

也就是说，向量中断由硬件提供中断服务程序入口地址，非向量中断由软件提供中断服务程序入口地址。

一个典型的非向量中断服务程序如代码清单 10.1 所示，它先判断中断源，然后调用不同中断源的中断服务程序。

代码清单 10.1 非向量中断服务程序的典型结构

```

1  irq_handler()
2  {
3      ...
4      int int_src = read_int_status();           /* 读硬件的中断相关寄存器 */
5      switch (int_src) {                         /* 判断中断源 */
6          case DEV_A:
7              dev_a_handler();
8              break;
9          case DEV_B:
10             dev_b_handler();
11             break;
12         ...
13     default:
14         break;
15   }
16   ...
17 }
```

嵌入式系统以及 x86 PC 中大多包含可编程中断控制器（PIC），许多 MCU 内部就集成了 PIC。如在 80386 中，PIC 是两片 i8259A 芯片的级联。通过读写 PIC 的寄存器，程序员可以屏蔽 / 使能某中断及获得中断状态，前者一般通过中断 MASK 寄存器完成，后者一般通过中断 PEND 寄存器完成。

定时器在硬件上也依赖中断来实现，图 10.1 所示为典型的嵌入式微处理器内可编程间隔定时器（PIT）的工作原理，它接收一个时钟输入，当时钟脉冲到来时，将目前计数值增 1 并与预先设置的计数值（计数目标）比较，若相等，证明计数周期满，并产生定时器中断且复位目前计数值。



图 10.1 PIT 定时器的工作原理

在 ARM 多核处理器里最常用的中断控制器是 GIC (Generic Interrupt Controller)，如图 10.2 所示，它支持 3 种类型的中断。

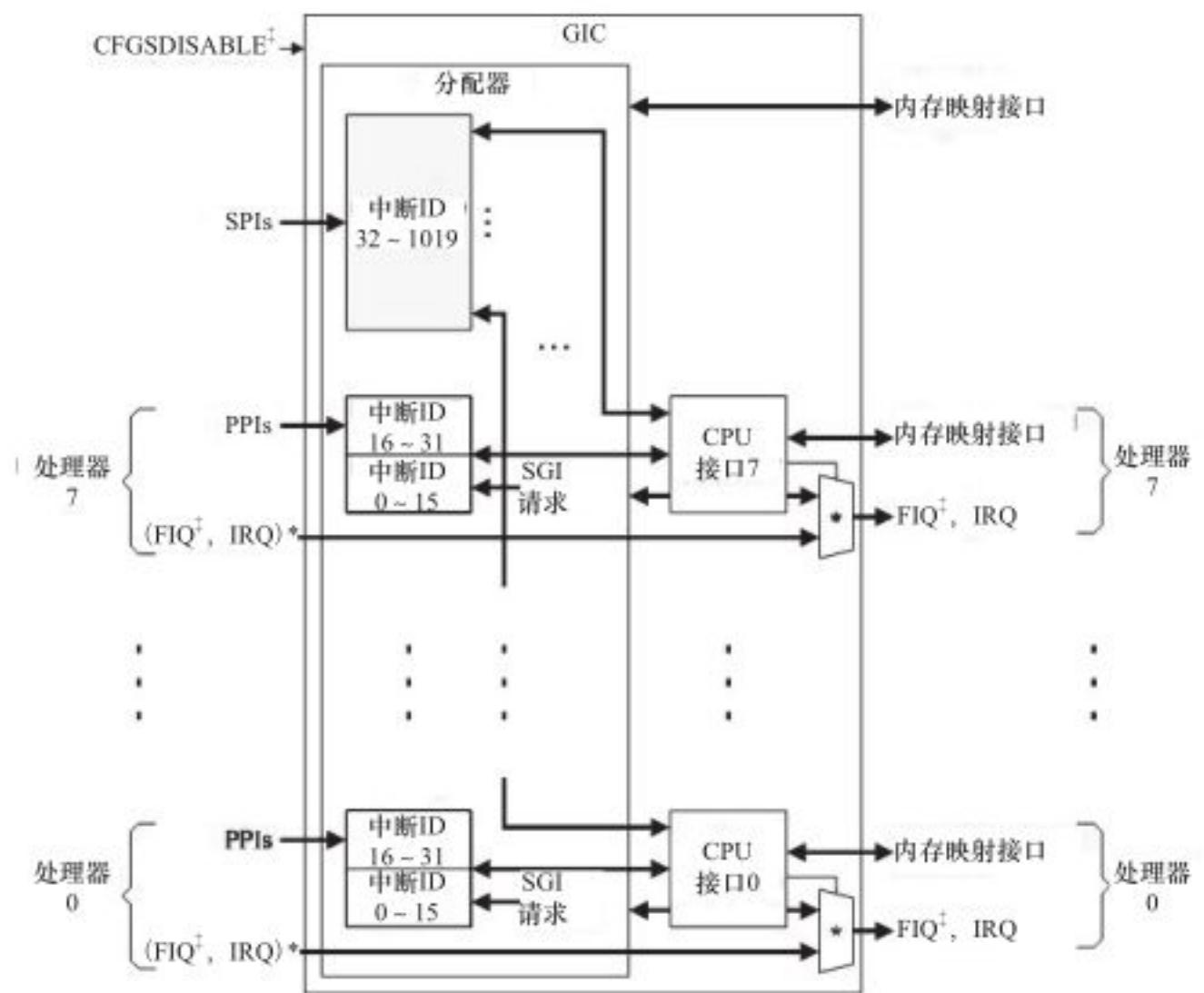


图 10.2 ARM 多核处理器里的 GIC

SGI (Software Generated Interrupt)：软件产生的中断，可以用于多核的核间通信，一个 CPU 可以通过写 GIC 的寄存器给另外一个 CPU 产生中断。多核调度用的 IPI_WAKEUP、IPI_TIMER、IPI_RESCHEDULE、IPI_CALL_FUNC、IPI_CALL_FUNC_SINGLE、IPI_CPU_STOP、IPI_IRQ_WORK、IPI_COMPLETION 都是由 SGI 产生的。

PPI (Private Peripheral Interrupt)：某个 CPU 私有外设的中断，这类外设的中断只能发给绑定的那个 CPU。

SPI (Shared Peripheral Interrupt)：共享外设的中断，这类外设的中断可以路由到任何一个 CPU。

对于 SPI 类型的中断，内核可以通过如下 API 设定中断触发的 CPU 核：

```
extern int irq_set_affinity (unsigned int irq, const struct cpumask *m);
```

在 ARM Linux 默认情况下，中断都是在 CPU0 上产生的，比如，我们可以通过如下代码把中断 `irq` 设定到 CPU i 上去：

```
irq_set_affinity(irq, cpumask_of(i));
```

10.2 Linux 中断处理程序架构

设备的中断会打断内核进程中的正常调度和运行，系统对更高吞吐率的追求势必要求中断服务程序尽量短小精悍。但是，这个良好的愿望往往与现实并不吻合。在大多数真实的系统中，当中断到来时，要完成的工作往往并不会是短小的，它可能要进行较大量的耗时处理。

图 10.3 描述了 Linux 内核的中断处理机制。为了在中断执行时间尽量短和中断处理需完成的工作尽量大之间找到一个平衡点，Linux 将中断处理程序分解为两个半部：顶半部（Top Half）和底半部（Bottom Half）。

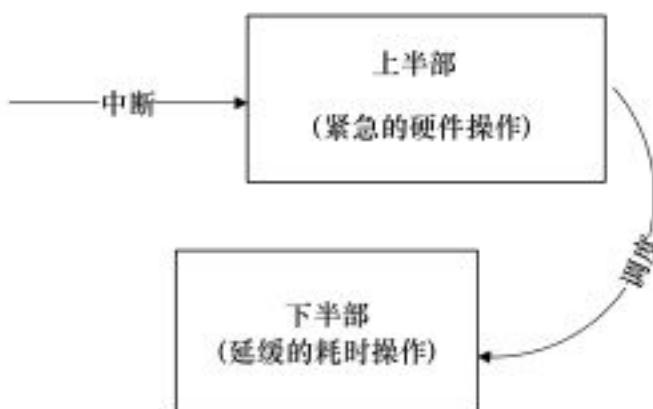


图 10.3 Linux 中断处理机制

顶半部用于完成尽量少的比较紧急的功能，它往往只是简单地读取寄存器中的中断状态，并在清除中断标志后就进行“登记中断”的工作。“登记中断”意味着将底半部处理程序挂到该设备的底半部执行队列中去。这样，顶半部执行的速度就会很快，从而可以服务更多的中断请求。

现在，中断处理工作的重心就落在了底半部的头上，需用它来完成中断事件的绝大多数任务。底半部几乎做了中断处理程序所有的事情，而且可以被新的中断打断，这也是底半部和顶半部的最大不同，因为顶半部往往被设计成不可中断。底半部相对来说并不是非常紧急的，而且相对比较耗时，不在硬件中断服务程序中执行。

尽管顶半部、底半部的结合能够改善系统的响应能力，但是，僵化地认为 Linux 设备驱动中的中断处理一定要分两个半部则是不对的。如果中断要处理的工作本身很少，则完全可以直接在顶半部全部完成。



其他操作系统中对中断的处理也采用了类似于 Linux 的方法，真正的硬件中断服务程序都应该尽量短。因此，许多操作系统都提供了中断上下文和非中断上下文相结合的机制，将中断的耗时工作保留到非中断上下文去执行。例如，在 VxWorks 中，网络设备包接收中断到来后，中断服务程序会通过 `netJobAdd()` 函数将耗时的包接收和上传工作交给 `tNetTask` 任务去执行。

在 Linux 中，查看 /proc/interrupts 文件可以获得系统中中断的统计信息，并能统计出每一个中断号上的中断在每个 CPU 上发生的次数，具体如图 10.4 所示。

	CPU0	CPU1	
0:	119	0	IO-APIC-edge timer
1:	14444	0	IO-APIC-edge i8042
8:	0	0	IO-APIC-edge rtc0
9:	0	0	IO-APIC-fasteoi acpi
12:	2483	0	IO-APIC-edge i8042
14:	0	0	IO-APIC-edge ata_pmix
15:	11679	0	IO-APIC-edge ata_pmix
19:	14749	0	IO-APIC 19-fasteoi ehci_hcd:usb1, eth0
20:	17379	0	IO-APIC 20-fasteoi vboxguest
21:	105402	0	IO-APIC 21-fasteoi 0000:00:0d.0
22:	30	0	IO-APIC 22-fasteoi ohci_hcd:usb2
NMI:	0	0	Non-maskable interrupts
LOC:	564521	478738	Local timer interrupts
SPU:	0	0	Spurious interrupts
PMI:	0	0	Performance monitoring interrupts
IWI:	0	0	IRQ work interrupts
RTR:	0	0	APIC ICR read retries
RES:	263390	382214	Rescheduling interrupts
CAL:	52	50	Function call interrupts
TLB:	830	1201	TLB shootdowns
TRM:	0	0	Thermal event interrupts
THR:	0	0	Threshold APIC interrupts
MCE:	0	0	Machine check exceptions
MCP:	40	40	Machine check polls
ERR:	0	0	
MIS:	0	0	

图 10.4 Linux 中的中断统计信息

10.3 Linux 中断编程

10.3.1 申请和释放中断

在 Linux 设备驱动中，使用中断的设备需要申请和释放对应的中断，并分别使用内核提供的 request_irq() 和 free_irq() 函数。

1. 申请 irq

```
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
                const char *name, void *dev);
```

irq 是要申请的硬件中断号。

handler 是向系统登记的中断处理函数（顶半部），是一个回调函数，中断发生时，系统调用这个函数，dev 参数将被传递给它。

irqflags 是中断处理的属性，可以指定中断的触发方式以及处理方式。在触发方式方

面，可以是 IRQF_TRIGGER_RISING、IRQF_TRIGGER_FALLING、IRQF_TRIGGER_HIGH、IRQF_TRIGGER_LOW 等。在处理方式方面，若设置了 IRQF_SHARED，则表示多个设备共享中断，dev 是要传递给中断服务程序的私有数据，一般设置为这个设备的设备结构体或者 NULL。

request_irq() 返回 0 表示成功，返回 -EINVAL 表示中断号无效或处理函数指针为 NULL，返回 -EBUSY 表示中断已经被占用且不能共享。

```
int devm_request_irq(struct device *dev, unsigned int irq, irq_handler_t handler,
                     unsigned long irqflags, const char *devname, void *dev_id);
```

此函数与 request_irq() 的区别是 devm_ 开头的 API 申请的是内核“managed”的资源，一般不需要在出错处理和 remove() 接口里再显式的释放。有点类似 Java 的垃圾回收机制。比如，对于 at86rf230 驱动，如下的补丁中改用 devm_request_irq() 后就删除了 free_irq()，该补丁对应的内核 commit ID 是 652355c5。

```
--- a/drivers/net/ieee802154/at86rf230.c
+++ b/drivers/net/ieee802154/at86rf230.c
@@ -1190,24 +1190,22 @@ static int at86rf230_probe(struct spi_device *spi)
     if (rc)
         goto err_hw_init;

-    rc = request_irq(spi->irq, irq_handler, IRQF_SHARED,
-                     dev_name(&spi->dev), lp);
+    rc = devm_request_irq(&spi->dev, spi->irq, irq_handler, IRQF_SHARED,
+                         dev_name(&spi->dev), lp);
     if (rc)
         goto err_hw_init;

     /* Read irq status register to reset irq line */
     rc = at86rf230_read_subreg(lp, RG_IRQ_STATUS, 0xff, 0, &status);
     if (rc)
-        goto err_irq;
+        goto err_hw_init;

     rc = ieee802154_register_device(lp->dev);
     if (rc)
-        goto err_irq;
+        goto err_hw_init;

     return rc;

-err_irq:
-    free_irq(spi->irq, lp);
 err_hw_init:
     flush_work(&lp->irqwork);
     spi_set_drvdata(spi, NULL);
@@ -1232,7 +1230,6 @@ static int at86rf230_remove(struct spi_device *spi)
```

```

at86rf230_write_subreg(lp, SR_IRQ_MASK, 0);
ieee802154_unregister_device(lp->dev);

- free_irq(spi->irq, lp);
flush_work(&lp->irqwork);

if (gpio_is_valid(pdata->slp_tr))

```

顶半部 handler 的类型 irq_handler_t 定义为：

```

typedef irqreturn_t (*irq_handler_t)(int, void *);
typedef int irqreturn_t;

```

2. 释放 irq

与 request_irq() 相对应的函数为 free_irq(), free_irq() 的原型为：

```
void free_irq(unsigned int irq, void *dev_id);
```

free_irq() 中参数的定义与 request_irq() 相同。

10.3.2 使能和屏蔽中断

下列 3 个函数用于屏蔽一个中断源：

```

void disable_irq(int irq);
void disable_irq_nosync(int irq);
void enable_irq(int irq);

```

disable_irq_nosync() 与 disable_irq() 的区别在于前者立即返回，而后者等待目前的中断处理完成。由于 disable_irq() 会等待指定的中断被处理完，因此如果在 n 号中断的顶半部调用 disable_irq(n)，会引起系统的死锁，这种情况下，只能调用 disable_irq_nosync(n)。

下列两个函数（或宏，具体实现依赖于 CPU 的体系结构）将屏蔽本 CPU 内的所有中断：

```

#define local_irq_save(flags) ...
void local_irq_disable(void);

```

前者会将目前的中断状态保留在 flags 中（注意 flags 为 unsigned long 类型，被直接传递，而不是通过指针），后者直接禁止中断而不保存状态。

与上述两个禁止中断对应的恢复中断的函数（或宏）是：

```

#define local_irq_restore(flags) ...
void local_irq_enable(void);

```

以上各以 local_ 开头的方法的作用范围是本 CPU 内。

10.3.3 底半部机制

Linux 实现底半部的机制主要有 tasklet、工作队列、软中断和线程化 irq。

1. tasklet

tasklet 的使用较简单，它的执行上下文是软中断，执行时机通常是顶半部返回的时候。我们只需要定义 tasklet 及其处理函数，并将两者关联则可，例如：

```
void my_tasklet_func(unsigned long); /* 定义一个处理函数 */
DECLARE_TASKLET(my_tasklet, my_tasklet_func, data);
/* 定义一个 tasklet 结构 my_tasklet，与 my_tasklet_func(data) 函数相关联 */
```

代码 DECLARE_TASKLET(my_tasklet, my_tasklet_func, data) 实现了定义名称为 my_tasklet 的 tasklet，并将其与 my_tasklet_func() 这个函数绑定，而传入这个函数的参数为 data。

在需要调度 tasklet 的时候引用一个 tasklet_schedule() 函数就能使系统在适当的时候进行调度运行：

```
tasklet_schedule(&my_tasklet);
```

使用 tasklet 作为底半部处理中断的设备驱动程序模板如代码清单 10.2 所示（仅包含与中断相关的部分）。

代码清单 10.2 tasklet 使用模板

```
1 /* 定义 tasklet 和底半部函数并将它们关联 */
2 void xxx_do_tasklet(unsigned long);
3 DECLARE_TASKLET(xxx_tasklet, xxx_do_tasklet, 0);
4
5 /* 中断处理底半部 */
6 void xxx_do_tasklet(unsigned long)
7 {
8     ...
9 }
10
11 /* 中断处理顶半部 */
12 irqreturn_t xxx_interrupt(int irq, void *dev_id)
13 {
14     ...
15     tasklet_schedule(&xxx_tasklet);
16     ...
17 }
18
19 /* 设备驱动模块加载函数 */
20 int __init xxx_init(void)
21 {
22     ...
23     /* 申请中断 */
24     result = request_irq(xxx_irq, xxx_interrupt,
25             0, "xxx", NULL);
26     ...
27     return IRQ_HANDLED;
28 }
```

```

29
30 /* 设备驱动模块卸载函数 */
31 void __exit xxx_exit(void)
32 {
33 ...
34 /* 释放中断 */
35 free_irq(xxx_irq, xxx_interrupt);
36 ...
37 }

```

上述程序在模块加载函数中申请中断（第 24 ~ 25 行），并在模块卸载函数中释放它（第 35 行）。对应于 `xxx_irq` 的中断处理程序被设置为 `xxx_interrupt()` 函数，在这个函数中，第 15 行的 `tasklet_schedule(&xxx_tasklet)` 调度被定义的 tasklet 函数 `xxx_do_tasklet()` 在适当的时候执行。

2. 工作队列

工作队列的使用方法和 tasklet 非常相似，但是工作队列的执行上下文是内核线程，因此可以调度和睡眠。下面的代码用于定义一个工作队列和一个底半部执行函数：

```

struct work_struct my_wq;           /* 定义一个工作队列 */
void my_wq_func(struct work_struct *work); /* 定义一个处理函数 */

```

通过 `INIT_WORK()` 可以初始化这个工作队列并将工作队列与处理函数绑定：

```

INIT_WORK(&my_wq, my_wq_func);
/* 初始化工作队列并将其与处理函数绑定 */

```

与 `tasklet_schedule()` 对应的用于调度工作队列执行的函数为 `schedule_work()`，如：

```

schedule_work(&my_wq);           /* 调度工作队列执行 */

```

与代码清单 10.2 对应的使用工作队列处理中断底半部的设备驱动程序模板如代码清单 10.3 所示（仅包含与中断相关的部分）。

代码清单 10.3 工作队列使用模板

```

1 /* 定义工作队列和关联函数 */
2 struct work_struct xxx_wq;
3 void xxx_do_work(struct work_struct *work);
4
5 /* 中断处理底半部 */
6 void xxx_do_work(struct work_struct *work)
7 {
8 ...
9 }
10
11 /* 中断处理顶半部 */
12 irqreturn_t xxx_interrupt(int irq, void *dev_id)
13 {

```

```

14 ...
15 schedule_work(&xxx_wq);
16 ...
17 return IRQ_HANDLED;
18 }
19
20 /* 设备驱动模块加载函数 */
21 int xxx_init(void)
22 {
23 ...
24 /* 申请中断 */
25 result = request_irq(xxx_irq, xxx_interrupt,
26 0, "xxx", NULL);
27 ...
28 /* 初始化工作队列 */
29 INIT_WORK(&xxx_wq, xxx_do_work);
30 ...
31 }
32
33 /* 设备驱动模块卸载函数 */
34 void xxx_exit(void)
35 {
36 ...
37 /* 释放中断 */
38 free_irq(xxx_irq, xxx_interrupt);
39 ...
40 }

```

与代码清单 10.2 不同的是，上述程序在设计驱动模块加载函数中增加了初始化工作队列的代码（第 29 行）。

工作队列早期的实现是在每个 CPU 核上创建一个 worker 内核线程，所有在这个核上调度的工作都在该 worker 线程中执行，其并发性显然差强人意。在 Linux 2.6.36 以后，转而实现了“Concurrency-managed workqueues”，简称 cmwq，cmwq 会自动维护工作队列的线程池以提高并发性，同时保持了 API 的向后兼容。

3. 软中断

软中断（Softirq）也是一种传统的底半部处理机制，它的执行时机通常是顶半部返回的时候，tasklet 是基于软中断实现的，因此也运行于软中断上下文。

在 Linux 内核中，用 softirq_action 结构体表征一个软中断，这个结构体包含软中断处理函数指针和传递给该函数的参数。使用 open_softirq() 函数可以注册软中断对应的处理函数，而 raise_softirq() 函数可以触发一个软中断。

软中断和 tasklet 运行于软中断上下文，仍然属于原子上下文的一种，而工作队列则运行于进程上下文。因此，在软中断和 tasklet 处理函数中不允许睡眠，而在工作队列处理函数中允许睡眠。

local_bh_disable() 和 local_bh_enable() 是内核中用于禁止和使能软中断及 tasklet 底半部

机制的函数。

内核中采用 softirq 的地方包括 HI_SOFTIRQ、TIMER_SOFTIRQ、NET_TX_SOFTIRQ、NET_RX_SOFTIRQ、SCSI_SOFTIRQ、TASKLET_SOFTIRQ 等，一般来说，驱动的编写者不会也不宜直接使用 softirq。

第 9 章异步通知所基于的信号也类似于中断，现在，总结一下硬中断、软中断和信号的区别：硬中断是外部设备对 CPU 的中断，软中断是中断底半部的一种处理机制，而信号则是由内核（或其他进程）对某个进程的中断。在涉及系统调用的场合，人们也常说通过软中断（例如 ARM 为 swi）陷入内核，此时软中断的概念是指由软件指令引发的中断，和我们这个地方说的 softirq 是两个完全不同的概念，一个是 software，一个是 soft。

需要特别说明的是，软中断以及基于软中断的 tasklet 如果在某段时间内大量出现的话，内核会把后续软中断放入 ksoftirqd 内核线程中执行。总的来说，中断优先级高于软中断，软中断又高于任何一个线程。软中断适度线程化，可以缓解高负载情况下系统的响应。

4. threaded_irq

在内核中，除了可以通过 request_irq()、devm_request_irq() 申请中断以外，还可以通过 request_threaded_irq() 和 devm_request_threaded_irq() 申请。这两个函数的原型为：

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
                         irq_handler_t thread_fn,
                         unsigned long flags, const char *name, void *dev);
int devm_request_threaded_irq(struct device *dev, unsigned int irq,
                             irq_handler_t handler, irq_handler_t thread_fn,
                             unsigned long irqflags, const char *devname,
                             void *dev_id);
```

由此可见，它们比 request_irq()、devm_request_irq() 多了一个参数 thread_fn。用这两个 API 申请中断的时候，内核会为相应的中断号分配一个对应的内核线程。注意这个线程只针对这个中断号，如果其他中断也通过 request_threaded_irq() 申请，自然会得到新的内核线程。

参数 handler 对应的函数执行于中断上下文，thread_fn 参数对应的函数则执行于内核线程。如果 handler 结束的时候，返回值是 IRQ_WAKE_THREAD，内核会调度对应线程执行 thread_fn 对应的函数。

request_threaded_irq() 和 devm_request_threaded_irq() 支持在 irqflags 中设置 IRQF_ONESHOT 标记，这样内核会自动帮助我们在中断上下文中屏蔽对应的中断号，而在内核调度 thread_fn 执行后，重新使能该中断号。对于我们无法在上半部清除中断的情况，IRQF_ONESHOT 特别有用，避免了中断服务程序一退出，中断就洪泛的情况。

handler 参数可以设置为 NULL，这种情况下，内核会用默认的 irq_default_primary_handler() 代替 handler，并会使用 IRQF_ONESHOT 标记。irq_default_primary_handler() 定义为：

```
/*
 * Default primary interrupt handler for threaded interrupts. Is
```

```

* assigned as primary handler when request_threaded_irq is called
* with handler == NULL. Useful for oneshot interrupts.
*/
static irqreturn_t irq_default_primary_handler(int irq, void *dev_id)
{
    return IRQ_WAKE_THREAD;
}

```

10.3.4 实例：GPIO 按键的中断

`drivers/input/keyboard/gpio_keys.c` 是一个放之四海皆准的 GPIO 按键驱动，为了让该驱动在特定的电路板上工作，通常只需要修改 `arch/arm/mach-xxx` 下的板文件或者修改 device tree 对应的 dts。该驱动会为每个 GPIO 申请中断，在 `gpio_keys_setup_key()` 函数中进行。注意最后一个参数 `bdata`，会被传入中断服务程序。

代码清单 10.4 GPIO 按键驱动中断申请

```

1 static int gpio_keys_setup_key(struct platform_device *pdev,
2                                struct input_dev *input,
3                                struct gpio_button_data *bdata,
4                                const struct gpio_keys_button *button)
5 {
6     ...
7
8     error = request_any_context_irq(bdata->irq, isr, irqflags, desc, bdata);
9     if (error < 0) {
10         dev_err(dev, "Unable to claim irq %d; error %d\n",
11                 bdata->irq, error);
12         goto fail;
13     }
14     ...
15 }

```

第 8 行的 `request_any_context_irq()` 会根据 GPIO 控制器本身的“上级”中断是否为 `threaded_irq` 来决定采用 `request_irq()` 还是 `request_threaded_irq()`。一组 GPIO（如 32 个 GPIO）虽然每个都提供一个中断，并且都有中断号，但是在硬件上一组 GPIO 通常是嵌套在上一级的中断控制器上的一个中断。

`request_any_context_irq()` 也有一个变体是 `devm_request_any_context_irq()`。

在 GPIO 按键驱动的 `remove_key()` 函数中，会释放 GPIO 对应的中断，如代码清单 10.5 所示。

代码清单 10.5 GPIO 按键驱动中断释放

```

1 static void gpio_remove_key(struct gpio_button_data *bdata)
2 {
3     free_irq(bdata->irq, bdata);

```

```

4     if (bdata->timer_debounce)
5         del_timer_sync(&bdata->timer);
6     cancel_work_sync(&bdata->work);
7     if (gpio_is_valid(bdata->button->gpio))
8         gpio_free(bdata->button->gpio);
9 }

```

GPIO 按键驱动的中断处理比较简单，没有明确地分为上下两个半部，而只存在顶半部，如代码清单 10.6 所示。

代码清单 10.6 GPIO 按键驱动中断处理程序

```

1 static irqreturn_t gpio_keys_gpio_isr(int irq, void *dev_id)
2 {
3     struct gpio_button_data *bdata = dev_id;
4
5     BUG_ON(irq != bdata->irq);
6
7     if (bdata->button->wakeup)
8         pm_stay_awake(bdata->input->dev.parent);
9     if (bdata->timer_debounce)
10        mod_timer(&bdata->timer,
11                   jiffies + msecs_to_jiffies(bdata->timer_debounce));
12    else
13        schedule_work(&bdata->work);
14
15    return IRQ_HANDLED;
16 }

```

第 3 行直接从 dev_id 取出了 bdata，这就是对应的那个 GPIO 键的数据结构，之后根据情况启动 timer 以进行 debounce 或者直接调度工作队列去汇报按键事件。在 GPIO 按键驱动初始化的时候，通过 INIT_WORK(&bdata->work, gpio_keys_gpio_work_func) 初始化了 bdata->work，对应的处理函数是 gpio_keys_gpio_work_func()，如代码清单 10.7 所示。

代码清单 10.7 GPIO 按键驱动的工作队列底半部

```

1 static void gpio_keys_gpio_work_func(struct work_struct *work)
2 {
3     struct gpio_button_data *bdata =
4         container_of(work, struct gpio_button_data, work);
5
6     gpio_keys_gpio_report_event(bdata);
7
8     if (bdata->button->wakeup)
9         pm_relax(bdata->input->dev.parent);
10 }

```

观察其中的第 3 ~ 4 行，它通过 container_of() 再次从 work_struct 反向解析出了 bdata。

原因是 work_struct 本身在定义时，就嵌入在 gpio_button_data 结构体内。读者朋友们应该掌握 Linux 的这种可以到处获取一个结构体指针的技巧，它实际上类似于面向对象里面的“this”指针。

```
struct gpio_button_data {
    const struct gpio_keys_button *button;
    struct input_dev *input;
    struct timer_list timer;
    struct work_struct work;
    unsigned int timer_debounce; /* in msecs */
    unsigned int irq;
    spinlock_t lock;
    bool disabled;
    bool key_pressed;
};
```

10.4 中断共享

多个设备共享一根硬件中断线的情况在实际的硬件系统中广泛存在，Linux 支持这种中断共享。下面是中断共享的使用方法。

1) 共享中断的多个设备在申请中断时，都应该使用 IRQF_SHARED 标志，而且一个设备以 IRQF_SHARED 申请某中断成功的前提是该中断未被申请，或该中断虽然被申请了，但是之前申请该中断的所有设备也都以 IRQF_SHARED 标志申请该中断。

2) 尽管内核模块可访问的全局地址都可以作为 request_irq(..., void *dev_id) 的最后一个参数 dev_id，但是设备结构体指针显然是可传入的最佳参数。

3) 在中断到来时，会遍历执行共享此中断的所有中断处理程序，直到某一个函数返回 IRQ_HANDLED。在中断处理程序顶半部中，应根据硬件寄存器中的信息比照传入的 dev_id 参数迅速地判断是否为本设备的中断，若不是，应迅速返回 IRQ_NONE，如图 10.5 所示。

代码清单 10.8 给出了使用共享中断的设备驱动程序的模板（仅包含与共享中断机制相关的部分）。

代码清单 10.8 共享中断编程模板

```
1 /* 中断处理顶半部 */
2 irqreturn_t xxx_interrupt(int irq, void *dev_id)
3 {
4     ...
5     int status = read_int_status(); /* 获知中断源 */
```

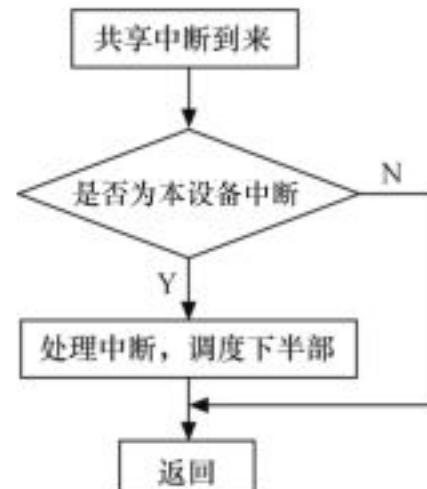


图 10.5 共享中断的处理

```

6   if(!is_myint(dev_id,status))           /* 判断是否为本设备中断 */
7       return IRQ_NONE;                  /* 不是本设备中断，立即返回 */
8
9   /* 是本设备中断，进行处理 */
10  ...
11  return IRQ_HANDLED;                 /* 返回 IRQ_HANDLED 表明中断已被处理 */
12 }
13
14 /* 设备驱动模块加载函数 */
15 int xxx_init(void)
16 {
17     ...
18     /* 申请共享中断 */
19     result = request_irq(sh_irq, xxx_interrupt,
20             IRQF_SHARED, "xxx", xxx_dev);
21     ...
22 }
23
24 /* 设备驱动模块卸载函数 */
25 void xxx_exit(void)
26 {
27     ...
28     /* 释放中断 */
29     free_irq(xxx_irq, xxx_interrupt);
30     ...
31 }

```

10.5 内核定时器

10.5.1 内核定时器编程

软件意义上的定时器最终依赖硬件定时器来实现，内核在时钟中断发生后检测各定时器是否到期，到期后的定时器处理函数将作为软中断在底半部执行。实质上，时钟中断处理程序会唤起 TIMER_SOFTIRQ 软中断，运行当前处理器上到期的所有定时器。

在 Linux 设备驱动编程中，可以利用 Linux 内核中提供的一组函数和数据结构来完成定时触发工作或者完成某周期性的事务。这组函数和数据结构使得驱动工程师在多数情况下不用关心具体的软件定时器究竟对应着怎样的内核和硬件行为。

Linux 内核所提供的用于操作定时器的数据结构和函数如下。

1. timer_list

在 Linux 内核中，timer_list 结构体的一个实例对应一个定时器，如代码清单 10.9 所示。

代码清单 10.9 timer_list 结构体

```

1 struct timer_list {
2     /*
3      * All fields that change during normal runtime grouped to the

```

```

4         * same cacheline
5         */
6         struct list_head entry;
7         unsigned long expires;
8         struct tvec_base *base;
9
10        void (*function)(unsigned long);
11        unsigned long data;
12
13        int slack;
14
15 #ifdef CONFIG_TIMER_STATS
16         int start_pid;
17         void *start_site;
18         char start_comm[16];
19#endif
20 #ifdef CONFIG_LOCKDEP
21         struct lockdep_map lockdep_map;
22#endif
23    };

```

当定时器期满后，其中第 10 行的 function() 成员将被执行，而第 11 行的 data 成员则是传入其中的参数，第 7 行的 expires 则是定时器到期的时间 (jiffies)。

如下代码定义一个名为 my_timer 的定时器：

```
struct timer_list my_timer;
```

2. 初始化定时器

init_timer 是一个宏，它的原型等价于：

```
void init_timer(struct timer_list * timer);
```

上述 init_timer() 函数初始化 timer_list 的 entry 的 next 为 NULL，并给 base 指针赋值。

TIMER_INITIALIZER(_function, _expires, _data) 宏用于赋值定时器结构体的 function、expires、data 和 base 成员，这个宏等价于：

```
#define TIMER_INITIALIZER(_function, _expires, _data) \
    .entry = { .prev = TIMER_ENTRY_STATIC }, \
    .function = (_function), \
    .expires = (_expires), \
    .data = (_data), \
    .base = &boot_tvec_bases,
```

DEFINE_TIMER(_name, _function, _expires, _data) 宏是定义并初始化定时器成员的“快捷方式”，这个宏定义为：

```
#define DEFINE_TIMER(_name, _function, _expires, _data)
```

```
struct timer_list _name =
    TIMER_INITIALIZER(_function, _expires, _data)
```

此外，`setup_timer()`也可用于初始化定时器并赋值其成员，其源代码为：

```
#define __setup_timer(_timer, _fn, _data, _flags)
    do {
        __init_timer((_timer), (_flags));
        (_timer)->function = (_fn);
        (_timer)->data = (_data);
    } while (0)
```

3. 增加定时器

```
void add_timer(struct timer_list * timer);
```

上述函数用于注册内核定时器，将定时器加入到内核动态定时器链表中。

4. 删除定时器

```
int del_timer(struct timer_list * timer);
```

上述函数用于删除定时器。

`del_timer_sync()`是`del_timer()`的同步版，在删除一个定时器时需等待其被处理完，因此该函数的调用不能发生在中断上下文中。

5. 修改定时器的 expire

```
int mod_timer(struct timer_list *timer, unsigned long expires);
```

上述函数用于修改定时器的到期时间，在新的被传入的`expires`到来后才会执行定时器函数。

代码清单 10.10 给出了一个完整的内核定时器使用模板，在大多数情况下，设备驱动都如这个模板那样使用定时器。

代码清单 10.10 内核定时器使用模板

```
1 /* xxx 设备结构体 */
2 struct xxx_dev {
3     struct cdev cdev;
4     ...
5     timer_list xxx_timer;           /* 设备要使用的定时器 */
6 };
7
8 /* xxx 驱动中的某函数 */
9 xxx_func1(...)
10 {
11     struct xxx_dev *dev = filp->private_data;
12     ...
13     /* 初始化定时器 */
14     init_timer(&dev->xxx_timer);
```

```

15     dev->xxx_timer.function = &xxx_do_timer;
16     dev->xxx_timer.data = (unsigned long)dev;
17             /* 设备结构体指针作为定时器处理函数参数 */
18     dev->xxx_timer.expires = jiffies + delay;
19     /* 添加(注册)定时器 */
20     add_timer(&dev->xxx_timer);
21     ...
22 }
23
24 /* xxx 驱动中的某函数 */
25 xxx_func2(...)
26 {
27 ...
28     /* 删除定时器 */
29     del_timer(&dev->xxx_timer);
30     ...
31 }
32
33 /* 定时器处理函数 */
34 static void xxx_do_timer(unsigned long arg)
35 {
36     struct xxx_device *dev = (struct xxx_device *)arg;
37     ...
38     /* 调度定时器再执行 */
39     dev->xxx_timer.expires = jiffies + delay;
40     add_timer(&dev->xxx_timer);
41     ...
42 }

```

从代码清单第 18、39 行可以看出，定时器的到期时间往往是在目前 jiffies 的基础上添加一个时延，若为 Hz，则表示延迟 1s。

在定时器处理函数中，在完成相应的工作后，往往会延后 expires 并将定时器再次添加到内核定时器链表中，以便定时器能再次被触发。

此外，Linux 内核支持 tickless 和 NO_HZ 模式后，内核也包含对 hrtimer（高精度定时器）的支持，它可以支持到微秒级别的精度。内核也定义了 hrtimer 结构体，hrtimer_set_expires()、hrtimer_start_expires()、hrtimer_forward_now()、hrtimer_restart() 等类似的 API 来完成 hrtimer 的设置、时间推移以及到期回调。我们可以从 sound/soc/fsl/imx-pcm-fiq.c 中提取出一个使用范例，如代码清单 10.11 所示。

代码清单 10.11 内核高精度定时器 (hrtimer) 使用模板

```

1 static enum hrtimer_restart snd_hrtimer_callback(struct hrtimer *hrt)
2 {
3     ...
4
5     hrtimer_forward_now(hrt, ns_to_ktime(iprtd->poll_time_ns));
6

```

```

7         return HRTIMER_RESTART;
8     }
9
10    static int snd_imx_pcm_trigger(struct snd_pcm_substream *substream, int cmd)
11    {
12        struct snd_pcm_runtime *runtime = substream->runtime;
13        struct imx_pcm_runtime_data *iprtd = runtime->private_data;
14
15        switch (cmd) {
16        case SNDDRV_PCM_TRIGGER_START:
17        case SNDDRV_PCM_TRIGGER_RESUME:
18        case SNDDRV_PCM_TRIGGER_PAUSE_RELEASE:
19            ...
20            hrtimer_start(&iprtd->hrt, ns_to_ktime(iprtd->poll_time_ns),
21                          HRTIMER_MODE_REL);
22            ...
23    }
24
25    static int snd_imx_open(struct snd_pcm_substream *substream)
26    {
27        ...
28        hrtimer_init(&iprtd->hrt, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
29        iprtd->hrt.function = snd_hrtimer_callback;
30
31        ...
32        return 0;
33    }
34    static int snd_imx_close(struct snd_pcm_substream *substream)
35    {
36        ...
37        hrtimer_cancel(&iprtd->hrt);
38        ...
39    }

```

第 28 ~ 29 行在声卡打开的时候通过 `hrtimer_init()` 初始化了 `hrtimer`，并指定回调函数为 `snd_hrtimer_callback()`；在启动播放（第 15 ~ 21 行 `SNDDRV_PCM_TRIGGER_START`）等时刻通过 `hrtimer_start()` 启动了 `hrtimer`；`iprtd->poll_time_ns` 纳秒后，时间到 `snd_hrtimer_callback()` 函数在中断上下文被执行，它紧接着又通过 `hrtimer_forward_now()` 把 `hrtimer` 的时间前移了 `iprtd->poll_time_ns` 纳秒，这样周而复始；直到声卡被关闭，第 37 行又调用了 `hrtimer_cancel()` 取消在 open 时初始化的 `hrtimer`。

10.5.2 内核中延迟的工作 `delayed_work`

对于周期性的任务，除了定时器以外，在 Linux 内核中还可以利用一套封装得很好的快捷机制，其本质是利用工作队列和定时器实现，这套快捷机制就是 `delayed_work`，`delayed_work` 结构体的定义如代码清单 10.12 所示。

代码清单 10.12 delayed_work 结构体

```

1 struct delayed_work {
2     struct work_struct work;
3     struct timer_list timer;
4
5     /* target workqueue and CPU ->timer uses to queue ->work */
6     struct workqueue_struct *wq;
7     int cpu;
8 };

```

我们可以通过如下函数调度一个 delayed_work 在指定的延时后执行：

```
int schedule_delayed_work(struct delayed_work *work, unsigned long delay);
```

当指定的 delay 到来时，delayed_work 结构体中的 work 成员 work_func_t 类型成员 func() 会被执行。work_func_t 类型定义为：

```
typedef void (*work_func_t)(struct work_struct *work);
```

其中，delay 参数的单位是 jiffies，因此一种常见的用法如下：

```
schedule_delayed_work(&work, msecs_to_jiffies(poll_interval));
```

msecs_to_jiffies() 用于将毫秒转化为 jiffies。

如果要周期性地执行任务，通常会在 delayed_work 的工作函数中再次调用 schedule_delayed_work()，周而复始。

如下函数用来取消 delayed_work：

```
int cancel_delayed_work(struct delayed_work *work);
int cancel_delayed_work_sync(struct delayed_work *work);
```

10.5.3 实例：秒字符设备

下面我们编写一个字符设备“second”（即“秒”）的驱动，它在被打开的时候初始化一个定时器并将其添加到内核定时器链表中，每秒输出一次当前的 jiffies（为此，定时器处理函数中每次都要修改新的 expires），整个程序如代码清单 10.13 所示。

代码清单 10.13 使用内核定时器的 second 字符设备驱动

```

1 #include <linux/module.h>
2 #include <linux/fs.h>
3 #include <linux/mmc.h>
4 #include <linux/init.h>
5 #include <linux/cdev.h>
6 #include <linux/slab.h>
7 #include <linux/uaccess.h>
8
9 #define SECOND_MAJOR 248

```

```

10
11 static int second_major = SECOND_MAJOR;
12 module_param(second_major, int, S_IRUGO);
13
14 struct second_dev {
15     struct cdev cdev;
16     atomic_t counter;
17     struct timer_list s_timer;
18 };
19
20 static struct second_dev *second_devp;
21
22 static void second_timer_handler(unsigned long arg)
23 {
24     mod_timer(&second_devp->s_timer, jiffies + HZ);      /* 触发下一次定时 */
25     atomic_inc(&second_devp->counter);                      /* 增加秒计数 */
26
27     printk(KERN_INFO "current jiffies is %ld\n", jiffies);
28 }
29
30 static int second_open(struct inode *inode, struct file *filp)
31 {
32     init_timer(&second_devp->s_timer);
33     second_devp->s_timer.function = &second_timer_handler;
34     second_devp->s_timer.expires = jiffies + HZ;
35
36     add_timer(&second_devp->s_timer);
37
38     atomic_set(&second_devp->counter, 0);                  /* 初始化秒计数为 0 */
39
40     return 0;
41 }
42
43 static int second_release(struct inode *inode, struct file *filp)
44 {
45     del_timer(&second_devp->s_timer);
46
47     return 0;
48 }
49
50 static ssize_t second_read(struct file *filp, char __user *buf, size_t count,
51   loff_t * ppos)
52 {
53     int counter;
54
55     counter = atomic_read(&second_devp->counter);
56     if (put_user(counter, (int *)buf))                         /* 复制 counter 到 userspace */
57         return -EFAULT;
58     else
59         return sizeof(unsigned int);
60 }

```

```
61
62 static const struct file_operations second_fops = {
63     .owner = THIS_MODULE,
64     .open = second_open,
65     .release = second_release,
66     .read = second_read,
67 };
68
69 static void second_setup_cdev(struct second_dev *dev, int index)
70 {
71     int err, devno = MKDEV(second_major, index);
72
73     cdev_init(&dev->cdev, &second_fops);
74     dev->cdev.owner = THIS_MODULE;
75     err = cdev_add(&dev->cdev, devno, 1);
76     if (err)
77         printk(KERN_ERR "Failed to add second device\n");
78 }
79
80 static int __init second_init(void)
81 {
82     int ret;
83     dev_t devno = MKDEV(second_major, 0);
84
85     if (second_major)
86         ret = register_chrdev_region(devno, 1, "second");
87     else {
88         ret = alloc_chrdev_region(&devno, 0, 1, "second");
89         second_major = MAJOR(devno);
90     }
91     if (ret < 0)
92         return ret;
93
94     second_devp = kzalloc(sizeof(*second_devp), GFP_KERNEL);
95     if (!second_devp) {
96         ret = -ENOMEM;
97         goto fail_malloc;
98     }
99
100    second_setup_cdev(second_devp, 0);
101
102    return 0;
103
104 fail_malloc:
105     unregister_chrdev_region(devno, 1);
106     return ret;
107 }
108 module_init(second_init);
109
110 static void __exit second_exit(void)
```

```

111 {
112     cdev_del(&second_devp->cdev);
113     kfree(second_devp);
114     unregister_chrdev_region(MKDEV(second_major, 0), 1);
115 }
116 module_exit(second_exit);
117
118 MODULE_AUTHOR("Barry Song <21cnbao@gmail.com>");
119 MODULE_LICENSE("GPL v2");

```

在 second 的 open() 函数中，将启动定时器，此后每 1s 会再次运行定时器处理函数，在 second 的 release() 函数中，定时器被删除。

second_dev 结构体中的原子变量 counter 用于秒计数，每次在定时器处理函数中调用的 atomic_inc() 会令其原子性地增 1，second 的 read() 函数会将这个值返回给用户空间。

本书配套的 Ubuntu 中 /home/baohua/develop/training/kernel/drivers/second/ 包含了 second 设备驱动以及 second_test.c 用户空间测试程序，运行 make 命令编译得到 second.ko 和 second_test，加载 second.ko 内核模块并创建 /dev/second 设备文件节点：

```
# mknod /dev/second c 248 0
```

代码清单 10.14 给出了 second_test.c 这个应用程序，它打开 /dev/second，其后不断地读取自 /dev/second 设备文件打开以后经历的秒数。

代码清单 10.14 second 设备用户空间测试程序

```

1 #include ...
2
3 main()
4 {
5     int fd;
6     int counter = 0;
7     int old_counter = 0;
8
9     /* 打开 /dev/second 设备文件 */
10    fd = open("/dev/second", O_RDONLY);
11    if (fd != -1) {
13        while (1) {
15            read(fd, &counter, sizeof(unsigned int)); /* 读目前经历的秒数 */
16            if(counter!=old_counter) {
18                printf("seconds after open /dev/second :%d\n",counter);
19                old_counter = counter;
20            }
21        }
22    } else {
25        printf("Device open failure\n");
26    }
27 }

```

运行 second_test 后，内核将不断地输出目前的 jiffies 值：

```
[13935.122093] current jiffies is 13635122
[13936.124441] current jiffies is 13636124
[13937.126078] current jiffies is 13637126
[13952.832648] current jiffies is 13652832
[13953.834078] current jiffies is 13653834
[13954.836090] current jiffies is 13654836
[13955.838389] current jiffies is 13655838
[13956.840453] current jiffies is 13656840
...
```

从上述内核的打印消息也可以看出，本书配套 Ubuntu 上的每秒 jiffies 大概走 1000 次。而应用程序将不断输出自 /dev/second 打开以后经历的秒数：

```
# ./second_test
seconds after open /dev/second :1
seconds after open /dev/second :2
seconds after open /dev/second :3
seconds after open /dev/second :4
seconds after open /dev/second :5
...

```

10.6 内核延时

10.6.1 短延迟

Linux 内核中提供了下列 3 个函数以分别进行纳秒、微秒和毫秒延迟：

```
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

上述延迟的实现原理本质上是忙等待，它根据 CPU 频率进行一定次数的循环。有时候，人们在软件中进行下面的延迟：

```
void delay(unsigned int time)
{
    while(time--);
}
```

ndelay()、udelay() 和 mdelay() 函数的实现方式原理与此类似。内核在启动时，会运行一个延迟循环校准（Delay Loop Calibration），计算出 lpj（Loops Per Jiffy），内核启动时会打印如下类似信息：

```
Calibrating delay loop... 530.84 BogoMIPS (lpj=1327104)
```

如果我们直接在 bootloader 传递给内核的 bootargs 中设置 lpj=1327104，则可以省掉这个

校准的过程，节省约百毫秒级的开机时间。

毫秒时延（以及更大的秒时延）已经比较大了，在内核中，最好不要直接使用 mdelay() 函数，这将耗费 CPU 资源，对于毫秒级以上时延，内核提供了下述函数：

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds);
```

上述函数将使得调用它的进程睡眠参数指定的时间为 millisecs，msleep()、ssleep() 不能被打断，而 msleep_interruptible() 则可以被打断。



受系统 Hz 以及进程调度的影响，msleep() 类似函数的精度是有限的。

10.6.2 长延迟

在内核中进行延迟的一个很直观的方法是比较当前的 jiffies 和目标 jiffies（设置为当前 jiffies 加上时间间隔的 jiffies），直到未来的 jiffies 达到目标 jiffies。代码清单 10.15 给出了使用忙等待先延迟 100 个 jiffies 再延迟 2s 的实例。

代码清单 10.15 忙等待时延实例

```
1 /* 延迟 100 个 jiffies */
2 unsigned long delay = jiffies + 100;
3 while(time_before(jiffies, delay));
4
5 /* 再延迟 2s */
6 unsigned long delay = jiffies + 2*Hz;
7 while(time_before(jiffies, delay));
```

与 time_before() 对应的还有一个 time_after()，它们在内核中定义为（实际上只是将传入的未来时间 jiffies 和被调用时的 jiffies 进行一个简单的比较）：

```
#define time_after(a,b)          \
    (typecheck(unsigned long, a) && \
     typecheck(unsigned long, b) && \
     ((long)(b) - (long)(a) < 0))
#define time_before(a,b)   time_after(b,a)
```

为了防止在 time_before() 和 time_after() 的比较过程中编译器对 jiffies 的优化，内核将其定义为 volatile 变量，这将保证每次都会重新读取这个变量。因此 volatile 更多的作用还是避免这种读合并。

10.6.3 睡着延迟

睡着延迟无疑是比忙等待更好的方式，睡着延迟是在等待的时间到来之前进程处于睡眠

状态，CPU 资源被其他进程使用。schedule_timeout() 可以使当前任务休眠至指定的 jiffies 之后再重新被调度执行，msleep() 和 msleep_interruptible() 在本质上都是依靠包含了 schedule_timeout() 的 schedule_timeout_uninterruptible() 和 schedule_timeout_interruptible() 来实现的，如代码清单 10.16 所示。

代码清单 10.16 schedule_timeout() 的使用

```

1 void msleep(unsigned int msecs)
2 {
3     unsigned long timeout = msecs_to_jiffies(msecs) + 1;
4
5     while (timeout)
6         timeout = schedule_timeout_uninterruptible(timeout);
7 }
8
9 unsigned long msleep_interruptible(unsigned int msecs)
10 {
11     unsigned long timeout = msecs_to_jiffies(msecs) + 1;
12
13     while (timeout && !signal_pending(current))
14         timeout = schedule_timeout_interruptible(timeout);
15     return jiffies_to_msecs(timeout);
16 }
```

实际上，schedule_timeout() 的实现原理是向系统添加一个定时器，在定时器处理函数中唤醒与参数对应的进程。

代码清单 10.16 中第 6 行和第 14 行分别调用 schedule_timeout_uninterruptible() 和 schedule_timeout_interruptible()，这两个函数的区别在于前者在调用 schedule_timeout() 之前置进程状态为 TASK_INTERRUPTIBLE，后者置进程状态为 TASK_UNINTERRUPTIBLE，如代码清单 10.17 所示。

代码清单 10.17 schedule_timeout_interruptible() 和 schedule_timeout_uninterruptible()

```

1 signed long __sched schedule_timeout_interruptible(signed long timeout)
2 {
3     __set_current_state(TASK_INTERRUPTIBLE);
4     return schedule_timeout(timeout);
5 }
6
7 signed long __sched schedule_timeout_uninterruptible(signed long timeout)
8 {
9     __set_current_state(TASK_UNINTERRUPTIBLE);
10    return schedule_timeout(timeout);
11 }
```

另外，下面两个函数可以将当前进程添加到等待队列中，从而在等待队列上睡眠。当超

时发生时，进程将被唤醒（后者可以在超时前被打断）：

```
sleep_on_timeout(wait_queue_head_t *q, unsigned long timeout);
interruptible_sleep_on_timeout(wait_queue_head_t*q, unsigned long timeout);
```

10.7 总结

Linux 的中断处理分为两个半部，顶半部处理紧急的硬件操作，底半部处理不紧急的耗时操作。tasklet 和工作队列都是调度中断底半部的良好机制，tasklet 基于软中断实现。内核定时器也依靠软中断实现。

内核中的延时可以采用忙等待或睡眠等待，为了充分利用 CPU 资源，使系统有更好的吞吐性能，在对延迟时间的要求并不是很精确的情况下，睡眠等待通常是值得推荐的，而 `ndelay()`、`udelay()` 忙等待机制在驱动中通常是为了配合硬件上的短时延迟要求。

第 11 章

内存与 I/O 访问

本章导读

由于 Linux 系统提供了复杂的内存管理功能，所以内存的概念在 Linux 系统中相对复杂，有常规内存、高端内存、虚拟地址、逻辑地址、总线地址、物理地址、I/O 内存、设备内存、预留内存等概念。本章将系统地讲解内存和 I/O 的访问编程，带读者走出内存和 I/O 的概念迷宫。

11.1 节讲解内存和 I/O 的硬件机制，主要涉及内存空间、I/O 空间和 MMU。

11.2 节讲解 Linux 的内存管理、内存区域的分布、常规内存与高端内存的区别。

11.3 节讲解 Linux 内存存取的方法，主要涉及内存动态申请以及通过虚拟地址存取物理地址的方法。

11.4 节讲解设备 I/O 端口和 I/O 内存的访问流程，这一节对于编写设备驱动的意义非常重大，设备驱动使用此节的方法访问物理设备。

11.5 节讲解 I/O 内存静态映射。

11.6 节讲解设备驱动中的 DMA 与 Cache 一致性问题以及 DMA 的编程方法。

11.1 CPU 与内存、I/O

11.1.1 内存空间与 I/O 空间

在 X86 处理器中存在着 I/O 空间的概念，I/O 空间是相对于内存空间而言的，它通过特定的指令 in、out 来访问。端口号标识了外设的寄存器地址。Intel 语法中的 in、out 指令格式如下：

```
IN 累加器, {端口号 | DX}  
OUT {端口号 | DX}, 累加器
```

目前，大多数嵌入式微控制器（如 ARM、PowerPC 等）中并不提供 I/O 空间，而仅存在内存空间。内存空间可以直接通过地址、指针来访问，程序及在程序运行中使用的变量和其他数据都存在于内存空间中。

内存地址可以直接由 C 语言指针操作，例如在 i86 处理器中执行如下代码：

```
unsigned char *p = (unsigned char *)0xF000FF00;
*p=11;
```

以上程序的意义是在绝对地址 0xF0000+0xFF00（186 处理器使用 16 位段地址和 16 位偏移地址）中写入 11。

而在 ARM、PowerPC 等未采用段地址的处理器中，p 指向的内存空间就是 0xF000FF00，而 *p = 11 就是在该地址写入 11。

再如，186 处理器启动后会在绝对地址 0xFFFF0（对应的 C 语言指针是 0xF000FFF0，0xF000 为段地址，0xFFF0 为段内偏移）中执行，请看下面的代码：

```
typedef void (*lpFunction) ();
lpFunction lpReset = (lpFunction)0xF000FFF0; /* 定义一个函数指针，指向 */
/* CPU 启动后所执行的第一条指令的位置 */
lpReset(); /* 调用函数 */
```

在以上程序中，没有定义任何一个函数实体，但是程序却执行了这样的函数调用：lpReset()，它实际上起到了“软重启”的作用，跳转到 CPU 启动后第一条要执行的指令的位置。因此，可以通过函数指针调用一个没有函数体的“函数”，这本质上只是换一个地址开始执行。

即便是在 X86 处理器中，虽然提供了 I/O 空间，如果由我们自己设计电路板，外设仍然可以只挂接在内存空间中。此时，CPU 可以像访问一个内存单元那样访问外设 I/O 端口，而不需要设立专门的 I/O 指令。因此，内存空间是必需的，而 I/O 空间是可选的。图 11.1 给出了内存空间和 I/O 空间的对比。

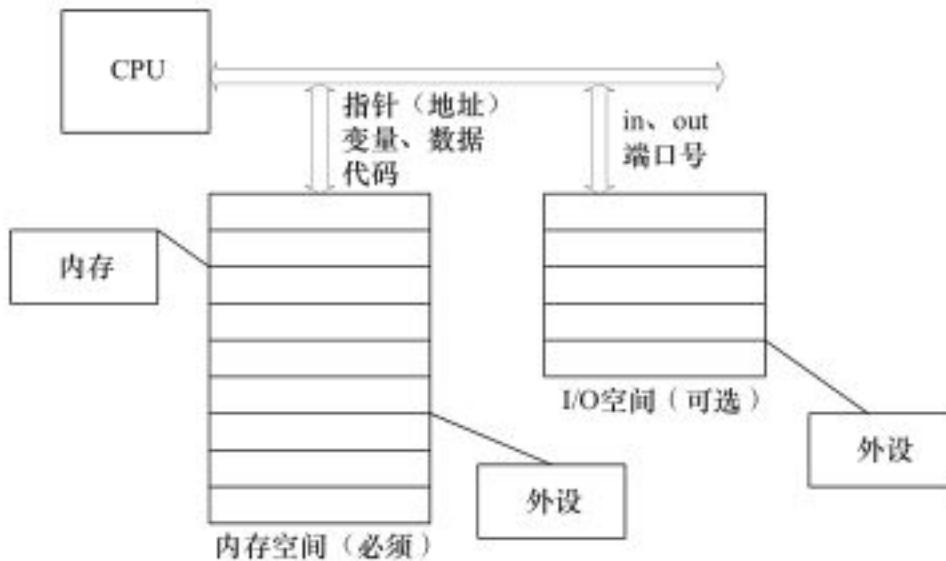


图 11.1 内存空间和 I/O 空间

11.1.2 内存管理单元

高性能处理器一般会提供一个内存管理单元（MMU），该单元辅助操作系统进行内存管

理，提供虚拟地址和物理地址的映射、内存访问权限保护和 Cache 缓存控制等硬件支持。操作系统内核借助 MMU 可以让用户感觉到程序好像可以使用非常大的内存空间，从而使得编程人员在写程序时不用考虑计算机中物理内存的实际容量。

为了理解基本的 MMU 操作原理，需先明晰几个概念。

1) TLB (Translation Lookaside Buffer)：即转换旁路缓存，TLB 是 MMU 的核心部件，它缓存少量的虚拟地址与物理地址的转换关系，是转换表的 Cache，因此也经常被称为“快表”。

2) TTW (Translation Table walk)：即转换表漫游，当 TLB 中没有缓冲对应的地址转换关系时，需要通过对内存中转换表（大多数处理器的转换表为多级页表，如图 11.2 所示）的访问来获得虚拟地址和物理地址的对应关系。TTW 成功后，结果应写入 TLB 中。

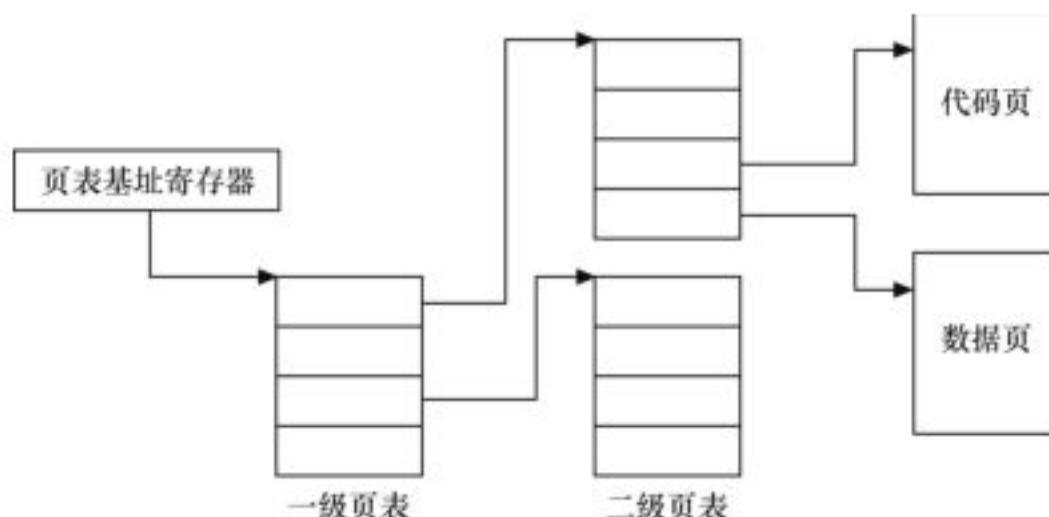


图 11.2 内存中的转换表

图 11.3 给出了一个典型的 ARM 处理器访问内存的过程，其他处理器也执行类似过程。当 ARM 要访问存储器时，MMU 先查找 TLB 中的虚拟地址表。如果 ARM 的结构支持分开的数据 TLB (DTLB) 和指令 TLB (ITLB)，则除了取指令使用 ITLB 外，其他的都使用 DTLB。ARM 处理器的 MMU 如图 11.3 所示。

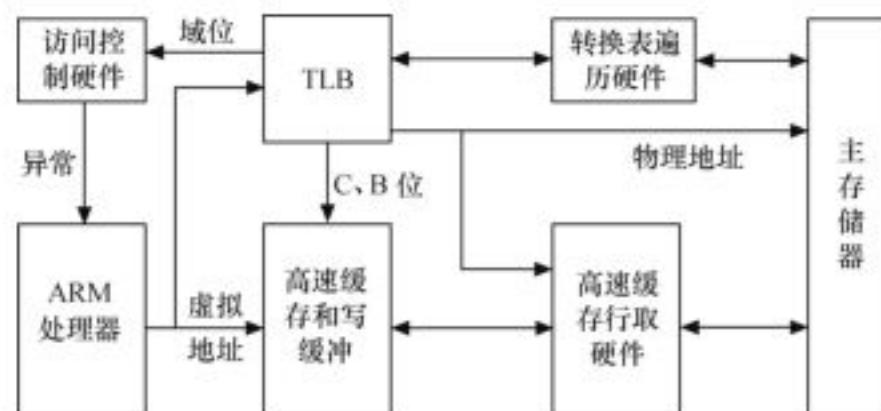


图 11.3 ARM 处理器的 MMU

若 TLB 中没有虚拟地址的入口，则转换表遍历硬件并从存放于主存储器内的转换表中获取地址转换信息和访问权限（即执行 TTW），同时将这些信息放入 TLB，它或者被放在一个没有使用的人口或者替换一个已经存在的人口。之后，在 TLB 条目中控制信息的控制下，当访问权限允许时，对真实物理地址的访问将在 Cache 或者在内存中发生，如图 11.4 所示。

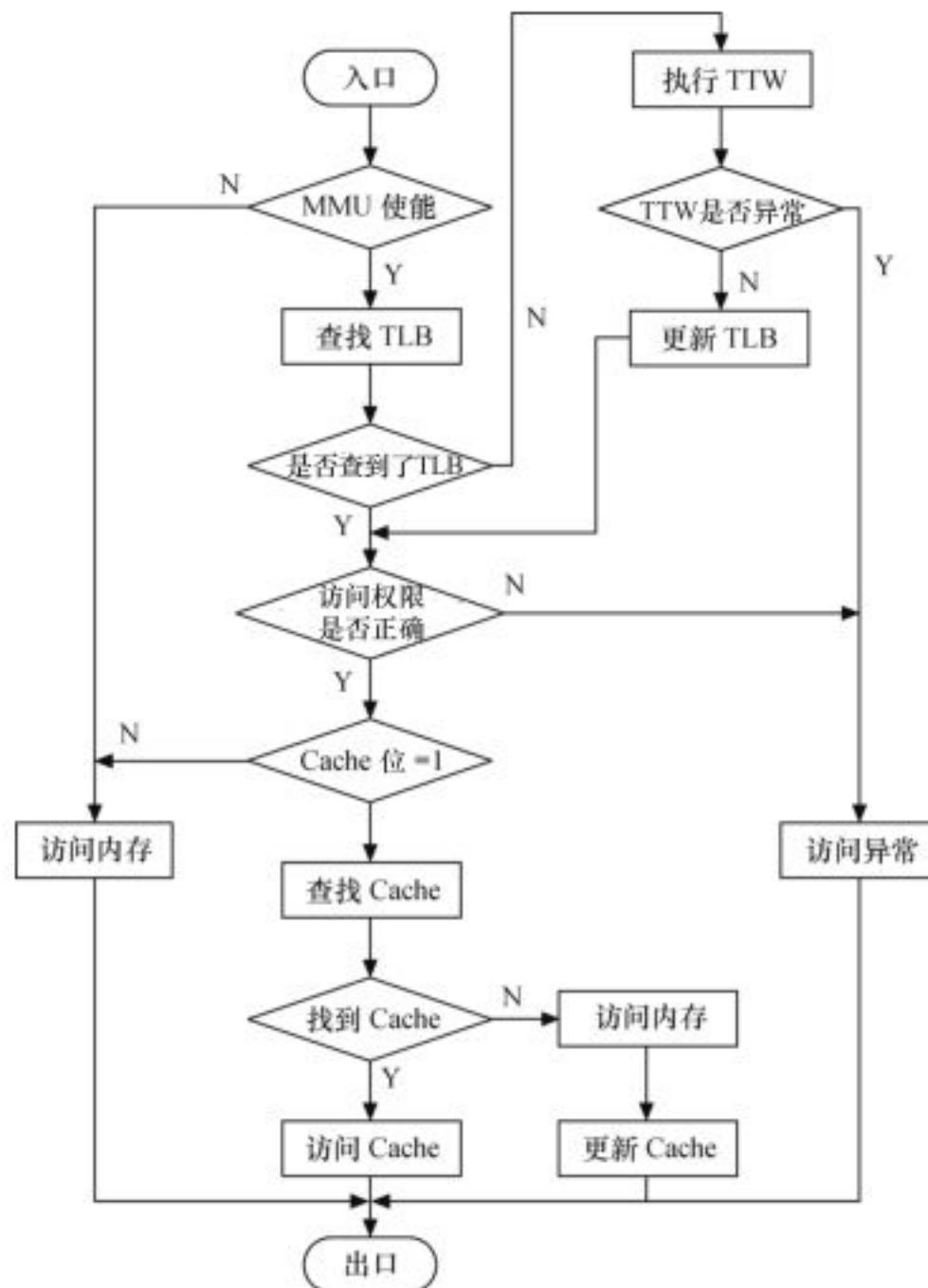


图 11.4 ARM CPU 进行数据访问的流程

ARM 内 TLB 条目中的控制信息用于控制对对应地址的访问权限以及 Cache 的操作。

- C（高速缓存）和 B（缓冲）位被用来控制对应地址的高速缓存和写缓冲，并决定是否进行高速缓存。
- 访问权限和域位用来控制读写访问是否被允许。如果不允许，MMU 则向 ARM 处理器发送一个存储器异常，否则访问将被允许进行。

上述描述的 MMU 机制针对的虽然是 ARM 处理器，但 PowerPC、MIPS 等其他处理器也均有类似的操作。

MMU 具有虚拟地址和物理地址转换、内存访问权限保护等功能，这将使得 Linux 操作系统能单独为系统的每个用户进程分配独立的内存空间并保证用户空间不能访问内核空间的地址，为操作系统的虚拟内存管理模块提供硬件基础。

在 Linux 2.6.11 之前，Linux 内核硬件无关层使用了三级页表 PGD、PMD 和 PTE；从 Linux 2.6.11 开始，为了配合 64 位 CPU 的体系结构，硬件无关层则使用了 4 级页表目录管理的方式，即 PGD、PUD、PMD 和 PTE。注意这仅仅是一种软件意义上的抽象，实际硬件的页表级数可能少于 4。代码清单 11.1 给出了一个典型的从虚拟地址得到 PTE 的页表查询（Page Table Walk）过程，它取自 arch/arm/lib/uaccess_with_memcpy.c。

代码清单 11.1 Linux 的四级页表与页表查询

```

1 static int
2 pin_page_for_write(const void __user *_addr, pte_t **ptep, spinlock_t **ptlp)
3 {
4     unsigned long addr = (unsigned long)_addr;
5     pgd_t *pgd;
6     pmd_t *pmd;
7     pte_t *pte;
8     pud_t *pud;
9     spinlock_t *ptl;
10
11     pgd = pgd_offset(current->mm, addr);
12     if (unlikely(pgd_none(*pgd) || pgd_bad(*pgd)))
13         return 0;
14
15     pud = pud_offset(pgd, addr);
16     if (unlikely(pud_none(*pud) || pud_bad(*pud)))
17         return 0;
18
19     pmd = pmd_offset(pud, addr);
20     if (unlikely(pmd_none(*pmd)))
21         return 0;
22
23     /*
24      * A pmd can be bad if it refers to a HugeTLB or THP page.
25      *
26      * Both THP and HugeTLB pages have the same pmd layout
27      * and should not be manipulated by the pte functions.
28      *
29      * Lock the page table for the destination and check
30      * to see that it's still huge and whether or not we will
31      * need to fault on write, or if we have a splitting THP.
32      */
33     if (unlikely(pmd_thp_or_huge(*pmd))) {
34         ptl = &current->mm->page_table_lock;

```

```

35     spin_lock(ptl);
36     if (unlikely(!pmd_thp_or_huge(*pmd)
37         || pmd_hugewillfault(*pmd)
38         || pmd_trans_splitting(*pmd))) {
39         spin_unlock(ptl);
40         return 0;
41     }
42
43     *ptep = NULL;
44     *ptlp = ptl;
45     return 1;
46 }
47
48 if (unlikely(pmd_bad(*pmd)))
49     return 0;
50
51 pte = pte_offset_map_lock(current->mm, pmd, addr, &ptl);
52 if (unlikely(!pte_present(*pte) || !pte_young(*pte) ||
53     !pte_write(*pte) || !pte_dirty(*pte))) {
54     pte_unmap_unlock(pte, ptl);
55     return 0;
56 }
57
58 *ptep = pte;
59 *ptlp = ptl;
60
61 return 1;
62 }

```

第1行的类型为 struct_mm_struct 的参数 mm 用于描述 Linux 进程所占有的内存资源。上述代码中的 pgd_offset、pud_offset、pmd_offset 分别用于得到一级页表、二级页表和三级页表的入口，最后通过 pte_offset_map_lock 得到目标页表项 pte。而且第33行还通过 pmd_thp_or_huge() 判断是否有巨页的情况，如果是巨页，就直接访问 pmd。

但是，MMU 并不是对所有的处理器都是必需的，例如常用的 SAMSUNG 基于 ARM7TDMI 系列的 S3C44B0X 不附带 MMU，新版的 Linux 2.6 支持不带 MMU 的处理器。在嵌入式系统中，仍存在大量无 MMU 的处理器，Linux 2.6 为了更广泛地应用于嵌入式系统，融合了 mClinix，以支持这些无 MMU 系统，如 Dragonball、ColdFire、Hitachi H8/300、Blackfin 等。

11.2 Linux 内存管理

对于包含 MMU 的处理器而言，Linux 系统提供了复杂的存储管理系统，使得进程所能访问的内存达到 4GB。

在 Linux 系统中，进程的 4GB 内存空间被分为两个部分——用户空间与内核空间。用户

空间的地址一般分布为 0 ~ 3GB(即 PAGE_OFFSET，在 0x86 中它等于 0xC0000000)，这样，剩下的 3 ~ 4GB 为内核空间，如图 11.5 所示。用户进程通常只能访问用户空间的虚拟地址，不能访问内核空间的虚拟地址。用户进程只有通过系统调用（代表用户进程在内核态执行）等方式才可以访问到内核空间。

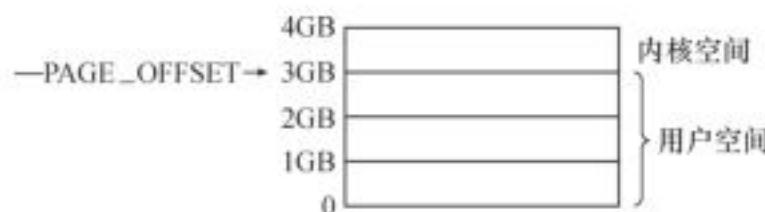


图 11.5 用户空间与内核空间

每个进程的用户空间都是完全独立、互不相干的，用户进程各自有不同的页表。而内核空间是由内核负责映射，它并不会跟着进程改变，是固定的。内核空间的虚拟地址到物理地址映射是被所有进程共享的，内核的虚拟空间独立于其他程序。

Linux 中 1GB 的内核地址空间又被划分为物理内存映射区、虚拟内存分配区、高端页面映射区、专用页面映射区和系统保留映射区这几个区域，如图 11.6 所示。

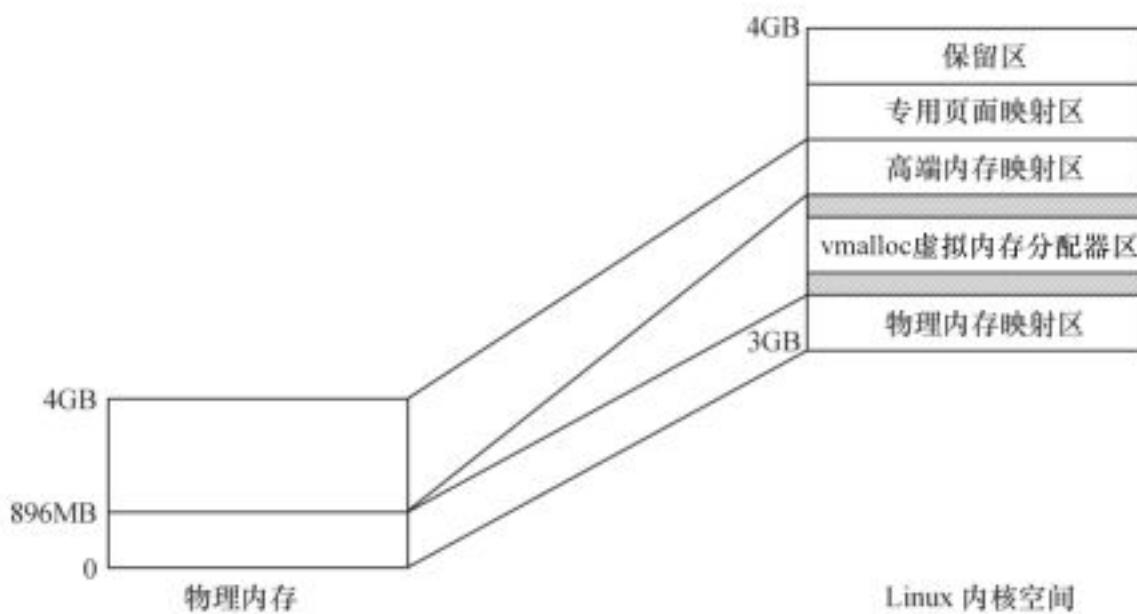


图 11.6 32 位 x86 系统 Linux 内核的地址空间

对于 x86 系统而言，一般情况下，物理内存映射区最大长度为 896MB，系统的物理内存被顺序映射在内核空间的这个区域中。当系统物理内存大于 896MB 时，超过物理内存映射区的那部分内存称为高端内存（而未超过物理内存映射区的内存通常被称为常规内存），内核在存取高端内存时必须将它们映射到高端页面映射区。

Linux 保留内核空间最顶部 FIXADDR_TOP ~ 4GB 的区域作为保留区。

紧接着最顶端的保留区以下的一段区域为专用页面映射区 (FIXADDR_START ~ FIXADDR_TOP)，它的总尺寸和每一页的用途由 fixed_address 枚举结构在编译时预定义，用

`_fix_to_virt(index)` 可获取专用区内预定义页面的逻辑地址。其开始地址和结束地址宏定义如下：

```
#define FIXADDR_START          (FIXADDR_TOP - __FIXADDR_SIZE)
#define FIXADDR_TOP             ((unsigned long) __FIXADDR_TOP)
#define __FIXADDR_TOP            0xfffffff000
```

接下来，如果系统配置了高端内存，则位于专用页面映射区之下的一段高端内存映射区，其起始地址为 `PKMAP_BASE`，定义如下：

```
#define PKMAP_BASE ( (FIXADDR_BOOT_START - PAGE_SIZE*(LAST_PKMAP + 1)) & PMD_MASK )
```

其中所涉及的宏定义如下：

```
#define FIXADDR_BOOT_START      (FIXADDR_TOP - __FIXADDR_BOOT_SIZE)
#define LAST_PKMAP    PTRS_PER_PTE
#define PTRS_PER_PTE   512
#define PMD_MASK      (~(PMD_SIZE-1))
#define PMD_SIZE      (1UL << PMD_SHIFT)
#define PMD_SHIFT     21
```

在物理区和高端映射区之间为虚拟内存分配器区 (`VMALLOC_START ~ VMALLOC_END`)，用于 `vmalloc()` 函数，它的前部与物理内存映射区有一个隔离带，后部与高端映射区也有一个隔离带，`vmalloc` 区域定义如下：

```
#define VMALLOC_OFFSET (8*1024*1024)
#define VMALLOC_START    (((unsigned long) high_memory +
                      vmalloc_earlyreserve + 2*VMALLOC_OFFSET-1) & ~(VMALLOC_OFFSET-1))

#ifndef CONFIG_HIGHMEM           /* 支持高端内存 */
#define VMALLOC_END    (PKMAP_BASE-2*PAGE_SIZE)
#else                           /* 不支持高端内存 */
#define VMALLOC_END    (FIXADDR_START-2*PAGE_SIZE)
#endif
```

当系统物理内存超过 4GB 时，必须使用 CPU 的扩展分页 (PAE) 模式所提供的 64 位页目录项才能存取到 4GB 以上的物理内存，这需要 CPU 的支持。加入了 PAE 功能的 Intel Pentium Pro 及以后的 CPU 允许内存最大可配置到 64GB，它们具备 36 位物理地址空间寻址能力。

由此可见，对于 32 位的 x86 而言，在 3 ~ 4GB 之间的内核空间中，从低地址到高地址依次为：物理内存映射区 → 隔离带 → `vmalloc` 虚拟内存分配器区 → 隔离带 → 高端内存映射区 → 专用页面映射区 → 保留区。

直接进行映射的 896MB 物理内存其实又分为两个区域，在低于 16MB 的区域，ISA 设备可以做 DMA，所以该区域为 DMA 区域（内核为了保证 ISA 驱动在申请 DMA 缓冲区的时候，通过 `GFP_DMA` 标记可以确保申请到 16MB 以内的内存，所以必须把这个区域列为一个

单独的区域管理)；16MB ~ 896MB之间的为常规区域。高于896MB的就称为高端内存区域了。

32位ARM Linux的内核空间地址映射与x86不太一样，内核文档Documentation/arm/memory.txt给出了ARM Linux的内存映射情况。0xffff0000 ~ 0xffff0fff是“CPU vector page”，即向量表的地址。0xffc00000 ~ 0xffefffff是DMA内存映射区域，dma_alloc_xxx族函数把DMA缓冲区映射在这一段，VMALLOC_START ~ VMALLOC_END-1是vmalloc和ioremap区域(在vmalloc区域的大小可以配置，通过“vmalloc=”这个启动参数可以指定)，PAGE_OFFSET ~ high_memory-1是DMA和正常区域的映射区域，MODULES_VADDR ~ MODULES_END-1是内核模块区域，PKMAP_BASE ~ PAGE_OFFSET-1是高端内存映射区。假设我们把PAGE_OFFSET定义为3GB，实际上Linux内核模块位于3GB-16MB ~ 3GB-2MB，高端内存映射区则通常位于3GB-2MB ~ 3GB。

图11.7给出了32位ARM系统Linux内核地址空间中的内核模块区域、高端内存映射区、vmalloc、向量表区域等。我们假定编译内核的时候选择的是VMSPLIT_3G(3G/1G user/kernel split)。如果用户选择的是VMSPLIT_2G(2G/2G user/kernel split)，则图11.7中的内核模块开始于2GB-16MB，DMA和常规内存区域映射区也开始于2GB。

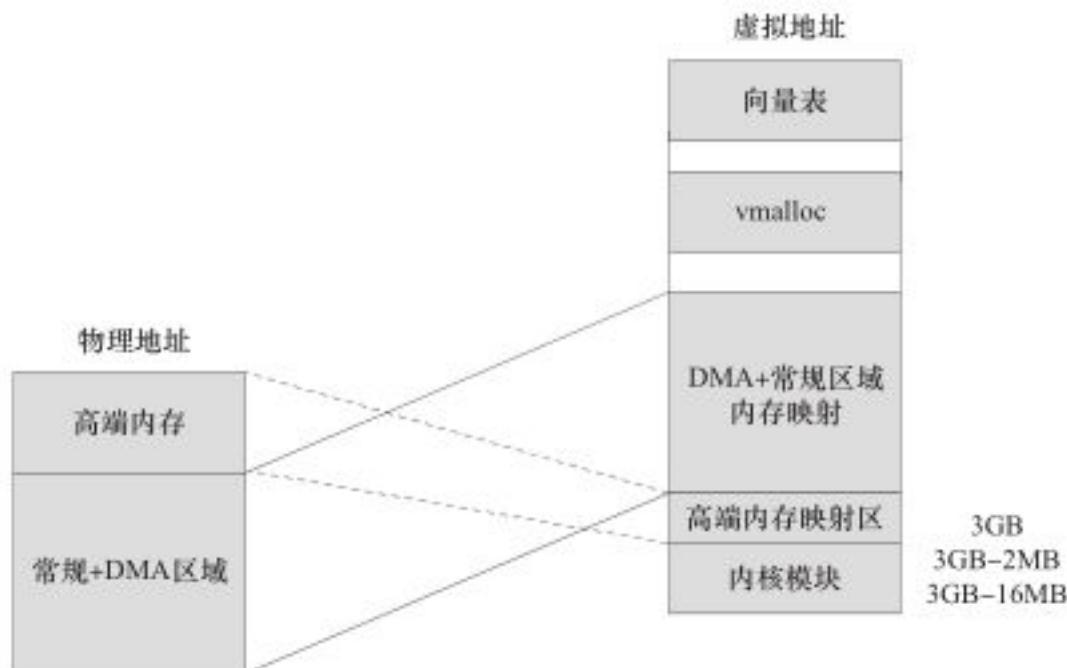


图11.7 32位ARM系统中Linux内核的地址空间

ARM系统的Linux之所以把内核模块安置在3GB或者2GB附近的16MB范围内，主要是为了实现内核模块和内核本身的代码段之间的短跳转。

对于ARM SoC而言，如果芯片内部有的硬件组件的DMA引擎访问内存时有地址空间限制(某些空间访问不到)，比如假设UART控制器的DMA只能访问32MB，那么这个低32MB就是DMA区域；32MB到高端内存地址的这段称为常规区域；再之上的称为高端内存区域。

图 11.8 给出了几种 DMA、常规、高端内存区域可能的分布，在第一种情况下，有硬件的 DMA 引擎不能访问全部地址，且内存较大而无法全部在内核空间虚拟地址映射下，存放有 3 个区域；第二种情况下，没有硬件的 DMA 引擎不能访问全部地址，且内存较大而无法全部在内核空间虚拟地址映射下，则常规区域实际退化为 0；第三种情况下，有硬件的 DMA 引擎不能访问全部地址，且内存较小可以全部在内核空间虚拟地址映射下，则高端内存区域实际退化为 0；第四种情况下，没有硬件的 DMA 引擎不能访问全部地址，且内存较小可以全部在内核空间虚拟地址映射下，则常规和高端内存区域实际退化为 0。



图 11.8 DMA、常规、高端内存区域分布

如图 11.9 所示，DMA、常规、高端内存这 3 个区域都采用 buddy 算法进行管理，把空闲的页面以 2^n 次方为单位进行管理，因此 Linux 最底层的内存申请都是以 2^n 为单位的。Buddy 算法最主要的优点是避免了外部碎片，任何时候区域里的空闲内存都能以 2^n 次方进行拆分或合并。

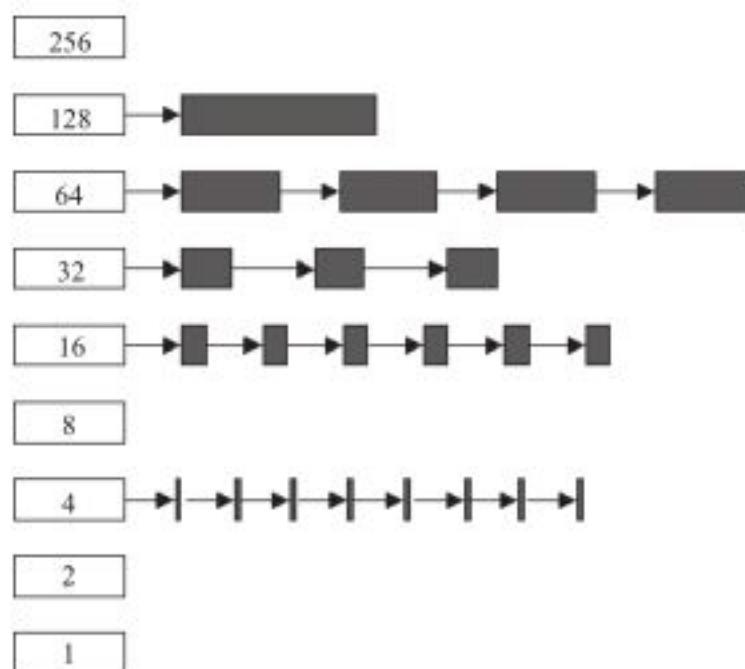


图 11.9 buddy 算法

/proc/buddyinfo 会显示每个区域里面 2^n 的空闲页面分布情况，比如：

```
$ cat /proc/buddyinfo
Node 0, zone      DMA     8      5      2      7      8      3      0      0
0       1       0
Node 0, zone    Normal   2002   1252    524    187    183    71      7      0
0       1       1
```

上述结果显示高端内存区域为 0，DMA 区域里 1 页空闲的内存还有 8 个，连续 2 页空闲的有 5 个，连续 4 页空闲的有 2 个，以此类推；常规区域里面 1 页空闲的还有 2002 个，连续 2 页空闲的有 1252 个，以此类推。

对于内核物理内存映射区的虚拟内存（即从 DMA 和常规区域映射过来的），使用 `virt_to_phys()` 可以实现内核虚拟地址转化为物理地址。与之对应的函数为 `phys_to_virt()`，它将物理地址转化为内核虚拟地址。

注意：上述 `virt_to_phys()` 和 `phys_to_virt()` 方法仅适用于 DMA 和常规区域，高端内存的虚拟地址与物理地址之间不存在如此简单的换算关系。

11.3 内存存取

11.3.1 用户空间内存动态申请

在用户空间中动态申请内存的函数为 `malloc()`，这个函数在各种操作系统上的使用都是一致的，`malloc()` 申请的内存的释放函数为 `free()`。对于 Linux 而言，C 库的 `malloc()` 函数一般通过 `brk()` 和 `mmap()` 两个系统调用从内核申请内存。

由于用户空间 C 库的 `malloc` 算法实际上具备一个二次管理能力，所以并不是每次申请和释放内存都一定伴随着对内核的系统调用。比如，代码清单 11.2 的应用程序可以从内核拿到内存后，立即调用 `free()`，由于 `free()` 之前调用了 `mallopt(M_TRIM_THRESHOLD, -1)` 和 `mallopt(M_MMAP_MAX, 0)`，这个 `free()` 并不会把内存还给内核，而只是还给了 C 库的分配算法（内存仍然属于这个进程），因此之后所有的动态内存申请和释放都在用户态下进行。

代码清单 11.2 用户空间内存申请以及 `mallopt`

```

1 #include <malloc.h>
2 #include <sys/mman.h>
3
4 #define SOMESIZE (100*1024*1024)           // 100MB
5
6 int main(int argc, char *argv[])
7 {
8     unsigned char *buffer;
9     int i;
10
11    if (mlockall(MCL_CURRENT | MCL_FUTURE))
```

```

12     mallopt(M_TRIM_THRESHOLD, -1);
13     mallopt(M_MMAP_MAX, 0);
14
15     buffer = malloc(SOMESIZE);
16     if (!buffer)
17         exit(-1);
18
19     /*
20      * Touch each page in this piece of memory to get it
21      * mapped into RAM
22     */
23     for (i = 0; i < SOMESIZE; i += page_size)
24         buffer[i] = 0;
25     free(buffer);
26     /* <do your RT-thing> */
27
28     return 0;
29 }

```

另外，Linux 内核总是采用按需调页（Demand Paging），因此当 malloc() 返回的时候，虽然是成功返回，但是内核并没有真正给这个进程内存，这个时候如果去读申请的内存，内容全部是 0，这个页面的映射是只读的。只有当写到某个页面的时候，内核才在页错误后，真正把这个页面给这个进程。

11.3.2 内核空间内存动态申请

在 Linux 内核空间中申请内存涉及的函数主要包括 kmalloc()、__get_free_pages() 和 vmalloc() 等。kmalloc() 和 __get_free_pages()（及其类似函数）申请的内存位于 DMA 和常规区域的映射区，而且在物理上也是连续的，它们与真实的物理地址只有一个固定的偏移，因此存在较简单的转换关系。而 vmalloc() 在虚拟内存空间给出一块连续的内存区，实质上，这片连续的虚拟内存物理内存中并不一定连续，而 vmalloc() 申请的虚拟内存和物理内存之间也没有简单的换算关系。

1. kmalloc()

```
void *kmalloc(size_t size, int flags);
```

给 kmalloc() 的第一个参数是要分配的块的大小；第二个参数为分配标志，用于控制 kmalloc() 的行为。

最常用的分配标志是 GFP_KERNEL，其含义是在内核空间的进程中申请内存。kmalloc() 的底层依赖于 __get_free_pages() 来实现，分配标志的前缀 GFP 正好是这个底层函数的缩写。使用 GFP_KERNEL 标志申请内存时，若暂时不能满足，则进程会睡眠等待页，即会引起阻塞，因此不能在中断上下文或持有自旋锁的时候使用 GFP_KERNEL 申请内存。

由于在中断处理函数、tasklet 和内核定时器等非进程上下文中不能阻塞，所以此时驱动

应当使用 GFP_ATOMIC 标志来申请内存。当使用 GFP_ATOMIC 标志申请内存时，若不存在空闲页，则不等待，直接返回。

其他的申请标志还包括 GFP_USER (用来为用户空间页分配内存，可能阻塞)、GFP_HIGHUSER (类似 GFP_USER，但是它从高端内存分配)、GFP_DMA (从 DMA 区域分配内存)、GFP_NOIO (不允许任何 I/O 初始化)、GFP_NOFS (不允许进行任何文件系统调用)、__GFP_HIGHMEM (指示分配的内存可以位于高端内存)、__GFP_COLD (请求一个较长时间不访问的页)、__GFP_NOWARN (当一个分配无法满足时，阻止内核发出警告)、__GFP_HIGH (高优先级请求，允许获得被内核保留在紧急状况使用的最后的内存页)、__GFP_REPEAT (分配失败，则尽力重复尝试)、__GFP_NOFAIL (标志只许申请成功，不推荐) 和 __GFP_NORETRY (若申请不到，则立即放弃) 等。

使用 kmalloc() 申请的内存应使用 kfree() 释放，这个函数的用法和用户空间的 free() 类似。

2. __get_free_pages()

__get_free_pages() 系列函数 / 宏本质上是 Linux 内核最底层用于获取空闲内存的方法，因为底层的 buddy 算法以 2^n 页为单位管理空闲内存，所以最底层的内存申请总是以 2^n 页为单位的。

__get_free_pages() 系列函数 / 宏包括 get_zeroed_page()、__get_free_page() 和 __get_free_pages()。

```
get_zeroed_page(unsigned int flags);
```

该函数返回一个指向新页的指针并且将该页清零。

```
__get_free_page(unsigned int flags);
```

该宏返回一个指向新页的指针但是该页不清零，它实际上为：

```
#define __get_free_page(gfp_mask) \
    __get_free_pages((gfp_mask), 0)
```

就是调用了下面的 __get_free_pages() 申请 1 页。

```
__get_free_pages(unsigned int flags, unsigned int order);
```

该函数可分配多个页并返回分配内存的首地址，分配的页数为 2^{order} ，分配的页也不清零。order 允许的最大值是 10 (即 1024 页) 或者 11 (即 2048 页)，这取决于具体的硬件平台。

__get_free_pages() 和 get_zeroed_page() 在实现中调用了 alloc_pages() 函数，alloc_pages() 既可以在内核空间分配，也可以在用户空间分配，其原型为：

```
struct page * alloc_pages(int gfp_mask, unsigned long order);
```

参数含义与 __get_free_pages() 类似，但它返回分配的第一个页的描述符而非首地址。

使用 `__get_free_pages()` 系列函数 / 宏申请的内存应使用下列函数释放：

```
void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
```

`__get_free_pages` 等函数在使用时，其申请标志的值与 `kmalloc()` 完全一样，各标志的含义也与 `kmalloc()` 完全一致，最常用的是 `GFP_KERNEL` 和 `GFP_ATOMIC`。

3. `vmalloc()`

`vmalloc()` 一般只为存在于软件中（没有对应的硬件意义）的较大的顺序缓冲区分配内存，`vmalloc()` 远大于 `__get_free_pages()` 的开销，为了完成 `vmalloc()`，新的页表项需要被建立。因此，只是调用 `vmalloc()` 来分配少量的内存（如 1 页以内的内存）是不妥的。

`vmalloc()` 申请的内存应使用 `vfree()` 释放，`vmalloc()` 和 `vfree()` 的函数原型如下：

```
void *vmalloc(unsigned long size);
void vfree(void * addr);
```

`vmalloc()` 不能用在原子上下文中，因为它的内部实现使用了标志为 `GFP_KERNEL` 的 `kmalloc()`。

使用 `vmalloc()` 函数的一个例子函数是 `create_module()` 系统调用，它利用 `vmalloc()` 函数来获取被创建模块需要的内存空间。

`vmalloc()` 在申请内存时，会进行内存的映射，改变页表项，不像 `kmalloc()` 实际用的是开机过程中就映射好的 DMA 和常规区域的页表项。因此 `vmalloc()` 的虚拟地址和物理地址不是一个简单的线性映射。

4. slab 与内存池

一方面，完全使用页为单元申请和释放内存容易导致浪费（如果要申请少量字节，也需要用 1 页）；另一方面，在操作系统的运作过程中，经常会涉及大量对象的重复生成、使用和释放内存问题。在 Linux 系统中所用到的对象，比较典型的例子是 `inode`、`task_struct` 等。如果我们能够用合适的方法使得对象在前后两次被使用时分配在同一块内存或同一类内存空间且保留了基本的数据结构，就可以大大提高效率。`slab` 算法就是针对上述特点设计的。实际上 `kmalloc()` 就是使用 `slab` 机制实现的。

`slab` 是建立在 `buddy` 算法之上的，它从 `buddy` 算法拿到 2^n 页面后再次进行二次管理，这一点和用户空间的 C 库很像。`slab` 申请的内存以及基于 `slab` 的 `kmalloc()` 申请的内存，与物理内存之间也是一个简单的线性偏移。

(1) 创建 slab 缓存

```
struct kmem_cache *kmem_cache_create(const char *name, size_t size,
    size_t align, unsigned long flags,
    void (*ctor)(void*, struct kmem_cache *, unsigned long),
    void (*dtor)(void*, struct kmem_cache *, unsigned long));
```

`kmem_cache_create()` 用于创建一个 slab 缓存，它是一个可以保留任意数目且全部同样大

小的后备缓存。参数 size 是要分配的每个数据结构的大小，参数 flags 是控制如何进行分配的位掩码，包括 SLAB_HWCACHE_ALIGN（每个数据对象被对齐到一个缓存行）、SLAB_CACHE_DMA（要求数据对象在 DMA 区域中分配）等。

(2) 分配 slab 缓存

```
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags);
```

上述函数在 kmem_cache_create() 创建的 slab 后备缓存中分配一块并返回首地址指针。

(3) 释放 slab 缓存

```
void kmem_cache_free(struct kmem_cache *cachep, void *objp);
```

上述函数释放由 kmem_cache_alloc() 分配的缓存。

(4) 收回 slab 缓存

```
int kmem_cache_destroy(struct kmem_cache *cachep);
```

代码清单 11.3 给出了 slab 缓存的使用范例。

代码清单 11.3 slab 缓存使用范例

```

1 /* 创建 slab 缓存 */
2 static kmem_cache_t *xxx_cachep;
3 xxx_cachep = kmem_cache_create("xxx", sizeof(struct xxx),
4                                0, SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL, NULL);
5 /* 分配 slab 缓存 */
6 struct xxx *ctx;
7 ctx = kmem_cache_alloc(xxx_cachep, GFP_KERNEL);
8 .../* 使用 slab 缓存 */
9 /* 释放 slab 缓存 */
10 kmem_cache_free(xxx_cachep, ctx);
11 kmem_cache_destroy(xxx_cachep);

```

在系统中通过 /proc/slabinfo 节点可以获知当前 slab 的分配和使用情况，运行 cat/proc/slabinfo:

```
# cat /proc/slabinfo
slabinfo - version: 2.1
# name          <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> :
      tunables <limit> <batchcount> <sharedfactor> : slabdata <active_slabs>
      <num_slabs> <sharedavail>
isoofs_inode_cache    66     66     360     22      2 : tunables      0     0     0 :
      slabdata    3      3      0
ext4_groupinfo_4k    156    156    104     39      1 : tunables      0     0     0 :
      slabdata    4      4      0
UDPLITEv6            0      0      0    768     21      4 : tunables      0     0     0 :
      slabdata    0      0      0
UDPV6                84     84    768     21      4 : tunables      0     0     0 :
      slabdata    4      4      0
tw_sock_TCPv6         0      0      0   192     21      1 : tunables      0     0     0 :
      slabdata    0      0      0
```

```

TCPv6          88     88   1472   22    8 : tunables   0     0     0 :
  slabdata      4      4     0
zcache_objnode 0      0     272   30    2 : tunables   0     0     0 :
  slabdata      0      0     0
kcopyd_job      0      0   2344   13    8 : tunables   0     0     0 :
  slabdata      0      0     0
dm_uevent      0      0   2464   13    8 : tunables   0     0     0 :
  slabdata      0      0     0
...

```

注意：slab不是要代替`_get_free_pages()`，其在最底层仍然依赖于`_get_free_pages()`，slab在底层每次申请1页或多页，之后再分隔这些页为更小的单元进行管理，从而节省了内存，也提高了slab缓冲对象的访问效率。

除了slab以外，在Linux内核中还包含对内存池的支持，内存池技术也是一种非常经典的用于分配大量小对象的后备缓存技术。

在Linux内核中，与内存池相关的操作包括如下几种。

(1) 创建内存池

```

mempool_t *mempool_create(int min_nr, mempool_alloc_t *alloc_fn,
                           mempool_free_t *free_fn, void *pool_data);

```

`mempool_create()`函数用于创建一个内存池，`min_nr`参数是需要预分配对象的数目，`alloc_fn`和`free_fn`是指向内存池机制提供的标准对象分配和回收函数的指针，其原型分别为：

```

typedef void *(mempool_alloc_t)(int gfp_mask, void *pool_data);

```

和

```

typedef void (mempool_free_t)(void *element, void *pool_data);

```

`pool_data`是分配和回收函数用到的指针，`gfp_mask`是分配标记。只有当`__GFP_WAIT`标记被指定时，分配函数才会休眠。

(2) 分配和回收对象

在内存池中分配和回收对象需由以下函数来完成：

```

void *mempool_alloc(mempool_t *pool, int gfp_mask);
void mempool_free(void *element, mempool_t *pool);

```

`mempool_alloc()`用来分配对象，如果内存池分配器无法提供内存，那么就可以用预分配的池。

(3) 回收内存池

```

void mempool_destroy(mempool_t *pool);

```

由`mempool_create()`函数创建的内存池需由`mempool_destroy()`来回收。

11.4 设备 I/O 端口和 I/O 内存的访问

设备通常会提供一组寄存器来控制设备、读写设备和获取设备状态，即控制寄存器、数据寄存器和状态寄存器。这些寄存器可能位于 I/O 空间中，也可能位于内存空间中。当位于 I/O 空间时，通常被称为 I/O 端口；当位于内存空间时，对应的内存空间被称为 I/O 内存。

11.4.1 Linux I/O 端口和 I/O 内存访问接口

1. I/O 端口

在 Linux 设备驱动中，应使用 Linux 内核提供的函数来访问定位于 I/O 空间的端口，这些函数包括如下几种。

1) 读写字节端口（8位宽）。

```
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
```

2) 读写字端口（16位宽）。

```
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
```

3) 读写长字端口（32位宽）。

```
unsigned inl(unsigned port);
void outl(unsigned longword, unsigned port);
```

4) 读写一串字节。

```
void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
```

5) insb() 从端口 port 开始读 count 个字节端口，并将读取结果写入 addr 指向的内存；outsb() 将 addr 指向的内存中的 count 个字节连续写入以 port 开始的端口。

6) 读写一串字。

```
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
```

7) 读写一串长字。

```
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
```

上述各函数中 I/O 端口号 port 的类型高度依赖于具体的硬件平台，因此，这里只是写出了 unsigned。

2. I/O 内存

在内核中访问 I/O 内存（通常是芯片内部的各个 I²C、SPI、USB 等控制器的寄存器或者

外部内存总线上的设备)之前，需首先使用 ioremap() 函数将设备所处的物理地址映射到虚拟地址上。ioremap() 的原型如下：

```
void *ioremap(unsigned long offset, unsigned long size);
```

ioremap() 与 vmalloc() 类似，也需要建立新的页表，但是它并不进行 vmalloc() 中所执行的内存分配行为。ioremap() 返回一个特殊的虚拟地址，该地址可用来存取特定的物理地址范围，这个虚拟地址位于 vmalloc 映射区域。通过 ioremap() 获得的虚拟地址应该被 iounmap() 函数释放，其原型如下：

```
void iounmap(void * addr);
```

ioremap() 有个变体是 devm_ioremap()，类似于其他以 devm_ 开头的函数，通过 devm_ioremap() 进行的映射通常不需要在驱动退出和出错处理的时候进行 iounmap()。devm_ioremap() 的原型为：

```
void __iomem *devm_ioremap(struct device *dev, resource_size_t offset,
                           unsigned long size);
```

在设备的物理地址(一般都是寄存器)被映射到虚拟地址之后，尽管可以直接通过指针访问这些地址，但是 Linux 内核推荐用一组标准的 API 来完成设备内存映射的虚拟地址的读写。

读寄存器用 readb_relaxed()、readw_relaxed()、readl_relaxed()、readb()、readw()、readl() 这一组 API，以分别读 8bit、16bit、32bit 的寄存器，没有_relaxed 后缀的版本与有_relaxed 后缀的版本的区别是没有_relaxed 后缀的版本包含一个内存屏障，如：

```
#define readb(c)          ({ u8 __v = readb_relaxed(c); __iormb(); __v; })
#define readw(c)          ({ u16 __v = readw_relaxed(c); __iormb(); __v; })
#define readl(c)          ({ u32 __v = readl_relaxed(c); __iormb(); __v; })
```

写寄存器用 writeb_relaxed()、writew_relaxed()、writel_relaxed()、writeb()、writew()、writel() 这一组 API，以分别写 8bit、16bit、32bit 的寄存器，没有_relaxed 后缀的版本与有_relaxed 后缀的版本的区别是前者包含一个内存屏障，如：

```
#define writeb(v,c)        ({ __iowmb(); writeb_relaxed(v,c); })
#define writew(v,c)        ({ __iowmb(); writew_relaxed(v,c); })
#define writel(v,c)        ({ __iowmb(); writel_relaxed(v,c); })
```

11.4.2 申请与释放设备的 I/O 端口和 I/O 内存

1. I/O 端口申请

Linux 内核提供了一组函数以申请和释放 I/O 端口，表明该驱动要访问这片区域。

```
struct resource *request_region(unsigned long first, unsigned long n, const char *name);
```

这个函数向内核申请 n 个端口，这些端口从 first 开始，name 参数为设备的名称。如果分配成功，则返回值不是 NULL，如果返回 NULL，则意味着申请端口失败。

当用 request_region() 申请的 I/O 端口使用完成后，应当使用 release_region() 函数将它们归还给系统，这个函数的原型如下：

```
void release_region(unsigned long start, unsigned long n);
```

2. I/O 内存申请

同样，Linux 内核也提供了一组函数以申请和释放 I/O 内存的范围。此处的“申请”表明该驱动要访问这片区域，它不会做任何内存映射的动作，更多的是类似于“reservation”的概念。

```
struct resource *request_mem_region(unsigned long start, unsigned long len, char *name);
```

这个函数向内核申请 n 个内存地址，这些地址从 first 开始，name 参数为设备的名称。如果分配成功，则返回值不是 NULL，如果返回 NULL，则意味着申请 I/O 内存失败。

当用 request_mem_region() 申请的 I/O 内存使用完成后，应当使用 release_mem_region() 函数将它们归还给系统，这个函数的原型如下：

```
void release_mem_region(unsigned long start, unsigned long len);
```

request_region() 和 request_mem_region() 也分别有变体，其为 devm_request_region() 和 devm_request_mem_region()。

11.4.3 设备 I/O 端口和 I/O 内存访问流程

综合 11.3 节和本节的内容，可以归纳出设备驱动访问 I/O 端口和 I/O 内存的步骤。

I/O 端口访问的一种途径是直接使用 I/O 端口操作函数：在设备打开或驱动模块被加载时申请 I/O 端口区域，之后使用 inb()、outb() 等进行端口访问，最后，在设备关闭或驱动被卸载时释放 I/O 端口范围。整个流程如图 11.10 所示。

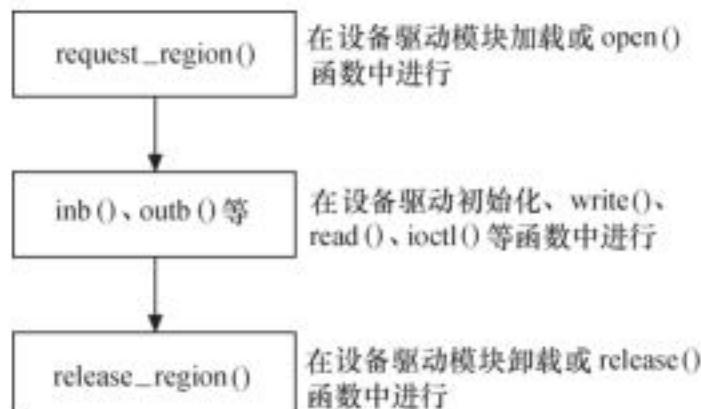


图 11.10 I/O 端口的访问流程

I/O 内存的访问步骤如图 11.11 所示，首先是调用 request_mem_region() 申请资源，接着

将寄存器地址通过 ioremap() 映射到内核空间虚拟地址，之后就可以通过 Linux 设备访问编程接口访问这些设备的寄存器了。访问完成后，应对 ioremap() 申请的虚拟地址进行释放，并释放 release_mem_region() 申请的 I/O 内存资源。

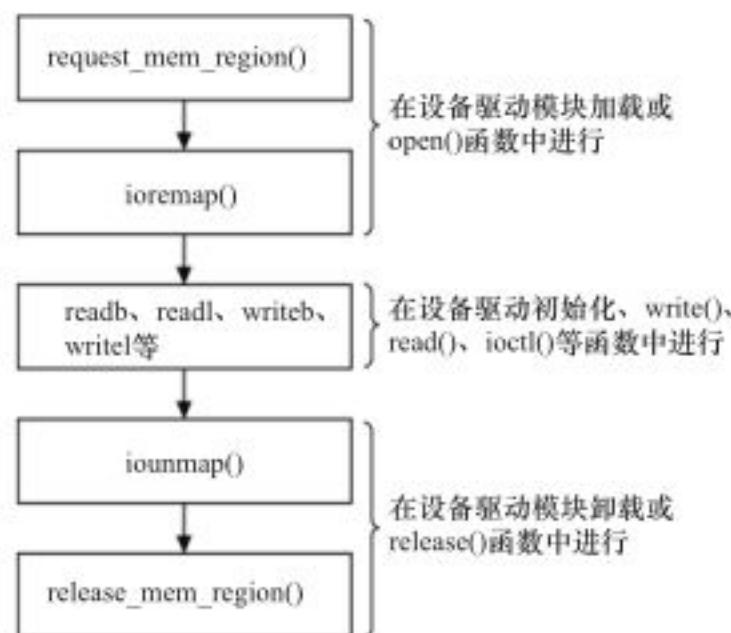


图 11.11 I/O 内存访问流程

有时候，驱动在访问寄存器或 I/O 端口前，会省去 request_mem_region()、request_region() 这样的调用。

11.4.4 将设备地址映射到用户空间

1. 内存映射与 VMA

一般情况下，用户空间是不可能也不应该直接访问设备的，但是，设备驱动程序中可实现 mmap() 函数，这个函数可使得用户空间能直接访问设备的物理地址。实际上，mmap() 实现了这样的一个映射过程：它将用户空间的一段内存与设备内存关联，当用户访问用户空间的这段地址范围时，实际上会转化为对设备的访问。

这种能力对于显示适配器一类的设备非常有意义，如果用户空间可直接通过内存映射访问显存的话，屏幕帧的各点像素将不再需要一个从用户空间到内核空间的复制的过程。

mmap() 必须以 PAGE_SIZE 为单位进行映射，实际上，内存只能以页为单位进行映射，若要映射非 PAGE_SIZE 整数倍的地址范围，要先进行页对齐，强行以 PAGE_SIZE 的倍数大小进行映射。

从 file_operations 文件操作结构体可以看出，驱动中 mmap() 函数的原型如下：

```
int(*mmap)(struct file *, struct vm_area_struct*);
```

驱动中的 mmap() 函数将在用户进行 mmap() 系统调用时最终被调用，mmap() 系统调用

的原型与 file_operations 中 mmap() 的原型区别很大，如下所示：

```
caddr_t mmap (caddr_t addr, size_t len, int prot, int flags, int fd, off_t offset);
```

参数 fd 为文件描述符，一般由 open() 返回，fd 也可以指定为 -1，此时需指定 flags 参数中的 MAP_ANON，表明进行的是匿名映射。

len 是映射到调用用户空间的字节数，它从被映射文件开头 offset 个字节开始算起，offset 参数一般设为 0，表示从文件头开始映射。

prot 参数指定访问权限，可取如下几个值的“或”：PROT_READ(可读)、PROT_WRITE(可写)、PROT_EXEC(可执行) 和 PROT_NONE(不可访问)。

参数 addr 指定文件应被映射到用户空间的起始地址，一般被指定为 NULL，这样，选择起始地址的任务将由内核完成，而函数的返回值就是映射到用户空间的地址。其类型 caddr_t 实际上就是 void*。

当用户调用 mmap() 的时候，内核会进行如下处理。

- 1) 在进程的虚拟空间查找一块 VMA。
- 2) 将这块 VMA 进行映射。
- 3) 如果设备驱动程序或者文件系统的 file_operations 定义了 mmap() 操作，则调用它。
- 4) 将这个 VMA 插入进程的 VMA 链表中。

file_operations 中 mmap() 函数的第一个参数就是步骤 1) 找到的 VMA。

由 mmap() 系统调用映射的内存可由 munmap() 解除映射，这个函数的原型如下：

```
int munmap(caddr_t addr, size_t len );
```

驱动程序中 mmap() 的实现机制是建立页表，并填充 VMA 结构体中 vm_operations_struct 指针。VMA 就是 vm_area_struct，用于描述一个虚拟内存区域，VMA 结构体的定义如代码清单 11.4 所示。

代码清单 11.4 VMA 结构体

```

1 struct vm_area_struct {
2     /* The first cache line has the info for VMA tree walking. */
3
4     unsigned long vm_start;           /* Our start address within vm_mm. */
5     unsigned long vm_end;            /* The first byte after our end address
6                                         within vm_mm. */
7
8     /* linked list of VM areas per task, sorted by address */
9     struct vm_area_struct *vm_next, *vm_prev;
10
11    struct rb_node vm_rb;
12
13    ...
14
15    /* Second cache line starts here. */

```

```

16
17     struct mm_struct *vm_mm;           /* The address space we belong to. */
18     pgprot_t vm_page_prot;          /* Access permissions of this VMA. */
19     unsigned long vm_flags;         /* Flags, see mm.h. */
20
21 ...
22     const struct vm_operations_struct *vm_ops;
23
24     /* Information about our backing store: */
25     unsigned long vm_pgoff;          /* Offset (within vm_file) in PAGE_SIZE
26                                         units, *not* PAGE_CACHE_SIZE */
26                                         /* File we map to (can be NULL). */
27     struct file * vm_file;           /* was vm_pte (shared mem) */
28     void * vm_private_data;
29 ...
30 };

```

VMA 结构体描述的虚地址介于 `vm_start` 和 `vm_end` 之间，而其 `vm_ops` 成员指向这个 VMA 的操作集。针对 VMA 的操作都被包含在 `vm_operations_struct` 结构体中，`vm_operations_struct` 结构体的定义如代码清单 11.5 所示。

代码清单 11.5 `vm_operations_struct` 结构体

```

1 struct vm_operations_struct {
2     void (*open)(struct vm_area_struct * area);
3     void (*close)(struct vm_area_struct * area);
4     int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);
5     void (*map_pages)(struct vm_area_struct *vma, struct vm_fault *vmf);
6
7     /* notification that a previously read-only page is about to become
8      * writable, if an error is returned it will cause a SIGBUS */
9     int (*page_mkwrite)(struct vm_area_struct *vma, struct vm_fault *vmf);
10
11    /* called by access_process_vm when get_user_pages() fails, typically
12       * for use by special VMAs that can switch between memory and hardware
13       */
14    int (*access)(struct vm_area_struct *vma, unsigned long addr,
15                  void *buf, int len, int write);
16 ...
17 };

```

整个 `vm_operations_struct` 结构体的实体会在 `file_operations` 的 `mmap()` 成员函数里被赋值给相应的 `vma->vm_ops`，而上述 `open()` 函数也通常在 `mmap()` 里调用，`close()` 函数会在用户调用 `munmap()` 的时候被调用到。代码清单 11.6 给出了一个 `vm_operations_struct` 的操作范例。

代码清单 11.6 `vm_operations_struct` 操作范例

```

1 static int xxx_mmap(struct file *filp, struct vm_area_struct *vma)
2 {
3     if (remap_pfn_range(vma, vma->vm_start, vm->vm_pgoff, vma->vm_end - vma

```

```

4      ->vm_start, vma->vm_page_prot))           /* 建立页表 */
5      return -EAGAIN;
6  vma->vm_ops = &xxx_remap_vm_ops;
7  xxx_vma_open(vma);
8  return 0;
9 }
10
11 static void xxx_vma_open(struct vm_area_struct *vma)      /* VMA 打开函数 */
12 {
13 ...
14 printk(KERN_NOTICE "xxx VMA open, virt %lx, phys %lx\n", vma->vm_start,
15       vma->vm_pgoff << PAGE_SHIFT);
16 }
17
18 static void xxx_vma_close(struct vm_area_struct *vma)      /* VMA 关闭函数 */
19 {
20 ...
21 printk(KERN_NOTICE "xxx VMA close.\n");
22 }
23
24 static struct vm_operations_struct xxx_remap_vm_ops = { /* VMA 操作结构体 */
25     .open = xxx_vma_open,
26     .close = xxx_vma_close,
27     ...
28 };

```

第3行调用的 remap_pfn_range() 创建页表项，以 VMA 结构体的成员（VMA 的数据成员是内核根据用户的请求自己填充的）作为 remap_pfn_range() 的参数，映射的虚拟地址范围是 vma->vm_start 至 vma->vm_end。

remap_pfn_range() 函数的原型如下：

```
int remap_pfn_range(struct vm_area_struct *vma, unsigned long addr,
                     unsigned long pfn, unsigned long size, pgprot_t prot);
```

其中的 addr 参数表示内存映射开始处的虚拟地址。remap_pfn_range() 函数为 $addr \sim addr + size$ 的虚拟地址构造页表。

pfn 是虚拟地址应该映射到的物理地址的页帧号，实际上就是物理地址右移 PAGE_SHIFT 位。若 PAGE_SIZE 为 4KB，则 PAGE_SHIFT 为 12，因为 PAGE_SIZE 等于 $1 \ll PAGE_SHIFT$ 。
prot 是新页所要求的保护属性。

在驱动程序中，我们能使用 remap_pfn_range() 映射内存中的保留页、设备 I/O、framebuffer、camera 等内存。在 remap_pfn_range() 上又可以进一步封装出 io_remap_pfn_range()、vm_iomap_memory() 等 API。

```
#define io_remap_pfn_range remap_pfn_range

int vm_iomap_memory(struct vm_area_struct *vma, phys_addr_t start, unsigned long len)
```

```

{
    unsigned long vm_len, pfn, pages;

    ...
    len += start & ~PAGE_MASK;
    pfn = start >> PAGE_SHIFT;
    pages = (len + ~PAGE_MASK) >> PAGE_SHIFT;
    ...
    pfn += vma->vm_pgoff;
    pages -= vma->vm_pgoff;

    /* Can we fit all of the mapping? */
    vm_len = vma->vm_end - vma->vm_start;
    ...

    /* Ok, let it rip */
    return io_remap_pfn_range(vma, vma->vm_start, pfn, vm_len, vma->vm_page_prot);
}

```

代码清单 11.7 给出了 LCD 驱动映射 framebuffer 物理地址到用户空间的典型范例，代码取自 drivers/video/fbdev/core/fbmem.c。

代码清单 11.7 LCD 驱动映射 framebuffer 的 mmap

```

1 static int
2 fb_mmap(struct file *file, struct vm_area_struct * vma)
3 {
4     struct fb_info *info = file_fb_info(file);
5     struct fb_ops *fb;
6     unsigned long mmio_pgoff;
7     unsigned long start;
8     u32 len;
9
10    if (!info)
11        return -ENODEV;
12    fb = info->fbops;
13    if (!fb)
14        return -ENODEV;
15    mutex_lock(&info->mm_lock);
16    if (fb->fb_mmap) {
17        int res;
18        res = fb->fb_mmap(info, vma);
19        mutex_unlock(&info->mm_lock);
20        return res;
21    }
22
23    /*
24     * Ugh. This can be either the frame buffer mapping, or
25     * if pgoff points past it, the mmio mapping.
26     */
27    start = info->fix.smem_start;

```

```

28     len = info->fix.smem_len;
29     mmio_pgoff = PAGE_ALIGN((start & ~PAGE_MASK) + len) >> PAGE_SHIFT;
30     if (vma->vm_pgoff >= mmio_pgoff) {
31         if (info->var.accel_flags) {
32             mutex_unlock(&info->mm_lock);
33             return -EINVAL;
34         }
35
36         vma->vm_pgoff -= mmio_pgoff;
37         start = info->fix.mmmio_start;
38         len = info->fix.mmmio_len;
39     }
40     mutex_unlock(&info->mm_lock);
41
42     vma->vm_page_prot = vm_get_page_prot(vma->vm_flags);
43     fb_pgprotect(file, vma, start);
44
45     return vm_iomap_memory(vma, start, len);
46 }

```

通常，I/O 内存被映射时需要是 nocache 的，这时候，我们应该对 vma->vm_page_prot 设置 nocache 标志之后再映射，如代码清单 11.8 所示。

代码清单 11.8 以 nocache 方式将内核空间映射到用户空间

```

1 static int xxx_nocache_mmap(struct file *filp, struct vm_area_struct *vma)
2 {
3     vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot); /* 赋 nocache 标志 */
4     vma->vm_pgoff = ((u32)map_start >> PAGE_SHIFT);
5     /* 映射 */
6     if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff, vma->vm_end - vma
7         ->vm_start, vma->vm_page_prot))
8         return -EAGAIN;
9     return 0;
10 }

```

上述代码第 3 行的 pgprot_noncached() 是一个宏，它高度依赖于 CPU 的体系结构，ARM 的 pgprot_noncached() 定义如下：

```
#define pgprot_noncached(prot) \
    __pgprot_modify(prot, L_PTE_MT_MASK, L_PTE_MT_UNCACHED)
```

另一个比 pgprot_noncached() 稍微少一些限制的宏是 pgprot_writecombine()，它的定义如下：

```
#define pgprot_writecombine(prot) \
    __pgprot_modify(prot, L_PTE_MT_MASK, L_PTE_MT_BUFFERABLE)
```

pgprot_noncached() 实际禁止了相关页的 Cache 和写缓冲（Write Buffer），pgprot_writecombine() 则没有禁止写缓冲。ARM 的写缓冲器是一个非常小的 FIFO 存储器，位于处理器核与主存之

间，其目的在于将处理器核和 Cache 从较慢的主存写操作中解脱出来。写缓冲区与 Cache 在存储层次上处于同一层次，但是它只作用于写主存。

2. fault() 函数

除了 remap_pfn_range() 以外，在驱动程序中实现 VMA 的 fault() 函数通常可以为设备提供更加灵活的内存映射途径。当访问的页不在内存里，即发生缺页异常时，fault() 会被内核自动调用，而 fault() 的具体行为可以自定义。这是因为当发生缺页异常时，系统会经过如下处理过程。

- 1) 找到缺页的虚拟地址所在的 VMA。
- 2) 如果必要，分配中间页目录表和页表。
- 3) 如果页表项对应的物理页面不存在，则调用这个 VMA 的 fault() 方法，它返回物理页面的页描述符。
- 4) 将物理页面的地址填充到页表中。

fault() 函数在 Linux 的早期版本中命名为 nopage()，后来变更为了 fault()。代码清单 11.9 给出了一个设备驱动中使用 fault() 的典型范例。

代码清单 11.9 fault() 函数使用范例

```

1 static int xxx_fault(struct vm_area_struct *vma, struct vm_fault *vmf)
2 {
3     unsigned long paddr;
4     unsigned long pfn;
5     pgoff_t index = vmf->pgoff;
6     struct vma_data *vdata = vma->vm_private_data;
7
8     ...
9
10    pfn = paddr >> PAGE_SHIFT;
11
12    vm_insert_pfn(vma, (unsigned long)vmf->virtual_address, pfn);
13
14    return VM_FAULT_NOPAGE;
15 }
```



大多数设备驱动都不需要提供设备内存到用户空间的映射能力，因为，对于串口等面向流的设备而言，实现这种映射毫无意义。而对于显示、视频等设备，建立映射可减少用户空间和内核空间之间的内存复制。

11.5 I/O 内存静态映射

在将 Linux 移植到目标电路板的过程中，有得会建立外设 I/O 内存物理地址到虚拟地址的静态映射，这个映射通过在与电路板对应的 map_desc 结构体数组中添加新的成员来完成，

map_desc 结构体的定义如代码清单 11.10 所示。

代码清单 11.10 map_desc 结构体

```

1 struct map_desc {
2     unsigned long virtual;           /* 虚拟地址 */
3     unsigned long pfn;              /* __phys_to_pfn(phy_addr) */
4     unsigned long length;           /* 大小 */
5     unsigned int type;              /* 类型 */
6 };

```

例如，在内核 arch/arm/mach-ixp2000/ixdp2x01.c 文件对应的 Intel IXDP2401 和 IXDP2801 平台上包含一个 CPLD，该文件中就进行了 CPLD 物理地址到虚拟地址的静态映射，如代码清单 11.11 所示。

代码清单 11.11 在电路板文件中增加物理地址到虚拟地址的静态映射

```

1 static struct map_desc ixdp2x01_io_desc __initdata = {
2     .virtual      = IXDP2X01_VIRT_CPLD_BASE,
3     .pfn          = __phys_to_pfn(IXDP2X01_PHYS_CPLD_BASE),
4     .length       = IXDP2X01_CPLD_REGION_SIZE,
5     .type         = MT_DEVICE
6 };
7
8 static void __init ixdp2x01_map_io(void)
9 {
10     ixp2000_map_io();
11     iotable_init(&ixdp2x01_io_desc, 1);
12 }

```

代码清单 11.11 中的第 11 行 iotable_init() 是最终建立页映射的函数，它通过 MACHINE_START、MACHINE_END 宏赋值给电路板的 map_io() 函数。将 Linux 操作系统移植到特定平台上，MACHINE_START（或者 DT_MACHINE_START）、MACHINE_END 宏之间的定义针对特定电路板而设计，其中的 map_io() 成员函数完成 I/O 内存的静态映射。

在一个已经移植好操作系统的内核中，驱动工程师可以对非常规内存区域的 I/O 内存（外设控制器寄存器、MCU 内部集成的外设控制器寄存器等）依照电路板的资源使用情况添加到 map_desc 数组中，但是目前该方法已经不值得推荐。

11.6 DMA

DMA 是一种无须 CPU 的参与就可以让外设与系统内存之间进行双向数据传输的硬件机制。使用 DMA 可以使系统 CPU 从实际的 I/O 数据传输过程中摆脱出来，从而大大提高系统的吞吐率。DMA 通常与硬件体系结构，特别是外设的总线技术密切相关。

DMA 方式的数据传输由 DMA 控制器（DMAC）控制，在传输期间，CPU 可以并发地执

行其他任务。当 DMA 结束后，DMAC 通过中断通知 CPU 数据传输已经结束，然后由 CPU 执行相应的中断服务程序进行后处理。

11.6.1 DMA 与 Cache 一致性

Cache 和 DMA 本身似乎是两个毫不相关的事物。Cache 被用作 CPU 针对内存的缓存，利用程序的空间局部性和时间局部性原理，达到较高的命中率，从而避免 CPU 每次都必须要与相对慢速的内存交互数据来提高数据的访问速率。DMA 可以作为内存与外设之间传输数据的方式，在这种传输方式之下，数据并不需要经过 CPU 中转。

假设 DMA 针对内存的目的地址与 Cache 缓存的对象没有重叠区域（如图 11.12 所示），DMA 和 Cache 之间将相安无事。但是，如果 DMA 的目的地址与 Cache 所缓存的内存地址访问有重叠（如图 11.13 所示），经过 DMA 操作，与 Cache 缓存对应的内存中的数据已经被修改，而 CPU 本身并不知道，它仍然认为 Cache 中的数据就是内存中的数据，那在以后访问 Cache 映射的内存时，它仍然使用陈旧的 Cache 数据。这样就会发生 Cache 与内存之间数据“不一致性”的错误。

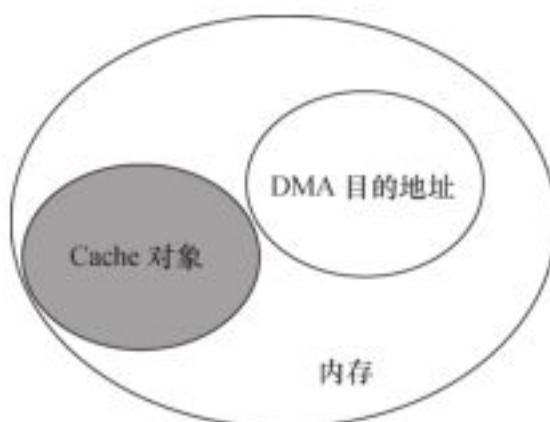


图 11.12 DMA 目的地与 Cache 对象没有重叠

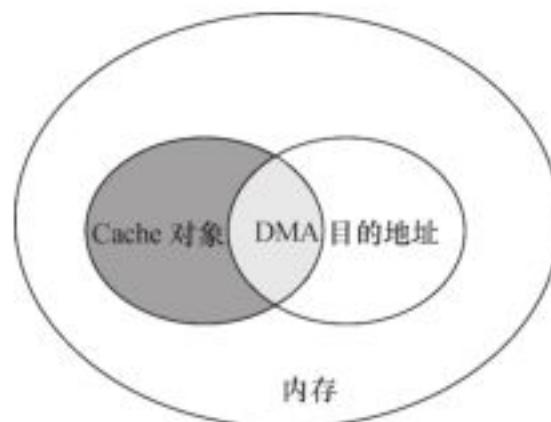


图 11.13 DMA 目的地与 Cache 对象有重叠

所谓 Cache 数据与内存数据的不一致性，是指在采用 Cache 的系统中，同样一个数据可能既存在于 Cache 中，也存在于主存中，Cache 与主存中的数据一样则具有一致性，数据若不一样则具有不一致性。

需要特别注意的是，Cache 与内存的一致性问题经常被初学者遗忘。在发生 Cache 与内存不一致性错误后，驱动将无法正常运行。如果没有相关的背景知识，工程师几乎无法定位错误的原因，因为这时所有的程序看起来都是完全正确的。Cache 的不一致性问题并不是只发生在 DMA 的情况下，实际上，它还存在于 Cache 使能和关闭的时刻。例如，对于带 MMU 功能的 ARM 处理器，在开启 MMU 之前，需要先置 Cache 无效，对于 TLB，也是如此，代码清单 11.12 给出的这段汇编可用来完成此任务。

代码清单 11.12 置 ARM 的 Cache 无效

```

1 /* 使 cache 无效 */
2 "mov    r0, #0\n"

```

```

3 "mcr    p15, 0, r0, c7, c7, 0\n"          /* 使数据和指令 cache 无效 */
4 "mcr    p15, 0, r0, c7, c10, 4\n"        /* 放空写缓冲 */
5 "mcr    p15, 0, r0, c8, c7, 0\n"          /* 使 TLB 无效 */

```

11.6.2 Linux 下的 DMA 编程

首先 DMA 本身不属于一种等同于字符设备、块设备和网络设备的外设，它只是一种外设与内存交互数据的方式。因此，本节的标题不是“Linux 下的 DMA 驱动”而是“Linux 下的 DMA 编程”。

内存中用于与外设交互数据的一块区域称为 DMA 缓冲区，在设备不支持 scatter/gather（分散 / 聚集，简称 SG）操作的情况下，DMA 缓冲区在物理上必须是连续的。

1. DMA 区域

对于 x86 系统的 ISA 设备而言，其 DMA 操作只能在 16MB 以下的内存中进行，因此，在使用 kmalloc()、__get_free_pages() 及其类似函数申请 DMA 缓冲区时应使用 GFP_DMA 标志，这样能保证获得的内存位于 DMA 区域中，并具备 DMA 能力。

在内核中定义了 __get_free_pages() 针对 DMA 的“快捷方式” __get_dma_pages()，它在申请标志中添加了 GFP_DMA，如下所示：

```
#define __get_dma_pages(gfp_mask, order) \
    __get_free_pages((gfp_mask) | GFP_DMA, (order))
```

如果不想使用 log_size（即 order）为参数申请 DMA 内存，则可以使用另一个函数 dma_mem_alloc()，其源代码如代码清单 11.13 所示。

代码清单 11.13 dma_mem_alloc() 函数

```

1 static unsigned long dma_mem_alloc(int size)
2 {
3     int order = get_order(size);      /* 大小 -> 指数 */
4     return __get_dma_pages(GFP_KERNEL, order);
5 }
```

对于大多数现代嵌入式处理器而言，DMA 操作可以在整个常规内存区域进行，因此 DMA 区域就直接覆盖了常规内存。

2. 虚拟地址、物理地址和总线地址

基于 DMA 的硬件使用的是总线地址而不是物理地址，总线地址是从设备角度上看到的内存地址，物理地址则是从 CPU MMU 控制器外围角度上看到的内存地址（从 CPU 核角度看到的是虚拟地址）。虽然在 PC 上，对于 ISA 和 PCI 而言，总线地址即为物理地址，但并不是每个平台都是如此。因为有时候接口总线通过桥接电路连接，桥接电路会将 I/O 地址映射为不同的物理地址。例如，在 PReP（PowerPC Reference Platform）系统中，物理地址 0 在设备端看起来是 0x80000000，而 0 通常又被映射为虚拟地址 0xC0000000，所以同一地址就具

备了三重身份：物理地址 0、总线地址 0x80000000 及虚拟地址 0xC0000000。还有一些系统提供了页面映射机制，它能将任意的页面映射为连续的外设总线地址。内核提供了如下函数以进行简单的虚拟地址 / 总线地址转换：

```
unsigned long virt_to_bus(volatile void *address);
void *bus_to_virt(unsigned long address);
```

在使用 IOMMU 或反弹缓冲区的情况下，上述函数一般不会正常工作。而且，这两个函数并不建议使用。如图 11.14 所示，IOMMU 的工作原理与 CPU 内的 MMU 非常类似，不过它针对的是外设总线地址和内存地址之间的转化。由于 IOMMU 可以使得外设 DMA 引擎看到“虚拟地址”，因此在使用 IOMMU 的情况下，在修改映射寄存器后，可以使得 SG 中分段的缓冲区地址对外设变得连续。

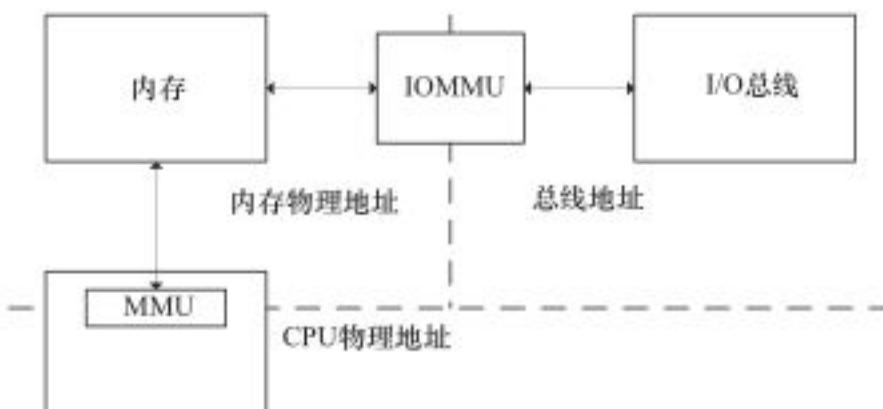


图 11.14 MMU 与 IOMMU

3. DMA 地址掩码

设备并不一定能在所有的内存地址上执行 DMA 操作，在这种情况下应该通过下列函数执行 DMA 地址掩码：

```
int dma_set_mask(struct device *dev, u64 mask);
```

例如，对于只能在 24 位地址上执行 DMA 操作的设备而言，就应该调用 `dma_set_mask(dev, 0xfffffff)`。

其实该 API 本质上就是修改 `device` 结构体中的 `dma_mask` 成员，如 ARM 平台的定义为：

```
int arm_dma_set_mask(struct device *dev, u64 dma_mask)
{
    if (!dev->dma_mask || !dma_supported(dev, dma_mask))
        return -EIO;

    *dev->dma_mask = dma_mask;

    return 0;
}
```

在 `device` 结构体中，除了有 `dma_mask` 以外，还有一个 `coherent_dma_mask` 成员。`dma_`

mask 是设备 DMA 可以寻址的范围，而 coherent_dma_mask 作用于申请一致性的 DMA 缓冲区。

4. 一致性 DMA 缓冲区

DMA 映射包括两个方面的工作：分配一片 DMA 缓冲区；为这片缓冲区产生设备可访问的地址。同时，DMA 映射也必须考虑 Cache 一致性问题。内核中提供了如下函数以分配一个 DMA 一致性的内存区域：

```
void * dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t *handle,
                           gfp_t gfp);
```

上述函数的返回值为申请到的 DMA 缓冲区的虚拟地址，此外，该函数还通过参数 handle 返回 DMA 缓冲区的总线地址。handle 的类型为 dma_addr_t，代表的是总线地址。

dma_alloc_coherent() 申请一片 DMA 缓冲区，以进行地址映射并保证该缓冲区的 Cache 一致性。与 dma_alloc_coherent() 对应的释放函数为：

```
void dma_free_coherent(struct device *dev, size_t size, void *cpu_addr,
                       dma_addr_t handle);
```

以下函数用于分配一个写合并 (Writecombining) 的 DMA 缓冲区：

```
void * dma_alloc_writecombine(struct device *dev, size_t size, dma_addr_t
                             *handle, gfp_t gfp);
```

与 dma_alloc_writecombine() 对应的释放函数 dma_free_writecombine() 实际上就是 dma_free_coherent()，它定义为：

```
#define dma_free_writecombine(dev,size,cpu_addr,handle) \
    dma_free_coherent(dev,size,cpu_addr,handle)
```

此外，Linux 内核还提供了 PCI 设备申请 DMA 缓冲区的函数 pci_alloc_consistent()，其原型为：

```
void * pci_alloc_consistent(struct pci_dev *pdev, size_t size, dma_addr_t *dma_addrp);
```

对应的释放函数为 pci_free_consistent()，其原型为：

```
void pci_free_consistent(struct pci_dev *pdev, size_t size, void *cpu_addr,
                         dma_addr_t dma_addr);
```

这里我们要强调的是，dma_alloc_xxx() 函数虽然是以 dma_alloc_ 开头的，但是其申请的区域不一定在 DMA 区域里面。以 32 位 ARM 处理器为例，当 coherent_dma_mask 小于 0xffffffff 时，才会设置 GFP_DMA 标记，并从 DMA 区域去申请内存。

在我们使用 ARM 等嵌入式 Linux 系统的时候，一个头疼的问题是 GPU、Camera、HDMI 等都需要预留大量连续内存，这部分内存平时不用，但是一般的做法又必须先预留

着。目前，Marek Szyprowski 和 Michal Nazarewicz 实现了一套全新的 CMA，(Contiguous Memory Allocator)。通过这套机制，我们可以做到不预留内存，这些内存平时是可用的，只有当需要的时候才被分配给 Camera、HDMI 等设备。

CMA 对上呈现的接口是标准的 DMA，也是一致性缓冲区 API。关于 CMA 的进一步介绍，可以参考 <http://lwn.net/Articles/486301/> 的文档《A deep dive into CMA》。

5. 流式 DMA 映射

并不是所有的 DMA 缓冲区都是驱动申请的，如果是驱动申请的，用一致性 DMA 缓冲区自然最方便，这直接考虑了 Cache 一致性问题。但是，在许多情况下，缓冲区来自内核的较上层（如网卡驱动中的网络报文、块设备驱动中要写入设备的数据等），上层很可能用普通的 kmalloc()、__get_free_pages() 等方法申请，这时候就要使用流式 DMA 映射。流式 DMA 缓冲区使用的一般步骤如下。

- 1) 进行流式 DMA 映射。
- 2) 执行 DMA 操作。
- 3) 进行流式 DMA 去映射。

流式 DMA 映射操作在本质上大多就是进行 Cache 的使无效或清除操作，以解决 Cache 一致性问题。

相对于一致性 DMA 映射而言，流式 DMA 映射的接口较为复杂。对于单个已经分配的缓冲区而言，使用 dma_map_single() 可实现流式 DMA 映射，该函数原型为：

```
dma_addr_t dma_map_single(struct device *dev, void *buffer, size_t size,
                           enum dma_data_direction direction);
```

如果映射成功，返回的是总线地址，否则，返回 NULL。第 4 个参数为 DMA 的方向，可能的值包括 DMA_TO_DEVICE、DMA_FROM_DEVICE、DMA_BIDIRECTIONAL 和 DMA_NONE。

dma_map_single() 的反函数为 dma_unmap_single()，原型是：

```
void dma_unmap_single(struct device *dev, dma_addr_t dma_addr, size_t size,
                      enum dma_data_direction direction);
```

通常情况下，设备驱动不应该访问 unmap 的流式 DMA 缓冲区，如果一定要这么做，可先使用如下函数获得 DMA 缓冲区的拥有权：

```
void dma_sync_single_for_cpu(struct device *dev, dma_handle_t bus_addr,
                             size_t size, enum dma_data_direction direction);
```

在驱动访问完 DMA 缓冲区后，应该将其所有权返还给设备，这可通过如下函数完成：

```
void dma_sync_single_for_device(struct device *dev, dma_handle_t bus_addr,
                                size_t size, enum dma_data_direction direction);
```

如果设备要求较大的 DMA 缓冲区，在其支持 SG 模式的情况下，申请多个相对较小的

不连续的 DMA 缓冲区通常是防止申请太大的连续物理空间的方法。在 Linux 内核中，使用如下函数映射 SG：

```
int dma_map_sg(struct device *dev, struct scatterlist *sg, int nents,
    enum dma_data_direction direction);
```

nents 是散列表（scatterlist）入口的数量，该函数的返回值是 DMA 缓冲区的数量，可能小于 nents。对于 scatterlist 中的每个项目，dma_map_sg() 为设备产生恰当的总线地址，它会合并物理上临近的内存区域。

scatterlist 结构体的定义如代码清单 11.14 所示，它包含了与 scatterlist 对应的页结构体指针、缓冲区在页中的偏移（offset）、缓冲区长度（length）以及总线地址（dma_address）。

代码清单 11.14 scatterlist 结构体

```
1 struct scatterlist {
2     #ifdef CONFIG_DEBUG_SG
3         unsigned long    sg_magic;
4     #endif
5         unsigned long    page_link;
6         unsigned int     offset;
7         unsigned int     length;
8         dma_addr_t      dma_address;
9     #ifdef CONFIG_NEED_SG_DMA_LENGTH
10        unsigned int    dma_length;
11    #endif
12 };
```

执行 dma_map_sg() 后，通过 sg_dma_address() 可返回 scatterlist 对应缓冲区的总线地址，sg_dma_len() 可返回 scatterlist 对应缓冲区的长度，这两个函数的原型为：

```
dma_addr_t sg_dma_address(struct scatterlist *sg);
unsigned int sg_dma_len(struct scatterlist *sg);
```

在 DMA 传输结束后，可通过 dma_map_sg() 的反函数 dma_unmap_sg() 除去 DMA 映射：

```
void dma_unmap_sg(struct device *dev, struct scatterlist *list,
    int nents, enum dma_data_direction direction);
```

SG 映射属于流式 DMA 映射，与单一缓冲区情况下的流式 DMA 映射类似，如果设备驱动一定要访问映射情况下的 SG 缓冲区，应该先调用如下函数：

```
void dma_sync_sg_for_cpu(struct device *dev, struct scatterlist *sg,
    int nents, enum dma_data_direction direction);
```

访问完后，通过下列函数将所有权返回给设备：

```
void dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg,
    int nents, enum dma_data_direction direction);
```

在 Linux 系统中可以用一个相对简单的方法预先分配缓冲区，那就是同步“mem=”参数预留内存。例如，对于内存为 64MB 的系统，通过给其传递 mem=62MB 命令行参数可以使得顶部的 2MB 内存被预留出来作为 I/O 内存使用，这 2MB 内存可以被静态映射，也可以被执行 ioremap()。

6. dmaengine 标准 API

Linux 内核目前推荐使用 dmaengine 的驱动架构来编写 DMA 控制器的驱动，同时外设的驱动使用标准的 dmaengine API 进行 DMA 的准备、发起和完成时的回调工作。

和中断一样，在使用 DMA 之前，设备驱动程序需首先向 dmaengine 系统申请 DMA 通道，申请 DMA 通道的函数如下：

```
struct dma_chan *dma_request_slave_channel(struct device *dev, const char *name);
struct dma_chan *__dma_request_channel(const dma_cap_mask_t *mask,
                                       dma_filter_fn fn, void *fn_param);
```

使用完 DMA 通道后，应该利用如下函数释放该通道：

```
void dma_release_channel(struct dma_chan *chan);
```

之后，一般通过如代码清单 11.15 的方法初始化并发起一次 DMA 操作。它通过 dmaengine_prep_slave_single() 准备好一些 DMA 描述符，并填充其完成回调为 xxx_dma_fini_callback()，之后通过 dmaengine_submit() 把这个描述符插入队列，再通过 dma_async_issue_pending() 发起这次 DMA 动作。DMA 完成后，xxx_dma_fini_callback() 函数会被 dmaengine 驱动自动调用。

代码清单 11.15 利用 dmaengine API 发起一次 DMA 操作

```

1 static void xxx_dma_fini_callback(void *data)
2 {
3     struct completion *dma_complete = data;
4
5     complete(dma_complete);
6 }
7
8 issue_xxx_dma(...)
9 {
10    rx_desc = dmaengine_prep_slave_single(xxx->rx_chan,
11                                         xxx->dst_start, t->len, DMA_DEV_TO_MEM,
12                                         DMA_PREP_INTERRUPT | DMA_CTRL_ACK);
13    rx_desc->callback = xxx_dma_fini_callback;
14    rx_desc->callback_param = &xxx->rx_done;
15
16    dmaengine_submit(rx_desc);
17    dma_async_issue_pending(xxx->rx_chan);
18 }
```

11.7 总结

外设可处于 CPU 的内存空间和 I/O 空间，除 x86 外，嵌入式处理器一般只存在内存空间。在 Linux 系统中，为 I/O 内存和 I/O 端口的访问提高了一套统一的方法，访问流程一般为“申请资源→映射→访问→去映射→释放资源”。

对于有 MMU 的处理器而言，Linux 系统的内部布局比较复杂，可直接映射的物理内存称为常规内存，超出部分为高端内存。`kmalloc()` 和 `_get_free_pages()` 申请的内存在物理上连续，而 `vmalloc()` 申请的内存在物理上不连续。

DMA 操作可能导致 Cache 的不一致性问题，因此，对于 DMA 缓冲，应该使用 `dma_alloc_coherent()` 等方法申请。在 DMA 操作中涉及总线地址、物理地址和虚拟地址等概念，区分这 3 类地址非常重要。