Linux Kernel Development

Copyright © 2004 by Sams Publishing

International Standard Book Number: 0-672-32512-8

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

When reviewing corrections, always check the print number of your book. Corrections are made to printed books with each subsequent printing. To determine the printing of your book, view the copyright page. The print number is right-most number on the line below the "First Printing" line. For example, the following indicates the 4th printing of a title.

First Printing: August 2003

06 05 04 03 10 9 8 7 6 5 4

Misprint	Correction
Page 12, first paragraph The kernel stack is neither large nor dynamic; it is small and fixed in size. The kernel stack is fixed at **4** KB on 32-bit architectures and **8** KB on 64-bit architectures.	The kernel stack is neither large nor dynamic; it is small and fixed in size. The kernel stack is fixed at **8** KB on 32-bit architectures and **16** KB on **most** 64-bit architectures.
Page 34, Figure 3.1 Bottom row of figure Minimum Default Maximum 10 ms **150**ms **300**ms	Minimum Default Maximum 10 ms **100**ms **200**ms
Page 39, code snippet at bottom of page `struct prio_array array = rq->active;`	`struct prio_array*array = rq->active;`
Page 42, Table 3.1, last line Maximum **3200**ms high low	Maximum **200**ms high low
Page 44, first paragraph Wait queues are created statically via **DECLARE_WAIT_QUEUE_HEAD()** or dynamically via init_waitqueue_head().	Wait queues are created statically via **DECLARE_WAITQUEUE()** or dynamically via init_waitqueue_head().
Page 48, last paragraph The first change in supporting kernel preemption was the addition of a preemption counter, preempt_count, to each process's **task_struct**. This counter begins...	The first change in supporting kernel preemption was the addition of a preemption counter, preempt_count, to each process's **thread_info** structure. This counter begins...

page 49, first line after the "Real-Time" heading Linux provides two real-time scheduling policies, **SCHED_FF** and SCHED_RR.	Linux provides two real-time scheduling policies, **SCHED_FIFO** and SCHED_RR.
Page 50, Table 3.3: The descriptions for **sched_setaffinity()** and **sched_getaffinity()** are swapped.	
Page 62, code snippet at bottom of page #define NR_open 5	#define_NR_open 5
Page 63, last bullet on page Chapter **13**, "Virtual Filesystems," provides more details.	Chapter **11**, "Virtual Filesystems," provides more details.
Page 67, text in "Top Halves Versus Bottom Halves" section The bottom half runs later, at a more conventient time, with all interrupts **disabled…**	The bottom half runs later, at a more conventient time, with all interrupts **enabled…**
Page 75, first paragraph If so, it calls **hardware_**irq_event() to run the installed interrupt handlers for the line.	If so, it calls **handle_**irq_event() to run the installed interrupt handlers for the line.
Page 77, second paragraph in "Interrupt Control" section Neither disabling interrrupt **deliver** nor disabling kernel preemption provides any protection from concurrent access from another processor, however.	Neither disabling interrrupt **delivery** nor disabling kernel preemption provides any protection from concurrent access from another processor, however.

Page 79, paragraph before "Status of the Interrupt System" Disabling the line disables interrupt **deliver** for *all* devices on the line.	Disabling the line disables interrupt **delivery** for *all* devices on the line.
Page 80, all occurences of **deliver** in Table 5.1 should be **delivery**	
Page 83, fourth paragraph The top half could mark whether the bottom half would run by **sitting** a bit in a 32-bit integer.	The top half could mark whether the bottom half would run by **setting** a bit in a 32-bit integer.
Page 89, last paragraph Tasklets are represented by the **tasklist**_struct structure.	Tasklets are represented by the **tasklet**_struct structure.
Page 90, first paragraph in "Scheduling Tasklets" section *Scheduled* tasklets (the equivalent of raised softirqs) are stored in two per-processor structures: **tasklist**_vec (for regular tasklets) and tasklet_hi_vec (for high-prioirty tasklets).	*Scheduled* tasklets (the equivalent of raised softirqs) are stored in two per-processor structures: **tasklet**_vec (for regular tasklets) and tasklet_hi_vec (for high-prioirty tasklets).
Page 90, third bullet Add the tasklet to-be-scheded to the head of the tasklet_vec or **tasklist**_hi_vec linked list, which is unique to each processor in the system.	Add the tasklet to-be-scheded to the head of the tasklet_vec or **tasklet**_hi_vec linked list, which is unique to each processor in the system.
Page 92, second paragraph Both of these macros statically create a struct **tasklist**_struct with the given name.	Both of these macros statically create a struct **tasklet**_struct with the given name.

Page 99, last paragraph To create the structure statically at **run**-time:	To create the structure statically at **compile**-time:
page 125, second paragraph below the "Spin Locks and Bottom Halves" heading Because a bottom half may preempt process context code, if data is shared between a bottom half process context, you must protect the data in process context with both a lock and the disabling of bottom halves.	Because a bottom half may preempt process context code, if data is shared between a bottom half **and** process context, you must protect the data in process context with both a lock and the disabling of bottom halves.
Page 128, first paragraph …one of the tasks on the wait queue will be **awakened** up so that it can acquire the semaphore.	…one of the tasks on the wait queue will be **woken** up so that it can acquire the semaphore.
Page 128, middle paragraph, next-to-last sentence Additionally, unlike spin locks, semaphores do not disable kernel preemption and, consequently, code holding a **spin lock** can be preempted.	Additionally, unlike spin locks, semaphores do not disable kernel preemption and, consequently, code holding a **semaphore** can be preempted.
Page 132, paragraph before Table 8.7 After the event has occurred, calling **complete()** signals **all** waiting **tasks** to wake up.	After the event has occurred, calling **complete()** signals **a** waiting **task** to wake up.
Page 133, Table 8.7 Signals **any** waiting **tasks** to wake up	Signals **a** waiting **task** to wake up
Page 153, paragraph in middle of page The **xtime.v_nsec** value stores the number of nanoseconds that have elapsed in the last second.	The **xtime.tv_nsec** value stores the number of nanoseconds that have elapsed in the last second.

Page 165, paragraph after second bulleted list The actual use and layout of the memory zones is architecture **independent**.	The actual use and layout of the memory zones is architecture **dependent**.
Page 172, Table 10.5, entry for GFP_HIGHUSER This is an allocation from **ZOME**_HIGHMEM and might block.	This is an allocation from **ZONE**_HIGHMEM and might block.
Page 173, second paragraph On the far other end of the spectrum is the **GFP**_ATOMIC flag.	On the far other end of the spectrum is the **GTP**_ATOMIC flag.
page 181, first line on the page This creates a cache named **task_struct**, which stores objects of type struct task_struct.	This creates a cache named **task_struct_cachep**, which stores objects of type struct task_struct.
Page 194, third line **/* file creation timestamp */**	**/* inode change time */**
Page 208, second paragraph in "Data Structures Associated with a Process" section The address of this table is pointed to by the **files** entry in the **processor** descriptor.	The address of this table is pointed to by the **files** entry in the **process** descriptor.
Page 215, paragraph before "The bio structure" …into **many** multiple buffer_head structures.	…into multiple buffer_head structures.
Page 217, first paragraph after first code snippet In each given block I/O operation, there are bi_vcnt vectors in the bio_vec array starting with **bi_io_vecs**.	In each given block I/O operation, there are bi_vcnt vectors in the bio_vec array starting with **bi_io_vec**.

Page 217, paragraph in middle of page **Table 12.3** is a diagram of the relationship between the bio structure, the bio_vec structure, and the page structure.	**Figure 12.2** is a diagram of the relationship between the bio structure, the bio_vec structure, and the page structure.
page 226, footnote 3 at the bottom of the page Newer versions of glibc implement malloc() via mmap() and **not** brk().	Newer versions of glibc implement malloc() via mmap() and brk().
Page 230, next-to-last paragraph Thus, vm_end - **vm_start** is the size (length) in bytes of the interval.	Thus, vm_end - **vm_start+1** is the size (length) in bytes of the interval.
page 231, 8th entry in Table 13.1 VM_MAYSHARE The **VM_SHARE** flag can be set	VM_MAYSHARE The **VM_SHARED** flag can be set
Page 244, first paragraph in "The address_space Object" section Checking the page cache to see if certain data has been cached is rendered more difficult because of the noncontinguous nature of the blocks that **can** up each page.	Checking the page cache to see if certain data has been cached is rendered more difficult because of the noncontinguous nature of the blocks that **can make** up each page.
page 245, second to last line of code struct address_space *assoc_mapping; /* associated **buffres** */	struct address_space *assoc_mapping; /* associated **buffers** */
Page 257, first paragraph The lone disadvantage of a circular buffer—the possibility of **loosing** messages—is a small price to pay for the simplicity and robustness it affords.	The lone disadvantage of a circular buffer—the possibility of **losing** messages—is a small price to pay for the simplicity and robustness it affords.

Page 276, first paragraph in "Byte Order" section The byte ordering is called *big-endian* if the most significant byte is **encoding** first with the remaining bytes decreasing in significance.	The byte ordering is called *big-endian* if the most significant byte is **encoded** first with the remaining bytes decreasing in significance.
Page 278, first line of code **u23** __cpu_to_be32(u32);	**u32** __cpu_to_be32(u32);

This errata sheet is intended to provide updated technical information. Spelling and grammar misprints are updated during the reprint process, but are not listed on this errata sheet.