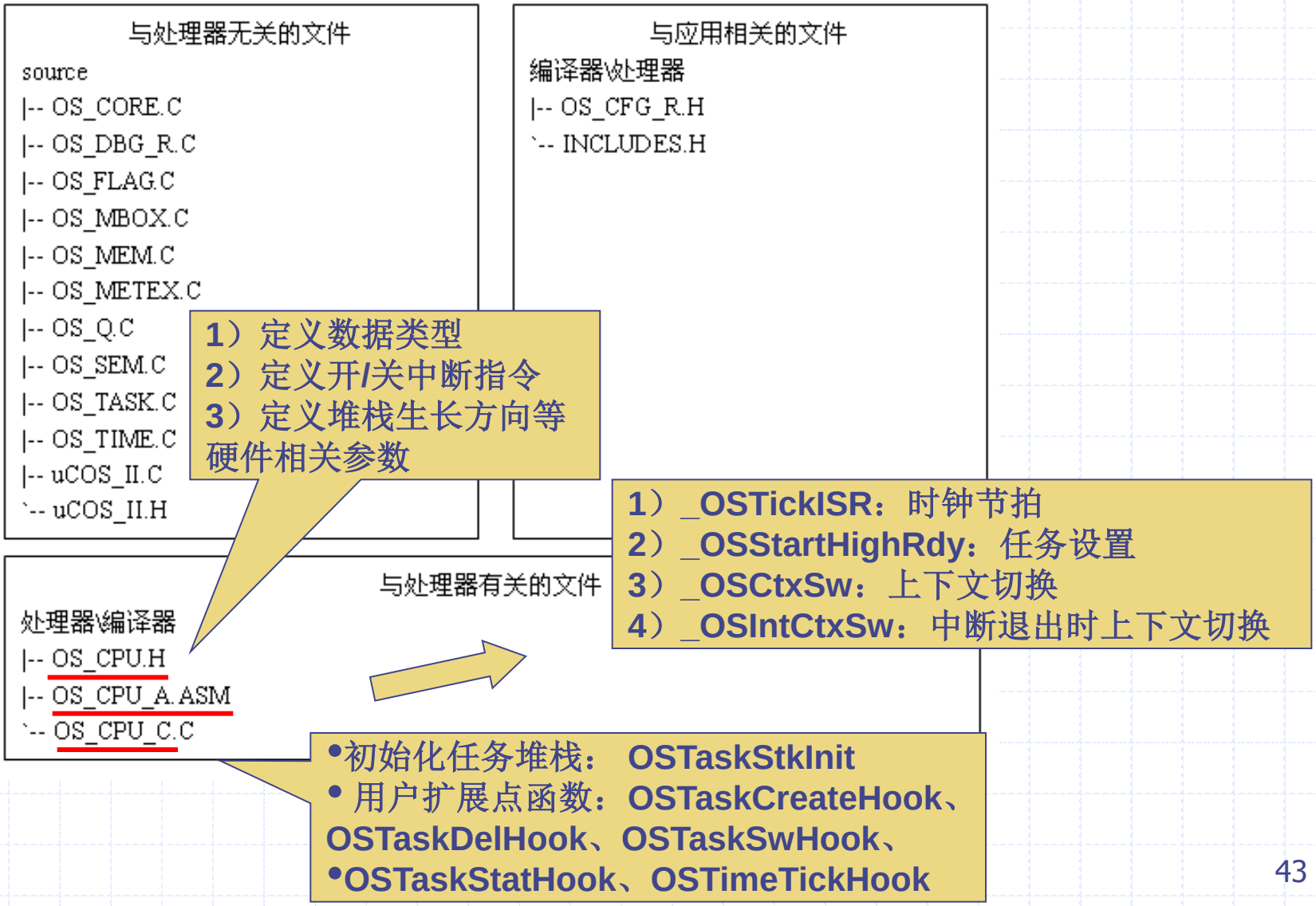
课程大纲

- μC/OSII操作系统简介
- ARM-Linux操作系统简介
- 操作系统内核移植简介

μC/OSII操作系统内核移植步骤(1/2)

- 要使μC/OSⅡ正常运行，必须满足以下要求
  - C 编译器能产生可重入代码
  - 处理器支持中断，并且能产生定时中断，通常在10～100Hz
  - 提供打开和关闭中断的指令
  - 处理器支持能够容纳一定量数据的堆栈
  - 处理器有将堆栈指针，以及寄存器读出、存储到堆栈，或内存中的指令

μC/OSII操作系统内核移植步骤(2/2)

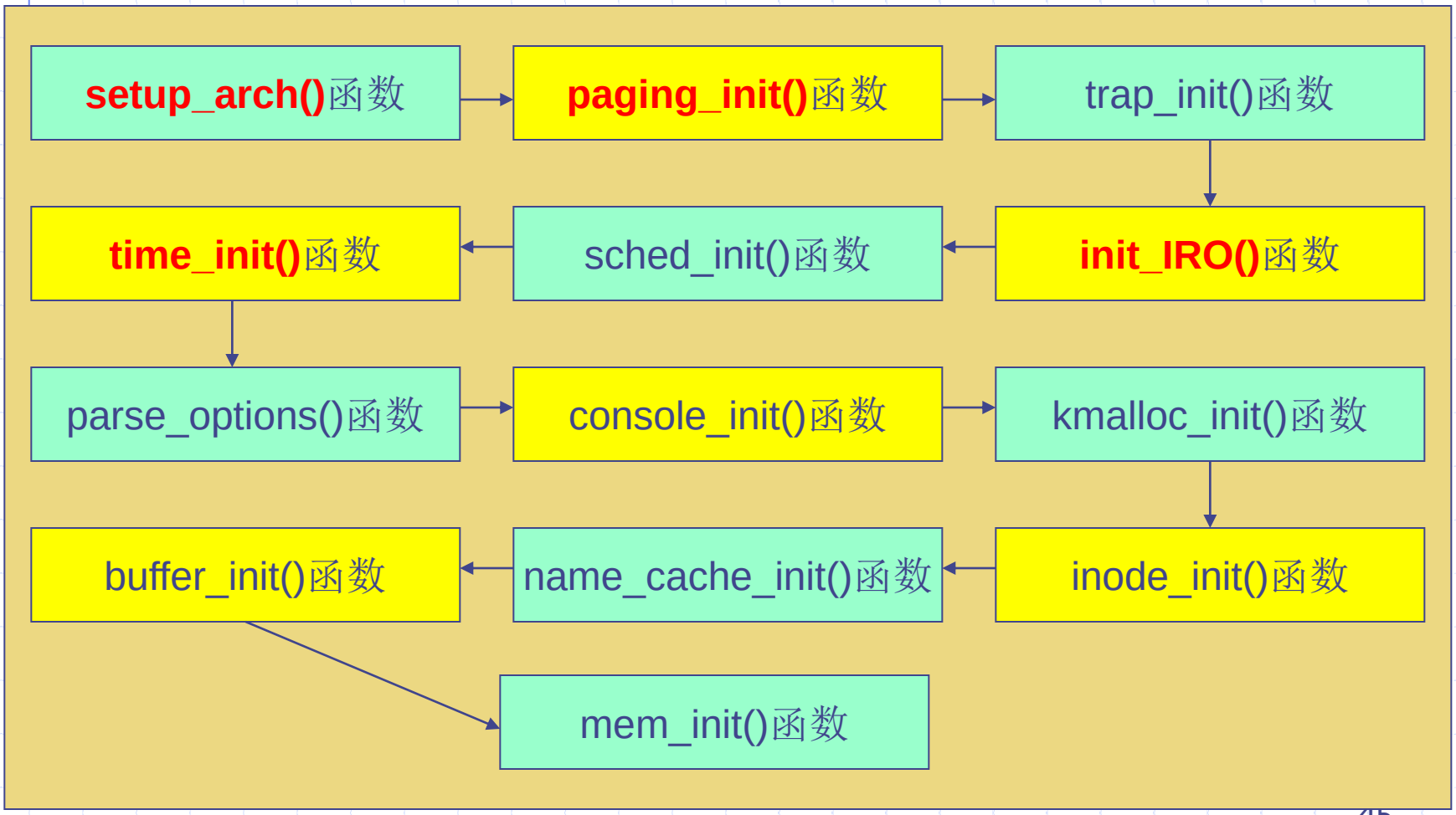


Linux操作系统内核移植的主要步骤(1/3)

- 开发环境选择
  - 交叉编译器、交叉链接器选择
  - 调试器选择（模拟器）
  - Makefile修改
- 机型号、机型描述信息增加
- 启动代码添加（arch/kernel/head_***.S）

Linux操作系统内核移植的主要步骤(2/3)

- Start_kernel()的初始化步骤



Linux操作系统内核移植的主要步骤(3/3)

- 修改与体系结构环境设置函数（setup_arch）
- 修改页表结构初始化函数（paging_init）
- 修改中断初始化函数（init_IRQ）
- 修改基本驱动程序（时钟、定时器、串口通讯）
- 内核编译、生成

谢 谢！