浙江大学
ZHEJIANG UNIVERSITY

嵌入式系统

第8课 嵌入式Linux内核设计

王总辉
zhwang@zju.edu.cn

http://course.zju.edu.cn

提纲

- ARM-Linux内核简介
- ARM-Linux内存管理
- ARM-Linux进程管理和调度
- ARM-Linux模块机制
- ARM-Linux中断管理
- ARM-Linux系统调用
- ARM-Linux系统启动和初始化

提纲

- ARM-Linux内核简介
- ARM-Linux内存管理
- ARM-Linux进程管理和调度
- ARM-Linux模块机制
- ARM-Linux中断管理
- ARM-Linux系统调用
- ARM-Linux系统启动和初始化

ARM-Linux内核简介

- Linux是可移植的操作系统。
- 被成功移植到多个处理器架构下，arch目录下即为Linux支持的体系结构
- ARM-Linux是基于ARM处理器计算机的Linux内核。

用户

应用程序（sh、vi、OpenOffice、KDE、GNOME等）

应用程序库（libgcc、glibc、glibc-devel、glibc-common等）

用户级

C函数库（open、read、socket、fork、exec等）

内核级

中断控制（系统调用接口）

进程管理

进程通信

虚拟文件系统

磁盘高速缓存

网际协议栈

字符设备驱动

进程调度

存储管理

块设备驱动

网络设备驱动

硬件级

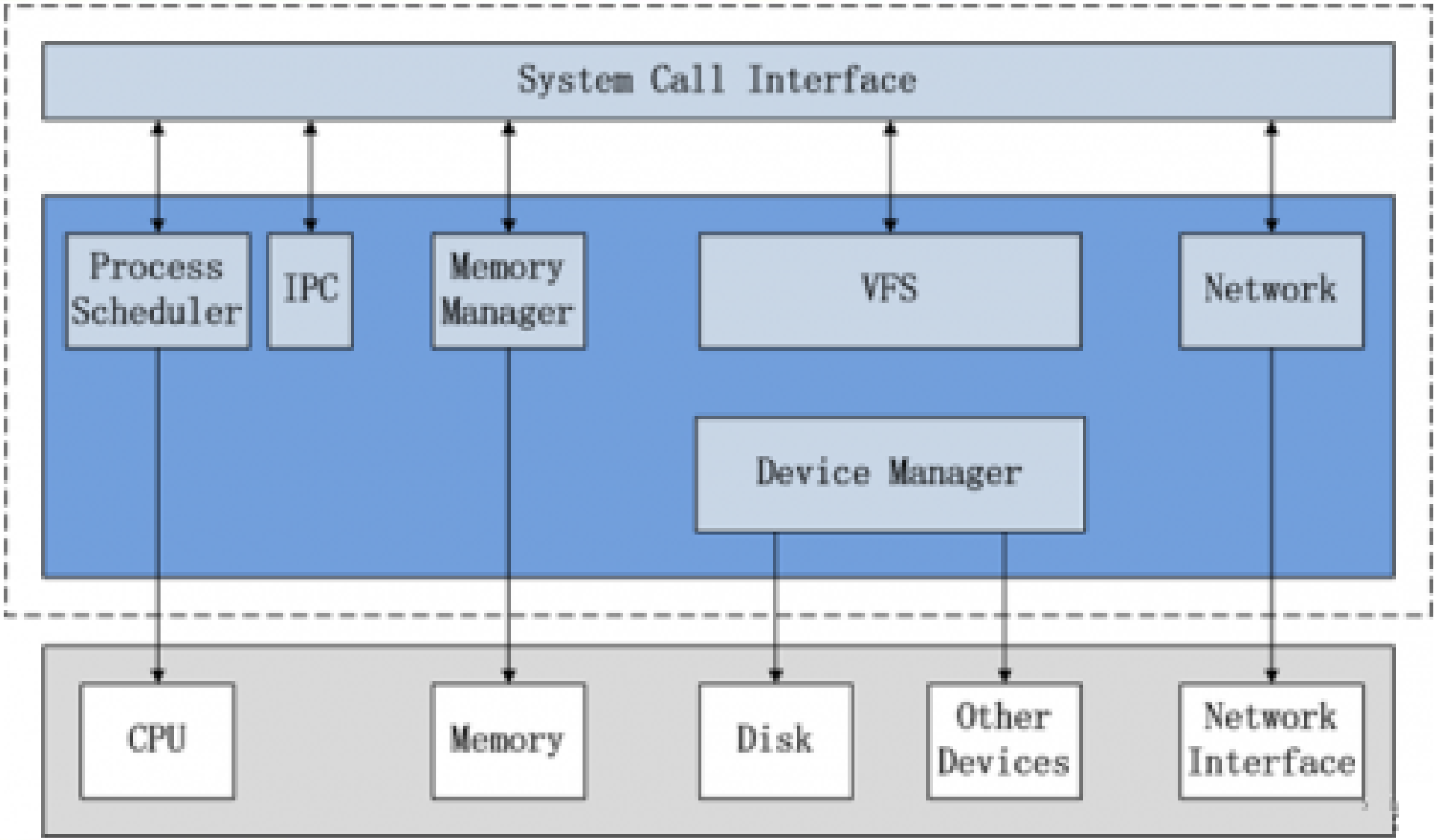
CPU

内存

磁盘

键盘/终端

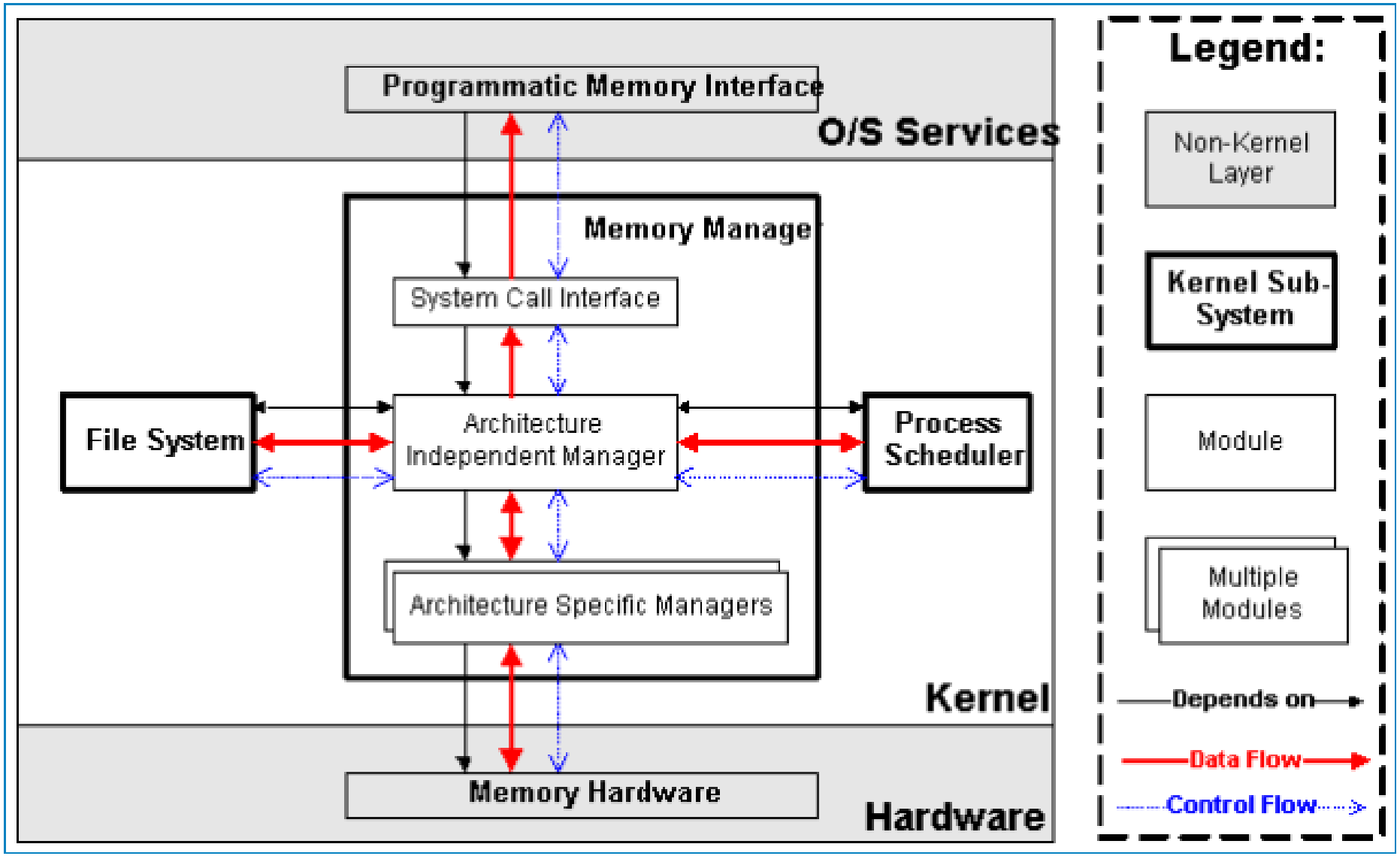
网卡


System Call Interface
Process Scheduler
IPC
Memory Manager
VFS
Network
Device Manager
CPU
Memory
Disk
Other Devices
Network Interface

提纲

- ARM-Linux内核简介
- ARM-Linux内存管理
- ARM-Linux进程管理和调度
- ARM-Linux模块机制
- ARM-Linux中断管理
- ARM-Linux系统调用
- ARM-Linux系统启动和初始化

Linux内存管理


Programmatic Memory Interface
O/S Services
Memory Manager
System Call Interface
File System
Architecture Independent Manager
Process Scheduler
Architecture Specific Managers
Kernel
Memory Hardware
Hardware
Legend:
Non-Kernel Layer
Kernel Sub-System
Module
Multiple Modules
Depends on
Data Flow
Control Flow

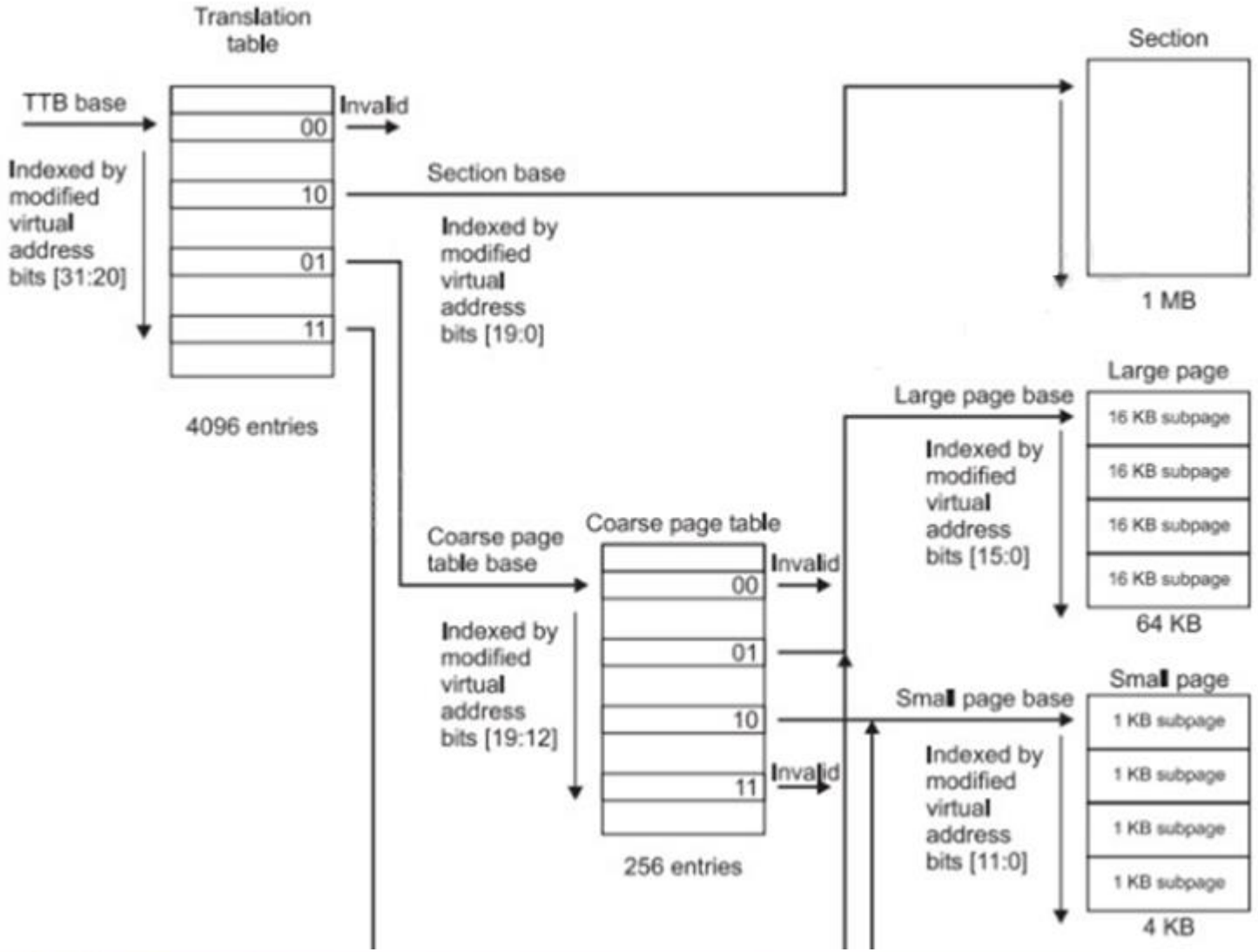
ARM-Linux内存管理（1）

- ARM体系结构MMU
  - ARM M/R系列不支持MMU
  - ARM A系列支持MMU
- Linux内核标准内存管理

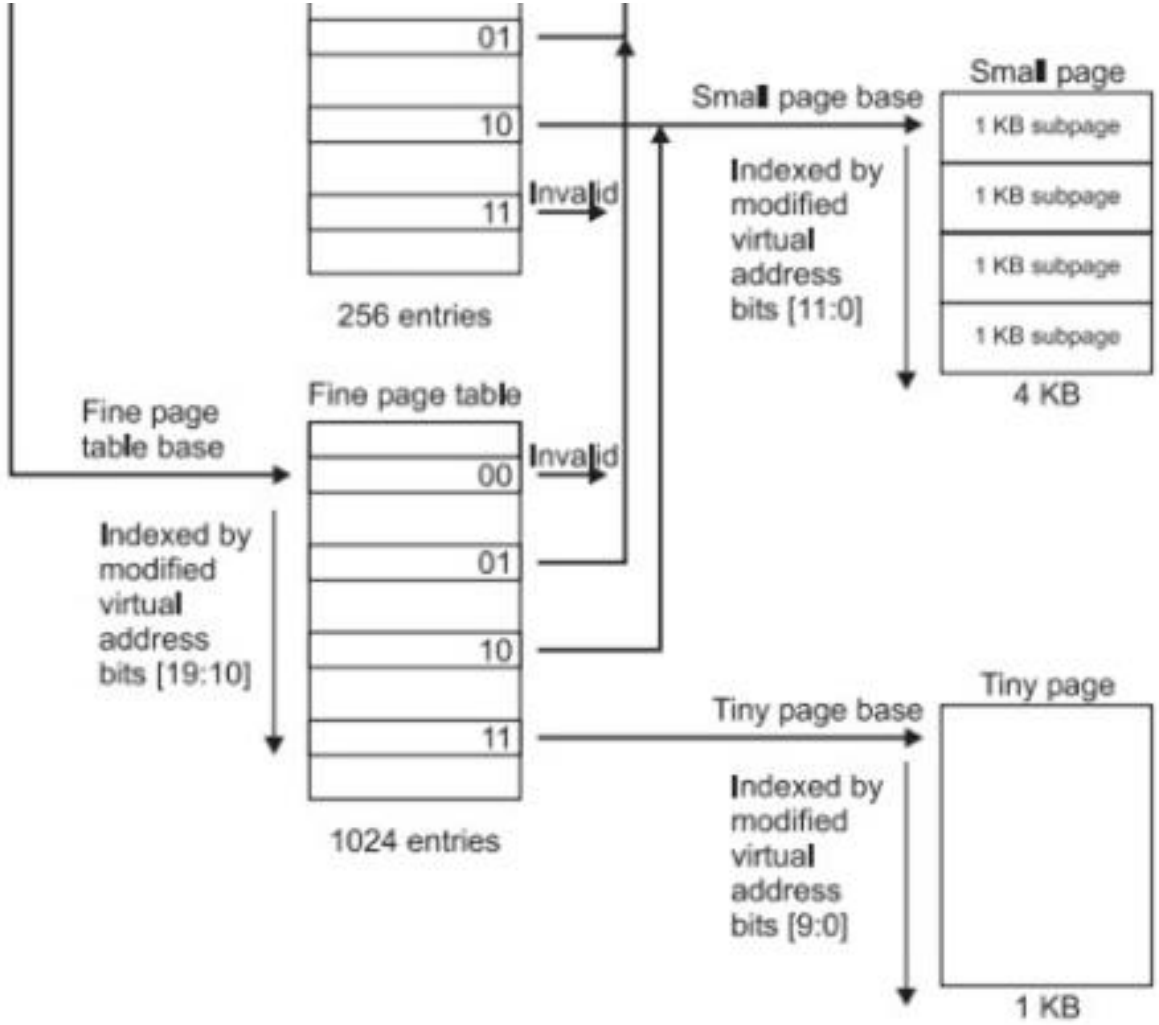
ARM-Linux内存管理（2）

- ARM-Linux存储机制
  - 32位地址形成4GB虚拟地址空间
  - 与x86类似，3G以下是用户空间，3G以上是内核空间
- 内存映射模型
  - 段映射：12位段表，大小1M
  - 粗页表映射：64k或4k/页（4位/8位）
  - 细页表映射：1k/页（10位）

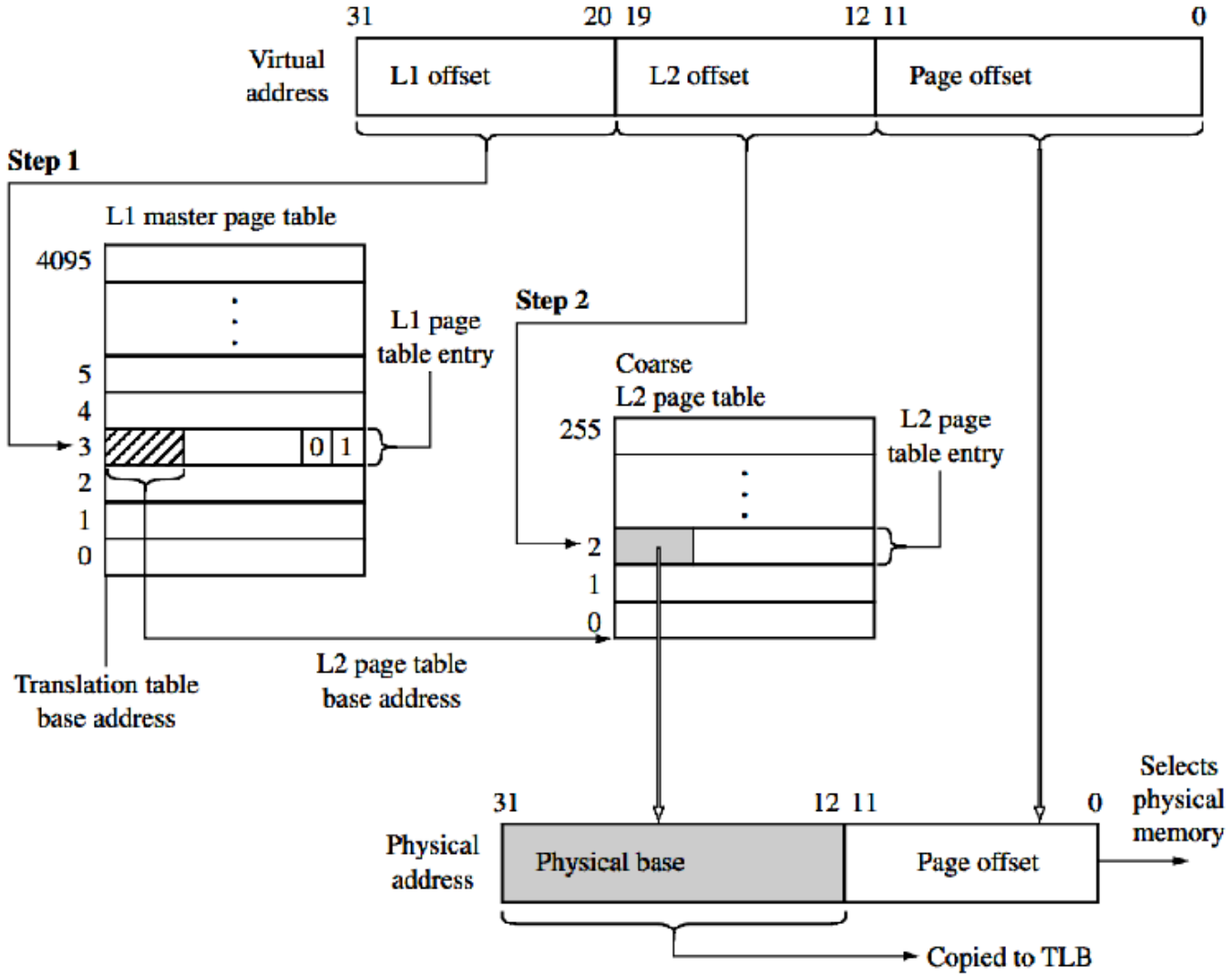
ARM-Linux内存管理（3）



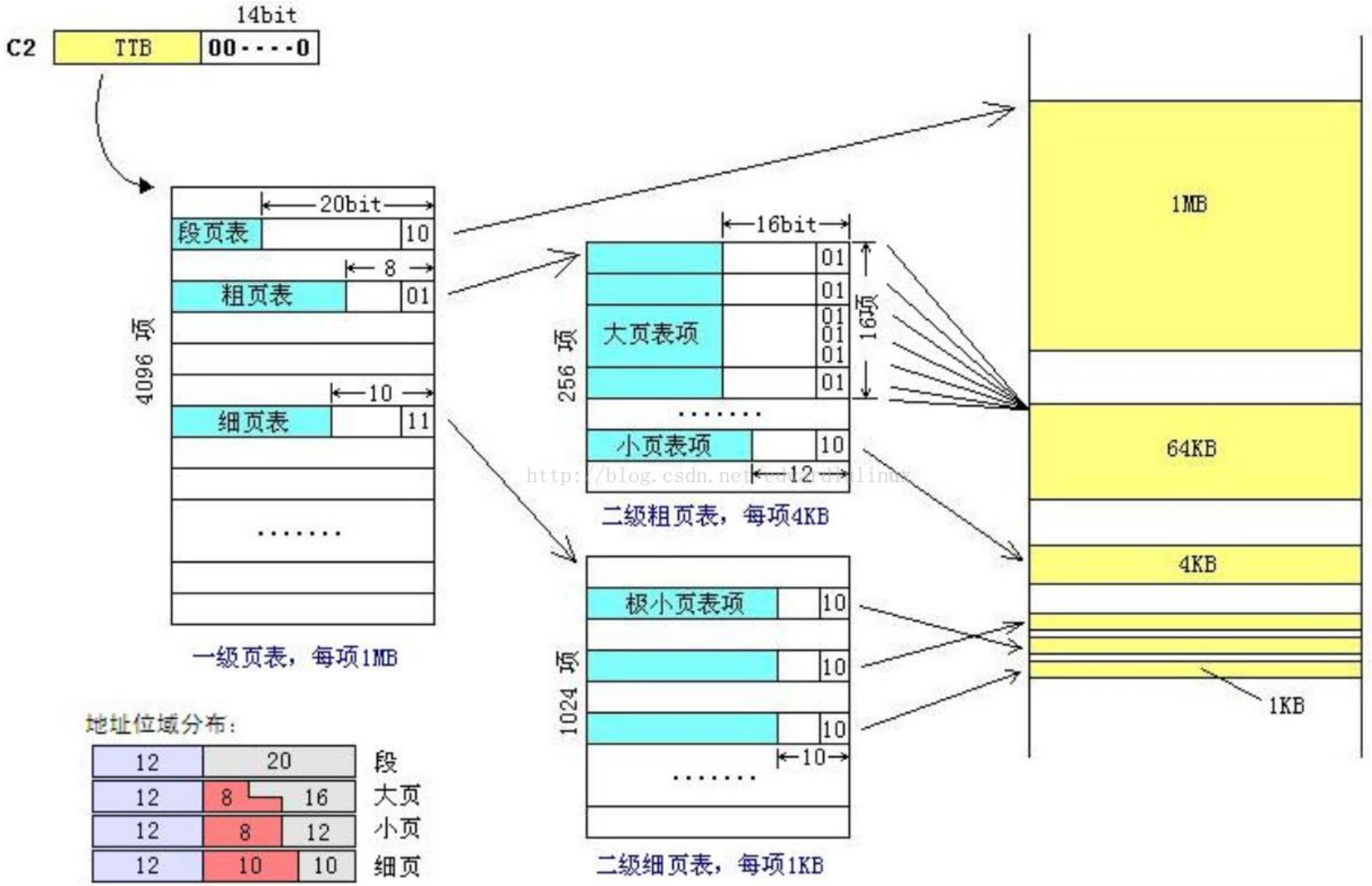
ARM-Linux内存管理（4）



ARM-Linux内存管理（5）

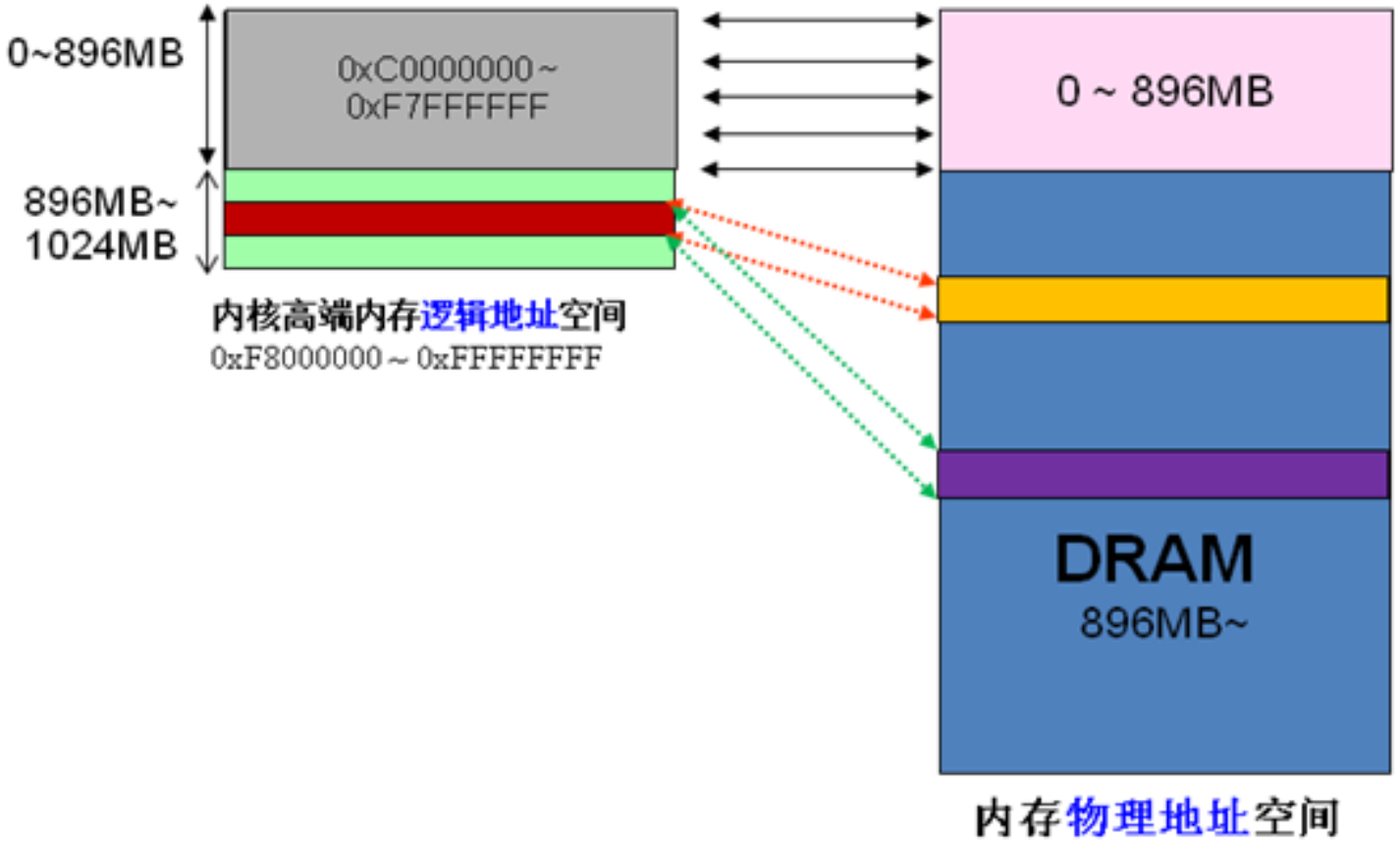


ARM-Linux内存管理（6）



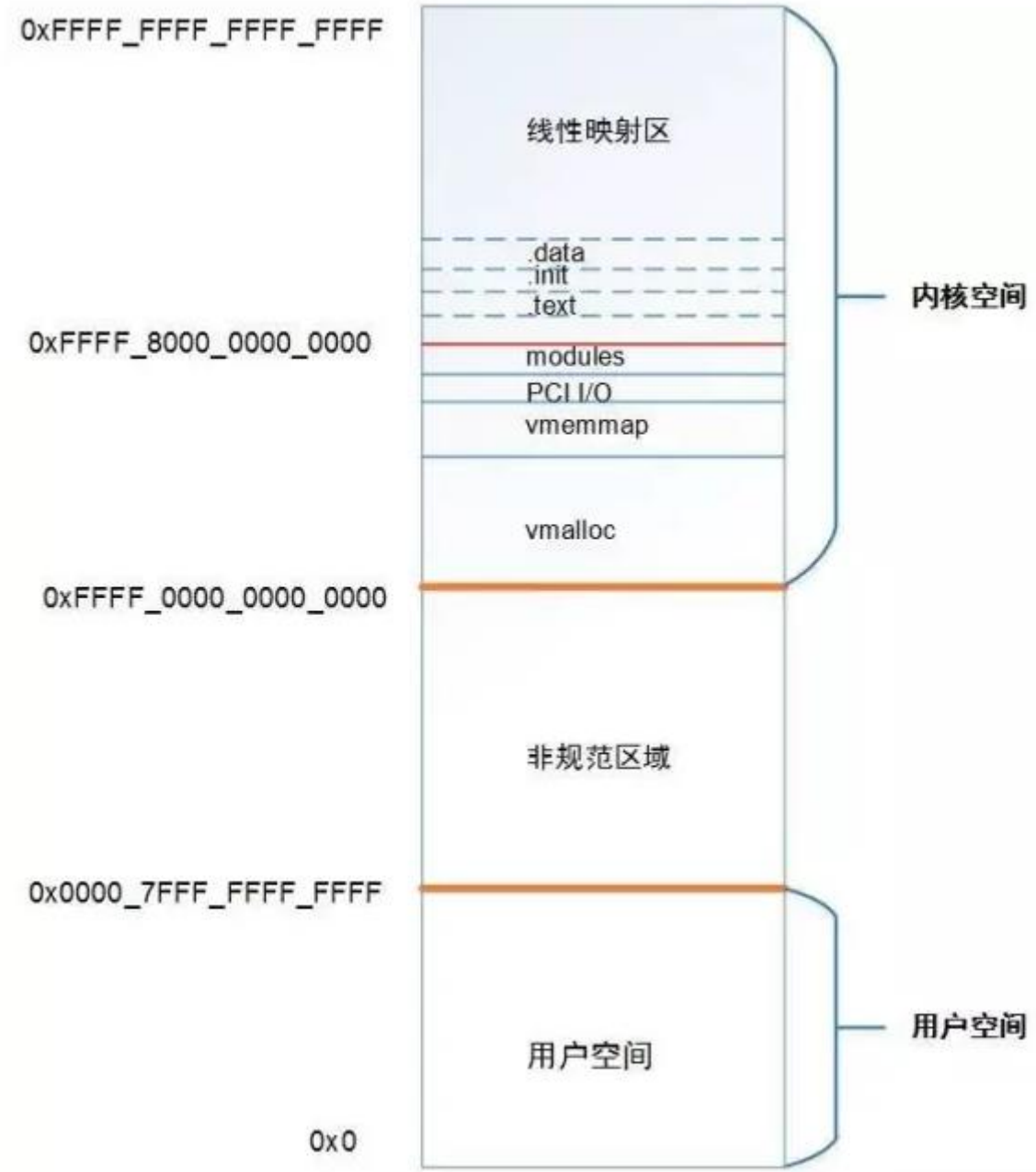
高端内存映射

如何访问大于4G的物理内存



64位Linux

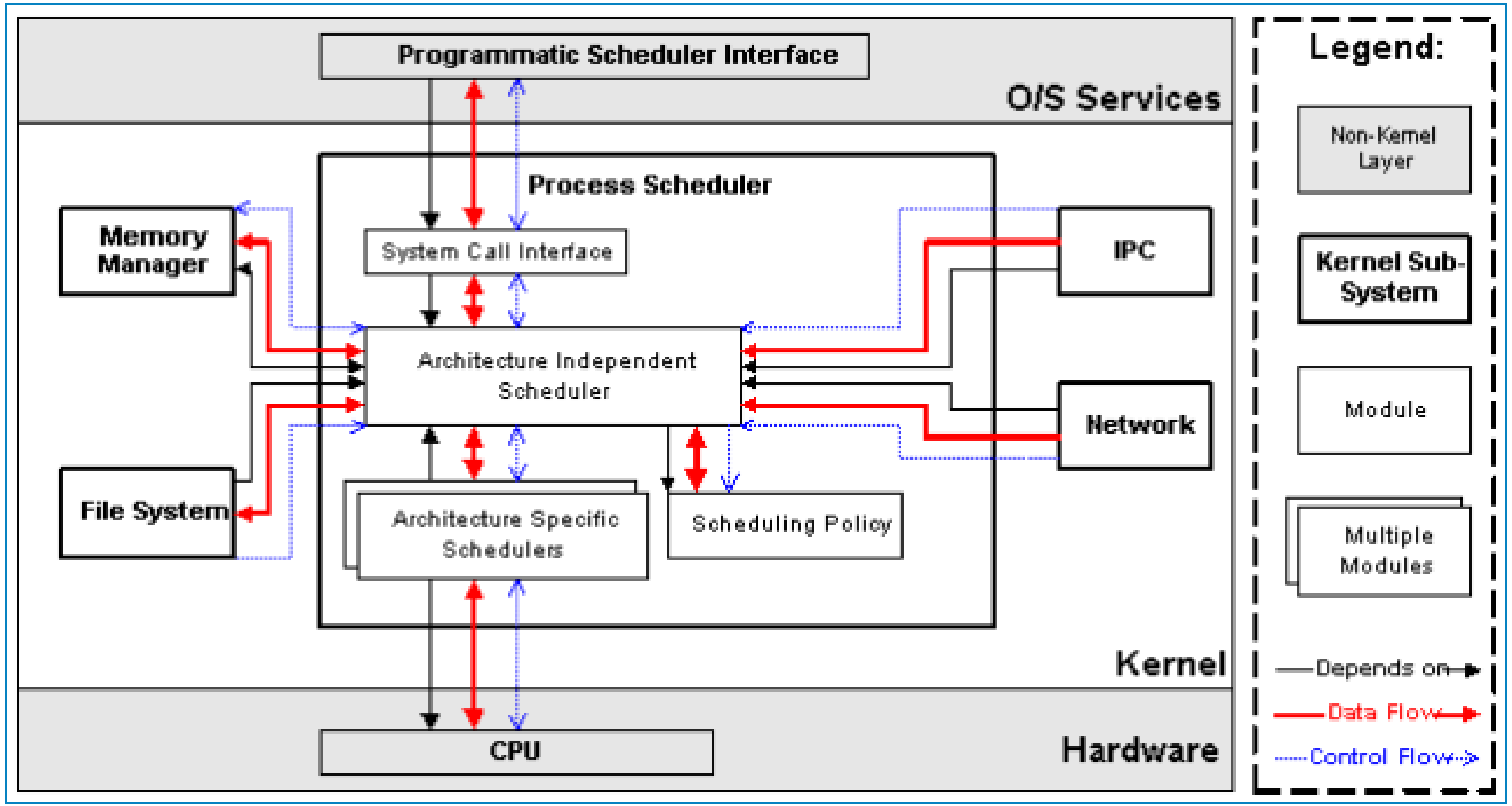
ARM64
采用
48位
物理
寻址



提纲

- ARM-Linux内核简介
- ARM-Linux内存管理
- ARM-Linux进程管理和调度
- ARM-Linux模块机制
- ARM-Linux中断管理
- ARM-Linux系统调用
- ARM-Linux系统启动和初始化

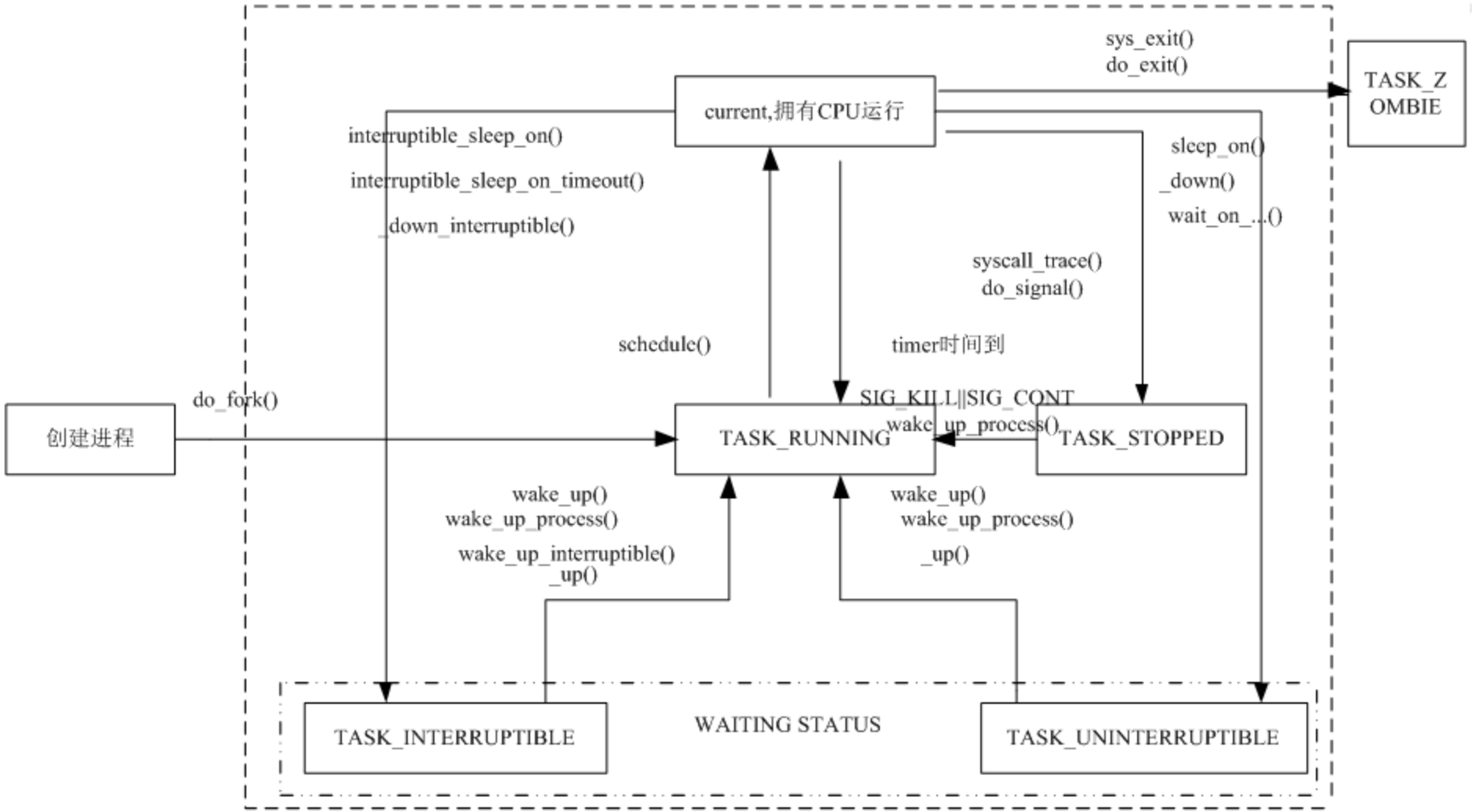
Linux进程管理和调度



ARM-Linux进程管理和调度（1）

- Linux进程的5种状态
  - TASK_RUNNING
  - TASK_INTERRUPTIBLE
  - TASK_UNINTERRUPTIBLE
  - TASK_ZOMBIE
  - TASK_STOPPED

ARM-Linux进程管理和调度（2）



系关变转态状程进

ARM-Linux进程管理和调度（3）

- 进程调度
  - 清理当前运行中的进程
  - 选择下一个投入运行的进程
  - 设置新进程的运行环境
  - 执行进程上下文切换
  - 后期整理
- 调度依据—goodness()
  - policy
  - priority
  - counter
  - rt_priority

提纲

- ARM-Linux内核简介
- ARM-Linux内存管理
- ARM-Linux进程管理和调度
- ARM-Linux模块机制
- ARM-Linux中断管理
- ARM-Linux系统调用
- ARM-Linux系统启动和初始化

ARM-Linux模块机制（1）

- 微内核（Micro-kernel）
  - 操作系统主要组件(内存管理器、进程管理器和I/O管理器)运行在独立进程
- 单一内核（monolithic-kernel）
  - 操作系统核心组件在同一进程实现
- Linux属于什么类型内核？
- Windows呢？

ARM-Linux模块机制（2）

- Linux支持动态可加载内核模块（Loadable Kernel Module，LKM）
- 模块的编译需要配置过的内核源码，生成的内核模块后缀为.ko
- 可根据需要动态加载/卸载，载入内核后，便为内核的一部分，与其他部分地位一致
  - 内核更加模块化
  - 减小内核代码规模
  - 配置更为灵活
  - 内核修改后不需重新编译内核和启动系统

ARM-Linux模块机制（3）

- 内核模块相关的命令
  - lsmod
  - insmod
  - rmmod
  - modprobe
  - depmod

提纲

- ARM-Linux内核简介
- ARM-Linux内存管理
- ARM-Linux进程管理和调度
- ARM-Linux模块机制
- ARM-Linux中断管理
- ARM-Linux系统调用
- ARM-Linux系统启动和初始化

ARM-Linux中断管理（1）

- 中断处理是一个过程
  - 中断响应
  - 中断处理
  - 中断返回
- 快速确定中断源，且使用尽可能少的引脚
  - 中断向量
  - 外部中断控制器
  - 将中断控制器集成在CPU中

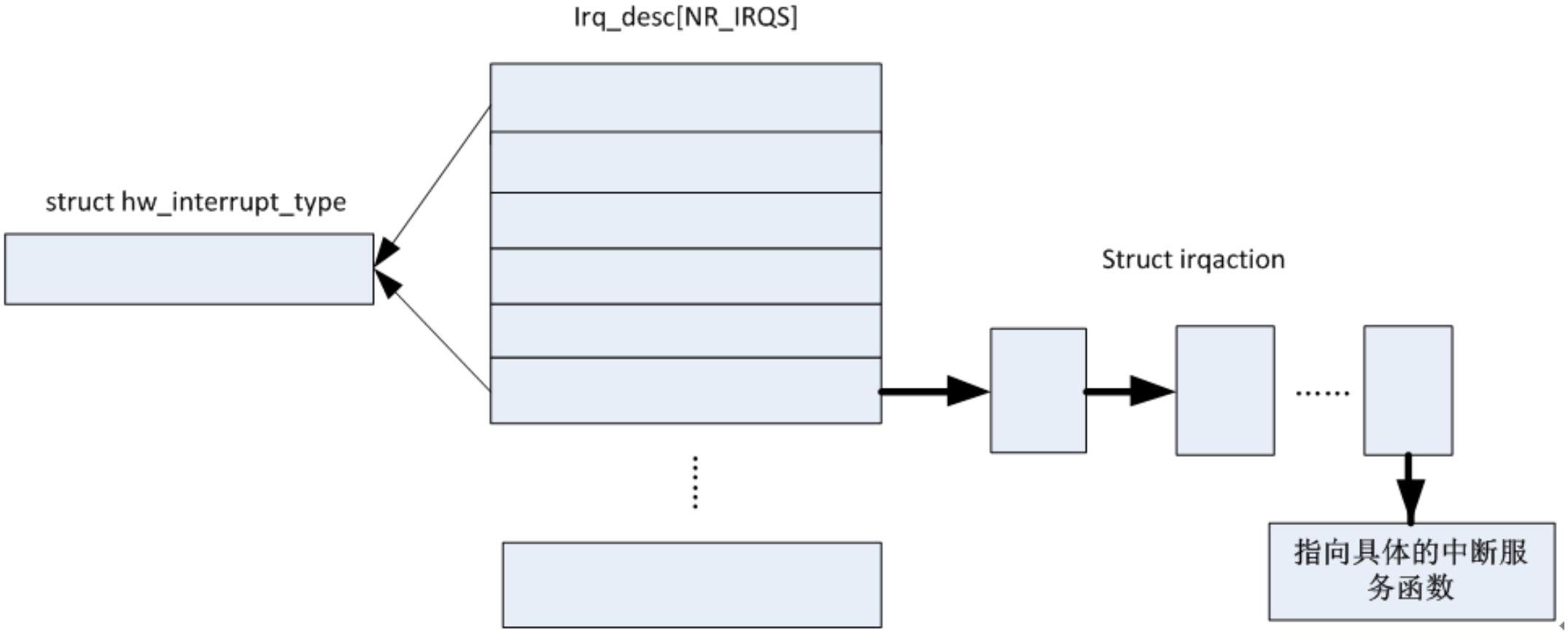
ARM-Linux中断管理（2）

- ARM将中断控制器集成在CPU内部，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中断控制寄存器
- GPIO是一个通用的可编程的I/O接口，每一位都可在程序的控制下设置用于输入或者输出 ；用于输入时，可引发中断请求。

ARM-Linux中断管理（3）

- ARM Linux将中断源分为三组：
  - 第一组是针对外部中断源
  - 第二组中是针对CPU内部中断源，来自集成在芯片内部的外围设备和控制器，比如LCD控制器、串行口、DMA控制器等
  - 第三组中断源使用一个两层结构

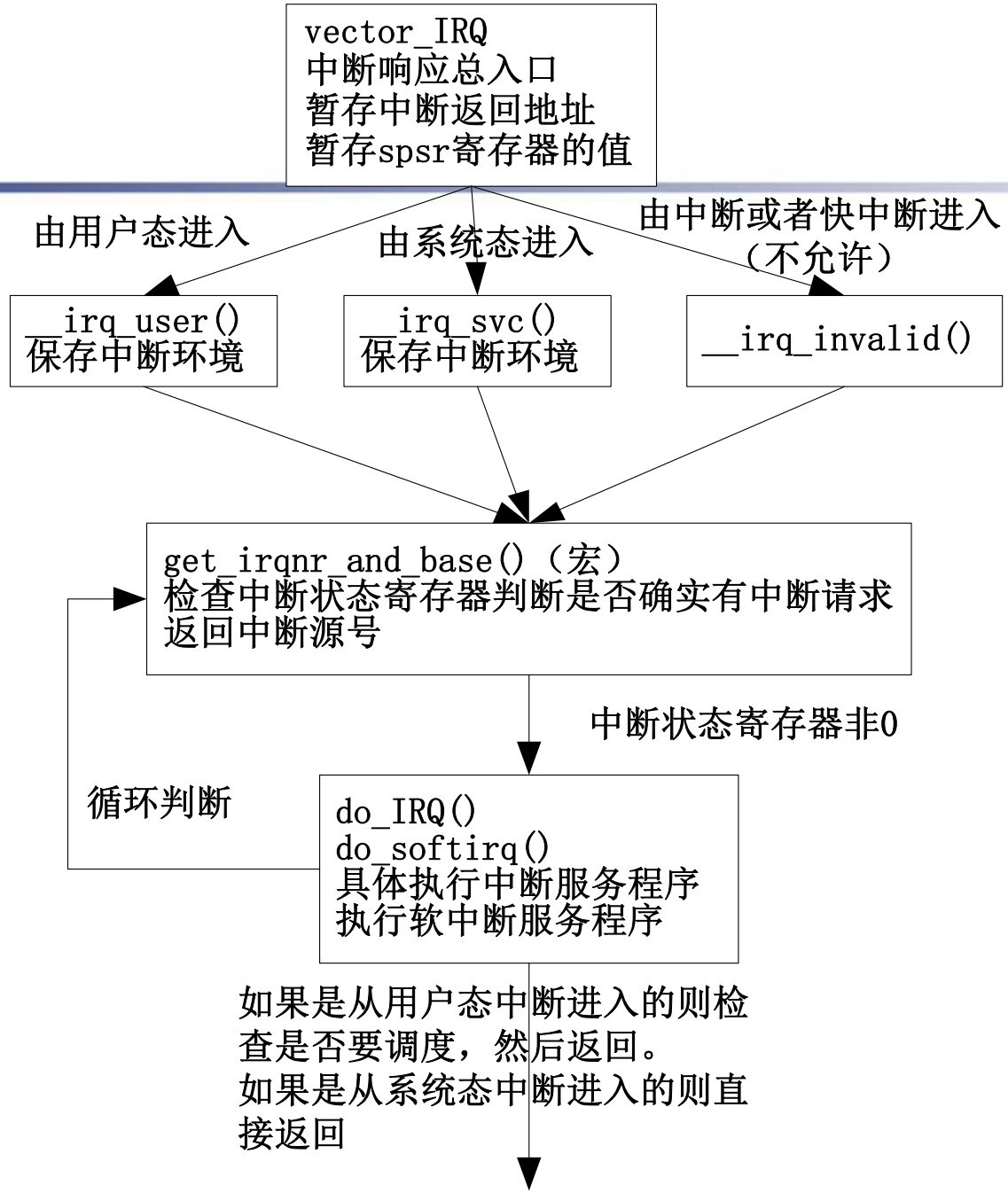
ARM-Linux中断管理（4）



构结据数关相断中

ARM-Linux中断管理（4）

- 中断处理前CPU动作
  - 将进入中断响应前的内容装入r14_irq，即中断模式的lr，使其指向中断点。
  - 将cpsr原来的内容装入spsr_irq，即中断模式的spsr；同时改变cpsr的内容使CPU运行于中断模式，并关闭中断。
  - 将堆栈指针sp切换成中断模式的sp_irq。
  - 将pc指向0x18。


vector_IRQ
中断响应总入口
暂存中断返回地址
暂存spsr寄存器的值
由用户态进入
由系统态进入
由中断或者快中断进入
（不允许）
__irq_user()
保存中断环境
__irq_svc()
保存中断环境
__irq_invalid()
get_irqnr_and_base()（宏）
检查中断状态寄存器判断是否确实有中断请求
返回中断源号
中断状态寄存器非0
循环判断
do_IRQ()
do_softirq()
具体执行中断服务程序
执行软中断服务程序
如果是从用户态中断进入的则检查是否要调度，然后返回。
如果是从系统态中断进入的则直接返回

提纲

- ARM-Linux内核简介
- ARM-Linux内存管理
- ARM-Linux进程管理和调度
- ARM-Linux模块机制
- ARM-Linux中断管理
- ARM-Linux系统调用
- ARM-Linux系统启动和初始化

ARM-Linux系统调用（1）

- 什么是系统调用
- 实现系统调用2种方式
  - 封装C库调用
  - 直接调用
- 自陷指令
  - Intel处理器下什么指令？中断号eax+参数ebx，ecx，edx，esi，edi
  - ARM处理器下呢？中断号？参数？

系统调用处理过程（2）

- 保存当前运行的信息
- 根据系统调用号查找相应的函数去执行
- 恢复原先保存的运行信息返回

提纲

- ARM-Linux内核简介
- ARM-Linux内存管理
- ARM-Linux进程管理和调度
- ARM-Linux模块机制
- ARM-Linux中断管理
- ARM-Linux系统调用
- ARM-Linux系统启动和初始化

ARM-Linux系统启动和初始化（1）

- 使用bootloader将内核映像载入
- 内核数据结构初始化（内核引导第一部分）
  - start_kernel()调用了一系列初始化函数
  - 调用init()过程，创建第一个内核线程
- 外设初始化（内核引导第二部分）
  - init()先锁定内核，调用do_basic_setup()完成外设及其驱动程序的加载和初始化
  - 使用execve()系统调用加载执行init程序

start_kernel（1）

- 输出Linux版本信息（printk(linux_banner)）
- 设置与体系结构相关的环境（setup_arch()）
- 页表结构初始化（paging_init()）
- 设置系统自陷入口（trap_init()）
- 初始化系统IRQ（init_IRQ()）
- 核心进程调度器初始化（包括初始化几个缺省的Bottom-half，sched_init()）
- 时间、定时器初始化（包括读取CMOS时钟、估测主频、初始化定时器中断等，time_init()）
- 提取并分析核心启动参数（从环境变量中读取参数，设置相应标志位等待处理，parse_options()）
- 控制台初始化（为输出信息而先于PCI初始化，console_init()）
- 剖析器数据结构初始化（prof_buffer和prof_len变量）
- 核心Cache初始化（描述Cache信息的Cache，kmem_cache_init()）
- 延迟校准（获得时钟jiffies与CPU主频ticks的延迟，calibrate_delay()）

start_kernel（2）

- 内存初始化（设置内存上下界和页表项初始值，mem_init()）
- 创建和设置内部及通用cache（"slab_cache"，kmem_cache_sizes_init()）
- 创建uid taskcount SLAB cache（"uid_cache"，uidcache_init()）
- 创建文件cache（"files_cache"，filescache_init()）
- 创建目录cache（"dentry_cache"，dcache_init()）
- 创建与虚存相关的cache（"vm_area_struct"，"mm_struct"，vma_init()）
- 块设备读写缓冲区初始化（同时创建"buffer_head"cache用户加速访问，buffer_init()）
- 创建页cache（内存页hash表初始化，page_cache_init()）
- 创建信号队列cache（"signal_queue"，signals_init()）
- 初始化内存inode表（inode_init()）
- 创建内存文件描述符表（"filp_cache"，file_table_init()）
- SMP机器其余CPU（除当前引导CPU）初始化（对于没有配置SMP的内核，此函数为空，smp_init()）
- 启动init过程（创建第一个核心线程，调用init()，原执行序列调用cpu_idle()等待调度）
- 至此start_kernel()结束，基本的核心环境已经建立起来了

do_basic_setup（1）

- 总线初始化（比如pci_init()）
- 网络初始化（初始化网络数据结构，包括sk_init()、skb_init()和proto_init()三部分，在proto_init()中，将调用protocols结构中包含的所有协议的初始化过程，sock_init()）
- 创建bdflush核心线程（bdflush()过程常驻核心空间，由核心唤醒来清理被写过的内存缓冲区，当bdflush()由kernel_thread()启动后，它将自己命名为kflushd）
- 创建kupdate核心线程（kupdate()过程常驻核心空间，由核心按时调度执行，将内存缓冲区中的信息更新到磁盘中，更新的内容包括超级块和inode表）
- 设置并启动核心调页线程kswapd（为了防止kswapd启动时将版本信息输出到其他信息中间，核心线调用kswapd_setup()设置kswapd运行所要求的环境，然后再创建kswapd核心线程）

do_basic_setup（2）

- 创建事件管理核心线程（start_context_thread()函数启动context_thread()过程，并重命名为keventd）
- 设备初始化（包括并口parport_init()、字符设备chr_dev_init()、块设备blk_dev_init()、SCSI设备scsi_dev_init()、网络设备net_dev_init()、磁盘初始化及分区检查等等，device_setup()）
- 执行文件格式设置（binfmt_setup()）
- 启动任何使用__initcall标识的函数（方便核心开发者添加启动函数，do_initcalls()）
- 文件系统初始化（filesystem_setup()）
- 安装root文件系统（mount_root()）
- 加载INIT程序

init函数完成后

- init()函数结束，内核的引导部分也到此结束，这个由start_kernel()创建的第一个线程已经成为一个用户模式下的进程。此时系统中存在着六个运行实体
  - start_kernel()本身所在的执行体，这其实是一个"手工"创建的线程，它在创建了init()线程以后就进入cpu_idle()循环了，它不会在进程（线程）列表中出现
  - init线程，由start_kernel()创建，当前处于用户态，加载了init程序
  - kflushd核心线程，由init线程创建，在核心态运行bdflush()
  - kupdate核心线程，由init线程创建，在核心态运行kupdate()
  - kswapd核心线程，由init线程创建，在核心态运行kswapd()
  - keventd核心线程，由init线程创建，在核心态运行context_thread()

ARM-Linux系统启动和初始化（2）

- init进程和inittab脚本
  - init进程是系统所有进程的起点，它的进程号是1
  - init进程到底是什么可以通过内核参数“init=XXX”设置
  - 通常，init进程是在根目录下的linuxrc脚本文件

linuxrc

```
#!/bin/sh
echo "Setting up RAMFS, please wait ... "
/bin/mount -n -t ramfs ramfs /etc
/bin/cp -a /mnt/etc/* /etc

echo "done and exiting"
exec /sbin/init
```

ARM-Linux系统启动和初始化（3）

- /sbin/init程序读取/etc/inittab文件
- inittab是以行为单位的描述性（非执行性）文本，每一个指令行都具有以下格式：

  id:runlevel:action:process

- rc启动脚本：rc.sysinit常见动作是激活交换分区，检查磁盘，加载硬件模块等
- Shell启动

□谢　谢！