

To

Bill Hackenberger

My fellow business partner at HighCloud Security where we certainly lived through the Silicon Valley experience of starting a company, going up and down Sand Hill Road raising money, building a team, products, getting customers, running low on money more times than we'd care to remember but ultimately leading to an exit. Not the big one we hoped for but it was a journey to remember for sure.

Contents

1	Introduction	13
1.1	About me	14
1.2	Isn't Everything Documented On-line?	17
1.3	Why Isn't the Material Open Source?	17
1.4	Feedback	17
1.5	Do you do anything for fun?	17
1.6	Conventions	18
1.6.1	What Error Checking?	18
1.6.2	Source Code References	19
1.6.3	Web References	19
1.7	I Versus we vs	19
1.8	Who Should Read This Book?	20
1.9	What's in the Book?	20
1.10	Why is Source Code Missing?	23
1.11	KGDB Demonstrations	23
1.12	Source Code Figures	24
1.13	Acknowledgements	24
2	File and Filesystem Programming	27
2.1	Programming Standards and Portability	27
2.1.1	What are all These Standards?	28
2.1.2	LSB — Linux Standard Base	30
2.2	Manual Pages	31
2.3	System Calls vs Library Functions – an Overview	32
2.4	Error Handling with <code>errno</code>	33
2.5	File Types and Mode	34
2.6	File Descriptors	35
2.6.1	How Many Open Files?	37
2.6.2	Closing Files	37
2.6.3	Duplicating File Descriptors	38
2.6.4	Named Pipes	41
2.7	Opening Files / Creating Files	41
2.7.1	Open Flags	42
2.8	Reading and Writing Files	44

2.9	Vectored Reads and Writes	45
2.10	File and Record Locking	48
2.10.1	Advisory File Locking	48
2.10.2	An Example of Advisory Locking	49
2.10.3	Viewing Existing Locks	52
2.10.4	Mandatory File Locking	52
2.10.5	Open File Description Locks	53
2.11	Synchronizing I/O Operations	54
2.12	Direct I/O	54
2.13	Sparse Files	56
2.13.1	Sparse Files in Action	56
2.13.2	Sparse File Example 1 — Virtual Machines	58
2.13.3	Sparse File Example 2 — HSM Applications	59
2.14	Buffered I/O and the Standard I/O Library	59
2.14.1	The Standard I/O Library FILE Structure	60
2.14.2	Analyzing the glibc Standard I/O Library	63
2.15	Memory Mapped Files	65
2.15.1	The mmap (2) / munmap () System Calls	66
2.15.2	Other Mapped File Functions	67
2.16	Asynchronous I/O	68
2.16.1	Additional Async I/O Functions	72
2.16.2	Performance Gains with Async I/O	72
2.16.3	procfs Async I/O Information	73
2.17	Truncating Files	73
2.18	Multi-threaded Applications	74
2.19	Directory Creation	75
2.20	Hard Links and Symbolic Links	75
2.20.1	Symbolic Links	77
2.21	Extended Attributes	77
2.21.1	Installing and Using Extended Attributes	78
2.22	Inode Flags	79
2.23	Reading Directory Entries	79
2.23.1	A Simple Implementation of the ls (1) Command	80
2.23.2	Other Directory Functions	82
2.23.3	Directory Entries at the System Call Level	84
2.24	File Notification	84
2.25	Filesystem-level Programming Interfaces	87
2.25.1	Mounting and Unmounting Filesystems	87
2.25.2	Getting Filesystem Statistics	88
2.25.3	Filesystem Sync	89
2.25.4	Changing Filesystem Properties	90
2.26	Conclusion	90
3	Filesystems	91
3.1	The Linux File Hierarchy	91

3.2	How Simple it Used to be	93
3.3	Filesystems, Disks and Partitions	93
3.3.1	Filesystem On-disk Structure	94
3.3.2	Referencing Data Blocks From The Inode	95
3.3.3	Full Filesystem Check vs Journaling	96
3.3.4	Disk Partitioning	96
3.3.5	The GPT Disk Format	97
3.3.6	Logical Volumes Management	98
3.3.7	Advantages of Logical Volume Management	98
3.3.8	Linux LVM	99
3.3.9	LVM Concepts	100
3.4	ZFS – Builtin Volume Management	102
3.5	Filesystem Types Supported by Linux	103
3.6	How Many Filesystems are Actually Used?	104
3.7	Comparing the Main Filesystem Types	105
3.7.1	The ext2/3/4 Filesystems	105
3.7.2	The XFS Filesystems	108
3.7.3	The btrfs Filesystems	109
3.7.4	The Flash??? Filesystems	110
3.7.5	Stratis - RHEL???	110
3.8	Kernel or Userspace?	110
3.9	Pseudo Filesystems	111
3.9.1	The proc Filesystem	112
3.9.2	The Ramfs Filesystem	113
3.9.3	The tmpfs Filesystem	114
3.9.4	The pipe Filesystem	114
3.9.5	The AuFS Filesystem	114
3.9.6	The OverlayFS Filesystem	114
3.10	FUSE-based Filesystems	114
3.10.1	Demonstrating the SSHFS FUSE Filesystem	115
3.11	Physical Filesystems and Partitioning	116
3.12	Linux Namespaces	117
3.12.1	Shared Subtrees	118
3.12.2	Root and <code>chroot</code> Environments	119
3.12.3	Linux cgroups to Isolate and Manage Resources	119
3.12.4	Mount Namespace Example	120
3.12.5	Containers	120
3.12.6	Creating a Container by Hand	120
3.13	Filesystem Operations	121
3.13.1	Making a Filesystem	121
3.13.2	Mounting Filesystems	122
3.13.3	The Mount Table	122
3.13.4	Automounting Filesystems	124
3.13.5	Unmounting Filesystems	124
3.13.6	The <code>lsof(1)</code> Command	124

3.13.7	Reporting File System Disk Space Usage	125
3.13.8	Fixing Filesystems with <code>fsck</code>	125
3.13.9	Filesystem Debugging	126
3.14	Loopback Mounts	127
3.14.1	Demonstrating Loopback Devices	129
3.15	Client Caching With FS-Cache	130
3.16	Network Filesystems	131
3.16.1	NFS – the Network File System	131
3.17	Backup / restore	134
3.17.1	Freezing and Thawing Filesystems	134
3.18	Quotas	135
3.18.1	An Example of Using Filesystem Quotas on Ubuntu	136
3.19	Swap space	138
3.20	The Boot Process and Run Levels	140
3.20.1	Run Level Scripts	140
3.21	Conclusion	142
4	The Linux Kernel Source Code	143
4.1	User-Space vs Kernel-Space	147
4.2	The System Call Interface	148
4.2.1	System Calls for File / Filesystem Activity	150
4.2.2	Using <code>vim</code> and <code>ctags</code> to Browse the Kernel Source	151
4.2.3	Using <code>cscope</code> to Browse the Kernel Source	154
4.2.4	Using Elixir to Browse the Kernel Source	156
4.2.5	Maintaining Your Own Notes	156
4.3	File Access Structures	158
4.3.1	File Descriptors	159
4.3.2	The <code>fdtable</code> Structure	160
4.3.3	The <code>file</code> Structure	161
4.3.4	The <code>inode</code> Structure	162
4.4	The Directory Cache	163
4.4.1	Introducing The <code>dentry</code> Structure	163
4.4.2	Dcache Hash Buckets	166
4.4.3	KGDB – Analyzing The Dcache	167
4.4.4	Overriding / Supporting Dcache Operations	169
4.5	The Inode Cache	169
4.5.1	KGDB – Analyzing Per-File Kernel Structures	172
4.5.2	Analyzing Per-File Kernel Structures Using <code>crash</code>	174
4.5.3	KGDB — Multiple dentries Per Single File	175
4.6	The Buffer Cache	176
4.6.1	Buffer Cache Size / Usage	178
4.6.2	Flushing Buffers	179
4.7	The Page Cache	179
4.7.1	The <code>page</code> Structure	181
4.7.2	Compound Pages / Page Folios	181

4.7.3	KGDB — Analyzing Page Cache Structures	183
4.7.4	Process Address Space	184
4.8	Mounted Filesystem Structures	185
4.8.1	The superblock and mount Structures	186
4.8.2	The vfsmount Structure	187
4.8.3	KGDB — Analyzing the List of Mounted Filesystems	187
4.9	Namespace Structures	188
4.10	Conclusion	189
5	Common Linux Types and Locks	191
5.1	Lists	191
5.1.1	An Unusual List Example	191
5.2	Locks	192
5.3	The xarray Structure	193
5.3.1	The Big Kernel Lock (BKL)	193
5.3.2	Local Locks	193
5.3.3	Semaphores	193
5.3.4	Mutexes	194
5.4	Black-Red Trees	194
5.5	Wrapping Structures With <code>container_of</code>	194
5.6	Semaphores, Mutexes, And Lockless Algorithms	196
5.7	Read-Copy Update (RCU) Synchronization	197
5.8	Conclusion	197
6	The VFS Layer	199
6.1	The VFS Entry Point	199
6.2	VFS to Filesystem Interfaces	200
6.3	Registering Filesystems	202
6.3.1	Unregistering Filesystems	203
6.3.2	KGDB — Analyzing the List of Registered Filesystems	204
6.4	Mounting and Unmounting Filesystems	205
6.4.1	Legacy Mounts	206
6.4.2	Mounting Using Filesystem Context	208
6.4.3	Mounting Filesystems at the VFS Layer	210
6.4.4	KGDB – A Look at Structures Following Mount	213
6.4.5	Unmounting Filesystems	214
6.5	Namespace Creation and Handling	215
6.5.1	Handling <code>chroot(2)</code>	215
6.6	KGDB – Analyzing Mount Namespace Creation	216
6.7	Opening a File / Pathname Resolution	217
6.7.1	File Descriptor Allocation and File Table Expansion	218
6.7.2	Allocating a <code>file</code> Structure	220
6.7.3	KGDB – Viewing File Descriptor Allocation	220
6.8	Pathname Resolution	221
6.8.1	Setting the Stage for Pathname Resolution	223

6.8.2	Initialization Structures And Starting The Process	225
6.8.3	The Nitty Gritty of Pathname Resolution	226
6.8.4	Callers of <code>link_path_walk()</code>	230
6.8.5	More on Hashing	230
6.8.6	Handling a dcache Miss	231
6.8.7	Crossing mount points	231
6.8.8	Handling Symlinks	234
6.8.9	Returning From <code>link_path_walk()</code>	234
6.8.10	RCU-walk vs REF-walk	235
6.8.11	KGDB – Setting The Stage For Pathname Resolution	236
6.8.12	KGDB – Inside <code>link_path_walk()</code> and Friends	239
6.8.13	KGDB – Crossing Mount Points	242
6.8.14	Further Information on Pathname Resolution	244
6.9	Linux Dcache Implementation	244
6.9.1	The Migration to RCU	245
6.9.2	Linking Dentries Together	245
6.9.3	The <code>dentry</code> State Diagram	246
6.9.4	KGDB – Monitoring dentry Transition State	249
6.9.5	Digging Deeper on The Core Functions	250
6.9.6	Pruning The Dcache	256
6.9.7	KGDB – Analyzing Dentries For a Simple File Hierarchy	257
6.9.8	KGDB – Inspecting the Per-Filesystem LRU List	260
6.9.9	Dcache Statistics	260
6.10	The Inode Cache Implementation	261
6.10.1	Inode Locks	261
6.11	The Buffer Cache Implementation	261
6.12	File Creation	261
6.12.1	Regular File Creation	262
6.12.2	Creating Directories	264
6.12.3	Creating Symbolic Links	266
6.12.4	Creating Hard Links and Device Nodes	267
6.13	Reading Directory Entries	267
6.14	The Linux Page Cache	269
6.14.1	Compound Pages	269
6.14.2	Memory Folios	271
6.14.3	Memory Mapped Files	272
6.15	Reading Files	272
6.15.1	The <code>new_sync_read()</code> and <code>generic_file_read_iter()</code> Functions	274
6.15.2	Vectored Reads	275
6.16	Writing Files	276
6.16.1	Vectored Writes	276
6.17	Direct I/O	277
6.18	File Locking Implementation	277
6.19	Dissecting The proc Filesystem	277

6.19.1	Use of Red-Black Trees in /proc	278
6.19.2	KGDB — Analyzing the /proc File Tree	279
6.20	Quotas Implementation	280
6.21	Handling Pipes	280
6.22	Flushing File Data and Metadata	280
6.23	Filesystem Interfaces	280
6.23.1	The super_operations Structure	280
6.23.2	The inode_operations Structure	282
6.23.3	The file_operations Structure	283
6.23.4	The address_space_operations Structure	285
6.24	Conclusion	287
7	Building a Linux Filesystem	289
7.1	The SPFS on-disk Format	290
7.2	A Note About Printing Pointers	291
7.3	From Module Load to Mounting a Filesystem	291
7.4	Initializing the per-filesystem Inode Cache	293
7.4.1	KGDB - Analyzing Inode Lists for a Specific Mountpoint	295
7.5	Mounting a Filesystem	299
7.5.1	Inside sp_mount and sp_fill_super	300
7.6	SPFS super_operations	301
7.6.1	KGDB – Analyzing the Effects of a Mount Operation	301
7.7	Unmounting	303
7.8	Creating a File	304
7.9	Creating a File	304
7.9.1	Inside sp_find_entry()	305
7.9.2	Inside sp_create_file()	307
7.9.3	Inside sp_new_inode()	308
7.9.4	Inside sp_ialloc()	308
7.9.5	More on Negative dentries	309
7.10	Creating a Directory	309
7.10.1	KGDB – Handling the mkdir(2) System Call	311
7.11	Reading Directory Entries	312
7.11.1	KGDB – Kernel Paths for Reading Directories	313
7.12	Writing to a File	315
7.12.1	KGDB – Handling the write(2) System Call	317
7.13	Reading from a File	320
7.13.1	KGDB – Handling the read(2) System Call	322
7.14	Memory Mapped Files	324
7.15	Removing a File	324
7.15.1	KGDB – Handling the unlink(2) System Call	326
7.16	Renaming a File	327
7.16.1	KGDB – Handling the rename(2) System Call	327
7.17	Other Operations	328
7.18	Obtaining Filesystem Information via statfs	329

7.18.1	KGDB – Handling the <code>statfs(2)</code> System Call	330
7.19	Hard Links and Symbolic Links	331
7.20	Integration with <code>/proc</code>	332
7.21	The SPFS <code>fsdb</code> Command	332
7.22	File Undelete	334
7.22.1	Enhancing Undelete	336
7.22.2	Why Undelete is More Complex Than you Might Think?	336
7.23	How to Test SPFS	337
7.23.1	Testing a Commercial Filesystem	337
7.24	An Extended SPFS Filesystem	338
7.25	Notes	339
7.26	<code>debugfs?</code>	339
7.27	Conclusion	339
8	The FUSE Filesystem Framework	341
8.1	A Background on FUSE	341
8.2	Source Code Files	343
8.2.1	Manual Pages	343
8.2.2	Understanding <code>FUSE_VERSION</code>	343
8.3	The FUSE Architecture	345
8.3.1	An Example to Get Started	347
8.4	Digging Under The Covers	348
8.4.1	FUSE Kernel Startup	348
8.4.2	Establishing Connection between the Kernel and <code>libfuse</code>	349
8.4.3	Mounting a <code>fuse</code> Filesystem	351
8.4.4	General Flow Between Kernel and User-space	352
8.4.5	FUSE VFS Interfaces	353
8.4.6	FUSE to <code>libfuse</code> Request Queues	356
8.4.7	Kernel to <code>libfuse</code> Messages	359
8.4.8	Controlling FUSE Through <code>fusectl</code>	360
8.4.9	Multithreading	361
8.4.10	Time to FORGET	362
8.4.11	FUSE Interrupts	362
8.4.12	Per-File DAX Option	363
8.4.13	The <code>fuseblk</code> Filesystem	363
8.4.14	FUSE and Fsnotify	364
8.4.15	User-space Character Devices	364
8.5	More on <code>libfuse</code>	364
8.5.1	The <code>fuse_file_info</code> Structure	364
8.6	Implementing a FUSE-based Filesystem	365
8.6.1	Installing FUSE and Compiling the Filesystem	365
8.6.2	Getting Started	365
8.6.3	Mounting the Filesystem	366
8.6.4	Initialization and Startup	368
8.6.5	What Happens During Filesystem Operations?	369

8.6.6	Running pSPFS in The Background	371
8.7	Example of Using The Low-level FUSE Interface	372
8.7.1	TFS Definitions, Initialization and Startup	373
8.7.2	TFS Startup and Initialization	374
8.7.3	Updating the Kernel Inode Buffers	375
8.7.4	TFS Operations	376
8.7.5	Low-level FUSE Operation Definitions	378
8.8	Adding Encryption to pSPFS	378
8.8.1	A Primer on Encryption	379
8.8.2	How The eSPFS Filesystem Works	380
8.8.3	Extending eSPFS	384
8.9	FUSE Performance	384
8.9.1	Improving Performance with eBPF	385
8.10	Conclusion	386
9	Filesystem Debugging	387
9.1	Kernel Debugging with kgdb	387
9.1.1	Building a Kernel with kgdb Support	388
9.1.2	Turning off Automatic Updates	390
9.1.3	Connecting gdb to the Target VM	390
9.1.4	gdb Command Reference	391
9.1.5	Linux Kernel Helper Functions	392
9.1.6	Basic gdb Examples	393
9.1.7	Walking Lists	394
9.1.8	Setting Breakpoints and Stepping Through Code	394
9.1.9	Calling <code>container_of()</code>	396
9.1.10	Listing Source Code	396
9.1.11	Getting Symbols From Loadable Modules	397
9.1.12	A Script to Load Modules	400
9.1.13	gdb User-Defined Commands	401
9.1.14	I Want to Edit the Command History Using vi	401
9.2	Exploring File/Filesystem Information with eBPF	401
9.2.1	Tracing Kernel Functions with eBPF	402
9.2.2	Warning	403
9.3	Debugging With The <code>crash(8)</code> Command	403
9.3.1	Installing crash and Linux Debugging Information	404
9.3.2	Upgrading the Kernel	405
9.3.3	Kernel Compilation Issues	406
9.3.4	Running crash	406
9.3.5	Exploring Lists With crash	409
9.3.6	Equivalent of <code>container_of()</code> With crash	409
9.3.7	Exploring Trees With crash	410
9.4	Conclusion	410
10	Filesystem Security	411

10.1	It Starts With File and Directory Permissions	411
10.2	Encryption and Key Management	411
10.2.1	Password-based Encryption	413
10.2.2	Encryption Algorithms and Key Sizes	413
10.2.3	Hardware Acceleration of AES	413
10.2.4	Access Controls	414
10.2.5	EKM – Enterprise Key Management and FIPS	414
10.2.6	Common Criteria	415
10.2.7	HSMs – Hardware Security Modules	416
10.2.8	Protecting Keys Over the Wire	416
10.2.9	Protecting Encryption Keys in Memory	417
10.2.10	NIST Standards	418
10.2.11	Keep it Simple — Encrypt it Yourself	418
10.3	Zero Blocks / Extents on Allocation or Free	419
10.3.1	Fixing the Issue	421
10.3.2	A DIY Solution	422
10.4	Security is a Multi-Layered Approach	423
10.5	SELinux	423
10.6	AppArmor	424
10.7	SELinux vs AppArmor	424
10.8	Security Standards	425
10.9	Secure Programming	426
10.10	Conclusion	427
11	Filesystem Performance	429
11.1	Performance of Different Filesystems	429
11.1.1	The Phoronix Study	429
11.2	Mount Options	430
11.3	Block Sizes	430
11.4	Extend-based Allocation	430
11.5	The Effects of Filesystem Fragmentation	430
11.6	Open Flags etc etc	430
11.7	Volume Management / Disk Subsystems	430
11.8	Buffer Cache and Page Cache Tuning	430
11.9	Other commands???	430
11.10	The sysstat Package	430
11.11	eBPF – a New World of Opportunities	432
11.11.1	BCC – BPF Compiler Collection	432
11.11.2	The bpftrace Command	433
11.11.3	The Annual eBPF summit	434
11.12	Conclusion	434
12	Filesystem Case Studies	435
12.1	The Extended Filesystems	435
12.1.1	The ext4 Disk Layout	435

12.1.2 ext4 Superblock	436
12.1.3 ext4 Reserved Inodes	436
12.1.4 Exploring on-disk ext4 Data Structures	437
12.1.5 ext4 Journaling (jbd2)	437
12.1.6 Analyzing the ext4 Journal	438
12.2 XFS	438
12.2.1 The XFS Disk Layout	438
12.2.2 XFS Journaling	439
12.2.3 Analyzing the XFS Journal	439
12.3 The btrfs Filesystems	439
12.3.1 The btrfs Disk Layout	440
12.3.2 btrfs COW vs Journaling	440
12.4 NFS	440
12.4.1 Standardization and Protocols	440
12.4.2 Client-side NFS Handling	440
12.4.3 Server-side NFS Handling	440
12.4.4 NFS Locking	440
12.4.5 NFS File Delegation	440
12.4.6 Other NFS Implementations	440
12.5 Conclusion	441
13 Filesystem Forensics	443
13.1 Conclusion	443

Chapter 1

Introduction

Welcome to *Linux Filesystems – Under the Cover*, the first book dedicated to Linux filesystems in over 20 years but only the first book dedicated to Linux filesystem internals.

The first book that I could find was *Linux Filesystems* by William Von Hagen [11], a book that, to be honest, I didn't know existed until I searched while writing this section. Unlike Von Hagen's book, this book goes very deep into the Linux kernel virtual filesystem implementation as well as the implementation of several popular filesystems.

This book also follows on from my second book *UNIX Filesystems – Evolution, Design and Implementation* [9] that was published in 2003. It's hard to think that was now 20 years ago! That book covered filesystems across many versions of UNIX, microkernel-based implementations of UNIX and also Linux, just as it was moving to the 2.4 kernel series. At the time, my team at VERITAS had just finished porting the VERITAS filesystem VxFS. VxFS was a filesystem that ran on SVR4 UNIX variants, Solaris, HP-UX, AIX and Linux.

In 2004 I VERITAS left to start working on a series of books on the Linux kernel. I set a lofty goal of trying to do for Linux what Donald Knuth had done for computer algorithms. I didn't get very far getting pulled back into corporate America and spending the next twenty years working in various startups, two of my own. I was quite surprised in 2022 when I came back to book writing to discover that there still wasn't a great deal of information on Linux filesystems and very few Linux kernel books had been published in many years. There are some great articles on the web but they are spread across many different sites, quite sparse in specific areas and are often quite dated.

And thus begins my journey one more time. The kernel has become too vast to try the Donald Knuth approach so I decided to focus on filesystems, the part of the kernel I've worked on and been most interested in for over thirty years now. Even with a focus on filesystems there is a lot of material and throughout the process, I've gone back and forth on whether this should be one or two volumes. Passing the 400 page mark with a vast amount of material still to cover, I decided that it should be two volumes which was my original goal. The first volume focuses on generic filesystem implementation while the second goes into detail on specific filesystems and covers the more unusual aspects of filesystem access.

I still wanted to focus on the practical side of things so there are lots of examples given

using a variety of different tools and example filesystems in both the kernel and user-space. To support the book I'm providing course materials that you can find online and my own website with blog material and more detailed examples.

I hope you enjoy reading the book as much as I've enjoyed writing it.

1.1 About me

Despite writing an interpreter in 6502 assembler at High School and wanting to work in *systems programming*, my first introduction to operating systems kernel came in 1985 when one of the instructors at Leeds University (in the UK) gave an informal walkthrough of the 3BSD kernel source code. At this time, the BSD source code could be found on every system under `/sys`. I believe that the machine we were using was a DEC PDP-11. It was also around this time that Kernighan and Ritchie's "*The C Programming Language*" [5] was one of the few books available about C. If you wanted to learn about UNIX, Kernighan and Pike's "*The UNIX Programming Environment*" [4] was the book of choice having just been launched the year I started college. Both are now available free on-line.

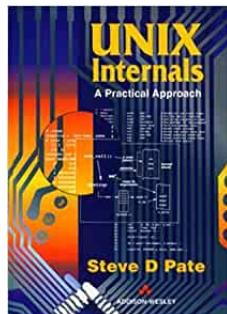
I was lucky to work on the Chorus Microkernel early during my career while I was at ICL (International Computers Limited), the UK's largest mainframe manufacturer, later acquired by Fujitsu. After working on European research projects based around Chorus, we started enhancing the SVR4 UNIX subsystem running on top of the microkernel to make it production ready for a Sparc-based distributed server we planned to ship. This involved porting the VERITAS filesystem (VxFS) and volume manager (VxVM) from vanilla SVR4 UNIX. I worked on the latter. Due to the very different interactions between filesystems and the virtual memory (VM) subsystems of both Chorus MiX (their version of SVR4) and standard SVR4 UNIX, the port of the filesystem was an arduous task. The volume manager less so (lucky for me). Over those years I gained an insight into the operations of different kernel implementations and my love of filesystems started.

In 1993 I joined SCO (Santa Cruz Operation) partly to help them to switch over from an SVR3 UNIX Filesystem Switch (FSS) architecture to support the Sun/USL VFS/vnode based architecture. As was ICL, we were looking to modernize our filesystem support and bring in support for newer filesystems such as VxFS which were light years ahead of anything on most UNIX variants at that time. Two things prevented that move. First, the cost of rewriting large parts of the kernel were prohibitive. Secondly, the cost of converting existing filesystems over to a new VFS/vnode architecture would also add significant expense. And on top of that, SCO acquired the rights to UNIX which then gave them SVR4 UNIX (and therefore "vnodes") anyway.

I led the SCO kernel architecture team whose job it was to look at more forward thinking features, new architectures like the upcoming RISC-based Intel P6 which was abandoned following their agreement with HP. We also started investigating a new microkernel-based architecture of UNIX to replace existing European telecom companies proprietary operating systems.

While running the architecture team, I wrote my first book covering SCO OpenServer UNIX internals [10], which was at that time, a much enhanced version of SVR3. We'd abandoned the idea of SVR4 VFS/vnodes but had implemented support for a journaling

filesystem from an east coast US company called Prologic. We also completely rewrote the virtual memory subsystem to support `mmap(2)` and therefore support of the SVR4 development environment (compiler etc). Much of that work was done by Hugh Dickens who went on to make many contributions to the Linux kernel and is still working on Linux for Google as I write.



To explain how things have become more complicated over the decades, in 1996, I was very happy to get my hands on Lyon's book "*A Commentary on the UNIX Operating System*" [6]. This described the 6th Edition UNIX kernel which was less than 9,000 LOC (Lines Of Code). At that time, I was working on porting the `truss(1)` command (UNIX equivalent of `strace(1)`) over to the Chorus microkernel. The `truss` command itself was around 10,000 LOC.

I was proud of this first book. It's not easy to write about a proprietary operating system but it started my journey in writing in as practical a way as possible to better help people understand the subject matter. The book had many programming examples and use of the `crash(1)` command to display the contents of kernel structures after user-space commands are run and system calls are made. I've followed a similar approach in this book.

I continued working on the Chorus microkernel as part of a consortium between SCO, Chorus and Siemens Private Networks. At first this was an SVR3-based UNIX subsystem running on Chorus but that was switched to SVR4.2 MP (multi-processing). Once again we started porting VxFS and VxVM over to the microkernel but made better choices to largely emulate SVR4 interfaces to simplify the port. And then SCO abandoned the project and started spiraling downwards as a company. I left the UK in 1996 and joined VERITAS to work in Mountain View, California. It was interesting, but at that time, SGI were building all around the area north of Shoreline. Years later, the vast majority of their buildings were taken over by Google and that's still where Google's headquarters is today.

As a side note, SCO used to hold its annual SCO Forum each summer at UC Santa Cruz. The summer before leaving SCO, I pulled together a panel session to discuss the future of operating systems and managed to get Bill Shannon (Sun employee 11), Michel Gien (co-founder of Chorus Systemes), someone representing the Mach microkernel whose name escapes me and Linus Torvalds. Linus was still living in Finland at the time so we flew him over to California to spend the week with us. Unfortunately I have no photos of the event. Linus's view was that the future of operating systems was the desktop. Perhaps that wasn't the right answer but at that time, who knew the future of server operating systems would be dominated by Linux? After searching to see where people still were, I was saddened to see that Bill Shannon died in 2020. You can read about him here:

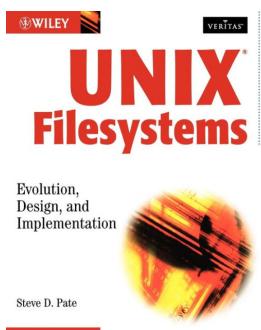


URL 1 – <https://tinyurl.com/4j9j496n>

As an operating system enthusiast, the VERITAS filesystem was a great place to work.

The VERITAS filesystem VxFS ran on a number of different operating systems. I initially started working on the portability project where we tried to share as much code as possible on UnixWare, HP-UX and Solaris. Later on we added AIX and I ran the team that ported VxFS to Linux.

While at VERITAS I wrote my second book. This one covered filesystems from a variety of different UNIX and UNIX-like operating systems. I included a sample Linux filesystem similar to the kernel-based filesystem presented in this book. At the time, this was for the 2.4.x kernel and a lot has changed since then.



Being in Silicon Valley, I was surrounded by startups and went to work for two unsuccessful storage companies. At the second we actually engaged with Hans Reiser to enhance ReiserFS. I shudder at the thought! I then moved to work for Vormetric, initially consulting on their cross-platform encryption filesystem product and then later as CTO. They were an enterprise encryption and key management company that encrypted at both the filesystem and volume layers, also for a wide array of operating systems (Solaris, AIX, HP-UX, Linux and Windows). I then spent the next 14 years working on virtualization security and encryption and key

management technologies in the two startups I co-founded, HyTrust and HighCloud Security. Both involved encrypting at the filesystem and device driver layer on both FreeBSD and Linux. As a strange twist of events, HyTrust acquired HighCloud in 2012. I stayed with HyTrust for 5 years post-acquisition before moving to Thales (who acquired Vormetric) where I worked with several old friends helping with the merger of Thales and Gemalto encryption technologies.

That brings us to the present day and back to book writing but now full time. I looked to see what material was out there about Linux filesystems and surprised to see that there was a gap that needed to be filled. And thus, I started coding again for the first time in years (apart from writing Space Invaders in Python/pygame) with the goal to build simple filesystems that are easy to explain and play with. They are all included on this book and github with additional examples on my website.

This book is allowing me to achieve something I've wanted to do for many years and that is to write it and typeset it using \LaTeX which I have used for many years. I actually typeset my first two books. My first book was written using Microsoft Word which actually worked fine but struggled with such a large number of pages. I was disappointed with FrameMaker when typesetting my second book. The program just didn't seem as simple and easy to use as it did when I used the early versions of FrameMaker on my old Sun Sparcstation 1.

Book writing is a lot of work for little financial gain. My last two books didn't exactly make me rich! This time around I intend to self-publish so I can hopefully make more money especially given that I am now writing books and educational material as a full-time job. It will also be quite a learning curve to understand the book industry beyond just writing and typesetting. It's a path I started back in 2014 with help from my first editor at Addison

Wesley, Karen Mosman. It's only taken 19 years to finally get there.

1.2 Isn't Everything Documented On-line?

The short answer is no. The longer answer is that some aspects of the kernel's filesystem-independent and filesystem-dependent code is documented and there is quite a lot of detailed information available on very specific topics. For example, some of the nitty gritty details around pathname resolution are documented, specifically around fast paths and locking. But even with this level of detail, the high-level picture of pathname resolution and with figures and examples, are very hard to come by.

I had no intention of cutting and pasting text from online and I think that too much detail isn't worth it in many areas. If you get the big picture, you can then go further to get the details. Or perhaps the big picture is enough for your goals. Where I have taken material from on-line, I reference the material and often, I'll tell you to read the on-line documentation to go further. I'll also be writing blog posts to pull things together further **XXX**.

1.3 Why Isn't the Material Open Source?

Since I'm writing about one of the largest open source projects of all time, I'm sure a lot of people will be wondering why write a book for profit and not just put the material online. My answer is quite simple. I can't afford to do that. This is the first book I'm writing while not currently employed and I need to make some money. After all, beer isn't free. My plan is to allow make on-line courses using the same material and have a web presence where I can blog, bring source code up to date and write about new filesystem changes that may make it into a future version. Perhaps one day I'll write a filesystem in Rust. Whether this can be profitable is to be seen. So in the meantime, I need to pay the bills.

1.4 Feedback

Because of self-publishing, I have more control over the content and distribution of material. I can print a smaller number of books and make additional editions to expand, correct and bring material up to date. To do this, feedback is essential. You can leave feedback in the regular places (amazon.com for example), on my website (**XXX—TBD** or by emailing me directly at spate@me.com (**XXX—I need a work email address**)).

1.5 Do you do anything for fun?

Credibility is important when writing a book, presenting technical information or representing your company in any formal capacity. Hopefully the information above will show that I have the knowledge and experience to write such a book. Having said that, when reading a review of my last book, one reviewer was complimentary of my book but asked

whether I actually did anything for fun. I was amused and vowed that if I ever wrote another book, I'd actually add information about myself other than what I've done technically. So here goes!



I've been an avid sports fan all my life playing almost every sport you can think of in competitive capacity. This was mostly football/soccer in my early years including a short stint with Southampton Football club youth training program back in the early 80s. But I competed for the high school team in basketball, volleyball, high jump, and cross country. I run every week (several half marathons under my belt), hike the mountains around Tucson, Arizona, do yoga and cycle a lot. When I turned 50, I completed 50 rides of 50+ miles including 2 centuries, 3 metric centuries and a lot of rides between 50-55 miles. At the time I'm writing this we just finished the Tour de Tucson metric century. Not bad for a vegan of 36 years!

My favorite teams are Newcastle United, San Francisco Giants, San Francisco 49ers, Golden State Warriors and the Oregon Ducks (my daughter went to college there).

Oh and I'm a great fan of beer, growing up with Newcastle Brown Ale and British Bitters but having developed more of a taste for IPAs over the last several years. On a cold winter's night (yes, we do get some in Arizona) I love a big, heavy imperial stout (or two).

1.6 Conventions

I try to follow man page sections when commands, libraries and system calls are referenced. So `umount(1)` refers to the `umount` command while `umount(2)` refers to the system call. If `umount` is used without a section reference it refers more to one or the other but generally just during the time when the filesystem is being unmounted.

Internal functions such as those used by the SPFS disk-based filesystem are shown without a manpage reference. An example would be `sp_read_inode()`. The same is true for Linux kernel variables and functions.

There are a few commands that are also used as verbs such as `grep` which will be shown in courier font but without the manpage number.

Processes run in either user-space or kernel-space. Some times these are referred to as user space and kernel space, or in the latter case just *in the kernel*. I refer to them as user-space and kernel-space.

Finally, there are a few peculiarities to point out. It's traditional in the UNIX/Linux space to say *filesystems* instead of *file systems*. UNIX is also written in all caps even though it is not an acronym. I had long arguments with a previous publisher over this.

1.6.1 What Error Checking?

Most of the example programs do little in the way of checking for errors. This is a terrible way to program but since many of the source code examples are displayed throughout the

book, and are there for teaching purposes only, I've omitted most error checking to reduce the amount of code that's displayed. In the chapter on building a disk-based filesystem, there is a lot more error checking since failing to check for errors in the kernel can lead to panics. At times when displaying this code, I've omitted `printk()` statements and error checking for the purpose of describing the flow more easily.

1.6.2 Source Code References

Where relevant we show where in the Linux source code relevant structures and functions can be found. For example, in the section that describes the kernel structures used for mounted filesystems, you will see a callout as follows:



`fs/filesystems.c` contains the `file_systems` variable, routines to register / unregister filesystems and functions for handling `struct filesystem_type` including searching for filesystems during `mount(2)`.

Pathnames are relative to the top of the Linux kernel source tree so `fs/filesystems.c` will be located at `linux-6.1.10/fs/filesystems.c` for example.

1.6.3 Web References

There are multiple times where I've given an internet URL knowing that at some point in the future the server may disappear and the link could become invalid. I've still included the link in short-form making it easier to type. I have also added a section on my website where I will endeavor to make sure that these links are up-to-date. So if a link does become invalid, I'll include a new reference as appropriate.

Each time I give a reference you'll see it like this:



2 <https://tinyurl.com/5bnxjrae>

Thus if you look at the list of ordered references on my website, it will be easy to see the number and be able to click on the link. Throughout the book I use `www.tinyurl.com` to generate the URLs so it's easier for you to type them. Unless of course the URL is small enough to be with. At times, you can just search for relevant terms and you'll end up at the correct site.

1.7 I Versus we vs ...

It used to be very common with technical material to write in the third person. While I do that a lot of most of the time, the main goal of the material to be instructive so will use I/we/our as appropriate depending on the subject matter being discussed and whether it's

more demonstration-based as opposed to a description. So although it may not be the most grammatically correct, I believe that it makes it more readable.

1.8 Who Should Read This Book?

Learning to program in the kernel is not a trivial task and with the huge growth of Linux in terms of its size (over 25 million LOC), learning about the Linux kernel is more of a daunting task today than in the past. There quite is a lot of documentation on-line that didn't exist years ago but it's largely incomplete and most of the Linux kernel internals books are now largely out of date. To some degree this is understandable. Linux has continued to evolve at a fast pace supporting more and more devices from laptops to servers, from mobile phones to supercomputers. There is still a lot of interest in how filesystems work and this is one area that hasn't had a lot of coverage over the years and very little in terms of published material.

This book should appeal to anyone interested in learning more about how filesystems operate but also give the tools to be able to develop simple filesystems in both user-space and in the kernel. That's where the fun is despite being challenging.

Readers should at least be familiar with the C programming language since the Linux kernel and all filesystems are written in C. At the time of writing, support for Rust has been announced but we're still a long way from being able to develop filesystems in Rust. Perhaps look for a blog entry on that or a newer edition of this book.

The book should appeal to undergraduate and post-graduate students who wish to pursue a career in the operating system or system programming field. It's always been the case the kernel-level engineers can work at any layer in the software stack but not necessarily the other way around. I feel that this is largely due to solid debugging skills that kernel engineers must develop often with limited tools.

1.9 What's in the Book?

Readers can jump around as needed of course but the book is written in a style such that each chapter builds on the prior chapter. You need to understand the programming interfaces before you start looking at the kernel. File access in particular uses many different flags resulting in a lot of "if ... then" statements in the kernel depending on which flags are set. The same is true for filesystem management. The basics are explained first before digging deep into how Linux supports them at different layers.

Once more basic concepts have been explained, the kernel source tree layout is described together with information about a set of tools that make it easy for navigating your way through the kernel sources. The location of file and filesystem structures and functions is detailed together with references to the kernel files where the structures and functions are defined.

With knowledge of how the kernel structures for file and filesystem access are linked together, the book then describes in more detail the flow through the kernel functions for supporting file and filesystem related system calls.

To help make sense of how everything works they will be an example disk-based file system which is fully functional and less than 1500 lines of source code making it easy to understand. There are also two user-based filesystems based on the FUSE infrastructure. First is a simple pass-through file system and the second encrypts file contents.

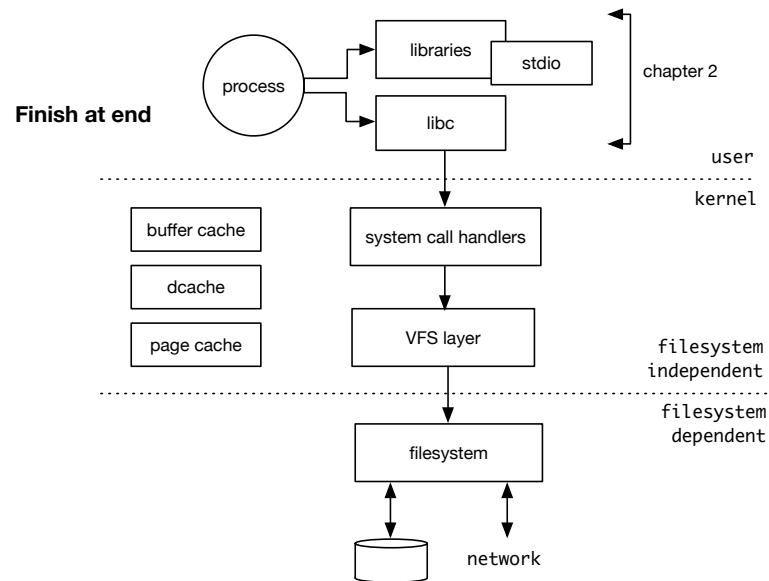


Figure 1.1: Various subsystems in Linux and their location in the book

The book concludes with chapters on debugging techniques, file system performance and filesystem security. Figure 1.1 highlights the parts of a Linux operating system and where each is covered in the book. In more detail, here is the breakdown of each chapter:

- **Chapter 1 – Introduction** – This chapter.
- **Chapter 2 – File and Filesystem Programming** – before understanding how the Linux kernel provides support for filesystems and file access, it's necessary to explore the different commands, system calls and library functions that applications use to access files and filesystems. This chapter provides information on the Linux standards followed, manual pages (manpages), file types, programming interfaces, sparse files and various types of I/O including direct I/O and asynchronous I/O. The standard I/O library is discussed in detail with information about how to browse the standard library source code. There is a section on reading directory entries with a simple implementation of the `ls(1)` command.
- **Chapter 3 – Filesystems** – this chapter covers everything from the different file systems that Linux provides to the operations that can be applied to them such as mounting, unmounting and reporting disk usage. Physical and pseudo filesystems

will be described together with highlights of network and FUSE-based file systems including a demonstration of SSHFS which allows you to view remote files as if they were mounted locally. We will also cover backup and restore, quotas, swap space and how the boot process works in terms of filesystem access.

- **Chapter 4 – A First Look at the Linux Source Code** – with 25 million lines of source code, the Linux kernel is one very large piece of code and can be daunting, especially if you've never spent much or any time looking at it! The chapter describes the layout of the source code and focuses on where the main file and filesystem structures and routines can be found. It describes various tools that can be used to navigate around the source code from the simple such as `vim` and tag stack to the more complete solutions such as `cscope(1)` and Elixir. The chapter then describes the main structures that relate to filesystems and show how the structures are linked together. The main routines will be highlighted together with a few simple examples and browsing through kernel structures using `gdb`.
- **Chapter 5 – Common Linux Types – TBD**
- **Chapter 6 – The VFS Layer** – Following on from the previous chapter which covered the main structures related to filesystems, this chapter goes much deeper in explaining the code paths through the kernel functions to respond to the various filesystem related system calls.
- **Chapter 7 – Filesystem Case Studies – TBD**
- **Chapter 8 – Building a Kernel-based Filesystem** – this is a fun chapter! It presents a fully functional, disk-based file system that contains less than 1500 lines of source code. The foul system was developed on Ubuntu Linux but should be able to run on any version of Linux. It comes with tools for making a file system and a file system debugger. Volatile system is very simple in nature it does support the majority of foul system operations. For fun it also includes an undelete command.
- **Chapter 9 – The FUSE Filesystem Framework** – the FUSE implementation in the Linux kernel allows developers to implement file systems in user space. This chapter provides two examples for FUSE, one which provides a pass-through set of operations reflecting what is under the file system on which it is mounted. The second example builds on the first by providing encryption of files and file names using very simple key management. The implementation of the FUSE architecture will be presented together with information about performance gains that have been achieved over the years including some new performance enhancements using eBPF.
- **Chapter 10 – Filesystem Debugging** – debugging kernel code is certainly a challenge. Application developers have an array of filesystem debug us to choose from including `gdb`. In my career I've spent many a year debugging file system code at assembler level which can be fun but not for the faint hearted. This chapter covers debugging the kernel at source level using GDP but also explores a number of other different debugging techniques including the more recent and very powerful eBPF tools.

- **Chapter 11 – Filesystem Performance** – they have been many books written about UNIX and Linux performance over the years. Although this book won't go into the detail that other books and articles online have covered but this chapter will explore the main performance related aspects of file access and in particular some of the new eBPF performance tools.
- **Chapter 12 – Filesystem Security** – security is a very large topic and there are many tools, standards and practices for securing file system data. This chapter highlights the bank security concerns with foul and foul system access and describes everything from standard UNIX file permissions to encryption key management. Security based sub systems like SELinux and AppArmor are also explored.

1.10 Why is Source Code Missing?

There are many examples throughout the book where I only display part of a kernel structure, part of a function or even strip down some functions to remove error checking and other parts of the function. The goal behind this is to show the most relevant structures and fields and the most prominent paths through functions. I think this is preferable to putting in large amounts of source code and making it harder to see the big picture. Just bear this in mind when browsing the kernel source.

1.11 KGDB Demonstrations

In various sections throughout the book you will see headings such as:

4.2.1 KGDB — Analyzing Per-file Kernel Structures

In these sections, I use `gdb` and the kernel source-level debugger stub `kgdb` to demonstrate the subject matter being displayed. There are also online videos on my YouTube channel showing more detailed versions that I recommend you watch. All demonstrations will help reinforce the material presented.

Generally speaking, when I demonstrate an area of the kernel using `gdb` there will be an accompanying YouTube video



VIDEO 2 – Analyzing VFS structures for open files

If you look at the list of video references on my website, it will be easy to see the number and be able to click on the link. I thought this would be easier than putting URLs in here.

Before looking through the `kgdb` examples, I'd recommend that you read the section 9.1 first. Even if you skip all the steps to set up `kgdb`, look at the command and examples to understand the process that's being followed.

In all examples, I'm going to make use of what `gdb` calls *convenience variables* that reference the structures that are being analyzed. No one can remember long memory addresses and I think this is the easiest way to help with understanding the flow. It's not the quickest method and there will be scripts to help with that. **XXXX-maybe / maybe not**

1.12 Source Code Figures

[h]

There are several different ways that source is described throughout the book. Often the code is just printed as is. Sometimes, only relevant parts of a function are described and there many times during which `gdb` is used to display stack backtraces which help to understand how certain functions are reached.

I find it helpful to use a drawing package to add code fragments and arrows showing how they're all connected. I use OmniGraffle, a drawing package that I've used for many years. Figure 1.2 gives an example of such a figure.

```
do_sys_open()
{
    return do_sys_openat2dfd, filename, &how);

do_sys_openat2dfd, const char __user *filename,
    struct open_how *how)
{
    int fd = build_open_flags(how, &op);
    fd = get_unused_fd_flags(how->flags);
    struct file *f = do_filp_open(dfd, tmp, &op);

    do_filp_open(int dfd, struct filename *pathname,
        const struct open_flags *op)
    {
        set_nameidata(&nd, dfd, pathname, NULL);
        filp = path_openat(&nd, op, flags, ...);
        restore_nameidata();
        return filp;
    }
}
```

Figure 1.2: Example Figure Showing Code Paths through the Linux Kernel

A few of these figures are shown throughout the book but rarely with this much detail. To help your understanding of how the Linux kernel works, I've collected several of these figures together. You can download them from my website – **details TBD**

1.13 Acknowledgements

First and foremost, a big thank you to Eleanor who encouraged me to work on book writing full-time, which is quite a commitment. It's now almost 30 years since the publication of my first book. I've really enjoyed the process in the past but to be able to do it full time rather than cram it into evenings and weekends has made the process much more enjoyable.

Thanks to **xxx and yyy** for reviewing the material.

Thanks to Lukas Brand who sent me an article about filesystem security. While planning the book, I forgot that I'd been in the security industry working on filesystem and

device encryption for the last 15 years!

TBD — do this last

Chapter 2

File and Filesystem Programming

Before diving into the Linux kernel to understand how files and filesystems are accessed and how filesystems operate internally, it's best to look at what's happening in user space by understanding how applications access files. Much of what the kernel does is to satisfy a set of system calls (functions in the kernel that programs access). A process, which is the running instantiation of an executable program, either access this set of system calls directly or uses a set of library functions which in turn use the system call interface.

This chapter explores all the different ways that applications can access files from basic `read(2)` and `write(2)` system calls to the standard I/O library to asynchronous I/O and less-used features such as extended attributes.

2.1 Programming Standards and Portability

Around the time of my last book, we were still living in a world of multiple different UNIX and UNIX-variants of which Linux was one key player. Over the years, there have been several standards bodies created to develop a common set of programming interfaces that each of the operating system vendors diligently worked to adopt to make the life of application developers much easier. I remember working with several dedicated people attending standards meetings of one form or another, implementing these changes and ensuring that the interfaces adhered to the standard by using a common set of test suites.

With Linux and Windows now dominating approximately 90% of the server operating system market, there are only really two platforms now to consider and a relatively small number of Linux distributions covering all Linux installed systems. Of course if we consider smart phones and tablets, that means Android (also Linux based) and iOS. Although these applications are very different from those in the server world, application portability is still important. Wherever possible, you should use standard interfaces. Having a high-level of portability across not just Windows and Linux but Apple OS/X and the various BSD operating systems is still a smart choice.

It's beyond the scope of this book to discuss all the issues around source code portability but I wanted to at least highlight the different standards that are out there to make you aware

before you delve into writing a complex application. Think about where that application may end up in future even if operating system portability isn't a goal initially. You will want to think carefully about the toolchain that you'll use as part of the development process.

The Linux manual pages indicate which standard the functions they describe adhere to. Let's take three examples. Starting with the `write(2)` system call:

```
CONFORMING TO
    SVr4, 4.3BSD, POSIX.1-2001.
```

For the `printf(3)` library function:

```
CONFORMING TO
    fprintf(), printf(), sprintf(), vprintf(), vfprintf(),
    vsprintf(): POSIX.1-2001, POSIX.1-2008, C89, C99.

    snprintf(), vsnprintf(): POSIX.1-2001 & 2008, C99.
```

`dprintf()` and `vdprintf()` functions were originally GNU extensions and were later standardized in POSIX.1-2008.

The `getenv(3)` manual page is a little different. It also describes `secure_getenv()` which it states "*is a GNU extension*":

```
CONFORMING TO
    getenv(): POSIX.1-2001, POSIX.1-2008, C89, C99,
    SVr4, 4.3BSD.

    secure_getenv() is a GNU extension.
```

This could mean that it's not portable across different operating systems so be careful of such extensions.

2.1.1 What are all These Standards?

There have been many attempts at standardization over the last few decades. Here are the prominent standards and how they relate to Linux:

- **POSIX** — The Portable Operating System Interface (POSIX and pronounced *pahz-icks*) is a family of standards specified by IEEE. It defines both system- and user-level "Application Programming Interfaces" (APIs), along with command line shells and utility interfaces, for software compatibility (portability) with variants of UNIX and other operating systems.
- **Single UNIX Specification (SUS)** — a standard that specifies programming interfaces for the C language, a command-line shell, and user commands. Conformance with SUS allowed operating systems vendors to use the name 'UNIX' as part of their brand offering. What started out as *Spec 1170*, a set of 1,170 commands/interfaces, the collection of all such interfaces would eventually become the Single Unix Specification. Very few Linux and BSD vendors/distributions have gone through certification and with the number of UNIX versions declining, we are unlikely to see any more.

- C89 - C2x — ANSI C - standards for the C programming language. As you can see from the manual pages of three functions above, many Linux functions state compliance.
- Linux Standard Base (LSB) — formed by the Linux Foundation, LSB was a joint project comprising multiple Linux distributions with the goal to standardize Linux software system structure, including the *Filesystem Hierarchy Standard* used in the Linux kernel. LSB was based on the POSIX specification, the Single UNIX Specification (SUS), and several other open standards, but it also extended them in some areas.

The first version of POSIX.1 was published in 1988 and the most recent in 2017. Prior to 1997, POSIX was split into several different sub-standards which I'm listing in full below to show how extensive the standard actually was. I recall working on an implementation of threads on an SVR4 UNIX subsystem on the Chorus microkernel while the standard was still being developed back in the early 1990s.

- POSIX.1: Core Services (incorporates Standard ANSI C)
 - Process Creation and Control
 - Signals
 - Floating Point Exceptions
 - Segmentation / Memory Violations
 - Illegal Instructions
 - Bus Errors
 - Timers
 - File and Directory Operations
 - Pipes
 - C Library (Standard C)
 - I/O Port Interface and Control
 - Process Triggers
- POSIX.1b: Real-time extensions
 - Priority Scheduling
 - Real-Time Signals
 - Clocks and Timers
 - Semaphores
 - Message Passing
 - Shared Memory
 - Asynchronous and Synchronous I/O

- Memory Locking Interface
- POSIX.1c: Threads extensions
 - Thread Creation, Control, and Cleanup
 - Thread Scheduling
 - Thread Synchronization
 - Signal Handling
- POSIX.2: Shell and Utilities
 - Command Interpreter
 - Utility Programs

After the first version of POSIX in 1988, there have been several other enhanced versions namely, POSIX.1-2001, POSIX.1-2008 and the final version, POSIX.1-2017. There is a nice FAQ on POSIX by the OpenGroup here if you wish to read further:



3 <https://tinyurl.com/5bnxjrae>

2.1.2 LSB — Linux Standard Base

The Linux Standard Base covered everything from standard libraries, commands and utilities that extended the POSIX standard, file system hierarchy layout, run levels, X Window System extensions, System V UNIX-style initialization scripts and others. Version 1.0 of LSB was released in 2001 and the last version (5.0) in 2015. There were 21 release versions of Linux certified for LSB 4.0 but considerably fewer for 4.1.

LSB has certainly had its critics over the years. Apparently LSB 3.0 was released with a big yawn and also had critics from the glibc team for which it created a lot of additional work. One of the original goals of LSB was for application compatibility allowing 3rd party developers to write-once, run-anywhere including binary and package levels. The goals were sound with Linux leaders not wanting Linux distributions to diverge as the various UNIX vendors had in the past.

Nowadays, relatively few Linux distributions dominate the landscape which reduces the demands for something like LSB and proprietary vendors tend to target specific distributions based on demands from their customer base. But LSB was largely successful in that it brought Linux vendors together at a time that divergence was a real possibility.

At the time of writing, the `lsb_release(1)` command, which did display LSB version details, now just displays information specific to the Linux release. As an example, for Ubuntu:

```
$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.10
Release:        22.10
Codename:       kinetic
```

2.2 Manual Pages

All functions that can be used by an application that are provided by the operating system have a corresponding manual page which is more commonly referred to as a "*manpage*". For programming interfaces, the manpage documents expected parameters, return values and header files that need to be included. There are also manpages for all of the Linux commands, file formats and so on. You can do a keyword search to find manpages that are relevant to the keyword that you supply. For example, let's search for write-related man pages.

Below is a subset of what is shown when searching for write-related functions and commands. There is a command called `write` as indicated by (1) and the `write` system call as indicated by (2).

```
$ man -k write
aio_write (3)          - asynchronous write
fwrite (3)            - binary stream input/output
pwritev (2)           - read or write data into multiple buffers
write (1)              - send a message to another user
write (2)              - write to a file descriptor
...
...
```

What do the numbers in parenthesis mean? The UNIX (and now Linux) manual pages are divided into different sections. The section for each manpage is shown above in parenthesis. To see the list of available sections, you can run "man man". Here are the nine sections for Linux:

```
1  Executable programs or shell commands
2  System calls (functions provided by the kernel)
3  Library calls (functions within program libraries)
4  Special files (usually found in /dev)
5  File formats and conventions, e.g. /etc/passwd
6  Games
7  Miscellaneous (including macro packages and conventions),
   e.g. man(7), groff(7), man-pages(7)
8  System administration commands (usually only for root)
9  Kernel routines [Non standard]
```

If you want to display the man page for the `write()` system call, run "man 2 write". If you want to see the "printf()" manpage run "man printf(3c)". And if you're unsure which section you need, use the "-k" option.

I have on occasion run into issues when trying to view manpages, specifically those related to programming. For example, if you see an error when running "man 2 open" you need to install the development manpages as follows:

```
$ sudo apt install manpages-dev
```

There is a wealth of information in the Linux manpages. If you don't have a running system at hand, just enter "write (2)" in your favorite search engine and you'll be able to view it online.

2.3 System Calls vs Library Functions – an Overview

A software library is just a collection of source code functions and variables that are combined together into a single *library file*. The library can't run by itself. When user programs are built, the source code is first compiled then it is *linked* with one or more libraries to make an executable or binary.

When writing applications you will likely use a number of different library functions. Just take the first program that people usually write in C (or most other languages):

```
#include <stdio.h>

int
main()
{
    printf("Hello world!\n");
}
```

The program can be compiled as follows to get an executable:

```
$ cc hello_world.c -o hello_world
```

or "make hello_world" will achieve the same result. In this program we use a single library function called `printf(3)` that is contained in what is known as the *standard C library* (see `libc7` for more information). To see what libraries our executable program is linked with, we can use the `ldd(1)` command as follows:

```
$ ldd hello_world
linux-vdso.so.1 (0x0000fffffb4e83000)
libc.so.6 =>
/lib/aarch64-linux-gnu/libc.so.6 (0x0000fffffb4c70000)
/lib/ld-linux-aarch64.so.1 (0x0000fffffb4e46000)
```

If library functions are just functions packaged together, what is the difference between a library function and a system call? When writing applications you use library functions which may perform the required function by themselves or they may call into the Linux kernel for services. For a simple program such as the one below, you may unaware of difference between one function type and another (error checking excluded for simplicity):

```
#include <fcntl.h>
#include <stdio.h>
```

```
#include <unistd.h>
#include <string.h>

int
main()
{
    char buf[64];
    int fd;

    fd = open("msg", O_CREAT | O_WRONLY);
    sprintf(buf, "hello world -> fd = %d\n", fd);
    write(fd, buf, strlen(buf));
}
```

There are four different functions called two of which are system calls (`open` and `write`) and two which are not. As programmers gain experience they will generally know which functions are system calls and which are not without having to check the manpages. Later sections in the book will describe system call handling for file and filesystem-related functions in detail.

2.4 Error Handling with `errno`

As the `errno(3)` manpage says:

The <errno.h> header file defines the integer variable `errno`, which is set by system calls and some library functions in the event of an error to indicate what went wrong.

If you're like me, trying to remember error codes and which `errno` value they correspond to, good luck! However, if you install the `moreutils` package, you have a handy utility at your disposal:

```
$ sudo apt install moreutils
$ errno 9
EBADF 9 Bad file descriptor
$ errno EBADF
EBADF 9 Bad file descriptor
```

I also have a simple script that dumps all of them out:

```
echo "number      hex symbol      description"
1  0x01      EPERM   Operation not permitted
2  0x02      ENOENT  No such file or directory
...
132 0x84     ERFKILL Operation not possible due to RF-kill
133 0x85     EHWPOISON Memory page has hardware error "
```

Note that the actual error values are not stored in `/usr/include/errno.h` but can be found in `/usr/include/asm-generic/errno-base.h`.

2.5 File Types and Mode

Everything stored in a Linux filesystem is a file regardless of the file type. Here are the different types of files that can be supported by Linux although note that not all filesystems support all file types. For example, the BFS filesystem is a specialized SVR4 UNIX filesystem for supporting boot-related files including the kernel. It is a flat filesystem so it doesn't support creation of directories (although it does have a root directory):

1. **Regular files** — a file that contains data that applications create and interpret. The filesystem or the kernel has no knowledge about the content of regular files.
2. **Directory files** — also called *folders* (although primarily in desktop operating systems such as Windows and OS/X), a directory can contain files and other directories.
3. **Symbolic links** — a file that contains a pathname that references another file. It could be a single name or multiple names separated by "/". A symbolic link can point anywhere in the Linux filesystem namespace. It does not have to point to a file within the same filesystem.
4. **Hard links** — a name within a directory that references another file's contents and meta-data directly. If file 'a' is a hard link to file 'b' and if you delete file 'b', you can still access the file through 'a'. For a symbolic link, If file 'a' is a symbolic link to file 'b' and if you delete file 'b', you cannot still access the file through 'a'.
5. **Device nodes** — a special file that contains both a *major* number which is used to reference a specific device driver and a *minor* number that it interpreted by the device driver. For example, on my Ubuntu VM there are two device nodes for a specific SCSI disk:

```
brw-rw---- 1 root disk 8, 1 Dec 15 16:41 /dev/sda1  
brw-rw---- 1 root disk 8, 2 Dec 15 16:41 /dev/sda2
```

The major number is 8 which references the SCSI device driver and the minor numbers 1 and 2 are used by the device driver to refer to specific partitions. Note that we can have block and character special files. A Character Device is a device for which the device driver reads and writes single characters. Examples include keyboards, serial and parallel ports. A Block Device is a device for which the device driver reads/writes blocks of data. An example would be hard disks. Note that filesystems can only reside on block devices.

6. **Sockets** — a special file used for inter-process communication between two processes. In addition to sending data, processes can send file descriptors across a UNIX domain socket using the `sendmsg(2)` and `recvmsg(2)` system calls.
7. **FIFOs** — also called a *named pipe*, a FIFO is a uni-directional channel through which one process can send data and another process can receive it. Pipes/FIFOs are created by the `mknod(1)` command.

You can find out the type of a file by using the `stat(1)` command or one of 4 different system calls namely `stat`, `lstat`, `fstat` and `fstatat` all which are described in the `stat(2)` manpage. The `st_mode` field returned by these functions contains both the file type and the file mode. To understand more about the mode, the `inode(7)` manpage explains how each file can be identified. We'll also cover system calls in more detail soon.

<code>S_IFMT</code>	0170000	bit mask for the file type bit field
<code>S_IFSOCK</code>	0140000	socket
<code>S_IFLNK</code>	0120000	symbolic link
<code>S_IFREG</code>	0100000	regular file
<code>S_IFBLK</code>	0060000	block device
<code>S_IFDIR</code>	0040000	directory
<code>S_IFCHR</code>	0020000	character device
<code>S_IFIFO</code>	0010000	FIFO

Below is a short program that calls `stat(2)` on a regular file. We print out the `st_mode` field and with the mask so we can just determine the file type.

```
struct stat buf;
int error = stat("lorem-ipsum", &buf);

printf("mode = %o (%o)\n", buf.st_mode, buf.st_mode & S_IFMT);
if (S_ISREG(buf.st_mode)) {
    printf("The file is a regular file\n");
}
if ((buf.st_mode & S_IFMT) == S_IFREG) {
    printf("... and again!\n");
}
```

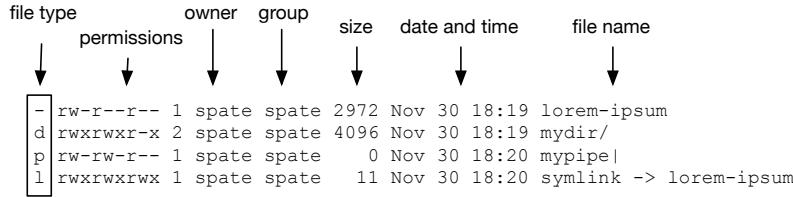
The program then use two different methods to check to see if this is a regular file.

```
$ ./st
mode = 100644 (100000)
The file is a regular file
... and again!
```

The remainder of the `st_mode` field is for storing the file's permissions. In this case we have 644 which we can see when running `ls(1)`. Figure 2.1 also shows several components of a file displayed by running "`ls -l`". Permissions show access rights by user, group and others.

2.6 File Descriptors

Every time a file is opened or a file is created, a file descriptor is returned that can be used to reference the open file through uses of one of several different operations such as `read(2)`, `write(2)` and `lseek(2)`. File descriptors are allocated on a per-process basis and are always non-negative. The first 3 file descriptors are reserved as shown in table 2.1.

Figure 2.1: The Components of a File as Shown by Running `ls -l`

Name	Value	<code><stdio.h></code> file stream	<code><unistd.h></code> constant
stdin	0	standard input	STDIN_FILENO
stdout	1	standard output	STDOUT_FILENO
stderr	2	standard error	STDERR_FILENO

Table 2.1: Standard File Descriptors

In the example below, you can see that the following program will receive the next available file descriptor which is 3.

```

$ cat fd.c
#include <stdio.h>
#include <fcntl.h>

int
main()
{
    int fd = open("lorem-ipsum", O_RDONLY);
    printf("fd = %d\n", fd);
}
$ make fd
$ ./fd
fd = 3

```

Jumping ahead a little to demonstrate how file descriptors are used in the kernel, the `read(2)` system call is handled by the kernel function `ksys_read()` of which I've included a fragment of the source code here:

```

ssize_t ksys_read(unsigned int fd, char __user *buf,
                  size_t count)
{
    struct fd f = fdget_pos(fd);
    loff_t pos, *ppos = file_ppos(f.file);
    ret = vfs_read(f.file, buf, count, ppos);
}

```

As you can see, the arguments are very similar to the `read(2)` system call. Inside this function the kernel uses the file descriptor to get an `fd` struct. From there, it gets the

position from the last read (or seek) and calls `vfs_read()` to continue read processing. We'll explore the kernel side of reading later in section XXX.

2.6.1 How Many Open Files?

Most applications only open a small number of files and the Linux kernel is set up to support this by default for each process. The `getdtablesize(2)` manpage has conflicting information. It states that it returns the maximum number of files a process can have open. But then it lists the return value as the *current limit*. The following program retrieves this value and also checks to see what is returned from calling `getrlimit()` which returns both the current limit (also called the *soft limit*) and the *hard limit* as returned by calling `getrlimit(2)`:

```
#include <unistd.h>
#include <stdio.h>
#include <sys/resource.h>

int
main()
{
    struct rlimit rlim;
    int nentries;

    nentries = getdtablesize();
    getrlimit(RLIMIT_NOFILE, &rlim);
    printf("Current file table entries = %d\n", nentries);
    printf("rlimit current / maximum = %d / %d\n",
           (int)rlim.rlim_cur, (int)rlim.rlim_max);
}
```

Here is what we get when running the program. 1,048,576 is certainly larger than most applications will need.

```
Current file table entries = 1024
rlimit current / maximum = 1024 / 1048576
```

You can also get the current limit by calling the shell built-in command "ulimit -S" and calling "ulimit -H" to get the hard limit.

2.6.2 Closing Files

There isn't a great deal to say about closing files. The system call, as described in `close(2)` is as follows:

```
#include <unistd.h>

int close(int fd);
```

If there are any record locks held on the file that `fd` refers to and are owned by the calling process, they are removed.

If `fd` is the last file descriptor referring to the underlying open file any resources that associated with the open file are freed. If the file descriptor is the last reference to a file which has been removed by a prior call to `unlink(2)`, the file will then be deleted. The latter point is an interesting one for filesystem developers as the file could remain in this state for an extended period of time and there are certainly implications if the system is restarted in between these two operations.

2.6.3 Duplicating File Descriptors

The `dup(2)` system call can be called to allocate a new file descriptor which will reference the same open file as the descriptor `oldfd`. The returned value is guaranteed to be the lowest-numbered file descriptor that is unused in the calling process.

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Since both old and new file descriptors reference the same file, they can then be used interchangeably. They also share file offset and file status flags. If the file offset is changed by using the `lseek(2)` system call on one of the file descriptors, the offset will also changed for the other file descriptor.

The `dup2(2)` system call is similar but instead of getting the lowest available file descriptor, the file descriptor specified by `newfd` is used. If `newfd` is currently in use, it is closed first. This function is the equivalent of calling `fcntl(2)` and passing the `F_DUPFD` command.

The `dup3(2)` system call was added to Linux version 2.6.27. It performs the same function as `dup2(2)` with one exception. The caller can force the close-on-exec flag to be set for the new file descriptor by setting `flags` to `O_CLOEXEC`. In this case the file descriptor will be closed following a call to `execve(2)`. The `open(2)` manpage describes how this flag is useful in multithreaded applications.

```
#include <fcntl.h>
#include <unistd.h>

int dup3(int oldfd, int newfd, int flags);
```

The most common use case for duplicating file descriptors is to connect processes via a pipe which is a common operation we all use when running one command in the shell and connecting the output to the input of the next command. In the following example:

```
$ find . -name *.c | xargs grep myfunc
```

the output of the `find` command is connected to the input of the `xargs` command.

An Example of Redirection

This section is really going beyond the scope of this book but redirection is such a fundamental part of the UNIX/Linux shell, that I thought it was worth taking a slight diversion

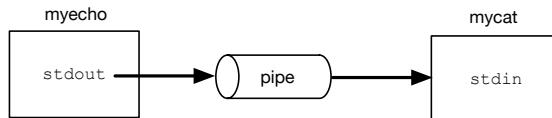


Figure 2.2: Tow processes connected via a pipe

for readers who may not be familiar with how pipes are used to achieve one of UNIX's most fundamental and wonderful capabilities.

Take a look at the following commands connected together where the standard output of echo is connected to the standard input of cat:

```
$ echo "Hello world" | cat
Hello world
```

Below is a program that forks/execs two other programs (myecho and mycat) and uses a combination of dup(2) and pipe(2) to connect the two processes together through stdin and stdout in a simple but similar way to how the shell operates. When the shell program is run we'll see our message displayed:

```
$ ./shell
Hello world!
```

To see this visually, refer to figure 2.2.

Below is the shell program. As per usual, we've left out error checking for the purpose of keeping the number of lines as small as possible.

```

1 #include <unistd.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 int main(int argc, char **argv) {
6     int pipefd[2];
7     pid_t child;
8
9     pipe(pipefd);
10    child = fork();
11
12    if (child == 0) {
13        /* child process */
14        close(STDOUT_FILENO);
15        dup(pipefd[1]);
16        execl("myecho", "myecho", "Hello world!\n",
17              (char *)NULL);
18    } else {
19        /* parent process */
20        close(STDIN_FILENO);
21        dup(pipefd[0]);
  
```

```
22         execlp("mycat", "mycat", (char *)NULL);
23     }
24 }
```

The following bullet points describe what the various parts of the program are doing (line numbers are on the left):

- line 9 – A pipe is created where `pipe[0]` is the file descriptor through which we can read data and `pipe[1]` is the file descriptor through which we can write data.
- line 10 – The program is forked.
- line 14 – `stdout` is closed for the `myecho` process.
- line 15 – `dup(2)` is called to duplicate the file descriptor in `pipefd[1]` which is the *write side* of the pipe. Recall that `dup(2)` is guaranteed to return the lowest-numbered file descriptor that unused in the calling process. Since we have just closed `stdout` which has a file descriptor of 1, `dup(2)` file descriptor 1 will be used which is `stdout`.
- line 16 – At this point we exec the `myecho` program and pass "Hello world!" as an argument.
- line 20 – `stdin` for the `mycat` process is closed.
- line 21 – Similar to line 15, we replace `stdin` for `mycat` with the *read side* of the pipe.
- line 22 – Finish with a call to exec the `mycat` program.

The actual shell is of course considerably more complicated than this program but in essence, this is what it is doing.

Here is the `myecho` program. It's very simple and just writes the argument passed to `stdout`.

```
#include <stdio.h>

int
main(int argc, char *argv[])
{
    fprintf(stdout, "%s", argv[1]);
}
```

Here is the `mycat` program. It reads from `stdin` and writes the output (to `stdout`).

```
#include <stdio.h>
#include <string.h>

#define BUFSZ 64

int
```

```
main()
{
    char buf[BUFSZ];

    fgets(buf, BUFSZ, stdin);
    printf("%s", buf);
}
```

Note that if you run these programs, make sure that PATH is set to include the current directory so that the myecho and mycat programs can be found.

If you remove the comments around fprintf you will see that for both processes, stderr is still connected to the terminal on which the programs are run from:

```
$ ./shell
I'm mycat
I'm myecho
Hello world!
```

2.6.4 Named Pipes

A file type that can be supported by different filesystems is a named pipe.

XXX—need to find out how Linux handles them

There are some VFS interfaces/arguments for this. See section 6.21 for the kernel side of things.

2.7 Opening Files / Creating Files

This section covers file creation together with the different file open operations since creation can also be performed using the open(2) system call. The interfaces themselves are very simple but explanation is complicated by the large number of *flags* that can be passed which determine how the file will be opened and accessed after open succeeds.

Here are the functions that can be used for open/create. They are described in the open2) manpage with the exception of openat2(2) which has its own manpage.

```
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

int creat(const char *pathname, mode_t mode);

int openat(int dirfd, const char *pathname, int flags);
int openat(int dirfd, const char *pathname, int flags,
           mode_t mode);

int openat2(int dirfd, const char *pathname,
            const struct open_how *how, size_t size);
```

In the first instance of open, pathname references the file to be opened and can be relative or absolute, for example:

/home/spate/src/spfs/kern/sp_dir.c — an absolute pathname
sp_dir.c — a relative pathname

Absolute pathnames always start with '/' and relative pathnames start from the current working directory which can be obtained with a call to `getcwd(3)`.

The flags argument must be one of `O_RDONLY`, `O_WRONLY`, or `O_RDWR` for reading, writing or reading and writing. If a file is being created, `O_CREAT` should be OR'd with flags.

There is a separate `creat(2)` system call which is equivalent to calling `open(2)` with flags equal to `O_CREAT | O_WRONLY | O_TRUNC`.

2.7.1 Open Flags

There are 19 different flags that can be passed to the `open(2)` system call. This section will spent quite a bit of time explaining each of the flags since the kernel has many conditional paths based on which flags are set. Note that are a few flags that are not related to file access and have been omitted from this list.

- `O_APPEND` – the file is opened in append mode so the file offset will be positioned at the end of the file.
- `O_CLOEXEC` – enable the close-on-exec flag for the new file descriptor otherwise the file descriptor will remain open in the child process following an `execve(2)` system call.
- `O_CREAT` – if the file does not exist, it is created as a regular file.
- `O_DIRECT` – used by applications that wish to do their own caching (databases for example). I/O will move directly between the user address space buffers and disk.
- `O_DIRECTORY` – if pathname is not a directory, cause the open to fail. This flag was added to avoid denial-of-service problems.
- `O_DSYNC` – if this flag is set, when a `write(2)` system call (or similar calls) return, the data will have been transferred to the underlying hardware, along with any file metadata that would be required to retrieve that data at a later date.
- `O_EXCL` – this flag should be used in conjunction with `O_CREAT`. If the file already exists, the open will fail. If the file does not exist, an attempt will be made to create it and open it. This allows a process to create a file for writing and sure that no other process will write to. It could used for example, when creating a files in `/tmp`, where several other processes are creating files with similar, possibly identical, names. The `mkstemp(3)` function calls open with `O_CREAT` and `O_EXCL`.

- `O_LARGEFILE` – allows files whose sizes cannot be represented in an `off_t` (but can be represented in an `off64_t`) to be opened. The `_LARGEFILE64_SOURCE` macro must be defined (before including any header files) in order to obtain this definition.
- `O_NOATIME` – do not update the file's last access time (`st_atime` in the inode) when the file is read using `read(2)` and friends.
- `O_NOFOLLOW` – if the basename of `pathname` is a symbolic link, the open will fail with the error `ELOOP`.
- `O_NONBLOCK` – when this flag is added, any operation including the open will not cause the calling process to wait. This flag is particularly useful when opening pipes.
- `O_NDELAY` – same as `O_NONBLOCK`.
- `O_PATH` – this is an interesting flag. The goal is to obtain a file descriptor that can be used to indicate a *location in the filesystem tree* and to perform operations that act purely at the file descriptor level. The file itself is not opened, and other file operations (e.g., `read(2)`, `write(2)`, `fchmod(2)`, `fchown(2)`, `fgetxattr(2)`, `ioctl(2)`, `mmap(2)`) fail with the error `EBADF`. Another use of this flag is in conjunction with the the "`*at`" calls.
- `O_SYNC` – this flag ensures that by the time a `write(2)` (or similar) system call returns, the data and associated file metadata will have been transferred to the underlying hardware.
- `O_TMPFILE` – create an unnamed temporary regular file. The `pathname` argument specifies a directory. An unnamed inode will be created in the specified directory's filesystem. Anything written to the resultin file will be lost when the last file descriptor is closed, unless the file is given a name.
- `O_TRUNC` – if the file already exists and it is a regular file and the access mode allows writing (`O_RDWR` or `O_WRONLY`) the file will be truncated to length 0.

There is a lot of detail behind several of these flags. For example, I suggest exploring the use cases for `O_PATH`. As mentioned above, the kernel is littered with checks for these flags. Just taking `O_TMPFILE` as one example, here are the places in the kernel's `fs` directory that reference this flag:

```
$ grep O_TMPFILE *.c
namei.c:           if (unlikely(file->f_flags & __O_TMPFILE)) {
open.c:#define WILL_CREATE(flags) (flags & (__O_TMPFILE | ...
open.c: * O_TMPFILE on old kernels, O_TMPFILE is ...
open.c: if (flags & __O_TMPFILE) {
open.c:           if ((flags & O_TMPFILE_MASK) != O_TMPFILE)
open.c:           if (flags & (O_TRUNC | O_CREAT | O_TMPFILE))
```

Subsequent chapters will cover the kernel side of these flags.

2.8 Reading and Writing Files

Reading from and writing to regular files was originally implemented by the `read(2)` and `write(2)` systems calls and enhanced by the standard I/O library which will be discussed in section 2.14. Over time there have been several other library and system call interfaces that support file I/O. To start, let's look at the original system calls:

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

To read from a file, specify a valid file descriptor (`fd`), a buffer in memory (`buf`) and the number of bytes to be read. Reading will start from the current offset and the number of bytes read will be returned. Writing is the opposite. Starting from the current offset, an attempt will be made to write `count` bytes from the address referenced by `buf` to the file referenced by `fd`.

When the file is opened, the offset is set to 0 and after each read or write operation, the offset will be advanced by the amount of data read or written.

Applications don't always wish to read and write sequentially so this is where the `lseek(2)` system call comes into play:

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

The `offset` argument specifies the offset within the file to move to and `whence` can be one of the following:

- `SEEK_SET` – the file offset is set to `offset` bytes.
- `SEEK_CUR` – the file offset is set to its current file location plus `offset` bytes.
- `SEEK_END` – the file offset is set to the size of the file plus `offset` bytes.
- `SEEK_DATA` – the file offset is set to the next location in the file that is greater than or equal to the specified `offset` containing data.
- `SEEK_HOLE` – the file offset is set to the next hole in the file that is greater than or equal to `offset`. If `offset` points into the middle of a hole, the file offset is set to `offset`. If there is no hole past `offset`, the file offset is adjusted to the end of the file.

The last two options were added in Linux 3.1 to allow applications to handle holes in sparse files. One particular use case is file backup tools, which can save space when creating backups by preserving holes.

For applications that move around with a file frequently, both `read/write` and `lseek` calls can be combined together in a single call using one of the following two system calls:

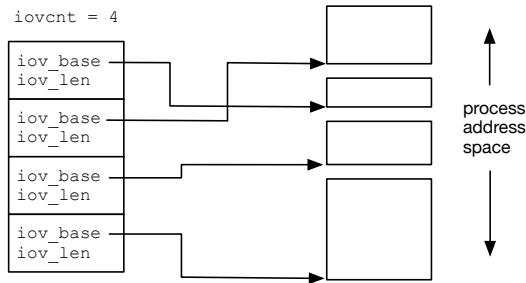


Figure 2.3: Issues multiple I/Os in a single system call using `readv(2)` / `writev(2)`

```
#include <unistd.h>

ssize_t pread(int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite(int fd, const void *buf, size_t count,
               off_t offset);
```

Both functions differ from `read(2)` and `write(2)` in that although reads/writes occur from the specified offset, the current file offset is not changed. These two system calls are particularly useful for multi-threaded applications where different threads may be operating on the file at the same time. It would not be very useful for one thread to call `lseek(2)` followed by `write(2)` only to find that another thread had made a call to `lseek(2)` in the meantime and changed the offset!

2.9 Vectored Reads and Writes

There are times when applications know ahead of time that they will need to perform multiple I/Os to/from different memory addresses and the file, and likely at different offsets. This type of I/O is called *vectored reads/writes* and also known as *scatter/gather I/O*. There are several functions that can be used all of which utilize the `iovec` structure, defined in `<sys/uio.h>`:

```
struct iovec {
    void *iov_base;      /* Starting address */
    size_t iov_len;     /* Number of bytes to transfer */
};
```

Figure 2.3 highlights a case where we have four different I/Os that need to be performed. Each I/O is of a different length and the base address of the data inside the process's address space is at a different location.

Here are the functions that are available to perform vectored I/O. All are described by the `readv(2)` manpage. The first two functions read / write from the current offset within the file. In figure 2.3 there are four I/Os to different locations in memory but the data must be consecutive on disk.

```
#include <sys/uio.h>

ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);

ssize_t preadv(int fd, const struct iovec *iov, int iovcnt,
               off_t offset);
ssize_t pwritev(int fd, const struct iovec *iov, int iovcnt,
                off_t offset);

ssize_t preadv2(int fd, const struct iovec *iov, int iovcnt,
                off_t offset, int flags);
ssize_t pwritev2(int fd, const struct iovec *iov, int iovcnt,
                off_t offset, int flags);
```

The `preadv()` and `pwritev()` functions combine reads/lseeks so that I/Os can take place from different offsets within the file as well as different areas in memory.

There are two main advantages of vectored I/O. The first is the reduction in the number of system calls that the application needs to make. Secondly if an application wishes to read data from a file into different locations in memory, using the `read(2)` system call, the data would need to be read into a contiguous buffer in memory and then copied to the desired locations.

Here is an example of using both the `readv()` and `preadv()` functions in the same program. We start with a file of data that we're going to read which has ten 0s followed by ten 1s and so on:

```
$ cat vector-file
0000000001111111112222222223333333444444444555555556666
6666677777777888888889999999999
```

The program source code is as follows:

```
1 #include <fcntl.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <sys/uio.h>
5
6 int
7 main(int argc, char *argv[])
8 {
9     int   fd;
10    char buf2[32];
11    char buf3[32];
12    char buf1[32];
13    struct iovec bufs[] = {
14        { .iov_base = (void *)buf1, .iov_len = 10 },
15        { .iov_base = (void *)buf2, .iov_len = 10 },
16        { .iov_base = (void *)buf3, .iov_len = 10 },
17    };
18}
```

```

19     fd = open("vector-file", O_RDONLY);
20     lseek(fd, 10, SEEK_SET);
21     readv(fd, bufs, sizeof(bufs) / sizeof(bufs[0]));
22     printf("buf1 = %.10s\n", buf1);
23     printf("buf2 = %.10s\n", buf2);
24     printf("buf3 = %.10s\n", buf3);
25
26     preadv(fd, bufs, sizeof(bufs) / sizeof(bufs[0]), 20);
27     printf("buf1 = %.10s\n", buf1);
28     printf("buf2 = %.10s\n", buf2);
29     printf("buf3 = %.10s\n", buf3);
30 }

```

Here are the main points to discuss for this program:

- line 10 - 12 – we’re allocating the buffers to read in to from the stack. But note that they’d been switched around so that they’re not contiguous. Any order would work just the same.
- line 13 - 17 – here are the definition of the `iovec` structures to be used. There will be 3 reads of 10 bytes each.
- line 19 – the file is opened read-only.
- line 20 – a call is made to `lseek(2)` to seek to byte 10 so the 0s will be skipped.
- line 21 – the I/O call is issued.
- line 22 - 24 – the first ten characters of each buffer is printed.
- line 26 – this time the `preadv(2)` system call is called. Prior to this call the file offset was changed to 40 following the call to `readv(2)`. This time, we want to read starting at offset 20.
- line 27 - 29 – print out ten characters from each buffer again.

Here is the output when running the program:

```

buf1 = 1111111111
buf2 = 2222222222
buf3 = 3333333333
buf1 = 2222222222
buf2 = 3333333333
buf3 = 4444444444

```

Vectored I/O became popular with databases where large amount of I/O were needed but to different locations in memory where the database cached data. The basic interfaces were enhanced to allow different offsets in the file for each I/O. File I/O was later enhanced to avoid kernel caching (direct I/O) followed by backgrounding multiple I/Os, all to improve performance. Direct I/O and Async I/O will be described later in the chapter.

2.10 File and Record Locking

In a single application, a file can be accessed by multiple processes or by multiple threads within a single process. To handle synchronization between threads in a single process there are various locking primitives that can be used by threads within the process. If multiple processes are accessing a file simultaneously, there are a different set of file and record locking interfaces that can be used to coordinate access. Historically, file locking has been a bit of a mess.

Some of these interfaces predate per-process multi-threading support. In particular, *advisory* and *mandatory* file locking have been around for many years and come with a set of issues that developers should be aware of.

2.10.1 Advisory File Locking

The first type of file locking introduced in UNIX and available in Linux, was *advisory locking* also called record locking since byte ranges within a file can be locked but processes must query to determine what portions of the file are locked. Thus the "advisory" part of the implementation.

To make things confusing, there are three functions that can be used for advisory locking namely `fcntl(2)`, `lockf(3)` and `flock(2)`. On Linux, `lockf(3)` is simply implemented on top of `fcntl(2)` and any locks implemented through use of `flock(2)` are ignored.

To get started, here is the interface for `fcntl(2)`:

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* arg */ );
```

The `cmd` argument will be `F_SET_LK`, `F_SETLKW` or `F_GETLK` dependent on whether the caller wants to acquire, release or test for the existence of a record lock. The third argument passed is a pointer to an `flock` structure which specifies what part of the file should be locked:

```
struct flock {
    ...
    short l_type;      /* Lock type: F_RDLCK, F_WRLCK, F_UNLCK */
    short l_whence;    /* How to interpret l_start. One of:
                        SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;     /* Starting offset for lock */
    off_t l_len;       /* Number of bytes to lock */
    pid_t l_pid;       /* PID of process blocking our lock
                        (set by F_GETLK and F_OFD_GETLK) */
    ...
};
```

The `l_type` field specifies whether the caller wants to take a read lock, a write lock or just unlock a previously locked set of bytes. The `l_whence`, `l_start` and `l_len` fields of this structure specify the range of bytes that the caller wishes to lock.

There is some deadlock detection built into the implementation such that if two processes attempt to access a region of the file that is locked by the other process, an EDEADLK error will be returned and thus the caller should release some of its locks allowing others to proceed.

The implementation of this style of locking has other issues which will be described in section 2.10.5.

2.10.2 An Example of Advisory Locking

This section shows two programs that work together to show how advisory locking can work between two processes. The first program (`alock`) takes a write lock on the file "hello" and over the whole file. It waits until it receives a signal (SIGUSR1 before releasing the lock).

The second program (`lcat` calls the `system(3)` library function to `cat` the contents of the file. Before doing so, it checks three times to see if the file is locked. If it's still locked after the third check, it sends SIGUSR1 to the process that holds the lock. It can then display the file.

Here is the `alock.c` program:

```
1 #include <unistd.h>
2 #include <fcntl.h>
3 #include <signal.h>
4 #include <stdio.h>
5
6 void
7 mysig_hdlr(int signo)
8 {
9     printf("alock - got signal\n");
10    return;
11 }
12
13 int
14 main()
15 {
16     struct flock      lock;
17     int              fd;
18     struct sigaction action;
19
20     action.sa_handler = mysig_hdlr;
21     sigemptyset(&action.sa_mask);
22     action.sa_flags = 0;
23     sigaction(SIGUSR1, &action, NULL);
24
25     fd = open("hello", O_RDWR);
26
27     lock.l_type = F_WRLCK;
28     lock.l_whence = SEEK_SET;
29     lock.l_start = 0;
```

```
30     lock.l_len = 0;
31     lock.l_pid = getpid();
32     fcntl(fd, F_SETLK, &lock);
33
34     printf("alock - file now locked\n");
35     pause();
36     lock.l_type = F_UNLCK;
37     fcntl(fd, F_SETLK, &lock);
38     printf("alock - file now unlocked\n");
39 }
```

Some comments on the code:

- lines 20 - 23 – set up a signal handler (`mysig_hdlr`) for `SIGUSR1`.
- lines 27 - 32 – set a write lock on the file for the whole file by setting `l_start` to 0 and `l_len` to 0.
- line 35 – pause waiting for `SIGUSR1`.
- lines 36 - 37 – release the lock (keeping the original fields that specify the start/length which is the whole file).

Here is the source code for the `lcat.c` program:

```
1 #include <signal.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <fcntl.h>
5 #include <stdio.h>
6
7 pid_t
8 file_is_locked(int fd)
9 {
10     struct flock    lock;
11
12     lock.l_type = F_RDLCK;
13     lock.l_whence = SEEK_SET;
14     lock.l_start = 0;
15     lock.l_len = 0;
16     lock.l_pid = 0;
17
18     fcntl(fd, F_GETLK, &lock);
19     return (lock.l_type == F_UNLCK) ? 0 : lock.l_pid;
20 }
21
22 int main()
23 {
24     pid_t  pid;
25     int    i, fd;
26
```

```

27     fd = open("hello", O_RDONLY);
28
29     for (i = 0 ; i < 3 ; i++) {
30         if ((pid = file_is_locked(fd)) != 0) {
31             printf("lcat - waiting for lock\n");
32             sleep(1);
33         } else {
34             system("cat hello"); /* shouldn't get here */
35         }
36     }
37     printf("pid = %d\n", pid);
38     kill(pid, SIGUSR1);
39     sleep(1);
40     if ((pid = file_is_locked(fd)) != 0) {
41         system("cat hello");
42     } else {
43         /* shouldn't get here */
44         printf("lcat - lock still not released\n");
45     }
46 }
```

Some comments on the code:

- lines 10 - 19 – the `file_is_locked()` function checks to see if this process can take a read lock over the whole file. If it can it returns 0 and if it can't it returns the PID of the process that holds a lock that covers this range.
- lines 29 - 34 – this program loops three times checking to see if the read lock can be taken. There is nothing special about this loop. The idea is just that if the lock can't be taken, the program could be doing something else. After the 3rd attempt, it drops out with the PID of the process that holds the lock.
- line 38 - 39 – send SIGUSR1 to the process that holds the lock. The `alock` process will catch the signal and release the lock. It then waits a second to give `alock` time to process the signal and unlock the file.
- lines 40 - 41 – another check is made to see if `lcat` can get the read lock. If it can, it will `cat` the contents of the file.

There are two places in the program where it states that the code path should not get there. This is certainly the case if only these two processes are accessing the file. If there are others, this could change. On line 43, this path could be reached if the "1" second sleep isn't enough to allow `alock` to unlock the file.

Here are the two programs running in separate terminals but with the output merged so you can see the order of events:

```

$ ./alock
alock - file now locked
|
| $ ./lcat
| lcat - waiting for lock
```

```
| lcat - waiting for lock
| lcat - waiting for lock
alock - got signal      |
alock - file now unlocked |
| pid = 9671
| Hello world!
```

Since this is advisory locking, applications must work in coordination with each other and non-cooperating applications cannot interfere otherwise assumptions will be broken. It would be best to segregate these applications and files so that no other users/processes can get access to them.

2.10.3 Viewing Existing Locks

Existing locks currently in use can be viewed in two ways. The first is to use the `lslocks (1)` command:

```
$ lslocks -b
COMMAND      PID  TYPE   SIZE MODE   M START END PATH
multipathd  441  POSIX    WRITE 0      0     0 /run/snapd/ns...
cron        838  FLOCK    WRITE 0      0     0 /run/snapd/ns...
snapd       653  FLOCK    WRITE 0      0     0 /...
alock      10397  POSIX   13    WRITE 0      0     0 /home/.../hello
```

In the last line you can see the `alock` process that holds the WRITE lock for 13 bytes which is the length of the file. The type of lock depends on the call made. POSIX locks are created using the `lockf(2)` or `fcntl(2)` system calls and FLOCK locks are created using the `flock(2)` system call.

The second method is to view the locks through `/proc` as follows:

```
$ cat /proc/locks
1: POSIX ADVISORY WRITE 10397 fd:00:396261 0 EOF
2: FLOCK ADVISORY WRITE 653 fd:00:528747 0 EOF
3: FLOCK ADVISORY WRITE 838 00:1b:1248 0 EOF
4: POSIX ADVISORY WRITE 441 00:1b:506 0 EOF
$ ps -ef | grep 10397
spate      10397  4862  0 23:35 pts/1    00:00:00 ./alock
```

The obvious difference here is that there the command name is not shown although the process ID is displayed and can be easily used to find the process.

2.10.4 Mandatory File Locking

Since not all processes check for file locks, it was deemed that advisory locking needed a fix. The solution was the introduction of *mandatory file locking* in SVR3 UNIX whereby access to a file would be denied if it was locked regardless of whether a process checks for locks or not. The intention of the scheme was to have as little impact as possible on existing applications. Of course, applications needed to be aware about how to handle the denial case. And to make matters worse, system calls such as `unlink(2)` were able to succeed.

To enable mandatory file locking in Linux there are two steps that are needed:

1. The underlying file system must be mounted with the `mand` option (for example "`mount -o mand -t spfs /dev/sdb1 /mnt`").
2. For the file to be locked, the set-group-ID bit must be turned on and the group-execute bit must be turned off for all files that need to be locked (for example "`chmod g+s, g-x <file>`"). This is a combination that otherwise makes no sense. You can't get an uglier solution than this!

After a process places a mandatory lock on a file, no other process can access the file for reading or writing.

As the Documentation/filesystems/mandatory-locking.txt file in the kernel source tree indicates, there are several problems with mandatory locking. Firstly, *"the write system call checks for a mandatory lock only once at its start. It is therefore possible for a lock request to be granted after this check but before the data is modified. A process may then see file data change even while a mandatory lock was held."* There are also problems with `read(2)` and `mmap(2)`. Please consult this document if you have further interest in the issues surrounding mandatory file locking.

In Linux it's very clear that there is no love for mandatory locking and there have been attempts to remove it for many years. The "no regressions" rule such that APIs cannot be removed in case it breaks an old application have resulted in the code still being there even though it's unknown whether anything still used mandatory locks.

As such, no further time will be given to mandatory locking.

2.10.5 Open File Description Locks

Linux introduced file-private POSIX locks in the 3.x kernel series with the goal of taking elements of both BSD-style and POSIX-style locks and combining them into a more robust locking API by removing some of the ambiguity that existed across the various UNIX platforms. The following article describes the work that was done and covers some of the problems that were solved. It also provides an example of using open file description locks in a multi-threaded environment. This section will highlight those changes.



URL 4

<https://lwn.net/Articles/586904/>

One specific problem with POSIX locks is that when a request for a lock is made that would conflict with an existing lock previously taken by the same process, the kernel treats it as a request to modify the existing lock. This makes POSIX locks as good as useless in a multi-threaded application.

If a process closes a file then all locks will be released even if a lock was taking using a different file descriptor (see `dup(2)` and friends). Also, as the above article states *"This is a particular problem for applications that use complex libraries that do file access. It's common to have a library routine that opens a file, reads or writes to it, and then closes it"*

again, without the calling application ever being aware that has occurred. If the application happens to be holding a lock on the file when that occurs, it can lose that lock without ever being aware of it. That sort of behavior can lead to silent data corruption, and loss of developer sanity.". This could be, and has been, a very difficult problem to debug.

In general, if you need to use advisory locking, use the newer Open File Description Locks and avoid using POSIX-style locking.

2.11 Synchronizing I/O Operations

Generally speaking, when data is written to a file, it is not automatically written to disk but written to the page cache that resides in the Linux kernel address space. Modified data will eventually be written to disk. Obviously this presents an issue if the system were to crash prior to this data being written. The same is true of the file's metadata which could include the file size.

There are two system calls that allow for flushing modified data and metadata that may be sitting in kernel caches that have not yet made it to disk. Both are described in the `fsync(2)` manpage:

```
#include <unistd.h>

int fsync(int fd);
int fdatasync(int fd);
```

The `fsync(2)` system call will flush any modified data that is still in memory. This includes any file data as well as file metadata (file attributes). This ensures that the changes will make it to disk such that if the system crashes, any data can still be retrieved.

The `fdatasync(2)` system call is similar but does not flush metadata.

2.12 Direct I/O

When file I/O occurs, data is cached in the kernel inside the *page cache*. If data read or written is in the page cache it can be accessed without additional I/Os. This type of I/O is also called *buffered I/O* since the data is *buffered* in the kernel. Traditionally it was cached in the *buffer cache*.

Not all applications want data to be cached since they can better manage that cache themselves. The typical example is that of databases which allocate very large caches in the user address space.

Figure 2.4 shows the three different paths that I/O can take. There are subtle differences between buffered I/O and mapped I/O.

1. **Buffered I/O** – this is the default type of I/O. Access goes through the filesystem and data is cached in the page cache. Strictly speaking, for writes, the data is written into page cache pages and at a later data, a call is made into the filesystem to flush them to disk. For reads, pages are allocated and a call is made into the filesystem to read data from disk.

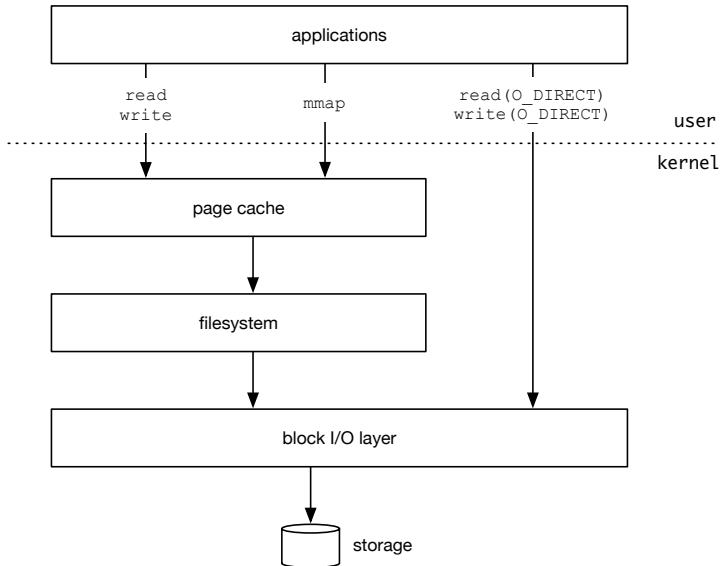


Figure 2.4: There are 3 paths to disk from applications

2. **Memory mapping** – applications access files through mappings into their address space. When data needs to be read or written, the kernel calls into the filesystem to perform the I/O. Once a page is in memory it will continue to be accessed through the mapping until a later date when data is flushed or the mapping is destroyed.
3. **Direct I/O** – the page cache is bypassed and data transfers directly between the process address space and disk.

Applications can request direct I/O by passing the `O_DIRECT` flag to the `open(2)` system call. To enable direct I/O applications must be compiled with `_ALL_SOURCE` enabled to see the definition of `O_DIRECT`.

The performance gains due to direct I/O are very dependent on the application's ability to effectively manage its own cache of data. There may be additional complexities if a file is being accessed through a mix of buffered I/O and direct I/O, certainly a situation that should be avoided.

Once a file is opened, reads and writes proceed as normal with the exception of alignment of user addresses and the amount of I/O requested. These restrictions are per-filesystem and there is no generic way in which to determine what the constraints imposed are.

- The number of bytes to be transferred is a multiple of 512 bytes.
- The file offset is a multiple of 512 bytes.
- The user-supplied memory address is aligned on a 512-byte boundary.

Note that some filesystems may have different requirements. For example, VxFS only requires a 8-byte boundary for the user address. XFS has the `XFS_IOC_DIOINFO` command that can be passed to `xfsctl(3)` to determine requirements.

There is no easy way to show direct I/O in progress or when it works or doesn't work. However, section 6.17 will describe the implementation of direct I/O and gives examples to show that the page cache is in fact bypassed.

2.13 Sparse Files

A sparse file is a regular file that has one or more *holes*. If you're not familiar with sparse files, that's going to sound like a very strange statement. Another way to put it is that a sparse file is a file that allows for efficient storage allocation for large amounts of data. If you've used virtual machines at any point in time, they're underpinned by sparse files. For example, consider a VM that has a 60 GB disk. When the VM is created, it doesn't need this much space and likely to be a small fraction. So why allocated 60 GB of disk space if it's not needed? This becomes a bigger problem as more and more VMs are created on the same storage infrastructure.

Sparse files are also useful when large parts of the file contain zeros. This avoids all data blocks being allocated up front. When data is written over a hole, new blocks will be allocated.

Sparse files can be found using the `find(1)` command. For example, looking inside `/var/log`, there are two sparse files:

```
$ sudo find /var/log -type f -printf "%S\t%p\n" | \
  gawk '$1 < 1.0 print'
0.013824      /var/log/lastlog
0.127872      /var/log/faillog
```

Selecting one of these files, `ls(1)` can be used to view the file size and the number of blocks backing the file:

```
$ ls -lh /var/log/lastlog
-rw-rw-r-- 1 root utmp 290K Aug  1 17:16 /var/log/lastlog
$ du -sh /var/log/lastlog
4.0K    /var/log/lastlog
```

There is only 4 KB allocated to the file but the file size is 290 KB.

2.13.1 Sparse Files in Action

Let's demonstrating with a small program which opens/creates a file, writes "hello" at offset 0 then seeks to an offset of 16 KB and writes "world". We're going to have to explore some details inside the filesystem to explain what's happening.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
```

```
#include <fcntl.h>

char *buf1 = "hello";
char *buf2 = "world";

int
main(int argc, char *argv[])
{
    int fd = open("/mnt/foo", O_CREAT|O_WRONLY, 0700);
    write(fd, buf1, strlen(buf1));
    lseek(fd, 16384, SEEK_SET);
    write(fd, buf2, strlen(buf2));
}
```

Now let's look at the file:

```
# ls -l /mnt/foo
-rwx----- 1 root root 16389 Dec 15 20:54 /mnt/foo
```

This is the expected size of the file but what's after "hello" and before we get to "world" at an offset of 16384? The answer isn't always consistent and depends on the underlying filesystem and its properties. But to give one example and where a file is sparse, we've created this file on top of SPFS, the disk-based filesystem whose implementation will be described in chapter 7.

This particular filesystem has a block size of 2048 bytes. This is the minimum amount of space that can be allocated to any file. Ignoring our sparse file example, if we were to start writing sequentially to a file at 0, SPFS would allocate blocks to the file as follows:

- bytes 0 - 2047 — allocate block
- bytes 2048 - 4195 — allocate block
- bytes 4196 - 6143 — allocate block
- and so on

If we only wrote 1 byte at offset 0, we'd get 1 block allocated. If we wrote 2048 bytes at offset 0 (for a total of 2049 bytes), we'd get two blocks allocated and so on.

Let's take a look at the file using the `stat(1)` command:

```
# stat /mnt/foo
  File: /mnt/foo
  Size: 16389          Blocks: 2 IO Block: 2048 regular file
Device: 800h/2048d      Inode: 4   Links: 1
Access: (0700/-rwx-----)  Uid: ( 0/  root)  Gid: ( 0/  root)
Access: 2022-12-15 20:54:09.000000000 +0000
Modify: 2022-12-15 20:54:09.000000000 +0000
 Change: 2022-12-15 20:54:09.000000000 +0000
```

There are 2 blocks allocated to this file as highlighted. Each block is 2048 bytes so that's a total of 4096 bytes. But our file is now 16389 bytes according to `ls(1)`. Both are correct.

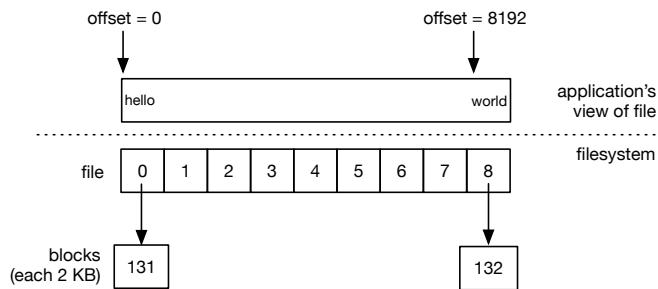


Figure 2.5: A sparse file only showing 2 allocated blocks

The file is sparse so has a hole in the middle. Next, we are going to use the SPFS filesystem debugger and display the contents of the inode for the file that we've written. We'll cover inodes in detail later but just think of it as a filesystem structure that records information about a file. You can see the file size and also see that a block (131) has been allocated for block offset 0 and another block (132) has been allocated at block offset 8. There are no allocated blocks for block offsets 1 through 7.

```
spfsdb > i4
inode number 4
  i_mode      = 81c0
  i_nlink     = 1
  i_atime     = Thu Dec 15 20:54:09 2022
  i_mtime     = Thu Dec 15 20:54:09 2022
  i_ctime     = Thu Dec 15 20:54:09 2022
  i_uid       = 0
  i_gid       = 0
  i_size      = 16389
  i_blocks    = 2
  i_addr[ 0] = 131 i_addr[ 8] = 132
```

The layout of the file is shown pictorially in figure 2.5. If the application were to read the file, it would see "hello" followed by a lot of zeroes followed by "world".

In fact, this doesn't work - **XXX—bug - see notes. Needs fixing**

2.13.2 Sparse File Example 1 — Virtual Machines

Virtual Machines are a wonderful application of how sparse files achieve the goal of mimicking large devices but only allocate data as and when needed. Disks are generally represented as sparse files in some underlying filesystem that is storing the virtual disks. When you create a virtual disk, under the covers, the hypervisor creates a sparse file on disk in a similar way to our simple program above. As the VM operating system writes blocks to disk, the hypervisor translates these writes into lseek/write combinations. Over time, you end up with a sparse file that has many holes and an ever increasing amount of allocated

storage as the VM writes more and more data

In commercial implementations with 100s or 1000s of VMs, the cost savings can be huge. Storage does not need to all be allocated up front and can be added later as VM disks start to grow. Furthermore, data deduplication is typically deployed. 100s of VMs have a very similar footprint in terms of operating system binaries and such like.

2.13.3 Sparse File Example 2 — HSM Applications

An older but very relevant example of how sparse files were used by applications was for HSM (Hierachal Storage Management) applications. There was a whole standard developed around this called DMIG (Data Management Interfaces Group). When I was at SCO, I used to travel from the UK to the US every 6 weeks or so to attend meetings. I got to visit several new cities and meet a lot of prominent operating system and filesystem engineers from the top companies at the time.

In the early 90s, disks were much smaller than you see today and also very expensive. Therefore to save space, several vendors developed what were then called HSM applications. The goal of such applications was to run periodically and perform the following:

- Look for files that have not recently been accessed.
- Migrate the *punched data* off to tape or other cheap storage.
- Punch a hole in the file. More likely, the hole would be from a specific offset to the end of the file.
- Make sure that the timestamps were not modified.

A small amount of data was left at the start of each *managed file* to satisfy requests from commands such as `file(1)` which would read from the start of the file to determine file type. If a request was made to read/write to a portion of the file that had been migrated, the application would be paused while the data was read from tape and written back to the file.

Sound complicated? Yes it was and was made even more complicated by the fact that there was a lot of hacking of existing filesystems and parts of the kernel to make this work across different operating systems. And with several commercial versions of UNIX at the time, porting HSM applications was prohibitive.

But this was a big enough problem and there was enough customer demand that the DMIG standard was developed and published and many operating system and filesystem vendors implemented support. Move forward twenty years and there is little information on the web now. In fact a Google search while writing this showed a fragment from my previous book on UNIX Filesystems but little else.

2.14 Buffered I/O and the Standard I/O Library

We covered the `read(2)` and `write(2)` system calls earlier and other sections in this chapter cover specialized I/O capabilities such as vectored reads/writes and asynchronous I/O. This section describes how the standard I/O library operates.

As the `stdio(3)` man page says:

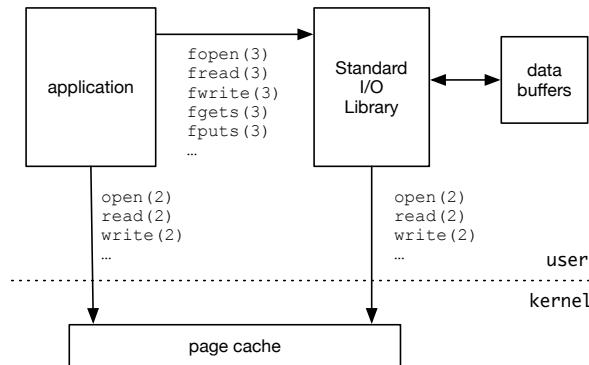


Figure 2.6: Using the Standard Library vs System Calls

The standard I/O library provides a simple and efficient buffered stream I/O interface. Input and output is mapped into logical data streams and the physical I/O characteristics are concealed.

Figure 2.6 shows the position of the standard I/O library in relation to an application and the Linux kernel. Developers have two choices when operating on a file. They can use system calls directly or they can use the standard I/O library which contains a wide array of functions for dealing with file I/O. The `stdio(3)` manpage provides a good overview of the standard I/O library and the functions that it provides. There are approximately 56 different functions available. All C developers will be familiar with the `printf(3)` function including their first "*Hello world*" program. Well, `printf(3)` is one of the 56 functions and by default, it writes to the standard output file stream.

Whether to use system calls directly or use the standard library is very dependent on what your application does. If you are developing an application that processes text files, the standard library is generally the better choice. You can read/write single characters at a time, read lines terminated by newlines or format the data then write to a file stream. For the standard I/O library, `printf(3)` and friends ease the job of formatting text to be written. The same is true for `scanf(3)` and similar functions. If you use the system calls `read(2)` and `write(2)` and want to process text, you'll likely repeat the work that the standard I/O library has already done for you.

2.14.1 The Standard I/O Library `FILE` Structure

Let's take a look at a few functions provided by the standard I/O library. In all cases the `stdio.h` header file needs to be included.

```
#include <stdio.h>

FILE *fopen(const char * restrict path,
            const char * restrict mode);
```

```

int fclose(FILE *stream);
size_t fread(void *restrict ptr, size_t size,
              size_t nitems, FILE *restrict stream);
int fscanf(FILE *restrict stream,
            const char *restrict format, ...);

```

The standard library operates on a *file stream* as defined by the FILE structure. The stdio.h header file doesn't contain the definition of struct FILE. For that you need to look at the following header file:

```
/usr/include/aarch64-linux-gnu/bits/types/FILE.h
typedef struct _IO_FILE FILE;
```

which is part of the glibc source repository. To get to "struct _IO_FILE" you need to look in the glibc sources:

- In the glibc source code

```
glibc/libio/bits/types/struct_FILE.h
```

- On my Ubuntu VM under /usr/include

```
aarch64-linux-gnu/bits/types/struct_FILE.h
```

Here is part of the structure, specifically those fields that we'll be covering in this section to help understand how buffered I/O works:

```

struct _IO_FILE
{
    int             _flags;
    char           *_IO_read_ptr;
    char           *_IO_read_end;
    char           *_IO_read_base;
    char           *_IO_write_base;
    char           *_IO_write_ptr;
    char           *_IO_write_end;
    char           *_IO_buf_base;
    char           *_IO_buf_end;
    char           *_IO_save_base;
    char           *_IO_backup_base;
    char           *_IO_save_end;
    struct _IO_marker *_markers;
    struct _IO_FILE  *_chain;
    int             _fileno;
    int             _flags2;
    unsigned short _cur_column;
    signed char     _vtable_offset;
    char           _shortbuf[1];
    _IO_lock_t      *_lock;
};
```

One of the best ways to view what is happening when you use the standard I/O library is run a simple program under `gdb`, set breakpoints and view the different fields in the `FILE` structure. Here is a very simple program which will be used to demonstrate this point:

```
1 #include <stdio.h>
2
3 int
4 main()
5 {
6     FILE *fp;
7     int         c;
8
9     fp = fopen("lorem-ipsum", "r");
10    c = fgetc(fp);
11    printf("char = %c\n", (char)c);
12 }
```

The "`lorem-ipsum`" file contains the commonly seen Latin text starting "`Lorem ipsum dolor sit ...`" which is used throughout the book. In case you've ever wondered where this text came from, Wikipedia informs us that:

*Lorem ipsum is typically a corrupted version of *De finibus bonorum et malorum*, a 1st-century BC text by the Roman statesman and philosopher Cicero, with words altered, added, and removed to make it nonsensical and improper Latin.*

In our `gdb` session, we'll set a breakpoint, run the program and display the character read ("L" as expected) as well as the contents of the `FILE` structure:

```
(gdb) b 11
Breakpoint 1 at 0x840: file fget.c, line 11.
(gdb) run
Starting program: /home/spate/spfs/test/fget
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu...

Breakpoint 1, main () at fget.c:11
11             printf("char = %c\n", (char)c);
(gdb) p c
$1 = 76
(gdb) p (char)c
$2 = 76 'L'
(gdb) p *fp
$3 = {_flags = -72539000,
       _IO_read_ptr = 0xaaaaaaaaac1481 "orem ipsum dolor sit amet"..., 
       _IO_read_end = 0xaaaaaaaaac201c "", 
       _IO_read_base = 0xaaaaaaaaac1480 "Lorem ipsum dolor sit"..., 
       _IO_write_ptr = 0xaaaaaaaaac1480 "Lorem ipsum dolor sit"..., 
       _IO_buf_base = 0xaaaaaaaaac1480 "Lorem ipsum dolor sit"..., 
       _IO_buf_end = 0xaaaaaaaaac2480 "", _IO_save_base = 0x0,
```

```

_IO_backup_base = 0x0, _IO_save_end = 0x0, _markers = 0x0,
_chain = 0xfffff7fa1520 <_IO_2_1_stderr_>, _fileno = 3,
_flags2 = 0, _old_offset = 0, _cur_column = 0,
_vtable_offset = 0 '\\000', _shortbuf = "",
_lock = 0xaaaaaaaaaac1380, _offset = -1, _codecvt = 0x0,
_wide_data = 0xaaaaaaaaaac1390, _freeres_list = 0x0,
_freeres_buf = 0x0, __pad5 = 0, _mode = -1,
_unused2 = '\000' <repeats 19 times>
}

```

The `_fileno` field is the file descriptor referencing the open file. Since this is the only file used by the application other than the standard file descriptors, when the file is opened a file descriptor of 3 is returned.

The `_IO_buf_base` points to a buffer in which data is read into and written from. The `_IO_buf_end` points to the end of the buffer so, in theory we have 4 KB of data that's been read from the file. But since the file is only 2,972 bytes in size we see that `_IO_read_end` is set to `0x201c`. Subtracting the two we have `0x201c - 0x1481 = 0xB9B` which in decimal is 2971. Since the first character of the file is stored at `0x201c`, this makes 2972, the size of the file.

The `_IO_read_ptr` and `_IO_write_ptr` fields reference the position within this 4 KB buffer where we are currently reading and writing from/to. Since we've read a single character from the file, `_IO_read_ptr` is incremented to point to the next position in the buffer (the character "o").

2.14.2 Analyzing the glibc Standard I/O Library

You can download the most recent glibc source code as follows:

```

# git clone https://sourceware.org/git/glibc.git
# cd glibc
# git checkout master

```

As an example of how to see what glibc is doing, navigate to the `libio` directory and choose the field of the `FILE` that you wish to analyze, for example:

```
# grep _IO_read_ptr *
```

You will see lots of references. To understand the flow through the library calls you'll need to spend some time analyzing the code. There are several indirections. For example, for `fread(3)` start at `_IO_fread()` in `iofread.c` and go from there. You can also use the Elixir Cross Referencer, for example:

```
https://elixir.bootlin.com/glibc/latest/source/libio/iofread.c
```

I actually found it easier to grep. For example, search for the following:

```

'fp->_IO_read_ptr'
'fp->_IO_read_ptr =' 
'fp->_IO_read_ptr+' 
'fp->_IO_read_ptr-'

```

and you'll see a smaller set of hits. For the highlighted option, there were only 3 hits:

```
$ grep 'fp->_IO_read_ptr+' *c
genops.c:     return *(unsigned char *) fp->_IO_read_ptr++;
genops.c:     return *(unsigned char *) fp->_IO_read_ptr++;
genops.c:     return *(unsigned char *) fp->_IO_read_ptr++;
```

Looking inside `genops.c` in the function `__uflow()`, there is a fragment of code:

```
if (fp->_IO_read_ptr < fp->_IO_read_end)
    return *(unsigned char *) fp->_IO_read_ptr++;
```

A character is being returned from the current read pointer and the read pointer is being incremented as long as the read pointer still references data in the buffer (first line).

I started to experiment further by extending the program to do the following and printing out various pointers in the `FILE` struct to see how the pointers changed. I was particularly curious when mixing reads and writes:

```
c = fgetc(fp);
c = fgetc(fp);
c = fgetc(fp);
c = fputc('1', fp);
c = fputc('2', fp);
c = fputc('3', fp);
c = fgetc(fp);
```

The result was not what I expected. After the first `fgetc()`, I display two addresses and for each get/put call, I printed out `_IO_read_ptr` and `_IO_write_ptr` as follows:

```
$ ls -l lorem-ipsum
-rw-r--r-- 1 spate spate 2972 Dec 12 21:00 lorem-ipsum
$ ./fget
_IO_read_base = 0xaaab0b8b3480
_IO_read_end  = 0xaaab0b8b401c (2972 difference)

_IO_read_ptr   _IO_write_ptr
-----
0xaaab0b8b3481 0xaaab0b8b3480
0xaaab0b8b3482 0xaaab0b8b3480
0xaaab0b8b3483 0xaaab0b8b3480
0xaaab0b8b401c 0xaaab0b8b3484
0xaaab0b8b401c 0xaaab0b8b3485
0xaaab0b8b401c 0xaaab0b8b3486
0xaaab0b8b3481 0xaaab0b8b3480
Character read = i
$ head -1 lorem-ipsum
Lor123ipsum dolor sit amet, consectetur adipiscing
```

This doesn't quite meet expectations. We know from the previous example that we get 'L', 'o' and 'r' for our first 3 `fgetc()` calls and the read pointer advances through the buffer as expected. When we issue the first `fputc()` call, we don't write at `_IO_write_ptr` but at `_IO_read_ptr`. This makes sense as we are advancing through the file. But after the first `fputc()`, the value of `_IO_read_ptr` is set to `_IO_read_end`. Quite strange.

And when we issue our final call to `fgetc()` we get the character 'i' which is contained at offset `_IO_write_ptr`!

I decided at this point that finding the answer was really beyond the scope of this book since the goal was to introduce the standard I/O library, not to explain the details of how it works internally. But I was intrigued enough that I set it as a goal to write a future blog entry so please take a look at my website.

2.15 Memory Mapped Files

Memory mapped files were first introduced by Sun in their SunOS version 4 operating system in late 1988 (soon to be renamed Solaris) and described in the classic Usenix paper "*Virtual Memory Architecture in SunOS*". I recommend reading it to find out the history of how UNIX moved from the buffer cache for all file I/O to a hybrid buffer cache / page cache. The new architecture allowed for the integration of all I/O to use the new virtual memory subsystem and a move away from the kernel buffer cache for file data. This architecture was adopted in the System V Release 4 version of UNIX and the architectural principles were later adopted in Linux.



5 <https://tinyurl.com/2p9cymm2>

The new architecture touched a large part of the operating system and soon after it was introduced, the development system was largely rewritten such that dynamic libraries made extensive use of `mmap(2)`.

How does `mmap(2)` work? Let's start with a simple example which opens a file and maps the whole file into memory. The call to `mmap(2)` returns the user-space address to where the file is mapped. The program then proceeds through the mapping, one address at a time, writing each character read to `stdout`.

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int
main()
{
    struct stat st;
    char        *addr;
    int         pgsz, fd, i;

    pgsz = getpagesize();
    printf("PAGESIZE = %d\n", pgsz);
    fd = open("/mnt/big-lorem-ipsum", O_RDONLY);
```

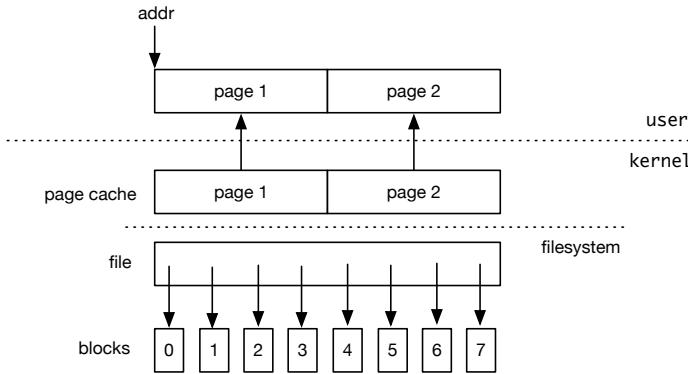


Figure 2.7: Mapping an 8 KB file into memory

```

fstat(fd, &st);
addr = (char *)mmap(NULL, st.st_size, PROT_READ,
                     MAP_SHARED, fd, 0);
close(fd);

for (i=0 ; i < st.st_size ; i++) {
    putchar(*addr);
    addr++;
}
}

```

When the program is run, the page size is displayed together with the contents of the file:

```

$ ./map
PAGESIZE = 4096
Lorem ipsum dolor sit amet, consectetur ...

```

This file being read is exactly 8KB so will occupy two full pages. What happens somewhat conceptually is shown in figure 2.7. This figure shows that the file has 8 disk blocks (1 KB block size) which may or may not be allocated next to each other on disk. The filesystem will attempt to place blocks together but this allocation is not guaranteed.

As we walk through the mapping, the kernel will call the filesystem to bring data into memory. This data is cached in the Linux page cache in the kernel. Here, two pages are allocated. Data is then copied from the kernel pages into the user pages starting at address `addr`. If we run the program again, it is very likely that the pages in the Linux page cache will still be present so no more I/O will take place.

2.15.1 The `mmap(2)` / `munmap()` System Calls

The definition of `mmap(2)` and `munmap(2)` are as follows:

```

void *mmap(void *addr, size_t length, int prot, int flags,

```

```
int fd, off_t offset);  
  
int munmap(void *addr, size_t length);
```

The `addr` argument is a hint as to where the region of the file will be mapped in memory. It must be correctly aligned according to the architecture on which the process is running. A page-aligned address should be used. The `prot` argument specifies the memory protection of the mapping and can be one of the following:

- `PROT_EXEC` – pages may be executed.
- `PROT_READ` – pages may be read.
- `PROT_WRITE` – pages may be written.
- `PROT_NONE` – pages may not be accessed.

The `flags` argument can be one of 20 different values. A few of the more widely used flags are:

- `MAP_SHARED` – the mapping can be shared with other processes. Updates to the mapping are visible to other processes mapping the same region of the file, and changes made by any process are carried through to the underlying file.
- `MAP_PRIVATE` – create a private copy-on-write mapping. Any updates to the mapping are not visible to other processes mapping the same part of the file, and are not carried through to the underlying file.
- `MAP_FIXED` – don't interpret `addr` as a hint and place the mapping at exactly the address specified by `addr`.

The `fd` argument specifies the file to be mapped and the `offset` argument is an offset within the file. It should be a multiple of the system's page size.

The `munmap(2)` system call can be called to delete the mapping for the specified address range (`addr` and `length`). Subsequent references to addresses within the range being unmapped will generate `SIGSEGV`. The `addr` argument must be a multiple of the page size (but the `length` argument need not be).

2.15.2 Other Mapped File Functions

There are several other functions related to file mappings that are highlighted below:

- `msync(2)` – flush changes made to the mapping back to the underlying file. An address and length are specified with flags indicating whether to perform writes synchronously, asynchronously and whether to invalidate any other mappings of the file so that subsequent access will bring in the new changes from disk.

- `mlock(2)` / `mlock2(2)` `mlockall(2)` – lock a number or all pages into memory to prevent them from being written to swap. A call to `mlock2(2)` differs only if a flag of `MLOCK_ONFAULT` is specified in which case resident pages are locked and if pages within the mapping are current paged out, they will be locked once they are faulted back into memory.
- `mprotect(2)` – when a call to `mmap(2)` is made, the `prot` argument specifies the access protections (for example `PROT_READ` or `PROT_WRITE`). This call allows those protections to be changed for an existing mapping. There are some additional access protections that are not part of the original `mmap(2)` system call.
- `mremap(2)` – expands or shrinks an existing memory mapping and potentially moving the mapping at the same time depending on what flags are specified.

Executables and corresponding libraries can be very large and it's unknown as to how much of the program binary will be used. The best method of reducing memory usage and I/O is to use *demand paging*. The loader will establish a mapping to the executable as well as the various dynamic libraries that the program is linked with. When the process starts executing it will only bring in pages that are accessed. To further reduce memory, the mappings to program binaries and libraries are shared between processes. Program executables only need read access.

2.16 Asynchronous I/O

The `read(2)` and `write(2)` system calls are synchronous, that is, each operation will not return until the data is read/written or an error condition occurs. Depending on the type of I/O, the Linux kernel may have to call the filesystem which may have to go to disk to complete the operation. This is a time-consuming task compared to what an application could be doing if it solely operated on data in memory. While vectored reads/writes allow multiple I/Os to take place during a single call, likely improving performance, the application will still pause until the operation completes. Further optimizations can take place with threads which we will discuss in the next section. But prior to the widespread availability of *pthreads* (process threads), asynchronous I/O, commonly called *async I/O*, was introduced whereby I/O operations could be started and the application would be notified asynchronously at a later time when the operation(s) completes. This allows the application to perform other tasks. One of the main drivers for *async I/O*, were the commercial databases such as Oracle.

The `aio(7)` manpage describes the different *async I/O* functions available. There is quite a lot to explain about these functions, but starting with the `read/write` interfaces:

```
#include <aio.h>

int aio_read(struct aiocb *aiocbp);
int aio_write(struct aiocb *aiocbp);
int lio_listio(int mode, struct aiocb *restrict const
              aiocb_list[restrict], int nitems,
              struct sigevent *restrict sevp);
```

The first two functions initiate single read/write operations while `lio_listio(3)` initiates the list of I/O operations described by the array `aiocb_list`. The `aiocb` structure (async I/O control block or AIO control block) contains the parameters used to specify the I/O operation or operations. It is defined in the `aiocb.h` header file as follows:

```
struct aiocb {
    int             aio_fildes;      /* File descriptor */
    off_t           aio_offset;     /* File offset */
    volatile void * aio_buf;        /* Location of buffer */
    size_t          aio_nbytes;     /* Length of transfer */
    int             aio_reqprio;    /* Request priority */
    struct sigevent aio_sigevent;  /* Notification method */
    int             aio_lio_opcode; /* Operation to be performed;
                                    lio_listio() only */
}
```

The first four fields will be recognizable as they are similar the arguments that would be passed to `pread(2)` or `pwrite(2)` so basically a call to `lseek(2)` followed by a read or write. The `aio_reqprio` field specifies the relative priority of the I/O request. If the process priority is X and `aio_reqprio` is 6, the resulting priority of the I/O task is $10 - 6 = 4$. **XXX—need to come back to this**

`aio_lio_opcode` is used by `lio_listio(3)` only and can be one of the following depending on what the operation is to perform:

```
enum { LIO_READ, LIO_WRITE, LIO_NOP };
```

The `lio_listio(3)` function can perform a mix of reads and writes with a single call. By setting `aio_lio_opcode` is set to `LIO_NOP` that specific AIO control block is ignored.

The `aio_sigevent` specifies the mechanism through which the application is notified when the async I/O operation completes.

```
struct sigevent {
    int             sigev_notify;   /* Notification type */
    int             sigev_signo;    /* Signal number */
    union sigval   sigev_value;   /* Signal value */
    void          (*sigev_notify_function)(union sigval);
                    /* Notification function */
    pthread_attr_t *sigev_notify_attributes;
                    /* Notification attributes */
};
```

I won't go into the details of each field of this structure but the example below will show how it is used. Refer to the `sigevent(7)` manpage for more information.

To demonstrate how async I/O works in practice, we will give an example of how to issue multiple async I/Os to the same but using the `lio_listio()` function. Figure 2.8 shows a file contains 10 0s followed by 10 1s, 10 2s and so on. Our goal is to read the 1s in an area of memory referenced by `buf1`, overwrite the 2s with the contents of the memory referenced by `buf2` and read the 3s in an area of memory referenced by `buf3`.

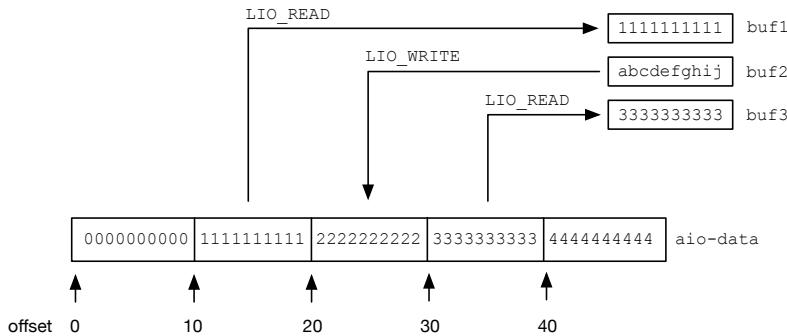


Figure 2.8: Reading and writing from/to a file using async I/O

Most of the program involves setting up our three AIO control blocks which we do in lines 31-45 after opening our file `aio-data`.

```

1 #include <fcntl.h>
2 #include <aio.h>
3 #include <signal.h>
4 #include <stdio.h>
5 #include <string.h>
6 #include <unistd.h>
7 #include <sys/uio.h>
8
9 char *buf2 = "abcdefgij";
10 char buf1[32];
11 char buf3[32];
12
13 void
14 aio_hdlr(int signo)
15 {
16     printf("buf1 = %.10s\n", buf1);
17     printf("buf3 = %.10s\n", buf3);
18 }
19
20 int
21 main(int argc, char *argv[])
22 {
23     struct sigaction act;
24     struct sigevent sevp;
25     struct aiocb *list_aio[3];
26     struct aiocb aio1, aio2, aio3;
27     int err, fd;
28
29     fd = open("aio-data", O_RDWR);

```

```

30
31     aio1.aio_fildes = fd;    aio1.aio_lio_opcode = LIO_READ;
32     aio1.aio_buf     = buf1;  aio1.aio_offset      = 10;
33     aio1.aio_nbytes  = 10;   aio1.aio_reqprio    = 0;
34
35     aio2.aio_fildes = fd;    aio2.aio_lio_opcode = LIO_WRITE;
36     aio2.aio_buf     = buf2;  aio2.aio_offset      = 20;
37     aio2.aio_nbytes  = 10;   aio2.aio_reqprio    = 0;
38
39     aio3.aio_fildes = fd;    aio3.aio_lio_opcode = LIO_READ;
40     aio3.aio_buf     = buf3;  aio3.aio_offset      = 30;
41     aio3.aio_nbytes  = 10;   aio3.aio_reqprio    = 0;
42
43     list_aio[0] = &aio1;
44     list_aio[1] = &aio2;
45     list_aio[2] = &aio3;
46
47     memset(&act, 0, sizeof(act));
48     act.sa_handler = aio_hdlr;
49     sigaction(SIGUSR1, &act, NULL);
50     act.sa_handler = SIG_IGN;
51     sigaction(SIGHUP, &act, NULL);
52
53     memset(&sevp, 0, sizeof(sevp));
54     sevp.sigev_signo = SIGUSR1;
55     sevp.sigev_notify = SIGEV_SIGNAL;
56     sevp.sigev_value.sival_ptr = (void *)list_aio;
57
58     err = lio_listio(LIO_NOWAIT, list_aio, 3, &sevp);
59     pause();
60 }

```

XXX—need to either explain the issue with SIGHUP or fix it!!! - also, perhaps we can have the signal handler analyze the aiocbs (re sigevent) otherwise all that stuff on 53-56 is kind of wasted

Additional info about the program is as follows:

- Lines 47 - 51 – When the async I/O operations complete the program will be notified by a signal. I’m choosing SIGUSR1 as the signal to use and my signal handler is the function `aio_hdlr`. Lines 47 - 49 provide the set up needed. **XXX—SIGHUP dunno**
- Lines 53 - 56 – `lio_listio(3)` requires a pointer to a `sigevent` structure which tells it which signal to post and a pointer to a structure which will be passed to the signal handler. This allows the program to check for the status of individual I/O operations. For more details about how async I/O uses signal handler, refer to the `sigevent (7)` manpage.
- Line 58 – Call `lio_listio(3)` to initiate the I/O operations.

- Line 59 – Pause the program to wait for signal handling. Once the signal handler runs, `pause(2)` returns and the program will exit.

Before running the program, we display the contents of the `aio-data` file, then run the program and then display the file contents again. As you can see, `buf1` and `buf2` contain the data expected and `aio-data` is overwritten at offset 20 with the string `abcdefghijklm` referenced by `buf2`.

```
$ cat aio-data
0000000000111111111222222222333333334444444444
$ ./aio
buf1 = 1111111111
buf3 = 3333333333
$ cat aio-data
00000000001111111111abcdefghijklm33333333334444444444
```

2.16.1 Additional Async I/O Functions

There are several other functions available that are needed to build enterprise applications and worth exploring. They are highlighted below. Also, view the `aio(7)` manpage for additional information.

- `aio_fsync(3)` — Queue a *sync* request for the I/O operations on the file descriptor specified in the AIO control block. This is somewhat equivalent to calling of `fsync(2)` and `fdatasync(2)`. If the `op` argument is `O_SYNC`, then all currently queued I/O operations shall be completed as if by making a call to `fsync(2)`, and if `op` is `O_DSYNC`, this call is the asynchronous analog of `fdatasync(2)`. This function doesn't wait for I/Os to completion.
- `aio_error(3)` — Obtain error status of a queued I/O request.
- `aio_return(3)` — Obtain the return status of a completed I/O request.
- `aio_suspend(3)` — Suspend the caller until one or more of a specified set of I/O requests completes.
- `aio_cancel(3)` — Attempt to cancel outstanding I/O requests on a specified file descriptor.

2.16.2 Performance Gains with Async I/O

Disk have come a long way since the POSIX async I/O standard was being developed back in the early 1990s. But even with faster disks, disk subsystems and SSDs, it's still slow to *go to disk* compared with reading from an in-core cache, whether the Linux page cache or a database cache. Also recall that at the time of the async I/O standard being developed, CPU cores were non-existent and the multi-threading standard known as pthreads was still being developed. When I was at ICL (International Computers Limited), we were implementing async I/O on SVR4 UNIX implementation on top of the Chorus microkernel to support

Oracle. I recall the performance gains made by the database to be huge although it was a long time ago so I don't recall the details.

You don't have to worry about threading or race conditions or losing error state (as much). AFIO does all that for you. On platforms which provide native asynchronous file i/o support and/or native scatter/gather file i/o support, AFIO will use that instead of issuing multiple filing system operations using a thread pool to achieve concurrency. This can very significantly reduce the number of threads needed to keep your storage device fully occupied — remember that queue depth i.e. that metric you see all over storage device benchmarks is the number of operations in flight at once, which implies the number of threads needed. Most storage devices are IOPS limited due to SATA or SAS connection latencies without introducing queue depth — in particular, modern SSDs cannot achieve tens of thousands of IOPS range without substantial queue depths, which without using a native asynchronous file i/o API means lots of threads. It's very, very easy to have AFIO turn off file system caching, readahead or buffering on a case by case basis which makes writing scalable random synchronous file i/o applications much easier.

XXX—need to come back and enhance once the FS performance chapter is finished

Oracle but any info? Not easy to find.

Lighty - <http://blog.lighttpd.net/articles/2006/11/12/lighty-1-5-0-and-linux-aio/>
wikipedia -

just search for "application+performance+gains+examples" - there are lots of examples

2.16.3 procfs Async I/O Information

There are two files in the proc file system that can be tuned for asynchronous I/O. Both can be found under /proc/sys/fs:

- aio-nr — the current number of system-wide asynchronous I/O requests. If there are no async I/O requests active, this number will be 0.
- aio-max-nr — the maximum number of allowable concurrent requests. This is generally 64 KB.

2.17 Truncating Files

When describing the `open(2)` system call in section 2.7.1, the `O_TRUNC` flag can be passed to ensure that the file is truncated to length of zero if the file already exists. If a file needs to be truncated either without opening it or after the file is opened, the following system calls can be used:

```
#include <unistd.h>
```

```
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

The difference between using `O_TRUNC` during a call to `open(2)` and the above interfaces is that the file will be truncated to exactly `length` bytes.

A file can be truncated down in size or it can be truncated up. I always found that amusing when I very first saw this description many years ago. How do you allocate space to a file? "Truncate the file up"!

If the size of the file is larger than the size specified by `length`, the extra data will be lost following the truncation. If the file is shorter, it will be extended. The extended part will read as null bytes ('\0').

2.18 Multi-threaded Applications

There have been several books written about multi-threaded applications and for the most part, multi-threading is beyond the scope of this chapter. The following web page gives a very good introduction into multi-threading and introduces not just multi-threading concepts but also pthreads, POSIX threads, the default method of multi-threading on Linux.



URL 6 <https://tinyurl.com/5ssbe8db>

The `pthreads(7)` manpage is the place to start to learn about the programming interfaces available that underpin pthreads. Over the last twenty years, Linux has seen two different implementations for pthreads:

- LinuxThreads – the original Pthreads implementation which, since glibc 2.4, is no longer supported.
- NPTL (Native POSIX Threads Library) – the modern pthreads implementation. By comparison with LinuxThreads, NPTL provides closer conformance to the requirements of the POSIX.1 specification than LinuxThreads. It also has better performance when creating large numbers of threads. NPTL has been available since glibc 2.3.2, and requires features that are present since the Linux 2.6 kernel.

Start with `pthreads(7)` and go from there. Most articles on line will reference the newer implementation. The `getconf(1)` command can be used to determine which implementation is available:

```
$ getconf GNU_LIBPTHREAD_VERSION
NPTL 2.36
```

When developing applications that access files / filesystems, there aren't many things to bear in mind when writing multi-threaded applications:

- All open files are shared between all process threads. If a thread (other than `main()`) opens a file, it should be closed before the thread terminates.

- Linux file locking has issues when deploying multi-threaded applications. Please refer to section 2.10 and make sure you use the newer *Open File Description Locks*.
- It should go without saying that any change that any thread makes, including calls resulting in a state change in the kernel, are visible to all other threads. Since file descriptors are shared, offsets within each file are also shared.

2.19 Directory Creation

Directory creation is handled by the `mkdir(2)` and `mkdirat(2)` system calls:

```
#include <sys/stat.h>

int mkdir(const char *pathname, mode_t mode);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <sys/stat.h>

int mkdirat(int dirfd, const char *pathname, mode_t mode);
```

For `mkdir(2)`, the pathname can be absolute or relative. The mode argument specifies the mode for the newly created directory. It is modified by the process's umask using the formula (`mode & ~umask & 0777`).

How the `mkdirat(2)` system call operates depends on pathname. Either:

- If pathname is relative, it is interpreted relative to the directory referred to by the file descriptor dirfd as opposed to being relative to the current working directory of the calling process (as is done by `mkdir(2)` for a relative pathname).
- If pathname is relative and dirfd is set to AT_FDCWD, pathname is interpreted relative to the current working directory of the calling process (as for `mkdir(2)`).
- If pathname is absolute, dirfd is ignored.

Removing a directory is very straightforward:

```
#include <unistd.h>

int rmdir(const char *pathname);
```

The directory must be empty before this operation can succeed.

2.20 Hard Links and Symbolic Links

Creating a hard link with the `link(2)` or `linkat(2)` system calls involves creation of a new directory entry that references the file to which it is linked to. Following a successful call, the file referenced by `newpath` can be used in exactly the same manner as the file `oldpath`, to which it is linked.

```
#include <unistd.h>

int link(const char *oldpath, const char *newpath);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <unistd.h>

int linkat(int olddirfd, const char *oldpath,
           int newdirfd, const char *newpath, int flags);
```

Below is a simple example showing the effects of calling `link(2)`. All the program does is create a hard link called "latin-text" to the existing file "lorem-ipsum".

```
#include <unistd.h>

int
main()
{
    link("lorem-ipsum", "latin-text");
}
```

Before the program is run, the attributes of "lorem-ipsum" are displayed. The link count for this file is "1". The program is then run and the link count goes to "2". When the attributes of "latin-text", you can see they mirror the attributes of "lorem-ipsum".

```
$ ls -li lorem-ipsum
409993 -rw-r--r-- 1 spate spate 2972 Apr  4 16:43 lorem-ipsum
$ ./link
$ ls -li lorem-ipsum
409993 -rw-r--r-- 2 spate spate 2972 Apr  4 16:43 lorem-ipsum
$ ls -li latin-text
409993 -rw-r--r-- 2 spate spate 2972 Apr  4 16:43 latin-text
$ diff lorem-ipsum latin-text
```

The `linkat(2)` system call differs from `link(2)` primarily in how the two files are referenced. See the previous section on how `mkdirat(2)` operates. The main difference here is that there is an additional `flags` argument which can be a bitwise OR of the following:

- `AT_EMPTY_PATH` – If `oldpath` is an empty string then the link will be created to reference the file `olddirfd`.
- `AT_SYMLINK_FOLLOW` – By default, `oldpath` will not be dereferenced if it is a symbolic link, the same being is true with `link(2)`. If `AT_SYMLINK_FOLLOW` is specified in `flags` and `oldpath` is a symbolic link, it will be dereferenced.

The `unlink(2)` and `unlinkat(2)` system calls can be used to remove a link to a file:

```
#include <unistd.h>

int unlink(const char *pathname);
```

```
#include <fcntl.h>           /* Definition of AT_* constants */
#include <unistd.h>

int unlinkat(int dirfd, const char *pathname, int flags);
```

If pathname is the last link to a file and no processes have the file open, the file will be deleted and any space it is using will be reclaimed and made available for other use. If pathname references a symbolic link, only the link is removed.

The `unlinkat(2)` system call operates the same way as both `unlink(2)` and `rmdir(2)`. If the flag argument is `AT_REMOVEDIR`, the equivalent of `rmdir(2)` will be performed.

2.20.1 Symbolic Links

The `symlink(2)` system call creates a symbolic link called `linkpath`. It will contain the string `target`. The existence of the target file is not checked during this operation.

```
#include <unistd.h>

int symlink(const char *target, const char *linkpath);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <unistd.h>

int symlinkat(const char *target, int newdirfd,
              const char *linkpath);
```

The `symlinkat(2)` system call operates in the same way as `symlink(2)` with the following two exceptions:

- If `linkpath` is relative, it will be interpreted relative to the directory referred to by the file descriptor `newdirfd`.
- If `linkpath` is relative and `newdirfd` is the value `AT_FDCWD`, `linkpath` is interpreted relative to the current working directory of the calling process (like `symlink()`).

2.21 Extended Attributes

A file contains both the regular stream of data as written by applications in addition to meta-data created and maintained by the filesystem. User-visible meta-data can be obtained by running one of the `stat(2)` system calls. Application developers pushed the operating system and small number of filesystem vendors for years to have them store user-defined attributes. As a specific example, back in the early to mid 1990s, I used to be a member of DMIG (the Data Management Interfaces Group) who were building storage management APIs to support Hierarchical Storage Management (HSM) applications. To save space on disk storage, HSM applications would punch holes in files accessed less frequently and

store that data on tape to be read back at a later time when the file was accessed. This was a complex process and required per-file meta-data about the data being migrated to tertiary storage. At this time few vendors support such *extended attributes*. But this was the beginning of what would become a POSIX standard.

Today, the following filesystems in Linux support extended attributes—ext2, ext3, ext4, JFS, XFS, btrfs, OCFS2 and squashfs. Extended attributes have a name and associated data. Attribute names can be up to 255 characters and the size of the associated data is variable depending on the filesystem.

Linux provides four namespaces for extended file attributes:

1. user
2. trusted
3. security
4. system

The name of the attribute must start with one of the four names shown above followed a "." and then followed an application supplied name. An example could be `user.comment`. The `xattr(2)` manpage contains details about the different extended attribute system calls and commands. The `setfattr(1)` and `getfattr(1)` commands can be used to set and get extended attribute. The systems calls are `setxattr(2)`, `getxattr(2)`, `removexattr(2)` and `listxattr(2)` (which is used by "getfattr -d").

2.21.1 Installing and Using Extended Attributes

On Ubuntu you need to install the `attr` package as follows:

```
$ sudo apt install attr
```

in order to start using extended attributes. In the example below, `ls` is run against the file "lorem-ipsum", an attribute is added and then `ls` is run again. You'll see that the file looks exactly the same.

```
$ ls -l lorem-ipsum
-rw-r--r-- 1 spate spate 2972 Dec  4 15:43 lorem-ipsum
$ setfattr -n user.comment -v "File contains Latin" lorem-ipsum
$ ls -l lorem-ipsum
-rw-r--r-- 1 spate spate 2972 Dec  4 15:43 lorem-ipsum
```

Extended attributes stored in a file can bee read the contents as follows:

```
$ getfattr -d lorem-ipsum
# file: lorem-ipsum
user.comment="File contains Latin"
```

Despite demand for extended attributes over the years, their adoption has been somewhat limited. On Linux, The Wikipedia page on extended attributes mentions that they are used by Beagle, OpenStack Swift, Dropbox, KDE's semantic metadata framework (Baloo),

Chromium, Wget and cURL. Desktop file manager usage has always been one of the most quoted examples but generally speaking, unless all underlying filesystems support extended attributes, the presentation to the user becomes more difficult to implementers since they can use extended attributes where available but then must use some other mechanism for storing attributes if the underlying filesystem does not support them. This results in a less than stellar presentation or an awkward implementation.

2.22 Inode Flags

Some of the Linux filesystems support the notion of *inode flags* which are described in the `ioctl_iflags(2)` manpage. Inode flags are attributes used to modify the semantics of files and directories. These flags can be retrieved and modified using the following two `ioctl(2)` operations:

```
int attr, fd;

fd = open("filename", ...);
ioctl(fd, FS_IOC_GETFLAGS, &attr); // get current flags
attr |= FS_NOATIME_FL;           // set the flags you want
ioctl(fd, FS_IOC_SETFLAGS, &attr); // update the file's flags
```

The first call retrieves the current flags and puts them in `attr`. The bitmask is then modified and a second call is made to set the new flags. There are 14 flags in total and changes apply even if the caller has superuser privileges. Here are a few of the available flags:

- `FS_APPEND_FL` – prevent the file from being opened only with `O_APPEND`.
- `FS_IMMUTABLE_FL` – the file will be set to be immutable; no subsequent changes can be made to the file contents or metadata.
- `FS_SYNC_FL` – make all future writes synchronous.
- `FS_NOATIME_FL` – the file's access time (atime) will not be updated on access. This can have significant performance benefits.

Inode flags are Linux specific, not part of any standard and not implemented by all of the Linux filesystems.

2.23 Reading Directory Entries

As you will see later in the book, parsing pathnames and handling directory entries is a complex task for both the kernel and filesystems to handle. In fact, supporting a very large number of files inside a single directory is a particularly complex task for filesystem developers.

Reading directory entries is performed by the `readdir(3)` library function which is passed a `DIR` structure, called a *directory stream*, obtained from calling `opendir(3)`. Both functions are defined as follows:

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
DIR *fdopendir(int fd);
int closedir(DIR *dirp);

struct dirent *readdir(DIR *dirp);
```

For `opendir(3)`, the name argument should reference a directory either as a relative or absolute pathname. On return, the directory stream will be positioned ready to read the first entry in the directory. The `fopendir(3)` library function also returns a directory stream but using a specified file descriptor. Note that the file descriptor should not be used for any other operation. The `closedir(3)` function closes the directory stream as well as the underlying file descriptor.

Given a directory stream, an application can make calls to `readdir(3)` with each call returning the next directory entry with each entry defined by the `dirent` structure. I've only included three of the fields of this structure. Only the first two fields are required by POSIX. In the following section, I'll show an example of a simple version of the `ls(1)` command which uses the first two fields but then uses the `lstat(2)` system call to get the file type.

```
struct dirent {
    ino_t          d_ino;        /* Inode number */
    char           d_name[256];  /* Null-terminated filename */
    unsigned char  d_type;      /* Type of file */
};
```

Once all directory entries have been returned, the return value will be `NULL`. You should not make any assumption about the order in which directory entries are returned. This will likely differ from one filesystem to the next.

2.23.1 A Simple Implementation of the `ls(1)` Command

To give an example of how `readdir(3)` can be used, here is a simple version of the `ls(1)` command which does the equivalent of `ls -al` on the current directory. The source code to our minimal `ls` command is shown below. For the `print_*` functions, the source code is available at the book's github site. There is a fair amount of processing of the `stat` structure, mapping between UID/GID and actual string representations and manipulation of the modification time to get it into the right format.

```
1 int
2 main() {
3     struct dirent *dir;
4     DIR            *mydir;
5     struct stat    st;
6
7     mydir = opendir(".");
8     while (1) {
```

```

9         dir = readdir(mydir);
10        if (dir == NULL) {
11            break;
12        }
13        lstat(dir->d_name, &st);
14        print_file_type(&st);
15        print_perms(&st);
16        printf(" %d ", st.st_nlink);
17        print_owner_group(&st);
18        printf("%5ld ", st.st_size);
19        print_mtime(&st);
20        printf("%s\n", dir->d_name);
21    }
22 }
```

The program is quite simple, looping through the directory making calls to `readdir(3)` (line 9) to read each directory entry. Once the last entry has been read, `NULL` will be returned. Then for each directory entry read, we print out the various fields to match the output returned by running `ls -al`.

As mentioned above, the `d_type` field of the `dirent` structure is not POSIX compliant but could be used to avoid a call to `lstat(2)`. In our example, we want more information about the file so this call is necessary. Furthermore, we need to call `lstat(2)` in place of `stat(2)` since, if this was a symbolic link, we would information about the symbolic link and not the file to which it points to.

The `print_file_type()` function simply checks the `S_IFMT` field of the `stat` structure. Here is a fragment of the source code:

```

print_file_type(struct stat *st)
{
    char c;

    switch (st->st_mode & S_IFMT) {
        case S_IFBLK:
            c = 'b';
            break;
        case S_IFCHR:
            c = 'c';
            break;
        ...
        default:
            c = '?';
            break;
    }
    printf("%c", c);
}
```

Below is output from the program running showing contents of a directory with several different file types followed by running the real `ls -l`:

```
$ ./myls
```

```
brw-r--r-- 1 root root 0 Dec 19 17:41 mydev
drwxrwxr-x 2 spate spate 4096 Dec 19 17:39 mydir
drwxrwxr-x 3 spate spate 4096 Dec 19 17:41 .
-rw-r--r-- 1 spate spate 325 Dec 19 17:39 sparse.c
prw-rw-r-- 1 spate spate 0 Dec 19 17:40 mypipe
drwxrwxr-x 4 spate spate 4096 Dec 19 19:41 ..
-rw-rwxr-x 1 spate spate 70504 Dec 19 17:39 fds
$ ls -al
total 32
drwxrwxr-x 3 spate spate 4096 Dec 19 17:41 .
drwxrwxr-x 4 spate spate 4096 Dec 19 19:41 ..
-rwrxrwxr-x 1 spate spate 70504 Dec 19 17:39 fds
brw-r--r-- 1 root root 4, 5 Dec 19 17:41 mydev
drwxrwxr-x 2 spate spate 4096 Dec 19 17:39 mydir
prw-rw-r-- 1 spate spate 0 Dec 19 17:40 mypipe
-rw-r--r-- 1 spate spate 325 Dec 19 17:39 sparse.c
```

The `ls(1)` program does a lot of formatting. When you run `ls -l`, the `"."` and `".."` entries are displayed first. The spacing around the owner, group and file size are nicely done. This implies that `ls` must make at least one pass through the list of directory entries before it starts printing anything. However, all entries are read using `readdir(3)` and then a call is made to `sort_files()` to sort / format them. You can find the source code to the `ls(1)` command here:

<https://github.com/wertarbyte/coreutils/blob/master/src/ls.c>

This simple version of `ls` doesn't display the device file major/minor and won't nicely format directory entries if owner, group and size of more characters than those shown here.

2.23.2 Other Directory Functions

One of the most useful functions, and one which could simply our version of `ls(1)` by getting an alphabetically sorted list of directory entries, is the `scandir(3)`:

```
#include <dirent.h>

int scandir(const char *restrict dirp,
            struct dirent ***restrict namelist,
            int (*filter)(const struct dirent *),
            int (*compar)(const struct dirent **,
                          const struct dirent **));
```

The directory specified by `dirp` is scanned and entries are sorted according to the function specified by `compar` which can be `alphasort` which sorts directory entries using `strcoll(3)`, or `versionsort` which sorts using `strverscmp(3)` on the strings `(*a)->d_name` and `(*b)->d_name`. As entries are being read, space is allocated by calling `malloc(3)`.

The `filter` argument can be used to select specific entries. An example, if our `myls` program was given wildcards as an argument, the `filter` function provided could

be used to match specific files. If `filter` is set to `NULL` all directory entries will be returned.

Here is a skeleton version of a program that we could have used for the `myls` command above. Note that as we display each directory entry, we free the space that was allocated by `scandir(3)`.

```
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
    struct dirent **namelist;
    int nentries, i;

    nentries = scandir(".", &namelist, NULL, alphasort);
    for (i=0 ; i<nentries ; i++) {
        printf("%s\n", namelist[i]->d_name);
        free(namelist[i]);
    }
    free(namelist);
}
```

Since we've passed `alphasort` as an argument, we get all entries sorted nicely:

```
$ ./scandir
.
..
fds
mydev
mydir
mypipe
sparse.c
```

There are other functions that allow you to browse directory trees which you can peruse at your own pleasure:

- `ftw(3)` — Walk through a directory tree and calls the specified function for each entry in the tree.
- `fts(3)` — More functions for walking a directory tree.
- `telldir(3)` — Return current location in directory stream.
- `seekdir(3)` — Set the location in the directory stream from which the next call to `readdir(2)` will start. A value returned by `telldir(3)` should be used.
- `rewinddir(3)` — Reset the position of the directory stream to the beginning of the directory.

2.23.3 Directory Entries at the System Call Level

The library functions described in the sections above all build on top of a simple system call interface defined as follows in the `getdents(2)` manpage:

```
#include <sys/syscall.h>    /* Defn of SYS_* constants */
#include <unistd.h>

long syscall(SYS_getdents, unsigned int fd,
            struct linux_dirent *dirp, unsigned int count);

#define _GNU_SOURCE.    /* See feature_test_macros(7) */
#include <dirent.h>

ssize_t getdents64(int fd, void *dirp, size_t count);
```

As the manpage says:

These are not the interfaces you are interested in. Look at readdir(3) for the POSIX-conforming C library interface. This page documents the bare kernel system call interfaces.

If you browse the glibc source code, you'll see that it uses `getdents64()` which returns one or more directory entries in the buffer pointed to by `dirp`. The size of the buffer is referenced by `count`. The number of entries read is returned and each entry returned is defined by the `linux_dirent` structure:

```
struct linux_dirent64 {
    ino64_t      d_ino;    /* 64-bit inode number */
    off64_t      d_off;   /* 64-bit offset to next struct */
    unsigned short d_reclen; /* Size of this dirent */
    unsigned char  d_type;  /* File type */
    char         d_name[]; /* Filename (null-terminated) */
};
```

Although it looks similar to `readdir(3)` it requires a lot more work on behalf of the caller and thus the very large warning in the manpage. Applications should use the functions described earlier but it's important to highlight the system call since we will be describing its implementation later in the book in section 6.13.

2.24 File Notification

File notification is a mechanism where applications can be notified of events that occur within the filesystem in response to actions such as file creation, reads, writes and deletions.

There are several applications that can make use of filesystem notification events. Examples that come to mind are desktop file managers which can update their display of a directory as changes are made to the directory, malware scanners such as ClamAV and hierarchical storage management which is discussed elsewhere in the book. Another important example is backup. When determining what files to backup, scanning a filesystem looking

for changes can be a time-consuming task. If the backup application already had a list of changed files, that scan could be avoided.

The three different file notification mechanisms that exist in Linux are as follows:

- **dnotify** – appearing in Linux in 2001, the first mechanism lacked the knowledge about how such frameworks should be used and relied on extensions to `fcntl(2)`, for example "`fcntl(fd, F_NOTIFY, mask)`". Only directories could be monitored and not regular files. Delivery of a signal indicated notification of an event but the application needs to scan the directory to see what actually changed. There were additional shortcomings.
- **inotify** – first appearing in Linux 2.6.13 in 2005, the inotify framework intended to resolve the issues with dnotify. The three system calls, namely `inotify_init(2)`, `inotify_add_watch(2)`, and `inotify_rm_watch(2)` were introduced. An initialization call to `inotify_init(2)` returns a file descriptor which is used by the other inotify system calls. Files and directories can be added to the list of file objects to be monitored and the list and events could be added or removed. The `inotify(7)` manpage lists all of the possible events. There are several advantages over dnotify. In addition to supporting regular files, the interface was simpler, there are more events and there is better information about each event received. For more details of inotify I recommend reading the lwn.net articles (search for "linux filesystem notification part").
- **fnotify** – first appeared in Linux 2.6.37 in 2011 and will be described in this section. It has a superset of the inotify functionality.

With fanotify, it's possible to monitor a mounted filesystem as a whole unlike inotify. Despite being introduced in 2011, fanotify has seen several improvements over the years with major enhancements appearing in 2019 such as support for create, move, and delete events. These events were part of a *directory events* addition at the time.

Before describing how to use fanotify, there is an example program at the following location that will be demonstrated here:



URL 7 <https://tinyurl.com/2p9dmz2n>

Note that you will likely need to replace the line that includes "linux/fanotify.h" with the following:

```
#include <sys/fanotify.h>
```

To run the program, specify a directory (this example uses "mydir") and events will be displayed as operations are performed on files within the specified directory:

```
$ sudo ./fanotify-example mydir
Started monitoring directory 'mydir'...
Received event in path '/home/spate/mydir/foo' pid=2813 (...):
```

```
FAN_OPEN
FAN_CLOSE_WRITE
Received event in path '/home/spate/mydir/foo' pid=1045 (-bash):
    FAN_OPEN
Received event in path '/home/spate/mydir/foo' pid=1045 (-bash):
    FAN MODIFY
Received event in path '/home/spate/mydir/foo' pid=1045 (-bash):
    FAN CLOSE_WRITE
```

The messages displayed are in response to the following operations:

```
$ touch mydir/foo
$ mkdir mydir/newdir
$ echo hello >> mydir/foo
```

Note that there is no events generate for the `mkdir` call. The complete set of events that can tracked can be found in the `fanotify(7)` manpage. The program as is looks for the following events:

```
static uint64_t event_mask =
(FAN_ACCESS | /* File accessed */
 FAN_MODIFY | /* File modified */
 FAN_CLOSE_WRITE | /* Writable file closed */
 FAN_CLOSE_NOWRITE | /* Unwritable file closed */
 FAN_OPEN | /* File was opened */
 FAN_ONDIR | /* We want to be reported of events in
               the directory */
 FAN_EVENT_ON_CHILD); /* We want to be reported of events in
                     files of the directory */
```

Try experimenting and seeing what you do and do not get events for. The event mask can also be changed to poll for other events as described in the `fanotify(7)` manpage.

Looking at the program to see how it works, it first starts with a call to `fanotify_init()` which will create new *fanotify device*:

```
if ((fanotify_fd = fanotify_init(FAN_CLOEXEC, O_RDONLY |
                                  O_CLOEXEC | O_LARGEFILE)) < 0)
```

and then call `fanotify_mark()` to add an *fanotify mark* using the file descriptor just received above. The directory passed to the program is shown here as `monitors[i].path`.

```
if (fanotify_mark(fanotify_fd, FAN_MARK_ADD, event_mask,
                  AT_FDCWD, monitors[i].path) < 0) {
    ...
}
```

The `event_mask` was shown above. After everything is initialized, the program loops in `main()`, calling `poll(2)` until there is an event to be processed. A call is then made to the function `event_process()` is called to display the information that is seen when the program is run.

There are also two example programs in the `fanotify(7)` manpage. Additional information can be found in `fanotify_init(2)` and `fanotify_mark(2)`.

2.25 Filesystem-level Programming Interfaces

So far, the chapter has focused on programming interfaces for access to files. Commands such as `mount(1)` and `df(1)` utilize system calls that operate on filesystems. This section will describe the most common filesystem-level operations.

2.25.1 Mounting and Unmounting Filesystems

This section jumps ahead somewhat since mounting and unmounting filesystems aren't covered until the next chapter. But since this chapter covers programming interfaces, both will be partially covered here. Only basic concepts will be described here with details coming in subsequent chapters.

The `mount(8)` and `umount(8)` commands are used for mounting and unmounting filesystems respectively. In their basic form, they are very simple. In the examples below, local and remote NFS filesystems are mounted and then the local filesystem is unmounted.

```
$ sudo mount -t spfs /dev/sda1 /mnt
$ sudo mount -t nfs 192.168.56.135:/remote-mnt /local-mnt
$ sudo umount /mnt
```

The `mount(8)` command run by itself will display all mounted filesystems.

There are two corresponding system calls, namely `mount(2)` and `umount(2)` for their command counterparts.

```
#include <sys/mount.h>

int mount(const char *source, const char *target,
          const char *filesystemtype, unsigned long mountflags,
          const void *data);

int umount(const char *target);
int umount2(const char *target, int flags);
```

For the `mount(2)` system call, some arguments match our examples above. Here are the different arguments:

- `source` – this argument refers to the place where the filesystem is located. In the examples above `source` is the device `/dev/sda1` for the local filesystem and the server:path `192.168.56.135:/remote-mnt` for the NFS filesystem.
- `target` – this is where the filesystem is mounted, for example, `/mnt` in the local filesystem example above.
- `filesystemtype` – what type of filesystem is being mounted, for example `spfs` or NFS.
- `mountflags` – a bitwise-OR of various flags that determine the behavior of the mount. By default, filesystems are mounted read/write. The `MS_RDONLY` flag specifies that the filesystem be mounted read-only. There are many flags and they will be described in more detail in section 3.13.2.

- `data` – there are times when filesystem-specific options are needed to be passed to the filesystem in the kernel during mount. For example, JFS has a `resize` mount option that requests JFS to resize the filesystem to the specified size, thus growing the filesystem. The `mount (8)` manpage describes options that are filesystem specific.

The following example repeats our local filesystem example but uses the `strace(1)` command to show the different arguments that are being passed to `mount(2)`. This is as simple as it gets.

```
$ strace mount -t spfs /dev/sda /mnt
...
mount("/dev/sda", "/mnt", "spfs", 0, NULL) = 0
...
```

The `umount(2)` system call is very simple. The only argument that is needed is the target argument specified during a call to `mount(2)`. There are three possible arguments for `umount(2)`:

- `MNT_FORCE` – a request is made to the filesystem to abort any pending requests before attempting the unmount. An example would be trying to unmount without waiting for an unresponsive or inaccessible server. Note that this option could result in data loss. This operation is not guaranteed to succeed. If, after aborting requests, there are still some processes with active references to the filesystem, the unmount will still fail. Not all filesystems support this option.
- `MNT_DETACH` – this can be seen as a halfway attempt at `MNT_FORCE`. Any new processes will be prevented from accessing the filesystem, it will be removed from the mount table (so not visible) and the unmount will take place once accessed are completed.
- `MNT_EXPIRE` – unmount a filesystem after a period of inactivity. The filesystem mount is marked as expired. If it is not currently being used, an initial call to `umount2()` with this flag fails with the error `EAGAIN`, but marks the mount as expired. The mount remains expired as long as it isn't accessed. A second call with `MNT_EXPIRE` unmounts the expired mount.
- `UMOUNT_NOFOLLOW` – don't dereference `target` if it is a symbolic link. This flag allows security problems to be avoided in set-user-ID-root programs that allow unprivileged users to unmount filesystems.

2.25.2 Getting Filesystem Statistics

There are two system calls that can be called to get information about a specific filesystem which are used by commands such as `df(1)`:

```
#include <sys/vfs.h>      /* or <sys/statfs.h> */
int statfs(const char *path, struct statfs *buf);
int fstatfs(int fd, struct statfs *buf);
```

The `statfs` structure contains information about the filesystem.

```
struct statfs {
    __fsword_t f_type;      /* Type of filesystem (see below) */
    __fsword_t f_bsize;     /* Optimal transfer block size */
    fsblkcnt_t f_blocks;   /* Total data blocks in filesystem */
    fsblkcnt_t f_bfree;    /* Free blocks in filesystem */
    fsblkcnt_t f_bavail;   /* Free blocks available to
                           unprivileged user */
    fsfilcnt_t f_files;    /* Total inodes in filesystem */
    fsfilcnt_t f_ffree;    /* Free inodes in filesystem */
    fsid_t     f_fsid;     /* Filesystem ID */
    __fsword_t f_namelen;  /* Maximum length of filenames */
    __fsword_t f_frsize;   /* Fragment size (since Linux 2.6) */
    __fsword_t f_flags;    /* Mount flags of filesystem
};

```

The `df(1)` command is implemented to use the `statfs(2)` system call. You can see how the information displayed by `df(1)` corresponds to the `stats` structure above. An example is shown here:

```
# df /mnt
Filesystem      1K-blocks  Used Available Use% Mounted on
/dev/sda          1520     280      1240  19% /mnt
```

For information on how this call is implemented by the filesystem see section 7.11.

There are two additional library calls that utilize `statfs(2)` namely `statvfs(3)` and `fstatvfs(3)`.

2.25.3 Filesystem Sync

Throughout the book, the use of caches in the kernel for many different things will be discussed from the buffer cache to the page cache to multiple inode caches. At some point that data needs to be written to disk. Historically, this was what the `sync(1)` command and `sync(2)` system call were for.

There is some logic to the old UNIX saying that you should sync three times before shutting down the system. I do actually recall one of the shutdown scripts on older versions of SCO UNIX (SVR3 UNIX variant at the time) doing something like this:

```
sync;sync;sync
haltsys
```

Perhaps there were only two calls to sync but this is how I remember it. There is some talk about whether this is folklore or was actually necessary. After all, a `sync(1)` call doesn't guarantee to have flushed data to disk, only to schedule it. The bigger concern over 30 years ago was the lack of journaling filesystems and the need to run `fsck(1)` if a system wasn't cleanly shutdown. But no modern operating system would reboot or shutdown without having flushed modified data to disk.

Whatever the history, the system calls do actually do something which will be discussed in section 6.22. For now, here are the system calls:

```
#include <unistd.h>

void sync(void);
int syncfs(int fd);
```

and the description in the manpage says:

`sync(1)` causes all pending modifications to filesystem metadata and cached file data to be written to the underlying filesystems. `syncfs(1)` is similar to `sync()`, but synchronizes just the filesystem containing file data that is referenced to by the open file descriptor `fd`.

2.25.4 Changing Filesystem Properties

For a filesystem that is already mounted, the `mount_setattr(2)` system call changes can be invoked to change its mount properties.

```
#include <linux/fcntl.h> /* Defn of AT_* constants */
#include <linux/mount.h> /* Defn of MOUNT_ATTR_* constants */
#include <sys/syscall.h> /* Defn of SYS_* constants */
#include <unistd.h>

int syscall(SYS_mount_setattr, int dirfd,
           const char *pathname, unsigned int flags,
           struct mount_attr *attr, size_t size);
```

There is no glibc implementation for `mount_setattr(2)` therefore callers must use the generic `syscall(2)`.

2.26 Conclusion

Although it may seem like overkill to dedicate so much time to describing user-level programming interfaces in a book that covers the kernel implementation of filesystems and individual filesystem implementation, much of this implementation is there to support the 40+ system calls that support access to files and filesystems. In particular, the kernel code can be confusing without knowledge of the main interfaces and the large array of flags that accompany file and filesystem access.

Whole books have been written to describe Linux programming and many of them will describe these functions presented here in much greater detail. The goal of this chapter was to introduce you to these functions to give you the knowledge necessary before embarking on analyzing the Linux kernel source code.

For readers familiar with the programming interfaces, the material here is for reference. For those who have not looked at UNIX or Linux libraries and system calls for some time, Linux has not stood still and purely followed a standards model. Many standards-based library / system calls have been extended for performance gains or to introduce new functionality so it's worth browsing to see what's new.

Chapter 3

Filesystems

Everyone is familiar with files and directories (folders). A filesystem is a collection of files that supports different operations such as file and directory creation, reading, writing and deleting files and so on. There are many other types of operations which may or may not be supported depending on the filesystem type. In fact, the BFS filesystem does not support creating directories. It was a special-purpose filesystem used as part of the bootstrap process for SVR4 UNIX.

Filesystems can be backed by physical storage, they can talk over the network to other servers or they can be virtual or pseudo filesystems where they present what looks like a hierarchy of files and directories but there is no underlying storage.

Much of the previous chapter provided information on programming interfaces related to accessing files with a few relevant interfaces related to filesystems. This chapter explores filesystem-related topics starting with the layout of the Linux file hierarchy, a brief discussion on how disks are partitioned and the main types of Linux filesystems. It then describes the difference between disk-based, network-based and pseudo filesystems giving examples of each and how they're used. Later sections will cover backup and restore, quotas and containers. **XXX—need more here**

3.1 The Linux File Hierarchy

The first filesystem mounted by the Linux kernel following bootstrap is called the root filesystem. The layout of files and directories inside the root filesystem is described by both the `file-hierarchy(7)` and `hier(7)` manpages. A visual view of the filesystem hierarchy is shown in figure 3.1. There are a lot of directories described in `hier(7)` which would be next to impossible to squeeze into an OmniGraffle diagram so I have only highlighted the main directories and particularly those related to filesystems. **XXX—need to show modules, /proc/filesystems etc.**

At the time of writing, a freshly installed Ubuntu 23.04 VM has 188,359 files in 24,480 directories. That's a lot of stuff! **XXX—actually measure it fresh. I have lots of junk in there now**

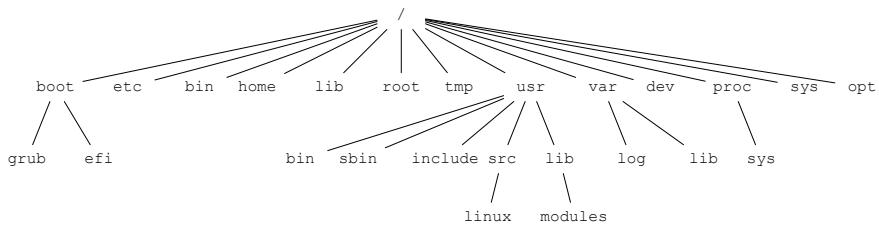


Figure 3.1: The main directories in the root filesystem

XXX—need to decide much of the tree I should describe.

- `/bin` – a symbolic link to `/usr/bin` which contains almost 1500 general purpose commands such as `ls(1)` and `mkdir(1)`.
- `/etc` – system-wide configuration files and system databases. It used to be the dumping ground for everything else thus the name "etc" (etcetera).
- `/boot` – the Linux kernel image, `initrd` (initial RAM disk), GRUB configuration and anything else that is needed to bootstrap the system.
- `/dev` – all of the devices in the system. More specifically, these are special files that are used to access Linux devices attached to the system.
- `/home` – home directories for users.
- `/lib` – a symbolic link to `/usr/lib`. This directory shared libraries.
- `/lib/modules` – all of the compiled kernel modules are located under this directory. The filesystem modules are located under `/lib/modules/<kernel version>/kernel/fs`.
- `/root` – the home directory for `root`. It's separated from `/home` since user data is often kept on a separate filesystem which is mounted later in the boot process. At various times it may be necessary to boot to single user mode and for `root` to be able to login and diagnose issues. This occurs before general filesystems are mounted.
- `/usr/src/linux` – the Linux headers are here by default but not the source code. Installing the kernel source code can be done using Linux distribution-specific commands or you can download a kernel of your choice from www.kernel.org. To build a Linux kernel you don't need to put the source code here. You can build and install the kernel as a general user. This will be discussed more in chapter 7.
- `/proc` – there are many interesting bits of information in `/proc` that are related to file access and filesystems. We'll be exploring these in section 3.9.1.

- `/sbin` – a symbolic link to `/usr/sbin`, this directory contains programs used to boot and administer the system. Programs here are not typically used by general users.
- `/tmp` – used for storing temporary files. Once upon a time, these files were cleared each time the system was rebooted. Now it is cleared after a predefined number of days which differs per Linux distribution. Just never assume that any files you place here will still be there after the next reboot. The `/var/tmp` directory is similar but has a longer retention period.
- `/var/log` – holds system log files.
- XXX – more? check when reading back again

3.2 How Simple it Used to be

In early versions of UNIX as well as Linux, a disk partitioning was very simple. There was a bootstrap block and the UNIX filesystem. To go back and see how UNIX handled the boot process back in 1975, you can get hold of John Lyons' "A Commentary on the 6th Edition UNIX Operating System" online. The source code is available on github at:



8 <https://tinyurl.com/u36a2a3a>

In the top-level `README.md` you will find a link to Lions' book. I remember when I first got hold of John Lions' book and seeing all of the UNIX source code at just over 11,000 lines of code. At the time I was working on getting the `truss(1)` utility (UNIX equivalent of `strace(1)`) running on the SVR4 subsystem on the Chorus microkernel at the time. It was about the same amount of code. Gulp! How things were much simpler back then.

Bootstrap was very simple. The initial boot loader was located in firmware. Its goal was to load block 0 which contained a larger boot program. This boot program looked for and loaded `/unix` to which it passes control. The superblock of the root filesystem was always stored in block 1. **did they really not have mountable filesystems? I thought I saw mount in there somewhere? Check**

Additional filesystems could be mounted - **XXX—walk through the code and see how it worked**

3.3 Filesystems, Disks and Partitions

Things to cover:

- physical vs logical block sizes
- maximum file and filesystem sizes and how fs block size affects this

- block allocation vs extents
- perhaps show old single, double and triple extents to show how things have changed
- full fsck vs journaling
- partition highlights before describing in more details in next sections

Filesystem on-disk structures have become considerably more complicated in the last thirty years compared to early days of UNIX and Linux.

3.3.1 Filesystem On-disk Structure

Disks are partitioned using different partitioning schemes which will be described soon but generally speaking, there can be multiple partitions within a single disk and each partition can contain a filesystem that can be mounted separately onto a directory in the Linux file hierarchy. Figure 3.2 shows the basic components of a traditional filesystem.

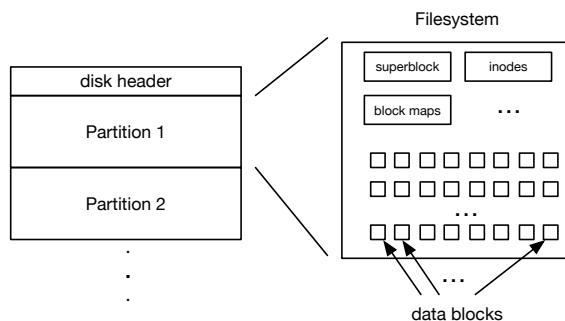


Figure 3.2: A disk partition with general filesystem disk structures

Todays filesystems are much more complicated and will be described later in the book. However, this is the minimal set of data structures needed to create a fully functioning filesystem.

- **superblock** – the first structure that will be read from disk when the filesystem is mounted. It contains enough information to locate the other structures needed to access files, allocate data blocks and so on. There may be multiple copies of the superblock to guard against disk errors.
- **inode** – a structure on disk that represents a file regardless of the file type. It contains file meta-data which describes the file including type, size, owner and the location of the file's *data blocks*. There may be a preallocated inode list or inodes may be allocated dynamically allowing for a larger number of files.
- **data blocks** – contain actual data whether it be regular file data, directory entries or symbolic links.

- **block map** – a data structure that indicates which blocks have been allocated. Data blocks are of fixed size and the size is chosen when the filesystem is created.

The filesystem on-disk structure has evolved considerably over time primarily to address resilience, performance, file and filesystem size but also to incorporate new features such as snapshots, file activity change logs and journaling capabilities.

3.3.2 Referencing Data Blocks From The Inode

To help explain file and filesystem sizes and how they differ depending on the filesystem block size, consider figure 3.3 which shows the layout of a file in the ext2 filesystem.

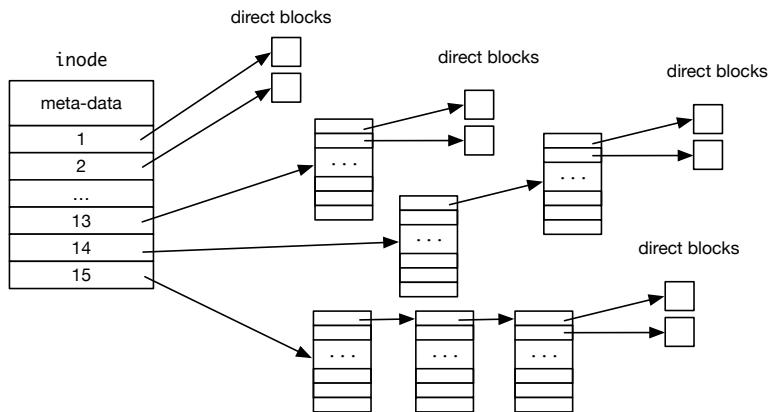


Figure 3.3: The UFS on-disk structure for a file

Inside the inode is meta-data about the file in addition to fifteen pointers that reference the file's data blocks either directly or indirectly. There are:

- twelve direct pointers that point to blocks of actual data for the file.
- one single-indirect pointer that points to a block of pointers that then point to actual data blocks.
- one double-indirect pointer that points to a block of pointers that point to other blocks of pointers that finally point to blocks of data for the file.
- one triple-indirect pointer that points to a block of pointers that point to other blocks of pointers that point to other blocks of pointers that finally point to blocks of the file's data.

The size of each block of data depends on the filesystem block size. Let's take ext4 as an example. It has four different filesystem block sizes that are specified when the filesystem is made. This means that each data block can range from 1 KB to 8 KB resulting in very

block size	maximum filesystem size	maximum file size
1 KiB	16 GB	4 TB
2 KiB	256 GB	8 TB
4 KiB	2 TB	16 TB
8 KiB	2 TB	32 TB

different file sizes depending on the block size chosen as show in table XXX—reverse direction and color

XXX

3.3.3 Full Filesystem Check vs Journaling

When a filesystem is mounted for read/write access, it is generally marked *dirty* indicating that changes are occurring. This has traditionally been done by setting a flag in the superblock. When the filesystem is unmounted it is marked *clean*. If the system crashes between these two events, the filesystem could be in an inconsistent state. For example, when creating a file, there are several structures on disk that need to be modified. If the system crashes in the middle of these operations, the filesystem needs to undo those operations that it performed to make the filesystem structurally intact again. To bring the filesystem back to being structurally intact, a `fsck` (filesystem check) is needed. This is performed prior to the filesystem being mounted again, typically as part of the boot process. This is a process that can take a very long time typically dependent on the number of files in the file system. Many hours there's not unlikely.

This is where *filesystem journaling* comes into play. First introduced in the VERITAS VxFS filesystem and now available in several (name them???) Linux filesystems, journaling involves writing upcoming structural changes to an *intent log* before changing the actual on-disk structures themselves. If the system crashes, log replay is performed which takes a very short amount of time. Journaling will be described in more detail in section TBD.

3.3.4 Disk Partitioning

This is another topic to which a whole chapter could be dedicated but it really beyond the scope of this book. But partitioning affects many things from bootstrap to choosing how to layout filesystem. Whether using disks directly or using logical volume managers to manage the disks for you, it's important to spend some time explaining how disks are partitioned.

The two primary gains of GPT over MBR are:

1. Large partitions, more than 2 TB
2. Unlimited number of primary partitions

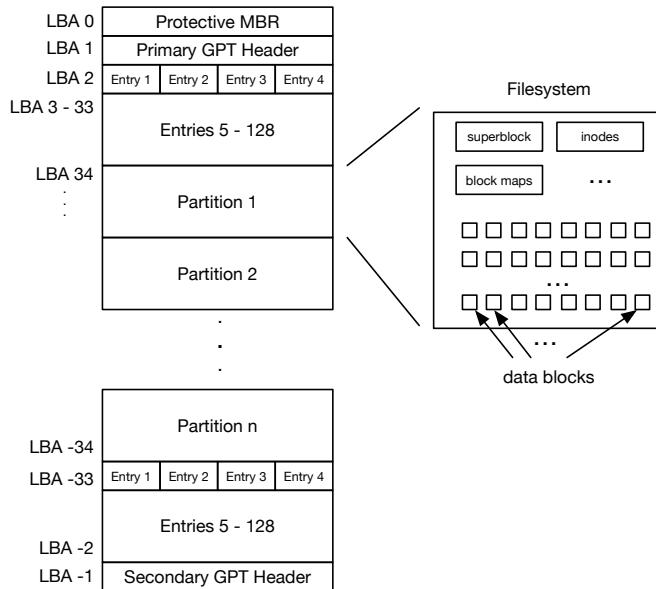


Figure 3.4: GPT partitioned disk and filesystem within a partition

3.3.5 The GPT Disk Format

This section highlights the GPT format and shows how filesystems are stored within GPT partitions. A GPT-partitioned disk is shown in figure 3.4 with a hypothetical filesystem shown on partition 1.

For each partition that contains a filesystem, the filesystem will *manage* the layout of the partition. There will be filesystem meta-data which includes information about the filesystem itself (superblock), structures for allocated files (usually inode structures) and other information such as tracking data blocks which have been allocated. The meta-data only takes up a relatively small amount of space and the rest of the partition is open for storing file data. The exception to this is the RAM-based filesystems that have been used in Linux since early days (**XXX—day 1?**) which will be discussed in section 3.9.2.

Some points to note about the figure:

- LBA stands for Logical Block Addressing. Each block is 512 bytes.
 - There are two copies of the GPT header, one at the start of the disk and one in the last block (LBA - 1).
 - The partition table starts at LBA 2 and there are up to 128 partitions. Each partition table entry is 128 bytes.
 - *Usable* partitions start with partition 1 and LBA 34.

blah

3.3.6 Logical Volumes Management

For developmental or experimental purposes, you will likely only need one or two disk. On my Ubuntu VMs, I have one disk for Ubuntu itself and another small disk which I partition and use for storing filesystems that I'm developing or playing with. You can also create loopback devices and place filesystems on there.

In the real world, the amount of data that organizations need to store well exceeds the size of individual disks so managing hundreds, thousands or tens of thousands of individual disks makes no sense. This is where *logical volume management* comes into play. Of course, most organizations will unlikely deal with disks themselves but rather get their storage management solutions from one of the many storage vendors on the market today. Each of these vendors will package many disks inside the storage arrays using logical volume manager technology for which some of the technology will be hidden from the customer.

Before working on the VERITAS filesystem VxFS, my first introduction to VERITAS came when I ported their volume manager VxVM to SVR4 UNIX running on the Chorus microkernel back in the early 1990s. VxVM was ported to many operating systems throughout the 90s, became a standard part of Microsoft Windows several during that time and my colleagues and friends ported it to Linux at the same time my team was porting VxFS to Linux.

3.3.7 Advantages of Logical Volume Management

There are many advantages of logical volume management (LVM) and perhaps top of the list is ease of management. Rather than partitioning large numbers of disks and having to manage them yourself, just give the disks to the volume manager and create new volumes as needed and not have them restricted by the size of the disk partitions.

Some have argued that LVM just adds another layer to the storage stack and has some performance overhead. However, the actual performance reduction is very minor and offset by the advantages that LVM offers. But as volumes are created and overlap on the underlying shared storage, fragmentation can become an issue leading to potential performance issues. These are all issues that the LVM vendors such as VERITAS addressed in their products.

Since this section is just highlighting the area of logical volume management, here are some of the major advantages:

- Ease of management compared to handling large numbers of disks / partitions.
- Avoid the limits imposed by physical disk sizes / partitions.
- Increase or decrease the size of volumes and therefore the size of the filesystems that reside on them. For example, when extending the size of a filesystem, the operation can be done in five steps using Linux LVM. Of course, if space is already available in the existing volume group, the first two steps can be omitted.
 1. Add new storage
 2. Create a new physical volume

- 3. Extend the volume group to include the new physical volume
 - 4. Extend the Logical Volume
 - 5. Extend the filesystem
- Performance gains through the use of *striping*, where data is in a logical volume is spread across multiple devices and performance is increased since reads/writes happen in parallel.
 - Redundancy through mirroring, where the same data is copied to two or more physical devices. If one of the devices dies or goes offline for some reason, the data on the other devices can still be accessed.
 - Other RAID (Redundant Array of Inexpensive Disks) levels including RAID4 and RAID 5 which provide parity, a common way of detecting errors in a storage system.
 - Snapshots for backup purposes independent of the filesystem type. Not all filesystems support snapshot mechanisms.

Generally speaking, logical volume management allows users to build enterprise-level capabilities with inexpensive disks. Most modern storage systems have all of these capabilities built-in but come at additional cost.

3.3.8 Linux LVM

Heinz Mauelshagen wrote the original LVM code back in 1998, taking its primary design guidelines from the HP-UX's volume manager, which was actually the VERITAS volume manager VxVM. I remember going to a small Linux conference in Frankfurt, Germany around that time that Heinz hosted. Version 1 of LVM was introduced into the 2.4 kernel series. LVM version 2 (or just LVM2) became available in kernel versions 2.6.9 and above.

Around this time, logical volume management was extremely popular with many organizations managing large numbers of disks which is problematic for many reasons. This popularity somewhat declined in subsequent years with newer storage arrays that had all of this functionality built-in. But in those cases, the logical volume management component simply shifted from the server or VM running Linux to the storage array itself. Take most IaaS (Infrastructure As A Service) solutions in the cloud. All of these capabilities are built in to the storage fabric and Linux just sees simple partitions. The same is true of most hypervisor solutions for which thin provisioning is an important capability but also hidden from the Linux operating system. Thin provisioning is a technique where storage blocks are only allocated as needed. It may look like you have 40 GB drive but if there is only 20 GB of data in used, only 20 GB of data is allocated from the underlying fabric.

Use of LVM still continues to this day. Some Linux distributions use LVM to configure the root device. For example, the Ubuntu VMs I am are partitioned as follows:

```
$ df -h
Filesystem           Size  Used Avail Use% Mounted on
tmpfs                 392M  1.2M  391M   1% /run
/dev/mapper/ubuntu--vg--...v  30G   21G  7.5G  74% /
```

tmpfs	2.0G	0	2.0G	0%	/dev/shm
tmpfs	5.0M	0	5.0M	0%	/run/lock
/dev/sda2	2.0G	269M	1.6G	15%	/boot
/dev/sda1	1.1G	6.1M	1.1G	1%	/boot/efi
tmpfs	392M	4.0K	392M	1%	/run/user/1000

XXX – at some point, install other distros and see if they use LVM by default

3.3.9 LVM Concepts

LVM functions by layering abstractions on top of physical storage devices. The basic layers that LVM uses, starting with the most primitive, are shown in figure 3.5:

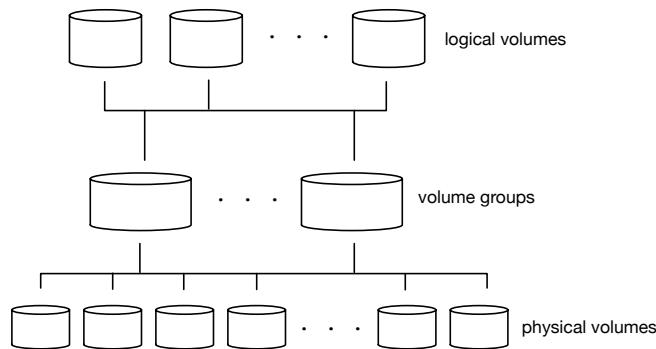


Figure 3.5: LVM Architecture

- **Physical Volumes (PVs)** – LVM uses physical block devices such as hard disk drives, solid-state drives or partitions. An LVM label is written near the start of the disk to correctly identify the device. This is particularly important when physical devices are removed/added since device names can change. The LVM utility prefix for physical volumes is `pv`.
- **Volume Groups (VGs)** – LVM combines physical volumes into storage pools called volume groups. You can think of them as very large disks. This concept was introduced in volume managers many years ago to allow for filesystems or raw volumes to be much bigger than the largest available disks at the time. The same is still true today. The LVM utility prefix for volume groups is `vg`.
- **Logical Volumes (LVs)** – A volume group can be sliced up into any number of logical volumes. Logical volumes are functionally equivalent to partitions on a physical disk in that you create a filesystem on top of a logical volume. However, they have much more flexibility. Logical volumes are the primary component that users and applications will interact with. The LVM utility prefix for logical volumes is `lg`.

Using the Ubuntu disk layout shown above, the following commands show the different LVM layers. First of all, here is the single physical volume which is constructed from a single disks /dev/sda3:

```
$ sudo pvdisplay
--- Physical volume ---
PV Name          /dev/sda3
VG Name          ubuntu-vg
PV Size          <60.95 GiB / not usable 3.00 MiB
Allocatable      yes
PE Size          4.00 MiB
Total PE         15602
Free PE          7801
Allocated PE     7801
PV UUID          wpN3k3-OjXr-8FKN-CG0I-kf0B-h1K6-h0zXh9
```

Next is the single volume group ubuntu-vg:

```
$ sudo vgdisplay
--- Volume group ---
VG Name          ubuntu-vg
System ID        lvm2
Format           lvm2
Metadata Areas   1
Metadata Sequence No 2
VG Access        read/write
VG Status        resizable
MAX LV           0
Cur LV           1
Open LV          1
Max PV           0
Cur PV           1
Act PV           1
VG Size          <60.95 GiB
PE Size          4.00 MiB
Total PE         15602
Alloc PE / Size  7801 / 30.47 GiB
Free PE / Size   7801 / 30.47 GiB
VG UUID          CdWVPi-TAwA-7jtZ-iPFb-xq6k-ybeO-NKJNFY
```

On top of this volume group there is a logical volume called ubuntu-lv:

```
$ sudo lvdisplay
[sudo] password for spate:
--- Logical volume ---
LV Path          /dev/ubuntu-vg/ubuntu-lv
LV Name          ubuntu-lv
VG Name          ubuntu-vg
LV UUID          NXbsIa-999P-8QeO-aHJh-EdVU-jY2C-zOruGY
LV Write Access  read/write
LV Creation host, time ubuntu-server, 2023-04-19 09:13:27 +0000
```

```
LV Status           available
# open              1
LV Size             30.47 GiB
Current LE          7801
Segments            1
Allocation          inherit
Read ahead sectors  auto
- currently set to  256
Block device        253:0
```

This is a very simple setup. If you want to play with LVM, you will need lots of disks and/or partitions. If you're operating in a virtualized world, that's very easy to do. You don't need to have large disks just to experiment.

3.4 ZFS – Builtin Volume Management

The Z File System (ZFS) was created inside Sun Microsystems in 2001 by Bill Moore, Matthew Ahrens and Jeff Bonwick. It was designed to be the next generation file system for Sun Microsystems' OpenSolaris following years of battles between Sun's UFS and the VERITAS filesystem VxFS. VxFS was much more feature rich and most of VERITAS' revenue came from Solaris. During this time Sun kept enhancing UFS mostly in the area of improving performance. This would be followed by an intense period at VERITAS to increase VxFS performance further.

In 2008, ZFS was ported to FreeBSD and we used it as part of the base OS at my second startup High Cloud Security. During the same year a port ZFS to Linux was initiated but has suffered due to the different licensing terms. ZFS is licensed under the Common Development and Distribution License (CDDL), which is incompatible with the GNU General Public License (GPL). Because of this, it cannot be included in the Linux kernel. To work around this problem ZFS is offered separately by some Linux distributions.

ZFS is a different filesystem altogether in that it includes builtin logical volume management.

Features include: **XXX – need to look at this list, add more and get info from different sites**

- Inline Data Compression – xxx
- Send/Receive snapshots – a *snapshot* of a file system can be taken and this snapshot image can be sent to a different server. This allows data from the original filesystem to be replicated for the purpose of backing up data or enabling data migration to the cloud.
- Inline Data Deduplication – xxx
- RAID-Z and Mirroring – xx
- Hierarchical checksumming of all data and metadata – xxx
- Automatic rollback of recent changes to the file system and data – xxx

- Native handling of tiered storage and caching devices – xxx
- Data integrity – xxx

Should you look at ZFS? The fact that it's not included directly in Linux distributions and may therefore have support issues, ZFS is a very feature-rich filesystem. Regardless of commercial use, it's worthy of study.

3.5 Filesystem Types Supported by Linux

At the time of writing there are over 80 different filesystems supported by Linux. How many of them are actually used?

It might be good to try and categorize them???

Wkipedia has a good page that i should cover - https://en.wikipedia.org/List_of_file_systems

- Disk file systems
- File systems with built-in fault-tolerance
- File systems optimized for flash memory, solid state media
- Record-oriented file systems
- Shared-disk file systems
- Distributed file systems
- Distributed fault-tolerant file systems
- Distributed parallel file systems
- Distributed parallel fault-tolerant file systems
- Peer-to-peer file systems
- Special-purpose file systems
- Pseudo file systems
- Encrypted file systems — this list excludes commercial solutions such as SecFS from Thales (formerly Vormetric) and their FUSE-based encryption solution.

In this book, we'll be covering pseudo filesystems as well as disk-based and FUSE-based filesystems. We'll also cover encrypted filesystems as part of looking at implementing a FUSE-based filesystem.

3.6 How Many Filesystems are Actually Used?

Although there are over 80 filesystems in the Linux kernel source and many FUSE and other filesystems elsewhere on the web, how many are actually used? Let's start with the root filesystem for the major Linux distributions today:

- Ubuntu / Debian — ext4
- Fedora — btrfs
- CentOS — XFS
- RHEL — XFS - was ext4 up to RHEL 6.0
- OpenSuSE — btrfs
- SLES — XFS
- Android — ext4 or one of the flash-based filesystems such as YAFFS2.

Generally speaking, either `ext4` or `btrfs` are chosen as the default root filesystem apart from the server versions of Linux which use XFS. There are many opinions about whether `btrfs` will replace `ext4` in desktop or general non-server environments. It certainly has a lot more features but stability and performance are raised as concerns (at the time of writing). Generally speaking, I would advise to go with the default filesystem for a specific distribution unless you have specific needs or want to experiment with a different filesystem. The default filesystem will be the one most tested in the largest number of environments and situations. Since Ubuntu is the most widely used Linux distribution with the exception of mobile devices and Android devices either use `ext4` or have been migrating towards `ext4`, it could be argued that `ext4` is the most widely used filesystem on Linux devices today.

On a default Ubuntu VM I have the following on the VMs I'm using when writing this book:

```
$ cat /proc/filesystems
nodev    sysfs
nodev    tmpfs
...
ext3
ext2
ext4
...
$ cat /proc/filesystems | wc -l
33
$ cat /proc/filesystems | grep nodev | wc -l
26
$ mount | wc -l
32
```

So interestingly enough, out of the 33 filesystem modules currently loaded, 26 of them are pseudo filesystems. By default, ext4 is the root filesystem on Ubuntu. Furthermore, running `mount(1)`, there are 32 mounted filesystems. Here are the 7 that are displayed by `df(1)`:

```
$ df -T
Filesystem  Type 1K-blocks   Used Available Use% Mounted on
tmpfs      tmpfs  400608     1340   399268   1% /run
/dev/map... ext4  39396672 1408380  25954836  31% /
tmpfs      tmpfs  2003028     0    2003028   0% /dev/shm
tmpfs      tmpfs   5120       0     5120   0% /run/lock
/dev/vda2   ext4  1992552   505692  1365620  28% /boot
/dev/vda1   vfat  1098628   5220  1093408   1% /boot/efi
tmpfs      tmpfs  400604      4    400600   1% /run/user/1000
```

There are 3 FUSE filesystems that are registered which are not installed by default. They're available on my VM as I had been developing FUSE-based filesystems at the time of writing.

If you want the default filesystems that operating systems have used going back to 1968, the Wikipedia page (https://en.wikipedia.org/wiki/List_of_default_file_systems) provides an insight. It's not a very complete list as it excludes the various UNIX versions over the years as well as mainframe operating systems. For an historical perspective on Linux variants, it's more interesting.

Another way to view mounted filesystems is to call the `findmnt(8)` command, the output of which is shown in figure 3.6.

You can also `cat /proc/mounts` and get the same information but in a less pretty form.

3.7 Comparing the Main Filesystem Types

Add some info about number of LOC for smallest, largest, most common etc

Filesystem	Lines Of Code (LOC)
ext2	9,672
ext4	62,677
xfs	70,520
btrfs	146,306

Table 3.1: Popular filesystems and their LOC in the kernel

3.7.1 The ext2/3/4 Filesystems

XXX—still some plagiarism particularly around bullet points

The first Linux filesystem mirrored the filesystem structure from Andrew Tanenbaum's operating system MINIX. This had a number of limitations such as 14 character filenames and a maximum filesystem size of 64 MB. This filesystem was soon replaced with ext,

```
$ findmnt
TARGET          SOURCE        FSTYPE   OPTIONS
/               /dev/mapper/ubuntu--vg-ubuntu--lv
                ext4      rw,relatime
/sys             sysfs       sysfs    rw,nosuid,nodev,noexec,r ...
├/sys/kernel/security securityfs  security  rw,nosuid,nodev,noexec,r ...
├/sys/fs/cgroup  cgroup2    cgroup2  rw,nosuid,nodev,noexec,r ...
├/sys/fs/pstore  pstore     pstore   rw,nosuid,nodev,noexec,r ...
├/sys/firmware/efi/efivars efivarsfs  efivarsf rw,nosuid,nodev,noexec,r ...
├/sys/fs/bpf     bpf        bpf      rw,nosuid,nodev,noexec,r ...
├/sys/kernel/debug debugfs    debugfs  rw,nosuid,nodev,noexec,r ...
├/sys/kernel/tracing tracefs    tracefs  rw,nosuid,nodev,noexec,r ...
├/sys/fs/fuse/connections fusectl   fusectl  rw,nosuid,nodev,noexec,r ...
└/sys/kernel/config configfs   configf  rw,nosuid,nodev,noexec,r ...
/proc            proc        proc     rw,nosuid,nodev,noexec,r ...
└/proc/sys/fs/binfmt_misc systemd-1 binfmt_misc  autofs   rw,relatime,fd=29,pgrp=1 ...
/dev             udev       dev     rw,nosuid,relatime,size= ...
├/dev/pts         devpts     devpts   rw,nosuid,noexec,relatim ...
├/dev/shm         tmpfs      tmpfs   rw,nosuid,inode64 ...
├/dev/hugepages  hugetlbfs  hugetlb  rw,relatime,pagesize=2M ...
└/dev/mqueue      mqueue     mqueue   rw,nosuid,nodev,noexec,r ...
/run             tmpfs       tmpfs   rw,nosuid,nodev,noexec,r ...
└/run/lock        tmpfs      tmpfs   rw,nosuid,nodev,noexec,r ...
└/run/credentials/systemd-sysusers.service ...
/run/snapd/ns     ramfs      ramfs   ro,nosuid,nodev,noexec,r ...
└/run/snapd/ns/lxd.mnt nsfs[mnt:[4026532420]] ...
                nsfs      rw ...
└/run/user/1000   tmpfs      tmpfs   rw,nosuid,nodev,relatime ...
-/snap/lxd/23687  /dev/loop0  squashf  ro,nodev,relatime,errors ...
-/snap/core20/1740 /dev/loop2  squashf  ro,nodev,relatime,errors ...
└/boot           /dev/vda2  ext4     rw,relatime ...
└/boot/efi        /dev/vda1  vfat     rw,relatime,fmask=0022,d ...
-/snap/lxd/23996  /dev/loop5  squashf  ro,nodev,relatime,errors ...
-/snap/snapd/17885 /dev/loop3  squashf  ro,nodev,relatime,errors ...
└/snap/core20/1699 /dev/loop4  squashf  ro,nodev,relatime,errors ...
```

Figure 3.6: The `findmnt (8)` command displays all mounted filesystems

the extended filesystem, which was added to Linux kernel version 0.96c in April 1992. It could handle file systems up to 2 GB and file names of up to 255 characters. But it had its own problems in that there was no support for separate timestamps for file access, inode modification, and data modification (atime, mtime and ctime).

To continue to make improvements, work on two new filesystems started in early 1993 for Linux kernel 0.99. These filesystems were xiafs developed by Frank Xia and the second extended file system (ext2) developed by Rémy Card. Xiafs was still modeled around the MINIX filesystem while ext2 got its inspiration from the BSD filesystem UFS. Xiafs was less powerful and had less functionality than ext2 so over time, it was phased out. You won't find the source code in the Linux kernel any more while ext2 is still there (`fs/ext2`).

The main features of ext2 or points to highlight are:

- Block sizes of 1, 2, 4 and 8 KB.
- A maximum individual file size can be from 16 GB to 2 TB depending on the filesystem block size.
- A maximum file system size ranges from 4 TB to 32 TB depending on block size.
- Compression and decompression of files with the `e2compr` patch.
- Since ext2 does not have journaling capabilities, if the system crashes, a full `fsck` needs to be run. **TBC**
- On flash drives and USB drives, ext2 is recommended to reduce the overhead of journaling. **TBC**

Ext3, or the third extended file system was introduced by Stephen Tweedie in 2001. It was merged into kernel version 2.4.15 in XXX. It had the following features:

- Full filesystem journaling reducing the need for a full `fsck` if the system crashes.
- Journaling has a dedicated area in the file system, where all the changes are tracked. When the system crashes, the
- possibility of file system corruption is less because of journaling.
- Maximum file size of 16 GB to 2 TB depending on block size.
- Filesystem between 2 TB to 32 TB depending on block size.
- You can convert a ext2 file system to ext3 file system directly (without backup/restore).

There are three types of journaling available in ext3 file system.

- Journal – Metadata and content are saved in the journal.

- Ordered – Only metadata is saved in the journal. Metadata are journaled only after writing the content to disk. This is the default.
- Writeback – Only metadata is saved in the journal. Metadata might be journaled either before or after the content is written to the disk.

The ext3 filesystem source code is no longer in the Linux kernel.

In June of 2006, Ted Ts'o, who was one of the principal developers of ext3, announced ext4, the fourth extended file system. The stable version was merged in the Linux kernel version 2.6.28 in October of 2008. He also stated that although ext4 has improved features and is much faster than ext3, "*it is not a major advance, it uses old technology, and is a stop-gap*". His belief is that btrfs is the better direction, because "*it offers improvements in scalability, reliability, and ease of management*".

Ext4 has the following enhancements over ext3:

- Supports huge individual file size and overall file system size.
- Maximum file size can be from 16 GB to 16 TB depending on the block size.
- Maximum file system size of 1 EB (exabyte). Note that 1 EB = 1024 PB (petabytes). 1 PB = 1024 TB (terabytes).
- Directories have a maximum of 64,000 subdirectories (this was increased from 32,000 in ext3)
- Several other new features are introduced in ext4: multi-block allocation, delayed allocation, journal checksum, fast fsck, etc. All you need to know is that these new features have improved the performance and reliability of the filesystem when compared to ext3.
- In ext4, you also have the option of turning the journaling feature “off”.

XXX - what to write here depends on volume 1/2 vs a single book.

3.7.2 The XFS Filesystems

XXX

Around the time we started the port of the VERITAS filesystem to Linux (late 1999), SGI were about to start porting their filesystem XFS. Soon after, both companies put out a press statement stating that the two companies had a joint project to bring a new, journaling filesystem to Linux. Since I was leading the effort at VERITAS I was surprised to see this news since I knew nothing of it. We quickly arranged a meeting between the two engineering teams (we were less than a mile apart) and all surprised that none of us ever knew anything about the press release. A quick marketing effort for some reason that none of us ever understood but we did stay in touch and shared our own experiences of porting what were likely, the two most feature rich commercial filesystems of the day.

- Journaling –

- Allocation groups –
- Extent-based allocation –
- Striped allocation –
- A wide variety of block sizes – from 512 bytes and 64 KB XXX
- Delayed allocation –
- Extended attributes - multiple data streams for files - hmm!
- Snapshots –
- On-line filesystem grow –

XXX

3.7.3 The btrfs Filesystems

Btrfs development began at Oracle in 2007. Btrfs is the next-generation general-purpose Linux file system that offers unique features like advanced integrated device management, scalability, and reliability. It is licensed under the GPL and open for contribution from anyone. There are different ways to pronounce the filesystem, including "Better FS", "B-tree FS" and "Butter FS". Or just spell it out letter by letter. It was merged into the mainline Linux kernel in 2009 and first appeared in the Linux 2.6.29 release.

Goals

to let [Linux] scale for the storage that will be available. Scaling is not just about addressing the storage but also means being able to administer and to manage it with a clean interface that lets people see what's being used and makes it more reliable.

Although btrfs was included as a "technology preview" in Red Hat Enterprise Linux versions 6 and 7, it was removed in RHEL 8 in 2018. However, Fedora continued to offer it. **XXX—need to follow up on this one.** Looks like Red Hat have several XFS developers working for them.

The main features of btrfs are:

1. **Extent-based allocation** – earlier in the chapter, traditional disk layouts were described where blocks are allocated to a file as direct, indirect, double indirect or triple indirects. In past times when files were not the huge sizes that can be seen today, this was adequate. Extent based allocation was introduced in VxFS and XFS. In this scheme, variable size extents are allocated to a file. Take the case of a database. If the size of the database is known upfront and, let's say, that it's going to be 1 TB, why not have a single extent of 1 TB. This simplifies the inode structure and guarantees that all the data is contiguous on disk getting the best in terms of performance.
2. Checksums –

3. Cloning –
4. Self-healing –
5. Deduplication –
6. Compression –
7. Snapshots – XXX. There is an example of using btrfs snapshots in section ??.
8. Quota groups –
9. File cloning –
10. In-place conversion –
11. Union mounting / seed devices –
12. Online defragmentation, volume growth, and shrinking –
13. Ability to handle swap files and swap partitions –

xxx

3.7.4 The Flash??? Filesystems

XXX - yaffs2 etc? F2FS

3.7.5 Stratis - RHEL???

Wrong place but need to cover it.

3.8 Kernel or Userspace?

Implementing a kernel-based filesystem is not trivial. I've worked on filesystem development for UNIX SVR3, SVR4, SCO UNIX, HP-UX, UnixWare, Solaris, Linux and various UNIX subsystems built on top of the Chorus microkernel. I actually started my filesystem career working on RFS back in the late 1980s. Debugging filesystems in the kernel is not easy. I've enjoyed using source level debuggers on HP-UX and Linux (through `gdb` / `kgdb`) but have also been stuck working at assembler level a lot and even had the pleasure (pain?) of working on debugging Solaris filesystems in `adb` (yes, all in assembler) on Sparc systems with their peculiar register windows and lack of developer trust in arguments displayed in a stack trace.

But if you want performance, you'll be using one of the disk-based filesystems running on top of a high-performance storage system. **raw vs XXX**

In many cases, performance is not critical. For example, for personal use, I keep all of my source code files on my Mac where they're backed up and use SSHFS, which is a user-space filesystem developed using FUSE. Performance is not critical and usability wins the day. **XXX—expand on this.**

does this section really belong here?

3.9 Pseudo Filesystems

A filesystem comprises a number of files which can be different types such as regular files or directories. In most cases, we deal with physical filesystems where these files are permanently stored. A pseudo filesystem on the other hand does not contain permanent storage. They are constructed as the kernel initializes and destroyed when the kernel shuts down.

The 3 well-known pseudo filesystems on Linux are proc (also called the /proc filesystem) which is mounted on /proc, ramfs (**image when loading - come back to this**) and the sysfs filesystem which is mounted on /sys.

devfs), or a middle point such as Linux devtmpfs ???

Which pseudo filesystems are generally used? In my Ubuntu server VM, I've installed FUSE (which has 3 registered filesystems) but in addition to these, there are another 23 registered pseudo filesystems:

```
$ cat /proc/filesystems | grep nodev
nodev    sysfs
nodev    tmpfs
nodev    bdev
nodev    proc
nodev    cgroup
nodev    cgroup2
nodev    cpuset
nodev    devtmpfs
nodev    configfs
nodev    debugfs
nodev    tracefs
nodev    securityfs
nodev    sockfs
nodev    bpf
nodev    pipefs
nodev    ramfs
nodev    hugetlbfs
nodev    devpts
nodev    ecryptfs
nodev    fuse
nodev    fusectl
nodev    efivarfs
nodev    mqueue
nodev    pstore
nodev    autofs
nodev    binfmt_misc
```

XXX—need to think about how much to say here or whether we cover 1 or more in the kernel chapter

3.9.1 The proc Filesystem

One filesystem that we will explore here is the proc pseudo filesystem (mounted on /proc and also sometimes called the /proc filesystem) which contains a wealth of information about running process and a lot more information about the running system. It also provides the means through which different kernel parameters can be changed. There are close to 17,000 lines of code in the proc filesystem.



Code for the proc filesystem can be found in `fs/proc` as well as `include/linux/proc_fs.h`

The proc filesystem can be traced all the way back to UNIX 8th Edition and was first described in the 1984 USENIX paper "*Processes as Files*":



9 <https://tinyurl.com/mvmb6nnj>

The proc filesystem was then further developed as a collaboration between Sun Microsystems and AT&T Bell Labs in the early 1990s for SVR4 UNIX and Solaris . Oddly enough, it actually began as a replacement for the `ptrace(2)` system call which was the primary interface for debugging applications. Another goal was to replace `ioctl(2)` calls with standard `read(2)` and `write(2)` system calls which are better suited to network access. Commands such as `ps(1)` and `truss(1)` (UNIX version of `strace(1)`) were rewritten to use `/proc`. I fully recommending reading the old USENIX paper.



10 <https://tinyurl.com/4wpmth6u>

The proc filesystem was introduced in to Linux in 1992. Processes are described by looking at `/proc/PID` where `PID` is the process ID. Figure 3.7 shows what's in `proc` on my Ubuntu VM. As you can see, there is a lot more in there other than just process-related information.

For a process to get information about itself it can look under `/proc/self`. Let's do a little exploring from the shell.

```
$ cd /proc/self
$ cat cmdline
-bash               # actually doesn't add a newline
$ ls -l exe
1rwxrwxrwx 1 spate spate 0 Nov 29 06:20 exe -> /usr/bin/bash
```

Need to look at following for `/proc/self/fds/255` -

Good site -

```
$ ls /proc
 1       169769  239   40    51   660    98253      iports      schedstat
10      17        25    41    52   688    acpi        irq        scsi
1071    182839  255   415   53   689    asound     kallsyms  self
1072    182964  26     42    54   69    bootconfig  kcore      slabinfo
11      182970  27     44    55   74    buddyinfo  keys       softirqs
12      182978  28     45    56   75    bus        key-users  stat
13      182982  280   451   587   76    cgroups   kmsg       swaps
134276  182988  29     452   588   77    cmdline  kpagecgroup sys
134277  19        3     453   59   78    consoles  kpagecount sysrq-trigger
134278  2         306   454   6    79    cpufreq   kpageflags sysvipc
14      20        31     455   60   8     crypto    loadavg   thread-self
15      203       32     458   603   84    devices   locks      timer_list
154     204       33     46    609   90    diskstats mdstat   tty
16      205       34     47    61   907   driver   meminfo   uptime
160329  21        35     48    610   91    dynamic_debug misc      version
169153  219       351   4812   615   913   execdomains modules  vmallocinfo
169158  22        352     49    648   921   fb        mounts   vmstat
169372  23        37     5    652   964   filesystems net      zoneinfo
169421  234       38     50    654   965   fs        pagetypeinfo
169762  235       39   5093   655   968   interrupts partitions
169763  236       4     5096   658   969   iomem    pressure
```

Figure 3.7: The contents of /proc on Ubuntu

3.9.2 The Ramfs Filesystem

One pseudo filesystem that is always used and part of the boot process is Ramfs, a RAM-based filesystem. XXX need to get info on how it's built.

The code for Ramfs is very small at only just over 600 lines of code. Since it's so small, the ramfs module is always loaded



All of the Ramfs code can be found in `fs/ramfs`.

There really isn't a lot too the code and the filesystem relies mostly on generic VFS-level operations. We'll describe these in detail in section 7.12.1.

Ramfs is only used as part of the boot process (really?). To create the older RAM-based filesystem, a pseudo block device (just allocate a virtually contiguous area of memory) was created on which to place the filesystem and a filesystem was placed in it (usually ext2). This imposed size limits which could be problematic at times and resulting in wasting time copying data to and from this *device* into the page cache plus using part of the device for ext2-specific data such as inodes and block maps. The pseudo device was removed a long time ago which simplified many things including the fixed space issue and a switch was made to the new ramfs filesystem. But this also potentially created other problems in that since it would be possible to fill up memory as more and more files were created. The first step to avoid this was to restrict it to superuser access.

3.9.3 The tmpfs Filesystem

See rest of chapter - it's used everywhere

3.9.4 The pipe Filesystem

fs/pipe.c calls register_filesystem().

3.9.5 The AuFS Filesystem

While most Live CD linux distributions used Aufs as of November 2016, Slackware used overlays for its live CD - so is it relevant or not?

3.9.6 The OverlayFS Filesystem

xxx

3.10 FUSE-based Filesystems

Linux provides a framework called FUSE (FileSystE in User space) that allows filesystems to be developed in user space, a much simpler process than developing a kernel-based filesystem. Although performance has been an issue over the years, there have been many improvements over the years and at the time of writing, there are some eBPF enhancements which could also increase performance.

Commercial vendors have a hard time implementing kernel-based filesystems due to lack of access to all kernel symbols (functions etc) as well as having to keep up with kernel updates. Either the commercial filesystem needs to be recompiled for each new kernel version or filesystem modules are force-loaded assuming testing reveals no issues. The latter works for minor Linux kernel updates. But there has been success in developing FUSE-based filesystems. One such example is the Thales CipherTrust Transparent Encryption UserSpace which provides encryption and access controls. This has proven to be acceptable in a number of customer environments, typically where performance is not critical.

Figure 3.8 shows components when using a FUSE-based filesystem.

Applications access files as normal unaware that they are using a FUSE filesystem. Inside the kernel, the FUSE kernel components route file operations up into user-space where they are handled by a FUSE library and the FUSE filesystem component which then routes these calls either back into the kernel to different part of the filesystem or over the network to a remote service.

cover mount options a little

There are a lot of FUSE-based filesystems. The FUSE Wikipedia page has a large list. We'll describe a few here and then give an example of using one in particular.

- s3fs — the ability to mount an AWS S3 bucket and view the objects as files

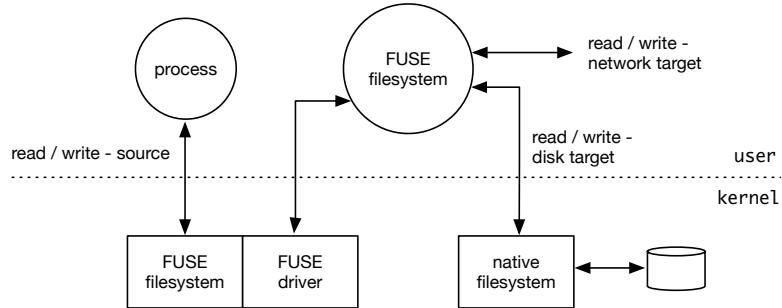


Figure 3.8: FUSE filesystems allow development in user-space

- EncFS — a filesystem that encrypts files and contents. Security concerns were raised about EncFS and there was rumored to be a newer version which would fix these concerns. That hasn't yet happened.
- SSHFS — access to a remote filesystem through use of SSH.
- WikipediaFS — the ability to view Wikipedia pages as if there were local files. It has since been deprecated.

In section 8 I'll be showing you how to develop two FUSE-based filesystems. The first will just "pass-through" operations whereby part of the file hierarchy will be visible through another directory. Files can be created, deleted and accessed as normal. The second filesystem will add encryption for file names and contents.

3.10.1 Demonstrating the SSHFS FUSE Filesystem

I wanted to demonstrate the WikipediaFS filesystem but found that it was deprecated many years ago. The same seems to be somewhat true of SSHFS. The github page indicates that the project is no longer maintained or developed although there have been changes within the last year. The repository has been archived so it's now read-only. It's also available to be installed, at least on Ubuntu. So let's give it a try. First, it needs to be installed:

```
$ sudo apt install sshfs
```

Next, we'll create a local directory on to which we will mount the remote directory. We run `mount` (1) to verify that it's mounted. Note that the `sshfs` command backgrounds itself.

```
$ mkdir remote
$ sshfs spate@192.168.4.21: /src/spfs remote
(spate@192.168.4.21) Password:
$ mount | grep remote
spate@192.168.4.21:/Users/spate/src/spfs on /home/spate/remote \
type fuse.sshfs (rw,nosuid,nodev,relatime,user_id=1000, \
group_id=1000)
```

At this point, all files are visible just as if they were local:

```
$ ls remote
cmds kern LICENSE README test
$ ls remote/kern
add_stuff dir.c      Makefile    sp_dir.c   spfs.h  sp_ioctl.c
dir2.c      fillfs.c  sp_alloc.c sp_file.c sp_inode.c
```

You can move around the directory structures, create, remove and read/write files. When you want to unmount the filesystem simply run:

```
$ fusermount -u remote
```

I spend a lot of time running scripts to sync files between my Mac and my Linux VMs. This looks like a much easier method so happy I gave it a try.

3.11 Physical Filesystems and Partitioning

- grub - fdisk or whatever is used now - limitations of disk sizes - relevant any more - raw vs block devices

XXX—example with partitioning and multiple filesystems on same disk / different partitions

I've partitioned my 1 GB virtual disk into two partitions using fdisk (8) so now I have two partitions:

```
# ls /dev/sd??
/dev/sda1  /dev/sda2
```

And now I can create a filesystem on each partition, mount each filesystem and copy some files into both:

```
# mkfs -t ext4 /dev/sda1
...
# mkfs -t btrfs /dev/sda2
...
# mount -t ext4 /dev/sda1 /mnt1
# mount -t btrfs /dev/sda2 /mnt2
# cp /etc/*/* /mnt1 2> /dev/null
# cp /etc/*/* /mnt2 2> /dev/null
# df -h | grep mnt
/dev/sda1      703M  552K  651M  1% /mnt1
/dev/sda2      291M  3.9M  210M  2% /mnt2
# df -h
Filesystem      Size  Used Avail Use% Mounted on
...
/dev/sda1      703M  552K  651M  1% /mnt1
/dev/sda2      291M  3.9M  210M  2% /mnt2
```

Note that although we have copied the same amount of data into each filesystem, the btrfs filesystem has used a lot more space than ext4. Some filesystems use more space for metadata and some filesystems allocated meta-data up front while others allocate it dynamically

therefore usage varies depending on the filesystem type. With the size of today's disks, it's really not a big deal.

3.12 Linux Namespaces

Linux *namespaces* are a feature of the Linux kernel that allow for different types of process isolation. Linux partitions different kernel resources such that one group of processes sees one set of resources while another group of processes can see a different set of resources. Namespaces were introduced into the Linux kernel in 2002 with enhancements and extensions continuing over the next several years. Namespaces are a fundamental aspect of how containers are implemented in Linux.

Since Linux kernel version 5.6, there are now eight different kinds of namespaces. All but the MNT namespace are beyond the scope of this book.

- **Mount (MNT)** – a list of mount points seen by the processes in this particular namespace. Filesystems can be mounted and unmounted in a mount namespace without impact on the main Linux filesystem.
- **Process ID (PID)** – this namespace has a different set of PIDs from other namespaces. PID namespaces are nested such that if a process is created it will have a one PID in the current namespace and a different PID in all namespaces above it up to the initial PID namespace. The first process created in any new PID namespace is assigned the process ID number 1.
- **Network (NET)** –
- **Inter-process Communication IPC** –
- **UTS** –
- **User ID (USER)** –
- **Control group (CGROUP) Namespace** –
- **Time Namespace** –

The MNT namespace is particularly relevant to filesystem access and you will see references to it through various parts of the filesystem code. It will be explained in the next section.

A Linux system starts out with a single namespace of each type and is used by all processes. Processes can create additional namespaces and/or join different namespaces. Namespaces are a fundamental aspect of containers on Linux and will be described in the next section.

Mount namespaces provide isolation regarding the list of mounts as seen by the process/tasks in each namespace. This ensures that different processes/tasks that belong to different mount namespaces will see distinct directories hierarchies.

Namespaces are documented in the `unshare(1)` command and `clone(2)` system call manpages. For an overview see the `mount_namespaces(7)` manpage.

More notes:

1. Namespaces are often used when untrusted code has to be executed on a given machine without compromising the host OS.
2. on process namespaces - All ‘ps’-like commands use the virtual procfs file system mount to deliver their functionalities.
3. Linux processes form a single hierarchy, with all processes rooting at init. Usually privileged processes in this tree can trace or kill other processes. Linux namespace enables us to have many hierarchies of processes with their own “subtrees” such that processes in one subtree can access or even know of those in another.
4. If we create a PID namespace and run a process in it, that first process becomes PID 1 in that namespace.
5. But this only means that the processes within the new namespace can not see parent process but the parent process namespace can see the child namespace. And the processes within new namespace now have 2 PIDs: one for new namespace and one for global namespace.
6. Linux kernel now tracks process’s PIDs using upid structure instead of a single pid value. The upid structure tells about the pid and the namespaces where that pid is valid.
7. There are new APIs - clone(), setns() and unshare() - need to look at them. The above "flags" (CLONE_NEWNS, CLONE_NEWUSER etc) are passed to clone() to create that type of namespace.

3.12.1 Shared Subtrees

Damn! This stuff is complicated!

- MS_SHARED – This mount point shares mount and umount events with other mount points that are members of its “peer group”. When a mount point is added or removed under this mount point, this change will propagate to the peer group, so that the mount or umount will also take place under each of the peer mount points. Propagation also occurs in the reverse direction, so that mount and umount events on a peer mount will also propagate to this mount point.
- MS_PRIVATE – This is the converse of a shared mount point. The mount point does not propagate events to any peers, and does not receive propagation events from any peers.
- MS_SLAVE – This propagation type sits midway between shared and private. A slave mount has a master — a shared peer group whose members propagate mount and umount events to the slave mount. However, the slave mount does not propagate events to the master peer group.
- MS_UNBINDABLE – Does not receive or forward any propagation events and cannot be bind mounted.

3.12.2 Root and chroot Environments

In a Unix-like OS, root directory(/) is the top directory. root file system sits on the same disk partition where root directory is located. And it is on top of this root file system that all other file systems are mounted. All file system entries branch out of this root. This is the system's actual root.

But each process has its own idea of what the root directory is. By default, it is actual system root but we can change this by using chroot() system call. We can have a different root so that we can create a separate environment to run so that it becomes easier to run and debug the process. Or it may also be to use legacy dependencies and libraries for the process.

It may appear that by separating a process using chroot() we ensure security by restricting the process not to access outside its environment. But in reality that is not very true. chroot() simply modifies pathname lookups for a process and its children, prepending the new root path to any name starting with /. Current directory is not modified and relative paths can refer any locations outside of new root.

Yeah so can you chroot when logging in to solve this???

3.12.3 Linux cgroups to Isolate and Manage Resources

Control groups(cgroups) is a Linux kernel feature which limits, isolates and measures resource usage of a group of processes. Resources quotas for memory, CPU, network and IO can be set. These were made part of Linux kernel in Linux 2.6.24.

sound great for virtualization - yes. what we want is:

Suppose we have an application we want to isolate usage for. Lets call it A1. Lets call rest of system as S. We will create a control group and assign resource limits on it: say 3GB of memory limit and 70% of CPU. Then we can add requisite application's process id to the group and application resource usage now is throttled. Though the application may exceed the limits in normal scenarios, it will be throttled back to pre set limits in case system is facing resource crunch. This makes even more sense when we are handling many VMs running on a machine-have a cgroup for VMs and throttle them individually to a set limit when resource contention happens.

to install:

```
sudo apt-get install cgroup-bin cgroup-lite cgroup-tools cgroupfs-mount libcgroupl
```

We can see a /cgroup directory created: this is used as mount point for cgroup virtual filesystems. etc/cgconfig.conf file gives info on what all mounts to expect. All controllers are mounted to /cgroup followed by controller name. eg- /cgroup/memory. To mount the requisite controllers, run sudo service cgconfig restart . Following this we see directories in /cgroup, each of which can be used to manage a cgroup subsystem.

<https://itnext.io/chroot-cgroups-and-namespaces-an-overview-37124d995e3d> - gives an example of using cgroups with a process that allocates more memory than allowed and gets killed. OK, good so far.

3.12.4 Mount Namespace Example

It always helps to see things in practice. This example shows how to create mount namespaces and show how mounts are shared or not shared depending on how XXX - TBD

```
$ sudo unshare -m bash  
# mount -t tmpfs tmpfs /mnt  
# touch /mnt/file
```

in another terminal:

```
$ ls /mnt  
lost+found/
```

the newly created file is not visible. Further, once you exit the namespace, the file and mount point are no longer visible.

But this doesn't work for disk-based mount points. even if you create two namespaces and then mount -t spfs in one namespace, it's visible everywhere. XXX - Must be doing something wrong or just not understanding this properly.

Inside proc you can see the mounting points visible for a specific process through the following paths:

1. /proc/[PID]/mounts
2. /proc/[PID]/mountinfo
3. /proc/[PID]/mountstats

3.12.5 Containers

Don't do too much. Just as things related to filesystems.

The fakeroot (1 command allows another command to be run such that it appears that the UID of the command has root privileges. A quick example:

```
$ fakeroot bash  
$ id  
uid=0(root) gid=0(root) groups=0(root),4(adm),24(cdrom),  
27(sudo),30(dip),46(plugdev),108(1xd),1000(spate)
```

The original goal of fakeroot was to allow for creation of packages while running as a normal user. This allowed the user to create files with permissions and ownership that would normally require root access to achieve, thus *fake root* privileges. This is particularly useful for creating Linux distribution packages, where the package maintainer does not need to run the process as root.

3.12.6 Creating a Container by Hand

To show how containers interact with the Linux filesystem, this example builds a container by hand XXX

Building a container by hand using namespaces: The mount namespace

FS	mkfs command	Utilities	Source Code
ext3	<code>mkfs.ext3(8)</code>	e2fsprogs	https://github.com/tytso/e2fsprogs
ext4	<code>mkfs.ext4(8)</code>	e2fsprogs	https://github.com/tytso/e2fsprogs
btrfs	<code>mkfs.btrfs(8)</code>	btrfs-progs	https://github.com/kdave/btrfs-progs
XFS	<code>mkfs.xfs(8)</code>	xfsprogs	https://github.com/mtanski/xfsprogs

Table 3.2: Various mkfs commands for the major filesystems

```
$ unshare -m - ??? - need sudo -no?
-> unshare - run program in new namespaces
try it and cd to $HOME - very odd
uses "Alpine Linux tarball" - small secure Linux something or other
talks about fakeroot command - good article here - https://unix.stackexchange.com/questions/9714/what-is-the-need-for-fakeroot-command-in-linux

$ sudo apt install fakeroot
run it and you get a root prompt

22.04-crash$ fakeroot
22.04-crash$ id
uid=0(root) gid=0(root) groups=0(root), 4(adm), 24(cdrom), 27(sudo), 30(dip), 46(plugd)

22.04-crash$ pwd
/home/spate/spfs/debug

in the same directory where you run the command XXX not sure how this fits in?????
XXX - look at mount_namespaces(7)
```

3.13 Filesystem Operations

There are several commands that relate to filesystems such as mounting and unmounting filesystems, remounting or collecting data about filesystem statistics such as running the `df(1)` command. This section covers relevant commands in this area.

3.13.1 Making a Filesystem

Filesystems can be created using the `mkfs(1)` command which is really just a front-end for a filesystem-specific `mkfs` command such as `mkfs.ext4(8)` and `mkfs.xfs(8)`.

The source code for many filesystem commands can be found at <https://github.com/util-linux/util-linux>. A comment in `mkfs.c` file as well as the manpage for `mkfs(8)` specifies that the command is deprecated in favor of the individual commands.

For the main Linux filesystems, Table 3.2 shows its `mkfs` command together with the package that contains the source code.

You can still use `mkfs(8)` for now. If no options are specified, an ext2 filesystem is created. To specify a filesystem type use the `-t` option (see section 3.11 for examples). Take a look at the source code. It's a very simple front-end to the specific filesystem commands which can be found in `/sbin`:

```
$ ls /sbin/mkfs*
/sbin/mkfs          /sbin/mkfs.cramfs  /sbin/mkfs.ext4
/sbin/mkfs.msdos    /sbin/mkfs.bfs     /sbin/mkfs.ext2
/sbin/mkfs.fat      /sbin/mkfs.vfat    /sbin/mkfs.btrfs s
/sbin/mkfs.ext3     /sbin/mkfs.minix   /sbin/mkfs.xfs
```

3.13.2 Mounting Filesystems

XXX—there is a section in the prog chapter. Need to cover mntopts etc here

The following is from Documentation/filesystems/sharedsubtree.rst - need to tie in with namespaces

Note: `mount(8)` command now supports the `-make-shared` flag, so the sample '`smount`' program is no longer needed and has been removed.

3.13.3 The Mount Table

The source code for the `mount(8)` command can also be found at <https://github.com/util-linux/util-linux>. It has quite a lot of work to do as you can see from the manpage, mostly in terms of managing a lot of possible options including passing filesystem-specific options to the kernel when it invokes the `mount(2)` system call.

The most basic forms of calling `mount(8)` are as follows:

```
# mount -t ext4 /dev/sda1 /mnt
# mount /dev/sda1 /mnt
# mount /mnt
```

If the `-t` option is passed, `mount` will find and invoke the filesystem-specific `mount` command. In the second two cases, it needs to determine what the additional parameters are and it does this by looking in the `/etc/fstab` file (`fstab(5)`). This file lists the filesystems to be mounted both during the bootstrap process and at a later time. The order of entries in this file is critical. There are six fields in each line of `fstab` which are listed in the order in which they should appear. Each field name shown is the field within `struct mntent`, the structure returned by a call to `getmntent(3)`. We borrow some words from the manpage but for details, please refer to the manpage.

1. `fs_spec` — this field describes the block special device, remote filesystem or filesystem image for loop device to be mounted or swap file or swap partition to be enabled.
2. `fs_file` — this field describes the mount point (target) for the filesystem. For swap partitions, this field should be specified as `none`.

3. `fs_vfstype` — the type of the filesystem (for example "ext4", "xfs" and so on). If `swap` is specified, this is a file or partition to be used as a swap device. We cover swapping in section 3.19.
4. `fs_mntops` — a comma-separated list of mount options which must include `ro` or `rw` (read-only or read/write).
5. `fs_freq` — this field is used by `dump(8)` command which will be covered in section 3.17
6. `fs_passno` — this field is used by `fsck(8)` to determine the order in which filesystem checks are done at boot time. The root filesystem should be specified with a `fs_passno` of 1. Other filesystems should have a `fs_passno` of 2

For device names you can use the actual path to the partition (for example `/dev/sda1`). By preference, `LABEL` or `UUID` should be used as device names can change as disks are added or removed. Here is an example of the largely unreadable `fstab` on my simple Ubuntu VM. I've abbreviated the long device names. Note that lines starting with "#" are comments and are ignored.

```
# <file system> <mount point> <type> <options> <dump> <pass>
# / was on /dev/ubuntu-vg/ubuntu-lv during curtin installation
/dev/disk/by-id/dm-uuid-LVM-AI...62ciB / ext4 defaults 0 1
# /boot was on /dev/vda2 during curtin installation
/dev/disk/by-uuid/03fb45b9...8427 /boot ext4 defaults 0 1
# /boot/efi was on /dev/vda1 during curtin installation
/dev/disk/by-uuid/368C-2362 /boot/efi vfat defaults 0 1
/swap.img      none      swap     sw      0      0
```

You can see things a little more clearly by running the following:

```
# df -h
Filesystem           Size  Used Avail Use% Mounted on
tmpfs                392M  1.4M  390M   1% /run
/dev/mapper/ubuntu--lv 38G   11G   25G  31% /
tmpfs                2.0G    0   2.0G   0% /dev/shm
tmpfs                5.0M    0   5.0M   0% /run/lock
/dev/vda2              2.0G  494M  1.4G  28% /boot
/dev/vda1              1.1G  5.1M  1.1G   1% /boot/efi
tmpfs                392M  4.0K  392M   1% /run/user/1000
# swapon
NAME      TYPE SIZE USED PRIO
/swap.img file 3.8G   3M   -2
```

There are two ways to view the current list of mounted filesystem. Either by looking at `/etc/mtab` or through `/proc/mounts`. If you search for "linux mtab /proc/mounts" you will see that there has been differences between the two with `/proc/mounts` being more up to date. The `/etc/mtab` file has generally been kept up to date by scripts which take the information from `/proc/mounts` for applications that still use `/etc/mtab`. That information may refer to some or older versions of Linux. On my Ubuntu VM I have:

```
# diff /etc/mtab /proc/mounts
```

which shows that both have the same information. Digging under the covers:

```
# ls -l /etc/mtab
lrwxrwxrwx 1 root root 19 Oct 19 14:05 /etc/mtab \
-> ../proc/self/mounts
# diff /proc/mounts /proc/self/mounts
```

Therefore the /proc is the correct one. My next thought was to see what the df(1) command used so I ran the following:

```
# strace -f -o s.out df -h
```

and searched through s.out. This is what I found:

```
168999 openat(AT_FDCWD, "/proc/self/mountinfo", O_RDONLY) = 3
```

Great! So not only do we have /proc/mounts and /proc/self/mounts but we also have /proc/self/mountinfo. Furthermore:

```
$ ls -l /proc/mounts
lrwxrwxrwx 1 root root 11 Nov 28 18:01 /proc/mounts \
-> self/mounts
```

3.13.4 Automounting Filesystems

It's covered during pathname resolution so far when crossing mount points

3.13.5 Unmounting Filesystems

The umount(1) command underpinned by the umount(2) system call is used to unmount a filesystem. Either the mount point or the device can be used as an argument. Unmounting a filesystem is easy, as long as it works. It can be as simple as the following example shows:

```
$ sudo mount -t spfs /dev/sda1 /mnt
$ sudo umount /mnt
$
```

A filesystem that resides on /dev/sda1 is mounted on /mnt and then immediately unmounted. You won't see anything printed but the umount(1) command unless there is an issue.

Filesystems can be force unmounted which is particularly useful for network filesystem mounts. XXX

3.13.6 The lsof(1) Command

If a filesystem is busy, you will need to locate the users and/or processes that are using the filesystem. This is where the lsof(1) command comes in useful. Assume that a filesystem is mounted on /mnt and in one terminal, the following command is run:

```
$ less /mnt/lorem-ipsum
Lorem ipsum dolor sit amet, consectetur
...
```

an attempt is made to unmount it:

```
$ sudo umount /mnt
umount: /mnt: target is busy.
```

The `lsof(1)` command can be used to list what processes are access files in this filesystem. In this simple example, here is the output shown when running `lsof` on the mount-point:

```
$ lsof /mnt
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
less    1012 spate 4r   REG    7,7    2972     8 /mnt/lorem-ipsum
```

This allows an administrator to locate the user and tell them to exit the filesystem or the process can be killed.

3.13.7 Reporting File System Disk Space Usage

In the previous section we displayed the result of running `df -h` and mentioned that it gets its information from opening and reading `/proc/self/mountinfo`. So what is in this file? Let's take a look at the first few lines:

```
26 32 0:24 / /sys rw,...,relatime shared:7 - sysfs sysfs rw
27 32 0:25 / /proc rw,...,relatime shared:13 - proc proc rw
28 32 0:5 / /dev rw,...,relatime shared:2 - devtmpfs udev ...
29 28 0:26 / /dev/pts rw,...,relatime shared:3 - devpts rw ...
```

XXX—no idea! come back and explain this I've removed some arguments so the output will fit on the screen so you should try running this and see the full set of arguments.

3.13.8 Fixing Filesystems with `fsck`

When a filesystem is mounted, one of the first things that the filesystem does it to mark the filesystem as dirty. If the system crashes, the filesystem will need to be repaired. This is where the `fsck(8)`, the FileSystem CHecker, command comes into play. There is a generic `fsck(8)` command as well as individual filesystem `fsck` commands.

Here is partial output from running `fsck` on a UFS filesystem. This demonstrates the types of messages that used to be display. In this example, `fsck` is running in interactive mode. I always found it amusing that system administrators would know how to answer these questions. As time evolved, `fsck` was generally run in non-interactive mode and the utility was left to decide how to fix things.

```
# fsck -t ufs -fy /dev/vtbd0p2
** /dev/vtbd0p2 (NO WRITE)
SETTING DIRTY FLAG IN READ_ONLY MODE

UNEXPECTED SOFT UPDATE INCONSISTENCY
```

```
** Last Mounted on /
** Root file system
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Cyl groups
FREE BLK COUNT(S) WRONG IN SUPERBLK
SALVAGE? no
...
```

talk about lost+found

XXX—how much time to spend on this?

3.13.9 Filesystem Debugging

In addition to running `fsck(8)` on a filesystem to repair it, most filesystems have another tool that allows administrators or developers to debug the filesystem. This allows reading and modifying various structures. Such a tool is invaluable when developing a filesystem in addition to log messages to see when objects get written to disk and to make sure that all counts and links are updated correctly. Usually such a tool is run when the filesystem is unmounted but for filesystem development, running when the filesystem is mounted is very useful.

Each filesystem debugger is different but there will be some similarities in terms of displaying directories, inodes and perhaps being able to walk through the filesystem (similar to using the `cd` command).

Before we run `debugfs(8)` let's take a quick look at the contents of the filesystem that we're going to debug. We have 2 files and one directory in addition to the root directory:

```
/mnt1
/mnt1/lorem-ipsum
/mnt1/mydir
/mnt1/mydir/hello.txt
```

To enter the debugger, simply run the command against the device where the filesystem resides. Type `?` to list the available commands.

```
# debugfs /dev/sda1
debugfs 1.46.5 (30-Dec-2021)
debugfs: ?
Available debugfs requests:

show_debugfs_params, params Show debugfs parameters
open_filesys, open      Open a filesystem
close_filesys, close    Close the filesystem
freefrag, e2freefrag   Report free space fragmentation
feature, features      Set/print superblock features
....
```

Next we show how to navigate through the directory structure, stat files and display their contents:

```
debugfs: pwd
[pwd] INODE: 2 PATH: /
[root] INODE: 2 PATH: /
debugfs: ls
2 (12) 2 (12) .. 11 (20) lorem-ipsum 12 (4040) mydir
debugfs: cd mydir
debugfs: ls
12 (12) . 2 (12) .. 13 (4060) hello.txt
debugfs: stat hello.txt
Inode: 13 Type: regular Mode: 0644 Flags: 0x80000
Generation: 885043848 Version: 0x00000000:00000001
User: 0 Group: 0 Project: 0 Size: 6
File ACL: 0
Links: 1 Blockcount: 8
Fragment: Address: 0 Number: 0 Size: 0
ctime: 0x6390b30e:dd5258b0 -- Wed Dec 7 15:36:46 2022
atime: 0x6390b30e:dd5258b0 -- Wed Dec 7 15:36:46 2022
mtime: 0x6390b30e:dd5258b0 -- Wed Dec 7 15:36:46 2022
crttime: 0x6390b30e:dd5258b0 -- Wed Dec 7 15:36:46 2022
Size of extra inode fields: 32
Inode checksum: 0x7cc9f75a
EXTENTS:
(0):33792
debugfs: bd 33792
0000 6865 6c6c 6f0a 0000 0000 0000 0000 0000 hello.....
0020 0000 0000 0000 0000 0000 0000 0000 0000 .....
*
```

There are many options for navigating through the filesystem, displaying inodes, blocks, xattrs and other objects. The real use comes in being able to modify various structures including inodes and the superblock, replaying the journal and so on. A word of warning—if you’re planning on modifying data structures on disk, be very careful. I would recommend creating a small demo filesystem that allows to mirror the operations you plan on performing on the real filesystem. You should also run the filesystem’s `fsck` command after you make any modifications to make sure that the filesystem is structurally intact.

A full list of commands can be found in the `debugfs` (8) manpage.

3.14 Loopback Mounts

Right place?

Even if you are not familiar with loop devices, you will likely have seen them at some point while using Linux. Furthermore, if you use Ubuntu, you’ll see several loop devices by running the `lsblk` command:

```
$ lsblk
```

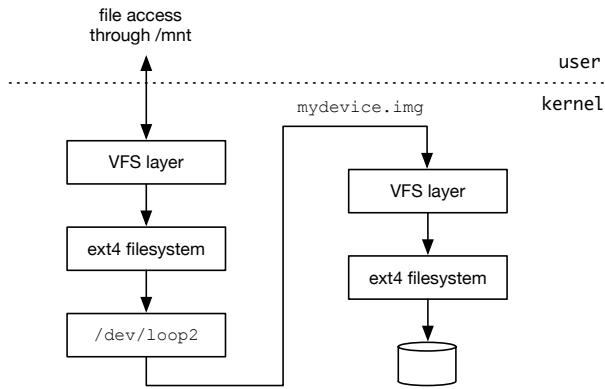


Figure 3.9: Accessing a filesystem through a loop device

```

NAME      MAJ:MIN RM    SIZE RO TYPE MOUNTPOINTS
loop0     7:0      0   59M  1  loop  /snap/core20/1740
loop1     7:1      0  59.1M 1  loop  /snap/core20/1782
loop2     7:2      0 132.8M 1  loop  /snap/lxd/24242
loop3     7:3      0 139.3M 1  loop  /snap/lxd/24331
loop4     7:4      0  43.2M 1  loop  /snap/snapd/17954
loop5     7:5      0    43M  1  loop  /snap/snapd/17885
...
  
```

This is due to *screens* which is a software packaging and deployment system developed by Canonical. Snap applications are mounted as loop devices.

In a virtualized world, it's easy to create new devices to play with when adding new filesystems, experimenting with volume managers and so on. But there is also another way which is also very useful, not just for experimentation but for **containers - check**. This is where *loopback devices* come into play. You can also create very small devices / filesystems if you're limited on space.

Great for playing with filesystems xxx

XXX—come back here once namespaces, virtualization etc are well understood

XXX odd because you can run `mkfs -t` on a regular file so we don't need loopback devices. So when are they really useful? See here for info -

Figure 3.9 shows how filesystems are accessed through a loop device. There is a file called `mydevice.img` in the root filesystem and onto which a loop device has been attached. The `/dev/loop2` block device can then be used to create an ext4 filesystem which is then mounted at `/mnt`.

xxx

3.14.1 Demonstrating Loopback Devices

This example shows how to create the loop device shown in figure 3.9, create an ext4 filesystem on it, mount it to make it available for general use.

There are only a few simple steps needed to create a loopback device, add a filesystem and mount it to make it available. First of all, we created a 10 MB file:

```
$ dd if=/dev/zero of= /mydevice.img bs=1024k count=10
5+0 records in
5+0 records out
5242880 bytes (5.2 MB, 5.0 MiB) copied, 0.00927079 s, 566 MB/s
```

You can experiment with sizes. If you make it 5 MB, it will be too small for ext4 to add a journal but 10 MB is fine. The next step is to call `losetup` to create a loopback device. The `find` option locates the first available loop device and the `show` option displays which loop device was assigned.

```
$ sudo losetup -find -show mydevice.img
/dev/loop2
```

The device is accessible via `/dev/loop2` so the next step is to make a filesystem on it, mount it and show space available:

```
$ sudo mkfs -t ext4 /dev/loop2
mke2fs 1.46.5 (30-Dec-2021)
Discarding device blocks: done
Creating filesystem with 2560 4k blocks and 2560 inodes

Allocating group tables: done
Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done
$ sudo mount /dev/loop2 /mnt
mount: /mnt: /dev/loop2 already mounted on /mnt.
$ df -h /mnt
Filesystem      Size  Used Avail Use% Mounted on
/dev/loop2      5.4M   24K  4.7M   1% /mnt
```

It looks like, and will function like, any other filesystem. Once you no longer need the filesystem/device you can unmount it and call `losetup` to detach the device. You can keep the file or remove it as needed.

```
$ sudo umount /mnt
$ sudo losetup -detach /dev/loop2
```

Once mounted, it looks like, and will function like, any other ext4 filesystem but with an additional overhead since there is now an ext4 filesystem on top of an ext4 file residing in another ext4 filesystem.

Once you no longer need the filesystem/device you can unmount it and call `losetup` to detach the device. You can keep the file or remove it as needed.

3.15 Client Caching With FS-Cache

is this the right place to cover this? I don't think so but come back later. Perhaps "miscellaneous topics"

FS-Cache is a persistent local cache introduced into Linux by Red Hat in 2001. It allows file systems to take data that has been retrieved from over the network and cache it on local disk. The goal is to help minimize network traffic for users on a client accessing data from a file system mounted over the network. NFS is one such example and an NFS mount option instructs the client to mount the NFS share with FS-Cache enabled.

Why not just utilize the page cache? What are the tradeoffs? Data out of sync? Multiple users accessing the same files?



The FS-cache code can be found in `fs/fscache`. There are only 2700 LOC.

Figure 3.10 shows how FS-Cache works. **XXX—not sure this is correct. Come back once i figure it out**

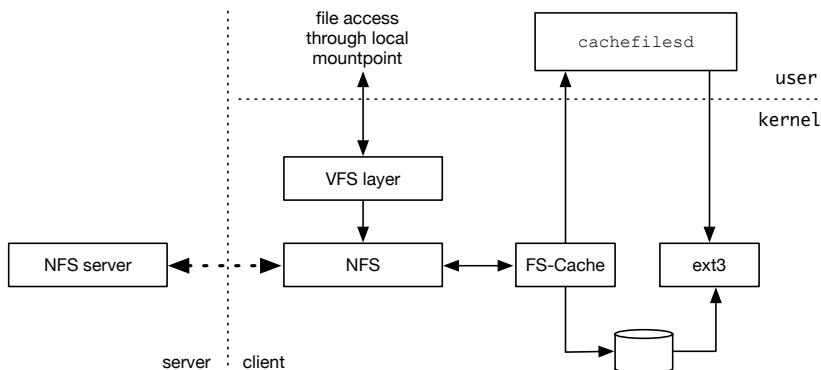


Figure 3.10: Client caching with FS-Cache

The goal of FS-Cache is that it should be as transparent as possible to the users and administrators of a system. In what circumstances is this transparency not achieved? **XXX—need to find out.**

FS-Cache does not alter the basic operation of a file system that works over the network - it merely provides that file system with a persistent place in which it can cache data. For instance, a client can still mount an NFS share whether or not FS-Cache is enabled. In addition, cached NFS can handle files that won't fit into the cache (whether individually or collectively) as files can be partially cached and do not have to be read completely up front. FS-Cache also hides all I/O errors that occur in the cache from the client file system driver.

Blah blah - NFS will not use FS-Cache unless explicitly stated when mounted as follows:

```
# mount nfs-share:/ /mount/point -o fsc
```

XXX - change args - try it first

3.16 Network Filesystems

Simply put, a network file system allows a remote client to access a filesystem over a network in a similar way to a local file system. Developed over the course of forty years, Sun's NFS (Network Filesystem) has become the dominant network filesystem and has been ported to and ran on just about every operating system developed over that period of time. But NFS isn't the only network filesystem that Linux supports. NFS will be described soon but here are some other network filesystems that Linux supports:

- dav – GVFS?
- davfs2 – GVFS?
- sftp – GVFS?
- SMB –
- CIFS –
- SSHFS – a FUSE-based filesystem that runs on top of the SSH protocol, is accessible by non-root users and makes a remote directory visible through a local directory.

Which filesystem to use can depend on several factors - TBD -

Section ?? shows an example of running SSHFS.

3.16.1 NFS – the Network File System

NFS has been the most popular network filesystem for several decades now. It started at Sun Microsystems in 1984. The initial release (version 1) was primarily used in-house. This was followed by three major versions which will be described below.

The original paper *The Sun Network Filesystem: Design, Implementation and Experience* can be found with the following link. It describes the goals of the project, how NFS utilized the new VFS/vnode architecture, architecture, design and implementation. The *hard issues* are also described such as file locking.



URL 11 – <https://tinyurl.com/5f5tyakz>

Figure 3.11 shows the overall architecture of NFS as it was in the 1980s and also, as it is now in the Linux kernel today. There are actually two distinct filesystems, one on the client

and one on the server. The client component appears as any other filesystem but takes VFS-level requests and communicates with the NFS server to fulfill them using NFS messages defined in the RFC standard. NFS was originally built on-top of UDP but for many years has primarily run over TCP/IP.

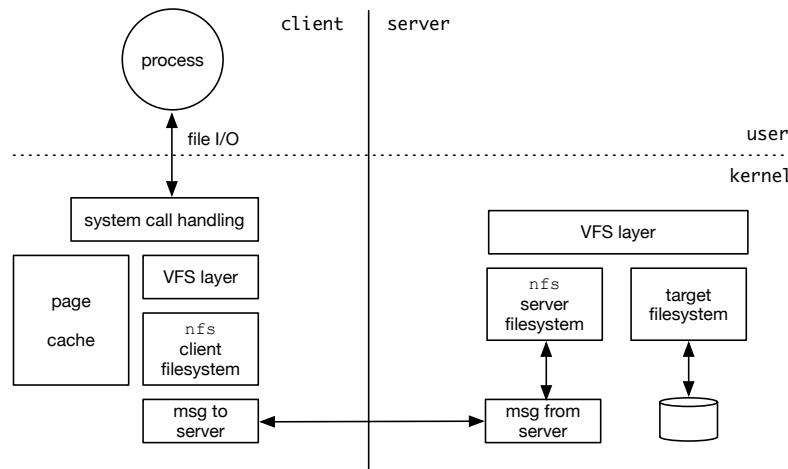


Figure 3.11: NFS client and server

NFS version 2 was defined in RFC 1094 and released in 1989, NFS v2 worked on top of UDP (User Datagram Protocol) with the goal of being stateless. This meant that the NFS server was unaware of the clients accessing it therefore locking was implemented outside of the protocol. NFS v2 had a 2 GB file limit and poor performance due to synchronous writes. It would take another few years before the Large File Summit (LFS) came into being with the goals of accessing files greater than 2 GB. And 32-bit architectures were prevalent. NFS v2 was implemented on the new VFS/vnode architecture.

NFS version 3, defined in RFC 1813 and released in June 1995 had two main goals: add support for *large files* (> 2 GB - 1) and improve performance. In addition to 64-bit file offsets, there were performance improvements in the following areas:

- handling asynchronous writes on the server, to improve write performance.
- additional file attributes returned for many operations removing the need to keep making additional calls to retrieve them.
- a new REaddirplus operation for which the server returned file handles and attributes along with file names when scanning a directory.

NFS was becoming very popular and interoperability annual testing occurred at *Connec-tathon* events in 1986 where vendors would bring along their NFS client, server or both and informally test with each other. When I joined SCO in 1993, Lachmann Associates were providing our NFS and TCP/IP implementations. Around the time NFS v3 was being

introduced, different operating system companies and NFS vendors such as Lachmann Associates, had already started to add NFS over TCP/IP where it offered better performance than UDP over WANs partly due to removing the 8 KB limit imposed by UDP.

For those interested in the history of the large file summit, details can be found here. Operating system and filesystem vendors spent a considerable amount of time in both attending these conferences and implementing support for large files.



URL 12 – <https://tinyurl.com/y4r8hep8>

Following NFS v3, there was an extension introduced by Sun called WebNFS.

WebNFS was an extension to NFSv2 and NFSv3 allowing it to function behind restrictive firewalls without the complexity of Portmap and MOUNT protocols. WebNFS had a fixed TCP/UDP port number (2049), and instead of requiring the client to contact the MOUNT RPC service to determine the initial file handle of every filesystem, it introduced the concept of a public filehandle (null for NFSv2, zero-length for NFSv3) which could be used as the starting point. Both of those changes have later been incorporated into NFSv4.

Sun Microsystems turned over the development of future NFS protocols to IETF (the Internet Engineering Task Force) and IETF was responsible for NFS versions 4 in 2000, 4.1 in 2010 and 4.2 in 2016.

NFS v4 introduced a stateful protocol and provided several security enhancements. TBD.

The NFS v4.1 protocol incorporated support for clustered server deployments. Under the pNFS extension, this provided support for scalable parallel access to files distributed among multiple servers. It also introduced a feature called *session trunking*, also called NFS Multi-pathing ... introduced in VMWare ESXi et al.

NFS version 4.2 (RFC 7862) was published in November 2016[9] with new features including: server-side clone and copy, application I/O advise, sparse files, space reservation, application data block (ADB), labeled NFS with sec_label that accommodates any MAC security system, and two new operations for pNFS (LAYOUTERROR and LAYOUTSTATS).

One big advantage of NFSv4 over its predecessors is that only one UDP or TCP port, 2049, is used to run the service, which simplifies using the protocol across firewalls

For more information about the history of NFS, Brent Callaghan's 1999 book *NFS Illustrated*[1] is a good reference and covers all versions including WebNFS. Details about the NFS implementation in Linux can be found in section 12.4.

3.17 Backup / restore

```
old days - dump(8)
dd vs FS backup (tar etc)?
xattrs etc
snapshots / freeze - usable - btrfs - anything that uses this stuff?
```

When I was at VERITAS and not long after we made our first port to Linux, we developed a feature that we called the File Change Log (FCL). One of the biggest problems with backup technology at the time was the time spent figuring out which files needed to be backed up. We also owned Netbackup which was the world's most successful enterprise backup software. The time taken to scan the filesystem to determine which files needed to be backed up was extremely time consuming and the File Change Log could reduce that time dramatically.

Living in a world where servers are generally virtualized has changed the backup space dramatically with vendors such as Veeam backing up at the VM layer or even within the storage subsystem.

Having said that, this section explores the different backup and restore capabilities pro

- rsync –
- tar / cpio –
- ReaR – Relax and Recover
- dump / restore–
- timeshift – and btrfs support?

what works in the commercial space in Linux - open source vs ...

XXX—expand ...

3.17.1 Freezing and Thawing Filesystems

Freezing and thawing a filesystem was a feature initially added to Linux to freeze an XFS filesystem when used with hardware RAID devices that support the creation of snapshots. This would involve a set of actions such as:

```
# xfs_freeze -f /xfs_mnt
Take a snapshot of the file system ...
# xfs_freeze -u /xfs_mnt
```

This initial feature has now been extended to support 3xt3/4 and JFS in addition to XFS and is described in the `fsfreeze(8)` manpage. There are two options:

1. `-freeze / -f` – freeze the filesystem from all future modifications. If there are any pending operations, they will be allowed to complete. Any subsequent calls to `write(2)` (or similar system calls) will be suspended until an unfreeze request is made.

2. `-unfreeze / -u` – un-freeze the filesystem allowing operations to continue. If filesystem modifications were blocked by call freeze call, they will be unblocked and allowed to complete.

A call to `fssfreeze` is no necessary for device-mapper devices since device-mapper (and LVM) automatically freeze filesystems on the device when a snapshot creation is requested. For more details see the `dmsetup(8)` man page.

For further information, visit the following site:



URL 13 <https://tinyurl.com/bkm7tyb3>

There are operations at the filesystem level in the kernel to support freeze / thaw which only a small number of filesystems support. These will be described in section 6.23.1.

Snapshots will be further described when the individual filesystems are discussed in detail - **section TBD**

3.18 Quotas

File system quotas, which have been around since the early BSD UNIX days, are a mechanism to control the amount of disk space that can be consumed by specific users, by groups of users, or by a site-determined group of users called an *admin set*. It's a feature that I haven't personally seen used a lot over my career but certainly has its place. I remember when I was working on VxFS at VERITAS, we spent considerable time on quotas but didn't use them ourselves. We opted for the old-fashioned way of running a script and emailing the filesystem team with who was using what space! This section shows how quotas can be used on Linux with an example of XXX.

xxx - blah

figure out who actually uses it and what filesystems?

There are three different types of quotas supported by Linux:

- `vfsold` – version 1 quotas.
- `vfsv0` – version 2 quotas.
- `xfs` – the quota on XFS filesystems.

To enforce `vfsold` and `vfsv0` quota checking, you must turn it on using the `quotaon` command. The newer version (`vfsv0`) provides a journald quota system, which means that a quota check is not needed during boot, something that can take considerable time.

There are both *block limits* and *file limits* for quotas. For both, quotas have both soft and hard limits. Generally speaking, a user is allowed to exceed the soft limit but not exceed the hard limit. However, there is a grace period for soft limits and once this grace period has expired the soft limit is treated like the hard limit.

To understand the different limits, here is output from the `repquota(8)` command that will be explored in more detail in the next section:

```
$ sudo repquota -v /mnt
*** Report for user quotas on device /dev/sda1
Block grace time: 7days; Inode grace time: 7days
      Block limits          File limits
User        used    soft    hard grace   used    soft    hard
grace
-----
root       --     20      0      0           2      0      0
spate      +-   10252    5120   10240  6days     4     10     20
ashley     --      0    5120   10240           0     10     20
sam        --      0   10240   15360           0      5     10
```

The command shows block and file limits. User spate has exceeded the block soft limit for which the grace period is 7 days.

3.18.1 An Example of Using Filesystem Quotas on Ubuntu

To show how filesystem quotas are setup and used, this section shows a simple example of using user quotas on a Ubuntu system. To get started, the `quota` tools package first needs to be installed as follows:

```
$ sudo apt install quota
```

To verify that the quota tools are installed correctly, run the following:

```
$ quota -version
Quota utilities version 4.06.
...
```

The goal here is to add user quotas for a filesystem on `/dev/sda2`, to be mounted on `/mnt`, to provide some limits for a specific set of users and show what happens when these limits are exceeded.

The first step is to make an ext4 filesystem and enable user quotas:

```
$ sudo mkfs.ext4 -Equotatype=usrquota /dev/sda1
mke2fs 1.46.5 (30-Dec-2021)
Discarding device blocks: done
Creating filesystem with 261888 4k blocks and 65536 inodes
Filesystem UUID: 68b55c65-22ee-454d-8a60-1b99ed1c861c
Superblock backups stored on blocks:
            32768, 98304, 163840, 229376

Allocating group tables: done
Writing inode tables: done
Creating journal (4096 blocks): done
Writing superblocks and filesystem accounting information: done
```

Mount the filesystem specifying the `usrquota` option: **XXX—when should quotaon be run?**

```
$ sudo mount -ousrquota /dev/sda1 /mnt
```

Check to make sure that quotas are enabled for this mountpoint:

```
$ mount | grep sda1
/dev/sda1 on /mnt type ext4 (rw,relatime,quota,usrquota)
```

Run quotacheck which can scan a filesystem for disk usage, create, check and repair quota files. Without any arguments it will create the file aquota.user in the root directory.

```
$ sudo quotacheck /mnt
22.10-target$ ls /mnt
aquota.user  lost+found/
```

The setquota command can be used to set quota limits. The arguments are:

```
$ sudo setquota -u [username] [soft disk limit] [hard disk limit]
[soft inode limit] [hard inode limit] [partition]
```

To set limits for user ashley:

```
$ sudo setquota -u ashley 5M 10M 10 20 /mnt
$ sudo setquota -u sam 10M 15M 5 10 /mnt
```

XXX

```
$ sudo quota -vs ashley
Disk quotas for user ashley (uid 1001):
      Filesystem   space   quota   limit   grace   files   quota
      limit   grace
/dev/mapper/ubuntu--vg-ubuntu--lv
          16K       0K       0K               4       0
0
          /dev/sda1     0K     5120K   10240K           0       10
20
```

To report usage the repquota (8) command can be used as follows:

```
$ sudo repquota -s /mnt
*** Report for user quotas on device /dev/sda1
Block grace time: 7days; Inode grace time: 7days
                                Space limits                  File limits
User             used    soft    hard   grace   used    soft    hard
grace
-----
root          --    20K     0K     0K           2    0    0
0
```

If users have not yet created any files, nothing will be displayed. But this displays more:

```
$ sudo repquota -v /mnt
*** Report for user quotas on device /dev/sda1
Block grace time: 7days; Inode grace time: 7days
                                Block limits                  File limits
User             used    soft    hard   grace   used    soft    hard
grace
-----
root          --    20      0      0           2    0    0
0
```

```
spate    --      0    5120    10240          0    10    20
ashley   --      0    5120    10240          0    10    20
sam      --      0   10240   15360          0     5    10

Statistics:
Total blocks: 7
Data blocks: 1
Entries: 4
Used average: 4.000000
```

but spate has created files and nothing is being shown. If I run:

```
$ sudo quotacheck /mnt
```

again I can then see info:

```
$ sudo repquota -v /mnt
*** Report for user quotas on device /dev/sda1
Block grace time: 7days; Inode grace time: 7days
                                         Block limits                         File limits
User           used    soft    hard   grace       used    soft    hard
grace
-----
root        --      20      0      0          2      0      0
spate      +-  10252    5120   10240  7days        4    10    20
ashley    --      0    5120    10240          0    10    20
sam       --      0   10240   15360          0     5    10

Statistics:
Total blocks: 7
Data blocks: 1
Entries: 4
Used average: 4.000000
```

they say that quotacheck should be run without users accessing the filesystem.

3.19 Swap space

Swap space has traditionally been an area on disk used to temporarily write memory pages, thus freeing them up for use by other processes or even the same process depending on memory usage. This creates the illusion of having more physical memory than is actually available although it comes at a cost, namely performance.

Traditionally, swap space has been to a raw device, a feature that Linux offers. Some versions of UNIX later (from SVR4 onwards) included a swap filesystem. Swaps was unusual in that it used physical memory for the swap space, but also required physical swap space on a disk.

Linux supports traditional swap devices in addition to the ability to swap to a file within a filesystem with use of the `mkswap(8)` and `swapon(8)` commands.

Rules about the amount of swap space being needed has been debated for years but really depends on the application load that is being run. If your mission critical database is constantly paging to/from swap, you're in trouble. The rule of thumb used to be 2x the amount of RAM in the system but with today's much larger RAM configurations or suspending a laptop, this general rule of thumb may not always make less sense. The following article gives a good guide as to how much swap space may be needed.



URL 14 <https://tinyurl.com/mryahu3m>

The `free(1)` command can be used to display amount of free and used memory in the system. In this example, on the VM I am using where I am doing nothing but configure swap devices at the time of writing, there is little activity. Thus the RAM is only being partially used and no swap space is being used.

```
# free
              total        used        free      shared  buff/cache
available
Mem:       4006060       251732     1004024        5368     2750304
3557676
Swap:      262140           0      262140
```

To illustrate how swapping to a file on a filesystem works, the following example makes use of the `mkswap(8)` and `swapon(8)` commands to create a swap file.

The first step is to create a file which will be used as a swap device. In this example, the file is `/swapfile` and it is 256 MB in size:

```
# dd if=/dev/zero of=/swapfile1 bs=1M count=256
262144+0 records in
262144+0 records out
268435456 bytes (268 MB, 256 MiB) copied, 0.384721 s, 698 MB/s
```

Next, the permissions should be changed so that it's readable and writable by root but nobody else. You do not want regular users being able to read from this file as it may contain sensitive data.

```
# chmod 0600 /swapfile
# ls -l /swapfile
-rw----- 1 root root 268435456 May  3 16:13 /swapfile
```

The `mkswap(8)` command is then run to set up a Linux swap area on the file.

```
# mkswap /swapfile
Setting up swapspace version 1, size = 256 MiB (268431360 bytes)
no label, UUID=043d78c9-295e-4a23-895f-c0ed6e4f51b5
```

The second to last step is to activate the swap space as follows:

```
# swapon /swapfile
# swapon
NAME      TYPE SIZE USED PRIO
/swapfile file 256M   0B    -2
```

Lastly, `/etc/fstab` should be modified to enable the swap file automatically on boot. An entry in `/etc/fstab` will look like:

```
/swapfile          swap      swap    defaults      0 0
```

As mentioned above, this topic is skirting the area of filesystem access since the the filesystem does not see this file as being anything special. The only other thing to point out is that the space in the file could be quite fragmented so if a file with an extent of the size of the swap device could be created, that would help with performance. More importantly would be to have the block size of the filesystem be the same as the page size so page-sized writes do not get split up and written to different locations on disk.

3.20 The Boot Process and Run Levels

As seen earlier and by running `mount (1)` there are 34 filesystems mounted on a simple Ubuntu VM. When do these filesystems get mounted or unmounted? Certainly by the time the system is up they are there and accessible. When to mount them is determine as part of boot process.

boot to single user and show what's mounted

apart from ramfs, say what?

Here are the 6 different run levels. The run level number is on the left.

1. System halt i.e the system can be safely powered off with no activity.
2. Single user mode.
3. Multiple user mode with no NFS(network file system).
4. Multiple user mode under the command line interface and not under the graphical user interface.
5. User-definable.
6. Multiple user mode under GUI (graphical user interface) and this is the standard runlevel for most of the LINUX based systems.
7. Reboot which is used to restart the system.

The `telinit (8)` command can be used to change the run level.

3.20.1 Run Level Scripts

At each run level, init (???) will execute a set of scripts. ... this is a lot more complex than it used to be. History:

below is cut n paste so needs rewording

- System V init – this is the traditional init – **not used anymore?**

- Upstart init, used by older Ubuntu and some RHEL (Red Hat) and older Fedora versions
- systemd init, used by modern Fedora, Ubuntu, Debian, RHEL, SUSE versions

The *systemd init* is described by the `systemd(1)` manpage.

`user@.service(5)` and a whole host of others
 everything is in `/etc/init.d` and `/etc/rcX.d/*` contains symlinks to files in `/etc/init.d`
 run level scripts are in `/etc/rcX.d`

```
$ ls /etc/rc
rc0.d/ rc1.d/ rc2.d/ rc3.d/ rc4.d/ rc5.d/ rc6.d/ rcS.d/
```

For example, here are the contents of `rc3.d`:

```
K01sysstat -> ../init.d/sysstat
S01apport -> ../init.d/apport
S01console-setup.sh -> ../init.d/console-setup.sh
S01cron -> ../init.d/cron
S01dbus -> ../init.d/dbus
S01grub-common -> ../init.d/grub-common
S01irqbalance -> ../init.d/irqbalance
S01lvm2-lvmpolld -> ../init.d/lvm2-lvmpolld
S01multipath-tools -> ../init.d/multipath-tools
S01open-vm-tools -> ../init.d/open-vm-tools
S01plymouth -> ../init.d/plymouth
S01rsync -> ../init.d/rsync
S01ssh -> ../init.d/ssh
S01unattended-upgrades -> ../init.d/unattended-upgrades
S01uuidd -> ../init.d/uuidd
```

Each file is a symlink to a script in `/etc/init.d`.

XXX still don't know when filesystems are mounted!!!

The meaning of each run level is as follows:

- 0 – Shut down (or halt) the system.
- s – Single-user-mode; usually aliased as 's' or 'S'.
- 2 – Multiuser mode without networking.
- 3 – Multiuser mode with networking.
- 5 – Multiuser mode with networking and the X Window System.
- 6 – Reboot the system.

Most people are unlikely to need to deal with run levels. With personal VMs/systems it's usual just to start, pause, shutdown or reboot. Installing software that needs to be started when the system boots and shutdown cleanly on reboot or shutdown will require knowledge of how `systemd(1)` works.

3.21 Conclusion

This chapter presented many topics related to filesystems from disk partitioning to mounting different filesystems, pseudo filesystems, network filesystems and how the different main Linux filesystems compare in terms of features and performance..

Now that we've described all the file/filesystem-related programming interfaces and this chapter has covered filesystem-level topics, it's time to start exploring how filesystems are implemented in the Linux kernel. The next chapter starts to explore the Linux source code, providing tools for navigating around very large code base and describing where the filesystem-related code is. Subsequent chapters then start to dig deeper into kernel operations culminating in developing of a kernel-based filesystem and multiple user-space filesystems.

Chapter 4

The Linux Kernel Source Code

There are rather a lot of lines of code (LOC) code in the Linux kernel source. Running the wonderful `cloc(1)` command on the `linux-6.3.3` kernel source tree, it shows that there are 69,167 unique files and over 22 million lines of C code. For all files, there are close to 26 million LOC. Yes, the Linux kernel is the biggest open source projects of all time and the rest of the operating system isn't included in this count. To give you can idea of how much things have changed, in the first Linux kernel, there were only 89 files and 8.397 LOC. When I was porting the SVR4 UNIX command `truss(1)` command (similar to Linux `strace(1)`) to the Chorus microkernel in 1995, that single command was around 11,000 LOC. It's quite amazing how much more complicated software has become.

Here is the output displayed when running `cloc(1)` on the 6.3.3 kernel:

```
$ cloc linux-6.3.3
 80363 text files.
 69167 unique files.
```

Language	files	blank	comment	code
C	31526	3190215	2531300	16348237
C/C++ Header	22854	651061	1258767	6309125
TeX	100	328722	2099	1497694
reStructuredText	3176	154913	67094	422577
Assembly	1307	47039	100449	226075
YAML	2694	48453	12358	221305
JSON	440	2	0	207449
Text	2185	31876	0	144305
Bourne Shell	838	25266	17044	98386
make	2735	10536	11609	48180
SVG	66	81	1162	41851
Perl	66	7362	5024	36442
Python	143	6154	5817	30771
D	92	0	0	27692
DOS Batch	772	2764	1	17586

yacc	9	698	409	4905
PO File	5	791	918	3077
lex	9	346	309	2108
C++	10	372	138	2016
Bourne Again Shell	54	354	314	1474
awk	13	217	157	1323
Glade	1	58	0	603
CSV	7	73	0	559
NAnt script	2	148	0	510
Cucumber	1	30	50	187
TNSDL	2	33	0	140
Clojure	44	5	0	134
Windows Module Definition	2	15	0	109
m4	1	15	1	95
CSS	2	35	37	90
XSLT	5	13	26	61
MATLAB	1	17	37	35
Umka	1	16	0	34
vim script	1	3	12	27
Ruby	1	4	0	25
INI	1	1	0	6
sed	1	2	5	5
<hr/>				
SUM:	69167	4507690	4015137	25695198
<hr/>				

Apart from the Linux kernel, source code for the rest of the Linux operating system has traditionally been stored in many different places and pulled together to form one of the many Linux distributions (Ubuntu, Red Hat, SuSE, Debian etc). Today much of the code for the Linux operating system can be found in various github.com locations. The kernel source however, has been accessible from www.kernel.org for many years. The home screen for this website is shown in figure 4.1. **XXX — will need updating towards the end – doesn't match above**

On the kernel.org home page there is a link to <https://docs.kernel.org>, which contains documentation covering everything from building the kernel to submitting patches to building loadable modules. There is a wealth of information here, although some of the documentation is more detailed than others and all vary as to when they were written. Don't take everything there as gospel. The source code is always has the final say.

This chapter won't cover all of the kernel source code since only a small subset is relevant to filesystem development. However, the goal of this section is to give you an overview of what everything is and where different subsystems can be found. Starting at the top level of the kernel source tree, these are the directories that you will see:

- Documentation – there are 8,579 files in this directory and subdirectories at the time of writing (the 6.3.3 kernel). Some of the documentation is quite old but it's still a good place to start in order to find information. You will see a lot of .rst files (reStructuredText – a simple markup language). You can build the documentation yourself although I found that a lot of disk space is needed to pull all the

The Linux Kernel Archives



[About](#) [Contact us](#) [FAQ](#) [Releases](#) [Signatures](#) [Site news](#)

Protocol	Location
HTTP	https://www.kernel.org/pub/
GIT	https://git.kernel.org/
RSYNC	rsync://rsync.kernel.org/pub/

Latest Release

6.1.4



mainline:	6.2-rc3	2023-01-08	[tarball]	[patch]	[inc. patch]	[view diff]	[browse]
stable:	6.1.4	2023-01-07	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]
stable:	6.0.18	2023-01-07	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]
longterm:	5.15.86	2022-12-31	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]
longterm:	5.10.162	2023-01-04	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]
longterm:	5.4.228	2022-12-19	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]
longterm:	4.19.269	2022-12-14	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]
longterm:	4.14.302	2022-12-14	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]
longterm:	4.9.337 [EOL]	2023-01-07	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]
linux-next:	next-20230110	2023-01-10					[browse]

Figure 4.1: www.kernel.org — home of the Linux source code

necessary packages to build it. Alternatively, and to simplify things, you can go to www.kernel.org/doc/html, locate the kernel of choice and view the same documentation there.

- **Makefile** – the top-level makefile which is used when building the kernel. This book doesn't describe much about kernel builds other than when discussing kgdb in section 9.1. For detailed information about building the kernel, submitting patches and so on, refer to <https://docs.kernel.org> or search for information specific to the distribution that you are using.
- **arch** – architecture-specific source code for the just over twenty different platforms that Linux supports / has supported. You will see a directory for each different architecture, for example, x86, arm64 and s390. System call entry points can be found under `arch/x86/entry/syscalls`. Platform independent system call handling will be described below in section 4.2. Platform initialization code (assembler and C) as well as lower-level memory management and interrupt handling code can be found within these directories.
- **block** – generic code for the kernel *block layer*. To see source code for each of the block device drivers, refer to the directories under `drivers/block`.
- **certs** – kernel module signing allows for cryptographically signing modules during installation and then checking these signatures when modules are loaded. This

increases kernel security by disallowing the loading of unsigned modules or modules signed with an invalid key.

- `crypto` – an array of cryptographic functions that can be used by the kernel and kernel modules. There are also crypto routines in `drivers/crypto` and architecture specific crypto routines. For example, you can find Intel AES-NI crypto code in `arch/x86/crypto`.
- `drivers` – contains the majority of the device driver code in driver-specific directories. As an example, you will see `usb` and `bluetooth` directories. Some directories contain source code that is generic for the hardware type (`drivers/scsi` for example) together with drivers for specific hardware cards.
- `fs` – filesystem independent (VFS – Virtual File System) and filesystem dependent code. Files at the top level of this directory are filesystem dependent. Each filesystem has its own directory, for example, `fs/ext4` and `fs/xfs`. This is the main place to look at when studying filesystems.
- `include` – header files and almost 6,000 of them. These are not the only header files in the kernel source. A quick "find . -name '*.h'" will show header files scattered throughout the source tree. Most of the header files in this directory are used by generic kernel subsystems or drivers and filesystems.
- `init` – a small amount of code that is executed during startup of the Linux kernel. Look for the function `start_kernel()` in `init/main.c` if you wish to study some of the early code paths. This function performs early initialization tasks including creating a kernel thread which ultimately becomes `/sbin/init`.
- `ipc` – System V IPC (Inter-Process Communication), source code for handling shared memory segments, semaphores and message queues.
- `kernel` – 80,000+ lines of kernel code that handles everything from process management to signal handling to reboot to system call handling (after entry through architecture-dependent code in `arch`). Most file names give a hint as to what the source code inside does.
- `lib` – generic routines that are of use to all kernel subsystems can be found here. There is everything from crypto and string operations to debugging routines and linked list management.
- `mm` – memory management which will be described later in the book in some detail since there is a lot of interaction between the virtual file subsystem and the memory management subsystem. For memory management code that's architecture-specific, see `arch/<platform>/mm`.
- `net` — everything networking related.
- `samples` – various sample pieces of code. For example, inside the `eBPF` directory there are tests for eBPF and files under `kobject` contains code to create a simple subdirectory in `sysfs`.

- `scripts` – there are a lot of makefiles and scripts. One such example is the directory `scripts/gdb/linux` which contains Python helper scripts which will be described in section 9.1.5 and used in the many `gdb` examples throughout the book.
- `security` – security-related source code including support functions for SELinux and AppArmor.
- `sound` – all things sound related including drivers and the *Advanced Linux Sound Architecture* (ALSA) which provides audio and MIDI functionality.
- `tools` – an array of different tools for use with x86 CPU power management, thermal monitoring, eBPF, tracing and PCI support.
- `usr` – source code which builds a `cpio` archive containing a temporary root filesystem image (referred to as the `initramfs` image). This is used for an early user-space file hierarchy on to which the real root filesystem will be mounted early during system initialization.
- `virt` – source code for handling virtualization, specifically KVM (Kernel Virtual Machine). This is located under `virt/kvm`.

It can be quite daunting to look at this vast amount of source code but to analyze filesystem and filesystem-related code, you only need to look at a relatively small number of files in a few directories. The following sections will help guide you as to where these files are and also describe a set of good tools to use to help with browsing the source code.

4.1 User-Space vs Kernel-Space

Before digging further into the kernel source, this is a good time to describe the difference between user-space and kernel-space. The latter is also called "*running in the kernel*" or "*operating in kernel-mode*".

Each Linux process runs in its own address space. The process address space is a protected range of memory addresses in which a Linux process runs supported by underlying hardware mechanisms such that one process is unable to access memory addresses of another. Figure 4.2 shows the layout of two different Linux processes in memory. The kernel is *mapped* into the address spaces of all process and at the same location. All memory addresses in the kernel-space are not accessible by the program that is running in user-space. Any attempt is made to access a kernel address will result in a *segmentation violation*. If this happens, the process will receive a `SIGSEGV` signal and terminate. A process can only enter the kernel voluntarily by invoking one of the many system calls which the kernel will then service.

Everything that is not in kernel-space is considered to be in user-space. This includes the program TEXT (the program's executable instructions) which is at the lower end of memory. Arguments passed to the program and the program's environment variables are at the higher portion of user-space. The stack grows down in memory as more functions are called and any data allocated via `malloc(2)` is taken from the heap which grows up

towards the stack. It is from the heap that most file I/O will occur since processes will generally allocate memory buffers dynamically.

Each time a call is made to fork a new process, the new process is started in its own address space. A call is then typically made to `execve(2)` to execute a new binary (typically via one of the `exec(3)` family of functions). For the differences between the parent process and the child process following a fork, refer to the `fork(2)` manpage.

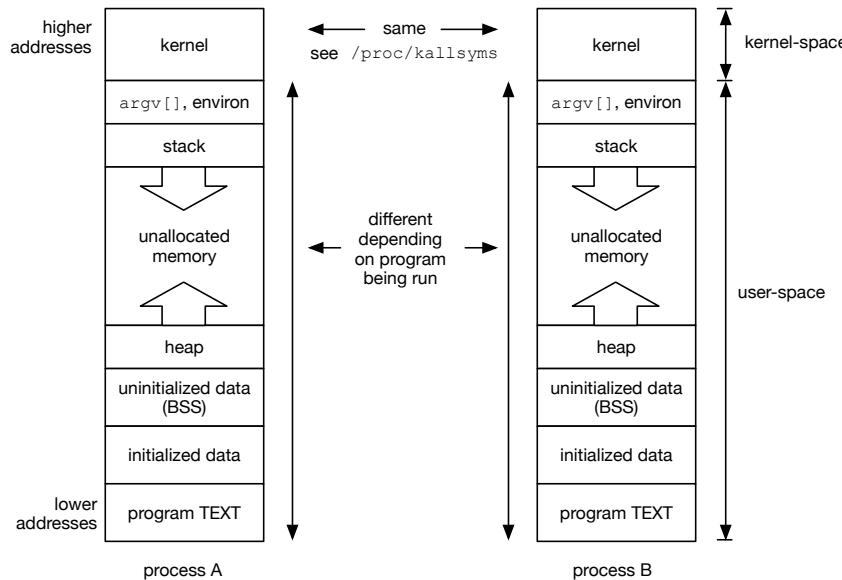


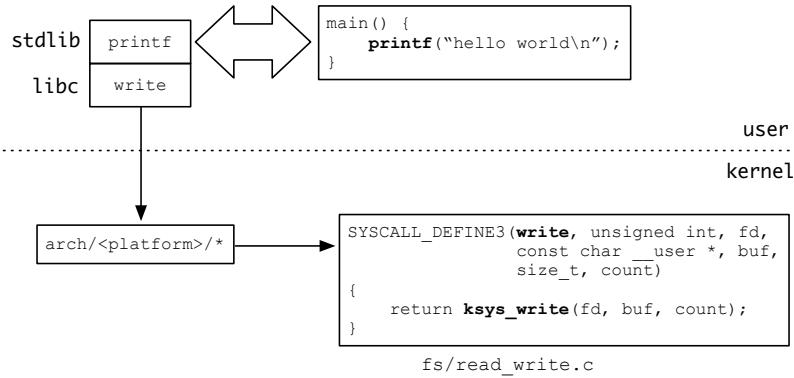
Figure 4.2: Process address space showing user-space and kernel-space

It would be fun to explore the process address space here in more detail but that's a topic for another book or blog post. In short, just recognize that each process has its own, unique address space that is separate from all other processes, the kernel is mapped into all address spaces, user programs cannot access kernel addresses directly but can enter the kernel intentionally by invoking a system call.

4.2 The System Call Interface

This section shows how applications enter the kernel through invoking *system calls*. The goal is to give readers a good place to start exploring execution paths through the kernel in response to familiar system calls that a program invokes such as `open(2)`, `read(2)` and `stat(2)`.

A *system call handler* is nothing more than a C function that operates in kernel space and calls other functions as needed. But the means of getting to this C function is somewhat complex and depends on the architecture on which the program is running. Consider figure

Figure 4.3: Entering the kernel through the `write(2)` system call

4.3 which shows the path from a everyone's favorite "*Hello world!*" C program which calls `printf(3)` which ends up in a call to the `ksys_write()` system call handler in the kernel. The program calls the standard library routine `printf(3)` which in turn will invoke the `write(2)` system call in order to write the string "hello world\n" to `stdout`.

The details of how system calls are handled on each architecture won't be described in detail here but what follows are the steps taken on the `x86_64` architecture by the standard library (`libc`) to enter the kernel for our "`printf("Hello world!\\n");`" program. Arguments are passed into the kernel via CPU registers. If there are more than six arguments, the remaining arguments as pushed onto the stack.

- Store "1" in `%rax` ("1" is the system call number for the "write" system call)
- Store argument 1 in `%rdi` (in this case it will be "1" for `stdout`)
- Store argument 2 in `%rsi` (a pointer to the string "hello world\n")
- Store argument 3 in `%rdx` (the length of the string "hello world\n")
- Call the `x86_64` instruction "syscall" to enter the kernel

For the Intel architecture, in the directory `arch/x86/entry/syscalls`, there are two files which contain system call tables for both the 32-bit and 64-bit architectures. These files are `syscall_32.tbl` and `syscall_64.tbl` respectively. Here are the first few lines of `syscall_32.tbl`:

0	i386	restart_syscall	sys_restart_syscall
1	i386	exit	sys_exit
2	i386	fork	sys_fork
3	i386	read	sys_read
4	i386	write	sys_write

You can see `sys_write` has the value of "4" which is different to the `x86_64` example above. On contrast, here are the first few lines of `syscall_64.tbl` for which the `write(2)` system call is at position "1":

0	common	read	sys_read
1	common	write	sys_write
2	common	open	sys_open
3	common	close	sys_close
4	common	stat	sys_newstat

For further details on how system calls on x86 are handled I'd recommend reading the *0xax* article "*System calls in the Linux kernel. Part 1.*":



URL 15 <https://tinyurl.com/rb3z8kds>

which explains how system calls are handled on the x86 platform in great detail. It shows an assembler version of "*Hello world*", describing the path through to the architecture-specific areas of the kernel and into the C system call handling functions.

4.2.1 System Calls for File / Filesystem Activity

There are a lot of system calls, in fact 322 for `x86_64` and 358 for x86. How many of these are related to filesystem activity? File / filesystem related system calls can be found in the `fs` directory at the top level of the Linux kernel source tree. To get a rough idea of the number of file / filesystem related system calls, head into the `fs` directory and run the following:

```
$ grep SYSCALL_DEFINE3 *.c | wc -l  
56
```

The actual number is a little less but this should help you get an idea of how many file / filesystem operations the kernel supports for applications.

Generally speaking, the easiest way to find the specific functions is to run "grep" and you'll see that the file/filesystem functions are spread across 16 files. Here are some examples where grep is run to look for specific function calls.

```
$ grep 'SYSCALL_DEFINE3(write)' *.c  
read_write.c:SYSCALL_DEFINE3(write, unsigned int, fd, ...  
read_write.c:SYSCALL_DEFINE3(writev, unsigned long, fd, ...  
$ grep 'SYSCALL_DEFINE3(read)' *.c  
read_write.c:SYSCALL_DEFINE3(read, unsigned int, fd, ...  
read_write.c:SYSCALL_DEFINE3(readv, unsigned long, fd, ...  
stat.c:SYSCALL_DEFINE3(readlink, const char __user *, path, ...  
$ grep 'SYSCALL_DEFINE3(fcntl)' *.c  
fcntl.c:SYSCALL_DEFINE3(fcntl, unsigned int, fd, ...  
fcntl.c:SYSCALL_DEFINE3(fcnt164, unsigned int, fd, ...  
fcntl.c:COMPAT_SYSCALL_DEFINE3(fcnt164, ...  
fcntl.c:COMPAT_SYSCALL_DEFINE3(fcntl, unsigned int, fd, ...
```

Most of these top-level functions don't do a great deal other than call other kernel functions. For example, in the case of the `write(2)` system call, a call is made to `ksys_write()` as follows:

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *,  
                 buf, size_t, count)  
{  
    return ksys_write(fd, buf, count);  
}
```

There are 76 source code files in the `fs` directory and over 90,000 LOC but the names of the files generally give a good hint as to what functionality they provide. For example:

- `open.c` – contains source code for the `open(2)` system call and other operations generally associated with opening a file such as `ftruncate(2)` and the different file ownership change system calls (`chown(2)`, `lchown(2)` etc),
- `read_write.c` – the code for `read(2)`, `write(2)` and vectored read / write system calls.
- `readdir.c` – system calls for reading directories including `getdents64(2)`.

There are several ways to browse through the kernel source but the best way will depend on your own personal preferences. Some people are happy with "cd" and "grep" but there are several tools that can help or at least give different options. Three such tools will be described in the following sections.

4.2.2 Using `vim` and `ctags` to Browse the Kernel Source

I remember the first time a colleague looked at me in surprise and said "*You don't use tag stacking?*". I learned very quickly and have been using it for 30 years now. Most editors can make use of `ctags(1)` but the program was originally built several decades ago to be used with `vi(1)`. What are ctags? From the manpage:

The ctags and etags programs (hereinafter collectively referred to as ctags, except where distinguished) generate an index (or "tag") file for a variety of language objects found in file(s). This tag file allows these items to be quickly and easily located by a text editor or other utility. A "tag" signifies a language object for which an index entry is available (or, alternatively, the index entry created for that object).

The following example generates a tags file for the whole of the Linux kernel source code. To do this, simply head to the top level directory of the source code and run the "`ctags`" command:

```
$ cd linux-6.3.3  
$ ctags -R *
```

This will create a very large file (for the 6.3.3 sources it's 1,067,091,840 bytes - ouch!).

Once you have the tags file, navigate to the top level Linux source tree and enter your favorite editor using the option to specify a tag. In `vi` pass the `-t` option together with the function or global variable that you're looking for. The following command will open `vi` in `fs/read_write.c` with `ksys_write()` in the center of the screen.

```
$ vi -t ksys_write
```

Note that you need to be in the directory where the tags file is located. You can also search for global variables too so:

```
$ vi -t file_systems
```

will open `vi` in `fs/filesystems.c` at the place where the `file_systems` global variable is defined.

I typically have this one-liner in my `.bashrc` file:

```
alias linux='cd ~/src/linux-6.3.3 ; pwd'
```

to get me to the top of the current source tree that I'm using. I can then run "`vi -t`" to enter the file at the place I want.

Another option is to build a little script that displays various routines. The following script switches to the top of the kernel source tree where it assumes you have a tags file and then enters `vi` using a tag that you choose. The advantage here is that once inside the file, you can now use tag stacking to move through the kernel source from one function to another.

Here is a shorter version of the bash script. It's very simple and obviously you can put any function or global variable in here.

```
TAGS_DIR=~/src/linux-6.3.3

echo "
1 - open
2 - read
3 - write
4 - readlink

a - task_struct
b - super_operations

q - quit
"

cd $TAGS_DIR
while :
do
    /bin/echo -n "System call? "
    read choice
    case $choice in
        1) vi -t do_sys_open ;;
        2) vi -t ksys_read ;;
    esac
done
```

```

ssize_t ksys_write(unsigned int fd, const char __user *buf, size_t count)
{
    struct fd f = fdget_pos(fd);
    ssize_t ret = -EBADF;

    if (f.file) {
        loff_t pos, *ppos = file_ppos(f.file);
        if (ppos) {
            pos = *ppos;
            ppos = &pos;
        }
        ret = vfs_write(f.file, buf, count, ppos);
        if (ret >= 0 && ppos)
            f.file->f_pos = pos;
        fdput_pos(f);
    }

    return ret;
}

```

Figure 4.4: Using tag-stacking to navigate through the kernel source code

```

3) vi -t ksys_write ;;
4) vi -t do_readlinkat ;;
a) vi -t task_struct ;;
b) vi -t super_operations ;;
q) break ;;
*) echo "Invalid choice"
esac
done

```

Now that you can get in to a file at the location of the system call handler you've been looking for, it's time to cover how tag stacking works so you can search through the code from one function to the next. Regardless of where you are in the file, if you want to go to the location of the tag where the cursor points to and then back again to the current file and location, use the following two commands in order:

- **Ctrl +]** – Move to the file where the tag under cursor is located (function, global variable etc). This is identical to running "vi -t" specifying the tag name from the command line.
- **Ctrl + t** – Move back to the previous file/position.

For example, in figure 4.4 we entered this file with "vi -t ksys_write" and the cursor is on the function name `vfs_read()`. By hitting "Ctrl +]", vi will take you to the location of `vfs_read()`. It happens to be in the same file but if it was a different file, you'd enter that file instead. If you then hit "Ctrl + t" you will go back to the file/location where you started.

The current tag stack is the list of all places in the source tree you have moved to through repeated calls to "Ctrl +]". You can view the current tag stack by typing ":tags" in

vi. See figure 4.5 for an example. This shows going from `vi -t ksys_write` to `vfs_read()` and then to `rw_verify_area()`.

```
static int warn_unsupported(struct file *file, const char *op)
:tags
# TO tag      FROM line  in file/text
1 1 ksys_write      1 // SPDX-License-Identifier: GPL-2.0
2 1 vfs_write       644 ret = vfs_write(f.file, buf, count, ppos);
3 1 rw_verify_area  582 ret = rw_verify_area(WRITE, file, pos, count);
>
Press ENTER or type command to continue
```

Figure 4.5: Current tag stack

I've tried some of the newer editors and have struggled to move around files as quickly as I can with `vi` and tag stacking.

4.2.3 Using cscope to Browse the Kernel Source

Another excellent tool to use, especially in conjunction with `vi` and tag stacking, is the `cscope(1)` command which has been around since the early 1980s. It's a tool I've used for over thirty years and have used extensively while writing this book. Notes about the early history of `cscope` can be found here:



URL 16 – <https://tinyurl.com/ywce8h9n>

I particularly enjoyed reading the goals behind starting the project:

Joe Steffen first started writing cscope in the early 1980's as an aid for his own work. It started as little more than a set of shell scripts containing greps and sed's. It became clear to Joe that this was not going to work on large projects (say, more than 20 or so files, which was a large project on a PDP-11!) as most of the time consumed was in parsing the C source code over and over again. So he wrote a C program that did the grunt work of parsing into a tagged database, and then presented the screen with common searches. The tagged database was updatable and saved between sessions. Productivity soared!

Imagine that – a project with 20 or so files? Today, the Linux kernel source tree now has close to 26 million files. Of course `grep` is still very useful so some things don't change.

History lesson beside, the first time you run `cscope` you will need to generate the cross-reference database as shown below. The command must be run from the top kernel source directory:

```
$ cd linux-6.3.3
$ cscope -Rk *
```

Since the kernel source is quite large, this takes quite a while and the database comes in at over just over 1 GB. Thus you'll need over 2 GB of free space for using "cscope" and "ctags". The "R" option instructs crash to recurse through all directories and the "k" option (Kernel Mode) will turn off using the default include directory (/usr/include) when building the database, since when building the kernel, these include files won't be used. Only header files within the kernel source tree are used since header files in /usr/include will likely be for a different kernel release. In the Linux source tree, the include directory used during kernel builds is at the top level of the source tree.

Once the database has been generated or on subsequent runs, you will see output similar to what is shown in figure 4.6. The cscope version will be displayed at the top of the screen together with a note to let you know that you can press "?" at any time to get help:

```
Find this C symbol: █
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Find assignments to this symbol:
```

Figure 4.6: The main cscope screen

Everything is set up such that the cursor is on the line where you can search for a symbol so all you have to do is enter the symbol you are looking for and press the return key. For example, after typing "ksys_read" and pressing "RETURN", the results will be displayed at the top of the screen as shown in figure 4.7.

For each search item displayed you can press the number key shown on the left-hand-side to enter the file at that location. In this example, press '2' to enter `read_write.c` at the location where the function `ksys_read()` is located. Or you can use the arrow keys to move between items displayed and just press "RETURN". If there are more items than can be displayed on a single screen, use the space bar to go to subsequent screens. The "TAB" key toggles between the search results and the menu at the bottom of the screen.

C symbol: ksys_read		
File	Function	Line
█ syscalls.h	<global>	1289 ssize_t ksys_read(unsigned int fd, char __user *buf, size_t count);
1 compat_linux.c	COMPAT_SYSCALL_DEFINE3	230 return ksys_read(fd, buf, count);
2 read_write.c	ksys_read	609 ssize_t ksys_read(unsigned int fd, char __user *buf, size_t count)
3 read_write.c	SYSCALL_DEFINE3	630 return ksys_read(fd, buf, count);

Figure 4.7: The cscope results of searching for `ksys_read`

"Ctrl + d" exits cscope.

While in menu mode, if you use the arrow keys to move down to "Find this global definition", type `ksys_read` and press the return key, `cscope` will locate the symbol and enter the file at the correct location. If you are looking for a function that is called by many other functions, the list of hits at the top of the screen could be several pages long. Searching for the global definition can save time searching.

Next time you run `cscope` just add the `-d` option to instruct it not to regenerate the database again. Unless of course some of the source code files have changed. If the database needs to be generated again, it should be faster than the first time it's generated.

Now the best part of using `cscope` is to combine it with your favorite editor and use tag stacking. Then you have the best of all worlds since you can search easily, locate the function of your choice and move down through functions it calls.

4.2.4 Using Elixir to Browse the Kernel Source

Elixir is a cross reference tool for the Linux source code hosted by embedded Linux provider Bootlin. It doesn't just support a specific version of the kernel but every version that exists all the way back to day one and up to the most recent development kernels. Everyone has their own way of navigating through source code using their favorite editor and using tools such as `ctags` but sometimes it's nice just scroll through a page of source code using the mouse. You can find Elixir by pointing the browser to elixir.bootlin.com to get to the latest sources or using the following link. Alternatively just search for "elixir linux" in your favorite browser.



17 <https://tinyurl.com/447v7urs>

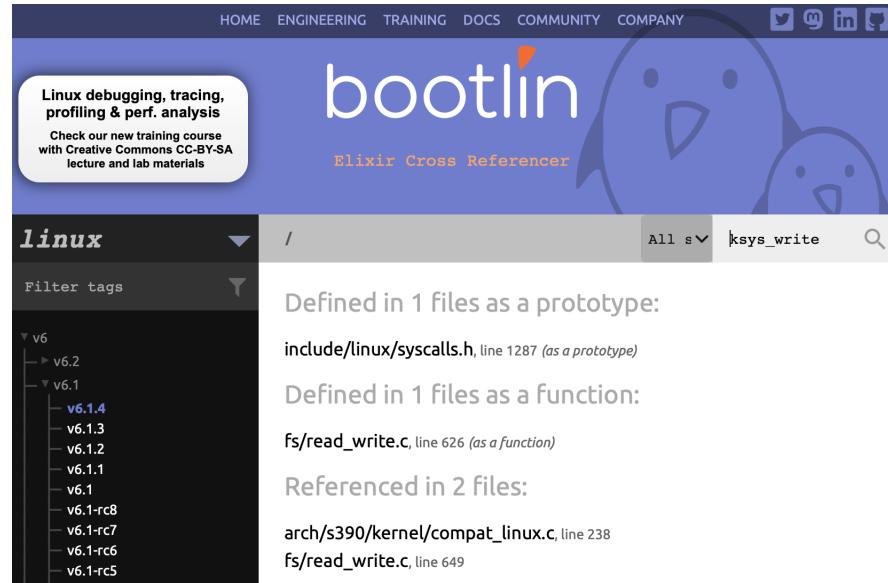
Figure 4.8 shows an example of using Elixir and displaying the results of searching for the symbol `ksys_write`. In this example, we are searching through the 6.1.4 source code tree. You can change which Linux kernel version to search by selecting the version on the left hand side of the screen.

Figure 4.9 shows the source code for our symbol. As well as typing the name of something to search for in the *All symbols* search bar you can also click on function names and global variables when the source code is being displayed.

You can also use Elixir on other source code projects including `glibc`. Just click the arrow on the right hand side of "Linux" towards the top left of the page and a number of other projects will be displayed.

4.2.5 Maintaining Your Own Notes

Of course you don't need to rely on all of the tools shown here for browsing the kernel source. You will develop your own methods of browsing the kernel source over time based on your preferences. How you keep notes is also a matter of personal choice. I have a

Figure 4.8: Elixir search for the function `ksys_write`

The screenshot shows the bootlin Elixir interface displaying the source code for the `ksys_write` function in `fs/read_write.c`. The code is as follows:

```

620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645

SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
{
    return ksys_read(fd, buf, count);
}

ssize_t ksys_write(unsigned int fd, const char __user *buf, size_t count)
{
    struct fd f = fdget_pos(fd);
    ssize_t ret = -EBADF;

    if (f.file) {
        loff_t pos, *ppos = file_ppos(f.file);
        if (ppos) {
            pos = *ppos;
            ppos = &pos;
        }
        ret = vfs_write(f.file, buf, count, ppos);
        if (ret >= 0 && ppos)
            f.file->f_pos = pos;
        fdput_pos(f);
    }
    return ret;
}

```

The code is annotated with line numbers from 620 to 645. The `ssize_t ksys_write` function is highlighted.

Figure 4.9: Elixir displaying the source code for the selected function

directory of simple text files that I use `vi` to add notes. You may wish to maintain one using rich-text or some other format.

I also have a file called "structures" which contains a copy of all of the Linux kernel structures that I typically look at. Even while browsing with `vi` or Elixir, I'm usually following a code path and like to have the file close by for easy searching. It looks like this:

```
-----  
mount - fs/mount.h  
  
struct mount {  
    struct hlist_node      mnt_hash;  
    struct mount          *mnt_parent;  
    struct dentry         *mnt_mountpoint;  
    struct vfsmount        mnt;  
    ...  
  
-----  
vfsmount - include/linux/mount.h  
  
struct vfsmount {  
    struct dentry          *mnt_root;  
    struct super_block     *mnt_sb;  
    ...  
}
```

I've even formatted the structures to make it visibly easy to read since most of the Linux header files are full of compiler directives and `#ifdefs` which make the code harder to read. Searching for structures is also easy. I use `"/^vfsmount"` to search for "vfsmount" on the first line and that takes me right to the structure I'm looking for. You can do the same in your editor of choice.

I also use Omnigraffle on my Macbook Pro and have drawn a lot of code walkthrough figures, some of which can be found throughout the book. Perhaps pencil and paper or a tablet may be your choice of tool.

4.3 File Access Structures

Now that you know where the filesystem code is located in the Linux kernel source tree and have some tools to navigate through the relevant filesystem system calls, we'll switch our focus to the main structures that are used for file / filesystem access. These structures are defined in the `include` directory at the top of the Linux kernel source tree. The following sections will reference the relevant fields of each structure as each structure is explained but not all of the fields of each structure will be described. For some structures, there are many fields and describing them all here will just be overload and will inhibit understanding how they are used. Other fields will be introduced as needed throughout subsequent chapters.

To get started, consider figure 4.10 which shows how the main structures in the kernel used for accessing files are linked together. It all starts with the `task_struct` structure,

for which there is one such structure per process.

In this figure there are some structures that are per-process only and some that are shared between all processes as indicated by the dotted line. For example, for every open file, each process needs its own file offset for reading and writing together with specific flags that it has set during an open call. This is where the `file` struct comes into play. However, there may be multiple processes accessing the same file simultaneously so, although each process has distinct `file` structures, they all share a common `inode` structure for the file in question.

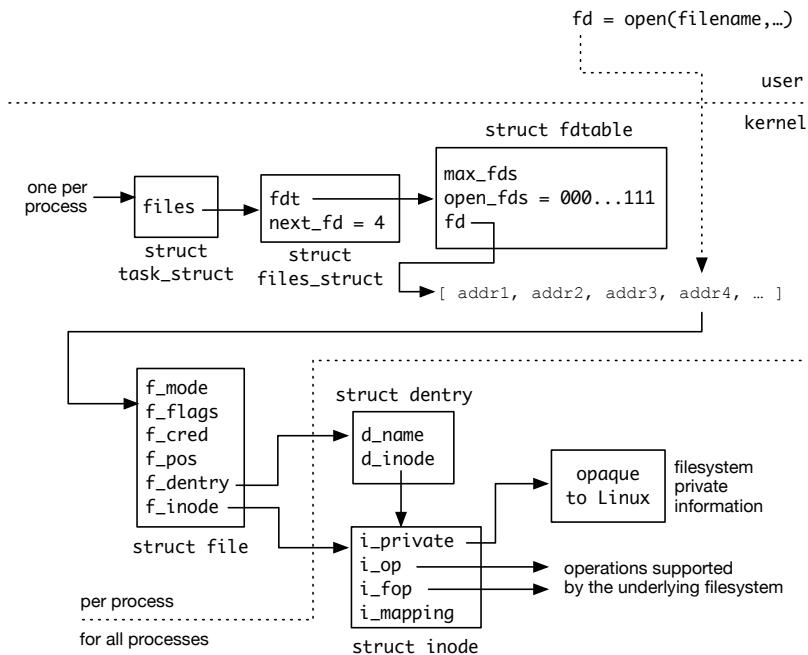


Figure 4.10: Kernel structures for accessing open files

4.3.1 File Descriptors

File access starts with a file descriptor that a process gets back from `open(2)` and friends. In older versions of UNIX, the file descriptor was an index into an array of pointers to "struct `file`" elements held within the `u_area` structure which in turn referenced a `proc` structure, similar to the Linux `task_struct`. This allowed for a fixed number of open files, a restriction that continued through to SVR3 UNIX into the early 1990s. At SCO, although we had an SVR3-based version of UNIX at the time, the limit was removed to allow for a more dynamic (and tunable) number of open files.

This hard limit was similar in the first version of Linux which had the following field

in the `task_struct` structure:

```
struct file * filp[NR_OPEN];
```

with `NR_OPEN` defined as 20. This meant that there was a hard coded limit of 20 open files. For most processes that's still reasonable since in a standard Ubuntu running instance, there are a relatively small number of processes with over 20 open files.

Using a file descriptor as an index into an array pointing to `file` structures was the reason that `stdin`, `stdout` and `stderr` originally had the values of 0, 1 and 2 respectively. These numbers were used to access the first 3 elements of the file descriptor array. Regardless of how an operating system may implement POSIX file access, this convention remains intact today.

Handling file descriptors in Linux is similar today although the implementation has become more complex when handling large numbers of open files, the details of which will be described later in section 6.7.1. To get started, there are three fields in the `task_struct` which have relevance to files, filesystems and namespaces:

```
struct task_struct {
    struct fs_struct     *fs;           /* Filesystem information: */
    ...
    struct files_struct *files;        /* Open file information: */
    ...
    struct nsproxy      *nsproxy;      /* Namespaces: */
}
```

Filesystems and namespaces will be described later but for file access, the `files` field is a pointer to a `files_struct` structure for which the most relevant fields are:

```
struct files_struct {
    struct fdtable __rcu *fdt;
    unsigned int          next_fd;
    ...
}
```

The `next_fd` field contains the number of the next file descriptor that will be allocated. For a new process this will be 3 which is 1 above `stderr`. The `fdt` field points to an `fdtable` structure.

4.3.2 The `fdtable` Structure

```
struct fdtable {
    unsigned int          max_fds;
    struct file __rcu **fd;           /* current fd array */
    unsigned long         *close_on_exec;
    unsigned long         *open_fds;
    unsigned long         *full_fds_bits;
};
```

Given a file descriptor, this is an efficient way to get to the `file` structure in most instances. Just index into the `fd` array. — need to look to see how it works. If the fd

is small and there is only one table, it's easy. If it's a larger number, can it uses the current fd array or not? Need to look. Therefore Linux has made it efficient to handle a small number of open files while still being extensible to support a much larger number.

The `max_fds` field is set to 256 but this is just the number of file descriptors that a process can have initially and, as discussed, for most processes, this will suffice. We'll cover how greater numbers of file descriptors are managed later in section 6.7.1. It's become somewhat messy in Linux today but involves allocation of multiple `fdtable` structures. **XXX – reword all this once that work is complete**

There is one file descriptor per open instance which results from a call to `open(2)`, `creat(2)` for example. The `file` structure however can be shared by multiple open instances (more than one file descriptor). This is achieved by calling `dup(2)` which will return a new file descriptor. In this case there will be two entries in the file descriptor array (one per file descriptor) but both will point to the same `file` structure.

4.3.3 The `file` Structure

The `file` structure has some fields that correspond to the arguments to `open(2)` call such as `f_flags` and `f_mode` as well as subsequent operations such as `read(2)`, `write(2)` and `lseek(2)`: **XXX – check on flags and mode when going through open code**

```
struct file {
    struct path          f_path;
    struct inode         f_inode;
    atomic_long_t        f_count;
    unsigned int         f_flags;
    fmode_t              f_mode;
    loff_t               f_pos;
    struct fown_struct  f_owner;
    const struct cred   f_cred;
    ...
}
```

Exploring these fields:

- `f_path` – contains a pointer to the dcache entry for this file as well as the filesystem to which this file belongs. The dcache allows for fast lookups to get to a file when system calls such as `open(2)` pass a pathname to search. To access the file's name, the field `f_path.dentry->d_name.name` can be used. The dcache will be described below in section 4.4.
- `f_inode` – the corresponding Linux inode for the file. There may be multiple references to a file within a single process or from multiple processes but there will only be one Linux inode. Inodes are described in section ??.
- `f_count` – a reference count for the number of file descriptors that reference this `file` structure. If we open a file for the first time, `f_count` will be 1. If we then call `dup(2)`, we have two file descriptors referencing this same `file` structure so the reference count will be 2. We'll demonstrate reference counts later on.

- `f_flags` – the open flags which were passed to `open(2)` and other system calls.
- `f_mode` – **this is essentially the mode argument.**
- `f_pos` – The position within the file that will be used for the next `read(2)` or `write(2)` system call. This field can also be changed by a call to `lseek(2)`.
- `f_owner` – **XXX—TBD**
- `f_cred` – **the process user credentials for this file open**

How and when these fields are changed will be described in more detail in the section 6.7 when describing the flow through various system calls.



Most of the structures described in this section can be found in the header file
`include/linux/fs.h`

You will see references to this header file throughout the book. It contains many structures that are related to file and filesystem access.

4.3.4 The `inode` Structure

Section 4.5 describes the inode cache and the relationship between in-core inodes and disk-based inodes. To complete this section in terms of how the main structures all link together, an *inode* is the in-core representation of an open file. For each file that is open, regardless of whether it is on local disk, RAM or over the network, there will be a Linux inode in-core represented by the `inode` structure.

As file I/O takes place for regular files, the inode references a *page cache mapping* which in turn references cached pages in memory. This is shown in figure 4.11.

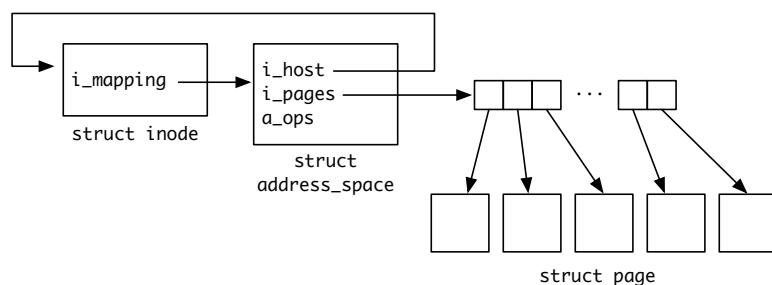


Figure 4.11: The Linux `inode` which references data cached in memory

Although there may be multiple `file` structures in memory representing different open instantiations of the same file from different processes, there will only be one `inode` struc-

ture which is shared across all processes. File I/O and locking together with the internal operation of the inode cache will be described in section `vfs-inode-cache`.

4.4 The Directory Cache

Consider the simple command:

```
$ ls /home/spate/spfs/kern/spfs.ko
```

To perform this operation, called *pathname resolution*, the `ls(1)` command calls the `statx(2)` system call passing the pathname supplied as an argument. When parsing this pathname, the kernel has to process five different directories starting with "/" and ending with "kern". As the path is being processed, it must validate that each directory in the path not only exists but that the caller has permission to access the directory. A permissions check is also performed on the base file. In this example, information about the regular file "spfs.ko" is returned through a `stat` structure. For an `open(2)` call, the kernel finishes by setting up the structures described in the previous section and returning a file descriptor.

Each component of the pathname is represented by an *inode* in a local filesystem or other entity over the network for filesystems such as NFS. Every time a call is made to the access a directory inside the filesystem, there can be many calls to read structures from disk which is a time-consuming operation. Accessing the same files happens frequently therefore minimizing disk access is imperative to improving the performance of the system. This is where the Linux *dcache* (directory cache) comes into play. The *dcache* is a variation of the *Directory Name Lookup Cache* (DNLC) which has been around for several decades now and was first introduced in BSD UNIX to solve the the *pathname resolution* problem.

The Linux directory cache (simply called the *dcache*) has the same goal as the original DNLC which is to provide a fast and efficient method of mapping any file name to a Linux inode regardless of the type of file. While resolving a pathname component for the first time, the kernel will ask the filesystem to lookup a filename within a specific directory. Once the filesystem locates this file, a reference to it can be stored in the *dcache* so that a subsequent lookup does not need to go to disk. Over the time, for frequently accessed files, the goal is to read from the *dcache* as much as possible. It is generally possible to get a *dcache* hit ratio of 80% or higher.

4.4.1 Introducing The `dirent` Structure

The main structure that underpins the *dcache* is the `dirent` (directory entry). *Dcache* entries are searched using the parent directory `dirent` and a string for the file that is being searched for.



The `dirent` structure can be found in the `include/linux/dcache.h` header file together with the global variable `dirent_hashtable` which references the *dcache* hash buckets.

For each file that is present in the Linux dcache there is a dentry. After resolving the path "/mnt/file-a", there are dentries in the dcache for "/", "mnt" and "file-a". Note that the dentry for "/" will actually be present in the dcache from early in the boot process.

If the file "/mnt/file-a" also has a hard link accessible via "/path-to-file-a" and both entries are in the dcache, there will be two distinct dentries, one for "file-a" and one for "path-to-file-a".

Here are the basic set of dentry fields. Others will be covered in later sections as the further details of the dcache are explored.

```
struct dentry {
    unsigned int                      d_flags;
    struct dentry                     *d_parent;
    struct qstr                        d_name;
    unsigned char                      d_iname [DNAME_INLINE_LEN];
    struct inode                       *d_inode;
    const struct dentry_operations   *d_op;
    struct super_block                 *d_sb;
    struct hlist_node                  d_hash;
    struct list_head                   d_lru;
    struct list_head                   d_child;
    struct list_head                   d_subdirs;
    unsigned long                      d_time;
    ...
}
```

For the structure fields that are shown above, here is a general overview of their purpose:

- **d_flags** – there are a lot of flags. A few will be discussed below and others introduced as needed throughout this section and in section 6.9.
- **d_parent** – a pointer to the dentry for the parent directory.
- **d_iname** – this field holds the name of the file if the size of the file name is less than (DNAME_INLINE_LEN - 1) bytes.
- **d_name** – for small file names, the name field within this **qstr** structure will point to the **d_iname** field described above. For file names that are DNAME_INLINE_LEN and longer, memory is allocated and this field will point to the allocated memory where the file name is then stored.
- **d_inode** – this field points to the Linux inode representing the underlying file. If the file doesn't exist, this field is set to NULL and the dentry is known as a *negative dentry*. Just as with files that exist, negative dentries are useful to prevent repeated filesystem lookups.
- **d_op** – filesystems can override / provide their own dcache functions that are called by the dcache when performing specific operations. For local filesystems the dcache is expected to be a correct representation of what's on disk but for network filesystems, a call may be needed to validate specific dentries since the cache may be out of

date with respect to the server. This is because other clients may have made changes to a file making the dcache on this node out of date.

- `d_sb` – the filesystem to which these dentries belong. This is particularly important during unmount and when pruning the dcache to reduce its memory footprint by getting easy access to all dentries for a specific filesystem.
- `d_hash` – the dcache itself is accessible through the `dentry_hashtable` global variable which is a large collection of hash buckets. Which hash bucket to use is determined by calling the `d_hash()` function passing the parent dentry and the file name. Each hash bucket can contain zero or more dentries linked through this field.
- `d_lru` – all dentries that are not currently being accessed are collected on an LRU list linked through this field. LRU lists are per filesystem and accessed through the `s_dentry_lru` field of the `super_block` structure.
- `d_subdirs / d_child` – this field points to a list of dentries, one for each cached file within the directory referenced by this dentry. These child dentries are linked through `d_child`.
- `d_time` – this field is only used by NFS and vboxfs to revalidate an entry after a specific time period has passed.

The `i_dentry` field of the inode points to a list of dentries that reference this file. Note that there can be multiple links to the same file. For example:

```
$ touch /mnt/mydir/myfile
$ ln /mnt/mydir/myfile link
```

If the file were accessed using both paths there would be one inode for the actual file but two dentries, one for each file name.

There are 35 different `d_flags` but not all flags will be covered in the book. A few are described below and other notable flags will be covered in section 6.9 when the dcache implementation is described in more detail.

- `DCACHE_MOUNTED` – generally speaking, this flag is set on each dentry that has a filesystem mounted on top of it. Since Linux supports multiple filesystem namespaces, it is possible that the dentry may not be mounted on in this namespace, therefore this flag is seen as a hint, not a guarantee.
- `DCACHE_LRU_LIST` – the dentry has no active holds and is therefore on the LRU list. It is quite possible that if the file is requested again, it will be found on this list before it is discarded.
- `DCACHE_REGULAR_TYPE` – this is a regular file. There are also flags for directories, symlinks and special files.
- `DCACHE_NEED_AUTOMOUNT` – if the kernel needs to traverse into a directory for which this dentry flag is set, it has hit an auto-mount directory so an attempt is made to mount the remote filesystem before pathname resolution can proceed.

- DCACHE_REFERENCED – the dentry has recently been accessed and therefore it should not be discarded.

There are seven flags that are set during dentry creation depending on whether the filesystem supports a list of `dentry_operations` (hanging off the `s_d_op` field of the `super_block` structure). Most filesystems do not have such a list of operations but they are common for network filesystems. The flags can be found in `dcache.h` by searching for `DCACHE_OP_*`. For example, if the `DCACHE_OP_COMPARE` flag is set when the kernel is looking up a file in the dcache, a call is made into the filesystem to actually perform the comparison.

To show how the dentry structures are related, consider figure 4.12 which shows a small file tree with a filesystem mounted on the directory `mount-dir`.

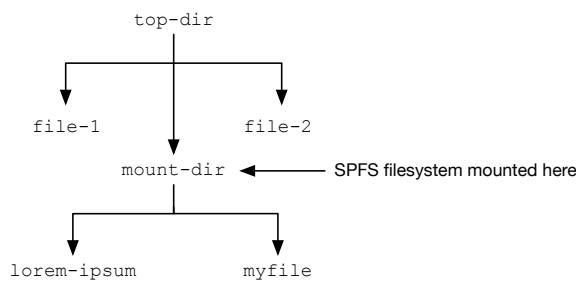


Figure 4.12: A simple example file hierarchy

For this particular tree, if we assume that all dentries are in the dcache, they are connected as shown in figure 4.13. The figure shows the connection between the parent and child dentries through the `d_parent`, `d_subdirs` and `d_child` fields. Notice that there is no direct connection between the base filesystem and the filesystem that is mounted on `mount-dir`. However, the `DCACHE_MOUNTED` mounted flag will generally be set on the dentry for `mount-dir` indicating that the directory has been mounted on. **This isn't always true in the case of namespace which will be covered later.**

4.4.2 Dcache Hash Buckets

The Linux dcache is basically a large list of hash buckets and is accessed through the global `dentry_hashtable` variable:

```
static struct hlist_head *dentry_hashtable
```

A simple example of how `dentry_hashtable` references the hash buckets is shown in figure 4.14. Each dentry in the cache is hashed using the dentry of its parent and the file name. To see if a file name is present in the cache, the kernel hashes to get the correct hash bucket then walks the appropriate list to look for the entry that it's searching for. Hash lists are generally fairly short, again helping with performance.

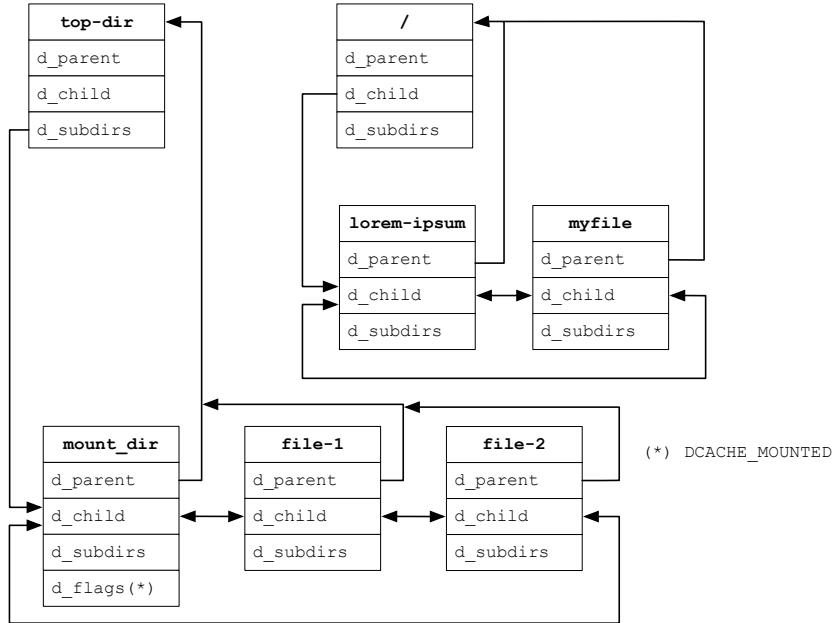


Figure 4.13: dentries for the example file hierarchy

Each element in the hash table is a pointer to a list of dentries that hash to the same value. The function `d_hash()` calculates the hash and thus gives access to the right hash bucket.

```
static inline struct hlist_bl_head *
d_hash(unsigned int hash)
{
    return dentry_hashtable + (hash >> d_hash_shift);
}
```

There is quite a lot of code behind hashing to locate the correct bucket. Section 6.8.5 covers the process in more detail. **XXX – not sure it does but come back to this.**

4.4.3 KGDB – Analyzing The Dcache

XXX – there is something wrong here. One entry in the left column and others the right column. Need to fix

This example shows how to dump out part of the dcache hash table and view some of the dentries. The dcache is accessed through `dentry_hashtable` and each hash bucket is of the following type:

```
struct hlist_bl_node {
    struct hlist_bl_node *next, **pprev;
```

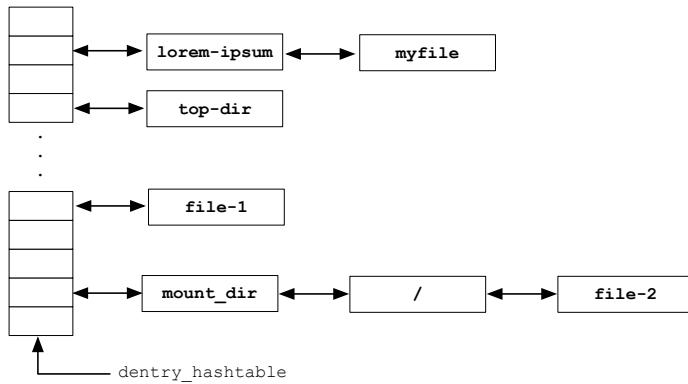


Figure 4.14: The dcache headed by dentry_hashtable

};

Starting at `dentry_hashtable`, each bucket occupies two addresses (forward and back pointers). Here we display 24 addresses:

```
(gdb) x/24a dentry_hashtable
0xfffff88817b800000: 0x0          0x0
0xfffff88817b800010: 0x0          0xfffff888103132248
0xfffff88817b800020: 0x0          0x0
0xfffff88817b800030: 0x0          0x0
0xfffff88817b800040: 0x0          0x0
0xfffff88817b800050: 0x0          0x0
0xfffff88817b800060: 0x0          0x0
0xfffff88817b800070: 0x0          0x0
0xfffff88817b800080: 0x0          0x0
0xfffff88817b800090: 0xfffff888101a816c8 0x0
0xfffff88817b8000a0: 0x0          0x0
0xfffff88817b8000b0: 0x0          0xfffff8881013a6788
```

The first thing to notice and especially if you dump more addresses is that most buckets are empty. Taking the first address displayed, we need to call `container_of()` to get to the dentry since this list goes through the `d_hash` field of the dentry structure.

```
(gdb) set $de = $container_of(0xfffff888103132248, \
                           "struct dentry", "d_hash")
```

Now let's display the name of the file and the contents of the `d_hash` field:

```
(gdb) p $de->d_iname
$213 = "warnings", '\000' <repeats 23 times>
(gdb) p $de->d_hash
$216 = {
    next = 0x0 <fixed_percpu_data>,
```

```
    pprev = 0xfffff88817b800018
}
```

In this example, the `next` field is NULL and the `pprev` field points back to the head of the hash bucket itself. Therefore this is the only dentry in this bucket. An interesting point is the dentry pointer at address `0xfffff88817b800090` which is also a single dentry. Why one entry is accessible through the `next` field of a `hlist_b1_node` structure and another is through the `pprev` field is a bit of a mystery. **XXX – come back here if time allows. I may have something wrong?**

4.4.4 Overriding / Supporting Dcache Operations

There is a structure defining a list of `dentry_operations` that some filesystems support. Most filesystems don't do anything in this regard as entries in the dcache are expected to be an accurate representation of what's on disk. The kernel and local filesystems work in conjunction to make sure that the dcache is accurate. However, this is not always the case with network filesystems such as NFS since other clients may perform operations that result in changes on the server which have not yet been reflected on this client potentially causing invalid dentries. Another example is the FAT filesystem which is not case sensitive so provides a `d_compare()` function to implement the comparison operations.

The list of operations are for this structure are:

```
struct dentry_operations {
    int     (*d_revalidate)(struct dentry *, unsigned int);
    int     (*d_weak_revalidate)(struct dentry *, unsigned int);
    int     (*d_hash)(const struct dentry *, const qstr *);
    int     (*d_compare)(const struct dentry *,
                        unsigned int, const char *,
                        const struct qstr *);
    int     (*d_delete)(const struct dentry *);
    int     (*d_init)(struct dentry *);
    void   (*d_release)(struct dentry *);
    void   (*d_prune)(struct dentry *);
    void   (*d_iput)(struct dentry *, struct inode *);
    char * (*d_dname)(struct dentry *, char *, int);
    struct vfsmount *(*d_automount)(struct path *);
    int    (*d_manage)(const struct path *, bool);
    struct dentry *(*d_real)(struct dentry *,
                           const struct inode *);
}
```

This structure and functions will further be discussed in section 6.9. **XXX – It will?**

4.5 The Inode Cache

In UNIX, a file stored on a physical disk was traditionally been referenced by a structure stored on disk called an *inode* which contained information about all parts of the file including owner, group, size of the file and pointers to the actual data. When a file was accessed

by the kernel, the filesystem would bring the inode into memory and parts of it were copied into an *in-core inode*. This is somewhat confusing since it didn't take long before there were multiple filesystems and some filesystems did not have inodes (NFS being the perfect example). Although both inode structures reference the same file, their purposes are different. The in-core inode is generic across all filesystems and has no built-in knowledge of the on-disk structure whereas the disk inode describes how the file's data and meta-data are stored on disk and is specific to the filesystem on which it belongs. With SunOS and SVR4 UNIX, the in-core structure was renamed and became a vnode structure to help distinguish between the two different entities which could often be very different. Add to that, network filesystems don't have inodes on the client and their internal structures were represented by different structures (*snodes* for NFS and *rnodes* for RFS). Linux kept the original UNIX nomenclature and has inodes for both in-core and on-disk structures. Of course Linux filesystems are free to implement whatever mechanisms they want for representing files.

Accessing a file for the first time can be a time-consuming operation and involve several disk I/Os as described above in section 4.4. In addition to the dcache, Linux maintains an in-core cache of recently accessed inodes. All cached inodes are associated with the mounted filesystem to which they belong so are easy to access when performing operations such as syncing data or unmounting a filesystem.

Inodes are shared between processes such that if two process have the same file open, there will only be one inode. Per-process information such as the file offset are process-specific and are held in the `file` structure as described earlier.



Linux inodes are represented by the `inode` structure which can be found in `include/linux/fs.h`

A subset of the fields of the inode are shown below. Several are self-explanatory and are returned when calling the `stat(2)` system call on a file.

```
struct inode {
    umode_t          i_mode;
    kuid_t           i_uid;
    kgid_t           i_gid;
    unsigned int     i_flags;
    const struct inode_operations *i_op;
    struct super_block *i_sb;
    struct address_space *i_mapping;
    unsigned long    i_ino;
    const unsigned int i_nlink;
    dev_t            i_rdev;
    loff_t           i_size;
    struct timespec64 i_atime;
    struct timespec64 i_mtime;
    struct timespec64 i_ctime;
    unsigned short   i_bytes;
```

```

blkcnt_t          i_blocks;
struct address_space   i_data;
void             *i_private;
}

```

Throughout the book the term *Linux inode* or *in-core inode* will be used in order to distinguish these inodes from the disk-based inodes implemented by various filesystems. The fields of the `inode` structure that require explanation at this point are:

- `i_ino` – the inode number which is unique across the filesystem to which this file resides.
- `i_nlink` – the number of links to this file. For regular files this will be "1" when the file is created and is incremented when a hard link is created for which the file name references this file.
- `i_op` – a list of operations that can be performed on this file. The operations are set by the filesystem when the file is opened and are generally applicable to directories. Operations include lookup (locate a file in the directory), file and directory creation and file removal.
- `i_sb` – a pointer to the `super_block` structure representing the mounted filesystem to which this file belongs. This will be described in section 4.8.1.
- `i_mapping` – for regular files, this points to an `address_space` structure which is used to reference pages in memory through which file I/O occurs. These pages contain cached data and can also contain data that needs to be written to disk. This will be explored in section 4.7.
- `i_private` – the `inode` structure represents the in-core instantiation of the file. When files are accessed, the underlying filesystem will read in its own inode (or other structure depending on the type of filesystem) and may cache parts of that structure in memory allocated by the filesystem. This field can point to a per-file private data structure. However, most Linux filesystems no longer use this field and have the Linux inode embedded within the filesystem inode and use the `container_of()` macro to move from the Linux inode structure to the filesystem data. This will be described in section 5.5 and examples are shown throughout the book.

There are various lists of inodes associated with each mounted filesystem. A particular inode can be on one of the following lists:

- `i_hash` – hash list used for locating inodes quickly using the inode number.
- `i_lru` – a list of least recently used inodes.
- `i_io_list` – TBD
- `i_sb_list` – a linked list of all inodes associated with the mounted filesystem to which this file belongs.

- `i_wb_list` – TBD
- `i_dentry` – a list of dentries that are referencing this file.
- `i_rcu` – TBD

TBD

There is no longer a single inode cache. Each filesystem has its own cache to which inodes are added when allocated. Each mounted filesystem has the following inode-related fields inside the `super_block` structure:

```
struct list_lru      s_inode_lru;
struct list_head     s_inodes;          /* all inodes */
struct list_head     s_inodes_wb;       /* writeback inodes */
```

Typing things together, figure 4.15 shows inodes hanging off the `super_block` structure referenced by the `s_inodes` field.

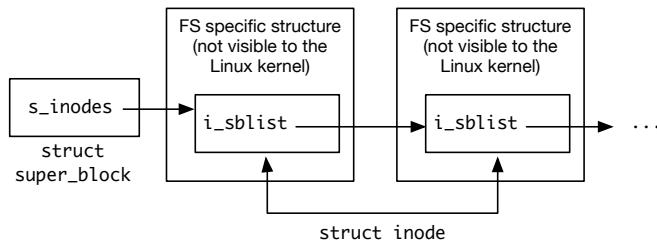


Figure 4.15: The inode cache for each mounted filesystem

This list makes it easy to locate all filesystem inodes which is particularly important when unmounting the filesystem.

4.5.1 KGDB – Analyzing Per-File Kernel Structures

In this demonstration, we will analyze a process that opens a file, keeps the file open allowing us to show the kernel structures from the `task_struct` through to the inode where we can verify the inode number. The goal is to follow the links shown in figure 4.10.

Below is a very simple program which opens a file and then pauses indefinitely (waiting for a signal). This allows us to search to files open by this process. We call the binary something obvious / unique to make it easy to search for, in this case "open-spate". The inode number associated with this file is also shown. An inode number is guaranteed to be unique within a filesystem although not across all filesystems. In this case the inode number is 407688.

```
int
main()
{
    int fd = open("lorem-ipsum", O_RDONLY);
```

```

        pause();
}

$ ls -li lorem-ipsum
407688 -rw-r--r-- 1 spate spate 2972 Dec 4 15:43 lorem-ipsum

```

Once gdb is running, several gdb commands will be executed to get us through the structures to locate the inode and verify that we have the right inode number. As we walk through gdb sessions, I'm going to assume that many readers may not be very familiar with the workings of gdb so will be somewhat more elaborate in showing the stages of moving from one structure to the next. For example, you don't need to use convenience variables but they certainly help a lot. A convenience variable is simply a variable that you can assign a value to. Once we find the `file` structure, we can assign the address to a variable called `file` for easier reference in future.

Now assuming the program above is running, we can use one of the Linux helper functions to locate the `task_struct` for the running process. The `pipe` command is very useful in this instance. We store the address displayed in the variable `ts`.

```

(gdb) pipe lx-ps | grep spate
0xfffff0000f34890c0 1213 open-spate

(gdb) set $ts = (struct task_struct *)0xfffff0000f34890c0

```

The next step is to display the `files` field and assign it to `files`. Note that we use `$2` here and could actually use `$2` rather than assign it to a convenience variable. You can use either option.

```

(gdb) p $ts->files
$2 = (struct files_struct *) 0xfffff0000f3aa2100

(gdb) set $files = $2

```

Next we display the whole `files_struct` structure.

```

(gdb) p *$files->fdt
$22 = {
    max_fds = 256,
    fd = 0xfffff0000f38c3000,
    close_on_exec = 0xfffff0000ec53efa0,
    open_fds = 0xfffff0000ec53ef80,
    full_fds_bits = 0xfffff0000ec53efc0,
    rcu = {
        next = 0x0,
        func = 0x0
    }
}

```

The `open_fds` field points to a bitmap of which file descriptors have been allocated. Below, you can see mostly 0s followed by 4 1s representing allocated file descriptors 0, 1, 2 and 3. The `fd` field of `files_struct` contains an array of file descriptors. We dump the contents of memory to display the first 4 addresses:

```
(gdb) x/t 0xfffff0000ec53ef80
0xfffff0000cacad200: 000000000000...000000000000000000000000000000001111
(gdb) x/4a $files->fdt.fd
0xfffff0000f38c3000: 0xfffff0000c1fb6f00 0xfffff0000c1fb6f00
0xfffff0000f38c3010: 0xfffff0000c1fb6f00 0xfffff0000ca4f7500
```

The `fd` field points to an array of pointers to `file` structures, one per file descriptor. The reason for displaying 4 addresses is that these addresses are the first 4 elements of the array and there is one for each of the 4 open files for this process (recall that 0, 1 and 2 are for `stdin`, `stdout` and `stderr`). The last address should point to the `file` structure for our open file `lorem-ipsum`.

Next we store its value and print out the `file` structure (I've omitted most fields). You can see the file position (`f_pos`) which is 0 since the file has just been opened but no reads or writes have taken place.

```
(gdb) set $file = (struct file *)0xfffff0000ca4f7500
(gdb) p *$file
f_path = {
    mnt = 0xfffff0000c08352e0,
    dentry = 0xfffff0000cfc92840
    f_inode = 0xfffff0000cfcf6c40,
    f_op = 0xfffff8000092c5e00 <ext4_file_operations>,
    f_flags = 131072,
    f_mode = 1212842013,
    f_pos = 0,
    f_cred = 0xfffff0000f4deb540,
    f_mapping = 0xfffff0000cfcf6db8,
}
```

You can also see a reference to the `dentry` for this file (we will cover `dentries` soon) but for now, we can just display the part of the `dentry` structure to confirm the file name:

```
(gdb) p $file->f_path.dentry->d_name.name
$32 = (const unsigned char *) 0xfffff0000cfc92878 "lorem-ipsum"
```

Finally, we save the address of the `inode` structure and display the `inode` number which matches what the `ls(1)` command showed above.

```
(gdb) set $ip = (struct inode *)0xfffff0000cfcf6c40
(gdb) p $ip->i_ino
$26 = 407688
```

There is a lot to take in here so I suggest that you try this example yourself. Perhaps open more files and look for each one or explore the `file` structure for `stdin`.

4.5.2 Analyzing Per-File Kernel Structures Using `crash`

Here is the same analysis as shown in the previous section but this time using the `crash(1)` command. All of the steps above are repeated but this time using `crash` commands on a live system (note different addresses due to running both sessions at different times).

The first step is to run the `ps` command and search for the running process:

```
crash> ps | grep spate
    7703      5214      3  fffff5a09d824af40 IN 0.0 2056 968 open-spate
```

The `task_struct` structure address is shown. Here we look for the `files` field. You can also run "struct `task_struct`" to achieve the same result but usually just typing the structure name by itself will suffice.

```
crash> task fffff5a09d824af40 | grep files
    files = 0xfffff5a09c08f7c80,
```

The next step is to locate the file table and from there, get the pointer to the file descriptor array:

```
crash> files_struct 0xfffff5a09c08f7c80 | grep "fdt ="
    fdt = 0xfffff5a09d8814800,
    fdtab = {

crash> fdtable 0xfffff5a09d8814800 | grep "fd ="
    fd = 0xfffff5a09d4292800,
```

As with the previous example, we dump memory and see `file` structure pointers for the first four file descriptors. From here we can show the filename through its `dentry` and locate the `inode` structure to get the inode number.

```
crash> rd 0xfffff5a09d4292800 4
fffff5a09d4292800:  ffff5a09d97a1200 ffff5a09d97a1200
fffff5a09d4292810:  ffff5a09d97a1200 ffff5a09b2bf7200

crash> file ffff5a09b2bf7200 | grep dentry
    dentry = 0xfffff5a09b6de2f00

crash> dentry 0xfffff5a09b6de2f00 | grep name
    d_name = {
        name = 0xfffff5a09b6de2f38 "lorem-ipsum"
        d_iname = "lorem-ipsum\000\000\000\0...0\000\000\000\000",

crash> file ffff5a09b2bf7200 | grep inode
    f_inode = 0xfffff5a095f42a140,

crash> inode 0xfffff5a095f42a140 | grep i_ino
    i_ino = 407688,
```

Generally speaking, with `gdb` integrated into `crash` you can achieve the same results with both debuggers and it's certainly easier to set up `crash` with only one system being needed. But when it comes to setting breakpoints, you will need `gdb`. Chapter ?? will show how to set breakpoints in the kernel with `gdb` / `kgdb`.

4.5.3 KGDB — Multiple dentries Per Single File

The `inode` structure is shared between different processes access the same file but can also be accessed by two different open files in the same process. In this example we show a case where we have a file with two hard links as follows:

```
# ls -li
total 1
4 -rw-r--r-- 2 root root 2972 Jan 23 21:07 hard-link
4 -rw-r--r-- 2 root root 2972 Jan 23 21:07 lorem-ipsum
```

You can see that both files have an inode number of 4. Both files will be opened by the following program which then pauses waiting on a signal giving us unlimited time to analyze the kernel structures:

```
int fd1 = open("lorem-ipsum", O_RDONLY);
int fd2 = open("hard-link", O_RDONLY);
pause();
```

In gdb we go through a similar sequence as the previous example to get to the file descriptor array to locate the file structures corresponding to file descriptors 4 and 5.

```
(gdb) p $ts->files->fdt->fd
$118 = (struct file **) 0xfffff0000e42a7000
(gdb) x/5a 0xfffff0000e42a7000
0xfffff0000e42a7000: 0xfffff0000f477a400 0xfffff0000f477a400
0xfffff0000e42a7010: 0xfffff0000f477a400 0xfffff0000d1050000
0xfffff0000e42a7020: 0xfffff0000d1050100
(gdb) set $file1 = (struct file *)0xfffff0000d1050000
(gdb) set $file2 = (struct file *)0xfffff0000d1050100
```

From here, we print out the inode structures which match as expected. We also show the dentries for each open instance showing the two filenames.

```
(gdb) p $file1->f_inode
$119 = (struct inode *) 0xfffff0000f18a9780
(gdb) p $file2->f_inode
$120 = (struct inode *) 0xfffff0000f18a9780
(gdb) p $file1->f_path->dentry->d_name.name
$121 = (const unsigned char *) 0xfffff0000d29573f8 "lorem-ipsum"
(gdb) p $file2->f_path->dentry->d_name.name
$122 = (const unsigned char *) 0xfffff0000d29397b8 "hard-link"
(gdb) p $file1->f_inode->i_ino
$2 = 4
(gdb) p $file1->f_inode->i_private
$4 = (void *) 0x8300000002
```

We also show that it's inode number 4. This time we have a private data structure that's held by the filesystem. This was how the SPFS filesystem was originally implemented to map the Linux inode to the SPFS inode. See 7 for more details.

4.6 The Buffer Cache

In earlier versions of UNIX up to around SVR3, all data that was read from, or written to disk, went through the *buffer cache*. At the heart of the buffer cache was a structure (`struct buf`) and a set of routes such as `getblk()`, `bread()`, `bwrite()` and

`brelse()` to get a block for a specified device, read or write a block and then release the block once the caller had finished with it.

Everything changed in the 1980s with the introduction of a new virtual memory (VM) architecture in Sun's SunOS which was also replicated in SVR4 UNIX. File I/O was taken out of the buffer cache and integrated into the new VM subsystem where filesystems were called with sets of pages which needed to be read or written. This hybrid buffer/page cache model is the approach that Linux has followed in that file I/O does not go through the buffer cache. There are still many things that the filesystem needs to read/write that are not regular file contents namely directory entries, inodes, extended attributes, symlinks and so on. And these structures and therefore corresponding blocks on disk, are likely to be read repeatedly. As discussed already, reading and writing data from/to disk repeatedly is slow and this results in the need for another cache, the *buffer cache*.

The main data structure used by the buffer cache is the `buffer_head` structure (akin to the old UNIX `buf` structure).



`include/linux/buffer_head.h` contains the definition of the `buffer_head` structure

The main fields in this structure are:

- `b_data` – a pointer to a memory area where the data is or will be located.
- `b_size` – the size of the buffer.
- `b_bdev` – the block device to which the buffer belongs.
- `b_blocknr` – the block number on the device to which this buffer refers to.
- `b_state` – the status of the buffer including `BH_Uptodate` (contents are valid) and `BH_Dirty` (buffer needs writing to disk).
- `b_count` – the buffer's reference count. Before a buffer is returned to a caller, this field is incremented. It is subsequently decremented when the caller releases the buffer (see `brelse()` below).

Here are some important functions:

- `__bread()` – reads a block using the specified block number and given size. If the read is successful, a pointer to the `buffer_head` structure, otherwise `NULL` is returned.
- `sb_bread()` – this is the same as `__bread()` but the size of the read block is taken from the `super_block` structure in addition to the device from which the read will be performed.

- `mark_buffer_dirty()` – marks the buffer as dirty (sets the `BH_Dirty` bit). The buffer will be written to the disk at a later time (from time to time the `bdflush` kernel thread wakes up and writes the buffers to disk). **XXX – need to check that this is still the case**
- `brelse()` – frees up the memory used by the buffer, after the buffer has been written to disk (if needed).
- `map_bh()` – associate the buffer-head with the corresponding sector. **very poor descriptions out there. I use in `sp_get_block()` so find a better description.**

Although jumping ahead a little, here is an example of how the buffer cache is used. For our disk-based filesystem SPFS (section 7), we need to read in the *superblock* from block 0 on disk to access basic information about the filesystem that is being mounted. The kernel calls into a filesystem-supplied function as part of mount processing. Here is a fragment of the code in this function:

```
spfs_fill_super(struct super_block *sb, void *data, int silent)
{
    struct sp_superblock      *spfs_sb;
    struct buffer_head         *bh;

    bh = sb_bread(sb, 0);
    if (!bh) {
        goto out;
    }
    spfs_sb = (struct sp_superblock *)bh->b_data;
```

When the filesystem is initialized, an internal `super_block` structure is allocated. One of the fields of this structure is the device on which the filesystem resides and another is the block size used for this filesystem/device. The SPFS `spfs_fill_super()` function is called as part of mount processing. It calls `sb_read()` to read in block 0 which is where the SPFS superblock is held. It fits in one block regardless of the block size.

Filesystems will generally call `sb_bread()` when reading blocks from disk (**what about writing?**). You will see a few exceptions in the individual files (run "grep `__bread` `fs/*`" at the top of the kernel source tree) to see examples. One specific example is ext4 which states above the function `__ext4_sb_bread_gfp()`:

Currently with `sb_bread` it's impossible to distinguish between ENOMEM and EIO situations (since both result in a NULL return).

Instead of calling `sb_bread()`, ext4 calls `sb_getblk_gfp()`. If it returns NULL, ext4 can interpret that as an ENOMEM condition.

4.6.1 Buffer Cache Size / Usage

The amount of space currently used by the buffer cache can be seen through `/proc`:

```
$ grep Buffers /proc/meminfo
Buffers:          326920 kB
```

To see this value change, simply run the command "find /" which will result in a lot of calls in to the different mounted filesystem to read inodes and directory entries. Let it run for a while and then stop it. Look inside `/proc/meminfo` again and you will likely see the value increase.

But how much space will be used? That depends on how much memory is available and what type of activity is being performed. If there is a lot of I/O that requires the buffer cache, the amount of space will increase. If memory is required for other purposes, it will be reclaimed and reused. Operating systems used to be much more restrictive on the amount of memory available for different caches. But some caches could be underutilized resulting in wasted memory. If there is a lot of activity around file meta-data, it makes sense to let the buffer cache grow. If the system is doing a lot of I/O, utilize the page cache. Memory can always be reclaimed to serve the needs of what is happening at any moment in time.

XXX—can be reclaimed but how often is that useful? Section in next chapter that covers the buffer cache in more detail?

4.6.2 Flushing Buffers

When dirty buffers are released, the data is not immediately written to disk. If there is no need to write it straightaway, why do so? It could be perfectly valid for a caller to request the same buffer again, update it and release it once more. Why issue two writes instead of one? When to write dirty buffers and how often has been a hotly debated topic for a very long time and in Linux, this is no exception.

In the early days of Linux, there was `bdfflush`, a user process that called into the kernel to periodically sync buffers. User processes can be killed so this was replaced by a kernel thread but that also ran into issues on larger systems. This single thread would become a bottleneck. The next step was to replace `bdfflush` with multiple threads (called `pdflush`), one per physical drive. The `pdflush` threads were eventually removed and replaced with a more elaborate scheme.

XXX—come back when doing the performance chapter

4.7 The Page Cache

As described above, meta data data is read from / written to disk through the buffer cache. All file data is read from / written to the *page cache*. Figure 4.16 shows how in-memory pages are associated with an individual inode. As with most data on disk, it's assumed that the data will be accessed repeatedly so caching it in memory to reduce the number of disk accesses is paramount to having a high-performing system. **XXX—check this figure is correct once kgdb running**



The `address_space` structure is defined in `include/linux/fs.h`

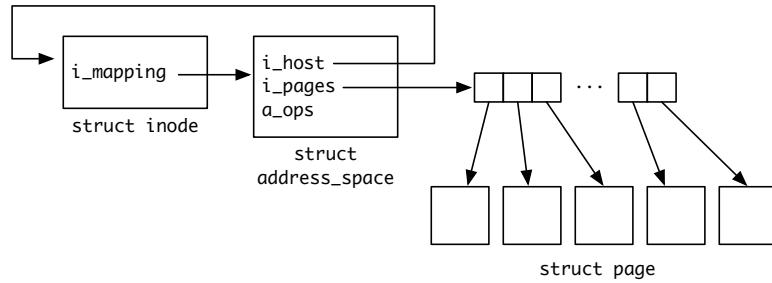


Figure 4.16: Relationship between inodes and the page cache

A few of the fields of the `address_space` structure are shown below:

```

struct address_space {
    struct inode                         *host;
    struct xarray                         i_pages;
    unsigned long                        nrpages;
    const struct address_space_operations *a_ops;
    unsigned long                        flags;
    ...
}
  
```

All pages that are currently cached in memory are referenced through the `i_pages` field. An `xarray` structure XXX

need to come back here once I understand it ... there is a "virtual" field but it's configurable. So how does the page get mapped in? It's covered in the VFS chapter and as the following shows, `kmap()` must be called to get a virtual address before data can be accessed

For AIO:

```

ev = kmap(page);
copy_ret = copy_to_user(event + ret, ev + pos,
                       sizeof(*ev) * avail);
kunmap(page);
  
```

The `a_ops` field is set by the filesystem. In the example disk-based filesystem, described in chapter 7, the following operations are specified:

```

struct address_space_operations sp_aops = {
    .dirty_folio      = block_dirty_folio,
    .invalidate_folio = block_invalidate_folio,
    .read_folio       = sp_read_folio,
    .writepage        = sp_writepage,
    .write_begin      = sp_write_begin,
    .write_end        = sp_write_end,
    .bmap             = sp_bmap
};
  
```

These operations will be described in detail in subsequent chapters but this examples shows that filesystems may or may not provide all of the functions in this structure (or others for that matter). They may provide a minimum set of functions and rely on the kernel for *generic* functions that can also be used by other filesystems. More complex file systems will want more control so will provide more functions.

Linux manages the physical memory by dividing it into PAGE_SIZE chunks which are typically the same as the CPU's page size. This is generally 4 KB but can go as large as 64 KB (**need example of this**).

Each page in the system is represented by a page structure and these structures can actually occupy quite a lot of memory (lwn - importance?).

Need to get this in somewhere: - you can use -t to touch every page. must be useful somewhere

```
$ vmtouch /mnt/lorem-ipsum
      Files: 1
      Directories: 0
      Resident Pages: 1/1 4K/4K 100%
      Elapsed: 0.002463 seconds
```

XXX - tie this with the VFS chapter

4.7.1 The page Structure

Each physical page in the system has an associated `page` structure which is used to keep track of the page regardless of whatever the page is being used for. Note that we have no way to track which tasks are using



The `page` structure is defined in `include/linux/mm_types.h`.

need to show at some point what the relevant fields are.

Talk about `kmap()` or where the page gets mapped into memory somewhere (low and `highmem`)

4.7.2 Compound Pages / Page Folios

XXX—need to do this section after understanding how they work and look at the VFS interfaces for folios

I/O occurs in page-sized chunks (thus the name *page cache*) and the page cache only deals with single pages, a fixed size chosen based on the CPU on which Linux is running. A page has generally been 4096 bytes since the first launch of Linux on the Intel x86 architecture.

Compound pages -> hugetlbfs or the transparent huge pages (see folios page above). Sets of pages with a head and getting to the head happens all the time. "struct page *com-

pound_head(struct page *page);" is called many times and is inline so makes the kernel bigger.

Folios represent a page structure that is guaranteed not to be a tail page.

```
* struct folio - Represents a contiguous set of bytes.  
* @flags: Identical to the page flags.  
* @lru: Least Recently Used list; tracks how recently this folio was used.  
* @mlock\_\_count: Number of times this folio has been pinned by mlock().  
* @mapping: The file this page belongs to, or refers to the anon\_\_vma for  
* anonymous memory.  
* @index: Offset within the file, in units of pages. For anonymous memory,  
* this is the index from the beginning of the mmap.  
* @private: Filesystem per-folio data (see folio\_\_attach\_\_private()).  
* Used for swp\_\_entry\_\_t if folio\_\_test\_\_swapcache().  
* @_mapcount: Do not access this member directly. Use folio\_\_mapcount() to  
* find out how many times this folio is mapped by userspace.  
* @_refcount: Do not access this member directly. Use folio\_\_ref\_\_count()  
* to find how many references there are to this folio.  
* @memcg\_\_data: Memory Control Group data.  
*
```

WEB - A folio is a physically, virtually and logically contiguous set of bytes. It is a power-of-two in size, and it is aligned to that same power-of-two. It is at least as large as PAGE_SIZE. If it is in the page cache, it is at a file offset which is a multiple of that power-of-two. It may be mapped into user-space at an address which is at an arbitrary page offset, but its kernel virtual address is aligned to its size.

```
struct folio {  
    /* private: don't document the anon union */  
    union {  
        struct {  
            /* public: */  
            unsigned long flags;  
            union {  
                struct list_head lru;  
                /* private: avoid cluttering the output */  
                struct {  
                    void *__filler;  
                };  
                /* public: */  
                unsigned int mlock_count;  
                /* private: */  
            };  
            /* public: */  
            };  
            struct address_space *mapping;  
            pgoff_t index;  
            void *private;  
            atomic_t _mapcount;  
            atomic_t _refcount;  
    #ifdef CONFIG_MEMCG
```

```

        unsigned long memcg_data;
#endif
/* private: the union with struct page is transitional */
};

struct page page;
};

};

TBD

```

4.7.3 KGDB — Analyzing Page Cache Structures

TBD - the idea is to find the pages incore after reading the file. In this example, after reading from 2 pages, there are no pages attached to the address_space struct. Not sure why? No page I/O?

This does work (last attempt) but there is no virtual address mapping. kmap() needs to be called and can't really do that from gdb. Wonder whether I can do it in SPFS or another driver?

```

#include <unistd.h>
#include <fcntl.h>

int
main()
{
    char buf[2];
    int fd;

    fd = open("big-lorem-ipsum", O_RDONLY);
    read(fd, buf, 1);
    lseek(fd, 4096, SEEK_SET);
    read(fd, buf, 1);
    pause();
}

TBD

get the task_struct:

crash> ps | grep 2pages
2308 1146 3 fffff7d2c80604ec0 IN 0.0 2056 968 2pages

```

```

crash> task fffff7d2c80604ec0 | grep files
files = 0xfffff7d2c8000b180,

```

```

crash> files_struct 0xfffff7d2c8000b180 | grep "fdt ="
fdt = 0xfffff7d2c8a4bd700,

```

```
crash> fdtable 0xfffff7d2c8a4bd700 | grep "fd = "
      fd = 0xfffff7d2c82b14000,  
  
crash> rd 0xfffff7d2c82b14000 4
fffff7d2c82b14000: ffff7d2c8b686f00 ffff7d2c8b686f00 .oh.,}...oh.,}..
fffff7d2c82b14010: ffff7d2c8b686f00 ffff7d2c91e0ad00 .oh.,}.....,}..  
  
                                         ^
                                         |
file struct for our file  
  
crash> file ffff7d2c91e0ad00 | grep dentry
      dentry = 0xfffff7d2c939fa600
crash> dentry 0xfffff7d2c939fa600 | grep name
      d_name = {
          name = 0xfffff7d2c939fa638 "big-lorem-ipsum"
      d_iname = "big-lorem-ipsum ..."  
  
crash> file ffff7d2c91e0ad00 | grep inode
      f_inode = 0xfffff7d2c93b23838,  
  
crash> inode 0xfffff7d2c93b23838 | grep mapping
      i_mapping = 0xfffff7d2c93b239b0,  
  
this is      - struct address_space     *i_mapping;
crash> address_space 0xfffff7d2c93b239b0 | grep host
      host = 0xfffff7d2c93b23838,  
  
host is our inode (see above)
```

There are no pages so not sure when they get created. Mapping only? Need to walk through the I/O paths.

4.7.4 Process Address Space

**just as an example to show how it also has address_space structs for each vm_area_struct
- doesn't really fit but show that there are files underlying these spaces and show the
ops to support them - perhaps put a binary on SPFS?**

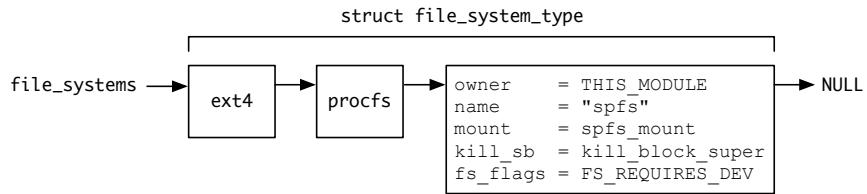


Figure 4.17: List of Available Filesystems

4.8 Mounted Filesystem Structures

There are two main structures that Linux uses when handling the management of and mounting of filesystems. The `super_block` structure is used for maintaining a list of mounted filesystems and the `file_system_type` structure holds the list of filesystem types that can be mounted.



`fs/filesystems.c` contains the `file_systems` variable, routines for registering and unregistering filesystems together with functions for handling `struct file_system_type` including searching for a filesystem type during mount (2).

A call to mount a filesystem generally specifies the filesystem type such as follows:

```
# mount -t spfs /mnt
```

Each time a filesystem module is loaded, the kernel adds the `file_system_type` to a list of available filesystems headed by `file_systems`. This is shown in figure 4.17. One of the structures in the figure has been expanded to show its contents. This is for the SPFS filesystem for which the implementation will be described in section 7.

The main arguments of the `struct file_system_type` are:

- `name` – the name of the filesystem. When you run "mount -t spfs", the kernel will compare "spfs" against the `name` field of each structure until a match is found or it determines that this filesystem type does not exist or its corresponding modules has not been loaded.
- `mount` – the filesystem function which will be called during mount processing.
- `kill_sb` – called when a filesystem is being unmounted.

The filesystem `kill_sb()` function performs cleanup operations (freeing structures etc) and then invokes one of the following functions: **XXX—that's what the kernel doc says but some filesystems set kill_sb to one of these functions. SPFS uses the first**

- `kill_block_super()` – which unmounts a file system on a block device

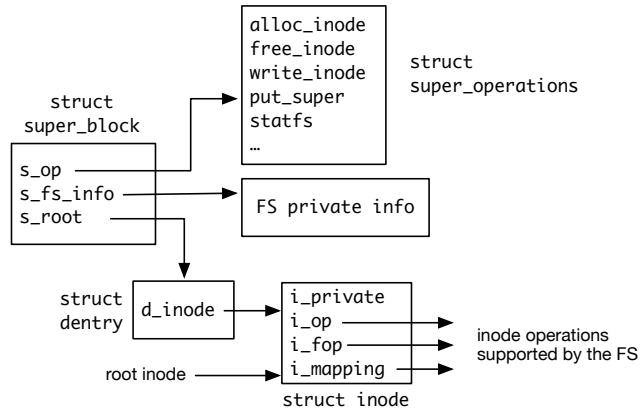


Figure 4.18: Structures Representing Mounted Filesystems

- `kill_anon_super()` – which unmounts a virtual file system (information is generated when requested)
- `kill_litter_super()` – which unmounts a file system that is not on a physical device (the information is kept in memory)

You can run "`cat /proc/filesystems`" to view the list of available filesystems. This will display all filesystems that are attached to `file_systems` as shown in figure 4.17.

4.8.1 The `superblock` and `mount` Structures

Once a filesystem is mounted, there is a `super_block` structure allocated for each mounted filesystem. The `super_blocks` variable points to a linked list of all mounted filesystems.



`fs/superblocks.c` contains the `super_blocks` global variable which references all mounted filesystems. It also has routines for mounting and unmounting filesystems. The `super_block` structure can be found in `include/linux/fs.h`

There are a lot of fields in the `super_blocks`. A few are shown in figure 4.18 and described below:

- `s_root` – references the root dentry for this filesystem from which we can get to the root inode.
- `s_op` – a set of filesystem-specific operations can be invoked. Examples include inode operations such as allocation, writing (to disk) and freeing, writing the superblock to disk and reading per-filesystem stats (think `df(1)`).

- `s_fs_info` – each filesystem will have information about this particular mount that it wishes to keep in-core. When a filesystem is mounted, the underlying filesystem can allocate such a structure and attach it to `s_fs_info`.
- `s_dev` – the device on which the filesystem resides. This will be accessed to read/write data through the buffer cache **XXX etc - or is it s_bdev???**

There are also several lists associated with each mount such as the list of dentries and several inode lists. **XXX—expand etc**

XXX—need to cover struct mount - see lx-mounts. also connection with vfsmount structure. not sure what the hell this is.

```
struct mount {
    struct hlist_node mnt_hash;
    struct mount *mnt_parent;
    struct dentry *mnt_mountpoint;
    struct vfsmount mnt;
```

Something about "mount"



The `mount` structure is defined in `fs/mount.h` which differs from the `mount.h` found in `include/linux`.

4.8.2 The `vfsmount` Structure

XXX - need to add this too.

4.8.3 KGDB — Analyzing the List of Mounted Filesystems

This example shows how to view the list of mounted filesystems from within `gdb`.



`fs/proc_namespace.c` contains code for the proc filesystem that deals with displaying mounted filesystems. The comment at the top of the file says *but has rather close relationships with fs/namespace.c, thus here instead of fs/proc*

The list of mounted filesystems is accessed through the `super_blocks` global variable (`fs/super.c`) which is declared as follows: **XXX—the following link looks out of place**

```
static LIST_HEAD(super_blocks);
```

Locating this variable in `gdb` gives us:

```
(gdb) p super_blocks
$56 = {
    next = 0xfffff0000c0022800,
    prev = 0xfffff0000e4455000
}
```

A simple way to view all of the mounted filesystems is to use the Linux helper function `lx_mounts` which displays information for all mounted filesystems. You can then display more information about each mounted filesystem given the addresses of the `mount` and `super_block` structure addresses displayed.

```
(gdb) lx-mounts
      mount          super_block      devname  pathname  fstype
0xfffff0000c0285540 0xfffff0000c0025000 rootfs   /        rootfs
0xfffff0000c0835900 0xfffff0000c08a5000 sysfs    /sys     sysfs
0xfffff0000c0834780 0xfffff0000c08a7800 proc     /proc    proc
...
...
```

For the first row (highlighted), let's see how these structures are connected. First we show the link from the `mount` structure to the `super_block` structure:

```
(gdb) set $mount = (struct mount *)0xfffff0000c0285540
(gdb) p $mount->mnt.mnt_sb
$5946 = (struct super_block *) 0xfffff0000c0025000
```

The mount point can also be displayed:

```
(gdb) p $mount->mnt_mountpoint->d_name.name
$5948 = (const unsigned char *) 0xfffff0000c0404f38 "/"
```

For another filesystem:

```
$ mount | grep spfs
/dev/sda on /mnt type spfs (rw,relatime)
```

we display the mount point and the name of the device:

```
(gdb) pipe lx-mounts | grep spfs
0xfffff0000d9e5da40 0xfffff0000e4455000 /dev/sda /mnt spfs ...
(gdb) set $mount = (struct mount *)0xfffff0000d9e5da40
(gdb) p $mount->mnt_mountpoint->d_name.name
$5955 = (const unsigned char *) 0xfffff0000f3fbaf38 "mnt"
(gdb) p $mount->mnt_devname
$5956 = 0xfffff0000e622e200 "/dev/sda"
```

Need to bring in vfsmount and perhaps show a figure or add a figure earlier. Seems like a bug dump

4.9 Namespace Structures

The `mount_namespaces` (7) manpage gives a good introduction to mount namespaces.

The `unshare` (1) command runs a program in a new namespace.

The `user_namespace` structure needs some explaining. Here is a subset of the structure:

```
struct user_namespace {
    struct uid_gid_map    uid_map;
    struct uid_gid_map    gid_map;
    struct uid_gid_map    projid_map;
    struct user_namespace *parent;
    int                  level;
    kuid_t                owner;
    kgid_t                group;
    struct ns_common      ns;
    unsigned long          flags;
    bool                  parent_could_setfcap;
    struct work_struct    work;
    struct ucounts        *ucounts;
    long                  ucount_max[UCOUNT_COUNTS];
    long                  rlimit_max[UCOUNT_RLIMIT_COUNTS];
}
```

Inside the `super_block` structure there is a field that references namespaces:

```
struct user_namespace *s_user_ns;
```

There are 60 references to this all over the place!!!

4.10 Conclusion

This chapter provided a lot of information starting with tools to help you explore the Linux kernel which is now over 26 million lines of code so somewhat daunting especially if you haven't looked at the kernel before or looked at much earlier versions.

This chapter started to dig into the Linux kernel. The first step is to get familiar with navigating around the kernel source so tools were presented to show how to navigate around the source code in addition to inline cross reference tools. Although there are over 26 million LOC in the kernel, the filesystem components are only found in a small number of places so browsing becomes a little easier.

The book has been following a path from user libraries through the system call handling and into the individual filesystems. The system call layer was explained and showed how to find system call handler for its respective system call functions that applications invoke. From there it's possible to navigate through the file handling kernel code.

There are several subsystems that comprise filesystem access including the directory cache, buffer cache, page cache and inode caches. Each of these subsystems were described showing how they are all interconnected. Before digging into the filesystem kernel paths in the next chapter, it's imperative to have an understanding of what these components deliver.

The main structures used for file and filesystem access were described showing how they link together. To help reinforce the material, kernel debugging using `gdb` was introduced and several examples were shown to highlight how the structures are linked together. Other examples made use of the `crash(1)` utility. This should provide you with enough

information to be able to explore how the kernel structures are linked together. It's recommended to try the examples shown here, look at the other fields of the structures displayed and try different examples.

Chapter 5

Common Linux Types and Locks

With 25 millions LOC you can be sure that there are many areas of the kernel that implement structures, locks and so on in very similar ways. To avoid this duplication, there are several types and structures provided by the kernel that kernel developers can use. As you browse the kernel code you'll see lots of references to linked lists, containers and so on XXX

This section describes the most common types and gives examples of how they're used.

5.1 Lists

```
xxx
struct list_head {
    struct list_head *next, *prev;
};

xxxx
#define list_for_each(pos, head) \
    for (pos = (head)->next; pos != (head); pos = pos->next)

xxxx
```

5.1.1 An Unusual List Example

In some structures, the list element is not at the start of the structure which can make it a challenge to debug. Take the `mount` structure as an example. The list of mounted filesystems is referenced by the global variable `super_blocks`. This is a list HEAD (?) structure which points to the front element of the list and the last. Each element is of type `struct super_block`. Inside this structure we have:

```
struct list_head    s_mounts; /* list of mounts;
                                _not_ for fs use */
```

In gdb you can use a Linux helper function to display the mounted filesystem list. For each, it displays the `super_block` structure as well as one of the `mount` structures. Section XXX will explain the "one vs many" issue. For now, let's look at one entry:

```
(gdb) lx-mounts
mount          super_block      devname    path  fstype options
...
0xf...812f3ea500 0xf...8127b7c800 /dev/loop8 /mnt  spfs   rw,r...
```

Taking the address of the `super_block` structure, we can display the `s_mounts` field as follows:

```
(gdb) set $sb = (struct super_block *)0xfffff888127b7c800
(gdb) p $sb->s_mounts
$3 = {
  next = 0xfffff88812f3ea578,
  prev = 0xfffff888101952cf8
}
```

Ignoring the `prev` element for now, the `next` field points to a `mount` structure but look at the address versus the address displayed by `lx-mounts`:

```
0xfffff88812f3ea500  # displayed by lx-mounts
0xfffff88812f3ea578  # what's in s_mounts
```

There is a difference of `0x78` between both addresses. This is because the linked list is actually **through?** the `mnt_instance` field:

```
struct list_head mnt_instance;
```

which is at an offset of `0x78` bytes:

```
(gdb) p &$mount->mnt_instance
$39 = (struct list_head *) 0xfffff88812f3ea578
```

If you search for `mnt_instance` in the kernel `fs` directory you will find code that walks this list:

```
list_for_each_entry(mnt, &sb->s_mounts, mnt_instance) {
    /* process each structure */
}
```

Internally, a call will be made to `container_of` which will be described in the next section (**or earlier/??????**)

5.2 Locks

5.3 The `xarray` Structure

XXX - `i_pages` from the `address_space` structure points is of type "struct `xarray`". it's a list of all cached pages.

See <https://docs.kernel.org/core-api/xarray.html> for information
and <https://lwn.net/Articles/745073/>

to see actual contents, will need to call `kmap()` - <https://stackoverflow.com/questions/31966298/can-we-access-memory-through-a-struct-page-structure>. Also `kmap_atomic()`

need to come back here once I understand it ... there is a "virtual" field but it's configurable. So how does the page get mapped in?

5.3.1 The Big Kernel Lock (BKL)

As Linux transitioned to SMP (Symmetric Multi-Processing) architectures, a temporary solution was needed until more kernel components could be changed to support fine-grain locking. This *solution* was called the Big Kernel Lock (BKL) which was a global spin-lock

```
lock_kernel();  
  
/*  
 * Critical section, synchronized against all other BKL users...  
 * Note, you can safely sleep here and the lock will be transparently  
 * released. When you reschedule, the lock will be transparently  
 * reacquired. This implies you will not deadlock, but you still do  
 * not want to sleep if you need the lock to protect data here!  
 */  
  
unlock_kernel();  
  
xxx
```

At the kernel there were three functions that could be called:

- `lock_kernel()` – acquire the BKL
- `unlock_kernel()` – release the BKL
- `kernel_locked()` – returns non-zero if the BKL is held and zero otherwise

The BKL was finally removed it in 2011 in kernel version 2.6.39.

5.3.2 Local Locks

xxx

5.3.3 Semaphores

xxx

5.3.4 Mutexes

xxx

5.4 Black-Red Trees

Black-red trees, often abbreviated to *rbtrees*, are a type of self-balancing binary search tree and are used for storing sortable key/value data pairs.



You can find the Linux rbtree implementation in `lib/rbtree.c`

Linux has *augmented rbtrees*

An *interval tree* is an example of augmented rbtree
From lwn:

There are a number of red-black trees in use in the kernel. The anticipatory, deadline, and CFQ I/O schedulers all employ rbtrees to track requests; the packet CD/DVD driver does the same. The high-resolution timer code uses an rbtree to organize outstanding timer requests. The ext3 filesystem tracks directory entries in a red-black tree. Virtual memory areas (VMAs) are tracked with red-black trees, as are epoll file descriptors, cryptographic keys, and network packets in the "hierarchical token bucket" scheduler.

5.5 Wrapping Structures With `container_of`

There are many structures in the kernel that embed other structures, particularly those that are linked together in a linked list. This is where the `container_of()` macro comes in useful. The basic idea is that we have one structure embedded (or contained within) another structure. The embedded structure is often one member of a linked list or tree structure. In this example, we have a linked list of VFS inodes and each is embedded in a structure used by the filesystem that allows it to get to filesystem-specific information about this file.

Let's take a look at this confusing looking macro:

```
/**  
 * container_of - cast a member of a structure out to the  
 * containing structure  
 * @ptr:    the pointer to the member.  
 * @type:   type of the container struct this is embedded in.  
 * @member: the name of the member within the struct.  
 */  
#define container_of(ptr, type, member) ({  
    void *__mptr = (void *)(ptr);          \  
    \
```

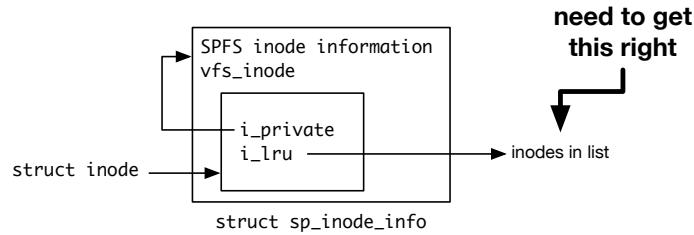


Figure 5.1: Using `container_of()` to get to the "*containing*" structure

```
static_assert (__same_type(*ptr, ((type *) 0)->member) || \
              __same_type(*ptr, void), \
              "pointer type mismatch in container_of()"); \
((type *) (_mptr - offsetof(type, member))); })
```

It really does look like something you'd see during an interview when they get into that annoying programming phase! But the idea is actually very simple and the function is basic pointer arithmetic. Let's take inodes as an example. Inodes, as seen in section ??, are structures representing files both in-core and on disk, thus an incore inode and a disk inode. When a file is opened, the disk inode will be brought into memory and the filesystem will likely cache all or part of it. The Linux inode is used by the VFS layer and will be passed to the filesystem through many of the different VFS to filesystem functions. Often, the first thing the filesystem will do is get hold of its disk inode from the incore inode.

The Linux inode also have a field called `i_private` which has traditionally been used to reference filesystem-specific information. In this case, when the inode is being initialized, the filesystem would allocate its own inode structure (or equivalent) and set `i_private` to point to this structure. For example:

```
struct sp_inode_info *spi;
...
inode->i_private = spi;
```

and then reference it using a macro such as:

```
#define ITOSPI(inode) (struct sp_inode_info *)&inode->i_private
```

Take the following BFS filesystem code fragment as an example: **XXX—replace with SPFS code after starting to use `container_of`**

```
static void bfs_evict_inode(struct inode *inode)
{
    ...
    struct bfs_inode_info *bi = BFS_I(inode);
    ...
}
```

The `BFS_I` function takes the incore inode as an argument and returns a pointer to BFS specific inode information for this file:

```
static inline struct bfs_inode_info *BFS_I(struct inode *inode)
{
    return container_of(inode, struct bfs_inode_info, vfs_inode);
}
```

Here is the definition of the `bfs_inode_info` structure:

```
struct bfs_inode_info {
    unsigned long i_dsk_ino;
    unsigned long i_sblock;
    unsigned long i_eblock;
    struct inode vfs_inode;
};
```

The Linux inode is embedded in this structure which can be described as the *container of* the Linux inode.

This is very commonly used in the Linux kernel. I found over 17,000 references to `container_of` in C files.

Section **kdgb-inodelist** shows how inodes are linked together and associated with a specific `super_block` structure representing a specific mount point.

5.6 Semaphores, Mutexes, And Lockless Algorithms

You will need an lwn.net subscription to be able to read this article.



URL 18 – <https://tinyurl.com/4y2d3yen>

some of this list comes from the lwn article so broaden as much as possible

1. semaphores vs mutexes history
2. spinlocks
3. lockless algorithms
4. the big lock
5. early days - nothing but cli() /sti()
6. SCSI semaphores -> binary semaphores
7. semaphores on the way out? lwn article
8. major locks used for FS structures
9. etc

5.7 Read-Copy Update (RCU) Synchronization

RCU (Read-Copy Update) was first implemented RCU in the Sequent DYNIX/ptx operating system and documented in 1998. DYNIX (DYNAMIC unIX) was a UNIX variant based on 4.2BSD and later System V UNIX designed to run on Intel SMP processors.

The first RCU mechanisms were introduced in Linux back in 2002 as a means to create large-scale scalability improvements by allowing reads to occur concurrently with updates.

The most common use of RCU in the Linux kernel is as an alternative to a read-write locking. Pathname resolution is one particularly important use case and is described in section ??.

5.8 Conclusion

There are several different locking primitives over the years and parts of the kernel have been heavily optimized to minimize taking locks unless absolutely necessary to avoid contention on today's multi-processor systems resulting in the loss of performance.

Like most operating systems, Linux took some time to transition from uniprocessor systems (UP) to symmetric multiprocessing (SMP) systems. After starting with the Big Kernel Lock (BKL), the kernel was in time, transitioned to use XXX.

TBD

Chapter 6

The VFS Layer

Now that we've covered the main kernel structures related to file and filesystem access, highlighted the system call interface and the major subsystems that surround file access, this section will go into more detail about how the system call handlers above the VFS layer work, the internals of the VFS as well as major subsystems such as the dcache, buffer, page and inode caches. It will also cover the VFS to filesystem layer where the kernel calls into individual filesystems.

This chapter will dip much deeper into the kernel operations for filesystem support covering everything from file lookup to file creation to how I/O and the page cache operates. There will be several examples of using `gdb` to analyze kernel structures to help.

6.1 The VFS Entry Point

Section 4.2 showed where in the kernel the source code can be found for the various Linux system call handlers. A lot of work performed at this layer includes marshaling system call arguments, calling common functions that can handle similar system calls, for example `open(2)` and `openat(2)`, and then calling into the *Virtual File System* (VFS) interface. Figure 6.1 shows where these functions reside with respect to other parts of the kernel.

There are 56 VFS functions, split across 17 files, and all can be found in the `fs` directory. In most cases, there is a function starting with `vfs_` which matches, or closely matches, the system call of the same name. Here are a few examples:

```
init.c:      vfs_getattr(&path, stat, STATX_BASIC_STATS,
init.c:      vfs_mknod(mnt_user_ns(path.mnt), ...
namei.c:     vfs_open(&path, file);
namei.c:     vfs_create(mnt_userns, ...
namei.c:     vfs_rmdir(mnt_userns, ...
read_write.c: vfs_read(f.file, buf, count, ppos);
read_write.c: vfs_write(f.file, buf, count, ppos);
```

The main structures handled in the VFS layer are `file`, `dentry` and `inode` structures as described in the previous chapter. You may see some figures in Linux kernel documentation

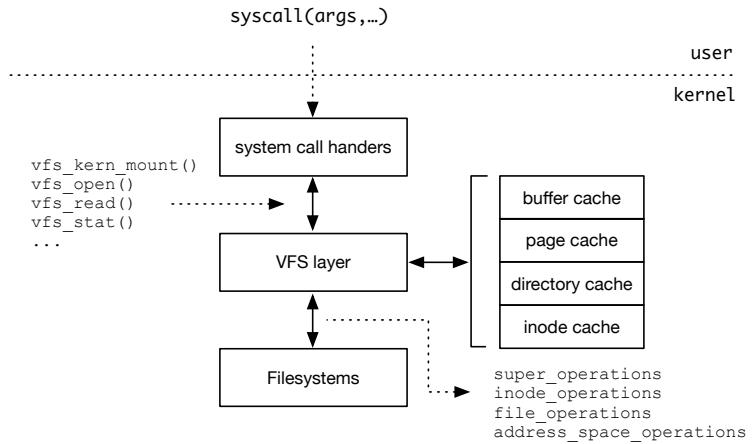


Figure 6.1: The Linux VFS Interface

that differ slightly from this figure, specifically having the system call layer as part of the VFS layer but I think it's cleaner to keep the two layers separate. Note that this also differs from the original Sun/SVR4 VFS layer which generally pointed to the division between filesystem independent and filesystem dependent code at both the top layer of the kernel and at the filesystem later. Either way, it's important to recognize that there are two distinct interfaces, one entering filesystem independent code as part of system call handling or file access from other areas in the kernel and one where individual filesystems are called. Of course, in addition to calling individual filesystems, the VFS layer also calls into other subsystems such as those managing the directory, inode, buffer and page caches.

6.2 VFS to Filesystem Interfaces

There are four structures of operations that the kernel uses to call through to the underlying filesystem. There are almost 100 functions altogether across the four structures and each filesystem supports a subset of these functions. For example, SPFS only supports 35 operations and makes uses of generic kernel functions in some circumstances. Some of the more feature rich filesystems, or those who want more control over specific operations, will provide more fine-grain functions and be called more often. As an example, XFS provides closer to 100 different functions that the kernel can call.

Here are the operation structures are provided by each filesystem. For details about the functions in each structure, refer to section 6.23.

- `super_operations` – a set of functions that operate at the filesystem level including functions to get filesystem statistics, sync the filesystem and several inode-related operations (allocating, writing, freeing, etc). Note that the filesystem function

to be called during mount processing is provided to the kernel when the filesystem registers itself.

- `inode_operations` – filesystems will likely define three different sets of inode operations which will be attached to directory inodes, regular file inodes and symlink inodes. Each exports a different set of operations.
- `file_operations` – these functions are attached to the `f_op` field of the file structure and include operations such as read, write and lseek.
- `address_space_operations` – attached to `inode->i_mapping->a_ops` by the filesystem, these functions provide interaction between the filesystem and the Linux page cache. Generally speaking, file I/O will utilize these functions. (**XXX**—see **read/write in file operations - what's the difference?**

Which operations to provide is dependent on the type of file. Using SPFS as an example, here are the operations attached for directory files, regular files and symlinks:

```
sp_dir.c:     inode->i_op = &sp_file_inops;
sp_dir.c:     inode->i_op = &sp_dir_inops;
sp_dir.c:     inode->i_op = &sp_symlink_operations;
```

Looking at directory and symlink operations, you can see operations that are some operations that are directory specific and some that are symlink specific:

```
struct inode_operations sp_dir_inops = {
    .create      = sp_create,
    .lookup      = sp_lookup,
    .mkdir       = sp_mkdir,
    .rmdir       = sp_rmdir,
    .link        = sp_link,
    .unlink      = sp_unlink,
    .symlink     = sp_symlink,
    .rename      = sp_rename
};

static const struct inode_operations sp_symlink_operations = {
    .readlink    = sp_readlink,
    .get_link    = sp_page_get_link,
    .getattr     = sp_getattr,
};
```

The SPFS operations vectors will be described in chapter 7. For the filesystems in the Linux source tree, search the directory for a specific filesystem to see what vectors it provides. For example:

```
$ pwd
/Users/spate/src/linux-6.3.3/fs/xfs
$ grep inode_operations *.c
xfs_iops.c:static const struct inode_operations \
            xfs_inode_operations = {
```

```
xfs_iops.c:static const struct inode_operations \
            xfs_dir_inode_operations = {
xfs_iops.c:static const struct inode_operations \
            xfs_dir_ci_inode_operations = {
xfs_iops.c:static const struct inode_operations \
            xfs_symlink_inode_operations = {
xfs_iops.c:    inode->i_op = &xfs_inode_operations;
xfs_iops.c:    inode->i_op = &xfs_dir_ci_inode_operations;
xfs_iops.c:    inode->i_op = &xfs_dir_inode_operations;
xfs_iops.c:    inode->i_op = &xfs_symlink_inode_operations;
xfs_iops.c:    inode->i_op = &xfs_inode_operations;
```

6.3 Registering Filesystems

Individual filesystems are registered and unregistered with the Linux kernel by calling the `register_filesystem()` and `unregister_filesystem()` functions. You will see calls to register filesystems from many places in the kernel source, not just from within the `fs` directory and individual filesystems under `fs`. These calls are made when a filesystem module is being loaded or unloaded.



`fs/filesystems.c` contains various functions for managing the list of registered filesystems together with code for displaying these filesystems through the `/proc/filesystems` interface.

The code for `register_filesystem()` is actually quite simple with the main parts of the function shown below:

```
int
register_filesystem(struct file_system_type * fs)
{
    int res = 0;
    struct file_system_type **p;

    write_lock(&file_systems_lock);
    p = find_filesystem(fs->name, strlen(fs->name));
    if (*p)
        res = -EBUSY;
    else
        *p = fs;
    write_unlock(&file_systems_lock);
    return res;
}
```

Using SPFS as an example, here is the `file_system_type` structure that is passed as an argument to `register_filesystem()`:

```
static struct file_system_type spfs_fs_type = {
```

```

    .owner      = THIS_MODULE,
    .name       = "spfs",
    .mount      = spfs_mount,
    .kill_sb    = kill_block_super,
    .fs_flags   = FS_REQUIRES_DEV,
};

}

```

The call to `find_filesystem()` is also straightforward. It walks through the linked list of current registered filesystems comparing their names against the name specified by the filesystem being registered.

```

static struct file_system_type **
find_filesystem(const char *name, unsigned len)
{
    struct file_system_type **p;
    for (p = &file_systems; *p; p = &(*p)->next)
        if (strcmp((*p)->name, name, len) == 0 &&
            !(*p)->name[len])
            break;
    return p;
}

```

On return from `register_filesystem()`, if a match was found, the filesystem has already been registered and an error is returned. If it had not been registered previously, the filesystem is simply added to the end of the linked list and can now be mounted. Note that the `file_systems_lock` must be taken since this action could be performed in parallel by two different processes.

6.3.1 Unregistering Filesystems

Filesystems are unregistered by a user calling the `rmmod(8)` command. The code for unregistering a filesystem is straightforward and involves walking the list of registered filesystems to see if the filesystem is registered.

```

int
unregister_filesystem(struct file_system_type * fs)
{
    struct file_system_type **tmp;

    write_lock(&file_systems_lock);
    tmp = &file_systems;
    while (*tmp) {
        if (fs == *tmp) {
            *tmp = fs->next;
            fs->next = NULL;
            write_unlock(&file_systems_lock);
            synchronize_rcu();
            return 0;
        }
        tmp = &(*tmp)->next;
    }
}

```

```
        }
write_unlock(&file_systems_lock);
        return -EINVAL;
}
```

If the filesystem is found on the list, it will be removed by setting the `next` field of the previous structure to point to the filesystem referenced by the `next` field of the filesystem that the kernel is removing.

You will notice that there is no check to see if there are any filesystems of this type that are currently mounted before the `file_system_type` structure is removed from the list of registered filesystems. This check is made during module unload (`rmmmod(8)`) time before the module exit function is called.



Module removal code can be found in `module/main.c`. Start with the function `delete_module()`

6.3.2 KGDB — Analyzing the List of Registered Filesystems

The list of registered filesystems is a simple linked list with the head of the list referenced by the global variable `file_systems`. Each element of the list is of type "struct `file_system_type`" which has a field `next` which points to the next filesystem in the list. In `gdb`, to print the first several entries, simply start at `file_systems` and keep using the `next` field to advance through the list

```
(gdb) pipe p *file_systems | grep 'name ='
      name = 0xffff8000098b4960 "sysfs",
(gdb) pipe p *file_systems->next | grep 'name ='
      name = 0xffff8000097fcbb0 "tmpfs",
(gdb) pipe p *file_systems->next->next | grep 'name ='
      name = 0xffff80000984b608 "bdev",
(gdb) pipe p *file_systems->next->next->next | grep 'name ='
      name = 0xffff80000984c210 "proc",
```

Walking through this list by hand is similar to what happens when you look into the contents of `/proc/filesystems` as follows:

```
$ cat /proc/filesystems
nodev    sysfs
nodev    tmpfs
nodev    bdev
nodev    proc
...
```

The `filesystems_proc_show()` function in `fs/filesystems.c` walks through the list and displays each filesystem and whether it requires a physical device or not.

```

static int
filesystems_proc_show(struct seq_file *m, void *v)
{
    struct file_system_type * tmp;

    read_lock(&file_systems_lock);
    tmp = file_systems;
    while (tmp) {
        seq_printf(m, "%s\t%s\n",
                   (tmp->fs_flags & FS_REQUIRES_DEV) ? "" : "nodev",
                   tmp->name);
        tmp = tmp->next;
    }
    read_unlock(&file_systems_lock);
    return 0;
}

```

As with the other functions shown in this section, `filesystems_proc_show()` starts with the global variable `file_systems`, walks through the list and prints the name of the filesystem (`tmp->name`) together with `nodev` if it's a pseudo filesystem.

6.4 Mounting and Unmounting Filesystems

Section 4.8 described the `file_system_type` and `super_block` structures which are used during mounting of filesystems and to manage the list of mounted filesystems. Here we describe how the VFS mounts a filesystem and utilizes these structures.

At the time of writing, Linux filesystems are mounted using one of two different mechanisms. The first path, now called *legacy mounts* has been around for many years with a few minor changes here and there. The new form which utilizes a *filesystem context* structure into which is placed:

- Filesystem type.
- Namespaces.
- Source/Device names (there may be multiple).
- Superblock flags (SB_*).
- Security details.
- Filesystem-specific data, as set by the mount options.

This structure is passed ... XXX

At the time of writing, some filesystems have been switched over to the new method of mounting while several are still using the legacy method. As an example, as of September 2023, btrfs has changes made to support the new method but the path has not yet been applied to the mainline Linux kernel sources. Both methods will be described in this section.

URL 19 <https://tinyurl.com/ykpvxkf>

The new style mount method is described in the following document:
but the rationale for why it was introduced is not well documented XXX

This section will start by showing the interaction between the VFS layer at the filesystem which is where the main differences apply between the two styles of mounting. The code through the VFS layer should then become clearer and will be described later in this section.

6.4.1 Legacy Mounts

The legacy mount process is described in this section and was what I originally implemented in SPFS using the 6.3.3 kernel. I then switched to using the filesystem context method but rather than throw away the legacy code, I'm documenting it here since the process is still used by many of the existing filesystems.

When I wrote my last book ([9]), 20 years ago, Linux filesystems were still declared by making a call to `register_filesystem()` as follows:

```
static DECLRE_FSTYPE_DEV(uxfs_fs_type, "uxfs", ux_read_super);

static int __init_uxfs_fs(void)
{
    return register_filesystem(&uxfs_fs_type);
}
```

Only one function was provided and that function was called to read in the filesystem's superblock after the kernel had already created the corresponding `super_block` structure.

The process is still very similar today for legacy mounts. The filesystem will create a `file_system_type` structure in which it places information about the filesystem.

```
static struct file_system_type spfs_fs_type = {
    .owner      = THIS_MODULE,
    .name       = "spfs",
    .mount      = spfs_mount,
    .kill_sb    = kill_block_super,
    .fs_flags   = FS_REQUIRES_DEV,
};
```

When a filesystem module is loaded, it passes a pointer to this structure through a call to `register_filesystem()`. One of the elements of the structure is a *mount* function which is called by the kernel each time a filesystem of this type is mounted. For SPFS, the filesystem that will be described in chapter 7, these functions and structures are shown in figure 6.2.

In the same structure, the filesystem may specify a function to be called when the filesystem is being unmounted. **Need to better understand all the places at which it's called.**

```

module_init(init_spfs_fs)
{
    static int __init
    init_spfs_fs(void)
    {
        register_filesystem(&spfs_fs_type);
    }

    static struct file_system_type spfs_fs_type = {
        .owner      = THIS_MODULE,
        .name       = "spfs",
        .mount      = spfs_mount,
        .kill_sb    = kill_block_super,
        .fs_flags   = FS_REQUIRES_DEV,
    };

    struct dentry *
    spfs_mount(struct file_system_type *fs_type, int flags,
               const char *dev_name, void *data)
    {
        printk("spfs: spfs_mount\n");
        return mount_bdev(fs_type, flags, dev_name,
                          data, spfs_fill_super);
    }

    static int
    spfs_fill_super(struct super_block *sb, void *data,
                   int silent)
    {
        sb->s_op = &spfs_sops;
    }
}

```

Figure 6.2: Mounting a Filesystem Using Legacy Mount Mode

The code for `spfs_mount()` is very simple, relying solely on a call to `mount_bdev()`. It passes `spfs_fill_super()` as an argument.

```

static struct dentry *
spfs_mount(struct file_system_type *fs_type, int flags,
           const char *dev_name, void *data)
{
    return mount_bdev(fs_type, flags, dev_name, data,
                      spfs_fill_super);
}

```

The role of `sp_fill_super()` is to:

- Allocate filesystem-specific data structures to be used in-core.
- Read the filesystem superblock from disk.
- Read in the root inode and instantiate it in the dcache.
- Set the `s_op` field of the `super_block` structure to reference a vector of filesystem operations. These typically involve inode-related operations such as allocating,

freeing, writing and eviction.

As the book progresses and has been out for some time, I suspect that most filesystems will move away from legacy mounts. To find those that still use legacy mounts, run the following inside the `fs` directory:

```
$ grep '\.mount' */*.c
9p/vfs_super.c: .mount = v9fs_mount,
adfs/super.c: .mount = adfs_mount,
affs/super.c: .mount = affs_mount,
autofs/init.c: .mount = autofs_mount,
befs/linuxvfs.c: .mount = befs_mount,
bfs/inode.c: .mount = bfs_mount,
btrfs/super.c: .mount = btrfs_mount,
...
...
```

6.4.2 Mounting Using Filesystem Context

It looks like `get_tree()` is the new version of `spfs_mount()` / `spfs_fill_super()` (perhaps)

xxxx

As described in Documentation/filesystems/mount_api.rst the new process consists of the following steps:

1. Create a filesystem context.
2. Parse the parameters and attach them to the context. Parameters are expected to be passed individually from user-space, though legacy binary parameters can also be handled.
3. Validate and pre-process the context.
4. Get or create a superblock and mountable root.
5. Perform the mount.
6. Return an error message attached to the context.
7. Destroy the context.

Don't know if this is the right order yet ... see Documentation/filesystems/mount_api.rst for more info

```
struct file_system_type {
    int (*init_fs_context) (struct fs_context *);
    const struct fs_parameter_spec *parameters;
    ...
    struct dentry *(*mount) (struct file_system_type *, int,
                           const char *, void *);
    void (*kill_sb) (struct super_block *);
    ...
}
```

There is an `fs_context` structure that is used during mount processing. It's mentioned all over the place in the `fs` directory. There is also a file `fs/fs_context.c`.

```
struct fs_context {
    const struct fs_context_operations *ops;
    struct file_system_type           *fs_type;
    void                             *fs_private;
    struct dentry                     *root;
    struct user_namespace            *user_ns;
    struct net                        *net_ns;
    const struct cred                 *cred;
    char                            *source;
    char                            *subtype;
    void                            *security;
    void                            *s_fs_info;
    unsigned int                      sb_flags;
    unsigned int                      sb_flags_mask;
    unsigned int                      s_iflags;
    unsigned int                      lsm_flags;
    enum fs_context_purpose          purpose:8;
    ...
};
```

XXX a bunch of filesystems deal with this struct. SPFS does not. take fs/ramfs as an example ..

```
static struct file_system_type ramfs_fs_type = {
    .name          = "ramfs",
    .init_fs_context = ramfs_init_fs_context,
    .parameters    = ramfs_fs_parameters,
    .kill_sb       = ramfs_kill_sb,
    .fs_flags      = FS_USERNS_MOUNT,
};

struct ramfs_fs_info {
    struct ramfs_mount_opts mount_opts;
};

int
ramfs_init_fs_context(struct fs_context *fc)
{
    struct ramfs_fs_info *fsi;

    fsi = kzalloc(sizeof(*fsi), GFP_KERNEL);

    fsi->mount_opts.mode = RAMFS_DEFAULT_MODE;
    fc->s_fs_info = fsi;
    fc->ops = &ramfs_context_ops;
    return 0;
}
```

only seems to be one argument and that's stuffed in a struct referenced from the private super_block pointer (`s_fs_info`) - if you look at ext4, the usage is much more extensive. Note that (`s_fs_info`) is listed as FS private data but search for it in `fs/*` and there are references to something else. Double check on this. my usage is the same as BFS and likely others. Hmm!

6.4.3 Mounting Filesystems at the VFS Layer

Figure 6.2 shows how a filesystem module establishes itself with the kernel during module load time. It also covered how the filesystem registers its `file_system_type` structure which is then held on the linked list of available filesystems accessible through global variable `file_systems`. Together with the "mount" function provided and the `fs_flags` field (which will be set to `FS_REQUIRE_DEV` if a block device is needed to host this filesystem, this provides enough for the kernel to be able to perform a mount operation when requested for this filesystem type.



Look for "SYSCALL_DEFINE5 (mount" in `fs/namespace.c` for the starting point for handling the `mount (2)` system call.

The system call handler for mounting a filesystem starts by copying in the type of the filesystem, the device name and any mount options before issuing a call to `do_mount()`.

```
SYSCALL_DEFINE5(mount, char __user *, dev_name, char __user *,
                dir_name, char __user *, type, unsigned long,
                flags, void __user *, data)
{
    char *kernel_type;
    char *kernel_dev;
    void *options;

    kernel_type = copy_mount_string(type);
    kernel_dev = copy_mount_string(dev_name);
    options = copy_mount_options(data);
    ret = do_mount(kernel_dev, dir_name, kernel_type,
                   flags, options);
}
```

The `do_mount()` function needs to resolve the mount point passed to `mount (2)` to a directory inode before the filesystem can be mounted. The copying of the pathname is done with a call to `user_path_at()`. The `path_put()` function calls `dput()` releases its hold on the dentry held in the path structure before calling `mnt_put` which calls `mntput_no_expire()` – **XXX – which look complicated – come back.**

```
long
do_mount(const char *dev_name, const char __user *dir_name,
         const char *type_page, unsigned long flags,
```

```

        void *data_page)
{
    struct path path;

    user_path_at(AT_FDCWD, dir_name, LOOKUP_FOLLOW, &path);
    ret = path_mount(dev_name, &path, type_page, flags,
                      data_page);
    path_put(&path);
    return ret;
}

```

The functions called by `user_path_at()` are:

```

user_path_at() → user_path_at_empty() → filename_lookup() →
path_lookupat() → link_path_walk()

```

Section 6.7 covers Linux kernel pathname resolution in detail and specifically covers the function `link_path_walk()` and functions calling it.

Back to mounting - need to expand `do_mount()`

`do_new_mount_fc()` is the function called once everything is set up (fs context etc) and is the one that allocates a new `vfsmount` structure.

Here are the `mount` and `vfsmount` structures. Note that `vfsmount` is embedded within the `mount` structure.

```

struct vfsmount {
    struct dentry      *mnt_root; /* root of mounted tree */
    struct super_block *mnt_sb;   /* ptr to superblock */
    int                mnt_flags;
    struct user_namespace *mnt_userns;
}

struct mount {
    struct hlist_node  mnt_hash;
    struct mount       *mnt_parent;
    struct dentry      *mnt_mountpoint;
    struct vfsmount    mnt;
    ...
    struct mnt_namespace *mnt_ns; /* containing namespace */
    struct mountpoint  *mnt_mp;   /* where is it mounted */
}

```

The `super_block` structure has a field that contains a list of mounts for this filesystem/device since a filesystem (one the one block device) can be mounted in multiple places at the same time. The comment in the source code specifies that this field should not be accessed by filesystems.

```
    struct list_head s_mounts;
```

Figure 6.3 shows how these structures are linked together following a call to `mount` a filesystem. If this is the last mount, we can locate its structures from the `prev` of the `super_blocks` list head.

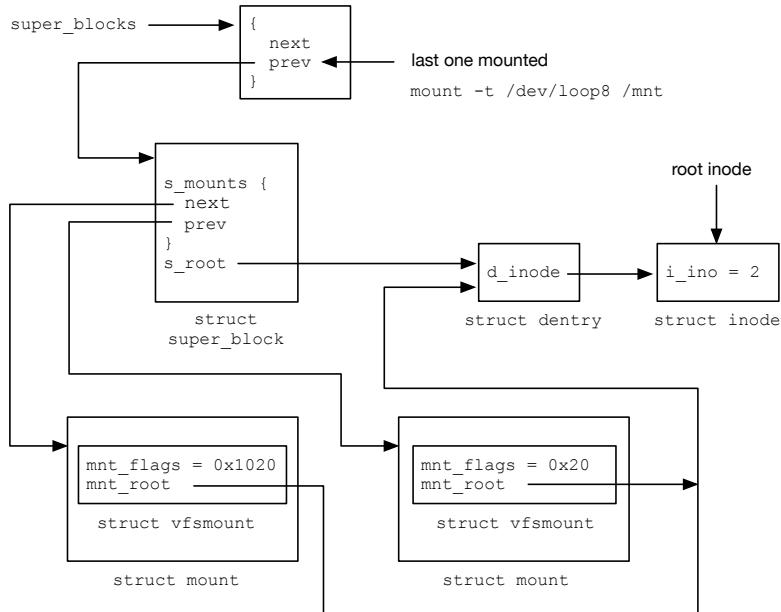


Figure 6.3: Structures for the last mounted filesystem

For the **mnt_flags** field here are the values for each of the **mount** structures:

```
#define MNT_RELATIME      0x20
#define MNT_SHARED         0x1000
```

XXX – not sure why at this point we have two mount structures, perhaps something to do with namespaces?

There is also **struct mountpoint - what the hell is that???**

```
struct mountpoint {
    struct hlist_node m_hash;
    struct dentry *m_dentry;
    struct hlist_head m_list;
    int m_count;
};
```

The following function allocates a new **mountpoint** structure as follows:

```
static struct mountpoint *
get_mountpoint(struct dentry *dentry)
{
    ...
    new = kmalloc(sizeof(struct mountpoint), GFP_KERNEL);
```

This is called indirectly from **do_add_mount()** as well as other places.
references

6.4.4 KGDB – A Look at Structures Following Mount

This section will explore what happens following a call to `mount(1)` to mount a filesystem as follows:

```
$ sudo mount -t spfs /dev/loop8 /mnt
```

It's easy to find the corresponding `super_block` structure associated with this mount by using one of the Linux Python helper functions as follows:

```
(gdb) l *mounts
mount          super_block      devname     path   fstype options
...
0xf...812f3ea500 0xf...8127b7c800 /dev/loop8 /mnt spfs    rw, r...
```

The second address is the address of the `super_block` structure and the first address is the associated `mount` structure for this mount point. We can also get the same information as follows:

```
(gdb) p super_blocks
$4 = {
    next = 0xfffff88810005a800,
    prev = 0xfffff888127b7c800
}
```

Recall that the list of `super_block` structures for all mounted filesystems can be accessed through the global variable `super_blocks`. What is shown here are pointers to the first and last elements of the list. In this example, the "spfs" filesystem is the last one mounted. This is saved as a convenience variable and the `s_mounts` field is displayed:

```
(gdb) set $sb = (struct super_block *)0xfffff888127b7c800
(gdb) p $sb->s_mounts
$3 = {
    next = 0xfffff88812f3ea578,
    prev = 0xfffff888101952cf8
}
```

The filesystem has only been mounted once so why are there two entries? Let's explore further. The `mount` structures are linked together through the `mnt_instance` field which is 0x78 bytes into the `mount` structure. Take the first address displayed above. We can confirm this offset as follows:

```
(gdb) p &$mount->mnt_instance
$39 = (struct list_head *) 0xfffff88812f3ea578
```

If we print out each one, both display `mnt` and `/dev/loop8` adding to the confusion:

```
(gdb) set $m1 = (struct mount *)0xfffff88812f3ea500      # - 0x78
(gdb) set $m2 = (struct mount *)0xfffff888101952cf8      # - 0x78
(gdb) p $m1->mnt_mountpoint->d_name.name
$49 = (const unsigned char *) 0xfffff88812f8324b8 "mnt"
(gdb) p $m1->mnt_devname
$46 = 0xfffff8881008d4b10 "/dev/loop8"
```

```
(gdb) p $m2->mnt_mountpoint->d_name.name
$47 = (const unsigned char *) 0xffff88812f8324b8 "mnt"
(gdb) p $m2->mnt_devname
$48 = 0xffff8881008d4230 "/dev/loop8"
```

One thing that is different is the mnt_flags field:

```
(gdb) p/x $m1->mnt.mnt_flags
$27 = 0x1020
(gdb) p/x $m2->mnt.mnt_flags
$28 = 0x20
```

As described above, they have the following values:

```
#define MNT_RELATIME      0x20
#define MNT_SHARED        0x1000
```

That's all well and good but what exactly does it mean?

6.4.5 Unmounting Filesystems

Unmounting a filesystem starts with the `umount (8)` command and/or the `umount (2)` / `umount2 (2)` system calls.



Source code for handling unmounting of filesystems can be found in
`fs/namespace.c` and `fs/super.c` AND XXX

The entry point in the kernel is `ksys_umount ()` which has two calls as follows:

```
static int
ksys_umount(char __user *name, int flags)
{
    struct path path;

    // basic validity checks on "flags" done first

    ret = user_path_at(AT_FDCWD, name, lookup_flags, &path);
    return path_umount(&path, flags);
}
```

The first call to `user_path_at` is responsible for looking up the mount point specified in the call to `umount (2)`. This function calls `filename_lookup ()` to resolve the pathname which in turn calls `path_lookupat ()` which is described in section `pathname-set`.

The `path` structure contains a dentry for the directory looked up and the `vfsmount` for the filesystem on which is found.

```
int
path_umount(struct path *path, int flags)
```

```

{
    struct mount *mnt = real_mount(path->mnt);
    int ret;

    ret = can_umount(path, flags);
    if (!ret)
        ret = do_umount(mnt, flags);

    dput(path->dentry);
    mntput_no_expire(mnt);
    return ret;
}

```

There are several checks performed by `can_umount()` such as whether the caller has permission to unmount, that the path is the root of the mounted filesystem and `check_mnt()?????`
TBD

6.5 Namespace Creation and Handling

XXX

All of the namespaces are created using the `clone(2)` or `unshare(2)` system calls.
XXX why so many entry points in the kernel???



The system call handler for the `clone` system call can be found in `kernel/fork.c`. Look for the functions `kernel_clone()` and `k`

XXX - the code looks horrendous! Not going to be easy to figure out.

`clone -> kernel_clone()`
`unshare -> ksys_unshare()`

6.5.1 Handling chroot (2)

The `chroot(2)` system call is very straightforward. Figure 6.4 shows the `fs` field of the `task_struct` pointing to an `fs_struct` structure which contains pointers to the root directory and the current working directory. All that the kernel needs to do for `chroot(2)` is to ensure that the `path` argument of the system call references a valid directory and that the process has permission to access it.

Here is the system call entry point for handling `chroot(2)`:

```

SYSCALL_DEFINE1(chroot, const char __user *, filename)
{
    struct path path;

    retry:
    error = user_path_at(AT_FDCWD, filename,

```

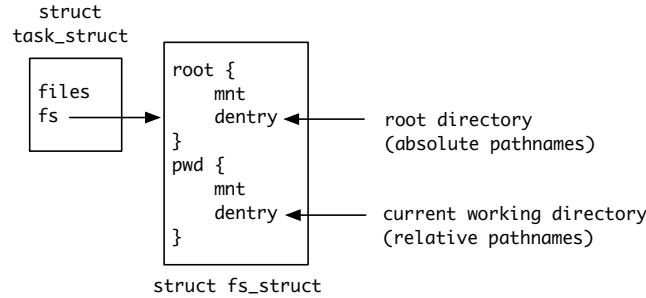


Figure 6.4: Switching a process's root directory with `chroot` (2)

```
        lookup_flags, &path);

error = path_permission(&path, MAY_EXEC | MAY_CHDIR);
set_fs_root(current->fs, &path);
error = 0;
path_put(&path);
return error;
}
```

Assuming the path is valid, `set_fs_root()` is called to update the root directory and a hold on the previous root is dropped.

```
void
set_fs_root(struct fs_struct *fs, const struct path *path)
{
    struct path old_root;

    path_get(path);
    spin_lock(&fs->lock);
    old_root = fs->root;
    fs->root = *path;
    spin_unlock(&fs->lock);
    if (old_root.dentry)
        path_put(&old_root);
}
```

At this point, any pathname resolution calls starting with "/" will start at this new root directory.

6.6 KGDB – Analyzing Mount Namespace Creation

6.7 Opening a File / Pathname Resolution

Figure 6.5 shows the high-level paths followed for opening a file from `do_sys_open()` through to `path_openat()` which calls for pathname resolution and opens the file. On a successful return from this function, the file has been found, it will have been opened and a `file` structure is allocated and returned for it.

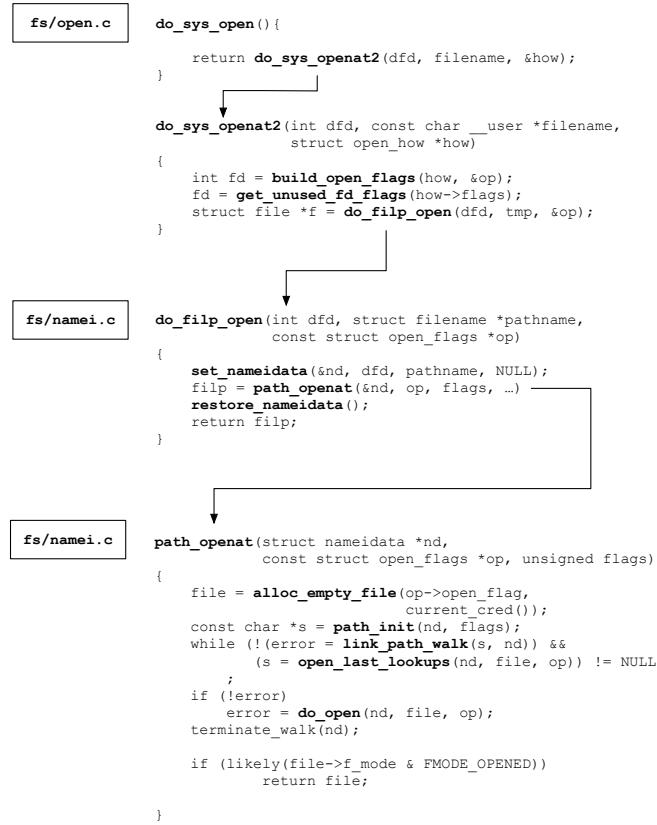


Figure 6.5: Main Kernel Routines for Opening a File / Allocating a File Descriptor

`get_unused_fd_flags()` could return very quickly if an available file descriptor is available. If not, the code gets quite complex in allocating new structures. This will be described in section ???. Either way, a call is made to `do_filp_open()` which allocates a `nameidata` structures that contains information needed to perform pathname resolution (`link_path_walk()`) on the pathname that has been passed to one of the open system call. This walks through the path one component at a time until the last component is

reached. If everything is successful at this point, a call is made to `do_open()` to actually open the file.

XXX - finish this section and do fd / file expansion

XXX - need to go back to previous chapter and see how open flags/mode map to f_flags and f_mode - around line 599

6.7.1 File Descriptor Allocation and File Table Expansion

There are two calls invoked during the open paths shown in figure 6.5. The first call is to allocate a file descriptor (`get_unused_fd_flags()`) and the second call is to allocate a file structure (`alloc_empty_file()`).

The fastest path to getting a file descriptor is through the following set of calls:

```
get_unused_fd_flags() → __get_unused_fd_flags()
                           → alloc_fd()
```

The `start` and `end` arguments specify the lowest and highest (?) file descriptor that should be allocated. In the calling stack for opening a file `start` is 0 indicating that the file descriptor with the lowest value should be returned and `end` is `nofile` which XXX

Figure 6.6 shows a simple case, before a process has opened any files, where the next file descriptor will be determined by what is in "next_fd". For a new process this will be "3" because of `stdin`, `stdout` and `stderr`.

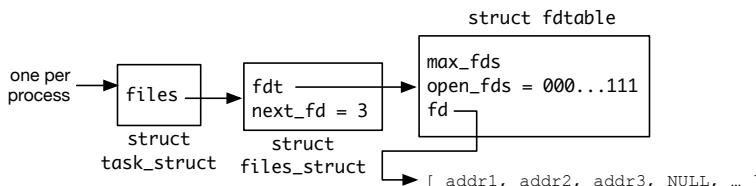


Figure 6.6: Simple fd allocation

Here are the main parts of `alloc_fd()`:

```
static int
alloc_fd(unsigned start, unsigned end, unsigned flags)
{
    struct files_struct *files = current->files;
    struct fdtable *fdt;

    spin_lock(&files->file_lock);
repeat:
    fdt = files_fdtable(files);
    fd = start;
    if (fd < files->next_fd)
```

```

        fd = files->next_fd;

        if (fd < fdt->max_fds)
            fd = find_next_fd(fd, fd);

        error = -EMFILE;
        if (fd >= end)
            goto out;
        ...
        error = expand_files(files, fd);
        if (error)
            goto repeat;
out:
        spin_unlock(&files->file_lock);
        return error;
    }
}

```

The function sets `fd` to start which is "0" (**dup???**) and sets it to `fd_next`. The next line looks confusing but I think that's also related to `expand_files()` in that we may have set `fd` to a value beyond what we can actually store so we'll need to expand. Need to confirm.

I always get confused when I see `expand_files()` and associate it with file structures. We are only talking about expansion of the ability to store more file descriptors.

If we are out of space, the kernel will call `expand_files()` to add more space. There are two checks at the top of the function. They first is to see if space is actually needed. The second checks against the file descriptor limit.

```

static int
expand_files(struct files_struct *files, unsigned int nr)
{
    struct fdtable *fdt;

    fdt = files_fdtable(files);

    if (nr < fdt->max_fds) /* Do we need to expand? */
        return expanded;

    if (nr >= sysctl_nr_open) /* Can we expand? */
        return -EMFILE;

    expanded = expand_fdtable(files, nr);
    return expanded;
}

```

The allocation of a new file descriptor table occurs in `alloc_fdtable()`. A call is then made to `copy_fdtable()` to copy the contents of the old table over to the new.

```

static int
expand_fdtable(struct files_struct *files, unsigned int nr)
{
    struct fdtable *new_fdt, *cur_fdt;

```

```
    new_fdt = alloc_fdtable(nr);
    cur_fdt = files_fdtable(files);
    copy_fdtable(new_fdt, cur_fdt);
}
```

XXX — should I really go much more into detail here? Feels like I should show how the structures are linked or grown etc but perhaps come back here when I know big this damn thing becomes???

6.7.2 Allocating a `file` Structure

After a file descriptor has been allocated and pathname resolution is complete, it's time to allocate a `file` structure. This is done through the following sequence of calls:

```
path_openat() → alloc_empty_file() → __alloc_file()
→ kmem_cache_zalloc(filp_cachep, GFP_KERNEL)
```

This path is very straightforward. Simply allocate and initialize the structure.

6.7.3 KGDB – Viewing File Descriptor Allocation

Section 4.5.1 showed how the kernel structures are linked together after a file had been opened. In this example, we'll walk through the process of opening a file.

First of all we'll set a breakpoint to be hit if the file being opened is "lorem-ipsum" as follows:

```
(gdb) br do_sys_openat2 if $_streq(filename, "lorem-ipsum")
```

Now that the breakpoint is set, we'll execute our command:

```
(gdb) cat "lorem-ipsum"
```

and wait for the breakpoint to be hit at which point we'll check the `filename` argument:

```
Thread 2 hit Breakpoint 17, do_sys_openat2 (dfd=-100,
    filename=0x7ffc4bd32785 "lorem-ipsum",
    how=how@entry=0xfffffc90000fd7e80) at fs/open.c:1261
1261      {
(gdb) p filename
$104 = 0x7ffc4bd32785 "lorem-ipsum"
```

At this point, we'll set a breakpoint in `alloc_fd()` and enter "c" to continue execution. You will need to check the stack backtrace when you hit the breakpoint as other processes will likely be opening files at the same time. When we get to the right processes, you can see the `start` and `end` arguments. Since we are simply opening the file, we'll get the lowest possible file descriptor.

```
(gdb) bt 1
#0  alloc_fd (start=start@entry=0, end=1024, flags=32768)
    at fs/file.c:501
```

Stepping through `alloc_fd()` gets some interesting results. First of all:

```
(gdb) n
501          struct files_struct *files = current->files;
(gdb) p files
$122 = <optimized out>
```

we can't see the value of `files` since it's optimized out. Why this is the case, I don't know. We can single step

```
508          fdt = files_fdtable(files);
(gdb) n
510          if (fd < files->next_fd)
(gdb) p fd
$123 = 0
(gdb) n
514          fd = find_next_fd(fdt, fd);
(gdb) p fd
$124 = 3
```

Note that `fd` is set to the value of `start` (`= 0`) at the start of the function. We find the lowest available file descriptor value (3 as expected) but then we continue as shown below:

```
(gdb) n
521          if (fd >= end)
(gdb) n
524          error = expand_files(files, fd);
(gdb) s
expand_files (files=files@entry=0xfffff888100d26680,
nr=nr@entry=3) at fs/file.c:217
217      {
(gdb) n
225          if (nr < fdt->max_fds)
(gdb) n
246          return expanded;
```

It seems unusual that we'll enter `expand_files()` when we know that we still have space but a check is made at the top of the function so the function returns straightaway. We then set "`files->next_fd = fd + 1`" and return the file descriptor just allocated (3) to the caller.

6.8 Pathname Resolution

This section describes the implementation of *pathname resolution* which takes a pathname and walks through it one component at a time in order to return an inode for the last component. It's one of the more complex pieces of filesystem code in the kernel as it must handle everything from `".` to `..`, symbolic links and potentially moving from one filesystem to another across mount points. It can also result in many calls into one or more filesystems often resulting in reads from disk if the dcache and inode caches do not contain requested

data. It was also one of the most complex paths to follow in the kernel using breakpoints with `gdb` trying to understand how it all worked. XXX

Many Linux system calls take a pathname as an argument. This pathname can either a single name representing one of the Linux file types (regular file, directory, symlink etc) or can consist of one or more *components* separated by "/" (or multiple slashes) and perhaps ending with a file type other than a directory depending on the system call being invoked.

Each process has the notion of a *current working directory* that is assigned when the process is created or inherited from its parent process. When logging into Linux, your current working directory is set to your home directory as specified in `/etc/passwd`. An absolute pathname starts with "/" (the *root directory*) whereas a relative pathname starts with a file or directory that is in the *current working directory*. Pathname resolution starts with an absolute or relative pathname unless one of the `*at` system calls is invoked such as `openat(2)`.

The last chapter covered different kernel structures and caches for file names (`dcache`) and files (`inodes`). The hope is that during pathname resolution, many of the components within the path will already be in one of these caches to avoid the filesystem having to read what could potentially be many different structures from disk.

A large part of opening a file involves resolving the pathname to get to the base filename. Where to start the search depends on how the pathname is formed (relative or absolute as described above). The following information may influence both:

1. A process inherits its root directory from its parent and usually, it will be the root directory of the Linux file hierarchy.
2. A process may get a different root directory through use of the `chroot(2)` system call, or may temporarily use a different root directory by using the `openat2(2)` system call with the `RESOLVE_IN_ROOT` flag set.
3. A process may get an entirely private mount namespace if it, or one of its ancestors, was started by an invocation of the pathname. **XXX – what does this mean?**
4. The current working directory is also inherited from the parent process but this can be changed by using the `chdir(2)` system call.

The `path_resolution(7)` manpage describes the general process of resolving a pathname down to a file. I recommend that you start with this manpage to understand the general process that the kernel will follow before the chapter describes the kernel functions and structures that provide pathname resolution. For now, `path_resolution(7)` describes the process as three steps which are rephrased here:

Step 1 – Start of the resolution process

The first step is to identify where the search starts from. If the pathname starts with a "/" it's an absolute pathname and resolution will start from the root directory of the process unless one of the `*at` system calls is used in which case a new root directory is chosen. If the pathname doesn't start with "/", it's a relative pathname and starts at the current working directory. The *current lookup directory* is set to one of these two directories.

Step 2 – Walking along the path

For each component of the pathname, where a component is a substring delimited by one or more '/' characters, the component is *looked up* in the current lookup directory. If the component is delimited by '/' characters it cannot be the final component.

Step 3 – Finding the final entry

The lookup of the final component of the pathname is no different than all other components, as described in the previous step, other than it has no terminating "/" character. Whether this component is a directory or other file type is not relevant as far as pathname resolution is concerned. Whether it is the correct type of file will be determined by the type of system call and therefore the caller of pathname resolution.

We covered how the initial file descriptor array is stored and referenced from the `task_struct` in section XXX. This is structured in a way that it's quick and easy to get the next file descriptor unless another file table needs to be allocated in which case a call to `expand_files()` is made. Section 4.5.1 also showed how to find file descriptor entries using `gdb` using the first file descriptor table.

6.8.1 Setting the Stage for Pathname Resolution

The current status of pathname resolution is stored in the allocated `nameidata` structure. A function called `namei()` (convert a name to an inode) was originally the name for pathname resolution in UNIX. In fact you can view the source to `namei()` in 6th Edition UNIX in the file `sys/ken/nami.c`. Yes, that's Ken as in Ken Thompson, one of the founders of UNIX and now one of the creators of the language Golang.



`nameidata` is defined in `include/linux/nameidata.h` and `path` is defined in `include/linux/path.h`. Most of the functions described in this section are in `fs/namei.c`.

Here is part of the `nameidata` structure. It has a lot of elements so only the most important fields in the structure will be described:

```
struct nameidata {
    struct path           path;
    struct qstr           last;
    struct path           root;
    struct inode          *inode;
    unsigned int          flags, state;
    unsigned              seq, m_seq, r_seq;
    int                  last_type;
    unsigned              depth;
    int                  total_link_count;
    struct saved {
        struct path       link;
```

```
    struct delayed_call done;
    const char          *name;
    unsigned           seq;
} *stack, internal[EMBEDDED_LEVELS];
struct filename        *name;
struct nameidata       *saved;
unsigned            root_seq;
int                 dfd;
kuid_t              dir_uid;
umode_t             dir_mode;
}
```

Here are the most important fields:

- **path** – this is a path structure contains a `vfsmount` and a `dentry` structure. Together, they record the current status of pathname resolution. Initially the `dentry` field is the starting point of the path walk such as the root directory or the current working directory. The `mnt` field references the filesystem which is currently being traversed. This structure is updated on each step through pathname resolution and potentially cross mount points. For example, if we cross a mount point, `mnt` will be updated to reference the *mounted on* filesystem.
- **last** – this is the next component to search for. Strictly speaking, it's actually the remaining part of the pathname to search for and the `hash len` field is the number of characters in the component that will be searched for. Each time we return from `walk_component()` it gets reduced by one directory component (ignoring symlinks for now). On return from `link_path_walk` it will contain the last component in the path.
- **root** – a reference to the root of the calling process's tree. This is only set for absolute pathnames since it is assumed that most callers of this function will pass relative pathnames. **Why bother?**
- **inode** – the current directory where we are currently looking. The `step_into()` function will result in this field being updated as pathname resolution moves from one directory to another. **why bother? it's in path->dentry**
- **state** –
- **saved** – used for symlink / recursion - explain later.
- **last_type** – `LAST_NORM`, `LAST_ROOT`, `LAST_DOT` or `LAST_DOTDOT`. This field is set to `LAST_ROOT` at the start of pathname resolution. XXX it's not set in many places. go look.

The second structure that is used during pathname resolution is the `path` structure:

```
struct path {
    struct vfsmount *mnt;
    struct dentry   *dentry;
}
```

This structure refers to a component in the path being resolved and records where the kernel is during that process. **I suspect it's the same for mount since a mount path is passed. It's allocated on the stack inside do_mount**

6.8.2 Initialization Structures And Starting The Process

When entering `do_filp_open()`, a call is made to `set_nameidata()` to initialize the `nameidata` structure. Most fields are just initialized to NULL/0 values. There is no attempt at this point to set the starting inode as root (absolute search) or the current working directory (relative search).

Figure 6.7 is a modified version of figure 6.5 and shows how pathname handling code gets access to the root directory for absolute pathnames and the current working directly for relative pathnames. The `fs` field in the `task_struct` references a `fs_struct` structure in which the kernel can easily get access to one or the other through directories.

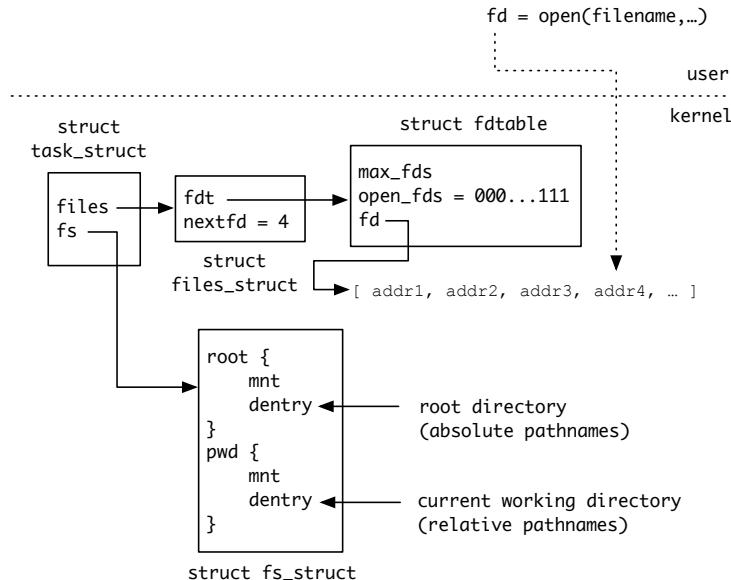


Figure 6.7: TBD

```

struct fs_struct *fs = current->fs;

nd->root = fs->root;                                /* to get the root */

nd->path = fs->pwd;                                 /* to get the CWD */
nd->inode = nd->path.dentry->d_inode;

```

Note that both contain the same fields as the `path` field of the `nameidata` structure.

After initializing the nameidata structure, `filp_open()` calls `path_openat()` which performs additional initialization before calling the main pathname walking function `link_path_walk()`. Here is a fragment of the code:

```
static struct file *
path_openat(struct nameidata *nd,
           const struct open_flags *op, unsigned flags)
{
    struct file *file;
    int error;

    file = alloc_empty_file(op->open_flag, current_cred());
    ...
} else {
    const char *s = path_init(nd, flags);
    while (!(error = link_path_walk(s, nd)) &&
           (s = open_last_lookup(nd, file, op)) != NULL)
    ;
    if (!error)
        error = do_open(nd, file, op);
    terminate_walk(nd);
}
```

The first step is to call `alloc_empty_file()` to allocate a new `file` structure. This will result in a call to `kmem_cache_zalloc()` to allocated the structure and then the fields are initialized.

The final initialization function to call is `path_init()`. Among other things (XXX) this function will locate the root directory or current working directory and set the appropriate fields in the `nameidata` structure. **XXX - it does a lot of other stuff too.**

Once pathname resolution is complete, a call is made to `restore_nameidata()` which restores the old `nameidata` structure that was saved during `set_nameidata()`. **XXX - need to understand why this is being done**

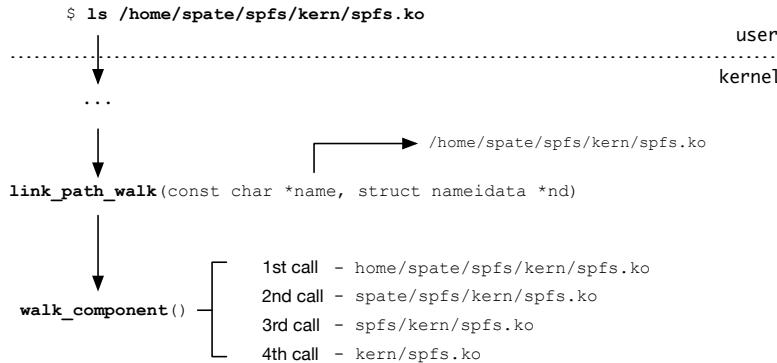
Need to figure out what `current->nameidata` - looks like the current one is saved? help continue from where we are?

6.8.3 The Nitty Gritty of Pathname Resolution

Pathnames can contain a mix of directories and end with either a directory or a regular file depending on which system call is being invoked (and assuming a valid pathname of course). They can also contain the directories `"."` and `".."` referring to the current directory and the directory above where "we are now"

Once everything is initialized, `path_openat()` will call `link_path_walk()` which is the function that iterates through each component of the pathname as shown in figure 6.8. You can see that the `name` argument contains the pathname that is being passed to `ls(2)`.

Inside `link_path_walk()` there are several things to do. First is to remove any `"/"` characters from the front of the remaining path. Next is to check that the directory has

Figure 6.8: Repeated calls to `walk_component()`

execute permission (`may_lookup()`). Then the function loops through the pathname one component at a time until it reaches the last component.

```

static int
link_path_walk(const char *name, struct nameidata *nd)
{
    while (*name=='/')
        name++;

    for(;;) { /* loop for each component */
        mnt_userns = mnt_user_ns(nd->path.mnt);
        err = may_lookup(mnt_userns, nd);

        hash_len = hash_name(nd->path.dentry, name);
        ...
        handle all hash stuff!
        ...

        if (unlikely(!*name)) {
            /* handle symlinks */
            link = walk_component(nd, 0);
        } else {
            /* not the last component */
            link = walk_component(nd, WALK_MORE);
        }
    }
}
  
```

For the last component, `nd->last` and `nd->last_type` are set but no call is made to `walk_component()`. Handling the last component will be done by the caller of `link_path_walk()`.

Really need to figure out all the hashing stuff as it penetrates everything. A call is made to `walk_component()` for each component in the path apart from the last one

(XXX) as this may be a file that doesn't exist and we may be creating it or just stat. Need to find out more here.

Here are the steps performed by `walk_component()`:

```
static const char *
walk_component(struct nameidata *nd, int flags)
{
    struct dentry *dentry;
    struct inode *inode;
    unsigned seq;

    if (unlikely(nd->last_type != LAST_NORM)) {
        if (!(flags & WALK_MORE) && nd->depth)
            put_link(nd);
        return handle_dots(nd, nd->last_type);
    }
    dentry = lookup_fast(nd, &inode, &seq);
    if (unlikely(!dentry)) {
        dentry = lookup_slow(&nd->last, nd->path.dentry,
                           nd->flags);
    }
    if (!(flags & WALK_MORE) && nd->depth)
        put_link(nd);
    return step_into(nd, flags, dentry, inode, seq);
}
```

The `flags` argument can be one of:

- `WALK_TRAILING` – this indicates that we are on the final component of the lookup. A check is made on the user-space flag `LOOKUP_FOLLOW` to determine whether follow it when the last component is a symlink. If this is the case, a call is made to `may_follow_link()` to check if we have privilege to follow the link.
- `WALK_MORE` – this flag indicates that it is yet too early to release the current symlink. This will further described in section 6.8.8.
- `WALK_NOFOLLOW` – this flag specifies that if a symlink is found it should not be followed.

The dentry for the component being looked up will either already be in the dcache or a call will need to be made into the filesystem. To check if it's in the cache, `walk_component()` calls `lookup_fast()` and if that fails to find it in the cache it then calls `lookup_slow()`.
XXX - it says "unlikely" before calling slow implying ... well ..?? Really looks like a locking issue too. fast takes less locks. slow takes more. just google search.

To see if the dentry is already in the dcache, a call is made to `lookup_fast()` which in turn calls `__d_lookup_rcu()`. Inside this function, the right dcache hash bucket is located and scanned to see if there is a matching dentry:

```
struct dentry *
__d_lookup_rcu(const struct dentry *parent,
```

```

                const struct qstr *name, unsigned *seqp)
{
    u64 hashlen = name->hash_len;
    const unsigned char *str = name->name;
    struct hlist_bl_head *b = d_hash(hashlen_hash(hashlen));
    struct hlist_bl_node *node;
    ...
    hlist_bl_for_each_entry_rcu(dentry, node, b, d_hash) {
        ...
        if (dentry_cmp(dentry, str,
                        hashlen_len(hashlen)) != 0)
            continue;
        }
    *seqp = seq;
    return dentry;
}

```

The `str` argument passed to `dentry_cmp()` is the remaining path in our search but we only compare against the correct number of characters for the current component. For example, `str` could be "spfs/kern/spfs.ko" but `hashlen_len(hashlen)` will return "4" which is the length of "spfs".

As a reminder, we have traversed through the following path trying to find the current component in the dcache:

```

link_path_walk() → walk_component() → lookup_fast()
→ __d_lookup_rcu()

```

Single stepping through `__d_lookup_rcu()`, we hit the line where we're comparing the string we're looking for against dentries in the hash bucket where it will be found (if present):

```
(gdb) n
2369      if (dentry_cmp(dentry, str, hashlen_len(hashlen)) != 0)
```

Here is the string being passed:

```
(gdb) p str
$3 = (const unsigned char *) 0xfffff888100f90021
"home/spate/spfs/kern/spfs.ko"
```

But we only want to compare each dentry found against the string "home". As mentioned above, `hashlen_len(hashlen)` will return "4" allowing us to compare only what we need and not the rest of the string.

Assuming the dentry is found (fast or slow path), `walk_component()` makes a call is made to `step_into()` which calls `handle_mounts()` to see if a mount point is being crossed. In this case, a new path structure is created which references to dentry for the root of the mounted-on filesystem and a reference to the new vfsmount.

It also handles the case of the current component being a symbolic link in which a call is made to `pick_link()` to handle it. If neither of these cases exits, `nd->path` is updated and return is made to `walk_component()`. Oddly enough, if we are not

crossed a mount point, it is still `handle_mounts()` that updates `path->dirent` to the new directory from where to continue searching.

6.8.4 Callers of `link_path_walk()`

There are only three callers of `link_path_walk()` but these three functions are in turn used by many paths throughout the kernel as shown in figure 6.9.

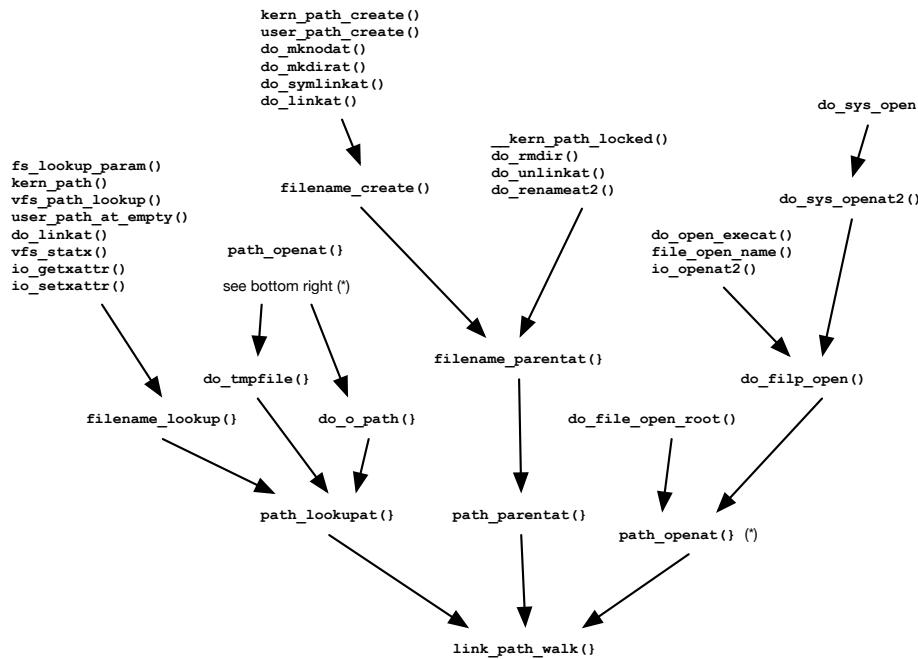


Figure 6.9: Callers of `link_path_walk()`

This figure will come in helpful when following several system call paths. For example, callers of `filename_create()` are typically system calls that involve creating a file such as a directory, symbolic link or hard link. For `open(2)`, `creat(2)` and similar system calls, see the right hand side of the figure.

6.8.5 More on Hashing

Described the algorithms in more detail

6.8.6 Handling a dcache Miss

If `__d_lookup_rcu()` fails to find the name in the dcache, `__lookup_slow()`, a call will likely be made into the filesystem to read the component from disk. It is "likely" because a check is made again to see if the name is in the dcache before a `lookup()` filesystem call is made. The extra dcache check is made because the first scan takes place without locks and the file could have been added in the meantime. This time however, a shared lock (`i_rwsem`) is taken on the directory inode. Entries are added to the dcache by taking an exclusive lock on `i_rwsem`.

```

static struct dentry *
__lookup_slow(const struct qstr *name,
             struct dentry *dir,
             unsigned int flags)
{
    struct dentry *dentry, *old;
    struct inode *inode = dir->d_inode;

    dentry = d_alloc_parallel(dir, name, &wq);
    if (unlikely(!d_in_lookup(dentry))) {
        int error = d_revalidate(dentry, flags);
        ...
    } else {
        old = inode->i_op->lookup(inode, dentry, flags);
        d_lookup_done(dentry);
        if (unlikely(old)) {
            dput(dentry);
            dentry = old;
        }
    }
    return dentry;
}

```

XXX – need to flush out this stuff

6.8.7 Crossing mount points

Before digging back into the source code, consider the file hierarchy in figure 6.10: The root filesystem has a directory called "mount-dir" on to which a filesystem is mounted. Inside this mounted filesystem there are two files. Figure 6.11 shows the dentries for the five files shown here. The `d_flags` field is set to `DCACHE_MOUNTED` indicating that a filesystem is mounted on top of it. Strictly speaking, this is a hint and is not always set but we will cover that later.

All dentries representing directories maintain a doubly linked list headed by the `d_subdirs` field that runs through the child dentries' `d_child` fields. Each child's `d_parent` pointer points to its parent unless it's the root dentry in which case it points to itself (see the `IS_ROOT()` macro in `include/linux/dcache.h`). Note that there is no direct pointer from the `mount-dir` dentry to the root of the SPFS filesystem. That is done

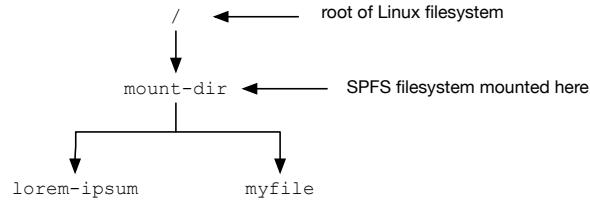


Figure 6.10: Simple file hierarchy with a mounted filesystem

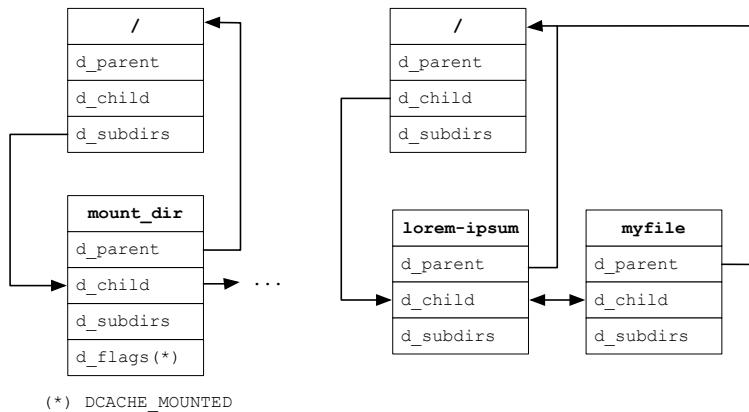


Figure 6.11: dentries for the simple file hierarchy

through `mount_hashtable` and will be covered below when we look further into the code.

As we scan through the pathname one component at a time, we lookup its dentry. If this is a directory and a filesystem is mounted on top of it, it's easy to determine by checking the dentry flags field. There can be one of the following flags that relate to mounted filesystems:

- `DCACHE_MOUNTED` – this flag indicates that there is a filesystem mounted on top of this directory.
- `DCACHE_NEED_AUTOMOUNT` – if this flag is set we need to automount a filesystem on this directory before we can traverse into it.

There is no direct link from the dentry for `mount-dir` to the root of the mounted filesystem. So even though the dentry for `mount-dir` has the `DCACHE_MOUNTED` flag set, XXX

SVR4-based UNIX systems encoded which filesystems were mounted where through the use of fields in the `vnode` structure (equivalent of the Linux `inode` structure. Linux is different in that this information is recorded in the `dcache` with the `dentry` structure.

NOTE — XXX

This is used to get to a root dentry from a mounted on dentry (DCACHE_MOUNTED) need to confirm this

Here is the path followed from `walk_component()`:

```
walk_component() → step_into() → handle_mounts()
→ traverse_mounts() → __traverse_mounts()
```

`pick_link()` handles symlinks.

The path structure is defined in `include/linux/path.h` and contains two members:

```
struct path {
    struct vfsmount *mnt;      - which FS tree we are in now
    struct dentry *dentry;     - root node or is it the CWD?
}
```

where the `dentry` references the root of the filesystem.

Here is the source code in `__traverse_mounts()` that covers a filesystem mounted on this directory. If we get a `vfsmount` back from `lookup_mnt()`, we release the current dentry where we're searching (`dput(path->dentry)`) and set it to the the dentry for the root of the filesystem that's mounted on top of this directory (`mounted->mnt_root`).

```
if (flags & DCACHE_MOUNTED) {    // something's mounted on it..
    struct vfsmount *mounted = lookup_mnt(path);
    if (mounted) {        // ... in our namespace
        dput(path->dentry);
        if (need_mntput)
            mntput(path->mnt);
        path->mnt = mounted;
        path->dentry = dget(mounted->mnt_root);
        flags = path->dentry->d_flags;
        need_mntput = true;
    }
}
```

The `lookup_mnt()` function in turn calls `__lookup_mnt()` which walks the mount hash table to locate the root dentry for the mounted filesystem:

```
struct mount *
__lookup_mnt(struct vfsmount *mnt, struct dentry *dentry)
{
    struct hlist_head *head = m_hash(mnt, dentry);
    struct mount *p;

    hlist_for_each_entry_rcu(p, head, mnt_hash)
        if (&p->mnt_parent->mnt == mnt &&
            p->mnt_mountpoint == dentry)
            return p;
    return NULL;
}
```

The goal of `m_hash()` is to return the correct hash bucket to search for the mount XXX we need. `mount_hashtable` is defined in `namespace.c` and described in section **dcache-links**.

Here are the first few fields of the mount structure:

```
struct mount {
    struct hlist_node mnt_hash;
    struct mount *mnt_parent;
    struct dentry *mnt_mountpoint;
    struct vfsmount mnt;
    ...
}
```

so the mount structures are held in a list accessed through the first field `mnt_hash`.

Back in `__traverse_mounts()`, once the new mount is found, the path structure is updated to reference the new filesystem that we are entering into as well as the dentry for the root of this tree.

6.8.8 Handling Symlinks

XXX

6.8.9 Returning From `link_path_walk()`

Before each call to `walk_component()`, table 6.1 shows the contents of `nd->last` and the local variable "name".

Call	<code>nd->last.name</code>	<code>name</code>
1	home/spate/spfs/kern/spfs.ko	spate/spfs/kern/spfs.ko
2	spate/spfs/kern/spfs.ko	spfs/kern/spfs.ko
3	spfs/kern/spfs.ko	kern/spfs.ko
4	kern/spfs.ko	spfs.ko

Table 6.1: Standard File Descriptors

Before returning from `link_path_walk()` the `last.name` field of `nd` will be set to `name` and `last.hash_len` set to the length of the string. This allows the caller to deal with the last component.

There are three callers of `link_path_walk()` namely:

- `path_openat()` – this is used for the `open(2)` system call and is more complex. It will be covered XXX.
- `path_parentat()` – this function returns the parent directory and the final component to its caller which will typically be creating, removing or renaming a file.
- `path_lookupat()` – operations such as `stat(2)` come in through this function. It will call `lookup_last()` which in turn calls `walk_component` for the last component passing `WALK_TRAILING` as the second argument.

The `path_parentat()` and `path_lookupat()` functions are described in section XXX. Since this chapter started with opening a file, let's look at what `path_openat()` does on return from `link_path_walk()`. Here is the code surrounding the call to `link_path_walk()`:

```
while (!error = link_path_walk(s, nd)) &&
    (s = open_last_lookups(nd, file, op)) != NULL)
{
    if (!error)
        error = do_open(nd, file, op);
    terminate_walk(nd);
```

The function loops around the calls to `link_path_walk()` and `open_last_lookups()` since the last component could be a symlink in which case another call will be made to `link_path_walk()`. For the non symlink case, `open_last_lookups()` calls `lookup_fast()` to see if the last component is in the dcache. If not a call is made to `lookup_open()` to open the file. **XXX — need to come back here once I understand where file creation doc will go. there is another section**

6.8.10 RCU-walk vs REF-walk

It was mentioned at the start of the section that pathname resolution was one of the more complex pieces of code in the Linux kernel and despite describing the process at depth, a lot of detail has still been left out. There are a lot of locks to be taken along the way and if components aren't in the dcache, that could mean a lot of calls into the filesystem and typically to disk.

There are actually two modes of pathname resolution, one that attempts to speed things up and a fallback when the fast path isn't available. These modes are:

- RCU-walk – introduced in 2.6.38, RCU allows for a large part of pathname resolution to be performed without taking locks. As on-line documentation states, it allows "*a significant part of the entire path walk (including dcache look-up) completely “store-free” (so, no locks, atomics, or even stores into cachelines of common dentries).*"
- RCU-walk – when the kernel can't continue in RCU mode, it drops into REF-walk which is the older method of pathname resolution whereby locks are taken as needed. This results in poor performance on multi-processor architectures where multiple threads can be competing for locks at the same time.

XXX

It's all to do with the *sequence count*. You will see references to a sequence count inside `lookup_fast()` when called by `walk_component()` as follows:

```
dentry = lookup_fast(nd, &inode, &seq);
```

XXX - there are comments in `lookup_fast()` indicating that this is really to do with things "changing" underneath us which makes sense if few locks are taken

There are some stats in the following page - see how to get them

one of the docs talks about the callers of `path_link_walk` and talks about what each does with the last component. take a look at that and write about it.

6.8.11 KGDB – Setting The Stage For Pathname Resolution

This example will show how the kernel is entered, everything is set up and the pathname resolution code is entered through a call to `link_path_walk()`. There are only three callers of `link_path_walk()` but multiple callers of these three functions.

We'll be starting with `filp_open()` as shown in figure 6.12.

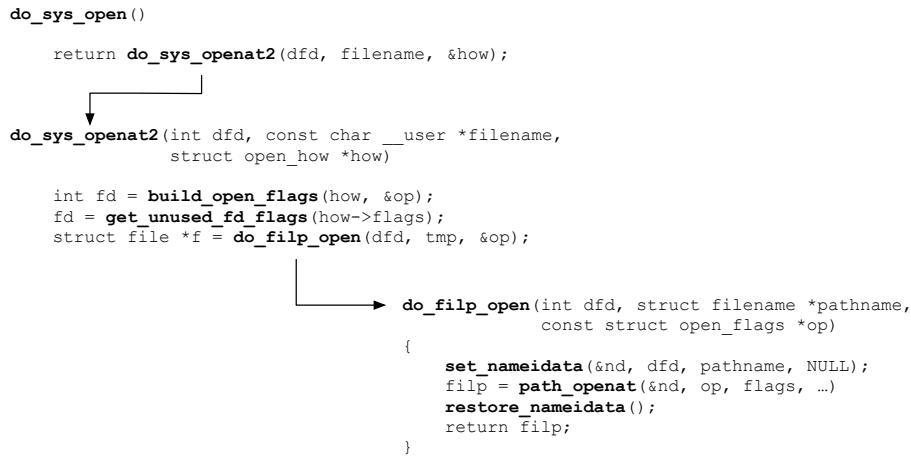


Figure 6.12: System call handling for opening a file

Linux processes are invoking system calls all of the time therefore setting a breakpoint in `gdb` for one of the more popular system calls (file access for example) will likely be hit by several processes in addition to the one you're tracking. And this is especially true of pathname resolution which is used by multiple different system calls.

Therefore to set a breakpoint you will want to set a *conditional breakpoint*. You can set a breakpoint anywhere but we'll set one in `do_filp_open()` after a file descriptor has been allocated:

```
struct file *do_filp_open(int dfd, struct filename *pathname,
                          const struct open_flags *op)
```

The `filename` structure has a field called `name` which points to the pathname being looked up. To make sure a breakpoint can be easily set, the following program will be run:

```
$ cat lorem-ipsum
```

and prior to running the program, a conditional breakpoint can be set as follows ensuring that the breakpoint will only be triggered if a process is accessing the file `lorem-ipsum`. Here is how the breakpoint is set:

```
(gdb) br do_filp_open if $_streq(pathname->name, "lorem-ipsum")
```

When the program runs and the breakpoint is triggered, you will see the following in gdb:

```
Thread 3 hit Breakpoint 4, do_filp_open (dfd=dfd@entry=-100,
 pathname=pathname@entry=0xfffff88810263b000,
 op=op@entry=0xfffffc90000fc3dd4) at fs/namei.c:3678
3678      {
(gdb)
```

Before going further, we can look at the `task_struct` for this process to see the root directory and current working directory as follows. One or the other will be used for pathname resolution depending on whether it's an absolute or relative pathname.

```
(gdb) p $lx_current()->fs->root.dentry->d_name.name
$40 = (const unsigned char *) 0xfffff88812f9ad578 "/"
(gdb) p $lx_current()->fs->pwd.dentry->d_name.name
$39 = (const unsigned char *) 0xfffff88811f163878 "spate"
```

The stack backtrace is shown below which will correspond to the calls shown in figure 6.12. All functions are shown in bold. You can see the low-level functions called as part of system call handling and resulting in a call to `do_sys_open()`.

```
(gdb) bt 3
#0  do_filp_open (dfd=dfd@entry=-100,
 pathname=pathname@entry=0xfffff888100870000,
 op=op@entry=0xfffffc90000d57e04) at fs/namei.c:3678
#1  do_sys_openat2 (dfd=-100, filename=<optimized out>,
 how=how@entry=0xfffffc90000d57e48) at fs/open.c:1275
#2  do_sys_open (mode=<optimized out>,
 flags=<optimized out>, filename=<optimized out>,
 dfd=<optimized out>) at fs/open.c:1291
(More stack frames follow...)
```

The `list` command shows where gdb is at this moment in time:

```
(gdb) list
3673          return ERR_PTR(error);
3674      }
3675
3676      struct file *do_filp_open(int dfd, struct filename
3677          *pathname, const struct open_flags *op)
3678      {
3679          struct nameidata nd;
3680          int flags = op->lookup_flags;
3681          struct file *filp;65wq
3682
```

I like to use `list` in conjunction with having the source code in an editor or using Elixir so that more of the source code is visible.

To keep walking through the code, use a combination of "n" or "s" depending on whether you want to skip over a function. Below, we don't want to enter the function `set_nameidata()` but we do want to step into `path_openat()`.

```
(gdb) n
3679         struct nameidata nd;
(gdb) n
3683         set_nameidata(&nd, dfd, pathname, NULL);
(gdb) n
610         p->state = 0;
(gdb) n
3683         set_nameidata(&nd, dfd, pathname, NULL);
(gdb) n
3684         filp = path_openat(&nd, op, flags | LOOKUP_RCU);
(gdb) s
path_openat (nd=nd@entry=0xfffffc90000ed3d30,
    op=op@entry=0xfffffc90000ed3e54, flags=flags@entry=65)
    at fs/namei.c:3639
3639 {
```

At this point, there isn't a lot of information in `nd` although the file we're looking for can be found through the `name` field as follows:

```
(gdb) p nd->name->name
$35 = 0xfffff888100870020 "lorem-ipsum"
```

Skipping ahead a little further:

```
(gdb) n
15         return this_cpu_read_stable(current_task);
(gdb) n
36         return IS_ERR_VALUE((unsigned long)ptr);
(gdb) n
3647         if (unlikely(file->f_flags & __O_TMPFILE)) {
(gdb) n
3649         } else if (unlikely(file->f_flags & O_PATH)) {
(gdb) n
3652             const char *s = path_init(nd, flags);
(gdb) n
3653             while (!(error = link_path_walk(s, nd)))
```

We stepped over the call to `path_init` which set up the `nameidata` structure with the information it needs to start searching. Here are some of the relevant fields:

```
(gdb) p *nd
$37 = {
    path = {
        mnt = 0xfffff8881001dd2e0,
        dentry = 0xfffff888103fa60c0
    },
    ...
    inode = 0xfffff888102c57a38,
    ...
}
```

Earlier in this section we showed the root and current working directories. We know that the path (`lorem-ipsum`) is relative and therefore we use the directory referenced by `fs->pwd.dentry` as the starting base:

```
(gdb) p $lx_current()->fs->root.dentry
$38 = (struct dentry *) 0xfffff88812f8aa000
(gdb) p $lx_current()->fs->pwd.dentry
$39 = (struct dentry *) 0xfffff888103fa60c0
(gdb) p $lx_current()->fs->pwd.dentry->d_inode
$40 = (struct inode *) 0xfffff888102c57a38
```

The dentry stored in `nd->path->dentry` is the same as `fs->pwd.dentry` in the task structure and this dentry and `nd->inode` reference the same directory inode. Take time to match the values shown here.

6.8.12 KGDB – Inside `link_path_walk()` and Friends

We know from the previous example that by the time we enter `link_path_walk()`, the fields of the `nameidata` structure have been initialized, in particular:

- `path.mnt` – the `vfsmount` for the starting filesystem
- `path.dentry` – the starting directory dentry
- `path.inode` – the inode for the starting directory
- `path.name.name` – the path to search for

We want to set a breakpoint in `link_path_walk()` but only if we see a specific path in response to the following command being called since this is a function that is called all the time:

```
$ ls /home/spate/spfs/kern/spfs.ko
$ ls -ld /
2 drwxr-xr-x 19 root root 4096 Apr 10 23:48 /
```

The root directory inode number (2) is also displayed.

Here is how we can set a breakpoint to hit only if "name" points to a specific pathname as follows:

```
(gdb) br link_path_walk if $_streq(name, \
                                     "/home/spate/spfs/kern/spfs.ko")
Breakpoint 1 at 0xffffffff813fb9a0: link_path_walk. (4 locations)
```

At this point, let's look at the contents of `nd`:

```
(gdb) p nd->path
$47 = {
    mnt = 0xfffff8881001dd2e0,
    dentry = 0xfffff88812f8aa000
}
(gdb) p nd->path->dentry->d_name.name
```

```
$53 = (const unsigned char *) 0xffff88812f8aa038 "/"
(gdb) p nd->path->dentry->d_inode
$52 = (struct inode *) 0xffff88812f8f13c8
(gdb) p nd->root
$48 =
{
    mnt = 0xffff8881001dd2e0,
    dentry = 0xffff88812f8aa000
}
(gdb) p nd->inode
$49 = (struct inode *) 0xffff88812f8f13c8
(gdb) p nd->inode->i_ino
$51 = 2
```

Thus, everything displayed shows that this is the root directory from which the search will begin. The pathname that is being searched for is here:

```
(gdb) p nd->name.name
$57 = 0xffff888100e6e020 "/home/spate/spfs/kern/spfs.ko"
```

Unlike the last example, the `root` field has been set since we are looking up an absolute pathname (why it matters i don't know).

Anyway, to the first breakpoint:

```
(gdb) bt
#0  link_path_walk  (nd=0xfffffc90000eebc70,
                     name=0xffff888100e6e020 "/home/spate/spfs/kern/spfs.ko")
                     at fs/namei.c:2488
#1  path_lookupat  (nd=nd@entry=0xfffffc90000eebc70,
                     flags=flags@entry=68,
                     path=path@entry=0xfffffc90000eebda8) at fs/namei.c:2494
#2  filename_lookup  (dfd=dfd@entry=-100,
                     name=name@entry=0xffff888100e6e000,
                     flags=flags@entry=4,
                     path=path@entry=0xfffffc90000eebda8,
                     root=root@entry=0x0 <fixed_percpu_data>) at fs/namei.c:2524
#3  vfs_statx  (dfd=dfd@entry=-100,
                 filename=filename@entry=0xffff888100e6e000,
                 flags=flags@entry=256,
                 stat=stat@entry=0xfffffc90000eebdf8,
                 request_mask=request_mask@entry=2) at fs/stat.c:228
#4  do_statx  (dfd=dfd@entry=-100,
                 filename=filename@entry=0xffff888100e6e000,
                 flags=flags@entry=256,
                 mask=mask@entry=2, buffer=buffer@entry=0x7ffc4924f4c0)
                 at fs/stat.c:629
```

At this point `nd->last` is empty

The first bit of code:

```
while (*name=='/')
    name++;
```

strips of the first "/" so name becomes:

```
(gdb) p name
$22 = 0xfffff88812edbb021 "home/spate/spfs/kern/spfs.ko"
```

Hashing - each time through hash_len seems to be the correct number of characters of the component to search for. For example, first time through it's 4 (home), then 5 (spate) and so on.

A call is made to walk_component (nd, WALK_MORE) and then the following set of calls are made to see if the component to search for is in the dcache:

```
walk_component () → lookup_fast () → __d_lookup_rcu()
```

This is the first call into walk_component () (searching for "home"). The dentry is found in the dcache and returned. We can check its name:

```
(gdb) p dentry->d_name.name
$86 = (const unsigned char *) 0xfffff88811b5e7338 "home"
```

We keep calling through to walk_component () and get a dcache hit for "spate" but on the next call to lookup_fast (), we get NULL back telling us that "spfs" is not in the dcache.

```
2012          dentry = lookup_fast(nd, &inode, &seq);
(gdb) n
36          return IS_ERR_VALUE((unsigned long)ptr);
(gdb) p dentry
$4 = (struct dentry *) 0x0 <fixed_percpu_data>
```

Now a call to lookup_slow() will be made which locks the parent inode that we're searching and calls __lookup_slow():

```
static struct dentry *
lookup_slow(const struct qstr *name, struct dentry *dir,
           unsigned int flags)
{
    struct inode *inode = dir->d_inode;
    struct dentry *res;

    inode_lock_shared(inode);
    res = __lookup_slow(name, dir, flags);
    inode_unlock_shared(inode);
    return res;
}

static struct dentry *
__lookup_slow(const struct qstr *name, struct dentry *dir,
              unsigned int flags)
{
    ...
    if (unlikely(!d_in_lookup(dentry))) {
```

```
        int error = d_revalidate(dentry, flags);
        ...
    } else {
        old = inode->i_op->lookup(inode, dentry, flags);
        d_lookup_done(dentry);
    }
    return dentry;
}
```

There is a final attempt to see if the entry has been added to the dcache since we last checked and if not, there will be a call into the filesystem to read the inode.

For this example, we have an SPFS filesystem mounted and we'll add a breakpoint in the `sp_lookup()` function:

```
(gdb) bt 4
#0  sp_lookup  (dip=0xfffff888104a9c500, dentry=0xfffff888104aca780,
   flags=4) at /home/spate/spfs/kern/sp_dir.c:386
#1  __lookup_slow  (name=name@entry=0xfffffc9000078fc70,
   dir=dir@entry=0xfffff888104acaa80, flags=flags@entry=4)
   at fs/namei.c:1703
#2  lookup_slow  (flags=4, dir=0xfffff888104acaa80,
   name=0xfffffc9000078fc70) at fs/namei.c:1720
#3  walk_component  (nd=nd@entry=0xfffffc9000078fc60,
   flags=flags@entry=1) at fs/namei.c:2016
```

When this function is entered, you can see the name of the component that the filesystem needs to lookup. Since this operation is about to be performed, there is no inode associated with the dentry at this point.

```
(gdb) p dentry->d_name.name
$13 = (const unsigned char *) 0xfffff888104aca7b8 "spfs"
(gdb) p dentry->d_inode
$14 = (struct inode *) 0x0 <fixed_percpu_data>
```

You can see how SPFS handles the lookup operation in section 7.8.

6.8.13 KGDB – Crossing Mount Points

This example shows what happens during pathname resolution when crossing a mount point. In the root filesystem we have the directory `/mnt/mount_dir` on to which a filesystem is mounted as follows:

```
$ sudo mount -t spfs /dev/loop0 /mnt/mount_dir
```

We are going to search for a file as follows:

```
$ ls /mnt/mount_dir/lorem-ipsum
```

Once again, we'll be setting a breakpoint in `link_path_walk()` to check for a specific path and also in `sp_lookup()` to see when a call is made into the filesystem::

```
(gdb) br link_path_walk if $_streq(name, \
                                     "/mnt/mount_dir/lorem-ipsum")
(gdb) br sp_lookup
```

As we resolve the pathname, we will see dentries for the following:

1. root directory - "/"
2. mount_dir
3. root directory for the SPFS filesystem
4. lorem-ipsum

As we hit the first breakpoint:

```
(gdb) bt 1
#0  link_path_walk  (name=name@entry=0xfffff888100929020
                     "/mnt/mount_dir/lorem-ipsum", nd=nd@entry=0xfffffc900007cf80)
                     at fs/namei.c:2270
```

Let's check the address nd:

```
(gdb) p nd
$18 = (struct nameidata *) 0xfffffc900007cf80
```

We know that as we iterate through the pathname, we should see a call path as follows when handling mount points:

```
link_path_walk() → walk_component() → step_into()
                           → handle_mounts()
```

so using the address of nd, we can set a breakpoint as follows:

```
(gdb) br handle_mounts if nd = 0xfffffc900007cf80
```

Then we can let the flow continue until the breakpoint is hit. We'll need to skip the first time it stops (to handle "/"). On the second occurrence, we can confirm that we have the stack backtrace that we expect:

```
(gdb) bt 4
#0  handle_mounts  (seqp=<optimized out>, inode=<optimized out>,
                    path=<optimized out>, dentry=<optimized out>,
                    nd=<optimized out>)  at fs/namei.c:1528
#1  step_into  (nd=nd@entry=0xfffffc900007cf80,
                flags=flags@entry=2, dentry=0xfffff888100722d80,
                inode=0xfffff8881306ba1c0, seq=2) at fs/namei.c:1846
#2  walk_component  (nd=nd@entry=0xfffffc900007cf80,
                     flags=flags@entry=2) at fs/namei.c:2022
#3  link_path_walk  (name=0xfffff888100929025
                     "mount_dir/lorem-ipsum", name@entry=0xfffff888100929025
                     "/mnt/mount_dir/lorem-ipsum",
                     nd=nd@entry=0xfffffc900007cf80) at fs/namei.c:2343
```

None of the arguments look useful at this stage and the reason why is beyond my level of `gdb` knowledge and likely this book. However, just single step and you'll get a better stack backtrace:

```
(gdb) s
handle_mounts (seqp=<synthetic pointer>,
    inode=<synthetic pointer>, path=0xfffffc900007cfb38,
    dentry=0xfffff8881031e83c0, nd=0xfffffc900007cfc80)
at fs/namei.c:1523
```

Hmm! Stuck!

6.8.14 Further Information on Pathname Resolution

It took me several weeks to walk through the pathname resolution code to write this section. It was a process that would have been much more complicated without the use of `gdb` to step through the paths in the kernel again and again. But there is still a level of detail that has been ignored. Some of that is covered by on-line documentation which describes the details but not so well the flow and certainly not with examples and diagrams.

There are some very good articles on pathname resolution that are worthy of reading in the Linux source tree. Just look in the `Documentation/filesystems` directory for `path-lookup.rst`. You can also go to this webpage:



URL 20 <https://tinyurl.com/4rd8ympk>

and then look for the section labeled "*Pathname lookup*". The article is based on three articles first published on `lwn.net`.

XXX – need to say what it covers that I don't

6.9 Linux Dcache Implementation

Section 4.4 introduced the Linux dcache and described the main structures involved including the `dentry` structure and the hash buckets referenced by the global variable `dentry_hashtable`. This section looks at the implementation of the dcache covering the `dentry` state diagram, algorithms for handling dentries and how the dcache is pruned when the system is under memory pressure.



The source code for the dcache can be found in `fs/dcache.c` and the header file `include/linux/dcache.h` which includes the `dentry` structure.

There are many facets to the dcache and all details can't be covered here. There is certainly enough material to be able to understand how the dcache works and be able to explore further adding `gdb` breakpoints to see the transition of dentries from one state to another.

6.9.1 The Migration to RCU

Seems like the right place but need to understand it first

Starting with <https://www.linuxjournal.com/article/7124> - Scaling dcache with RCU

6.9.2 Linking Dentries Together

Understanding how the dcache works involves understanding how one dentry is related to another. This section explores the various lists, the dcache itself and LRU lists.

Here are the fields in the `dentry` structure that are lists that a dentry can be on:

- `d_hash` – the dcache, accessed through `dentry_hashtable` is a collection of hash buckets accessed using `d_hash()` using the file name and the parent (XXX). The hash collision list has links through this field.
- `d_lru` – all unused dentries are collected on an LRU list linked through this field. Note that for most of their lifespan, dentries are on the hash list and LRU list at the same time. Only if the dentry is removed, perhaps due to memory pressure, will it be moved off the hash list.
- `d_child / d_subdirs` – each dentry representing a directory maintains a doubly linked circular list referenced by the `d_subdirs` field which runs through the child dentry's `d_child` fields. For each child's dentry in this list, the `d_parent` field points to the same parent. Note that for the root directory (""/"), `d_parent` points to itself and can be tested with `IS_ROOT()`.
- `d_alias` – all dentries that belong to the same inode are collected in a list headed by the `inode` structure field `i_dentry` and with links through the dentry field `d_alias`.

There is one more list hanging off each mounted filesystem's `super_block` structure:

```
struct list_lru      s_dentry_lru;
```

This is the actual LRU list (one per mounted filesystem) which allows easy access to per-filesystem dentries which is required when the filesystem is being unmounted and during purging under memory pressure. For more details on this see section 6.9.6.

The actual dcache itself is a list of hash buckets referenced by `dentry_hashtable` which is defined in `fs/dcache.c`:

```
static struct hlist_bl_head *dentry_hashtable;

static inline struct hlist_bl_head *d_hash(unsigned int hash)
{
    return dentry_hashtable + (hash >> d_hash_shift);
```

There is a kernel tunable called `dhash_entries` which determines the number of hash buckets. On different VMs here are two different values for this variable. First using gdb:

```
(gdb) p dhash_entries
$184 = 14757395258967641292
```

and on a different VM using crash(1)

```
crash> dhash_entries
dhash_entries = $2 = 9584312021952758120
```

so a very big difference (50% more) and ridiculously huge anyway! XXX there is no doc on this and the code isn't easy to understand so may skip it

To locate the appropriate hash bucket callers invoke `d_hash()`

```
static inline struct hlist_bl_head *
d_hash(unsigned int hash)
{
    return dentry_hashtable + (hash >> d_hash_shift);
}
```

TBD - see `__d_rehash()` info below for more info.

6.9.3 The `dentry` State Diagram

An old Linux Journal article, "Scaling dcache with RCU" [8], had a very nice state diagram showing how dentries moved from one state to another based on the different dcache functions called. A variant of this is shown in figure 6.13 bringing the figure up to date. The functions that result in transition from one state to another are also shown. The details of these functions will be described in subsequent sections.

Just to be clear, there are three states that say "hashed". Just to be clear, in all of these three states, the dentry is hashed through the `d_hash` field. Here are the details of each transition:

XXX — if I keep a file open (less file) will it still be on the hashed list ?

Step 1

The first way that a dentry comes into existence is with a call to `d_alloc()` which in turn calls `__d_alloc()`. This calls `kmem_cache_alloc_lru(dentry_cache, ...)` and initializes the fields of the `dentry` structure. You will see how the name is stored (whether to use `d_iname` or `d_name` depending on the size of the file name (`DNAME_INLINE_LEN`)).

Before a return is made from `d_alloc()`, the dentry will be added to the parent's `d_subdirs` list. This list goes through the children's `d_child` field. At this point the dentry is not hashed.

Step 2

The second way that a dentry comes into existence is if the parent of this new entry is not in the dcache. In this case, an *anonymous dentry* will be created. This occurs is when mounting a filesystem. Since the inode for the root of the tree has no direct parent, the filesystem will read in the filesystem's root inode and call `d_make_root()` which in

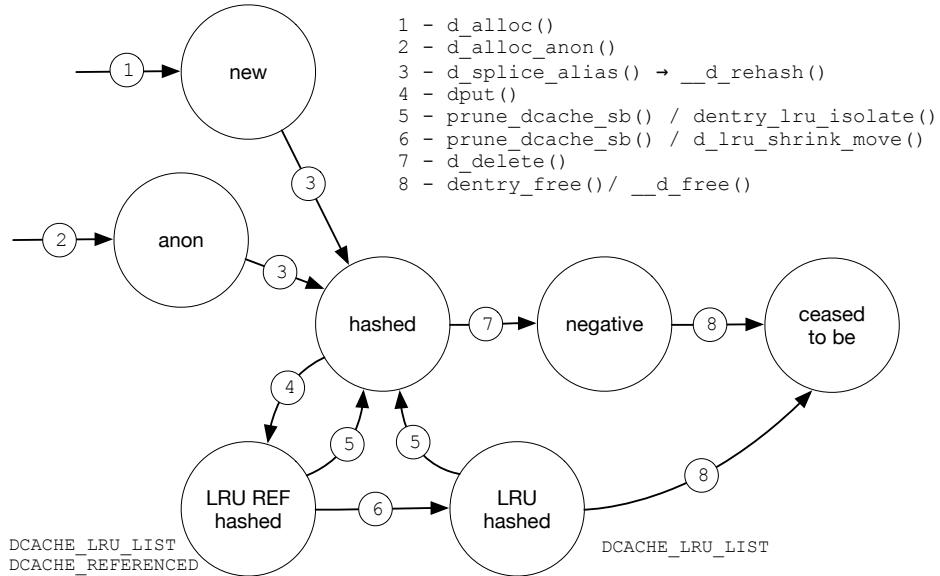


Figure 6.13: The dentry state diagram

turn calls `d_alloc_anon()` to create the associated dentry. This function is typically called by the filesystem's `fill_super` function.

Step 3

After a dentry is allocated, a call is typically made to the filesystem's `lookup` function which locates and reads in the inode and then calls `d_splice_alias()` to associate the dentry with the newly created inode. The filesystem's `lookup` function will be part of the `inode_operations` structure that the filesystem attaches to the `i_op` field of the `inode` structure for a directory file type.

The following sequence of calls occurs:

filesystem → `d_splice_alias()` → `__d_add()` → `__d_rehash()`

The function `__d_rehash()` adds the dentry to the appropriate hash bucket:

```
static void
__d_rehash(struct dentry *entry)
{
    struct hlist_node *b = d_hash(entry->d_name.hash);

    hlist_node_lock(b);
    hlist_node_add_head_rcu(&entry->d_hash, b);
    hlist_node_unlock(b);
}
```

This is the only function that adds a dentry to the hash table. Other callers will be described in a later section.

Step 4

When a file is no longer needed to perform the particular task required, `dput()` is called which decrements the dentry's reference count. If it drops to zero, it will be moved to the LRU list and may be reclaimed later if there is a memory shortage. `DCACHE_LRU_LIST` is added to `d_flags` to show that the dentry is on the LRU list. A call is also made to `retain_dentry()` to see if the dentry should be retained. If this is the case, the `DCACHE_REFERENCED` flag will also be set in `d_flags`.

```
static void
d_lru_add(struct dentry *dentry)
{
    dentry->d_flags |= DCACHE_LRU_LIST;
    list_lru_add(&dentry->d_sb->s_dentry_lru, &dentry->d_lru);
}
```

Why might the `DCACHE_REFERENCED` flag not be set? See the `retain_dentry()` for details but there are four conditions under which this could occur including the case where `DCACHE_DONTCACHE` is set or if the dentry is disconnected. **XXX – really need to read through `fast_dput()` and see why it's setting these flags. most work is done in there**

`d_lru_add()` also called by `dentry_kill()` and `dput_to_list()`. They will be described in more detail later in this section.

Step 6

When the per-superblock LRU list is being scanned to see if dentries can be retired to reclaim memory, if the dentry is still in use, the `DCACHE_REFERENCED` flag is turned off by `dentry_lru_isolate()` and it returns to the simple hashed state but it taken off the LRU list. Section 6.9.6 describes the pruning process.

Step 6

If the dentry has not been accessed in quite a while and there are no references to it, the `DCACHE_REFERENCED` will be turned off in `dentry_lru_isolate()` allowing for the dentry to be reclaimed. A call is made to `d_lru_shrink_move()` which turns on the `DCACHE_SHRINK_LIST` flag and moved to the disposable list (see `prune_dcache_sb()`). Pruning the dcache is covered in section 6.9.6. **XXX – OK so need to understand this better**

Step 7

When a file is unlinked a call is made to `d_delete()` to turn the dentry into a negative dentry. The path starting at the system call layer will be:

```
vfs_unlink() → d_delete_notify() → d_delete()
```

and from `d_delete()` onwards:

```
d_delete() → dentry_unlink_inode() → __d_clear_type_and_inode()
```

The `DCACHE_ENTRY_TYPE` and `DCACHE_FALLTHRU` flags are turned off in `d_flags` and the `d_inode` field is set to `NULL`. By turning off the `DCACHE_ENTRY_TYPE` flag, any caller trying to determine the type of the file from the dentry will not get a valid response back from `__d_entry_type()`.

Step 8

The last phase in the lifecycle of a dentry involves a call to `dentry_free()`. This can be from one of several functions namely `dentry_kill()`, `d_prune_aliases()`, `shrink_dentry_list()` or `shrink_dcache_parent()`

`dentry_free()` calls `t__d_free()` which calls `kmem_cache_free()` to free the dentry structure. There may also be a call to `__d_free_external()` for dentries that needed external space allocated for a longer filename.

6.9.4 KGDB – Monitoring dentry Transition State

One of the hardest aspects of understanding the dcache for me was the transition between different states. I followed an old Linux Journal article that gave a state diagram but the article was almost twenty years old and some functions didn't exist. Recreating that state diagram with a newer Linux kernel wasn't easy and took a lot of sessions in `gdb` in conjunction with similar sessions for pathname resolution. And this is why being able to analyze the kernel when running and setting breakpoints is imperative to understanding how the kernel works.

This example takes a simple command and shows the file's dentry after pathname resolution is complete to show that it should be on both hashed and LRU lists. This is the command that we'll use which keeps the file open:

```
$ less lorem-ipsum
```

The first step is to set a breakpoint in `do_filp_open()` so that we can locate the dentry returned for "lorem-ipsum":

```
(gdb) br do_filp_open if $_streq(pathname->name, "lorem-ipsum")
Breakpoint 26 at 0xffffffff813fe8c0: file fs/namei.c, line 3678.
```

When running the command, we hit the breakpoint and step through `do_filp_open()` until we see a return from `path_openat()`:

```
Thread 2 hit Breakpoint 26, do_filp_open (dfd=dfd@entry=-100,
 pathname=pathname@entry=0xfffff8881009a2000,
 op=op@entry=0xfffffc900014dbebc) at fs/namei.c:3678
3678      {
(gdb) n
3679          struct nameidata nd;
(gdb) n
3683          set_nameidata(&nd, dfd, pathname, NULL);
```

```
(gdb) 610 p->state = 0;(gdb) [*n
3683          set_nameidata(&nd, dfd, pathname, NULL);
(gdb) n
3684          filp = path_openat(&nd, op, flags | LOOKUP_RCU);
(gdb) n
3685          if (unlikely(filp == ERR_PTR(-ECHILD)))
```

One return from `path_openat()` we can look at the file structure and check that we have the right file:

```
(gdb) p filp->f_path
$225 = {
    mnt = 0xffff88812ecffba0,
    dentry = 0xffff8881032cd0c0
}
(gdb) p filp->f_path->dentry->d_iname
$226 = "lorem-ipsum", '\000' <repeats 20 times>
```

and from the file structure we can also access the dentry for the file and look at the flags:

```
(gdb) p/x filp->f_path->dentry->d_flags
$231 = 0x480040
```

Here are the values for the `d_flags` field:

```
#define DCACHE_REGULAR_TYPE      0x00400000 /* Regular file type */
#define DCACHE_LRU_LIST           0x00080000
#define DCACHE_REFERENCED         0x00000040 /* Recently used,
                                              don't discard. */
```

We can also check the hash list and LRU list. The dentry should be on both lists according to the flags:

```
(gdb) p filp->f_path->dentry->d_hash
$232 = {
    next = 0x0 <fixed_percpu_data>,
    pprev = 0xffff88817b975948
}
(gdb) p filp->f_path->dentry->d_lru
$233 = {
    next = 0xffff8881032cdb00,
    prev = 0xffff888103050c80
}
```

It is indeed on both lists as expected. Since it's hashed, if we access the file again, the dentry will be found quickly. Note that the hold on the dentry is dropped once pathname resolution is complete even though we still have the file open (due to running `less(1)`).

6.9.5 Digging Deeper on The Core Functions

With the state diagram in mind, the following sections dig deeper on the internal operations of the dcache.

Here are the main functions ... there is a mismatch between the above diagram and this list. I know I am using `d_splice_alias()`

Expand on the basics so I really understand them. Will help with the state diagram.

Pruning the dcache is covered in section 6.9.6.

`d_alloc()`

Callers of `d_alloc()` will get back a new dentry with this new dentry linked on the `d_subdirs` list of the parent but the dentry will not be on the hash queue at this point.

Here are the main paths through `d_alloc()`: **XXX – come back to described locking**

```
struct dentry *
d_alloc(struct dentry * parent, const struct qstr *name)
{
    struct dentry *dentry = __d_alloc(parent->d_sb, name);
    spin_lock(&parent->d_lock);
    __dget_dlock(parent);
    dentry->d_parent = parent;
    list_add(&dentry->d_child, &parent->d_subdirs);
    spin_unlock(&parent->d_lock);
    return dentry;
}
```

An empty dentry is returned by `__d_alloc()`, a lock is taken on the parent dentry while this new entry is added to the `d_subdirs` field. On return from `d_alloc()`, there is no associated inode. Much of the code in `__d_alloc()` deals with allocation and initialization of the different fields in the dentry structure:

```
static struct dentry *
__d_alloc(struct super_block *sb, const struct qstr *name)
{
    struct dentry *dentry;
    char *dname;

    dentry = kmem_cache_alloc_lru(dentry_cache,
                                  &sb->s_dentry_lru, GFP_KERNEL);
    dentry->d_iname[DNAMES_INLINE_LEN-1] = 0;
    ...
} else if (name->len > DNAMES_INLINE_LEN-1) {
    /* kmalloc() space for the name */
    *p = kmalloc(...);
    dname = p->name;
} else {
    dname = dentry->d_iname;
}

dentry->d_name.len = name->len;
dentry->d_name.hash = name->hash;
```

```

    memcpy(dname, name->name, name->len);

    dentry->d_inode = NULL;
    dentry->d_parent = dentry;
    dentry->d_sb = sb;
    INIT_HLIST_BL_NODE(&dentry->d_hash);
    INIT_LIST_HEAD(&dentry->d_lru);
    INIT_LIST_HEAD(&dentry->d_subdirs);
    INIT_HLIST_NODE(&dentry->d_u.d_alias);
    INIT_LIST_HEAD(&dentry->d_child);

    return dentry;
}

```

If the length of the filename is small enough, it's copied to `d_inode` otherwise memory is allocated and it will be copied to `d_name.name`. All lists are initialized but at this stage, the dentry is only added to `d_subdirs` when a return is made to `d_alloc()`.

Callers of `d_alloc()` are shown on the left-hand side of figure 6.14.

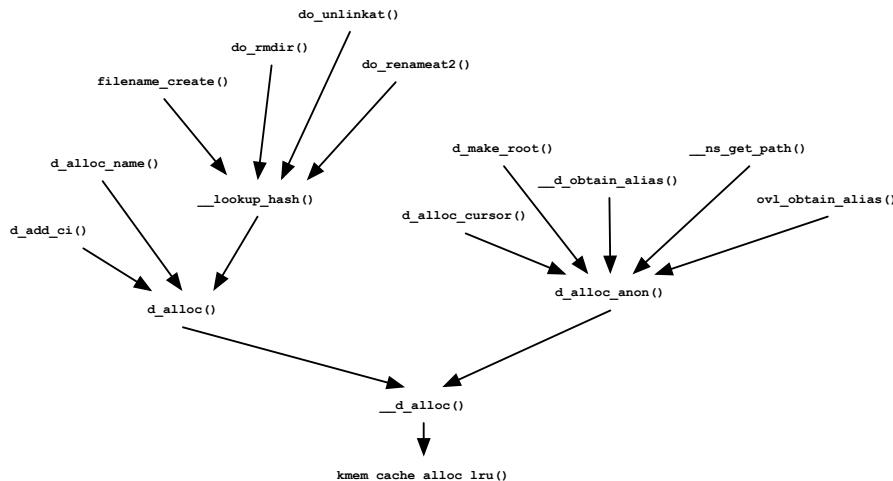


Figure 6.14: Callers of `d_alloc()`

The function `d_add_ci()` is called to lookup or allocate a new dentry with case-exact name. The only callers are the NTFS and XFS filesystems. By default, NTFS is case-insensitive (although that can be disabled during mount). XFS provides an option for filenames to be case-insensitive. All other Linux filesystems are case-sensitive so "filename" and "FILENAME" are both valid files in the same directory.

The `d_alloc_name()` function is called by procfs, configfs, SELinux and others. Sometimes it is to create a dentry for the root inode and in the case of procfs, one use is to set up `/proc/self`.

Callers of `filename_create()` are shown in figure 6.9 but include creation of symlinks, hard links and device nodes. Along with the other functions shown, it calls `__lookup_hash()` which will call the `lookup` operation

d_alloc_anon()

Shown on the right-hand side of figure 6.14, `d_alloc_anon()` simply makes a call is made to `__d_alloc()` as follows:

```
struct dentry *
d_alloc_anon(struct super_block *sb)
{
    return __d_alloc(sb, NULL);
}
```

There is no name field passed as an so `NULL` is passed to `__d_alloc()` and on return, there is no parent to associated this new dentry with (thus an anonymous dentry). Non of the initialization described above applies to anonymous dentries.

There are several callers of `d_alloc_anon()` but the most common case occurs when mounting a filesystem. When the filesystem `fill_super` function is called, it reads in the root inode and calls `d_make_root()` which in turn calls `d_alloc_anon()`.

retain_dentry() and d_put

Each mounted filesystem has an LRU list hanging off the `list` field of the `super_block` structure. This makes a lot of sense. When a filesystem is unmounted, walking the list of dentries needs to be efficient.

In the original state diagram referenced earlier in this chapter, dentries were moved from the pure *hashed* state and added to the LRU list through a call to `dput()`. Today, there are multiple callers and it's actually `retain_dentry()` which is the only function to call `d_lru_add` to add a dentry to the LRU list. The overall paths to this function are shown in figure 6.15. Note that the `DCACHE_LRU_LIST` flag is set by `d_lru_add` and on return, `DCACHE_REFERENCED` is set by `retain_dentry()` if it is not already set by another thread.

You will notice that there are two paths going through `dput()`. The first path involves calling `retain_dentry()` directly and the second passes through `dentry_kill()`. Here is the code to `dput()`:

```
void
dput(struct dentry *dentry)
{
    if (likely(fast_dput(dentry))) {
        return;
    }
    if (likely(retain_dentry(dentry))) {
        return;
    }
    dentry = dentry_kill(dentry);
```

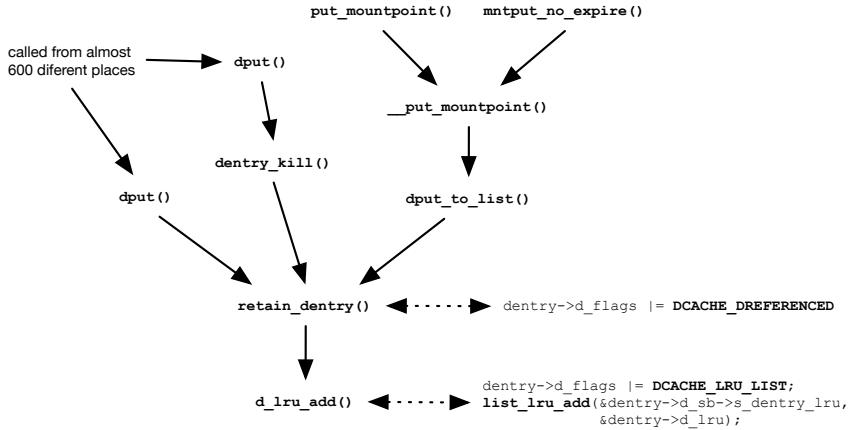


Figure 6.15: Adding a dentry to the superblock LRU list

```

        }
    }
}
```

A call is made to `fast_dput()` to perform a *lockless* `dput()`. **XXX — need to look through this code**

In almost all circumstances, the dentry will be moved to the LRU list. There are checks inside `retain_dentry()` which specify that the dentry should no longer be kept and **XXX — need to explain some of these at least**.

`d_splice_alias` and `d_instantiate()`

`d_splice_alias()` is called during lookup where it will instantiate the dentry if the file is found otherwise it will instantiate a negative dentry. As the comment at the top of the function states that this function will "*splice a disconnected dentry into the tree if one exists*". This function calls `__d_add()` which in turn calls `__d_set_inode_and_type()` to set the `d_inode` field to point to the inode and initialize `d_flags`. It also calls `__d_rehash()` which will add the dentry to the correct hash bucket. The callers of are shown in figure 6.16.

`d_instantiate()` is called by the filesystem during a creation type event such as "create", "mkdir", "symlink" and "link". In all of these cases a dentry is passed as an argument. For example, take a call into SPFS for directory creation:

```
sp_mkdir(struct user_namespace *mnt_userns, struct inode *dip,
          struct dentry *dentry, umode_t mode)
```

In all of these functions a call to the filesystem's *lookup* function will have been called which will have previously instantiated a negative dentry.

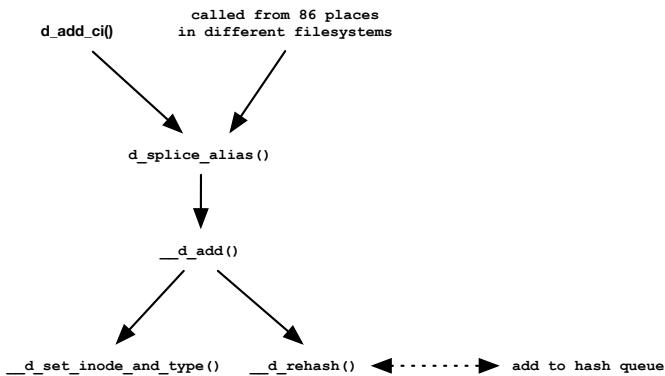


Figure 6.16: Using `d_splice_alias()` to add an inode to a negative dentry

The "i_count" member in the inode structure should be set/incremented. I don't cover this anywhere

d_add()

This function is called to get a hold on the dentry to ensure that it stays in the cache while some other operation is performed like **XXX**.

```
static inline struct dentry *
dget(struct dentry *dentry)
{
    if (dentry)
        lockref_get(&dentry->d_lockref);
    return dentry;
}
```

The `d_lockref` uses the lockref primitive to provide both a spinlock and a reference count. Inside `lockref_get()`, the spinlock is taken and the reference count is incremented:

```
spin_lock(&lockref->lock);
lockref->count++;
spin_unlock(&lockref->lock);
```

xxx

d delete

TBD

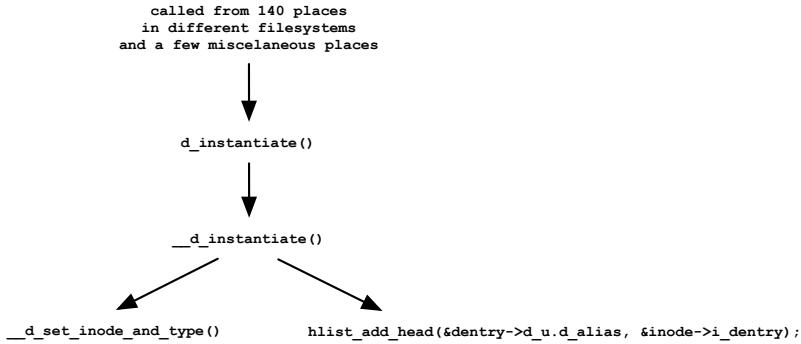


Figure 6.17: Callers of `d_instantiate()` which are passed a negative dentry

d_delete

delete a dentry. If there are no other open references to the dentry then the dentry is turned into a negative dentry (the `d_put()` method is called). If there are other references, then `d_drop()` is called instead

dget

open a new handle for an existing dentry (this just increments the usage count)

dput

close a handle for a dentry (decrements the usage count). If the usage count drops to 0, and the dentry is still in its parent's hash, the "d_delete" method is called to check whether it should be cached. If it should not be cached, or if the dentry is not hashed, it is deleted. Otherwise cached dentries are put into an LRU list to be reclaimed on memory shortage.

d_drop

this unhashes a dentry from its parents hash list. A subsequent call to `dput()` will deallocate the dentry if its usage count drops to 0

d_add

add a dentry to its parents hash list and then calls `d_instantiate()`

6.9.6 Pruning The Dcache

To help with putting the state diagram together, a key piece of the puzzle is to look at when `DCACHE_REFERENCED` was turned on and off in `d_flags`.

As described earlier in this section and shown in figure 6.15 the only place where the DCACHE_REFERENCED is set is inside `retain_dentry()` which is generally called through the `dput()` path.

Similarly, the flag is only turned off in `dentry_lru_isolate()` and this function is only called by `prune_dcache_sb()` which is shown in figure 6.18. in turn is only called by `super_cache_scan()` which will be discussed in the next section.

At some point, the dentry will have the DCACHE_REFERENCED flag removed and be a candidate for deletion. Refer to figure 6.18. The `super_cache_scan()` function is assigned to the `s_shrink.scan_objects` field of the `super_block` during a call from `alloc_sb()`. This occurs when a filesystem is being mounted. This function will run **XXX - when does it run?**

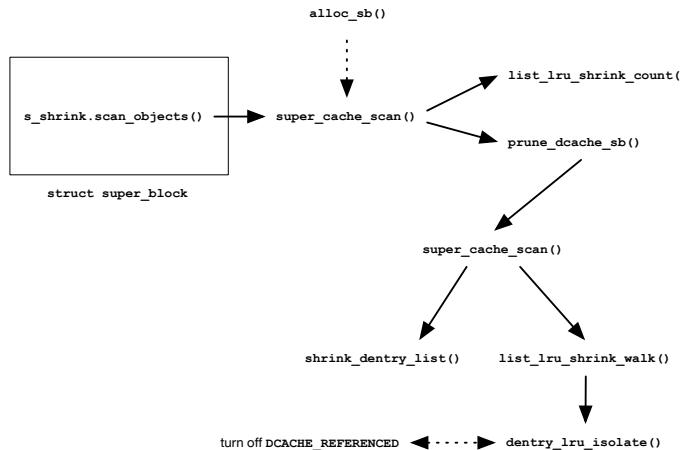


Figure 6.18: Pruning dcache entries for a specific filesystem

Inside `super_cache_scan()`, a call is made to `prune_dcache_sb()` which processes the LRU list eventually calling `dentry_lru_isolate()`. This turns off the DCACHE_REFERENCED flag and **XXX** allows for the dentry to be deleted.

6.9.7 KGDB – Analyzing Dentries For a Simple File Hierarchy

This example will show how dentries are connected together for a filesystem with a small number of files and directories which is mounted on `/mnt`. We have the following files and since we're accessing them all, they will have been brought into the dcache.

```
$ find /mnt
/mnt
/mnt/lost+found
/mnt/mydir
/mnt/mydir/lorem-ipsum
/mnt/mydir/hello
```

Let's get the root dentry for our mounted filesystem. I know it was the last filesystem mounted so I just use the `prev` pointer when displaying the `super_blocks` list. This address is assigned to the convenience variable `$sb` and the `s_root` field is displayed. This points to the root dentry for this filesystem. The address is saved in the convenience variable `$rd`.

```
(gdb) p super_blocks
$136 = {
    next = 0xfffff88810005a800,
    prev = 0xfffff88810cc90800
}
(gdb) set $sb = (struct super_block *)0xfffff88810cc90800
(gdb) pipe p *$sb | grep s_root
    s_root = 0xfffff888101cc7540,
(gdb) set $rd = (struct dentry *)0xfffff888101cc7540
```

As a sanity check we look to see that its parent is the same address (the root of a filesystem) and that the name is "/".

```
(gdb) p $rd->d_parent
$142 = (struct dentry *) 0xfffff888101cc7540
(gdb) p *$rd->d_iname
$141 = 47 '/'
```

Now let's look at the `d_flags` field for this root dentry:

```
(gdb) p/x $rd->d_flags
$144 = 0x200000
```

Here is the type (`dcache.h`). This is the only flag set.

```
DCACHE_DIRECTORY_TYPE      0x00200000 /* Normal directory */
```

The next step is to look at the `d_subdirs` field which points to a doubly linked list of children that are in the dcache:

```
(gdb) p $rd->d_subdirs
$145 = {
    next = 0xfffff888102eb3e10,
    prev = 0xfffff888102eb1a50
}
```

This list goes through the `d_child` field of the dentry structure so we need to call `container_of()` to get the correct address of each structure:

```
(gdb) set $c1 = $container_of(0xfffff888102eb3e10, \
                           "struct dentry", "d_child")
(gdb) set $c2 = $container_of(0xfffff888102eb1a50, \
                           "struct dentry", "d_child")
```

From here, we can check the the filenames:

```
(gdb) p $c1->d_iname
$152 = "lost+found", '\000' <repeats 21 times>
```

```
(gdb) p $c2->d_iname
$153 = "mydir", '\000' <repeats 26 times>
```

and then the `d_flags` field of each dentry:

```
(gdb) p/x $c1->d_flags
$155 = 0x280040
(gdb) p/x $c2->d_flags
$156 = 0x280040
```

Breaking down `d_flags` (`dcache.h`) shows that the following flags are set:

```
DCACHE_LRU_LIST      0x00080000 /* dentry is on the LRU list*/
DCACHE_REFERENCED    0x00000040 /* Recently used,
                                 don't discard. */
DCACHE_DIRECTORY_TYPE 0x00200000 /* Normal directory */
```

Now let's look at the child files:

```
(gdb) p $c2->d_subdirs
$183 = {
    next = 0xfffff888103050bd0,
    prev = 0xfffff8881030502d0
}
(gdb) set $f1 = $container_of(0xfffff888103050bd0, \
                           "struct dentry", "d_child")
(gdb) set $f2 = $container_of(0xfffff8881030502d0, \
                           "struct dentry", "d_child")

(gdb) p $f1->d_iname
$160 = "hello", '\000' <repeats 26 times>
(gdb) p $f2->d_iname
$162 = "lorem-ipsum", '\000' <repeats 20 times>
```

and the contents of `d_flags` for each dentry:

```
(gdb) p/x $f1->d_flags
$164 = 0x480040
(gdb) p/x $f2->d_flags
$163 = 0x480040
```

Breaking down `d_flags` (`dcache.h`) shows that the following flags are set:

```
DCACHE_LRU_LIST      0x00080000
DCACHE_REFERENCED    0x00000040 /* Recently used,
                                 don't discard. */
DCACHE_REGULAR_TYPE  0x00400000 /* Regular file type */
```

The `mydir` directory and both of the files `lorem-ipsum` and `hello` have both the `DCACHE_LRU_LIST` and `DCACHE_REFERENCED` flags set indicating that they are on the LRU list although there are no holds on the dentries at this point.

6.9.8 KGDB – Inspecting the Per-Filesystem LRU List

could be combined with above. need to rerun the example. i lost the fs somehow.

XXX - small FS - look at everything on there - need to come back to this. harder than I thought. Can't find the right stuff

```
$ find /mnt
/mnt
/mnt/lost+found
/mnt/lorem-ipsum
/mnt/mydir
/mnt/mydir/myfile

XXX

(gdb) p super_blocks
$16 = {
    next = 0xfffff88810005a800,
    prev = 0xfffff8881216de800
}
(gdb) set $sb = (struct super_block *)0xfffff8881216de800

XXX

struct list_lru_node {
    spinlock_t      lock;
    struct list_lru_one lru;
    long           nr_items;
};

struct list_lru {
    struct list_lru_node *node;
#ifdef CONFIG_MEMCG_KMEM
    struct list_head   list;
    int               shrinker_id;
    bool              memcg_aware;
    struct xarray     xa;
#endif
};
```

XXX

6.9.9 Dcache Statistics

The dcache exports information through the following /proc interface:

```
$ cat /proc/sys/fs/dentry-state
159548 137775 45      0      48194    0
```

These entries correspond to the following structure defined in fs/dcache.c:

```
struct dentry_stat_t {
    long nr_dentry;
```

```
long nr_unused;
long age_limit;      /* age in seconds */
long want_pages;    /* pages requested by system */
long nr_negative;   /* # of unused negative dentries */
long dummy;          /* Reserved for future use */
};
```

The routines for providing this information can also be found in the same file. If you look throughout the various dcache functions you will see calls to `this_cpu_inc()` and `this_cpu_dec()` which increment and decrement the number of entries used.

Come back here near the end to see if newer OS versions have picked up any changes

6.10 The Inode Cache Implementation

TBD

XXX – Need to cover the list fields in the kern-structs chapter

Look at

6.10.1 Inode Locks

XXX – Not sure that this is the correct place. Perhaps later in VFS?

- `i_rwsem` – a read/write semaphore that serializes changes to a directory. It's also involved in pathname resolution re: looking at the dcache **XXX – need to come back and do this once I have a better handle on things**
- `i_lock` – TBD
- `i_mutex` – TBD

TBD

6.11 The Buffer Cache Implementation

TBD

6.12 File Creation

This section covers the creation of regular files, directories, symlinks, hard links and device nodes and shows the reliance on pathname resolution to get to all but the last component which needs special handling based on the type of file that is being created. It is recommended to first refer to section 6.7 to understand the basics of pathname resolution before reading this section.

6.12.1 Regular File Creation

The `open(2)` manpage describes the five different system calls that can result in creation of a regular file. These five system calls are:

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

int creat(const char *pathname, mode_t mode);

int openat(int dirfd, const char *pathname, int flags);
int openat(int dirfd, const char *pathname, int flags,
           mode_t mode);

int openat2(int dirfd, const char *pathname,
            const struct open_how *how, size_t size);
```

Section 6.7 described the flow through the kernel for opening a file. We'll show how these system calls converge when a file is being created.

Here are the system call entry points in the kernel for all functions:

```
open.c:SYSCALL_DEFINE2(creat, const char __user *,
fhandle.c:SYSCALL_DEFINE3(open_by_handle_at, int,
fhandle.c:COMPAT_SYSCALL_DEFINE3(open_by_handle_at,
open.c:SYSCALL_DEFINE3(open, const char __user *,
open.c:SYSCALL_DEFINE4(openat, int, dfd, const char __user *,
open.c:SYSCALL_DEFINE4(openat2, int, dfd,
open.c:COMPAT_SYSCALL_DEFINE3(open, const char __user *,
open.c:COMPAT_SYSCALL_DEFINE4(openat, int, dfd,
```

There is a lot of code to walkthrough before a call is made into the filesystem to create a file, mostly to do with pathname resolution. Wandering around the top-level system call handling code for the above routines will show a lot of functions called. Sometimes it's very helpful to work backwards by setting a breakpoint on a filesystem's *create* function and seeing the path that the kernel gets there. Using the SPFS filesystem (described in chapter 7), a breakpoint will be set on `sp_create()` as follows:

```
(gdb) br sp_create
Breakpoint 28 at 0xfffffffffa0656f7f:
        file /home/spate/spfs/kern/sp_dir.c, line 279.
```

and then it's simply a matter of copying a file into an SPFS filesystem:

```
$ cp lorem-ipsum /mnt
```

When the breakpoint is hit, here is the stacktrace:

```
#0  sp_create (mnt_userns=0xffffffff82a81240 <init_user_ns>,
    dip=0xffff88812ff34500, dentry=0xffff8881031dad80,
    mode=33188, excl=true) at /home/spate/spfs/kern/sp_dir.c:279
#1  lookup_open (op=0xfffffc90001177e44, got_write=true,
    file=0xffff8881333c2500, nd=0xfffffc90001177d20)
    at fs/namei.c:3376
```

```
#2 open_last_lookup (op=0xfffffc90001177e44,
file=0xfffff8881333c2500, nd=0xfffffc90001177d20)
at fs/namei.c:3446
#3 path_openat (nd=nd@entry=0xfffffc90001177d20,
op=op@entry=0xfffffc90001177e44, flags=flags@entry=64)
at fs/namei.c:3654
#4 do_filp_open (dfd=dfd@entry=-100,
pathname.pathname@entry=0xfffff88810087d000,
op=op@entry=0xfffffc90001177e44) at fs/namei.c:3684
#5 do_sys_openat2 (dfd=-100, filename=<optimized out>,
how=how@entry=0xfffffc90001177e88) at fs/open.c:1275
#6 do_sys_open (mode=<optimized out>,
flags=<optimized out>, filename=<optimized out>,
dfd=<optimized out>) at fs/open.c:1291
```

Inside `path_open_at()`, there is a call to `open_last_lookup()` which in turn calls `lookup_open()` which checks to see if there is a negative dentry for this file (highlighted in bold) and if this is the case, makes a call through to the filesystem's `create` function to actually create the file. Note that `dir_inode` is the parent directory.

```
static struct dentry *
lookup_open(struct nameidata *nd, struct file *file,
            const struct open_flags *op,
            bool got_write)
{
    struct dentry *dir = nd->path.dentry;
    struct inode *dir_inode = dir->d_inode;

    if (!dentry->d_inode && (open_flag & O_CREAT)) {
        file->f_mode |= FMODE_CREATED;
        if (!dir_inode->i_op->create) {
            error = -EACCES;
            goto out_dput;
        }
        error = dir_inode->i_op->create(idmap, dir_inode,
                                         dentry, mode,
                                         open_flag & O_EXCL);
    }
}
```

The filesystem will create an inode on disk (local filesystem) and make a call to instantiate the dentry as follows:

```
d_instantiate(dentry, inode);
```

For more information on the steps taken by SPFS to create a file, see section 7.9.

Having gone backwards to get to this point, it's time to go back to the system call handling. For `open(2)`, `creat(2)` and `open_at()` there is little work done other than call through to `do_sys_open()`. The difference between the three functions is that the directory where the file is located is specified by the caller giving the following:

```
SYSCALL_DEFINE4(openat, int, dfd, const char __user *, filename,
```

```
        int, flags, umode_t, mode)
{
    return do_sys_open(dfd, filename, flags, mode);
}
```

and for open/create:

```
return do_sys_open(AT_FDCWD, pathname, flags, mode);
```

The AT_FDCWD informs do_sys_open() to start from the current working directory. The value of AT_FDCWD is -100 so it can be distinguished from a valid file descriptor (which is non-negative). There is little work to do in do_sys_open() other than call build_open_low() and then do_sys_openat2() which is the entry point for the openat2(2) system call. The early parts of this function is described in section 6.7.

Good time to pause. Really need to better understand the whole phase of link_path_walk() avoiding the last component and how it's different with create vs plain open

6.12.2 Creating Directories

There are two system calls for directory creation namely mkdir(2) and mkdirat(2). The system call handlers for both are as follows:

```
$ grep SYSCALL *.c | grep mkdir
namei.c:SYSCALL_DEFINE3(mkdirat, int, dfd, ...
namei.c:SYSCALL_DEFINE2(mkdir, const char __user *, ...
```

Both of these functions do little other than making a call to do_mkdirat() either passing AT_FDCWD in the case of mkdir(2) or for mkdirat(2) it will be the directory file descriptor passed as the first argument.



Directory creation handling code can be found in fs/namei.c.

Most of the work in directory creation involves resolving the pathname passed. A call is made to filename_create() which should return a negative dentry for the new directory to be created. The creation is then performed by a call to vfs_mkdir().

```
int
do_mkdirat(int dfd, struct filename *name, umode_t mode)
{
    struct dentry *dentry;
    struct path path;
    unsigned int lookup_flags = LOOKUP_DIRECTORY;

    dentry = filename_create(dfd, name, &path, lookup_flags);
    error = security_path_mkdir(&path, dentry,
        if (!error) {
            error = vfs_mkdir(mnt_idmap(path.mnt),
```

```

        path.dentry->d_inode,
        dentry, mode);
    }
    done_path_create(&path, dentry);
}

```

Inside `vfs_mkdir`, a call is made to `vfs_prepare_mode()` to set the directory's *mode* and then the filesystem *mkdir* function is called passing the parent's inode and the negative dentry.

```

int
vfs_mkdir(struct mnt_idmap *idmap, struct inode *dir,
          struct dentry *dentry, umode_t mode)
{
    unsigned max_links = dir->i_sb->s_max_links;

    may_create(idmap, dir, dentry);
    mode = vfs_prepare_mode(idmap, dir, mode,
                           S_IRWXUGO | S_ISVTX, 0);
    security_inode_mkdir(dir, dentry, mode);

    if (max_links && dir->i_nlink >= max_links)
        return -EMLINK;

    dir->i_op->mkdir(idmap, dir, dentry, mode);
}

```

Inside the filesystem, assuming the operation is successful a call to `d_instantiate()` to instantiate the dentry with the new inode that the filesystem creates. To see how SPFS handles the *mkdir* call refer to section 7.10.

Returning to `filename_creation()`. This routine that is called by several functions when new files are created, whether directories, symbolic links or hard links. If all goes well, a negative dentry will be returned by this function resulting in a call to the filesystem to perform the appropriate create function. Much of what happens within `filename_creation()` is handling pathname resolution as the following fragment of the function shows:

```

static struct dentry *
filename_create(int dfd, struct filename *name,
               struct path *path, unsigned
               int lookup_flags)
{
    ...
    error = filename_parentat(dfd, name, reval_flag, path,
                               &last, &type);
    if (last.name[last.len] && !want_dir)
        create_flags = 0;
    dentry = __lookup_hash(&last, path->dentry,
                          reval_flag | create_flags);
    if (d_is_positive(dentry))

```

```
        goto fail;

        return dentry;

fail:
    dput(dentry);
    dentry = ERR_PTR(error);
    ...
    return dentry;
}
```

There will be a call sequence as follows:

```
filename_parentat() → path_parentat() → link_path_walk()
```

This will resolve all but the last component for which a call to `__lookup_hash()` is made to see if the file being created is in the dcache. We should only find a negative dentry if a new file is to be created and thus the check for `d_is_positive()`.

6.12.3 Creating Symbolic Links

There are two system calls for creation of symbolic links namely `symlink(2)` and `symlinkat(2)`. The system call handlers for both can be found in the `fs` directory as follows:

```
$ grep SYSCALL *.c | grep symlink
namei.c:SYSCALL_DEFINE3(symlinkat, const char __user *, ...
namei.c:SYSCALL_DEFINE2(symlink, const char __user *, ...
```

For `symlink()` the code is very simple:

```
return do_symlinkat(getname(oldname), AT_FDCWD,
                     getname(newname));
```

Inside `do_symlinkat()`, the path followed is very similar to that of `do_mkdirat()` as described in section 6.12.2:

```
int
do_symlinkat(struct filename *from, int newdfd,
              struct filename *to)
{
    unsigned int lookup_flags = 0;
    ...
    dentry = filename_create(newdfd, to, &path, lookup_flags);
    error = security_path_symlink(&path, dentry, from->name);
    if (!error)
        error = vfs_symlink(mnt_idmap(path.mnt),
                            path.dentry->d_inode,
                            dentry, from->name);
    done_path_create(&path, dentry);
    ...
}
```

Other than check permissions, `vfs_symlink()` calls in to the filesystem to create the symlink passing the parent inode and a negative dentry for the symlink file:

```
error = dir->i_op->symlink(idmap, dir, dentry, oldname);
```

Section 7.19 describes the filesystem handling side of this operation.

6.12.4 Creating Hard Links and Device Nodes

See the previous two sections for directory and symbolic link creation. For hard links, the process followed is nearly identical with the following call sequence:

```
link() → do_linkat() → filename_lookup()
→ vfs_link() → dir->i_op->link()
```

Where to start depends on whether `link(2)` or `linkat(2)` is called.

For hard links, the process is the same as follows:

```
mknod() → do_mknodat() → filename_lookup()
→ vfs_mknod() → dir->i_op->mknod()
```

For all functions, see `fs/namei.c` for details.

6.13 Reading Directory Entries

The first time I came across the term *cookies* was over thirty years ago and in conjunction with how the kernel handled reading directory entries. User programs don't necessarily know how many entries there are so they allocate a fixed size buffer and make repeated calls into the kernel. The kernel kept track of where the application was so that it could start from the place it stopped on the previous call.

Like pathname resolution, reading directory entries can be a little messy to follow. **is it-come back to be sure.**

Section 2.23 described the user-level interfaces for reading directory entries and mentioned the `getdents(2)` system call. In the manpage it instructs users not to use the system call interfaces `dirent` but to look at the `readdir(3)` manpage for the POSIX-conforming C library interface.



Directory handling code can be found in `fs/readdir.c`

There are three different entry points as follows:

```
SYSCALL_DEFINE3(getdents, ...
SYSCALL_DEFINE3(getdents64, ...
COMPAT_SYSCALL_DEFINE3(getdents, ...
```

All functions basically do the same thing so we'll pick `getdents64()` and follow things from there. Here is a skeleton version of this function:

```
SYSCALL_DEFINE3(getdents64, unsigned int, fd,
               struct linux_dirent64 __user *, dirent,
               unsigned int, count)
{
    struct fd f;
    struct getdents_callback64 buf = {
        .ctx.actor = filldir64,
        .count = count,
        .current_dir = dirent
    };
    int error;

    f = fdget_pos(fd);

    error = iterate_dir(f.file, &buf.ctx);
    if (buf.prev_reclen) {
        struct linux_dirent64 __user * lastdirent;
        typeof(lastdirent->d_off) d_off = buf.ctx.pos;

        lastdirent = (void __user *) buf.current_dir
                     - buf.prev_reclen;
        if (put_user(d_off, &lastdirent->d_off))
            error = -EFAULT;
        else
            error = count - buf.count;
    }
    fdput_pos(f);
    return error;
}
```

The first step is to locate the file descriptor structure that is associated with the argument `fd` that is passed through the system call interface. This structure allows us to get to the `file` structure but also has several flags **that are useful or not? I don't set the `f_op` field anywhere. No filesystem sets it???? Nothing in fs/*.c sets it either**

```
struct fd {
    struct file *file;
    unsigned int flags;
};
```

From here a call is made to `iterate_dir()` to call into the filesystem to read some of the directory entries:

```
int iterate_dir(struct file *file, struct dir_context *ctx)
{
    struct inode *inode = file_inode(file);
    bool shared = false;
```

```

        if (file->f_op->iterate_shared)
            shared = true;
        else if (!file->f_op->iterate)
            goto out;

        if (!IS_DEaddir(inode)) {
            ctx->pos = file->f_pos;
            if (shared)
                res = file->f_op->iterate_shared(file, ctx);
            else
                res = file->f_op->iterate(file, ctx);
            file->f_pos = ctx->pos;
            fsnotify_access(file);
            file_accessed(file);d
        }
    }
}

```

Which operation in the filesystem should be called depends on what the filesystem has set.
XXX - this is a little weird. I actually set f_op in the inode to the list of file_operations in which I set iterate_shared = sp_readdir so who sets f_op?

To see the filesystem-side of reading directories refer to section 7.11.

6.14 The Linux Page Cache

Not sure I can just cover read/write/mmap without better describing the page cache. Do the section in kern-structs first though - means moving the stuff below once there is a good dividing line

6.14.1 Compound Pages

TBD

I/O occurs in page-sized chunks (thus the name *page cache*) and the page cache only deals with single pages, a fixed size chosen based on the CPU on which Linux is running. A page has generally been 4096 bytes since the first launch of Linux on the Intel x86 architecture. But there are times when it's more efficient to use a larger unit of allocation and that's where *compound pages* come into the picture.

Generally speaking, most development in the kernel will not use compound pages -
XXX - need to figure out how this affects filesystems. I don't think it does as there is a folio interface

And that is pretty much everything that distinguishes a compound page from an ordinary, higher-order allocation. Most developers will not encounter compound pages in their area of the kernel. In cases where it is truly necessary to treat a set of pages as a single unit, though, compound pages may well be part of the solution toolkit.

most of this is from lwn so rewrite, shuffle and change big time -

- page is simply a grouping of two or more physically contiguous pages into a unit that can, in many ways, be treated as a single, larger page.

- most commonly used to create huge pages, used within hugetlfs or the transparent huge pages subsystem
- can serve as anonymous memory or be used as buffers within the kernel
 - they cannot, however, appear in the page cache, which is only prepared to deal with singleton pages
- Allocating a compound page is a matter of calling a normal memory allocation function like alloc_pages() with the __GFP_COMP allocation flag set and an order of at least one.
- can't create an order-zero (single-page) compound page
- pages = alloc_pages(GFP_KERNEL, 2); /* no __GFP_COMP */ will not return a compound page creating a compound page involves the creation of a fair amount of metadata; much of the time, that metadata is unneeded so the expense of creating it can be avoided - odd - explain???
-

compound page meta-data

So what does that metadata look like? Since most of it is stored in the associated page structures, one can assume that it's complicated. Let's start with the page flags. The first (normal) page in a compound page is called the "head page"; it has the PG_head flag set. All other pages are "tail pages"; they are marked with PG_tail. At least, that is the case on systems where page flags are not in short supply — 64-bit systems, in other words. On 32-bit systems, there are no page flags to spare, so a different scheme is used; all pages in a compound page have the PG_compound flag set, and the tail pages have PG_reclaim set as well. The PG_reclaim bit is normally used by the page cache code, but, since compound pages cannot be represented in the page cache, that flag can be reused here.

Code dealing with compound pages need not worry about the different marking conventions, though. No matter which convention is in use, a call to Page-Compound() will return a true value if the passed-in page is a compound page. Head and tail pages can be distinguished, should the need arise, with Page-Head() and PageTail(). - huh?

Every tail page has a pointer to the head page stored in the first_page field of struct page. This field occupies the same storage as the private field, the spinlock used when the page holds page table entries, or the slab_cache pointer used when the page is owned by a slab allocator. The compound_head() helper function can be used to find the head page associated with any tail page.

6.14.2 Memory Folios

TBD

- - inefficient to uses small PAGE_SIZE chunks
- - compound pages help
- - Even functions which are aware of compound pages may expect a head page, and do the wrong thing if passed a tail page.
- - folios make it easier
- - lots of kernel code handle pages so compound pages blah blah.
- - Folios are supposed to help
- - "An earlier version of this patch set found it was worth about a 7% reduction of wall-clock time on kernel compiles" - that amounts to about 5 minutes for my apple mac / VM on UTM. Finding other performance numbers isn't easy. It's been speculated that the performance gain will be somewhere between 0-10% and assumed that there is no degradation for any test.

There are several functions that filesystems can provide that handle folios. All are contained in the `inode_operations` structure and are shown below:

```
struct address_space_operations {
    ...
    int (*read_folio)(struct file *, struct folio *);
    bool (*dirty_folio)(struct address_space *, struct folio *);
    void (*invalidate_folio)(struct folio *, size_t offset, size_t len);
    bool (*release_folio)(struct folio *, gfp_t);
    void (*free_folio)(struct folio *folio);
    int (*launder_folio)(struct folio *);
    bool (*is_partially_uptodate)(struct folio *, size_t from,
                                 size_t count);
    void (*is_dirty_writeback)(struct folio *, bool *dirty, bool *wb);
    ...
}
```

xxxx

A simple filesystem doesn't need to handle folios directly. For SPFS, the disk-based filesystem that will described in section 7, here are the functions defined. As you can see, the first two are set to generic kernel functions.

```
sp_file.c: .dirty_          = block_dirty_folio,
sp_file.c: .invalidate_folio = block_invalidate_folio,
sp_file.c: .read_folio      = sp_read_folio,
```

The `sp_read_folio()` function does little than make a call into the generic kernel function `block_read_full_folio()`.

6.14.3 Memory Mapped Files

TBD



Kernel entry points for `mmap(2)` can be found in `mm/mmap.c`

The `mmap(2)` system call enters the kernel as follows (`mm/mmap.c`):

```
mac: grep SYSCALL */*.c | grep mmap
mm/mmap.c:SYSCALL_DEFINE1(brk, unsigned long, brk)
mm/mmap.c:SYSCALL_DEFINE6(mmap_pgoff, unsigned long, addr,
mm/mmap.c:SYSCALL_DEFINE1(old_mmap, struct mmap_arg_struct
mm/mmap.c:SYSCALL_DEFINE2(unmap, unsigned long, addr,
mm/mmap.c:SYSCALL_DEFINE5(remap_file_pages, unsigned long
mm/nommu.c:SYSCALL_DEFINE6(mmap_pgoff, unsigned long, addr
mm/nommu.c:SYSCALL_DEFINE1(old_mmap, ...
```

It's gotta be in there somewhere?!

6.15 Reading Files

Below may not flow – cut n pasted from redundant section above

I have to say that trying to understand the file I/O code as it relates to the page cache is an arduous task. Without the use of `gdb` and being able to walk through the code, it would have taken much longer to understand. Documentation in this area is somewhat sparse.
XXX – need to give references as appropriate.

Reading regular files involves a lot of interaction with the page cache and the underlying filesystem in which the file resides. Filesystems can also have greater control about how files are read or they can have the kernel do most of the work through generic functions. This chapter will start with the general path and subsequent sections will explore the alternative paths. Figure 6.19 shows the a process interacts with the kernel utilizing both the page cache and the filesystem.

If the process enters the kernel through calling one of the read-related system calls (path p1a), the kernel will attempt to locate pages of cached data. For each page of data, if the page is resident (p1c) the kernel can simply copy the data from the cached page into the user-supplied buffer. If the page is not present (p1b), a call is made into the filesystem to read the data from disk into the page cache. From there, the data is copied to user-space (p1c) and the kernel returns with the amount of data read.

If the file had previously mapped the file and then accesses a location in memory that is covered by the mapping, it will get a page fault and enter the kernel (path p2a). The kernel validates the address and make a call into the filesystem (p2b) whereby data will be read from disk to fill the page. Once that is complete, the kernel returns to the process (p3) and the read is successfully made on the second attempt.

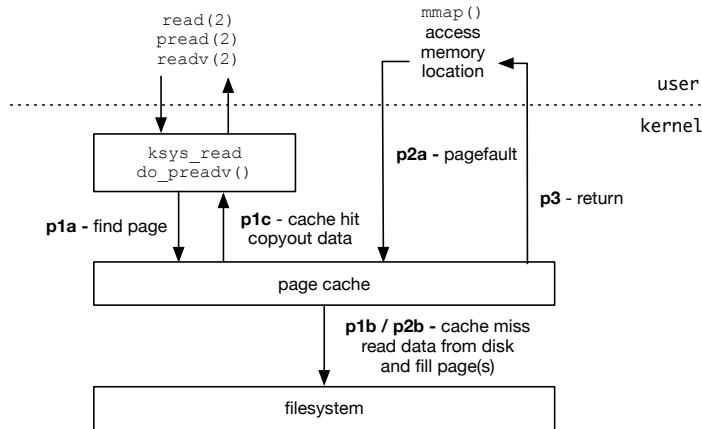


Figure 6.19: The different ways that data is read from a file

Note all reads and writes go through the page cache unless a file is opened with `O_DIRECT` which will be covered in section **XXX – TBD**.



The kernel read paths starts in `fs/read_write.c` and `XXX`

First of all, the `read(2)` system call enters the kernel as follows:

```
SYSCALL_DEFINE3(read, unsigned int, fd,
                char __user *, buf, size_t, count)
{
    return ksys_read(fd, buf, count);
}
```

Tasks performed by `ksys_read()` are shown below and include using the file descriptor to locate the appropriate file structure from which the current file offset can be found with a call to `file_ppos()`. The offset is stored in `file->f_pos`.

```
ssize_t
ksys_read(unsigned int fd, char __user *buf, size_t count)
{
    struct fd f = fdget_pos(fd);
    ssize_t ret = -EBADF;

    if (f.file) {
        loff_t pos, *ppos = file_ppos(f.file);
        if (ppos) {
            pos = *ppos;
```

```
        ppos = &pos;
    }
    ret = vfs_read(f.file, buf, count, ppos);
    if (ret >= 0 && ppos)
        f.file->f_pos = pos;
    fdput_pos(f);
}
return ret;
}
```

Once we have the file offset, a call is made to `vfs_read()` which first checks to see that the number of bytes to read and the file offset (and combination thereof) are valid values with a call to `rw_verify_area()`. Next, the kernel checks to see which read function the underlying filesystem has exported and makes a call to it. Some filesystems provide their own `read()` function and others utilize the generic `generic_file_read_iter()` function. **XXX – need to say why at some point - it's all a bit of a mess. There is a .read function and a .read_iter function. No FS seems to set .read other than for directories. Take BFS as an example, .read is set to generic_read_dir() and /read_iter is set to generic_file_read_iter().**

```
ssize_t
vfs_read(struct file *file, char __user *buf, size_t count,
          loff_t *pos)
{
    ssize_t ret;

    ret = rw_verify_area(READ, file, pos, count);

    if (file->f_op->read)
        ret = file->f_op->read(file, buf, count, pos);
    else if (file->f_op->read_iter)
        ret = new_sync_read(file, buf, count, pos);
    else
        ret = -EINVAL;
    return ret;
}
```

X

NOTE - `read_iter` may have something to do with async i/o - *possibly asynchronous read with iov_iter as destination* - see <https://lwn.net/Articles/535034/> for more information

SPFS doesn't set a "read" operation but it does set "generic_file_read_iter" as the "read_iter" function.

6.15.1 The `new_sync_read()` and `generic_file_read_iter()` Functions

X

```

vfs_read() → new_sync_read() → call_read_iter()
          → generic_file_read_iter() → filemap_read()

```

In `filemap_read()` data will be copied from the page cache into buffers in user-space. If the data is not currently present, a combination(?) of the readahead and `read_folio` address_space operations are used to fetch it. Here is the code ...

```

filemap_read()
- loop around each iocb (one for regular read)
  - error = filemap_get_pages(iocb, iter->count, &fbatch, iov_iter_is_pipe(
  - copied = copy_folio_to_iter(folio, offset, bytes, iter); calls followin

copy_page_to_iter():
- void *kaddr = kmap_local_page(page);
- size_t n = min(bytes, (size_t)PAGE_SIZE - offset);
- n = _copy_to_iter(kaddr + offset, n, i);

```

Hmm! I can't see where the FS gets calls. Perhaps it just faults on the mapping above?

6.15.2 Vectored Reads

There are several entry points into the kernel for vectored reads. In the `fs` directory:

```

$ grep SYSCALL *.c | grep readv
read_write.c:SYSCALL_DEFINE3(readv, unsigned long, fd, ...
read_write.c:SYSCALL_DEFINE5(preadv, unsigned long, fd, ...
read_write.c:SYSCALL_DEFINE6(preadv2, unsigned long, fd, ...
read_write.c:COMPAT_SYSCALL_DEFINE4(preadv64, ...
read_write.c:COMPAT_SYSCALL_DEFINE5(preadv, ...
read_write.c:COMPAT_SYSCALL_DEFINE5(preadv64v2, ...
read_write.c:COMPAT_SYSCALL_DEFINE6(preadv2, ...

```

XXX - Most functions end up calling `do_preadv()` which simply gets the file structure associated with the file descriptor and then makes a call to `vfs_readv()`:

```

static ssize_t
do_preadv(unsigned long fd, const struct iovec __user *vec,
          unsigned long vlen, loff_t pos, rwf_t flags)
{
    struct fd f;

    f = fdget(fd);
    if (f.file) {
        ret = -ESPIPE;
        if (f.file->f_mode & FMODE_PREAD)
            ret = vfs_readv(f.file, vec, vlen, &pos, flags);
        fdput(f);
    }
    return ret;
}

```

XX

6.16 Writing Files

How data is written to disk depends on the flags that were passed to the `open(2)` system call. The first part of this section will explore the general case where no flags are specified. This type of writing is called *buffered I/O*. The basic set of operations to be performed in this case follow the general path as follows:

I like the idea but need to really understand this before deciding whether or not to keep it.

By default, kernel marks written pages dirty and flushes after a delay: write(fd, "hello world", 11); 1. Kernel writes "hello world" to page for cached disk 2. Kernel adds the page to the "dirty list"—pages that have been modified but not yet written to disk 3. Periodically, kernel writes all pages in dirty list to disk

The `write(2)` system call enters the kernel as follows (`fs/read_write.c`):

```
SYSCALL_DEFINE3(write, unsigned int, fd,
               const char __user *, buf, size_t, count)
{
    return ksys_write(fd, buf, count);
}

ssize_t
ksys_write(unsigned int fd, const char __user *buf,
           size_t count)
{
    struct fd f = fdget_pos(fd);
    ssize_t ret = -EBADF;

    if (f.file) {
        loff_t pos, *ppos = file_ppos(f.file);
        if (ppos) {
            pos = *ppos;
            ppos = &pos;
        }
        ret = vfs_write(f.file, buf, count, ppos);
        if (ret >= 0 && ppos)
            f.file->f_pos = pos;
        fdput_pos(f);
    }

    return ret;
}
```

X

6.16.1 Vectored Writes

TBD

6.17 Direct I/O

TBD

6.18 File Locking Implementation

TBD - advisory and mandatory locking

6.19 Dissecting The proc Filesystem

The main features of the proc filesystem was described in section 3.9.1. This section describes some of the interfaces of the filesystem that are used by other parts of the kernel including other filesystems.



Much of the proc filesystem code can be found under `fs/proc`. The code for displaying the contents of `/proc/filesystems` is in `fs/filesystems.c`.

To demonstrate a part of procfs, consider the output displayed when looking at the list of available filesystems. There are over 30 entries on my Ubuntu 22.10 VM. Note the output shown here is available and not mounted filesystems.

```
$ cat /proc/filesystems
nodev    sysfs
nodev    tmpfs
nodev    bdev
nodev    proc
...
nodev    sockfs
...
nodev    devpts
        ext3
        ext2
        ext4
        squashfs
        vfat
nodev    ecryptfs
```

There are two components to creating the `filesystems` entry under `/proc` and displaying the contents of the file. First, a call is made to create an entry in the namespace (`fs/filesystems.c`).

```
static int __init proc_filesystems_init(void)
{
    proc_create_single("filesystems", 0, NULL,
                      filesystems_proc_show);
```

```
        return 0;
    }
```

Note that this is a macro and results in a call to `proc_create_single_data()`. In the same file the code to display the contents of the available filesystems is very simple. It simply walks through the list of available filesystems referenced by `file_systems`, displays each entry together with information about whether the filesystem requires a device or not. For example `proc` is a pseudo filesystem so doesn't require a device but the `ext*` filesystems do.

```
static int filesystems_proc_show(struct seq_file *m, void *v)
{
    struct file_system_type * tmp;

    read_lock(&file_systems_lock);
    tmp = file_systems;
    while (tmp) {
        seq_printf(m, "%s\t%s\n",
                  (tmp->fs_flags & FS_REQUIRES_DEV) ? "" : "nodev",
                  tmp->name);
        tmp = tmp->next;
    }
    read_unlock(&file_systems_lock);
    return 0;
}
```

`seq_printf()` is used to display each entry in the file - calls multiple fns to handle arguments. Need to find last one?

A call to `proc_create_single_data()` differs from `proc_create_single()` in that callers may wish to pass data that is stored in the `data` field of the `proc_dir_entry` structure that will be allocated for this node. The data passed will be available on a subsequent call - **XXX—where does this happen or do the callers extract it themselves somehow?**

The function to display the file contents is passed to `proc_create_single()` and is stored XXX

Note that you can pass more than a single name to `proc_create_single_data()` as some modules do:

```
proc_create_single_data("driver/atmel", ...
proc_create_single_data("driver/rtc"
```

6.19.1 Use of Red-Black Trees in /proc

The proc filesystem tree is stored in memory as a *red-black tree* (`struct rb_node`). Red-black trees are a type of self-balancing binary tree, and are used by several places in the kernel for storing sortable key/value data pairs. Each node node in the tree has a bit representing *color* (red or black) which is used to ensure that the tree remains balanced during insertion and delete operations.

You can read the background and details about the Linux red-black tree implementation in Documentation/rbtree.txt.

The root of the proc tree is accessed by `proc_root`. Some of the fields are shown below. The operations for this root node are very simple in terms of returning attributes and providing a *lookup* function so that directories and files under `/proc` can be located and displayed.

```
struct proc_dir_entry proc_root = {
    proc_iops = 0xfffff8000092be640 <proc_root_inode_operations>,
    {
        proc_ops = 0xfffff8000092be700 <proc_root_operations>,
        proc_dir_ops = 0xfffff8000092be700 <proc_root_operations>
        ...
        name = 0xfffff80000984c208 "/proc",
    },
    subdir = {
        rb_node = 0xfffff0000c01bc808
    },
    struct inode_operations proc_root_inode_operations = {
        .lookup      = proc_root_lookup,
        .getattr     = proc_root_getattr,
    }
}
```

Nodes are added to the tree during system initialization or when kernel modules are loaded.

6.19.2 KGDB — Analyzing the `/proc` File Tree

Lookup walks through the `rb_left` / `rb_right` links recursively until it finds the node

There is a `gdb` command `lx_rb_first` / function that can help - see `apropos lx`. There are several of them.

```
crash> p proc_root | grep rb_node
rb_node = 0xfffff5a09c01ba208
crash> tree -t rbtree -o proc_dir_entry.subdir -N 0xfffff5a09c01ba208
fffff5a09c01ba188
fffff5a09c01ba488
fffff5a09c01ba008
fffff5a09c01ba248
fffff5a09c0422308
fffff5a09c01baaa88
fffff5a09c05190c8
fffff5a09c0422788
fffff5a09c01bad88
fffff5a09c01ba908
...
...
```

6.20 Quotas Implementation

TBD

6.21 Handling Pipes

TBD

6.22 Flushing File Data and Metadata

TBD - no other place that covers this?

6.23 Filesystem Interfaces

This section is a reference to the operations that filesystems can support through the following interfaces:

- `super_operations`
- `inode_operations`
- `file_operations`
- `address_space_operations`

It is unlikely that each filesystem will support all of the operations for each of these structures. What is supported generally depends on the features that the filesystem supports in addition to how much control over file operations it requires. For example, some filesystems will have simpler implementations and therefore include generic kernel functions for some aspects.

It isn't necessary to read everything in this section as there is a lot of detail and understanding all interfaces isn't necessary to understanding the basics of filesystem operations. But to go into detail, knowledge of more of these interfaces is necessary so they're all listed here.

6.23.1 The `super_operations` Structure

The `super_operations` structure contains operations that operate on the filesystem as opposed to individual files. Filesystems attach their `super_operations` structure to the `s_op` field of the `super_block` structure during a call to their `fill_super()` function. For SPFS, the flow is shown in figure 6.2 from a call to `module_init()` when the module is loaded followed by a call into SPFS (via `sp_mount()`) each time an SPFS filesystem is mounted.

There are a lot of `super_operations` of which many functions are not implemented by each filesystem. For example, out of the 29 possible functions. 3 of the functions are only used if `CONFIG_QUOTA` is defined. SPFS only implements 6 of these functions.

Below we give a brief description of each function:

- `alloc_inode` – his method is called by `alloc_inode()` to allocate memory for struct `inode` and initialize it. If this function is not defined, a simple 'struct inode' is allocated. Normally `alloc_inode` will be used to allocate a larger structure which contains a 'struct inode' embedded within it.
- `destroy_inode` –
- `free_inode` –
- `dirty_inode` –
- `write_inode` – called when the kernel needs to write an inode to disk. An example would be for file creation. A new file is created and the inode is marked dirty. The same is true for the inode of the parent directory. The filesystem calls `mark_inode_dirty()` for both inodes. The second parameter specifies whether the write should be synchronous or not but not all filesystems check this flag.
- `drop_inode` –
- `evict_inode` – this should only be called when both the inode's `i_nlink` and `i_count` both go to zero. When a file has been removed but may still be referenced by processes, it can't be deleted until the last reference goes away. When this happens, the `evict_inode` operation is called.
- `put_super` – called as part of `umount` before the `super_block` will be freed.
- `sync_fs` –

Freeze/Thaw Functions

The only filesystem to provide support for these operations is gfs2.

- `freeze_super` –
- `freeze_fs` –
- `thaw_super` –
- `unfreeze_fs` –

Information Gathering Functions

- `statfs` – called in response to `df(1)` and `stat -f` commands. See `fs/statfs.c` for kernel routines that handle filesystem statistics. **XXX—need some sort of trace to figure out which of the many functions is called. There are several.**
- `remount_fs` –
- `umount_begin` –
- `show_options` –
- `show_devname` –
- `show_stats` – if a filesystem supports this function, it is called by the proc filesystem (see `proc_namespace.c`) when doing **XXX**
- `show_path` – as above, this function is called by the proc filesystem (`show_mountinfo()`) - **XXX** seems to do very little other than add tabs and stuff. Weird.
- `nr_cached_objects` –
- `free_cached_objects` –

Quotaas-related Functions

- `quota_read` –
- `quota_write` –
- `get_dquots` –

TBD

6.23.2 The `inode_operations` Structure

The `inode_operations` is accessed from the `i_iop` field of the Linux `inode` structure. It is set by the filesystem when new files are created or files are read from disk. This allows the kernel to call into the filesystem for a specific inode.

Several of these options take a `user_namespace` structure as an argument **XXX—how is it used?**

- `readlink` –
- `create` –
- `link` –
- `unlink` –
- `symlink` –

- `mkdir` – create a directory. The parent inode is passed together with the file mode and a negative dentry for the new directory entry. If the call is successful, the filesystem will call `d_instantiate()` to XXX the new file. XXX need right words here.
- `rmdir` – remove a directory. The parent inode is passed together with the dentry for the file to be removed. XXX who "deletes" the dentry? Look at the kernel source.
- `mknod` –
- `rename` –
- `setattr` –
- `getattr` –
- `listxattr` –
- `fiemap` –
- `update_time` –
- `atomic_open` –
- `tmpfile` –
- `set_acl` –
- `fileattr_set` –
- `fileattr_get` –

XXX

6.23.3 The `file_operations` Structure

Functions that apply to regular files are held in the `file_operations` (XXX—except i have one for dirops which has a few functions in it. Need to explain

A pointer to this structures is attached to the `f_op` field of the inode structure.

Filesystems typically define two `file_operations` structures - or do they?

- `llseek` – ext4 dir lseek / XFS does something with holes.
- `read` – XXX - Many filesystems don't provide this function and instead, have all I/O go through the page cache which utilizes the `address_space_operations` structure. procfs and others for which page cache I/O make no sense. There are a lot of them that provide this so need to look.
- `write` – XXX - Many filesystems don't provide this function and instead, have all I/O go through the page cache which utilizes the `address_space_operations` structure.

- `read_iter-`
- `write_iter-`
- `iopoll-`
- `iterate-`
- `iterate_shared-`
- `poll-`
- `unlocked_ioctl-`
- `compat_ioctl-`
- `mmap-`
- `open-`
- `flush-`
- `release-`
- `fsync-`
- `fasync-`
- `lock-`
- `sendpage-`
- `get_unmapped_area-`
- `check_flags-`
- `flock-`
- `splice_read-`
- `splice_write-`
- `setlease-`
- `fallocate-`
- `show_fdinfo-`
- `mmap_capabilities-`
- `copy_file_range-`
- `remap_file_range-`
- `fadvise-`

- uring_cmd -
 - uring_cmd_iopoll) - xx
- xxx

6.23.4 The `address_space_operations` Structure

page cache doc - <http://sy lab-srv.cs.fiu.edu/lib/exe/fetch.php?media=paperclub:lkd3ch16.pdf>
Folios - <https://lwn.net/Articles/869942/>

Memory Folio Functions

Section 6.14.2 described changes in Linux kernel memory management for supporting *memory folios*, a mechanism that allow filesystems and the page cache to manage memory in larger chunks than `PAGE_SIZE` which can, depending on the workload, give a performance enhancement of 0-10%.

The functions that filesystems provide for supporting memory folios, utilize the `folio` structure which is defined in `include/linux/mm_types.h` as:

```
struct folio {  
    union {  
        struct {  
            unsigned long flags;  
            union {  
                struct list_head lru;  
                struct {  
                    void * __filler;  
                    unsigned int mlock_count;  
                };  
            };  
            struct address_space *mapping;  
            pgoff_t index;  
            void *private;  
            atomic_t _mapcount;  
            atomic_t _refcount;  
            unsigned long memcg_data;  
        };  
        struct page page;  
    };  
};
```

Here are the interfaces into the filesystem for supporting memory folios.

- `read_folio` - read XXX looks like this replaced the old `readpage()` function.
- `dirty_folio` -
- `invalidate_folio` -

- release_folio -
- launder_folio -
- free_folio -

XXX check which filesystems do interesting things.

SPFS, which will be introduced in the next chapter, represents the simplest case of filesystem folio support. It simply relies on kernel support functions for handling folios:

```
struct address_space_operations sp_aops = {  
    .dirty_folio      = block_dirty_folio,  
    .invalidate_folio = block_invalidate_folio,  
    .read_folio       = sp_read_folio,  
    .writepage        = sp_writepage,  
    .write_begin      = sp_write_begin,  
    .write_end        = sp_write_end,  
    .bmap            = sp_bmap  
};
```

with `sp_read_folio` being a simple function calling `block_read_full_folio()` passing `sp_get_block()` so that the kernel can get block numbers for specific portions of the file covered by the folio.

```
int  
sp_read_folio(struct file *file, struct folio *folio)  
{  
    return block_read_full_folio(folio, sp_get_block);  
}
```

Other filesystems have more complicated implementations. XXX will be described at some point.

Blah blah Functions

- writepage -
- writepages -
- readahead -
- write_begin -
- write_end -
- bmap -
- direct_IO -
- migratepage -
- isolate_page -

- `putback_page` –
- `is_partially_uptodate` –
- `is_dirty_writeback` –
- `error_remove_page` –

Swapping Functions

- `swap_activate` –
- `swap_deactivate` –
- `swap_rw` –

XXX

6.24 Conclusion

This chapter continued to build on the previous chapter which described the main kernel structures, by describing in detail, the kernel functions in the Linux VFS layer that deal with filesystem access. There has been a lot of material presented here together with a lot of examples of using `gdb` to help analyze the paths through the kernel. It's not an easy subject to study and there is limited documentation online. There are a few, detailed examples, particularly in the area of pathname resolution, XXX and YYY. The goal of this chapter is to pull those areas of information together with areas that are not well documented and give you the tools to explore each area in more detail.

Setting up `gdb` / `kgdb` is often a challenge. For my Macbook Pro with Apple's m1 CPU, I spent the better part of six months trying to find answers about how to get breakpoints working. At the time of writing, my only option was to emulate x86_64 VMs which resulted in a kernel compilation that was about 17 times longer than compiling for the native CPU. But it's a task that I think is well worth the effort. Setting breakpoints and walking through the kernel code is the best way to understand what is happening.

The next chapter will help with putting the pieces together by showing how to implement a simple, disk-based filesystem and use debugging tools to help better understand the flow through the kernel into the filesystem in response to various system calls.

Chapter 7

Building a Linux Filesystem

XXX—add proc_create_single() call and display files accessed by walking through the inode list

There are two ways to understand filesystems. Either you can study one of more of the 80+ existing filesystems or you can develop one. The issue with studying existing filesystems is that they all have their peculiarities which make analyzing the source code more complicated. For example, I've looked at BFS which is a simple filesystem developed for UNIX SVR4 (System V Release 4) for the /stand filesystem. It's a filesystem I worked with many years ago. It only has 1200 lines of kernel source code but still has a few twists. It's background:

It was designed to hold all binaries needed during the boot phase. It's simple disk layout helped with simplifying the boot loader which didn't need to know about the structure of more complex filesystems such as UFS and VxFS which were commonplace on SVR4 at the time. As such it was not designed for general purpose usage.

- It's a flat filesystem (only files within a single root directory).
- Only regular files can be created. You can't create symbolic links.
- Files are allocated contiguously, again to simplify boot. This makes writing files more complex since blocks need to be moved to retain this requirement.
- XXX - anything else?

It would be better to have a simple filesystem without these limitations but with a goal of keeping it about the same size in terms of lines of code. I developed such a filesystem for my first book (UNIX Filesystems - Design and Implementation) in 2003 for the Linux 2.4 kernel. I've used that as a base (things have changed quite a bit since) but hopefully made it easier to understand by adding more tracing and fixing some bugs that slipped under the rug. This new filesystem is called SPFS (no prizes for guessing the name).

Here are the main characteristics of SPFS:

- Multi-level directories (directories within directories)

- Fixed block size (2048 bytes).
- A filename length up to 28 characters.
- 760 blocks within the whole filesystem.
- A maximum file size of approximately 505 KB.
- File undelete using the SPFS fsdb command.
- File creation, deletion, rename, ... XXX

Most of these limitations are in place to keep the on-disk structures very simple which result in the code to access them being much easier to understand. The goal here is for teaching.

7.1 The SPFS on-disk Format

As discussed earlier, simplicity is key. We are not trying to design a high-performing, crash-resistant filesystem but one with the simplest structure and the least amount of code (around 2,000 lines including mkfs, fsdb and lots of debug messages and debug ioctl).

The filesystem block size for SPFS is 2-38 bytes. If we have a regular file with size of 0, it will have no allocated blocks. If the size of the file is \leq 2048 bytes it will have 1 block. If the size of the file is 2049 bytes it will have 2 blocks allocated and so on. The same is true for directories. There is never an empty directory because a newly allocated directory will always have entries for "." and "..". Directory entries are 32 bytes so a new directory will have one allocated 2048 byte block and a size of 64 bytes.

The superblock is limited to one block due to the way superblocks are accessed as part of mount processing. Since this structure maintains a list of used/unused inodes and blocks in the filesystem, this is where our limitation of the filesystem size and number of files comes from. To increase the size of the superblock beyond 1024 bytes would add additional complexity which we are trying to avoid.

There are three components to the disk layout as shown in figure 7.1.

1. **Superblock** — this is stored in block 0.
2. **Inodes** — stored in blocks 1-128. There is one inode per block. This is very inefficient in terms of space but as we shall see later, locating the block where the inode is stored is very simple. It's just block 1 + inode number. For example, the root inode is stored at block 3. There can only be 128 files in the filesystem and since inodes 0, 1 aren't used and inode 2 is for the root directory, a maximum of 124 new files that can be created.
3. **Data blocks** — stored from block 129 onwards. Data blocks are used for regular file data as well and directory entries for directories and symbolic links.

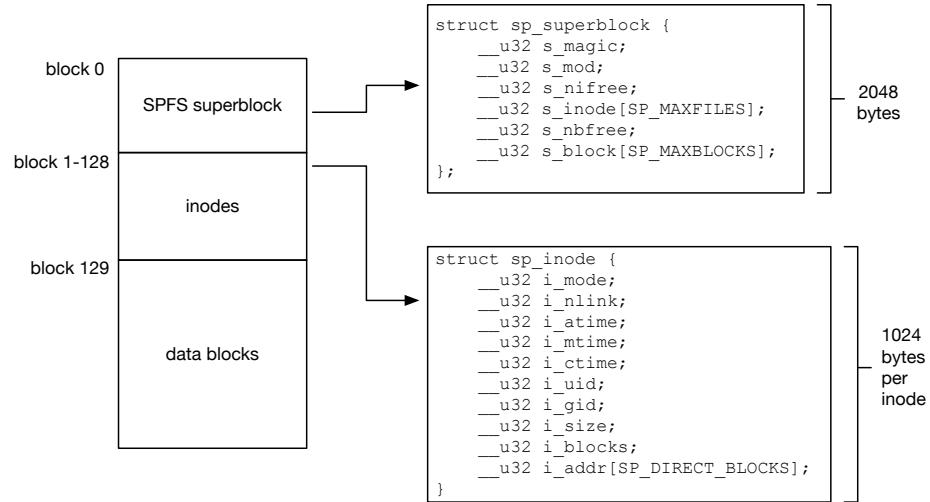


Figure 7.1: Filesystem layout on disk

7.2 A Note About Printing Pointers

There are several places in the code where I print out pointers. This helps with debugging. If structures are still valid, I can drop into `gdb` and display them easily. If you make a call such as:

```
printf("*** set i_private to \%p\n", spi);
```

you will see something along the lines of:

```
*** set i_private to 0000000aaale52ff
```

which is not a valid address. You should be seeing an address more along the lines of:

```
*** set i_private to ffff0000c98dd470
```

This is because an address printed with `%p` will first be hashed to prevent the real address from being exposed. On 64-bit machines, the first 32 bits are zeroed before printing as seen above.

To see the real address, use `%px` in place of `%p`. For development purposes this is fine but for production code, be well aware of what you're printing in case there are potential security issues with exposing kernel addresses.

7.3 From Module Load to Mounting a Filesystem

In 7.2 we show which functions are called as part of loading the module (and unloading) and how the filesystem's mount function is identified to the kernel and called.

Loading a filesystem module and how “mount” is identified and called

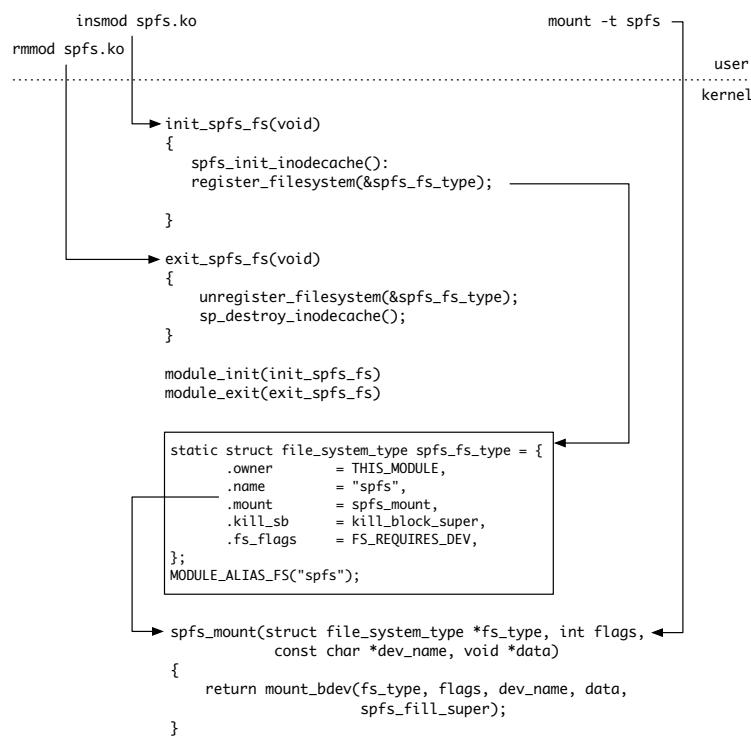


Figure 7.2: Module Initialization

First of all we declare our filesystem module through the `file_system_type` structure (shown in gray). Here we have:

- The name of the filesystem.
 - The filesystem's `mount` function that should be called when a filesystem of type `spfs` is mounted (`mount -t spfs`).
 - The function that should be called during `umount` processing to clean up everything. XXX - need to figure out if anything is left other than setting the SB to CLEAN and flushing it.

There are two functions declared:

- `init_spfs_fs()` - this will be called when the module is loaded (either `insmod` or `mount -t spfs XXX` need to figure out the `MODULE_ALIAS_FS("spfs")` stuff)
 - `exit_spfs_fs()` - this is called when the module is unloaded (`rmmod`).

When the module is loaded, the first thing that `init_spfs_fs()` performs is to register the file system by calling `register_filesystem()`. The main goal of this function is to check that the filesystem has not already been registered. If not, it is added to the list of available filesystems as shown in figure 7.3 . XXX - need to check this with a debugger to show how many filesystems and make sure spfs is last on the list.

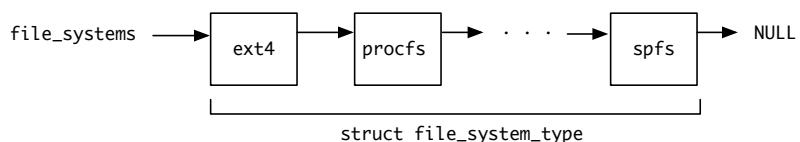


Figure 7.3: The list of available Linux filesystems that can be mounted

The `file_systems` variable can be found in the file `fs/filesystems.c` next to `register_filesystem()` and associated functions.

When issuing a mount -t spfs call, the kernel can walk through this list checking the spfs argument against the name field of each `file_system_type` structure in the list. If there is a match, the kernel can locate the filesystem-specific mount function and call it.

7.4 Initializing the per-filesystem Inode Cache

Every filesystem has its own `init_inodecache()` function for maintaining a list of per-filesystem inodes (all mounts?). When allocating a Linux inode, a call is made to `alloc_inode_sb()` (`linux/fs.h`) where they say:

This must be used for allocating filesystems specific inodes to set up the inode reclaim context correctly.

When creating or opening a file, a Linux inode is created. There is one inode regardless of the number of open file descriptors. To populate many of the fields of the inode, the kernel requests the filesystem to allocate the inode. For existing files, this involves reading the corresponding SPFS inode from disk. Much of the information about the file is used to populate the Linux inode but there is some information that SPFS must also keep in-core. This information relates to where the data blocks for the file are stored on disk (all file types—regular, directories and symlinks) and how many blocks have been allocated so far.

Here is both the SPFS in-core inode and the SPFS disk inode. The Linux inode is embedded within the SPFS in-core inode. All inodes for this mounted filesystem are referenced from the `super_block` structures as shown in figure 7.4. When a `sp_inode_info` structure is allocated, we store "SPFS" in the `i_fs` field. This is just for debugging purposes to show the structure type. It will be shown in different KGDB demonstration sessions throughout the book.

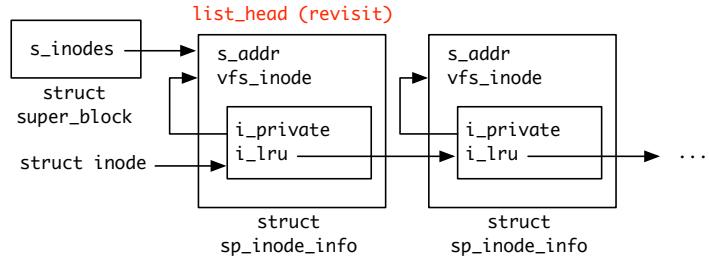
```
struct sp_inode_info {
    char          i_fs[4];
    int           i_blocks;
    int           i_addr[SP_DIRECT_BLOCKS];
    char          i_symlink[SP_NAMELEN];
    struct inode  vfs_inode;
};
```

The disk-based inode contains more fields. All but the last two fields are copied to the Linux inode so there is no point replicating them in the in-core SPFS inode.

```
struct sp_inode {
    __u32   i_mode;
    __u32   i_nlink;
    __u32   i_atime;
    __u32   i_mtime;
    __u32   i_ctime;
    __u32   i_uid;
    __u32   i_gid;
    __u32   i_size;
    __u32   i_blocks;
    __u32   i_addr[SP_DIRECT_BLOCKS];
};

#define ITOSPI(inode)  (struct sp_inode_info *)&inode->i_private
```

Many calls into SPFS pass the Linux inode. We often need to access the SPFS in-core inode which we can do through use of the `ITOSPI()` macro. We'll discuss how this LRU list is used later on in the book XXX.

Figure 7.4: Inode LRU List referenced from the `super_block` structure

7.4.1 KGDB - Analyzing Inode Lists for a Specific Mountpoint

Section 5.5 describes how the `container_of()` inline function is used to located a structure given an element within the containing structure. The previous section described how this is used for Linux VFS inodes and SPFS inode information. In this section we'll describe how to analyze inode lists associated with a specific mount point.

First, we'll show how to walk the list by hand in `gdb` which is quite painful, especially if there are a lot of elements in the list. We'll then combine this approach with `crash` which makes walking linked lists much easier. **XXX—can we do this for `for_each()` or something like that?? - or write some `gdb` scripts at some point.**

The goal is to walk the inode list and located an inode that matches a specific file. Here are the files in the filesystem being used. We'll located `file-3` which has an inode number of 6.

```
$ ls -li /mnt
total 1
4 -rw-r--r-- 1 root root 0 Feb 7 17:26 file-1
5 -rw-r--r-- 1 root root 0 Feb 7 17:26 file-2
6 -rw-r--r-- 1 root root 0 Feb 7 17:26 file-3
3 drwxr-xr-x 2 root root 64 Feb 7 17:26 lost+found/
```

All mounted filesystems are on a doubly linked list headed by the `super_blocks` global variable:

```
(gdb) p super_blocks
$1 = {next = 0xfffff0000c0020800, prev = 0xfffff0000cdc18000}
```

Looking at the top of the `super_block` structure (one per mounted filesystem) we can see that the `s_list` field keeps all such structures in a linked list:

```
struct super_block {
    struct list_head s_list;      /* Keep this first */
    ...
}
```

Thus the two addresses shown above are the addresses of the first `super_block` in the list and also the last one.

Inodes are different in that the list is embedded part way down the inode structure so we will need to use `container_of()` to get to the actual inode whereas we can use the addresses above directly to reference the `super_block` structure.

```
struct inode {
    ...
    struct list_head    i_sb_list;
    ...
}
```

We're going to look at the last filesystem mounted so we can look at the `super_block` structure shown by `prev` and get the inode list which is stored in the `s_inodes` field. Take note of the first address in bold (`0xfffff000004418518`) which is the first inode in the list and when it is shown later when printing out `$14`. In the second case it's the `next` element in the double linked list shown that we've walked through the whole list.

```
(gdb) p ((struct super_block *)0xfffff0000cdc18000)->s_inodes
$5 = {
    next = 0xfffff000004418518,
    prev = 0xfffff0000044f3f98
}
```

Now the fun begins! We walk this list by hand from the head of the list to the tail. Luckily it's actually a very short list.

```
(gdb) p ((struct list_head *)0xfffff000004418518)->next
$6 = (struct list_head *) 0xfffff0000b38e1f18
(gdb) p *$6
$9 = {
    next = 0xfffff0000b38e3918,
    prev = 0xfffff000004418518
}
(gdb) p ((struct list_head *)0xfffff0000b38e3918)->next
$12 = (struct list_head *) 0xfffff0000b38e2598
(gdb) p *$12
$13 = {
    next = 0xfffff0000044f3f98,
    prev = 0xfffff0000b38e3918
}
(gdb) p ((struct list_head *)0xfffff0000044f3f98)->next
$14 = (struct list_head *) 0xfffff0000cdc18588
(gdb) p *$14
$15 = {
    next = 0xfffff000004418518,
    prev = 0xfffff0000044f3f98
}
```

Let's pick one of these addresses, say `0xfffff000004418518` which is highlighted. We can use `container_of()` go go from this address to the address of the inode in question. Here is the call. This particular inode list is referenced from the `i_sb_list` field of the

inode structure so we pass this address, the structure type and the structure element name for which the address applies:

```
(gdb) set $inode = *$container_of(0xfffff000004418518, \
"struct inode", "i_sb_list")
```

Now we have the inode, we can print out the inode number which matches the inode number displayed when running `ls` earlier.

```
(gdb) p $inode.i_ino
$19 = 6
```

Getting to a `dentry` structure to find the filename requires another call to `container_of()` since there could be a list of dentries (if there were hard links to the same file and more than one was accessed). First thing is to locate the list inside the inode:

```
(gdb) p $inode.i_dentry.first
$23 = (struct hlist_node *) 0xfffff0000b3b409b0
```

and then call `container_of()` again to get the first dentry:

```
(gdb) set $dentry = *$container_of(0xfffff0000b3b409b0, \
"struct dentry", "d_u")
```

The `dentry` structures are also linked as follows which is why we use the `d_u` field.

```
struct dentry {
    ...
    union {
        struct hlist_node d_alias;
        struct hlist_node d_in_lookup_hash;
        struct rcu_head d_rcu;
    } d_u;
} __randomize_layout;
```

Now we have our `dentry`, we can display the file name:

```
(gdb) pipe p $dentry | grep 'name = "'\n      d_iname = "file-3", '\\"000' <repeats 25 times>,
```

A better alternative to walking Linux kernel linked lists by hand is to define a `gdb` function to do the work for us.

```
define plist
set var $n = $arg0->head
while $n
    printf "%d ", $n->data
    set var $n = $n->next
end
end
```

XXX—this does not work! At least I've tried it and it doesn't work. Python may be a better choice

The `crash(1)` command has a nice command to walk through lists so rather than the previous example in `gdb` where we walked through by hand, one command does it in

crash. First of all, we follow the same path as the earlier example of finding the last mounted filesystem to get its `super_block` structure and from there, find the list of inodes. In this example, we have a different filesystem. Here are the files in the `/mnt` directory:

```
$ ls -li /mnt
total 28
15 -rw-r--r-- 1 root root      6 Feb  8 19:03 file-1
14 -rw-r--r-- 1 root root      6 Feb  8 19:04 file-2
16 -rw-r--r-- 1 root root      6 Feb  8 19:03 file-3
11 drwx----- 2 root root 16384 Feb  3 15:51 lost+found/
```

The first step is to find the list of `super_block` structures and, knowing that this was the last filesystem mounted, use the `prev` field, display the `super_block` structure for this filesystem and get the list of associated inodes:

```
crash> super_blocks
super_blocks = $2 = {
    next = 0xfffff5a09c0026000,
    prev = 0xfffff5a09d2fcb000
}
crash> super_block 0xfffff5a09d2fcb000
...
s_inodes = {
    next = 0xfffff5a09d3ffb950,
    prev = 0xfffff5a09d3ff86c8
},
...
```

Using the head of the list (shown in bold), the `list` command is called to display all entries. Magical!

```
crash> list 0xfffff5a09c1f27318
fffff5a09d3ffb950
fffff5a09c0ef8ff8
fffff5a09c0eff070
fffff5a09197f79a0
fffff5a09d3ffa6f0
fffff5a09d3ff9dc0
fffff5a09d3ff86c8
fffff5a09d2fcb548
```

I've already looked at some of the inodes to find out which ones correspond to our files so picking the address highlighted, here is teh equivalent of calling `container_of()` in `gdb`. The `-l` option tells `crash` to display the containing inode structure using the structure element `i_sb_list` as a field within this structure.

```
crash> inode -l inode.i_sb_list fffff5a09c0ef8ff8
...
i_ino = 14,
{
```

```

    i_nlink = 1,
    __i_nlink = 1
},
i_rdev = 0,
i_size = 6,
...
i_dentry = {
    first = 0xfffff5a0900dc10b0
},
...

```

You can now see the inode number and file size which matches `file-2` displayed by `ls` above. To confirm the file name, we need to use the same technique to find the `dentry` structure:

```

crash> dentry -l dentry.d_u 0xfffff5a0900dc10b0
d_name = {
{
    hash = 2573278399,
    len = 6
},
hash_len = 28343082175
},
name = 0xfffff5a0900dc1038 "file-2"
},
d_inode = 0xfffff5a09c0ef8ee0,
d_iname = "file-2\000\0 ... \000\000",

```

This is quite a bit simpler than walking the linked list by hand with `gdb` but is otherwise very similar.

7.5 Mounting a Filesystem

When a filesystem is mounted, the filesystem type is passed as an argument, for example:

```
# mount -t spfs /dev/sdb1 /mnt
```

The kernel uses this argument to find the correct `file_system_type` structure (see figure 7.3) and call the filesystem's `mount` function. Recall that this is one of the fields in `struct file_system_type` and was passed to the kernel during module load by calling `register_filesystem()`.

Here are the set of calls (viewable using `dmesg`) made from the kernel into SPFS to complete the mount call. The root inode of the filesystem is inode 2.

```
spfs_mount() → spfs_fill_super() → sp_read_inode()
```

Following a call to `mount`, any operation on this filesystem must start with the root directory and thus, the kernel will call the `read_inode()` function in the `super_operations` structure to read the root inode in from disk.

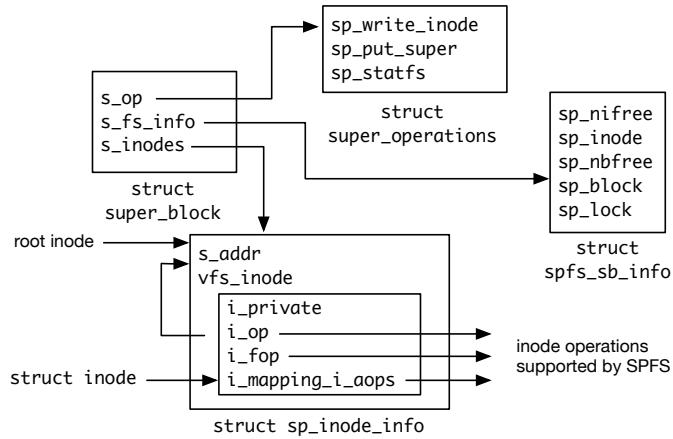


Figure 7.5: Per-mount structures following a call to `sp_fill_super()`

Xxx - Need a figure of kernel structures for this

7.5.1 Inside `sp_mount` and `sp_fill_super`

The code for `spfs_mount()` is very simple, relying solely on a call to `mount_bdev()`. It passes `spfs_fill_super()` as an argument.

```
static struct dentry *
spfs_mount(struct file_system_type *fs_type, int flags,
           const char *dev_name, void *data)
{
    return mount_bdev(fs_type, flags, dev_name, data,
                      spfs_fill_super);
}
```

After calling `spfs_fill_super()`, figure 7.5 shows how the structures are linked together.

At this point we have enough structure in place to allow us to do the following type of operations:**XXX—we can do more than this so elaborate**

- Get filesystem information (think about calling the `df(1)` command) through use of the `sp_statfs()` function.
- List the contents of the root directory.
- Lookup files in a specified directory.

As we progress through the filesystem, more inodes will be read into memory and from there we can perform all the basic filesystem operations. **XXX - we'll expand ...**

For the moment, let's look more at `sp_fill_super()` ...

7.6 SPFS super_operations

So we have a separation from mount and unmount (see module loading section below)

```
static const struct super_operations spfs_sops = {
    .alloc_inode. = sfs_alloc_inode,
    .free_inode. = sfs_free_inode,
    .write_inode = sfs_write_inode,
    .evict_inode = sfs_evict_inode,
    .put_super. = sfs_put_super,
    .statfs. = sfs_statfs,
};
```

Any of the operations that your filesystem provides are called by the kernel once the filesystem is mounted.

bfs_put_super() doesn't do much other than free the structure. I guess because this is a read-only FS? It was similar for ufs which is incomplete as we don't set the CLEAN / DIRTY flag in the superblock correctly. XXX - see what others are doing here

7.6.1 KGDB – Analyzing the Effects of a Mount Operation

When the integration with the proc filesystem is done, grab the superblock address from there.

Figure 7.5 showed the per-mount structures following a call to `sp_fill_super()` to mount an SPFS filesystem. This example shows how to view these structures from within gdb.

After the breakpoint is set, an SPFS filesystem is mounted as follows:

```
$ sudo mount -t spfs /dev/loop7 /mnt
```

which will hit the breakpoint for which the stack backtrace is:

```
(gdb) bt
#0  spfs_fill_super  (sb=sb@entry=0xffff888123bef800,
                     data=data@entry=0x0 <fixed_percpu_data>,
                     silent=silent@entry=0)
    at /home/spate/spfs/kern/sp_inode.c:321
#1  mount_bdev  (fs_type=fs_type@entry=0xfffffffffa06adaa0
                 <spfs_fs_type>, flags=flags@entry=0,
                 dev_name=dev_name@entry=0xffff88810a8314f0 "/dev/loop7",
                 data=data@entry=0x0 <fixed_percpu_data>,
                 fill_super=fill_super@entry=0xfffffffffa06abbb8
                 <spfs_fill_super>) at fs/super.c:1367
#2  spfs_mount  (fs_type=0xfffffff0a06adaa0 <spfs_fs_type>,
                 flags=0, dev_name=0xffff88810a8314f0 "/dev/loop7",
                 data=0x0 <fixed_percpu_data>)
    at /home/spate/spfs/kern/sp_inode.c:420
#3  legacy_get_tree  (fc=0xffff88812eb326c0)
    at fs/fs_context.c:610
#4  vfs_get_tree  (fc=fc@entry=0xffff88812eb326c0)
```

```
        at fs/super.c:1497
#5 do_new_mount (data=0x0 <fixed_percpu_data>,
    name=0xfffff88810a831130 "/dev/loop7",
    mnt_flags=<optimized out>, sb_flags=<optimized out>,
    fstype=<optimized out>, path=0xfffffc900011b7e70)
    at fs/namespace.c:3040
#6 path_mount
    (dev_name=dev_name@entry=0xfffff88810a831130 "/dev/loop7",
    path=path@entry=0xfffffc900011b7e70,
    type_page=type_page@entry=0xfffff888127b4a8b0 "spfs",
    flags=<optimized out>, flags@entry=0,
    data_page=data_page@entry=0x0 <fixed_percpu_data>)
    at fs/namespace.c:3370
#7 do_mount (data_page=0x0 <fixed_percpu_data>,
    flags=0, type_page=0xfffff888127b4a8b0 "spfs",
    dir_name=<optimized out>,
    dev_name=0xfffff88810a831130 "/dev/loop7")
    at fs/namespace.c:3383
#8 __do_sys_mount (data=<optimized out>, flags=0,
    type=<optimized out>, dir_name=<optimized out>,
    dev_name=<optimized out>) at fs/namespace.c:3591
    ...

```

Take notice of the address for the `super_block` structure passed to `spfs_fill_super()`. Enter "c" to continue to let the mount complete and then hit "ctrl-c" to enter gdb once again. The SPFS superblock is referenced from the `s_fs_info` field of the `super_block` structure.

```
(gdb) set $sb = (struct super_block *)0xfffff888118af5000
(gdb) p $sb->s_fs_info
$4 = (void *) 0xfffff88810cfca000
(gdb) set $spfs = (struct sp_superblock *)0xfffff88810cfca000
(gdb) p *$spfs
$5 = {
    s_magic = 124,
    s_mod = 0,
    s_nifree = 1,
    s_inode = {0, 1, 0, 1, 0, 1, 0 <repeats 122 times>},
    s_nbfree = 0,
    s_block = {0 <repeats 252 times>, 629, 0, 0, 0, 1, 0, 0, 0,
    1, 0 <repeats 499 times>}
}
```

You can keep checking back at a later time to see how these fields are updated as files are created and blocks are allocated.

Below, we located the root inode which is referenced indirectly from its `dentry` that is stored in the `s_root` field of the superblock structure:

```
(gdb) p $sb->s_root
$27 = (struct dentry *) 0xfffff88810315a240
```

```
(gdb) p $sb->s_root->d_name.name
$22 = (const unsigned char *) 0xfffff88810315a278 "/"
(gdb) p $sb->s_root->d_inode->i_ino
$31 = 2
(gdb) p $sb->s_root->d_inode->i_op
$29 = (const struct inode_operations *) 0xfffffffffa06ad300 <sp_dir_inops>
(gdb) p $sb->s_root->d_inode->i_fop
$30 = (const struct file_operations *) 0xfffffffffa06ad3c0 <sp_dir_operations>
```

The root inode is then located and the inode number (2) is displayed together with the operations vector that is set inside `sp_read_inode`.

7.7 Unmounting

When issuing the `umount` system call, there can be a lot of data in-core that needs to be written to disk. Let's start with an example of calls that are made to SPFS

```
# mount -t spfs /mnt
# mkdir /mnt/mydir
# mkdir /mnt/mydir/next_dir
# umount /mnt
```

Just looking at the operations, you can see that we will create two new inodes for the directories that we're creating and we will also make modifications to the root directory. Since we have allocated two new inodes and corresponding data blocks to store their directory entries, we have also made changes to the SPFS superblock.

```
sp_write_inode (ino=4) - /mnt/mydir
sp_write_inode (ino=2) - root
sp_write_inode (ino=5) - /mnt/mydir/next_dir

sp_evict_inode - nlink > 0 (ino=2)
sp_evict_inode - nlink > 0 (ino=4)
sp_evict_inode - nlink > 0 (ino=5)

sp_put_super

sp_free_inode (ino=2)
sp_free_inode (ino=4)
sp_free_inode (ino=5)
```

XXX - Why is evict being called if we've already done write_inode?

In all 3 cases, `inode->i_count = 0` and this is a check that is made from `evict_inodes()` (`fs/inode.c`) - this is called by `generic_shutdown_super()` which then calls `put_super()`. It gets complicated here so we shall have to revisit.

```
kill_anon_super() -> generic_shutdown_super()
-> evict_inode -> put_super
```

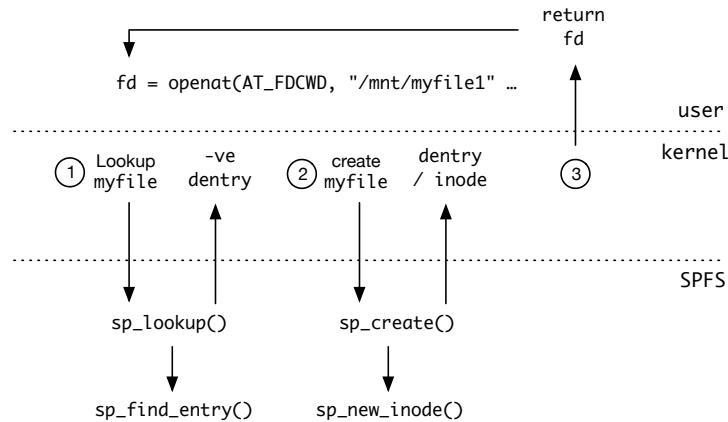


Figure 7.6: Calls into SPFS when creating a file

7.8 Creating a File

lookup is mentioned in the next section but just for create. need a section that just covers lookup I think

7.9 Creating a File

Before we explore reading and writing files, file creation has several interactions between the kernel and SPFS. Let's view the calls that the kernel makes in response to:

```
# mount -t spfs /dev/sda /mnt
# touch /mnt/myfile
# umount /mnt
```

The `touch(1)` command will invoke the `openat(2)` system call as follows:

```
openat(AT_FDCWD, "/mnt/myfile1",
       O_WRONLY|O_CREAT|O_NOCTTY|O_NONBLOCK, 0666)
```

The `openat(2)` system call is identical to `open(2)` except that the path argument is interpreted relative to the starting point implied by the first argument. If this argument has the special value `AT_FDCWD`, the filename argument will be resolved relative to the current working directory. If the path argument is absolute, the first argument is ignored. This is irrelevant in this case as we're specifying the full pathname. **XXX** need to come back to this at some point.

Figure 7.6 show the calls made into SPFS in response to a request to create a new file. The root directory inode (now at `/mnt`) was instantiated during mount processing so the kernel can query using this inode. If you recall, when a directory inode is instantiated, SPFS attaches the following operations to the `i_op` field of the Linux inode.

```

struct inode_operations sp_dir_inops = {
    create:    sp_create,
    lookup:    sp_lookup,
    mkdir:     sp_mkdir,
    rmdir:     sp_rmdir,
    link:      sp_link,
    unlink:    sp_unlink,
};

```

The first call that the kernel makes into SPFS when creating a file is `sp_lookup()` on the root inode (directory). It passes `myfile` as an argument in addition to the directory inode and a dentry to associate with the file whether it exists or not. In turn, `sp_lookup()` calls `sp_find_entry()` to scan the list of directory entries for the specified directory. As expected, this fails since the file does not exist. Before returning, `sp_lookup()` will call `d_splice_alias(inode, dentry)` to create a negative dentry for `myfile`. If another lookup operation were to occur on this file, the kernel now knows that the file doesn't exist.

The kernel then calls `sp_create()` to now create the file. Once `sp_create()` creates the new file, it calls `d_instantiate(dentry, inode)` to map the entry for `myfile` to the inode just created.

7.9.1 Inside `sp_find_entry()`

This function is called to locate a file in the specified directory inode. Before we show the source code, let's assume that the root directory has 96 entries. The disk structure for this would resemble figure 7.7.

When the filesystem is created, we allocate block 50 as the first data block of the root inode. This is stored in the `i_addr` field of the on-disk inode. Inside the actual data block we write directory entries for ".", ".." and "lost+found". As time goes by and new files are created, we will also add them to block 50 until there is no more space. At this point, we allocated a new block (`i_addr[1] = 58`) and start adding the new directory entries in this block. And so on. A simplified form of the algorithm to search for a directory entry is:

When the filesystem is created, we allocate block 50 as the first data block of the root inode. This is stored in the `i_addr` field of the on-disk inode. Inside the actual data block we write directory entries for ".", ".." and "lost+found". As time goes by and new files are created, we will also add them to block 50 until there is no more space. At this point, we allocated a new block (`i_addr[1] = 58`) and start adding the new directory entries in this block. And so on. A simplified form of the algorithm to search for a directory entry is:

```

for each allocated block in the inode do
    for all 32 directory entries in the block do
        if the name to find matches the name in this entry then
            return the inode number found in this entry
        end if
    end for
end for

```

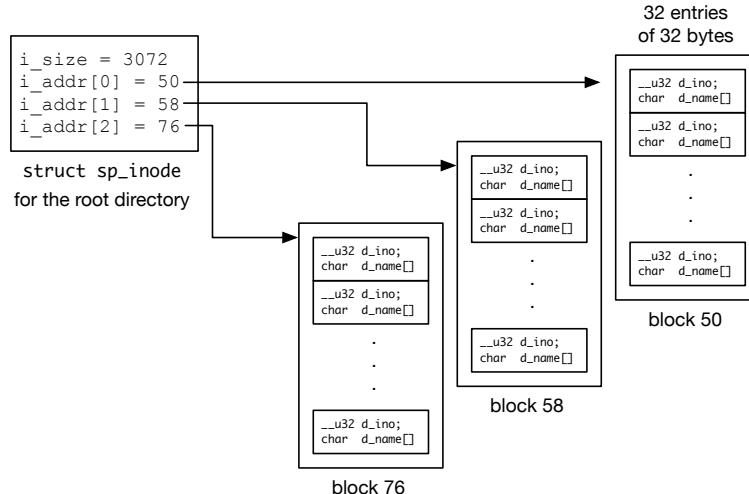


Figure 7.7: The root directory with 96 entries

```
return 0 /* we didn't find an entry */
```

The code for `sp_find_entry()` is shown below. If the root directory had 3 allocated blocks as shown in figure 7.7 we would require reading 3 1024 byte blocks from disk before we determine that the file does not exist.

```
int
sp_find_entry(struct inode *dip, char *name)
{
    struct sp_inode_info      *spi = ITOSPI(dip);
    struct super_block         *sb = dip->i_sb;
    struct buffer_head        *bh;
    struct sp_dirent          *dirent;
    int                         i, blk = 0;

    printk("spfs: sp_find_entry - looking for %s (dip = %p)\n", name, dip);
    for (blk=0 ; blk < spi->i_blocks ; blk++) {
        bh = sb_bread(sb, spi->i_addr[blk]);
        dirent = (struct sp_dirent *)bh->b_data;
        for (i=0 ; i < SP_DIRS_PER_BLOCK ; i++) {
            if (strcmp(dirent->d_name, name) == 0) {
                brelse(bh);
                printk("spfs: sp_find_entry - found inum %d for %s\n",
                       dirent->d_ino, name);
                return dirent->d_ino;
            }
            dirent++;
        }
    }
}
```

```

        }
    }
    brelse(bh);
    printk("spfs: sp_find_entry - failed for %s\n", name);
    return 0;
}

```

If the file is found, we return its inode number (allowing for a subsequent call to read the inode from disk). If it is not found, we return 0. XXX—recall that inode number 0 is not used. For this reason.

7.9.2 Inside `sp_create_file()`

There are 3 different types of file creation in SPFS. Although we are covering regular file creation, we'll touch on the other two since all three operations share common paths. Let's take a look at what's needed for each but first, what is needed for all 3:

- Allocate an inode on disk resulting in superblock changes.
- Allocate a Linux in-core inode together with its corresponding in-core SPFS inode.
- Update fields of both inodes to specify file properties.
- Write the SPFS inode to disk.
- Flush the superblock to disk to reflect the changes after everything else is done.

For each of the 3 operations, there are things that are unique to the file type that we must take care of:

1. Creating a regular file.
 - Nothing. Just make sure the size of the file (`i_size`) is set to 0.
2. Creating a directory.
 - Allocate a block for initial directory entries ("." and "..").
 - Write this block to disk.
 - Update inode properties (`(i_size = 2 * (sizeof(struct sp_direct), i_blocks = 1)`)
3. Creating a symbolic link.
 - Allocate a block to store the target name
 - Write this block to disk.
 - Update inode properties (`(i_size = 2 * (sizeof(struct sp_direct), i_blocks = 1)`)

Having said all of that, the core for (`sp_create_file()`) is very simple:

```
int
sp_create(struct user_namespace *mnt_userns, struct inode *dip,
          struct dentry *dentry, umode_t mode, bool excl)
{
    struct inode     *inode;
    char            *name = (char *)dentry->d_name.name;
    int             error;

    inode = sp_new_inode(dip, dentry, S_IFREG | mode, NULL);
    if (IS_ERR(inode)) {
        error = PTR_ERR(inode);
        goto out;
    }
out:
    return error;
}
```

We simply rely on a call to `sp_new_inode()` to do all of the work. The source code for `sp_mkdir()` and `sp_symlink()` is almost identical. The exception is the last argument to `sp_new_inode()` which we will cover during the section on symlinks (XXX).

7.9.3 Inside `sp_new_inode()`

```
for each allocated block in the inode do
    for all 32 directory entries in the block do
        if the name to find matches the name in this entry then
            return the inode number found in this entry
        end if
    end for
end for
return 0 /* we didn't find an entry */
```

7.9.4 Inside `sp_ialloc()`

There are two fields in the SPFS superblock (in-core and on disk) that record how many inodes have been allocated (`s_nifree` and which inodes have been allocated (the array `cfs_inode`).

```
ino_t
sp_ialloc(struct super_block *sb)
{
    struct spfs_sb_info  *sbi = SBTOSPFSSB(sb);
    int                  i;

    if (sbi->s_nifree == 0) {
        printk("spfs: Out of inodes\n");
    } else {
        mutex_lock(&sbi->s_lock);
```

```

        for (i = 4 ; i < SP_MAXFILES ; i++) {
            if (sbi->s_inode[i] == SP_INODE_FREE) {
                sbi->s_inode[i] = SP_INODE_INUSE;
                sbi->s_nifree--;
                printk("spfs: sp_ialloc alloc inode %d\n", i);
                mutex_unlock(&sbi->s_lock);
                break;
            }
        }
        return i;
    }
}

```

We check up front if any inodes are available. If not we return 0. If there is at least one inode available, we grab the SPFS superblock lock and walk through the array. As soon as we find an entry marked free, we set it to SP_INODE_INUSE, drop the superblock lock and return.

7.9.5 More on Negative dentries

As mentioned above, when a call is made to `sp_lookup()`, the file `myfile` does not exist so a negative dentry is created. In our example, the kernel follows up with a call to `sp_create()` so why is this necessary? There are times that a request will be made to test for the presence of a file as the following example shows:

```
# stat /mnt/myfile
stat: cannot statx '/mnt/myfile': No such file or directory
```

In figure 7.6 steps 1 and 2 will be performed by the kernel to see if the file exists. This will more than likely result in the filesystem reading from disk and there could be multiple reads depending on how many directories exist. Since the file does not exist, it returns ENOENT and retains the negative dentry.

If another `stat(1)` call is made, the kernel doesn't need to call the filesystem again since it now knows that the file does not exist. And thus the role of negative dentries.

7.10 Creating a Directory

We covered the basics of directory creation earlier during the section 7.9.2. The steps performed are very straightforward. We have expanded on them here.

1. Allocate a block for initial directory entries ("." and "..").
2. Allocate an inode and set the inode properties (size, owner, group etc). The size will be set to `2 * sizeof(struct sp_direct)`. Also reference the newly allocated block from within the inode allocated for the new directory.
3. Update the parent directory to add a new directory entry that points to the new file. This may also result in allocating a new data block for the parent directory if needed.

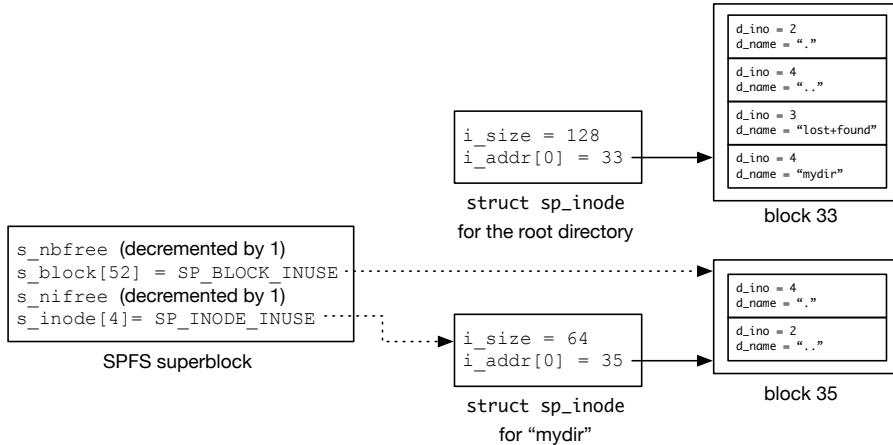


Figure 7.8: Structures Allocated/Modified on Disk When Creating a Directory

Figure 7.8 is a visual presentation of how structures on disk are allocated and/or changed. In this example we assume we are creating new directory "mydir" in the root directory (thus ".." referencing inode 2). The filesystem is empty so we only have inodes for "/" and "lost+found" for which we have allocated data blocks 33 and 34 for the root directory and lost+found directory entries.

In this example:

- We have allocated a new inode (number 4) for mydir. We set the superblock `s_inode[4]` field to `SP_INODE_INUSE` and decrement `s_nifree`, the number of inodes free.
- We have allocated a new disk block (number 35) for mydir. We set the superblock `s_block[52]` field to `SP_BLOCK_INUSE` and decrement `s_nbfree`, the number of blocks free.
- Fields of the inode for my_dir needs to be filled in (owner, group, size) etc. Since this is inode 4, the inode can be found at block `SP_INODE_BLOCK + 4` which is block 6.

Here are the steps taken during `sp_new_inode()` when creating a directory:

```

} else if (S_ISDIR(mode)) {
    inode->i_blocks = 1;
    inode->i_op = &sp_dir_inops;
    inode->i_fop = &sp_dir_operations;
    inode->i_mapping->a_ops = &sp_aops;
    inode->i_size = 2 * SP_DIRENT_SIZE;

    spi->i_blocks = 1;
}

```

```

blk = sp_block_alloc(sb); /* XXX failure? */
spi->i_addr[0] = blk;
bh = sb_bread(sb, blk);
if (!bh) {
    /* XXX - need to handle failure */
    return 0;
}
memset(bh->b_data, 0, SP_BSIZE);
dirent = (struct sp_dirent *)bh->b_data;
dirent->d_ino = inum;
strcpy(dirent->d_name, ".");
dirent++;
dirent->d_ino = dip->i_ino;
strcpy(dirent->d_name, "..");

mark_buffer_dirty(bh);
brelse(bh);
}

```

In addition to setting fields in the Linux inode, we need to initialize the newly created SPFS inode. We call `sp_block_alloc()` to allocate the new directory block and then call `sb_bread()` for this block. After we have set the entries for `"."` and `".."` we mark the buffer dirty and release it so that it gets written to disk.

7.10.1 KGDB – Handling the `mkdir(2)` System Call

The previous section showed how SPFS provides support for creating new directory entries. To see `sp_mkdir()` being called, a breakpoint is set and `mkdir(1)` is run on the command line to create a new directory called `"new-dir"` in the root directory.

A breakpoint is set in `sp_readdir()` and then the `mkdir(1)` command is run. A partial stack backtrace is shown below after the breakpoint in `sp_mkdir()` is hit.

```

(gdb) bt
#0  sp_mkdir (mnt_userns=0xffffffff82a81240 <init_user_ns>,
    dip=0xfffff888103031100, dentry=0xfffff888102fe3840, mode=493)
    at /home/spate/spfs/kern/sp_dir.c:303
#1  vfs_mkdir (mnt_userns=0xffffffff82a81240 <init_user_ns>,
    dir=0xfffff888103031100, d
    entry=dentry@entry=0xfffff888102fe3840,
    mode=<optimized out>, mode@entry=493)
    at fs/namei.c:3979
#2  do_mkdirat (dfd=dfd@entry=-100, name=0xfffff88812eeeea000,
    mode=493, mode@entry=511) at fs/namei.c:4005
#3  __do_sys_mkdir (mode=<optimized out>,
    pathname=0x7ffcb1d3287d "new-dir") at fs/namei.c:4025
...

```

The `dip` argument specifies the directory in which the new directory is being created and `dentry` is for the newly created directory. The inode number of the parent (2) is printed out together with the name of the new directory.:

```
(gdb) p dip->i_ino
$38 = 2
(gdb) p dentry->d_name.name
$39 = (const unsigned char *) 0xfffff888102fe3878 "new-dir"
```

Below is the path from the `mkdir(2)` system call invoked by `mkdir(1)` to the system call handler to the filesystem entry point. The `do_mkdirat()` system call handler can be found in `fs/namei.c`. Search for `SYSCALL_DEFINE3(mkdirat)`.

```
mkdir(2) → do_mkdirat() →...→ sp_unlink()
```

For further information on the user-space implementation see TBD and for detailed handling on the VFS side of `mkdir(2)` see section TBD.

7.11 Reading Directory Entries

When we instantiate a directory inode we in `sp_read_inode()` we attach the following operations:

```
if (S_ISDIR(disk_ip->i_mode)) {
    inode->i_op = &sp_dir_inops;
    inode->i_fop = &sp_dir_operations;
```

of which the function used to read directory entries can be found here:

```
struct file_operations sp_dir_operations = {
    .read = generic_read_dir, XXXXXXXXXXXX
    .iterate_shared = sp_readdir,
    .fsync          = generic_file_fsync,
};
```

The `sp_readdir()` function is called as follows:

```
sp_readdir(struct file *f, struct dir_context *ctx)
{
    struct inode          *ip = file_inode(f);
    struct sp_inode_info  *dip = ITOSPI(ip);
```

We grab the Linux inode and from there we have access to the SPFS in-core inode. We may get called multiple times for a single directory. The `ctx->pos` field specifies the starting positions from where we should read directory entries. It will be 0 the first time called. On subsequent calls, it will be something different. This depends on the size of the user-space buffer into which we're copying data. For example, if we copy 64 entries and the buffer becomes full, SPFS will make sure that `ctx->pos` will be set to the position of entry number 65 for subsequent calls.

Why do we see a `struct file` as an argument and not an inode as with most other operations? Recall that inodes are shared across multiple processes each of which could be reading directory entries. **XXX—kind of. Could still use an inode since they pass ctx so could do this above the FS. Need more info.**

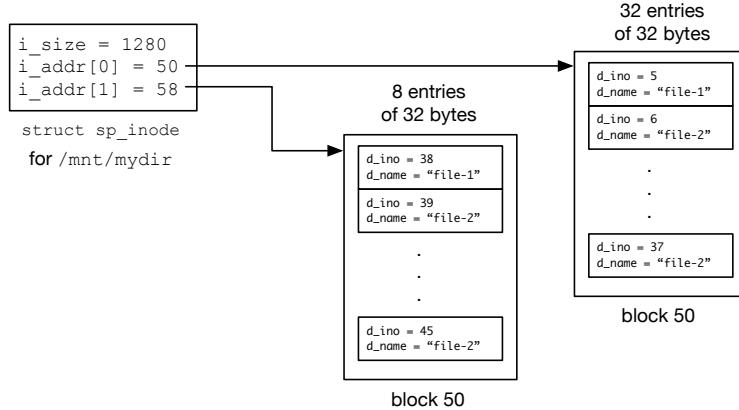


Figure 7.9: Reading from a directory with 40 entries

To continue our discussion of `sp_readdir()` works, consider figure 7.9. The directory we are reading from has 102 entries. We created 100 files in this directory and we also have `"."` and `".."`.

As we read through directory entries we may find empty slots. For example, suppose we create files A, B and C and then delete file B. We handle this in SPFS by simply setting the `i_ino` field of the `sp_dirent` structure to 0. Therefore, the process for reading directory entries is as follows.

- Walk through all entries from the requested directory.
- For each valid file (an inode without an inode number of 0), call the `dir_emit()`. If `dir_emit()` function to copy the inode number and filename to user space **(XXX—confirm)**. If a 0 is returned we continue reading more entries otherwise the user-space buffer so we must stop at this point.
- Increment `ctx->pos` to point to the next entry.

As we're walking through directory entries, we may need to read in multiple blocks from disk. Recall that `SP_BSIZE` is 4096 and each `struct sp_dirent` is 32 bytes so we have 64 directory entries per block.

7.11.1 KGDB – Kernel Paths for Reading Directories

1. small number of entries 2. much larger (100s) Fix the issue with `ctx->pos` before working on the larger example

The previous section showed how SPFS provides support for reading directory entries. To see `sp_readdir()` being called, a breakpoint is set and `ls(1)` is run on the command line to list the contents of two different directories, one with a small number of files and one with a much larger number.

A breakpoint is set in `sp_readdir()` and then the `ls(1)` command is run specifying one or other of the directories:

```
(gdb) ls mydir  
file-1 file-2 file-3  
(gdb) ls mydir-2
```

Unlike other examples in this chapter, here is the full backtrace when the breakpoint in `sp_readdir()` is hit. It shows early routines called prior to calling into routines in the Linux `fs` directory.

In `sp_readdir()` the `dip` argument

```
(gdb) bt  
#0 sp_readdir (f=0xffff888069f0e300, ctx=0xfffffc90001063e20)  
    at /home/spate/spfs/kern/sp_dir.c:130  
#1 0xffffffff81403a4b in iterate_dir  
    (file=file@entry=0xffff888069f0e300,  
     ctx=ctx@entry=0xfffffc90001063e20)  
    at fs/readdir.c:67  
#2 0xffffffff81404714 in __do_sys_getdents64 (count=32768,  
    dirent=<optimized out>, fd=<optimized out>)  
    at fs/readdir.c:369  
#3 __se_sys_getdents64 (count=32768, dirent=<optimized out>,  
    fd=<optimized out>) at fs/readdir.c:354  
#4 __x64_sys_getdents64 (regs=<optimized out>)  
    at fs/readdir.c:354  
#5 0xffffffff81eaal6b in do_syscall_x64 (nr=<optimized out>,  
    regs=0xfffffc90001063f58) at arch/x86/entry/common.c:50  
#6 do_syscall_64 (regs=0xfffffc90001063f58, nr=<optimized out>)  
    at arch/x86/entry/common.c:80  
#7 0xffffffff8200009b in entry_SYSCALL_64 ()  
    at arch/x86/entry/entry_64.S:120
```

The first argument specifies the directory that is being read as follows:

```
(gdb) p f->f_path.dentry  
$20 = (struct dentry *) 0xffff88806447e240  
(gdb) p $20->d_name.name  
$21 = (const unsigned char *) 0xffff88806447e278 "mydir"
```

The second argument (`ctx`) specifies the offset of the directory to start reading entries from in addition to a function that should be called by the filesystem for each entry in the directory to copy the data to user-space. This function is used indirectly by first calling `dir_emit()`. You can find the `filldir64()` function in `fs/readdir.c`.

```
(gdb) p *ctx  
$23 = {  
    actor = 0xffffffff81404350 <filldir64>,  
    pos = 0  
}
```

The breakpoint is actually hit twice. For the second time, we can see that `ctx->pos` now has a different value (192).

```
(gdb) p *ctx
$25 = {
    actor = 0xffffffff81404350 <filldir64>,
    pos = 192
}
```

We know that each SPFS directory entry is 32 bytes. In this directory there are 3 regular files in addition to "." and "..". Each time a directory entry is read, the offset advances by 32 bytes. **XXX – and this is where stuff is badly broken somewhere. It works fine but needs fixing**

File	Offset
.	0
..	32
file-1	64
file-2	96
file-3	128
next file	160

Below is the path from the `getdents64(2)` system call invoked by `ls(1)` to the system call handler to the filesystem entry point. The `getdents()` system call handler can be found in `fs/readdir.c`. Search for `SYSCALL_DEFINE3(getdents"`.

`getdents64(2) → do_unlinkat() → ... → sp_unlink()`

For further information on the user-space implementation see TBD and for detailed handling on the VFS side of `readdir(3)` see section TBD.

7.12 Writing to a File

To demonstrate how writing to a file works, let's copy a file which is under 3KB into an SPFS filesystem and use `fsdb` to show the newly allocated file:

```
$ ls -l lorem-ipsum
-rw-r--r-- 1 spate spate 2972 Nov 28 22:17 lorem-ipsum
$ cp lorem-ipsum /mnt
$ umount /mnt
$ fsdb /dev/sda
spfsdb > i2

inode number 2
i_mode      = 41ed
i_size      = 160
i_blocks    = 1
i_addr[ 0] = 33
...

```

```
Directory entries:  
    inum[ 2], name[.]  
    inum[ 2], name[...]  
    inum[ 3], name[lost+found]  
    inum[ 4], name[mydir]  
    inum[ 5], name[lorem-ipsum]  
  
spfsdb > i5  
  
inode number 5  
    i_mode      = 81a4  
    i_nlink     = 1  
    i_uid       = 0  
    i_gid       = 0  
    i_size      = 2972  
    i_blocks    = 3  
    i_addr[ 0] = 36    i_addr[ 1] = 37    i_addr[ 2] = 38
```

Since the block size of SPFS is 1024 bytes, the file needs 3 blocks to store the 2972 bytes of the file that is being copied. These are blocks 36, 27 and 38. Let's now look at SPFS calls that are made from the kernel to create the file and write the contents to the newly created file:

```
sp_lookup - for lorem-ipsum  
sp_find_entry - looking for lorem-ipsum (dip = 00000001d410682)  
sp_find_entry - failed for lorem-ipsum  
sp_lookup - lorem-ipsum not found  
In sp_create for lorem-ipsum  
sp_new_inode for lorem-ipsum - mode=100644  
sp_ialloc allocated inode 5  
In sp_diradd for lorem-ipsum (inum = 5)  
sp_create - inode 00000007b2520c4 created for lorem-ipsum  
sp_write_begin for inode=00000007b2520c4, off=0, len=2972  
sp_get_block (inode = 00000007b2520c4, block=0)  
sp_get_block - allocated blk=36  
sp_get_block (inode = 00000007b2520c4, block=1)  
sp_get_block - allocated blk=37  
sp_get_block (inode = 00000007b2520c4, block=2)  
sp_get_block - allocated blk=38  
sp_write_begin got page=00000000d5c0d094  
sp_write_end for inode=00000007b2520c4, off=0, len=2972,  
            page=00000000d5c0d094  
sp_writepage for page=00000000d5c0d094
```

The initial calls should now be familiar but we'll cover them again in addition to calls made to write the data:

- SPFS is called to lookup the file. Since this is a new file, there is no entry. This results in `sp_lookup()` calling `d_instantiate()` to instantiate a negative dentry.
- `sp_create()` is called to create the new file.

- The kernel then calls `sp_write_begin()` for this file which in turn simply calls `block_write_begin()` to handle the write for us **XXX-needs explaining**
- There are 3 calls to `sp_get_block()` to allocate new blocks for the file.
- Need to explain `sp_write_begin()` and `sp_write_end()` calls and what that does.

Since the length of the file is less than a single page (4 KB), we only see one call into SPFS to write data (the call to `sp_writepage()`). If the file that we're copying is greater than a page, we'll see multiple calls.

XXX—this explanation sucks. We need to explain it in much more detail or keep it simple here and reference a more detailed description somewhere else

7.12.1 KGDB – Handling the `write(2)` System Call

XXX – come back and add more context once page/buffer cache paths are better understood

This section shows the path followed by the kernel in response to creating the file and writing to it:

```
$ ls -l big-Lorem-ipsum
-rw-r--r-- 1 spate spate 5944 Apr 25 14:32 big-Lorem-ipsum
$ cp big-Lorem-ipsum /mnt
```

In the previous section, it was shown that writing to a file results in calls to the following SPFS functions after creating the file:

- `sp_write_begin()` – prepare for write.
- `sp_get_block()` – allocate blocks.
- `sp_write_end()` – unlock the page and increase the inode size.
- `sp_writepage()` – write the page to disk.

They were only called once each as the file being created was less than a page (4096 bytes). The filesystem's `write_begin()` function is called by the kernel to ask the filesystem to prepare to write the requested number of bytes at the given offset in the file. After a successful call to `write_begin`, and data has been copied into the allocated page, the filesystem's `write_end()` function must be called. The filesystem must unlock the page and release it, and also update `i_size`. SPFS relies on generic kernel functions to do this. Chapter discusses page/buffer cache and filesystem interactions in more detail.

Since this file is greater than a single page (4096 bytes) but also less than two pages, there will be two calls into each of these functions. Here is the stack backtrace for the first call to `sp_write_begin()`. You can see that the kernel is requesting to write 4096 bytes and an offset of 0.

```
(gdb) bt
#0  sp_write_begin (file=0xfffff88812ef25500,
    mapping=0xfffff888103556d78, pos=0, len=4096,
    pagep=0xfffffc900012a3ca8, fsdata=0xfffffc900012a3cb0)
    at /home/spate/spfs/kern/sp_file.c:75
#1  generic_perform_write (iocb=<optimized out>,
    i=<optimized out>) at mm/filemap.c:3779
#2  __generic_file_write_iter
    (iocb=iocb@entry=0xfffffc900012a3da8,
    from=from@entry=0xfffffc900012a3d80) at mm/filemap.c:3907
#3  generic_file_write_iter (iocb=0xfffffc900012a3da8,
    from=0xfffffc900012a3d80) at mm/filemap.c:3939
#4  call_write_iter (iter=0xfffffc900012a3d80,
    kio=0xfffffc900012a3da8, file=0xfffff88812ef25500)
    at ./include/linux/fs.h:2058
#5  new_sync_write (filp=filp@entry=0xfffff88812ef25500,
    buf=buf@entry=0x7f43bce1b000 "Lorem ipsum dolor sit amet,
    consectetur adipiscing elit. Pellentesque dignissim nisl
    quis arcu ... Class aptent taciti so"..., len=len@entry=5944,
    ppos=ppos@entry=0xfffffc900012a3e40) at fs/read_write.c:504
#6  vfs_write (file=file@entry=0xfffff88812ef25500,
    buf=buf@entry=0x7f43bce1b000 "Lorem ipsum dolor sit amet,
    nissim tempus lorem. Class aptent taciti so"...,,
    count=count@entry=5944, pos=pos@entry=0xfffffc900012a3e40)
    at fs/read_write.c:597
#7  ksys_write (fd=<optimized out>,
    buf=0x7f43bce1b000 "Lorem ipsum dolor sit amet, ...
    citi so"..., count=5944) at fs/read_write.c:650
```

Next, SPFS will see a call to `sp_write_end()` to information the filesystem that 4096 bytes have been copied into the page.

```
(gdb) bt
#0  sp_write_end (file=0xfffff88812ef25500,
    mapping=0xfffff888103556d78, pos=0, len=4096, copied=4096,
    page=0xfffffea0005277780, fsdata=0x0 <fixed_percpu_data>)
    at /home/spate/spfs/kern/sp_file.c:93
#1  generic_perform_write (iocb=<optimized out>,
    i=<optimized out>) at mm/filemap.c:3790
    ...
```

The process then repeats although this time SPFS is being informed about writing 1848 bytes (the remainder of the file) at an offset of 4096:

The process

```
(gdb) bt 2
#0  sp_write_begin (file=0xfffff8881023dc800,
    mapping=0xfffff8881033b9278, pos=4096, len=1848,
    pagep=0xfffffc900011cfcd8, fsdata=0xfffffc900011cfce0)
    at /home/spate/spfs/kern/sp_file.c:75
#1  generic\_\perform\_\write*] (iocb=<optimized out>,
```

```
i=<optimized out>) at mm/filemap.c:3779
```

A final call is then made to `sp_write_end()` informing the filesystem that 1848 bytes have been copied:

```
(gdb) bt 2
#0  sp_write_end (file=0xfffff8881023dc800,
    mapping=0xfffff8881033b9278, pos=4096, len=1848, copied=1848,
    page=0xfffffea0005143940, fsdata=0x0 <fixed_percpu_data>)
    at /home/spate/spfs/kern/sp_file.c:93
#1  generic_perform_write (iocb=<optimized out>,
    i=<optimized out>) at mm/filemap.c:3790
```

Depending on what activity there is at the time, the file copy may not be finished from the caller's perspective. I see a shell prompt return and a short while later, a breakpoint is hit in `sp_writepage()`. Note that there is a single page passed as an argument:

```
(gdb) bt
#0  sp_writepage (page=0xfffffea0005277780,
    wbc=0xfffffc9000118bc60)
    at /home/spate/spfs/kern/sp_file.c:105
#1  __writepage (page=page@entry=0xfffffea0005277780,
    wbc=wbc@entry=0xfffffc9000118bc60,
    data=data@entry=0xfffff888103556d78)
    at mm/page-writeback.c:2399
#2  write_cache_pages (mapping=mapping@entry=0xfffff888103556d78,
    wbc=wbc@entry=0xfffffc9000118bc60,
    writepage=writepage@entry=0xffffffff812f4680 <__writepage>,
    data=data@entry=0xfffff888103556d78)
    at mm/page-writeback.c:2334
#3  generic_writepages (wbc=0xfffffc9000118bc60,
    mapping=0xfffff888103556d78)
    at mm/page-writeback.c:2425
#4  do_writepages
    (mapping=mapping@entry=0xfffff888103556d78,
    wbc=wbc@entry=0xfffffc9000118bc60)
    at mm/page-writeback.c:2445
#5  __writeback_single_inode
    (inode=inode@entry=0xfffff888103556c00,
    wbc=wbc@entry=0xfffffc9000118bc60) at fs/fs-writeback.c:1587
#6  writeback_sb_inodes (sb=sb@entry=0xfffff888118af5000,
    wb=wb@entry=0xfffff888100ee0860,
    work=work@entry=0xfffffc9000118be08)
    at fs/fs-writeback.c:1870
#7  __writeback_inodes_wb (wb=wb@entry=0xfffff888100ee0860,
    work=work@entry=0xfffffc9000118be08)
    at fs/fs-writeback.c:1941
#8  wb_writeback (wb=wb@entry=0xfffff888100ee0860,
    work=work@entry=0xfffffc9000118be08)
    at fs/fs-writeback.c:2046
#9  wb_check_background_flush (wb=<optimized out>)
```

```
        at fs/fs-writeback.c:2112
#10 wb_do_writeback (wb=<optimized out>
    at fs/fs-writeback.c:2200
#11 wb_workfn (work=0xfffff888100ee09f0)
    at fs/fs-writeback.c:2227
#12 process_one_work (worker=worker@entry=0xfffff888119f2a180,
    work=0xfffff888100ee09f0) at kernel/workqueue.c:2289
#13 worker_thread (__worker=0xfffff888119f2a180)
    at kernel/workqueue.c:2436
#14 kthread (_create=0xfffff888127a279c0) at kernel/kthread.c:376
#15 ret_from_fork () at arch/x86/entry/entry_64.S:306
```

This will be followed soon after by another call to `sp_write_page()`:

```
(gdb) bt 2
#0 sp_writepage (page=0xfffffea00051ed180,
    wbc=0xfffffc900012abc60)
    at /home/spate/spfs/kern/sp_file.c:105
#1 __writepage (page=page@entry=0xfffffea00051ed180,
    wbc=wbc@entry=0xfffffc900012abc60,
    data=data@entry=0xfffff8881033b9278)
    at mm/page-writeback.c:2399
```

For each call, `sp_write_page()` makes a call to `block_write_full_page()` to do the job of actually writing the data to disk. For further details see section **TBD**.

7.13 Reading from a File

Let's start with a file that is a little over 8 KB which is shown in `fsdb` as follows. You can see that we need 8 blocks to store this file.

```
spfsdb > i6
inode number 6
  i_mode      = 81a4
  i_nlink     = 1
  i_atime     = Mon Nov 28 23:13:21 2022
  i_mtime     = Mon Nov 28 23:11:17 2022
  i_ctime     = Mon Nov 28 23:11:17 2022
  i_uid       = 0
  i_gid       = 0
  i_size      = 8916
  i_blocks    = 9
  i_addr[0]   = 48 i_addr[1] = 49 i_addr[2] = 50 i_addr[3] = 51
  i_addr[4]   = 52 i_addr[5] = 53 i_addr[6] = 54 i_addr[7] = 55
  i_addr[8]   = 56
```

To read the file we'll just issue a `cat /mnt/big-Lorem-ipsum` command to read the whole file. Here are the sequence of operations that we'll see the kernel make into SPFS to see if the file exists and then read it.

```

sp_lookup - for big-lorem-ipsum
sp_find_entry - looking for big-lorem-ipsum
(dip = 00000000cec55eef)
sp_find_entry - found inum 6 for big-lorem-ipsum
sp_read_inode for ino=6
sp_read_inode gets inode = 00000000d52b9004
sp_lookup name = big-lorem-ipsum, inode = 00000000d52b9004
(ino=6)
sp_read_folio
sp_get_block (inode = 00000000d52b9004, block=0)
sp_get_block - NOT create blk = 48
sp_get_block (inode = 00000000d52b9004, block=1)
sp_get_block - NOT create blk = 49
sp_get_block (inode = 00000000d52b9004, block=2)
sp_get_block - NOT create blk = 50
sp_get_block (inode = 00000000d52b9004, block=3)
sp_get_block - NOT create blk = 51
sp_read_folio
sp_get_block (inode = 00000000d52b9004, block=4)
sp_get_block - NOT create blk = 52
sp_get_block (inode = 00000000d52b9004, block=5)
sp_get_block - NOT create blk = 53
sp_get_block (inode = 00000000d52b9004, block=6)
sp_get_block - NOT create blk = 54
sp_get_block (inode = 00000000d52b9004, block=7)
sp_get_block - NOT create blk = 55
sp_read_folio
sp_get_block (inode = 00000000d52b9004, block=8)
sp_get_block - NOT create blk = 56

```

We actually do very little inside SPFS for reading files. The `read_folio` function is called and we simply invoke the generic kernel `block_read_full_folio()` function to do the work for us. All we need to provide is our `sp_get_block()` function which will return a block within the file given an offset.

```

sp_read_folio(struct file *file, struct folio *folio)
{
    printk("spfs: sp_read_folio\n");
    return block_read_full_folio(folio, sp_get_block);
}

```

As the kernel walks through the file, it makes calls into `sp_get_block()` to get a specific block number

```

sp_get_block(struct inode *inode, sector_t block,
            struct buffer_head *bh_result, int create)
{
    ...
} else { /* non-create path */
phys = spi->i_addr[block]; /* sector = BLKSZ */
printk("spfs: sp_get_block - NOT create blk = %ld\n", phys);

```

```
        map_bh(bh_result, sb, phys);  
    }
```

We are passed the block number and told this is a non-create call. Since we have the SPFS inode in-core, looking up the block given the block number is very straightforward (`spi->i_addr[block]`). The call to `map_bh()` is made to

```
static inline void  
map_bh(struct buffer_head *bh, struct super_block *sb,  
       sector_t block)  
{  
    set_buffer_mapped(bh);  
    bh->b_bdev = sb->s_bdev;  
    bh->b_blocknr = block;  
    bh->b_size = sb->s_blocksize;  
}
```

XXX—Need to find out what `set_buffer_mapped()` does. Also need to walk through what the function `block_read_full_folio()` does.

7.13.1 KGDB – Handling the `read(2)` System Call

This one will get an audible groan as you look through the stack backtrace. We start with our 2,972 byte (one page) file `lorem-ipsum` and simply run the `cat(1)` command on it to read all of the file's contents. Here is the long stack backtrace and, as per usual, omitting the architecture-specific system call handling portion. Note that if you are using `gdb` to analyze read paths, you will likely need to unmount/mount the filesystem between experiments since a subsequent read will just get a page cache hit and not enter the filesystem.

```
(gdb) bt  
#0  sp_read_folio  (file=0xfffff888116625500,  
                   folio=0xfffffea000598a4c0)  
    at /home/spate/spfs/kern/sp_file.c:112  
#1  0xffffffff812f976b in read_pages  
    (rac=rac@entry=0xfffffc90000e2fc38) at mm/readahead.c:178  
#2  0xffffffff812f99d2 in page_cache_ra_unbounded (  
    ractl=ractl@entry=0xfffffc90000e2fc38, nr_to_read=1,  
    lookahead_size=lookahead_size@entry=32)  
    at mm/readahead.c:263  
#3  0xffffffff812f9ece in do_page_cache_ra  
    (lookahead_size=<optimized out>,  
     nr_to_read=<optimized out>, ractl=0xfffffc90000e2fc38)  
    at mm/readahead.c:293  
#4  page_cache_ra_order (ractl=ractl@entry=0xfffffc90000e2fc38,  
    ra=ra@entry=0xfffff888116625598, new_order=<optimized out>,  
    new_order@entry=0) at mm/readahead.c:550  
#5  0xffffffff812fa275 in ondemand_readahead  
    (ractl=0xfffffc90000e2fc38, folio=folio@entry=0x0  
     <fixed_percpu_data>, req_size=<optimized out>)  
    at mm/readahead.c:672
```

```

#6 0xffffffff812fa59a in page_cache_sync_ra (
    ractl=ractl@entry=0xfffffc90000e2fc38,
    req_count=<optimized out>, req_count@entry=32)
    at mm/readahead.c:699
#7 0xffffffff812eacbb in page_cache_sync_readahead
    (req_count=32, index=0, file=0xfffff888116625500,
     ra=0xfffff888116625598, mapping=0xfffff888103344678)
    at ./include/linux/pagemap.h:1234
#8 filemap_get_pages (iocb=iocb@entry=0xfffffc90000e2fe30,
    iter=iter@entry=0xfffffc90000e2fe08,
    fbatch=fbatch@entry=0xfffffc90000e2fcf8) at mm/filemap.c:2594
#9 0xffffffff812eb284 in filemap_read
    (iocb=iocb@entry=0xfffffc90000e2fe30,
     iter=iter@entry=0xfffffc90000e2fe08, already_read=0)
    at mm/filemap.c:2688
#10 0xffffffff812eda24 in generic_file_read_iter
    (iocb=0xfffffc90000e2fe30, iter=0xfffffc90000e2fe08)
    at mm/filemap.c:2834
#11 0xffffffff813e5911 in call_read_iter
    (iter=0xfffffc90000e2fe08, kio=0xfffffc90000e2fe30,
     file=0xfffff888116625500) at ./include/linux/fs.h:2052
#12 new_sync_read (filp=filp@entry=0xfffff888116625500,
    buf=buf@entry=0x7f018c5df000 <error: Cannot access memory at
    address 0x7f018c5df000>, len=<optimized out>,
    ppos=ppos@entry=0xfffffc90000e2fed0) at fs/read_write.c:401
#13 0xffffffff813e83ba in vfs_read
    (file=file@entry=0xfffff888116625500,
     buf=buf@entry=0x7f018c5df000 <error: Cannot access memory
     at address 0x7f018c5df000>, count=count@entry=131072,
     pos=pos@entry=0xfffffc90000e2fed0) at fs/read_write.c:482
#14 0xffffffff813e8f23 in ksys_read (fd=<optimized out>,
    buf=0x7f018c5df000 <error: Cannot access memory at
    address 0x7f018c5df000>, count=131072)
    at fs/read_write.c:626
#15 0xffffffff813e8fc9 in __do_sys_read (count=<optimized out>,
    buf=<optimized out>, fd=<optimized out>)
    at fs/read_write.c:636

```

XXX – really need to go over the read paths and mm first. I can't see anything useful in the folio structure to display. Perhaps look at other filesystems to see if they break it down

Below is the path from the `read(2)` system call invoked by `cat(1)` to the system call handler to the filesystem entry point. The `ksys_read()` system call handler can be found in `fs/read_write.c`.

```
read(2) → ksys_read() →...→ sp_readfolio()
```

For further information on the user-space implementation see TBD and for detailed handling on the VFS side of `ksys_read(2)` see section TBD.

7.14 Memory Mapped Files

The filesystem is going to see similar calls from the kernel for memory mapped files as it does for reading and writing. The program from section 2.15 opens a file, maps it and walks through the address space one byte at a time calling `putchar(2)` to write the output to `stdout`.

Since we are essentially walking through the file byte by byte we will see calls into the filesystem requesting data from the start of the file and progressing throughout it. Thus, we see the following calls into SPFS where the kernel requests blocks in turn (0, 1, 2 and so on) and SPFS returns the corresponding physical block.

```
sp_read_folio
sp_get_block (block = 0) - not create, disk blk = 48
sp_get_block (block = 1) - not create, disk blk = 49
sp_get_block (block = 2) - not create, disk blk = 50
sp_get_block (block = 3) - not create, disk blk = 51
sp_read_folio
sp_get_block (block = 4) - not create, disk blk = 52
sp_get_block (block = 5) - not create, disk blk = 53
sp_get_block (block = 6) - not create, disk blk = 54
sp_get_block (block = 7) - not create, disk blk = 55
sp_read_folio
sp_get_block (block = 8) - NOT create, blk = 56
```

But what happens if we walk backwards through the file accessing the last byte of the file first and walking through the file to the first byte as the following modified program shows:

```
fd = open("/mnt/big-lorem-ipsum", O_RDONLY);
fstat(fd, &st);
addr = (char *)mmap(NULL, st.st_size, PROT_READ, MAP_SHARED,
                     fd, 0);
close(fd);

addr += st.st_size;
for (i=0 ; i <= st.st_size ; i++) {
    putchar(*addr);
    addr--;
}
```

We would expect blocks in the file to be accessed in reverse or close to. But this doesn't happen. Instead we see the same series of calls into SPFS. **XXX—need to answer why but need some bigger files and explore patterns further.**

7.15 Removing a File

The `sp_unlink()` function removes a file (regular file, symlink, **XXX?**).

```
int
sp_unlink(struct inode *dip, struct dentry *dentry)
```

```

{
    struct super_block      *sb = dip->i_sb;
    struct spfs_sb_info     *sbi = SBTOSPFSSB(sb);
    struct inode             *inode = dentry->d_inode;
    struct sp_inode_info     *spi = ITOSPI(inode);
    int                      blk, i, error = -ENOENT;
    ino_t                   inum = 0;

    inum = sp_find_entry(dip, (char *)dentry->d_name.name);
    if (!inum) {
        goto out;
    }
    printk("spfs: In sp_unlink for %s (inum = %ld)\n",
           (char *)dentry->d_name.name, inum);
    sp_dirdel(dip, (char *)dentry->d_name.name);

    dip->i_ctime = dip->i_mtime = current_time(dip);
    mark_inode_dirty(dip);

    inode->i_ctime = dip->i_ctime;
    inode_dec_link_count(inode);

    /*
     * Update the superblock summaries and free any blocks
     * that this file owned.
     */

    mutex_lock(&sbi->s_lock);
    sbi->s_inode[inode->i_ino] = SP_INODE_FREE;
    sbi->s_nifree++;
    sbi->s_nbfree += spi->i_blocks;
    for (i=0 ; i < spi->i_blocks ; i++) {
        blk = spi->i_addr[i] - SP_FIRST_DATA_BLOCK;
        sbi->s_block[blk] = SP_BLOCK_FREE;
    }
    mutex_unlock(&sbi->s_lock);
    error = 0;

out:
    return error;
}

```

XXX—change the above. unlink should not free resources. That will be done by sp_evict_inode()

One last point to consider is how to handle the directory inode `i_size` field. When a directory is created, `i_size` is set to `2 * SP_DIRENT_SIZE` which is 64 bytes. The directory will initially have a single block of 2048 bytes. If we keep adding directory entries, we increase `i_size` by 32 bytes. Additional blocks will get allocated as needed.

But what should happen when we remove a file?

figure of examples???? when to remove unused blocks? At end or in the middle?
defragmentation - must be doc on this.

7.15.1 KGDB – Handling the `unlink(2)` System Call

The previous section showed how SPFS unlinks a file. To see `sp_unlink()` being called, a breakpoint is set and `rm(1)` is run on the command line to remove a file. Below is the file hierarchy for the file `lorem-ipsum` which is being removed. You can see the inode numbers of the file (7) and the parent directory (4):

```
# ls -ld mydir
4 drwxr-xr-x 2 root root 96 Apr 25 15:08 mydir
# ls -li mydir/lorem-ipsum
7 -rw-r--r-- 1 root root 2972 Apr 25 20:15 mydir/lorem-ipsum
```

A breakpoint is set in `sp_unlink()` and then the `rm(1)` is run:

```
# rm mydir/lorem-ipsum
```

Here is a partial backtrace when the breakpoint in `sp_unlink()` is hit :

```
(gdb) bt
#0  sp_unlink(dip=0xfffff88810327df00, dentry=0xfffff888102e38300)
    at /home/spate/spfs/kern/sp_dir.c:506
#1  0xffffffff813f9986 in vfs_unlink (mnt_userns
    =<optimized out>, dir=0xfffff88810327df00,
    dentry=dentry@entry=0xfffff888102e38300,
    delegated_inode=delegated_inode@ \
    entry=0xfffffc90000e2fea0) at fs/namei.c:4196
#2  0xffffffff813ff2fe in do_unlinkat (dfd=dfd@entry=-100,
    name=0xfffff888100d77000) at fs/namei.c:4264
#3  0xffffffff813ff657 in __do_sys_unlinkat(flag=<optimized out>,
    pathname=<optimized out>, dfd=<optimized out>)
    at fs/namei.c:4307
...

```

In `sp_unlink()` the `dip` argument is the directory that contains the file to be removed which is referenced by the `dentry` argument. Let's check the inode number of the parent and both the name and inode number of the file to be removed.

```
(gdb) p dip->i_ino
$9 = 4
(gdb) p dentry->d_name.name
$13 = (const unsigned char *) 0xfffff888102e38338 "lorem-ipsum"
(gdb) p dentry->d_inode
$14 = (struct inode *) 0xfffff888103279780
(gdb) p $14->i_ino
$15 = 7
```

Below is the path from the `unlink(2)` system call invoked by `rm(1)` to the system call handler to the filesystem entry point. The `do_unlinkat()` system call handler can be found in `fs/namei.c`. Search for `SYSCALL_DEFINE3(unlinkat")`.

```
unlink(2) → do_unlinkat() → ... → sp_unlink()
```

For further information on the user-space implementation see TBD and for detailed handling on the VFS side of `unlink(2)` see section TBD.

7.16 Renaming a File

I thought that `rename` would be a difficult operation to implement, especially when adding a directory entry in the target. But since I already have a function to add a directory entry for file creation and remove a directory entry for file removal, the `sp_rename()` implementation was nothing more than a call to both functions as follows:

```
int
sp_rename(struct user_namespace *mnt_userns,
          struct inode *old_dir, struct dentry *old_dentry,
          struct inode *new_dir, struct dentry *new_dentry,
          unsigned int flags)
{
    struct inode     *inode = d_inode(old_dentry);
    int             error;

    printk("spfs: In sp_rename %s -> %s\n",
           old_dentry->d_name.name, new_dentry->d_name.name);

    error = sp_diradd(new_dir, new_dentry->d_name.name,
                      inode->i_ino);
    if (error == 0) {
        sp_dirdel(old_dir, (char *)old_dentry->d_name.name);
    }
    return error;
}
```

XXX—at some point need to cover locking. Are both inodes locked on entry?

7.16.1 KGDB – Handling the `rename(2)` System Call

The previous section showed how SPFS renames a file. To see `sp_rename()` being called, a breakpoint is set in `gdb` and `mv(1)` is run on the command line to move a file to a different directory and with a new name. Here is the file to be moved showing its inode number (4) and the directory which has an inode number of 5:

```
$ ls -li
4 -rw-r--r-- 1 root root 2972 Apr 27 22:18 lorem-ipsum
3 drwxr-xr-x 2 root root   64 Apr 27 17:03 lost+found/
5 drwxr-xr-x 1 root root   64 Apr 27 22:36 mydir/
```

A breakpoint is set in `sp_rename()` and then the `mv(1)` is run as follows:

```
# mv lorem-ipsum mydir/another-name
```

Here is a partial backtrace when the breakpoint in `sp_rename()` is hit :

```
(gdb) bt
#0  sp_rename  (mnt_userns=0xffffffff82a81240 <init_user_ns>,
    old_dir=0xfffff888103031100, old_dentry=0xfffff88810307e600,
    new_dir=0xfffff88810316a480, new_dentry=0xfffff888102fe3840,
    flags=1) at /home/spate/spfs/kern/sp_dir.c:113
#1  vfs_rename  (rd=rd@entry=0xfffffc90001263e20)
    at fs/namei.c:4723
#2  do_renameat2  (olddfd=olddfd@entry=-100,
    from=<optimized out>, newdfd=newdfd@entry=-100,
    to=to@entry=0xfffff88812eee8000, flags=flags@entry=1)
    at fs/namei.c:4874
#3  __do_sys_renameat2  (flags=1, newname=<optimized out>,
    newdfd=-100, oldname=0x7ffff8062872 "lorem-ipsum",
    olddfd=-100) at fs/namei.c:4907
...
...
```

In `sp_rename()`, the `old_dir` argument is the directory that contains `lorem-ipsum` which is referenced by the `old_dentry` argument. The `new_dir` argument is the directory to which the file is being moved to and the `new_dentry` argument is for the file in its new location. Let's check the inode numbers of both directories and both the names of the files being moved.

```
(gdb) p old_dir->i_ino
$33 = 2
(gdb) p new_dir->i_ino
$34 = 5
(gdb) p old_dentry->d_name.name
$35 = (const unsigned char *) 0xfffff88810307e638 "lorem-ipsum"
(gdb) p new_dentry->d_name.name
$36 = (const unsigned char *) 0xfffff888102fe3938 "another-name"
```

Below is the path from the `rename(2)` system call invoked by `rename(1)` to the system call handler to the filesystem entry point. The `do_renameat2()` system call handler can be found in `fs/namei.c`. Search for `SYSCALL_DEFINE5(renameat2)`.

```
rename(2) → do_renameat2() → ... → sp_rename()
```

For further information on the user-space implementation see TBD and for detailed handling on the VFS side of `rename(2)` see section TBD.

7.17 Other Operations

```

struct address_space_operations sp_aops = {
    .dirty_folio      = block_dirty_folio,
    .invalidate_folio = block_invalidate_folio,
    .read_folio       = sp_read_folio,
    .writepage        = sp_writepage,
    .write_begin      = sp_write_begin,
    .write_end        = sp_write_end,
    .bmap             = sp_bmap
};

struct inode_operations sp_file_inops = {
    .link   = sp_link,
    .unlink = sp_unlink,
};

struct file_operations sp_dir_operations = {
    .read      = generic_read_dir,
    .iterate_shared = sp_readdir,
    .fsync     = generic_file_fsync,
};

struct file_operations sp_file_operations = {
    .fsync      = generic_file_fsync,
    .llseek     = generic_file_llseek,
    .read_iter  = generic_file_read_iter,
    .write_iter = generic_file_write_iter,
    .mmap       = generic_file_mmap,
    .unlocked_ioctl = sp_ioctl
};

```

7.18 Obtaining Filesystem Information via statfs

The `df(1)` command displays information about each mounted filesystem. Here is the information displayed by `df` for a newly created and mounted SPFS filesystem:

```
# df -h /mnt
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda        440K   52K  388K  12% /mnt
```

Refer to figure 7.1 for how an SPFS filesystem is laid out.

You'll see in `spfs.h` that `SP_MAXBLOCKS` is 470 and `SP_BSIZE` is 1024 and thus the size of the filesystem is 440k. The first data block starts at block 50 and blocks 50 and 51 are used for directory entries for the root directory and `lost+found`.

The `df` command uses the `statfs(2)` system call to gather information for each mounted filesystem. This results in a call to `sp_statfs()` to get the necessary information for a specific mount. This function is very straightforward returning some defaults and some information about the filesystem that is kept in the in-core `sp_superblock` structure.

```
buf->f_type = SP_MAGIC;
buf->f_bsize = SP_BSIZE;
buf->f_blocks = SP_MAXBLOCKS;
buf->f_bfree = sbi->s_nbfree;
buf->f_bavail = sbi->s_nbfree;
buf->f_files = SP_MAXFILES;
buf->f_ffree = sbi->s_nifree;
buf->f_fsid = u64_to_fsid(huge_encode_dev(sb->s_bdev->bd_dev));
buf->f_namelen = SP_NAMELEN;
```

Note that `f_bfree` and `f_bavail` are both set to the number of free blocks available. SPFS does not draw a distinction between the two:

- `f_bfree`—Free blocks in filesystem
- `f_bavail`—Free blocks available to unprivileged user

XXX it would be good to see who actually sets these differently

More complex filesystems won't use as many defaults and the filesystem size will vary as well as the block size and the number of files available (potentially). But even with more complex filesystems, such information will be held in an in-core superblock or other such structure.

7.18.1 KGDB – Handling the `statfs(2)` System Call

The previous section showed how SPFS returns filesystem attributes/statistics. To see `sp_statfs()` being called, a breakpoint is set and `df(1)` is run on the command line. Here is a partial backtrace when the breakpoint is hit in `sp_statfs()`:

```
(gdb) bt
#0  sp_statfs  (dentry=0xfffff888104f563c0,
    buf=0xfffffc90000f17da0)
    at /home/spate/spfs/kern/sp_inode.c:273
#1  0xffffffff8143349d in statfs_by_dentry
    (dentry=0xfffff888104f563c0,
    buf=buf@entry=0xfffffc90000f17da0) at fs/statfs.c:66
#2  0xffffffff81433d3a in vfs_statfs  (buf=0xfffffc90000f17da0,
    path=0xfffffc90000f17d58) at fs/statfs.c:90
#3  user_statfs  (pathname=0x55630f2edf70 "/mnt",
    st=st@entry=0xfffffc90000f17da0) at fs/statfs.c:105
#4  0xffffffff81433ded in __do_sys_statfs
    (pathname=<optimized out>,
    buf=0x7fffc5850380) at fs/statfs.c:195
...

```

The first argument to `sp_statfs()` is the `dentry` for the root directory of the filesystem. From this `dentry` we can get its associated `inode` structure and make sure it's the SPFS root directory. It should have an inode number of 2 and its name should be "/".

```
(gdb) p dentry->d_name.name
$6 = (const unsigned char *) 0xffff888104f563f8 "/"
(gdb) p dentry->d_inode
$7 = (struct inode *) 0xffff888104d4e580
(gdb) p $7->i_ino
$8 = 2
```

Below is the path from the `statfs(2)` system call invoked by `df(1)` to system call handler to filesystem entry point. The `user_statfs()` system call handler can be found in `fs/statfs.c`. Search for `SYSCALL_DEFINE2(statfs"`

`statfs(2) → user_statfs() → ... → sp_statfs()`

For further information on the user-space implementation see **TBD** and for detailed handling on the VFS side of `statfs(2)` see section **TBD**.

7.19 Hard Links and Symbolic Links

Implementing support for symbolic links turned out to be much harder than I'd ever have thought with a lot of kernel panics and confusion on my part. I'll explain the approach I took to get this working then will further discuss the different options later (section XXX).

First I decided to keep it simple and store the symlink in a data block and read it in when requested. This is the old style way of storing symlinks since space inside the inode has generally been at a premium. With a disk-based inode where I'm storing disk blocks, I'd rather have more blocks, and therefore a larger file size, than space for a symlink since symlinks are used much less frequently. This method of storing symlinks within their own disk blocks is called "slow symlinks" due to the disk access.

But I soon ran into a problem trying to get it to work after `umount/mount`.

```
static const
struct inode_operations sysv_symlink_inode_operations = {
    .get_link    = page_get_link,
    .getattr     = sysv_getattr,
};
```

For symlinks read

- <https://www.halolinux.us/kernel-reference/lookup-of-symbolic-links.html>
- <https://www.kernel.org/doc/Documentation/filesystems/porting.rst>
- <https://unix.stackexchange.com/questions/147535/fast-and-slow-symlinks>

When creating a new inode for a symlink, a call is made as follows:

```
page_symlink(inode, symlink_target, slen);
```

This kernel function gets the address space operations we attached to the new inode and calls:

```
aops->write_begin(NULL, mapping, 0, len-1, &page, &fsdata);
memcpy(page_address(page), symname, len-1);
aops->write_end(NULL, mapping, 0, len-1, len-1, page, fsdata);
mark_inode_dirty(inode);
```

NOTE—you must hash an inode before making it dirty otherwise making it dirty later won’t work. I was calling `page_symlink()` before calling `insert_inode_hash()` and so the inode never gets marked dirty. This is done in `sp_new_inode()`. I set up each inode by type (reg,dir,lnk) and then call `mark_inode_dirty()` but symlinks are different due to the call to `page_symlink()` which internally calls `cfmark_inode_dirty()`.

7.20 Integration with /proc

I had thought about implementing support for ioctls to be able to fetch SPFS-specific information. For example, I could return current open files to help with debugging. But that’s a lot of work when there’s already a mechanism to be able to do this and that’s integration with the proc filesystem.

TBD

7.21 The SPFS `fsdb` Command

One tool that was instrumental in developing SPFS was a version of `fsdb` that could analyze and print various SPFS structures such as the superblock, directory entries and inodes. Also, the ability to display blocks of data in hexadecimal and ASCII formats. This allowed me to see when data structures and data were being written to disk, to make sure inodes and data blocks were allocated and deallocated correctly. If you want to take on some of the exercises at the end of this chapter, I strongly recommend that you make enhancements to `fsdb` to display any structures that you might be writing to disk. The program is less than 200 lines of code and very easy to enhance.

Here are a few commands to give you an idea of how `fsdb` works. First, we run `fsdb` on a partition where an SPFS filesystem resides. The `h` command displays commands available:

```
# ./fsdb /dev/sda
spfsdb > h
q - quit
i - display inode (e.g. "i2")
s - display superblock
r - read & display file contents (e.g. "r4")
u - undelete file (will prompt for inode)
```

The `s` command displays the superblock. We show inodes inuse/free but not blocks. That would be an easy option to add.

```
spfsdb > s
Superblock contents:
s_magic    = 0x53465053
```

```

s_mod      = SP_FSCLEAN
s_nifree   = 123
s_nbfree   = 627

i_inode[ 0] = SP_INODE_INUSE
i_inode[ 1] = SP_INODE_INUSE
i_inode[ 2] = SP_INODE_INUSE
i_inode[ 3] = SP_INODE_INUSE
...
i_inode[126] = SP_INODE_FREE
i_inode[127] = SP_INODE_FREE

```

Next the **i** command is used to display inode 2. Since this is a directory we show directory entries.

```

spfsdb > i2
inode number 2
    i_mode      = 41ed
    i_nlink     = 4
    i_atime     = Wed Dec  7 18:26:25 2022
    i_mtime     = Wed Dec  7 18:26:25 2022
    i_ctime     = Wed Dec  7 18:26:25 2022
    i_uid       = 0
    i_gid       = 0
    i_size      = 128
    i_blocks    = 1
    i_addr[ 0] = 129

Directory entries:
    inum[ 2], name[.]
    inum[ 2], name[..]
    inum[ 3], name[lost+found]
    inum[ 4], name[lorem-ipsum]

```

We can see one file (`lorem-ipsum`) so we display the contents of inode 4:

```

spfsdb > i4
inode number 4
    i_mode      = 81a4
    i_nlink     = 1
    i_atime     = Wed Dec  7 18:26:50 2022
    i_mtime     = Wed Dec  7 18:26:50 2022
    i_ctime     = Wed Dec  7 18:26:50 2022
    i_uid       = 0
    i_gid       = 0
    i_size      = 2972
    i_blocks    = 2
    i_addr[ 0] = 131    i_addr[ 1] = 132

```

The file fits in 2 blocks (131 and 132) so we dump out the contents of each block. If a character is non-printable a "." is displayed.

```
spfsdb > b131
0000 4c6f 7265 6d20 6970 7375 6d20 646f 6c6f    Lorem ipsum dolo
0020 6563 7465 7475 7220 6164 6970 6973 6369    ectetur adipisci
0040 6573 7175 6520 6469 676e 6973 7369 6d20    esque dignissim
...
07e0 2076 656c 2c20 736f 6c6c 6963 6974 7564    vel, sollicitud
spfsdb > b132
0000 2050 7261 6573 656e 7420 6174 2074 656c    Praesent at tel
0020 6f6e 7661 6c6c 6973 206e 756e 6320 7365    onvallis nunc se
0040 6e61 2e20 4165 6e65 616e 2074 656d 706f    na. Aenean tempo
...
07c0 0000 0000 0000 0000 0000 0000 0000 0000    .....
07e0 0000 0000 0000 0000 0000 0000 0000 0000    .....
```

7.22 File Undelete

This is a fun topic to talk about. In SPFS, we've added a file undelete option which was easy it is to implement given the simplicity of this filesystem (it only took a couple of hours to implement and longer to write about). We're also going talk about why it gets more complicated in more sophisticated filesystems becomes when journaling capabilities to avoid the all-time-consuming full `fsck`.

We've implemented undelete as part of the SPFS `fsdb` command. We're also going to *cheat* by making a lot of assumptions that we will discuss. But first, let's take a look at a regular file and see what structures it impacts. Figure 7.10 shows a filesystem that has a file in the root filesystem called `lorem-ipsum`. This regular file is stored in the root filesystem and has 2 data blocks. The superblock fields for the number of free inodes (`s_ifree`) and free blocks (`s_nbfree`) were decremented when the file was created. The superblock field for the inode allocated for `lorem-ipsum` (`s_inode[4]`) was set to `SP_INODE_INUSE` and the superblock field for the two blocks allocated (`s_block`) were updated to show that they're in use. (`SP_BLOCK_INUSE`).

The reason we undelete is easy to implement is that we perform the most basic set of operations when removing the file only changing those fields in figure 7.10 that are shown in bold.

1. Mark the inode as free in the superblock (`s_inode[4]`).
2. For each allocated block in the deleted file, mark the block as free in the superblock `s_block`).
3. Update superblock counts (increment number of available inodes (`s_nifree`) and decrement number of blocks available by 2 `s_nbfree`).
4. Remove the directory entry for `lorem-ipsum` from block 29 (from the root directory).

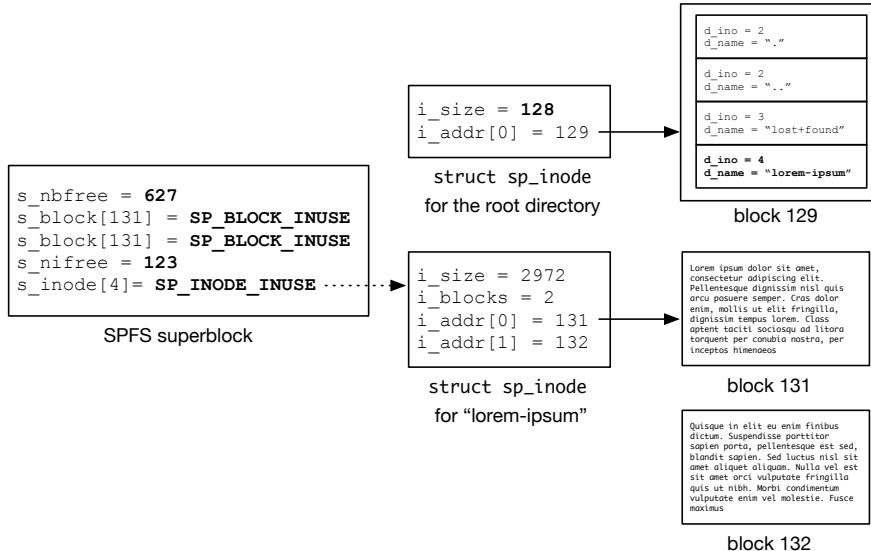


Figure 7.10: Changes needed to implement file undelete

The contents of the inode are left unmodified. This includes the file size, mode, file ownership and the allocated blocks. If we were to zero the fields of the inode before deleting the file then undelete would not be possible unless this information was recorded elsewhere.

File delete won't always work. Here is the list of restrictions:

- Generally speaking, the undelete operation needs to be performed as soon as possible after a file gets deleted. This is because the inode for the deleted file may be reallocated or the data blocks owned by the deleted file may have been reused for another file.
- The user needs to know the node number for the file.
- Only regular files can be deleted.

How does someone get the inode number? When a file is removed, we display a message which is visible by running `dmesg(1)`:

```
spfs: In sp_unlink for lorem-ipsum (inum = 4)
```

Note however, these messages are not visible by regular users and if this were a production filesystem we wouldn't display this information at all. Therefore to make the undelete operation more user friendly, we would need a way to make this information available to regular users. One option would be to search the list of directory entries by name. This won't work "as is" since currently in `sp_dirdel()`, we delete a directory entry as follows:

```
dirent->d_ino = 0;
dirent->d_name[0] = '\0';
mark_buffer_dirty(bh);
```

So we don't overwrite `d_name` but we wouldn't see the filename there if we searched through possible old directory entries. We don't actually need to change the `d_name` field since having `d_ino` set to 0, we would know that the entry is free. Thus, it would be possible, given a directory, to search for all possible entries (`d_ino == 0`) and display possible names. We'll leave that an exercise to the reader if you choose to accept this mission.

7.22.1 Enhancing Undelete

Here are 3 possible options to consider for improving our undelete operation:

1. Print the filename / inode number somewhere so that regular users can access them. Writing to files from a kernel module is frowned upon. Perhaps look at using sysfs.
2. Perhaps don't allocate recently deleted inodes to new files. This wouldn't completely fix the issue but we could maintain a pointer that references the next inode to be allocated (initially it would start at 4 (just after `lost+found`)).
3. Search by name as opposed to inode number. A possible approach was described above.

Of course in a production system, the answer would be to maintain regular backups. We cover backups in section 3.17.

7.22.2 Why Undelete is More Complex Than you Might Think?

This section highlights cases where undelete can get very complicated if not impossible.

Security conscious organizations don't wish to leave data lying around on disk even if it's no longer used. Commercial filesystems such as VERITAS VxFS had options for handling this decades ago based on mount options. Deleted data blocks were zeroed before being freed. Directory entries were zeroed as part of file deletion.

In our simple undelete operation, we already stated that time is a concern. Data blocks could get reused. Inodes can be reallocated. What happens if a data block were reused and then the newer file was deleted? We could end up with the reinstated file having data that belonged to another user. This is actually very easy to demonstrate with our simple filesystem. Perhaps a partially restored file could be useful but there are risks for sure.

The way that SPFS deletes files is not the way a production filesystem would handle delete. We would actually start by removing data blocks (last block first). This way we at least retain some structural integrity. We can mark the inode as being in process of being removed. That way, `fsck` knows to continue cleaning up if the system crashes before a deletion completes. This was how we handled file deletions in VxFS which was one of the earliest transaction-based filesystems. We had the concept of "extended operations". File deletion was one such operation. Although a file can be deleted, it can remain in

use for some time after the deletion completes. Only when the last reference to the file goes away can the file be deleted. Thus, this operation was marked on disk so that it could be completed during log replay (think fast version of `fsck`) if the system crashes. Furthermore, when removing a file, the removal is broken up into a series of operations which are logged prior to the actual filesystem structures being modified. As highlighted above, blocks will be removed first resulting in block / extent maps need to be updated and this could be more than a simple operations as it is in SPFS. In all my time at VERITAS, I only saw one occasion when a filesystem engineer managed to undelete a file.

And at VERITAS, we had multiple backup products which did the job of "undelete" very well indeed.

7.23 How to Test SPFS

As I added more capabilities to SPFS, I did the usual hand testing to the point of where things seemed to work. In other words, simple unit testing and also best path testing. Then as I continued to write other sections in the book, I tried different operations and as expected, found a lot of bugs or missing functionality altogether. For example, it wasn't until I was playing with the `wipe(1)` command that I realize that I hadn't implemented support for file rename.

But hand testing is only going to reveal so many issues. A real filesystem test suite is needed to really understand how stable the filesystem is. **XXX**.

The `fstest` test suite is a Posix conformance test suite for filesystems, originally written by Paweł Jakub Dawidek, well known for his work on ZFS for FreeBSD. I set myself a goal of getting this testsuite to run on SPFS and fix as many bugs as possible before book launch.

You can find `fstest` here:



URL 21 – <https://tinyurl.com/2b66w58a>

I also have a copy that's with the SPFS source code – **XXX** to try and make sure it runs / make mods etc etc. Take a look at this version. There is an `SPFS.readme` at the top level to show changes.

7.23.1 Testing a Commercial Filesystem

When I was at VERITAS working on VxFS, we had an extensive test suite that was built by engineering and for many years, we had no QA team. Engineers were expected to do all testing and obviously fix any issues found. This didn't scale forever and we found that a dedicated QA team was instrumental especially as the number of platforms and versions increased.

The test suite was divided into three main parts as follows:

- Conformance – a very large array of tests whereby different operations were tried and tests would *expect* specific results. A simple example could be to create a file with specific attributes, then run `stat(2)` and compare the results returned with expected results. But these tests went far beyond POSIX-style functions as VxFS had a lot of features that were filesystem specific.
- Stress testing – just hammer the filesystem with lots of I/Os of different sizes. It could be simply utilizing a single process/thread or multi-threaded.
- Noise testing – another set of tests which would run and have the rug pulled out of the filesystem at random times to emulate a system crash to make sure that log replay would complete successfully. Noise and stress testing at the same time was a particular good way to test.

I remember one particular test suite we got from one of our OEM partners that panicked the kernel/filesystem many times. It was one of the worst pieces of code I'd ever seen and actually had a comment in the test suite that said that it was the author's first ever program in C. Therefore, he was doing everything you weren't supposed to. As a consequence, it turned out to be very useful!

We also had random testing whereby one of our QA engineers was instructed to do whatever he/she wanted to break the filesystem. That also worked very well and could be any combination of anything. Perhaps stress+noise while taking snapshots/clones or any other combination that engineers wouldn't typically think of.

7.24 An Extended SPFS Filesystem

Hopefully at this point, you have a good idea of how a simple filesystem works on Linux. Exploring other filesystems in detail is a great next step and that will be the goal of volume 2 of this series of books. Another path is to take the base SPFS filesystem and enhance it to add more features, remove some of the limits and explore what is possible. Here are the list of things that you could work on next. I will be doing the same once I've finished writing this book. I'll be adding blog posts about each enhancement and will make all code available on github. So either head over to my website or try to make these enhancements yourself and see how your results compare with mine. I'd also love to hear what other features you'd like to see so send me email.

A list of topics that we'll add EPFS:

- Multiple block sizes. This shouldn't be fairly straightforward assuming that all code (including the SPFS `mkfs` and `fsdb` commands utilize constants in `spfs.h` correctly).
- Remove the superblock limits so there is either no limits on the number of inodes or blocks in the filesystem or we at least remove the basic limits that are there today.
- Extended attributes. A simple approach would be to use space in the inode for one or more attributes while understanding how the `xattr` interfaces work. Perhaps a list of external inodes in the inode where the `xattrs` reside. Obviously that uses more space in the inode.

- A use of direct, indirect, double and triple indirect blocks allowing for much larger file sizes.
- Use of bitmaps for free/used inode/block lists.
- Quotas. Not a trivial exercise by any means. **XXX—need to look at APIs**
- A `fsck` command to repair a damaged filesystem.
- Better logging. **XXX**
- ioctls. **XXX**
- Silent argument to `mount_super???`
- superblock dirty
- SMP issues?
- get familiar with `fsdb` and add code to view new stuff

Come back to this.

7.25 Notes

`MODULE_ALIAS_FS` `MODULE_ALIAS_FS` is defined un `linux/fs.h` as follows:

```
#define MODULE_ALIAS_FS(NAME) MODULE_ALIAS("fs-" NAME)
```

Seems like the goal is to load the module automatically when you do “`mount -t spfs`”. So how does that work?

7.26 debugfs?

`XXX`

7.27 Conclusion

Previous chapters covered the path from applications using libraries and invoking system calls and going through the different kernel layers including system call handling routines and through the VFS layer before making a call into the filesystem. This chapter presented a simple but fully functional disk-based filesystem called SPFS which implements most of the VFS to filesystem functions or requests that generic kernel support functions be called.

By showing examples of `gdb` stack backtraces for each of the major operations, it’s possible to see the full path through the kernel into the filesystem. Exact paths can sometimes differ depending on open flags and the method of I/O being performed.

An undelete option was shown which is easy for a simple filesystem such as SPFS to implement but more complex in today’s more feature rich filesystems.

Finally, testing SPFS was discussed showing how to develop test suites that can support heavy-duty commercial filesystems.

Developing a kernel-based filesystem is not trivial especially when adding enterprise capabilities such as journaling, snapshots, extended attributes and ensuring that it is highly performant. This chapter showed that it's definitely possible to build a filesystem without a huge amount of code so extending it to make it fit simple use cases, is not too difficult. But if performance is not an issue, it is possible to develop a filesystem in user-space which makes the process considerably easier. The next chapter will show how this can be done using the FUSE framework.

Chapter 8

The FUSE Filesystem Framework

FUSE (Filesystem in User SpacE) is a framework that allows filesystems to run in user space and has been available in the mainline kernel since 2.6.14. There are dozens of different FUSE-based filesystems. I recommend the FUSE Wikipedia page as a good starting point to see what types of filesystems have been developed using FUSE.

Since FUSE has been around for many years now, I was surprised to see how little good documentation there was about the internals. Hopefully this chapter will rectify that.

As well as exploring the internals of the user-space and kernel components of FUSE, I'll be showing you how to develop two FUSE-based filesystem. The first pSPFS, is a simple passthrough filesystem that allows you to view the contents of a specified directory through specified mount point. The second filesystem, eSPFS, extends SPFS by providing encryption of regular files. You can download both from github as follows:

`XXX--github.com/spate/XXX*`

Later in the chapter performance of FUSE will be covered, showing how FUSE has improved over the years. It will also cover some promising technologies including use of eBPF which offers additional performance improvements.

8.1 A Background on FUSE

FUSE system was originally part of AVFS (A Virtual Filesystem), a filesystem implementation heavily influenced by the translator concept of the GNU Hurd. Hurd itself is a GNU project that started in 1990 with the goal to provide a replacement for the UNIX kernel and consists of a microkernel and 24 servers providing UNIX-like functionality. Without the arrival of Linux we may have seen Hurd achieving its goal of replacing UNIX although Linux has firmly taken that place. Now, Hurd is still not ready for production

AVFS started in 2003 provides the ability to *look inside archived or compressed files, or access remote files without recompiling the programs or changing the kernel*. It supports floppy disks (if anyone still uses them), tar, gzip files, zip, bzip2, ar and rar files, ftp sessions, http, webdav, rsh/rcp, ssh/scp. AVFS also supports FUSE so would be an interesting project to experiment with.

The source code is here.



URL 22 – <https://avf.sourceforge.net>

In addition to Linux, FUSE is available for FreeBSD, OpenBSD, NetBSD (as `pufffs`), OpenSolaris, Minix 3, MacOS, and Windows although some of these versions differ considerably both in terms of implementation as well as support.

Linux FUSE comprises a kernel-based filesystem of approximately 16,000 lines of kernel code and `libfuse`, a user-space library, which is almost 20,000 lines of code. There are also several example filesystems on github including a passthrough filesystem that mirrors the contents of the root directory under the mount point. The pSPFS filesystem which will be described later in this chapter is similar but can mirror the contents of any directory.

The official FUSE github page can be found at:



URL 23 – <https://tinyurl.com/bdetb7th>

Generally speaking, a FUSE file system is implemented as a standalone application that links with the `libfuse` library. The example filesystems follow this method. We show how to both run the filesystem in the foreground (which helps with debugging) and in the background which would be used for production. You can also find an example of the SSHFS FUSE filesystem that runs in the background in section 3.10.1.

The `libfuse` library provides functions to mount the file system, unmount it, read requests from the kernel, and send responses back. There are two APIs. In both cases, incoming requests from the kernel are passed to the main program using callbacks.

The high-level API that is primarily specified in `fuse.h`. The low-level API that is primarily documented in `fuse_lowlevel.h`. Both of these methods will be described with appropriate examples.

On my Ubuntu VM there is a directory `/usr/share/doc/libfuse-dev` that contains useful information about the high-level and low-level APIs.



The kernel FUSE filesystem code can be found in `fs/fuse`. There is also `fuse.h` under `uapi/linux`

One concern about FUSE is that `libfuse` currently has no active, regular contributors and only high-impact issues are being resolved. This is of great concern to the Linux community at large and sadly speaks to the nature of many open-source projects. People have limited bandwidth and can't be expected to develop a project for years on end.

8.2 Source Code Files

There are several header files in different places. First are those header files that are part of `libfuse` that are installed in `usr/include/fuse` after installing `libfuse-devel`. In the `libfuse` sources you can find them in the `include` directory. The main header file to be used by user-space filesystems is `fuse.h`. Note that this differs from the kernel `fuse.h` described below.

In the kernel sources there are two additional header files:

- `fs/fuse/fuse_i.h` – contains C structures, function declarations and macro definitions used internally by FUSE kernel components.
- `include/uapi/linux/fuse.h` – This file defines the kernel interface of FUSE, for use by the kernel and the `libfuse` library.

XXX

8.2.1 Manual Pages

The FUSE manpages contain some information about the FUSE protocol but are mainly useful for understanding mount options and configuration options.

- `fuse(4)` – contains a description of FUSE, information about messages sent between the kernel and `libfuse` and a description of some but not all of the messages.
- `fusermount(1)` / `fusermount3(1)` – this command mounts and unmounts FUSE filesystems.
- `mount.fuse(8)` / `mount.fuse3(8)` – configuration and mount options for FUSE file systems.
- `grub-mount(1)` – this command issues a read-only mount of any filesystem or filesystem image that GRUB understands utilizing FUSE. For more information run `info grub-mount`.
- `ulockmgr_server(1)` – a user-space lock manager server for FUSE filesystems. A very small piece of code with next to no visible documentation as to how it's used.

The chapter won't discuss `grub-mount(1)` and `ulockmgr_server(1)` further but analysis may be of interest to some readers.

8.2.2 Understanding FUSE_VERSION

There is one thing that can lead to confusion when compiling FUSE filesystems and it's the FUSE version defined by `FUSE_USE_VERSION`.

For example, in the header file `/usr/include/fuse_lowlevel.h` you will see a comment as follows:

```
* Low level API
*
* IMPORTANT: you should define FUSE_USE_VERSION before
* including this header. To use the newest API define
* it to 26 (recommended for any new application), to
* use the old API define it to 24 (default) or 25
```

Here is what I see on Ubuntu 22.04 and Ubuntu 22.10:

```
$ grep '#define FUSE_USE_VERSION' /usr/include/fuse/*h
cuse_lowlevel.h:#define FUSE_USE_VERSION 29
fuse.h:#define FUSE_USE_VERSION 21
fuse_lowlevel.h:#define FUSE_USE_VERSION 24
```

But I downloaded the FUSE library (libfuse) from gitlab and failed trying to compile one of the example programs. Inside `example/notify_inval_inode.c` they define `FUSE_USE_VERSION` as follows:

```
#define FUSE_USE_VERSION 34
```

therefore the libfuse version on gitlab is much further ahead than the version I have installed. Furthermore, the user header files will match the implementation in the kernel so to compile one of the examples from libfuse you will need a version to match.

Older versions of libfuse can be found here:



URL 24 – <https://tinyurl.com/2p8neya4>

The trick is knowing which version to download in order to get a match with what's installed on your system. This is where the `fusermount(1)` command can help:

```
$ fusermount -v
fusermount3 version: 3.11.0
$ ls -l /usr/bin/fusermount
lrwxrwxrwx 1 root root 11 May  7  2022 /usr/bin/fusermount -> \
fusermount3
```

Therefore you should download libfuse version 3.11.0. Note that `fusermount3` superseded `fusermount` which is now a symlink to `fusermount3`.

XXX—but that didn't work. I downloaded it and still see `FUSE_USE_VERSION` defined as 33 in `example/notify_inval_inode.c`

When I run spfs, I see the following:

```
$ .mountit mnt target
FUSE library version: 2.9.9
```

The version I am using is 2.9.9 apparently. I downloaded 2.9.7 but don't see 2.9.9 on gitlab. But the examples that deal with low-level interfaces I want to show are not there! I can see `hello_ll.c` which does compile on my system:

```
$ gcc -Wall hello_ll.c `pkg-config fuse -cflags -libs` -o hello
```

Once again, I'm too far ahead of the curve. **XXX—looks like I will have to come back to this next Ubuntu version and see how much things have changed or just talk about hello_ll.c**

8.3 The FUSE Architecture

FUSE consists of multiple kernel components and a FUSE daemon in user space. The daemon consists of the `libfuse` library linked with a user-space filesystem that registers a list of callback functions with the library. The kernel components are linked together as a Linux kernel module called `fuse.ko`. There are three file system types inside the `fuse.ko` module all of which are visible in `/proc/filesystems`:

```
$ grep fuse /proc/filesystems
      fuseblk
nodev    fuse
nodev    fusectl
```

Both `fuse` and `fuseblk` are pseudo filesystems for which the underlying filesystems are provided by different FUSE daemons, also called `libfuse` daemons since the `libfuse` library is a major part of the implementation. Filesystems that utilize the `fuse` type do not require an underlying block device and are typically RAM-based, stackable, or network-based file systems. The `fuseblk` filesystem type in contrast, is for user-space file systems that are stored on block devices similar to local file systems.

XXX—odd but there is no "fuse" module so what is it? For a built kernel there is `cuse.ko` and `virtiofs.ko` in the `fs/fuse` directory but these aren't loaded (not visible running `lsmod`)

The `fusectl` file system provides users with the means to control and monitor any FUSE file system behavior (e.g., setting thresholds and counting the number of pending requests). This will be described further in section 8.4.8.

```
$ mount | grep fusectl
fusectl on /sys/fs/fuse/connections type fusectl (rw,nosuid,...)
```

To understand how FUSE works at a high-level consider figure 8.1. In both the high-level and low-level APIs, incoming requests from the kernel are passed to the `libfuse` daemon filesystem using *callbacks*. When using the high-level API, the callbacks work with file names and pathnames and processing of a request finishes when the callback function returns. When using the low-level API, the callbacks work with inodes and responses must be sent to the kernel explicitly using a separate set of API functions.

Here is the path followed when interacting with a `fuse`-based filesystem:

1. A processes issue a file operation to the FUSE mount point (the source). It enters the `fuse` filesystem.
2. A message is constructed containing all the arguments needed to process the request. It is queued and any threads waiting for requests from this filesystem are woken up.

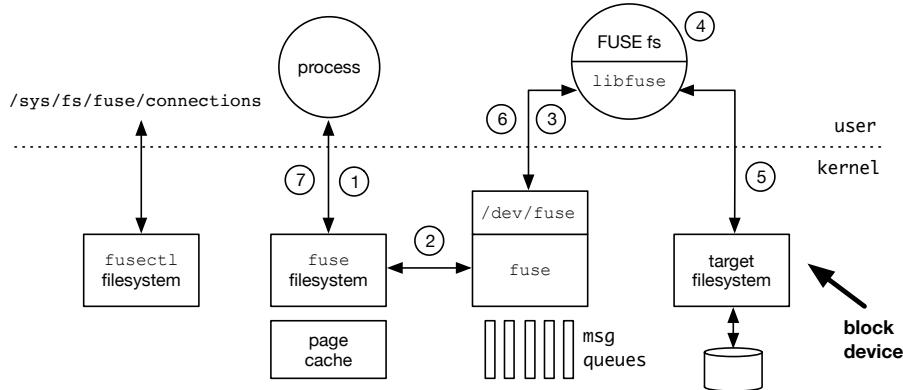


Figure 8.1: The FUSE Filesystem Architecture

3. The thread that received the request returns to user space and prepares the request. For example, it may need to construct the full pathname of the file being operated on.
4. libfuse calls the appropriate FUSE operation exported by the FUSE daemon. The filesystem processes the request. For example, in the eSPFS FUSE filesystem described later, this may involve encrypting the data received from the kernel to be written to the file.
5. The FUSE daemon makes file-related system calls into the target filesystem. For example, if a read request is being processed, the FUSE filesystem may make a `read(2)` system call to read the data for the file. Or in the case of eSPFS, the data encrypted in phase 3 will be written to the target filesystem using `write(2)`.
6. libfuse replies to the request passing back any data and / or error codes.
7. The original system call completes by the `fuse` filesystem responding to the calling application.

When a FUSE filesystem is mounted the libfuse daemon establishes connection with the kernel by opening `/dev/fuse`. Here is the device and the pSPFS filesystem, that will be described later in the chapter, is shown to be accessing the device.

```
$ ls -l /dev/fuse
crw-rw-rw- 1 root root 10, 229 Feb  9 12:08 /dev/fuse
$ fuser /dev/fuse
/dev/fuse:          34006
$ ps -ef | grep 34006
spate      34006  34005  0 15:53 pts/0    00:00:00 spfs \
           -omodules=subdir,subdir=/home/spate/target -d -s -f mnt
```

Furthermore, information about a specific mount can be found by looking under `sys/fs`:

```
$ ls /sys/fs/fuse/connections/
48/
$ ls /sys/fs/fuse/connections/48
abort congestion_threshold max_background waiting
```

This information is provided by the `fusectl` filesystem, also shown in figure 8.1, and will be described in section 8.4.8.

Performance issues will be addressed later but it's worthy of mention to say that since the `fuse` filesystem operates just like any other Linux filesystem, in that it benefits from the page cache such that any data read from any file will be cached and this can reduce the lengthy path from to the target to filesystem.

8.3.1 An Example to Get Started

Before too digging into the different functions and structures that make up FUSE, here is an example to help paint a picture. Figure 8.2 shows the structures involved in processing the `unlink(2)` system call. Details of the sequence also showing how the `libfuse` daemon interacts with the kernel is described in .

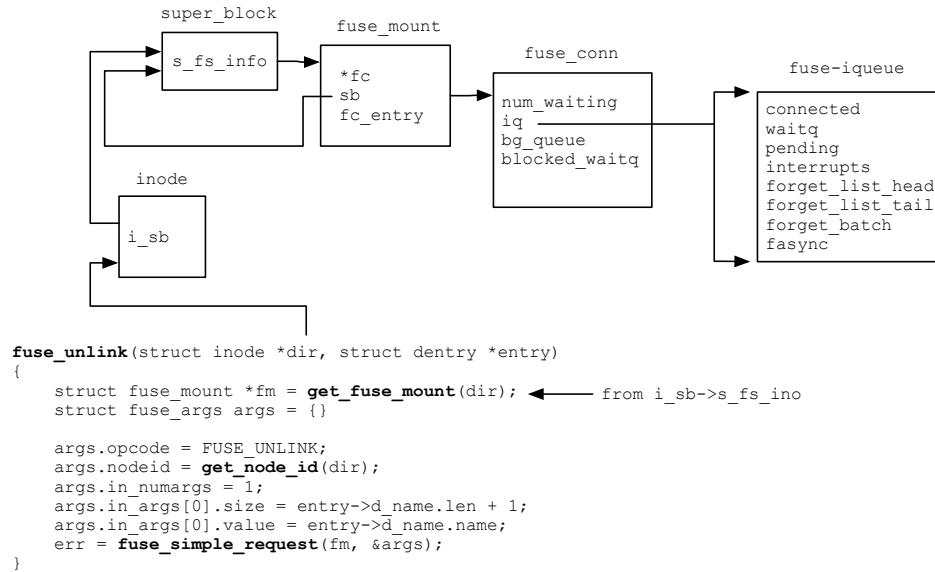


Figure 8.2: Structures accessed during most FUSE file operations

The `unlink` request enters the `fuse` filesystem through `fuse_unlink()` which creates a `FUSE_UNLINK` request message and calls `fuse_simple_request()` to add the message to the list of pending messages. This queue is referenced by the `pending` field of the `fuse_iqueue` structure. This structure is accessed through the list of struc-

tures shown in the figure. The following sections will described these structures in more detail.

The libfuse daemon will read from `/dev/fuse` and should be waiting for messages from the kernel unless it is already processing existing messages. Once a message is posted, the read will complete and the daemon will process the request by calling into the user-space filesystem through the supplied operations vector.

8.4 Digging Under The Covers

There are just over 16,000 LOC in the FUSE kernel components split across the following source files in `/home/spate/linux-5.19.17/fs/fuse`:

```
$ ls
Kconfig      control.c      dev.c       fuse_i.h    readdir.c
Makefile     cuse.c        dir.c       inode.c    virtio_fs.c
acl.c        dax.c        file.c     ioctl.c    xattr.c
```

and one header file `include/linux/uapi/linux/fuse.h` which defines the kernel interface of FUSE. Overall, FUSE doesn't have a huge code base but like many Linux kernel components, it hasn't been well documented.

8.4.1 FUSE Kernel Startup

Figure 8.3 shows the steps performed by `fuse_init()` from when the FuSE module is first loaded and the module initialization routine (`module_init(fuse_init)`) is called. This can be found in `fs/fuse/inode.c`.

The four functions performed during this phase are:

1. `fuse_fs_init()` – this call initializes the `fuse` filesystem which is the first entry point into FUSE when file requests enter the kernel through the mount point. It also registers the `fuseblk` filesystem. **XXX—more when understood.**
2. `fuse_dev_init()` – the character device driver `/dev/fuse` is registered with the kernel through the call to `misc_register()`.
3. `fuse_sysfs_init()` – this allows all connections (FUSE-mounted filesystems) to be visible through `/sys/fs/fuse/connections`.
4. `fuse_ctl_init()` – this registers the `fusectl` filesystem which allows control and monitoring of FUSE file system behavior.

After `fuse_init()` has completed there are two filesystems (`fuse` and `fusectl`) and one device driver (also called `fuse`) that share code in the same kernel module.

```

fuse_init() - fuse.c
- INIT_LIST_HEAD(&fuse_conn_list);

① fuse_fs_init();
② fuse_dev_init();
③ fuse_sysfs_init();
④ fuse_ctl_init();

① fuse_fs_init() - inode.c
- fuse_inode_cachep = kmalloc_cache_create(...)
- register_fuseblk();
- register_filesystem(&fuse_fs_type);

② fuse_dev_init() - dev.c
- fuse_req_cachep = kmalloc_cache_create("fuse_request",
- misc_register(&fuse_misdevice);
misc_register() - drivers/char/misc.c
- Register a miscellaneous device with the kernel

③ fuse_sysfs_init() - inode.c
- kobject_create_and_add("fuse", fs_kobj);
- sysfs_create_mount_point(fuse_kobj, "connections");

④ fuse_ctl_init() - control.c
- register_filesystem(&fuse_ctl_fs_type);

```

Figure 8.3: FUSE Kernel Startup

8.4.2 Establishing Connection between the Kernel and **libfuse**

Section 8.6, describes the pSPFS filesystem, and is the first of three example FUSE filesystems in this chapter showing a passthrough filesystem where operations are passed through a FUSE mount point, through the libfuse library and into pSPFS. Here is an abbreviated example of how this filesystem daemon startups up. It declares a list of filesystem operations that it supports and calls into `fuse_main()` passing this operations vector together with any arguments passed on the command line (including mount point and target filesystem).

```

static struct fuse_operations spfs_operations = {
    .create        = sp_create,
    .unlink        = sp_unlink,
    ...
    .statfs        = sp_statfs
};

int

```

```
main(int argc, char *argv[])
{
    return fuse_main(argc, argv, &spfs_operations, NULL);
}
```

When the libfuse daemon starts there are several things operations that are performed before the user-space filesystem can be accessed. Note that two functions described are defined as macros in `include/fuse.h`:

- `fuse_main` is defined as `fuse_main_real`.
- `fuse_new` is defined as `fuse_new_31`.

You can find `fuse_main_real()` in the `lib` directory. Note that the following text will still use the function names `fuse_main()` and `fuse_new`. There is a useful comment above the macro definition in this header file that describes the process followed by `fuse_main()` and error codes that can be returned.

The steps followed by `fuse_main()` are as follows and shown in figure 8.4. All functions in the libfuse library described can be found under the `lib` directory.

1. pSPFS calls `fuse_main()` which parses the arguments passed to the program by calling `fuse_parse_cmdline()`. It then calls `fuse_mount()`.
2. `fuse_mount()` creates a UNIX domain socket pair, then forks and executes the `fusermount(1)` command (`util/fusermount.c`) passing it one end of the socket.
3. `fusermount()` makes sure that the FUSE module is loaded then opens `/dev/fuse` and sends the file handle back to `fuse_mount()`.
4. `fuse_mount()` returns the file handle for `/dev/fuse` to `fuse_main()`.
5. `fuse_main()` calls `fuse_new()` (in `lib/fuse.c`) to allocate a 'struct fuse' object that stores and maintains a cached image of the filesystem data. **XXX—what cached image?**
6. `fuse_main()` calls either `fuse_loop()` or `fuse_loop_mt()` which starts to read the filesystem system calls from `/dev/fuse`, call the user-supplied functions stored in a 'struct fuse_operations' object before calling `fuse_main()`. The results of those calls are then written back to the `/dev/fuse` file where they can be forwarded back through the `fuse` filesystem and allow the system calls to return.

The figure shows the structures that are allocated prior to entering `fuse_loop()` or `fuse_loop_mt_32()`. From here you can see the file descriptor that is used to access `/dev/fuse` as well as the low-level and high-level operations vectors that the user-space filesystem provided when calling `fuse_main()`.

The `fuse_loop()` function calls `fuse_session_loop()` which sits in a loop as the following fragment of code shows:

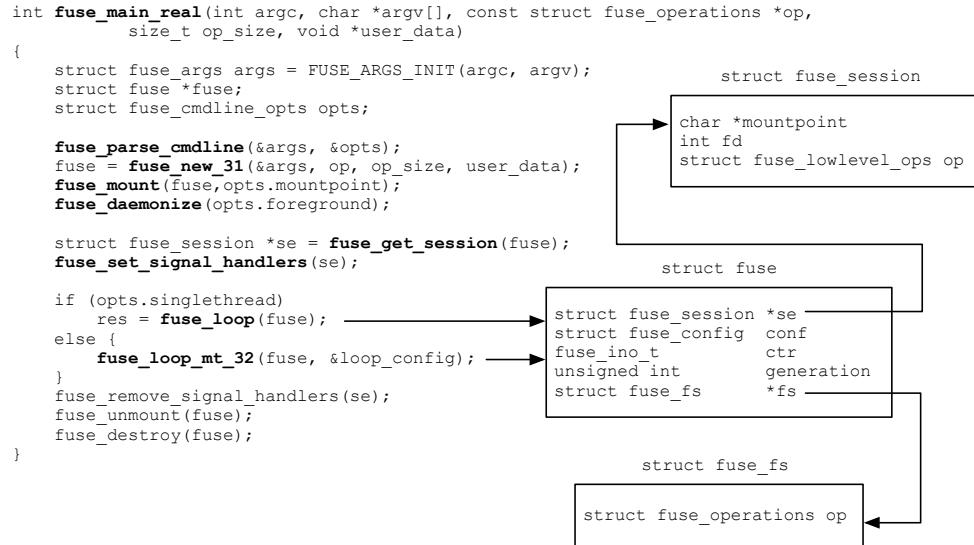


Figure 8.4: Startup paths for libfuse

```

int fuse_session_loop(struct fuse_session *se)
{
    while (!fuse_session_exited(se)) {
        res = fuse_session_receive_buf_int(se, &fbuf, NULL);
        fuse_session_process_buf_int(se, &fbuf, NULL);
    }
}

```

XXX—where does `struct fuse` / `struct fuse_fs` go? The latter has the `ops` vector

8.4.3 Mounting a `fuse` Filesystem

XXX - should cover `fusemount` since that's what kicks it off

XXX - cover INIT - when fs is being mounted see one 2. DESTROY sent when fs is unmounted and there will be no more kernel requests. probably new section for latter

XXX—not sure this is the right place but will see

When most filesystems register they declare a `mount` function as part of the `file_system_type` structure which is passed to `register_filesystem()`. The `fuse` filesystem does not register such a function. There is a fn `fuse_fill_super()` in the kernel but can't see who calls it.

https://docs.kernel.org/filesystems/mount_api.html – describes a newer mount process. Looks like a lot of filesystems are doing this now so might want to switch SPFS too?

NOTE – see `fs/fs_context.c` for how this function is handled

The creation of new mounts is now to be done in a multistep process:

1. Create a filesystem context.
2. Parse the parameters and attach them to the context. Parameters are expected to be passed individually from userspace, though legacy binary parameters can also be handled.
3. Validate and pre-process the context.
4. Get or create a superblock and mountable root.
5. Perform the mount.
6. Return an error message attached to the context.
7. Destroy the context.

To support this, the `file_system_type` struct gains two new fields:

```
int (*init_fs_context)(struct fs_context *fc);
const struct fs_parameter_description *parameters;
```

Let's go explore this some more. The fuse filesystem declares `fuse_init_fs_context()` and here it is:

```
static int fuse_init_fs_context(struct fs_context *fsc)
{
    struct fuse_fs_context *ctx;

    ctx = kzalloc(sizeof(struct fuse_fs_context), GFP_KERNEL);

    ctx->max_read = ~0;
    ctx->blksize = FUSE_DEFAULT_BLKSIZE;
    ctx->legacy_opts_show = true;

    fsc->fs_private = ctx;
    fsc->ops = &fuse_context_ops;
    return 0;
}

static const struct fs_context_operations fuse_context_ops = {
    .free        = fuse_free_fsc,
    .parse_param = fuse_parse_param,
    .reconfigure = fuse_reconfigure,
    .get_tree    = fuse_get_tree,
};
```

xxx

8.4.4 General Flow Between Kernel and User-space

The kernel doc has a nice example of the flow between the kernel and `libfuse` in response to an `unlink(2)` system call. Figure 8.5 gives an expanded version and brings it more

up to date with the existing FUSE code base. The angle brackets show when a function is entered and when it exits. For example, > `sys_read()` shows the function being entered and

Can really put a pause in the daemon and see the kernel stack from the requesting process < `sys_read()` shows when it exits.

unused_list - doesn't exist in the source code so need to figure out what's correct
xxx

8.4.5 FUSE VFS Interfaces

The fuse kernel filesystem provides the following operations vectors. These vectors are defined in `fs/fuse/dir.c` and `fs/fuse/file.c`

struct super_operations	fuse_super_operations
struct export_operations	fuse_export_operations
struct inode_operations	fuse_dir_inode_operations
struct inode_operations	fuse_common_inode_operations
struct inode_operations	fuse_symlink_inode_operations
struct file_operations	fuse_dir_operations
struct address_space_operations	fuse_file_aops

The first two vectors deal with filesystem-related operations. The first is attached to the `s_op` and `s_export_op` fields of the `super_block` structure respectively. The other five vectors are attached to the `i_op` or `i_fop` and `i_data.a_ops` fields of the `inode` structure depending on the vector and the file type.

Each fuse filesystem VFS entry point follows a similar path. Take `fuse_mkdir()` as an example and also compare with a very similar path taken by `fuse_unlink()` (figure 8.2). A message is created with the appropriate arguments needed for the libfuse daemon to fulfill the request, the message is queued and if the daemon is waiting for a request, it will be woken up.

```
static int fuse_rmdir(struct inode *dir, struct dentry *entry)
{
    int err;
    struct fuse_mount *fm = get_fuse_mount(dir);
    FUSE_ARGS(args); /* struct fuse_args */

    /* Fill in fuse_args with information needed by libfuse
       and the user-space filesystem */

    args.opcode = FUSE_RMDIR;
    args.nodeid = get_node_id(dir);
    args.in_numargs = 1;
    args.in_args[0].size = entry->d_name.len + 1;
    args.in_args[0].value = entry->d_name.name;

    /* Add the message to the pending list*/
```

Kernel	FUSE filesystem daemon
<pre> > sys_unlink() > fuse_unlink() [get request from fc->unused_list] > request_send() [queue req on fc->pending] [wake up fc->waitq] >request_wait_answer() [sleep on req->waitq] [woken up] </pre>	<pre> read from /dev/fuse > sys_read() > fuse_dev_read() > fuse_dev_do_read() >request_wait() [sleep on fc->waitq] [woken up] <request_wait() [remove req from fc->pending] [copy req to read buffer] [add req to fc->processing] < fuse_dev_do_read() < fuse_dev_read() < sys_read() returns [perform unlink] > sys_write() [look up req in fc->processing] [remove from fc->processing] [copy write buffer to req] [wake up req->waitq] < fuse_dev_write() < sys_write() < request_wait_answer() < request_send() [add request to fc->unused_list] <f use unlink() < sys_unlink() </pre>

Figure 8.5: Flow between the kernel and the FUSE daemon

```

err = fuse_simple_request(fm, &args);

if (!err) {
    fuse_dir_changed(dir);
    fuse_entry_unlinked(entry);
}
return err;
}

```

A call to `get_node_id()` gets the fuse inode (of type `struct fuse_inode`) and returns the `nodeid` field. This is a Unique ID, which identifies the inode between user-space and kernel. **XXX—it must be the inode number returned by the FUSE daemon otherwise what else would it be? It's passed as an argument so look some more at the inode/daemon code.**

A call to `get_fuse_mount()` returns the `fuse_mount` structure that is referenced from the `s_fs_info` field of the `super_block` structure for this mount point.

The `struct fuse_args` structure is defined in `fuse/fuse_i.h`. Here is a subset of the structure:

```

struct fuse_args {
    uint64_t nodeid;
    uint32_t opcode;
    unsigned short in_numargs;
    unsigned short out_numargs;
    ...
    struct fuse_in_arg in_args[3];
    struct fuse_arg out_args[2];
    void (*end)(struct fuse_mount *fm, struct fuse_args *args,
                int error);
};

```

The `opcode` field represents the operation that is performed by the user-space filesystem. In the `libfuse` source code, the `fuse_ll_ops` structure (`fuse_lowlevel.c`) contains all of the opcodes and functions that will be called. For example, for `FUSE_RMDIR`, a call will be made to `do_rmdir()`:

```

static void do_rmdir(fuse_req_t req, fuse_ino_t nodeid,
                     const void *inarg)
{
    char *name = (char *) inarg;

    if (req->se->op.rmdir)
        req->se->op.rmdir(req, nodeid, name);
    else
        fuse_reply_err(req, ENOSYS);
}

```

The `rmdir` function called here is the function provided by the user-space filesystem.

8.4.6 FUSE to libfuse Request Queues

There are five FUSE queues that are used to communicate between the `fuse` filesystem and `libfuse` through the `fuse` device driver. These queues are:

1. **pending** – for synchronous requests - looks like FUSE calls this the input queue (see definition of `fuse_conn`, element `iq`)
2. **background** – for async requests???
3. **processing** – when a message has been picked up for processing it is moved from the pending queue to the processing queue. XXX - Same for others????
4. **forgets** – not related to above (requests posted here separately)
5. **interrupts** – not related to above (requests posted here separately)

Some are initialized in `fuse_iqueue_init()` which is passed a `fuse_iqueue` structure.

`fuse_get_tree->fuse_conn_init() ->fuse_iqueue_init()` - first function called during mount processing and also described elsewhere here.

```
static void fuse_iqueue_init(struct fuse_iqueue *fiq,
                             const struct fuse_iqueue_ops *ops,
                             void *priv)
{
    memset(fiq, 0, sizeof(struct fuse_iqueue));
    spin_lock_init(&fiq->lock);
    init_waitqueue_head(&fiq->waitq);
    INIT_LIST_HEAD(&fiq->pending);
    INIT_LIST_HEAD(&fiq->interrupts);
    fiq->forget_list_tail = &fiq->forget_list_head;
    fiq->connected = 1;
}
```

Here are the relevant fields in the `fuse_iqueue` structure:

```
struct fuse_iqueue {
    struct list_head      pending;
    struct list_head      interrupts;
    struct fuse_forget_link  forget_list_head;
    struct fasync_struct  *fasync;
}
```

The *processing queue* is inside the `fuse_dev` structure as follows: **XXX - but where is this bugger?**

```
struct fuse_dev {
    struct fuse_conn     *fc;      /* Fuse connection for this dev */
    struct fuse_pqueue   pq;      /* Processing queue */
    struct list_head     entry;   /* list entry on fc->devices */
};
```

It's not obvious who sets file->private_data as it's set to NULL in open below and then assumed to be there when reading /dev/fuse. It's set by fuse_do_open() but who calls this?

```

static int fuse_dev_open(struct inode *inode, struct file *file)
{
    /*
     * The fuse device's file's private_data is used to hold
     * the fuse_conn(ection) when it is mounted, and is used to
     * keep track of whether the file has been mounted already.
     */
    file->private_data = NULL;
    return 0;
}

static ssize_t
fuse_dev_read(struct kiocb *iocb, struct iov_iter *to)
{
    struct file *file = iocb->ki_filp;
    struct fuse_dev *fud = fuse_get_dev(file);

    return fuse_dev_do_read(fud, file, &cs, iov_iter_count(to));
}

static ssize_t
fuse_dev_do_read(struct fuse_dev *fud, struct file *file,
                 struct fuse_copy_state *cs, size_t nbytes)
{
    struct fuse_pqueue *fpq = &fud->pq;
}

fuse_get_tree() calls fuse_conn_init() which calls fuse_iqueue_init()
Requests are zzz the fuse_req structure which can be found in fs/fuse/fuse_i.h

struct fuse_req {
    struct list_head list;
    struct list_head intr_entry;
    struct fuse_args *args;
    refcount_t count;
    unsigned long flags;
    struct {
        struct fuse_in_header h; /* The request input header */
    } in;
    struct {
        struct fuse_out_header h; /* The request output header */
    } out;
    wait_queue_head_t waitq;
    void *argbuf;
    struct fuse_mount *fm; /** fuse_mount this request belongs to */
};

```

XXX

```
fuse_request_alloc() -> kmem_cache_zalloc()
```

As described earlier, the `fuse` filesystem operations for many regular file operations involves creating a message containing all the information needed to process the request in the `libfuse` daemon and then make a call to `fuse_simple_request()` to queue the request.

```
ssize_t fuse_simple_request(struct fuse_mount *fm,
                           struct fuse_args *args)
{
    req = fuse_request_alloc(fm, GFP_KERNEL | __GFP_NOFAIL);
or
    req = fuse_get_req(fm, false);

    __fuse_request_send(req);
}

static void __fuse_request_send(struct fuse_req *req)
{
    struct fuse_iqueue *fiq = &req->fm->fc->iq;

    __fuse_get_request(req);
    queue_request_and_unlock(fiq, req);
}
```

XXX and then adds the message to the tail end of the *pending* queue as follows:

```
static void queue_request_and_unlock(struct fuse_iqueue *fiq,
                                     struct fuse_req *req)
{
    req->in.h.len = sizeof(struct fuse_in_header) +
                    fuse_len_args(req->args->in_numargs,
                                  (struct fuse_arg *) req->args->in_args);
    list_add_tail(&req->list, &fiq->pending);
    fiq->ops->wake_pending_and_unlock(fiq);
}
```

XXX

```
static void fuse_dev_wake_and_unlock(struct fuse_iqueue *fiq)
{
    wake_up(&fiq->waitq);
    kill_fasync(&fiq->fasync, SIGIO, POLL_IN);
    spin_unlock(&fiq->lock);
}
```

XXX

8.4.7 Kernel to libfuse Messages

In the `fuse.h` header file, "enum `fuse_opcode`" defines all of the kernel to libfuse message types that both sides used during communication. Several of them are also described in the `fuse(4)` manpage.

```
enum fuse_opcode {
    FUSE_LOOKUP          = 1,
    FUSE_FORGET          = 2, /* no reply */
    FUSE_GETATTR          = 3,
    FUSE_SETATTR          = 4,
    FUSE_READLINK         = 5,
    FUSE_SYMLINK          = 6,
    ...
    FUSE_REMOVEMAPPING   = 49,
    FUSE_SYNCFS           = 50,
}
```

For most of these message types, there will be a corresponding `fuse` filesystem VFS function that processes the system call. This was shown earlier but for completeness, here is the `fuse` VFS function supporting the `symlink(2)` system call with the VFS function and opcode highlighted:

```
static int fuse_symlink(struct user_namespace *mnt_userns,
                        struct inode *dir, struct dentry *entry,
                        const char *link)
{
    struct fuse_mount *fm = get_fuse_mount(dir);
    unsigned len = strlen(link) + 1;
    FUSE_ARGS(args);

    args.opcode = FUSE_SYMLINK;
    args.in_numargs = 2;
    ...
}
```

The message to send to libfuse is constructed and put on the pending queue to be picked up by the daemon. In libfuse there is a structure in `fuse_lowlevel.c` called `fuse_ll_ops` that contains the same list of opcodes plus all of the functions that are called in response. Here is a fragment showing `FUSE_SYMLINK`:

```
static struct {
    void (*func)(fuse_req_t, fuse_ino_t, const void *);
    const char *name;
} fuse_ll_ops[] = {
    [FUSE_LOOKUP]      = { do_lookup,      "LOOKUP"      },
    ...
    [FUSE_SYMLINK]   = { do_symlink,     "SYMLINK"    },
    ...
}
```

The libfuse functions in this structure follow a similar path in terms of marshaling the arguments and calling the user-level filesystem provided function if one is available. Here is the `do_symlink` function:

```
static void do_symlink(fuse_req_t req, fuse_ino_t nodeid,
                      const void *inarg)
{
    char *name = (char *) inarg;
    char *linkname = ((char *) inarg) +
                     strlen((char *) inarg) + 1;

    if (req->se->op.symlink)
        req->se->op.symlink(req, linkname, nodeid, name);
    else
        fuse_reply_err(req, ENOSYS);
}
```

If you look at other functions in `fuse_lowlevel.c` you will see that they all follow a very similar path.

8.4.8 Controlling FUSE Through `fusectl`

For each fuse mounted filesystem, the `fusectl` filesystem creates a directory named by a unique number that gives information about each mountpoint. **XXX - find where it is created**

The `fusectl` source code can be found in `fs/fuse/control.c`. Each time a FUSE filesystem is mounted, `fuse_fill_super_common()` in the `fuse` filesystem will make a call into `fusectl` to create the appropriate tree structure for the new mount point under `/sys/fs/fuse/connections` as follows:

```
fuse_ctl_add_conn()
{
    sprintf(name, "%u", fc->dev);
    parent = fuse_ctl_add_dentry(parent, fc, name,
                                 S_IFDIR | 0500, 2,
                                 &simple_dir_inode_operations,
                                 &simple_dir_operations);

    if (!fuse_ctl_add_dentry(parent, fc, "waiting",
                           S_IFREG | 0400, 1,
                           NULL, &fuse_ctl_waiting_ops) ||
        !fuse_ctl_add_dentry(parent, fc, "abort",
                           S_IFREG | 0200, 1,
                           NULL, &fuse_ctl_abort_ops) ||
        !fuse_ctl_add_dentry(parent, fc, "max_background",
                           S_IFREG | 0600,
                           1, NULL, &fuse_conn_max_background_ops) ||
        !fuse_ctl_add_dentry(parent, fc, "congestion_threshold",
                           S_IFREG | 0600, 1, NULL,
```

```

    &fuse_conn_congestion_threshold_ops))
    ...
}

```

For each connection the following files exist within this directory:

- `waiting` – the number of requests which are waiting to be transferred to user-space or being processed by the FUSE filesystem daemon. If there is no filesystem activity and `waiting` is non-zero, the filesystem is hung or deadlocked.
- `abort` – writing anything into this file will abort the filesystem connection. All requests waiting will be aborted and an error returned for all aborted and new requests.
- `congestion_threshold` – if the number of outstanding requests exceeds the value visible in this file (9 by default and 75% of `max_background`, the filesystem will be marked as "congested". This changes the way in which FUSE components handle subsequent requests assuming that the filesystem daemon will take some time to complete requests. One such action is to sleep rather than waiting in a busy loop.
- `max_background` – requests in the background (async) queue move to the pending queue over time. This value defines the limit of such requests that can reside on the pending queue in order to prevent synchronous requests from being delayed significantly - **XXX—the ACM doc has good info so come back to this once i do the performance section**

Only the owner of the mount may read or write these files.

```

$ ls /sys/fs/fuse/connections/
47/
$ ls /sys/fs/fuse/connections/47
abort  congestion_threshold  max_background  waiting

```

XXX—i'm going to do an example where an op goes to sleep so i can look at stack traces etc. come back and show `waiting` which should go to "1"

8.4.9 Multithreading

I guess mostly in the library? **this is all copied from the web**

Background: When a file is opened, the Linux kernel creates a "file description" for the I/O state, and returns a "file descriptor" to userland. That descriptor can be freely passed to the `dup(2)` functions to duplicate the descriptor, but the underlying description remains unary.

The FUSE kernel driver implicitly locks access to the `/dev/fuse` file descriptor so that each `read()` and `write()` syscall is atomic. This implies that multiple threads can safely share the descriptor, but also that they will face lock contention and reduced performance.

To get the best performance out of a multi-threaded filesystem server, open `/dev/fuse` once as a "session FD" and again in each thread as "worker FDs". After

initializing the session with a standard FUSE handshake, the workers can be associated with the session by calling `ioctl(worker_fd, FUSE_DEV_IOC_CLONE, &session_fd)`.

This allows multiple threads to serve FUSE requests without contending for the descriptor lock.

8.4.10 Time to FORGET

Cover FORGET. Daemon May cache a lot of stuff rather than my simple pass through so important - this is from the ACM paper

Every time an existing inode is looked up (or a new one is created), the kernel keeps the inode in the inode and directory entry cache (dcache). When removing an inode from the dcache, the kernel passes the FORGET request to the FUSE daemon. FUSE's inode reference count (in user-space file systems) grows by one with every reply to LOOKUP, CREATE, and so forth, requests. FORGET requests pass an nlookups parameter which informs the user-space file system (the FUSE daemon) how many lookups to forget. At this point, the user-space file system (the FUSE daemon) might decide to deallocate any corresponding data structures (once their reference count goes to 0). BATCH_FORGET allows the kernel to forget multiple inodes with a single request.

8.4.11 FUSE Interrupts

taken from web so rewrite

If a process issuing a FUSE filesystem request is interrupted, the following happens:

- 1) if the request is not yet sent to user-space AND the signal is fatal, the request is dequeued and returns immediately.
- 2) if the request is not yet sent to user-space AND the signal is not fatal, an 'interrupted' flag is set for the request. When the request has been successfully transferred to user-space and this flag is set, an INTERRUPT request is queued.
- 3) if the request is already sent to user-space, then an INTERRUPT request is queued.

INTERRUPT requests take precedence over other requests, so the user-space filesystem receives queued INTERRUPTS before any others.

The user-space filesystem may ignore the INTERRUPT requests entirely, or may honor them by sending a reply to the "original" request, with the error set to EINTR (the call did not succeed because it was interrupted)

It is also possible that there is a race between processing the original request and its INTERRUPT request. There are two possibilities:

- 1) The INTERRUPT request is processed before the original request is processed
- 2) The INTERRUPT request is processed after the original request has been answered

If the filesystem cannot find the original request, it should wait for some timeout and/or a number of new requests to arrive, after which it should reply to the INTERRUPT request with an EAGAIN error. In case

1) the INTERRUPT request will be re-queued. In case 2) the INTERRUPT reply will be ignored.

The `fuse_read_interrupt()` function sends a message of type `FUSE_INTERRUPT`.
XXX—hmm need to look more at this

8.4.12 Per-File DAX Option

TBD - per-file direct access (DAX) - just something else to look at

8.4.13 The `fuseblk` Filesystem

might help - and

One area that has little documentation is around `fuseblk` for which the target is block device and not a filesystem or a directory within an existing filesystem. There has been little usage offuseblk although the most common usage that has been described on the web is for supporting NTFS-3G, a FUSE-based NTFS filesystem that was introduced in 2007. The developers of NTFS-3G later formed Tuxera Inc., producing Tuxera NTFS, a proprietary version. NTFS-3G is now the free "community edition".

In 2021, a different version of NTFS was merged into the mainline kernel. Linux Torvalds complained about the performance of NTFS-3G. Can you read about the discussions/arguments around this choice here:



URL 25 – <https://tinyurl.com/yvnm5bp>

xxx - code is in `fuse/inode.c` for registering the filesystem There is also `fuse/dev.c`

```
static struct file_system_type fuseblk_fs_type = {
    .owner          = THIS_MODULE,
    .name           = "fuseblk",
    .init_fs_context = fuse_init_fs_context,
    .parameters     = fuse_fs_parameters,
    .kill_sb        = fuse_kill_sb_blk,
    .fs_flags       = FS_REQUIRES_DEV | FS_HAS_SUBTYPE,
};

MODULE_ALIAS_FS("fuseblk");

static inline int register_fuseblk(void)
{
    return register_filesystem(&fuseblk_fs_type);
}

static inline void unregister_fuseblk(void)
{
    unregister_filesystem(&fuseblk_fs_type);
}
```

Note that there is no mount function declared as per SPFS et al so why a filesystem type?

Examples

NTFS - NTFS-3G - <https://en.wikipedia.org/wiki/NTFS-3G>

8.4.14 FUSE and Fsnotify

article about FUSE and fsnotify which dates back to 2016 and says that fsnotify doesn't work - take a look

8.4.15 User-space Character Devices

TBD

8.5 More on libfuse

XXX - needed or not? could add earlier

8.5.1 The `fuse_file_info` Structure

For all operations in the low-level (or high-level?) API, a `fuse_file_info` structure is passed as an argument. This structure is passed between the kernel and libfuse for open instances of a file.

This structure is only in `fuselib` and not in the kernel so the library must construct it from whatever messages it receives from the kernel.

```
struct fuse_file_info {
    int          flags;
    int          writepage;
    unsigned int direct_io : 1;
    unsigned int keep_cache : 1;
    unsigned int flush : 1;
    unsigned int nonseekable : 1;
    unsigned int flock_release : 1;
    uint64_t     fh;
    uint64_t     lock_owner;
};
```

The `fh` field is a file handle set by a FUSE daemon during an open call to store the file descriptor returned by `open(2)`. When passed to the daemon in subsequent calls (say read and write), the daemon can make system calls using this file handle and avoid having to open the file on each call. This has obvious performance advantages.

XXX—need to see what other fields are worth describing. Who uses them?

8.6 Implementing a FUSE-based Filesystem

This section presents three different FUSE-based filesystems. The most simple filesystem called pSPFS ("p" for passthrough) is only 232 lines of code (including a lot of logging) so is very simple to understand. A modified version called eSPFS is a little more complicated since it provides encryption/decryption of regular file contents. Both of these filesystems use the high-level FUSE APIs. A third example explores one of the libfuse example filesystems which provides a single file which shows the current time each time the file is read.

All code can be found here - [github-XXX](#)

8.6.1 Installing FUSE and Compiling the Filesystem

To use an existing FUSE filesystem requires XXX - XXX—redo all of this. it's not just to use and existing FS. libfuse-dev is needed to build daemons. Divide into what's needed to say 1) use SSHFS and 2) build pSPFS

On Ubuntu it's as simple as:

```
$ sudo apt-get update -y
$ sudo apt install fuse
$ sudo apt install libfuse-dev
$ sudo apt install pkg-config
```

or do I need to install fuse3?

8.6.2 Getting Started

First we need to determine what operations we will support. The full list can be found in `fuse.h` and there are 50 functions that a FUSE daemon can support at the time of writing. Here are the operations that pSPFS supports (11 in total). It's interesting to know which functions to support. For example, I didn't have anything for `utimens` but then saw an error when running `touch` file.

```
static struct fuse_operations spfs_operations = {
    .create        = sp_create,
    .unlink        = sp_unlink,
    .open          = sp_open,
    .read          = sp_read,
    .write         = sp_write,
    .getattr       = sp_getattr,
    .readdir       = sp_readdir,
    .mkdir         = sp_mkdir,
    .rmdir         = sp_rmdir,
    .rename        = sp_rename,
    .statfs        = sp_statfs
};
```

The source code for fuse-SPFS can be found on gitlab here:



URL 26 – TBD--URL to go here

The main function is very simple. We create a log file to record which operations we see and then enter the FUSE library by calling `fuse_main()`.

```
int
main(int argc, char *argv[])
{
    logfile = fopen("spfs.log", "w+");
    if (logfile == NULL) {
        printf("spfs: failed to open logfile\n");
        return -1;
    } else {
        setbuf(logfile, NULL);
        return fuse_main(argc, argv, &spfs_operations, NULL);
    }
}
```

Here is the makefile for pSPFS:

```
all:
gcc -Wall spfs.c -D_FILE_OFFSET_BITS=64 `pkg-config fuse \
--cflags --libs` -o spfs
```

8.6.3 Mounting the Filesystem

Once the program has been compiled it can be mounted using the `mountit` script which is in the same source directory:

```
if [ $# != 2 ] ; then
    echo "Usage: mountit <mntpt> <target>"
    exit 1
fi

spfs -omodules=subdir,subdir=$HOME/$2 -d -s -f $1
```

It takes two arguments, the source directory and the target directory. It's assumed the the source directory will be in your current directory and the target directory is relative to your home directory. The `spfs` binary must in in `$PATH`.

When running the script as follows:

```
$ mountit mnt target
```

you will access files through `mnt` and the files are stored in the directory `target`.

The `"-f"` option specifies that the program should run in the foreground which allows us to see debugging information. The `"-d"` option results in debugging options being displayed. The `"-s"` option informs FUSE to use a single thread.

Running `mount(1)` we can see the following (XXX more words).

```
$ mount | grep spate
/home/spate/efuse-spfs/spfs on /home/spate/efuse-spfs/mnt \
    type fuse.spfs (rw,nosuid,nodev,relatime,
                      user_id=1000,group_id=1000)
```

The `subdir` option is also specified. Without this option, all pathnames that FUSE passes to the filesystem are relative. This could be changed such that the filesystem looks at the mount point arguments passed and uses `target` as the path from which the relative pathnames passed by `libfuse` can be used. This would save a lot of string manipulation in the library. But the goal here was to show another methods that's available.

The debugging messages are pretty verbose. For example, just by running the following command:

```
$ cp lorem-ipsum mnt
```

the following debug messages are displayed by `libfuse`:

```
$ mountit mnt target
FUSE library version: 2.9.9
nullpath_ok: 0
nopath: 0
utime_omit_ok: 0
unique: 2, opcode: INIT (26), nodeid: 0, insize: 104, pid: 0
INIT: 7.36
flags=0x73fffffff
max_readahead=0x00020000
INIT: 7.19
flags=0x00000011
max_readahead=0x00020000
max_write=0x00020000
max_background=0
congestion_threshold=0
unique: 2, success, outsize: 40
unique: 4, opcode: GETATTR (3), nodeid: 1, insize: 56, ...
getattr /
unique: 4, success, outsize: 120
unique: 6, opcode: LOOKUP (1), nodeid: 1, insize: 52, ...
LOOKUP /lorem-ipsum
getattr /lorem-ipsum
unique: 6, error: -2 (No such file or directory), ...
unique: 8, opcode: LOOKUP (1), nodeid: 1, insize: 52, ...
LOOKUP /lorem-ipsum
getattr /lorem-ipsum
unique: 8, error: -2 (No such file or directory), ...
unique: 10, opcode: CREATE (35), nodeid: 1, insize: 68, ...
create flags: 0x200c1 /lorem-ipsum 0100644 umask=0002
create[5] flags: 0x200c1 /lorem-ipsum
fgetattr[5] /lorem-ipsum
NODEID: 2
unique: 10, success, outsize: 160
```

```
unique: 12, opcode: GETXATTR (22), nodeid: 2, insize: 68, ...
getxattr /lorem-ipsum security.capability 0
    unique: 12, error: -38 (Function not implemented),...
unique: 14, opcode: WRITE (16), nodeid: 2, insize: 3052, ...
write[5] 2972 bytes to 0 flags: 0x20001
    write[5] 2972 bytes to 0
    unique: 14, success, outsize: 24
unique: 16, opcode: FLUSH (25), nodeid: 2, insize: 64, ...
flush[5]
lock[5] F_SETLK F_UNLCK start: 0 len: 0 pid: 0
    unique: 16, error: -38 (Function not implemented), ...
unique: 18, opcode: RELEASE (18), nodeid: 2, ...
release[5] flags: 0x20001
    unique: 18, success, outsize: 16
```

I found this a little too verbose so decide to implement my own logging by creating a separate log file. This is also useful when the daemon is run in the background.

8.6.4 Initialization and Startup

To get started, simply declare the FUSE operations vector, create the log file (optional) and call `fuse_main()` as follows:

```
static struct fuse_operations spfs_operations = {
    .create      = sp_create,
    .open        = sp_open,
    .read        = sp_read,
    .write       = sp_write,
    .getattr     = sp_getattr,
    .utimens    = sp_utimens,
    .readdir    = sp_readdir,
    .mkdir       = sp_mkdir,
    .rmdir       = sp_rmdir,
    .rename      = sp_rename,
    .statfs     = sp_statfs
};

int
main(int argc, char *argv[])
{
    logfile = fopen("spfs.log", "w+");
    if (logfile == NULL) {
        printf("spfs: failed to open logfile\n");
        return -1;
    } else {
        setbuf(logfile, NULL);
        return fuse_main(argc, argv, &spfs_operations, NULL);
    }
}
```

Any arguments that are passed to the program are passed through to `fuse_main()`. The logfile will be created in the directory from which you run the script. There are calls to print messages to the log file in each of the FUSE operations, for example:

```
fprintf(logfile, "sp_statfs for %s\n", path);
```

Of course, these logging messages are optional if you prefer to just rely on the debug messages displayed by FUSE.

8.6.5 What Happens During Filesystem Operations?

One of the nice things about developing a FUSE-based filesystem is that you can place files in the target directory and first implement functions to access these files such as retrieving file attributes, reading directories and reading file contents. So let's start with a system where we have already copied two files into the target directory.

Following a call to `mountit` and after running `"ls mnt"` here are the operations that are logged:

```
$ cat spfs.log
sp_getattr for /home/spate/fuse-spfs/target/
sp_readdir for /home/spate/fuse-spfs/target/
sp_getattr for /home/spate/fuse-spfs/target/newfile
sp_getattr for /home/spate/fuse-spfs/target/lorem-ipsum
```

and following `cat mnt/lorem-ipsum`:

```
sp_getattr for /home/spate/fuse-spfs/target/
sp_readdir for /home/spate/fuse-spfs/target/
sp_getattr for /home/spate/fuse-spfs/target/lorem-ipsum
sp_open for /home/spate/fuse-spfs/target/lorem-ipsum
sp_open got fd = 5
sp_read for /home/spate/fuse-spfs/target/lorem-ipsum (fd = 5)
```

Since `-o modules=subdir,subdir=/home/spate/target` was specified for the mount, each time a call is made into pSPFS, FUSE will prepend `/home/spate/target`. Without this, we would not know where to access the file in question.

Let's look at `sp_open()`:

```
static int
sp_open(const char *path, struct fuse_file_info *fi)
{
    int      fd;

    fprintf(logfile, "sp_open for %s\n", path);
    fd = open(path, fi->flags);
    if (fd == -1) {
        return -errno;
    }
    fprintf(logfile, "sp_open got fd = %d\n", fd);
    fi->fh = fd;
    return 0;
}
```

When we open the file in the target filesystem, we get back a file descriptor `fd` of 5. We set `fi->fh = fd` so that we can use this on subsequent calls. The `fuse_file_info` structure is defined in `fuse_common.h`

Let's take a look at `sp_read()`:

```
static int
sp_read(const char *path, char *buf, size_t size, off_t offset,
        struct fuse_file_info *fi)
{
    size_t sz;
    int fd;

    if (fi == NULL) {
        fprintf(logfile, "sp_read for %s (open first)\n", path);
        fd = open(path, O_RDONLY);
    }
    else {
        fd = fi->fh;
        fprintf(logfile, "sp_read for %s (got fd = %d)\n",
                path, fd);
    }
    if (fd == -1) {
        return -errno;
    }

    sz = pread(fd, buf, size, offset);
    if (sz == -1) {
        sz = -errno;
    }
    if (fi == NULL) {
        close(fd);
    }
    return sz;
}
```

Now the file descriptor is passed back to use through the `fuse_file_info` structure. We also receive the offset to start reading from so we call (`pread(2)`) to seek/read from the correct offset within the file.

Most functions are fairly similar in that we take the arguments passed and call into the base/target filesystem. There are a few things to watch out for. As an example, consider `sp_getattr`:

```
static int
sp_getattr(const char *path, struct stat *stbuf)
{
    int error;

    fprintf(logfile, "sp_getattr for %s\n", path);
    error = lstat(path, stbuf);
    if (error == -1) {
```

```

        error = -errno;
    }
    return error;
}

```

The `lstat(2)` is identical to `stat(2)` unless the file is a symbolic link. In this case, `lstat(2)` returns information about the symlink itself, not the file that the symlink references.

The final function I want to point out is `sp_readdir()` which is very similar to the `myls` command presented earlier in section 2.23.1:

```

static int
sp_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
           off_t offset, struct fuse_file_info *fi)
{
    DIR *dp;
    struct dirent *de;
    (void) offset;
    (void) fi;

    fprintf(logfile, "sp_readdir for %s\n", path);
    dp = opendir(path);
    if (dp == NULL) {
        return -errno;
    }
    while ((de = readdir(dp)) != NULL) {
        struct stat st;
        memset(&st, 0, sizeof(st));
        st.st_ino = de->d_ino;
        st.st_mode = de->d_type << 12;
        if (filler(buf, de->d_name, &st, 0)) {
            break;
        }
    }
    closedir(dp);
    return 0;
}

```

If you recall from section 2.23.1, when we call `readdir(3)` we are not guaranteed any specific order in the way that directory entries are called unless we use `scandir(3)` instead. In our example here, the `ls` command is being run on top of the pSPFS so we don't need to call `scandir(3)` since `ls` will format everything for us.

Please feel free to download the source run, try different commands and see what operations are called.

8.6.6 Running pSPFS in The Background

Section 3.10.1 demonstrated SSHFS, a FUSE-based version of SSH, that allowed a directory to be mounted using the SSH protocol and give access to files and directories as if they

were local. To make the filesystem usable, it runs in the background and could be started from a user's `.bash_login` script on first login.

To background the pSPFS process all that is required is to change the `mounit` script to remove the FUSE logging options. I actually went through the process of daemonizing the application before I realized this! Here is the `dmountit` version.

```
spfs -omodules=subdir,subdir=/home/spate/target mnt
```

Run the script and you'll get your prompt back straightaway. Of course you can check to make sure that it's mounted:

```
$ df -h | grep spate
spfs           38G   12G   25G  32% /home/spate/fuse-spfs/mnt
```

and from here, you can access the filesystem through `mnt` as before. To unmount pSPFS simply enter `umount mnt`.

8.7 Example of Using The Low-level FUSE Interface

In the example above, pSPFS used the FUSE *high-level* interfaces, dealing with pathnames which are supplied to the pSPFS FUSE operations. This section explores the low-level interfaces which deal with inodes as opposed to pathnames and file names.

In the FUSE source code (`libfuse-master/example`) there are several examples and earlier in the chapter, we already mentioned `passthrough.c` on which pSPFS is based. There are several other examples including the following two filesystems that use the low-level FUSE interfaces:

- `hello_ll.c` – a very small 226 LOC filesystem that has one file "hello" which, when read, returns the string "Hello World!\n".
- `notify_inval_inode.c` – implements a single file which when read, returns the current time.
- `notify_inval_entry.c` – similar to `notify_inval_inode.c`, this example calls `fuse_lowlevel_notify_inval_entry()` to invalidate the inode kernel buffer so that a call must be made into the FUSE filesystem on next read.
- `passthrough_ll.c` – the filesystem in `passthrough.c` uses the high-level API (as does pSPFS) whereas `passthrough_ll.c` implements a passthrough filesystem using the low-level API.

Source code for various FUSE components can be found in two main places namely, the kernel source (`fs/fuse`) and in the official FUSE repository.



`/usr/include/fuse/fuse_lowlevel.h` makes for good reading about the low-level FUSE interfaces. There are several example low-level filesystems in `libfuse-master/example` in the FUSE gitlab sources.

If you're developing FUSE filesystems, be careful about looking at the FUSE gitlab repository as opposed to the header files that get installed when you install FUSE on your Linux distribution. Generally speaking, develop for the version installed on your system since user head files will generally need to match those used by the kernel.

The following subsections will explore `notify_inval_inode.c`, which for ease, will be called TFS (the Time Filesystem). It's a simple low-level FUSE filesystem that provides a single file called "current_time" that displays the current time. It has an inode number of 2 and the file contents (the current time) are stored in the global variable `file_contents` and returned in response to a read request.

The filesystem is only 428 LOC so fairly easy to understand. You may want to contrast this with `hello_ll.c` which has a slightly different implementation supporting different FUSE operations and about half the code.

8.7.1 TFS Definitions, Initialization and Startup

Here are the definitions for the single TFS file, its inode number and the buffer where the contents reside and are continually updated:

```
#define FILE_INO 2
#define FILE_NAME "current_time"
static char file_contents[MAX_STR_LEN];
```

Most of the TFS implementation is involved in demonstrating the asynchronous nature in which the filesystem operates showing how to update kernel buffers when the file contents change. **XXX - anything else of interest?**

The best place to start with this filesystem is to look at the FUSE operations. Note that this is of type `fuse_lowlevel_ops` as opposed top `fuse_operations` for high-level FUSE filesystems.

```
static const struct fuse_lowlevel_ops tfs_oper = {
    .lookup      = tfs_lookup,
    .getattr     = tfs_getattr,
    .readdir     = tfs_readdir,
    .open         = tfs_open,
    .read         = tfs_read,
    .forget       = tfs_forget,
    .retrieve_reply = tfs_retrieve_reply,
};
```

Although the implementation is different, the first five (XXX 4 of them) operations are similar to those of pSPFS. The filesystem must be able to lookup a file (during pathname resolution), return attributes about a file (for `stat(2)` and friends, return directory entires and be able to open and read the file. Since you can only read the file, there are no need for a "write" operation or other directory operations. The other two operations are not present in the high-level interface. **XXX lookup???**

The interfaces are different between the high-level and low-level operations. Let's take the lookup and read operations as examples:

```
static void tfs_lookup(fuse_req_t req, fuse_ino_t parent,
                      const char *name)
```

This interface mirrors the filesystem lookup for kernel-based filesystems. The parent inode is passed together with the name of the file to lookup. XXX return inode number but more complex than that. xxx look at code

```
struct fuse_entry_param {
    fuse_ino_t      ino;          # inode number
    unsigned long   generation;   # generation count
    struct stat     attr;         # attributes
    double          attr_timeout; # validity timeout
    double          entry_timeout;# validity timeout
};
```

For each field, there is a description in `/usr/include/fuse/fuse_lowlevel.h` together with instructions as to how these fields should be handled. Basic comments have been added above.

And contrasting the read operations:

```
sp_read(const char *path, char *buf, size_t size, off_t offset,
        struct fuse_file_info *fi)

tfs_read(fuse_req_t req, fuse_ino_t ino, size_t size, off_t off,
          struct fuse_file_info *fi)
```

The high-level function interfaces looks similar to the `read(2)` system call but for the low-level interface, the inode number is passed together with offset and size read.

8.7.2 TFS Startup and Initialization

Figure 8.6 shows TFS starting a new FUSE session and specifying the TFS low-level operations. If the program is multi-threaded it will start a thread which loops inside `update_fs_loop()`. A call is made to XXX and then YYY.

Depending on whether the filesystem is single-threaded or multi-threaded it will then call one of the following functions:

- `fuse_session_loop()` – Enter a single threaded event loop
- `fuse_session_loop_mt()` – Enter a multi-threaded event loop

There are two circumstances under which the event loop terminates. Either the filesystem is unmounted or the connection is explicitly severed by writing 1 to the FUSE abort file in `/sys/fs/fuse/connections/NNN`.

When the event loop terminates because the connection to the FUSE kernel module has been closed, this function returns zero and TFS will call `fuse_session_unmount()` to unmount the filesystem if it is still mounted. The session is then destroyed.

```

main() {
    se = fuse_session_new(&args, &tfs_oper,
                         sizeof(tfs_oper), NULL);

    fuse_session_mount(se, opts.mountpoint)
    ret = pthread_create(&updater, NULL, update_fs_loop, (void *)se);
    if (opts.singlethread)
        ret = fuse_session_loop(se);
    else {
        config.clone_fd = opts.clone_fd;
        config.max_idle_threads = opts.max_idle_threads;
        ret = fuse_session_loop_mt(se, &config);
    }

    pthread_create(&updater, NULL, update_fs_loop, (void *)se);
}

→ update_fs_loop(void *data) {
    while (1) {
        update_fs();
        if (!options.no_notify && lookup_cnt) {
            fuse_lowlevel_notify_store
            fuse_lowlevel_notify_retrieve()
        }
    }
→ update_fs() {
    file_size = strftime(file_contents,
                         MAX_STR_LEN,
                         "The current time is %H:%M:%S\n",
                         now);
}

```

Figure 8.6: Updating the time in the TFS file "file_contents"

8.7.3 Updating the Kernel Inode Buffers

There is a global variable called `lookup_cnt` which is initially set to 0. It is incremented during `tfsl_lookup()` and decremented during `tfsl_forget()`. You can see in figure 8.6 that the TFS thread will update the kernel's buffers associated with this inode by making a call to `fuse_lowlevel_notify_store()` and checking the result by calling `fuse_lowlevel_notify_retrieve()`. This allows processes to see new file contents (new time).

XXX - what are these kernel buffers? need to look at that really good implementation guide ... if i can find it

```
fuse_lowlevel_notify_retrieve()
```

Retrieve data from the kernel buffers

Retrieve data in the kernel buffers belonging to the given inode. If successful then the `retrieve_reply()` method will be called with the returned data.

8.7.4 TFS Operations

There are seven FUSE operations that TFS provides which will be described below. Relevant sections of the code have been copied here but see the full source code for error checking and so on.

tfs_lookup()

The TFS lookup function returns information about `file_contents` assuming this is the file being looked up otherwise an error is returned. The most interesting aspect of the function is incrementing `lookup_cnt` which is checked inside `update_fs_loop()`. If it's set, a call is made into FUSE to inform it to synchronously store data in the kernel buffers belonging to the given inode. The stored data is marked up-to-date (no read will be performed against it, unless it's invalidated or evicted from the cache).

```
tfs_lookup:  
{  
    if (strcmp(name, FILE_NAME) == 0) {  
        e.ino = FILE_INO;  
        lookup_cnt++;  
    }  
    e.attr_timeout = NO_TIMEOUT;  
    e.entry_timeout = NO_TIMEOUT;  
    tfs_stat(ino, &e.attr)  
    fuse_reply_entry(req, &e);  
}
```

`tfs_stat()` returns basic file information for the root directory or for "file_contents" (`ino == FILE_INO`).

tfs_getattr()

This function is called in response to `stat(2)` and friends to return file attributes for the specified inode.

```
tfs_getattr(fuse_req_t req, fuse_ino_t ino,  
            struct fuse_file_info *fi)  
{  
    struct stat stbuf;  
  
    memset(&stbuf, 0, sizeof(stbuf));  
    tfs_stat(ino, &stbuf);  
    fuse_reply_attr(req, &stbuf, NO_TIMEOUT);  
}
```

`tfs_stat()` is very simple returns a small number of attributes for the root directory or for "file_contents" as follows:

```
stbuf->st_ino = ino;  
if (ino == FUSE_ROOT_ID) {
```

```

        stbuf->st_mode = S_IFDIR | 0755;
        stbuf->st_nlink = 1;
    } else if (ino == FILE_INO) {
        stbuf->st_mode = S_IFREG | 0444;
        stbuf->st_nlink = 1;
        stbuf->st_size = file_size;
    }
}

```

Notice that for the root directory `st_nlink` is set to 1 and not 2 as for most filesystems.
XXX—run this and show what it looks like.

tfs_readdir()

Reading a directory is very simple in TFS since only information about "file_contents" needs to be returned.

```

static void tfs_readdir(fuse_req_t req, fuse_ino_t ino,
                       size_t size, off_t off,
                       struct fuse_file_info *fi)
{
    struct dirbuf b;

    memset(&b, 0, sizeof(b));
    dirbuf_add(req, &b, FILE_NAME, FILE_INO);
    reply_buf_limited(req, b.p, b.size, off, size);
    free(b.p);
}

```

As with `tfs_read()` a check is made to validate the offset. **XXX—need to do better at describing**

tfs_open()

There is only one thing to do for opening a file:

```

tfs_open(fuse_req_t req, fuse_ino_t ino,
         struct fuse_file_info *fi)
{
    fi->keep_cache = 1;
    fuse_reply_open(req, fi);
}

```

By default, fuse invalidates the page cache for an inode on every file open. Since TFS is a read-only filesystem this makes no sense. Setting `keep_cache` tells the kernel that any cached data for this file does not need to be invalidated. Other than that, there is little for `tfs_open()` to do.

tfs_read()

There is little for `tfs_read()` to do other than assert that `ino` is equal to `FILE_INO` and then return the contents of the file. A check is made inside `reply_buf_limited()` to see if the offset is valid and then a call is made to `fuse_reply_buf()`.

```
tfs_read(fuse_req_t req, fuse_ino_t ino, size_t size,
          off_t off, struct fuse_file_info *fi
{
    reply_buf_limited(req, file_contents, file_size, off, size);
}

static int
reply_buf_limited(fuse_req_t req, const char *buf,
                  size_t bufsize,
                  off_t off, size_t maxsize)
{
    if (off < bufsize)
        return fuse_reply_buf(req, buf + off,
                              min(bufsize - off, maxsize));
    else
        return fuse_reply_buf(req, NULL, 0);
}
```

If offset == 0 then the whole string in file_contents will be returned. The fuse_reply_buf() function returns the data in the file.

8.7.5 Low-level FUSE Operation Definitions

The /usr/include/fuse/fuse_lowlevel.h header file defines all of the operations - XXX need to figure out what to say about them. Who uses them? Any good examples?

```
fuse_lowlevel_notify_poll() - NO
fuse_lowlevel_notify_inval_inode() - YES (TFS)
fuse_lowlevel_notify_inval_entry() - YES (TFS)
fuse_lowlevel_notify_delete() - NO
fuse_lowlevel_notify_store() - YES (TFS)
fuse_lowlevel_notify_retrieve() - YES (TFS)
fuse_lowlevel_is_lib_option() - NO
```

xxx

8.8 Adding Encryption to pSPFS

A passthrough filesystem is a fine way to explore how FUSE works but beyond that, it doesn't make for a very interesting filesystem. In this section, the filesystem will be enhanced to including encrypting regular file contents. It's also very simple XXX. The encryption key and IV (initialization vector) are hardcoded in the program. Although fine for demo purposes, this would not work in the real world. Keys and IVs should be separate from the running program. Section 10.2 describes encryption and key management best practices.

The source code can be found on gitlab here:

Come back here later



URL 27 – TBD--URL to go here

I based the encryption/decryption code on a simple example that the OpenSSL Foundation have on their website that produces a standalone program which encrypts then decrypts the string "*The quick brown fox jumps over the lazy dog*". You can find the program here:



URL 28 – <https://tinyurl.com/2p96usf6>

It's also in `crypto.c` in the `espfs` source tree.

To get this program to work by itself, copy and paste the code snippets into `crypto.c` and compile as follows:

```
$ cc -o crypto crypto.c -lcrypto -lssl
```

although make sure you have OpenSSL development package installed. The following command should be enough to get access to all the OpenSSL libraries and header files you need:

```
$ sudo apt install libssl-dev
```

If you're interested in playing with encryption/decryption, this is a good start as you can experiment with algorithms and key sizes.

8.8.1 A Primer on Encryption

The eSPFS filesystem will use AES symmetric encryption which is the mostly widely used algorithm for file and disk-based encryption. It will also use 256-bit keys although changing the key size is very simple for new filesystems. AES supports 128-bit, 192-bit, and 256-bit implementations, with AES 256 being the most commonly used. Intel AES-NI instructions support all three key sizes.

There are several things to point out and consider before implementing encryption support:

1. AES operates on *plaintext* and produces *ciphertext* for encryption. It works in the opposite direction for decryption.
2. AES encryption/decryption operates on 128-bit block sizes. You can't encrypt or decrypt data unit that is smaller or larger in a single AES operation. Thus to encrypt say a page of data, it will need to be broken down in 128-bit chunks. The 128-bit block size is divided into a 4x4 array containing 16 bytes. Since there are eight bits per byte, the total in each block is 128 bits. The size of the encrypted data remains the same as the plaintext – 128 bits of plaintext yields 128 bits of ciphertext. This is particularly important for file encryption as we do not want to change the size of the file's data.

3. Initialization Vector (IV) – xxx. I recall interesting conversations with one government agency who wanted a unique IV per AES block. We just couldn't imagine how badly that would perform!
4. previous block vs next block re: IV
5. To encrypt/decrypt, you always need to use the same key and IV.
6. encryption / decryption occurs in 128-bit blocks (256 or bigger keys) therefore we need a complete block before we can perform an encrypt or decrypt. This may involve reading data from disk.
7. The end of a file we may not be on a 128-bit block boundary. If this occurs there are several ways to make sure that we still have a 128-bit block in which to perform the encryption operation. Either we can pad the data with known data (zeroes for example) or can utilize a technique called "*ciphertext stealing*" XXX - need to come back and see what to do here - size of the file. Well we could ... explain ... CBC mode for stealing or others. explain. perhaps just add padding but talk about options and pros/cons

This was just a quick overview of the encryption process. There are many fine books and articles on line if you wish to learn more.

8.8.2 How The eSPFS Filesystem Works

We need to modify each function in the `fuse_operations` structure that deals with file data. Here is our structure again. In this case we only have to modify two of the functions, namely `sp_read()` and `sp_write()` since only those functions deal with regular file data:

```
static struct fuse_operations spfs_operations = {  
    .create          = sp_create,  
    .open            = sp_open,  
    .unlink          = sp_unlink,  
    .read            = sp_read,  
    .write           = sp_write,  
    .getattr         = sp_getattr,  
    .utimens         = sp_utimens,  
    .readdir         = sp_readdir,  
    .mkdir           = sp_mkdir,  
    .rmdir           = sp_rmdir,  
    .rename          = sp_rename,  
    .statfs          = sp_statfs  
};
```

Note that if we were encrypting file names, we would have to modify every function since the filesystem gets passed the full path of each file on which it needs to operate.

Both read and write paths rely on two functions namely `encrypt()` and `decrypt()` which follow very similar paths as we'll see soon.

The encryption key and IV are hardcoded at the top of `spfs.c` as follows:

```

/* A 256-bit key */

unsigned char *key = \
    (unsigned char *)"01234567890123456789012345678901";

/* A 128-bit IV */

unsigned char *iv = (unsigned char *)"0123456789012345";

```

Notice the comment about the fact that both of these values should not be hardcoded.

Since OpenSSL is being used for encryption, there is a specific sequence of OpenSSL operations performed in the following stages:

- Setting up a context
- Initializing the encryption operation
- Providing plaintext bytes to be encrypted
- Finalizing the encryption operation

During initialization we will provide an `EVP_CIPHER` object. In this case we are using `EVP_aes_256_cbc()`, which uses the AES algorithm with a 256-bit key in CBC mode. The code for encryption and decryption is very similar. Here is the function for encryption:

```

1 int
2 encrypt(unsigned char *plaintext, int plaintext_len,
3         unsigned char *key, unsigned char *iv,
4         unsigned char *ciphertext)
5 {
6     EVP_CIPHER_CTX *ctx;
7     int             len, ciphertext_len;
8
9     ctx = EVP_CIPHER_CTX_new();
10
11    EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv);
12
13    EVP_EncryptUpdate(ctx, ciphertext, &len,
14                      plaintext, plaintext_len);
15    ciphertext_len = len;
16
17    EVP_EncryptFinal_ex(ctx, ciphertext + len, &len);
18    ciphertext_len += len;
19
20    EVP_CIPHER_CTX_free(ctx);
21
22    return ciphertext_len;
23 }

```

Here are the steps taken:

- line 9 – Sets up a context
- line 11 – Initializes the encryption operation
- line 13-14 – Provides the plaintext bytes to be encrypted. The ciphertext is returned in ciphertext.
- line 17 – Finalize the encryption operation

XXX — shouldn't get a different size back. Need to do some research here.

The decrypt () function is very similar. Differences are highlighted in bold but you will still see the same sequence of steps:

```
1 int
2 decrypt(char *ciphertext, int ciphertext_len, unsigned char *key,
3         unsigned char *iv, char *plaintext)
4 {
5     EVP_CIPHER_CTX *ctx;
6     int len, plaintext_len;
7
8     ctx = EVP_CIPHER_CTX_new();
9
10    EVP_DecryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv);
11
12    EVP_DecryptUpdate(ctx, (unsigned char *)plaintext,
13                      &len, (unsigned char *)ciphertext,
14                      ciphertext_len))
15
16    plaintext_len = len;
17
18    EVP_DecryptFinal_ex(ctx, plaintext + len, &len);
19
20    plaintext_len += len;
21
22    EVP_CIPHER_CTX_free(ctx);
23
24    return plaintext_len;
25 }
```

This is the easy part. If we could simply call encrypt () from within sp_write () as follows:

XXX—NO! Can't do strlen(). It might not be text. Use "size"

```
sp_write(const char *path, const char *buf, size_t size,
         off_t offset, struct fuse_file_info *fi)
{
    char     *ciphertext, *plaintext = (char *)buf;
    ...
    ciphertext = (char *)malloc(size);
    encrypt(plaintext, strlen(plaintext), key, iv, ciphertext);
    pwrite(fd, ciphertext, size, offset);
```

```

    free(ciphertext);
    ...
}

```

we would be done. Just simply allocate a buffer the same size as `buf`, encrypt the data and write it out to disk. In fact, this is how I started and it works to some degree. Similarly for `sp_read()` / `decrypt()`. But problems arise very quickly. Just try copying in our `lorem-ipsum` file:

```
$ ls -l lorem-ipsum
-rw-r--r-- 1 spate spate 2972 Dec  4 15:43 lorem-ipsum
```

The file is 2972 bytes and recall that the AES block size is 128-bits (16 bytes). This means that there are 12 bytes at the end of the file. XXX padding is used so 2972 bytes are written but when reading the file back we only get 2960 as the length returned from `decrypt()`.

We could pad the last block but that would result on the ciphertext being longer than the plaintext. We certainly don't want to write extra ciphertext to disk so we utilize a technique called *ciphertext stealing*.

There is a NIST special publication that covers ciperhtext stealing namely *Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode*.



URL 29 – <https://tinyurl.com/37nd7x6v>

NIST special publications will be covered in more detail in chapter ???. The mathematics is quite intense so you may be better off viewing the Wikipedia page.

Need a figure here

we will do CBC-CS1 or CS2???

C example - another description - video -

Here is the algorithm that is being followed for the write path

```

sp_write(buf, size, ...)
{
    if amount is divisible by 128? {
        call encrypt_decrypt()
        write data
    } else {
        if we are in the middle of the file {
            read in more data to make everything 128-bit divisible
            call encrypt_decrypt()
            write data
        } else {
            steal
            call encrypt_decrypt()
            write data
        }
    }
}

```

Need to understand various reads/writes in the middle of the file.

Reading is a little easier XXX or is it?

```
sp_read(buf, size, ...)
{
    if amount is divisible by 128? {
        call encrypt_decrypt()
        write data
    } else {
        if we are in the middle of the file {
            read in more data to make everything 128-bit divisible
            call encrypt_decrypt()
            write data
        } else {
            steal
            call encrypt_decrypt()
            write data
        }
    }
}

xxx
```

8.8.3 Extending eSPFS

There are several things that can be done to enhance eSPFS. First of all, embedding encryption keys in the program is not a secure way to provide encryption. One enhancement would be to provide a password for the encryption key that is entered on the command line when starting the FUSE program. You should take a look at PBKDF2 (Password-Based Key Derivation Function 2) which is a key derivation function.

Another possible enhancement would be to utilize the OpenSSL AES-NI engine so that all encryption/decryption uses the Intel AES-NI hardware instructions. It's very likely that these instructions are available on your system but to find out for sure, run the following command:

```
$ grep -m1 -o aes /proc/cpuinfo
aes
```

and check to see that aes is displayed

An additional enhancement — xxx cache info so that we don't set up and EVP stuff every time

8.9 FUSE Performance

TBD

Although the performance of FUSE can be an issue in many production environments, the simplicity of providing a FUSE-based file system without dealing with kernel releases and symbol versioning can be a big win for both foul system, vendors, and customers. For

example, when I was a Thales, we needed to provide a new version of the kernel based filesystem. Every time there was a kernel update. This was a substantial headache for both Thales and the customer base. We also had a FUSE-based encryption file system that avoided many of these headaches and then several examples any performance of a head that's a filesystem hat well, not an issue in specific custom environments.

XXX need to explain - context switches etc. copying data > 2 times ...
good article here - more architecture perhaps see FUSE_CAP - capabilities -

8.9.1 Improving Performance with eBPF

eBPF has been covered several times in the book and is seen as offering many new capabilities to the Linux kernel going forwards. One additional area of interest is in improving FUSE performance.

Ashish Bijlani, and Umakishore Ramachandran who are at Georgia Institute of Technology have been working on some FUSE enhancements under the umbrella of the EXT-FUSE project. They presented their findings at the 2019 USENIX Annual Technical Conference

- See methodology section for some performance features - page 63
- Faster storage is worse for fuse. Page 64
- Performance - We found that for many workloads, an optimized FUSE can perform within 5% of native Ext4. However, some workloads are unfriendly to FUSE and even if optimized, FUSE degrades their performance by up to 83%. Also, in terms of the CPU utilization, the relative increase seen is 31%.

The presentation can be found here:



URL 30 – <https://tinyurl.com/2p8zfv29>

There is also a detailed Usenix paper



URL 31 – <https://tinyurl.com/h8ztwwht>

Work continues at Georgia Tech at the time of writing under the ExtFUSE project. Changes being looked at apply to Gluster, Ceph, EncFS, Android and SDCardFS filesystems.

The ExtFUSE feature source code can be found here:



URL 32 – <https://github.com/extfuse/extfuse>

It will be interesting to see if/when these changes are adopted. Hmm! There have been no changes since 2020. Odd!

8.10 Conclusion

This chapter covered the FUSE architecture and showed how to build user-space filesystems that increases the speed of development and ease of debugging. There has been a little written about FUSE internals, so the goal of this chapter was to document the structures and functions that make up both the FUSE kernel components, as well as the `libfuse` library.

Three example filesystems were presented using both the high-level and low-level APIs. The first just passed operations through to the underlying filesystem without modification which showed how to build a FUSE-based filesystem. The second added encryption for regular file contents using AES encryption in CBC-CS2 mode, a popular encryption algorithm with widely used and NIST recommended cipher mode (**XXX—check words are correct**). The third example, taken from the `libfuse` source code, showed how to use the low-level API

There have been concerns about the performance of FUSE since its inception and there have been many improvements throughout its history with some new changes coming via eBPF. For high-performance filesystem, regular disk-based filesystems will suffice but there are several use cases where the FUSE-based approach works well. When I was at Thales we actually had two encrypting filesystems, one in the kernel and one FUSE-based. Customers used both depending on their use cases.

Chapter 9

Filesystem Debugging

If you’re doing development in the kernel you’ll get used to kernel panics which can occur for a number of reasons (XXX - list some). While developing SPFS, I ran into kernel panics when implementing support for symbolic links. I managed to create the symlink fine but doing an `ls -l` on the directory where the symlink resided, resulted in a panic when trying to read the contents of the symlink (what file it was pointing to).

9.1 Kernel Debugging with kgdb

I recall when the arguments around source level debuggers back in 2000 when we started porting VxFS to Linux. I never understood the arguments against it. Having a good debugger saves a lot of time. Granted, without one, you’ll spend more time thinking through when you’re coding to make sure you get things right. The rapid prototyping (write/compile/test) method can lead to some sloppiness. But a good debugger is paramount to understanding how things work. Setting breakpoints without source code support is painful. I’ve spent way too many years in that position.

We worked on helping add `kgdb` support to the Linux kernel many years ago through one of our VERITAS engineers Amit Kale. And a great help it was to our kernel engineers and many others around the world in the 20 years since.

I’ve found that best way to understand the Linux kernel is to analyze it while running, see how the structures are linked together and watch the flow from function to function. To get going, you’ll need to build your own kernel and have two Linux instances running side by one, one containing the instrumented kernel (the target) and one containing the kernel source, debugging information generated during a kernel build and `gdb` through which to remotely connect to the target.

Showing examples using `gdb` was paramount to having a good book. But while writing I ran into so many issues getting anything to work. Since I have an Apple MacBook Air I turned to VMware Fusion to run my VMs, a hypervisor I’d used for many years. But it didn’t support Apple Silicon. Neither did VirtualBox. I tried UTM. Nothing could be better than a free hypervisor could it? Well, it mostly worked. I was able to get a setup where I

could run `gdb` between two Linux VMs and I could access memory, view structures and so on. But no break points! After a few months I tried Fusion which had since become available. I couldn't connect between the VMs. Sigh! And VirtualBox was still in beta. I then went back to UTM and did as much as I could without breakpoint support to give the other hypervisors time to stabilize. As time passed, there was still no answer. The way I resolved it was to install two Ubuntu VMs in x86_64 emulation. My kernel compilation went from one hour on the native ARM64 to 17 hours under emulation. But it worked and I can't stress how much easier it is to understand the paths through the kernel if you can breakpoint and walk through the kernel paths seeing what arguments are passed and being able to display various kernel structures.

9.1.1 Building a Kernel with kgdb Support

Getting a kernel to compile isn't hard but can be a little fiddly at times and can differ depending on the platform. The instructions listed below are for Ubuntu 22.10. Recognizing that people will want to build SPFS for different Linux distributions, I will add those plus YouTube videos on my website so take a look there and let me know if other distributions that would be useful.

To build a kernel you need a lot of disk space. I created my Ubuntu VM with 80 GB of disk space. **The OS XXX ...** After building the new kernel, 20GB of disk space had been used in the root filesystem (`/home` is in the root filesystem on my VM by default). Be careful with partitioning though.

There are distribution specific instructions for building the kernel. I'm following a more generic process that will hopefully work across distributions. I want to download a kernel from www.kernel.org. My plan is to get a kernel version that's very close to the one that's currently running:

```
$ uname -r  
5.19.0-29-generic
```

I ended up choosing the 5.19.17 kernel sources. To get started, download the kernel, install the necessary development packages and **copy the current config file - make oldconfig???**:

```
$ wget https://cdn.kernel.org/pub/linux/kernel/v5.x/ \
linux-5.19.17.tar.xz
$ sudo apt-get install git fakeroot build-essential \
ncurses-dev xz-utils libssl-dev bc flex libelf-dev bison
$ cd linux-5.19.17
$ tar xvfz linux-5.19.17.tar.xz
$ cp /boot/config-$(uname -r) .config
```

Next you need to make a small number of changes to the `.config` file to enable kgdb support. Since we know which configuration (config) options to change, just edit the file directly. Here are the relevant config options. I've noted which ones are on (y) or off (n) by default:

```
CONFIG_DEBUG_KERNEL=y          # on by default
CONFIG_DEBUG_INFO=y            # on by default
```

```

CONFIG_MAGIC_SYSRQ=y          # on by default
CONFIG_KGDB=y                 # on by default
CONFIG_KGDB_SERIAL_CONSOLE=y  # on by default
CONFIG_KGDB_HONOUR_BLOCKLIST=y # on by default
CONFIG_STRICT_KERNEL_RWX=n    # on by default
CONFIG_STRICT_MODULE_RWX=n    # on by default
CONFIG_DEBUG_INFO_SPLIT=y     # commented out by default
CONFIG_GDB_SCRIPTS=y          # on by default
CONFIG_DEBUG_FS=y              # on by default
CONFIG_RANDOMIZE_BASE=n       # commented out by default

```

On Ubuntu you will also have to set both of the following config options to "":

```

CONFIG_SYSTEM_TRUSTED_KEYS=""
CONFIG_SYSTEM_REVOCATION_KEYS=""

```

According to the official Linux build documentation, only the following configuration options are needed: **XXX—need to check what the actual set really needs to be**

```

# CONFIG_STRICT_KERNEL_RWX is not set
CONFIG_FRAME_POINTER=y
CONFIG_KGDB=y
CONFIG_KGDB_SERIAL_CONSOLE=y

```

Once all that is out of the way, build the kernel. This will take a while so make sure you utilize the available CPUs. Run nproc to get the number of CPUs and then compile as follows (for my VM nproc returns 4):

```
$ time make -j4
```

NOTE do a 2>&1 | tee make.out and explain it but try it first. I'm compiling now so can't try it

I have a Ubuntu VM running on MacOS on an Apple M1 CPU and the whole build takes about an hour. I usually add the time(1) command so I know how long it takes. This helps plan with future builds. However, to get breakpoints working I needed to run a Ubuntu VM under x86_64 emulation. In this case, the compilation takes 17 hours.

The final step is to install modules, generate a new initrd, install the kernel and update GRUB. **XXX—need to get the right lingo**

```

$ sudo make modules_install
$ sudo make install
$ sudo update-initramfs -c -k 5.19.17
$ sudo update-grub

```

One more change is needed for GRUB. We need to disable KASLR as it will make the debugging harder. This is done by booting the kernel with the nokaslr option. To make this change, open /etc/default/grub, find GRUB_CMDLINE_LINUX_DEFAULT and add nokaslr within the quotes. Then update GRUB as follows:

```
$ sudo update-grub
```

At this point, you're ready to reboot. Again, run uname -r before and after rebooting to make sure you end up with the right kernel. So before reboot:

```
$ uname -r  
5.19.0-29-generic
```

and after reboot:

```
$ uname -r  
5.19.17
```

9.1.2 Turning off Automatic Updates

Getting a system in place to use `gdb`/KGDB is painful at times so when you have it working, you don't want it disturbed and have to go through the pain once again. The biggest hurdle you'll face is your Linux distribution applying automatic updates which can include installing a new kernel. One way to avoid this is to snapshot VMs and periodically go back to the snapshot if things no longer work. Another option is to just disable automatic updates. Disabling updates isn't usually a good idea but these are test VMs. Please refer to instructions for your distribution but for Ubuntu, here are the steps to be followed.

The first step is to make changes by editing the following file:

```
$ sudo vi /etc/apt/apt.conf.d/20auto-upgrades
```

Change both lines to the following (both will be "0" by default):

```
APT::Periodic::Update-Package-Lists "0";  
APT::Periodic::Unattended-Upgrade "0";
```

Afterwards, you should run the following command for the changes to take effect:

```
$ sudo dpkg-reconfigure unattended-upgrades
```

Press TAB to select "No" and press ENTER.

9.1.3 Connecting gdb to the Target VM

So back on the target VM, got one level above where your source code tree is and run the following to build a tar file and copy it to your kgdb client VM:

```
$ tar cvfz linux-5.19.17.tgz linux-5.19.17  
$ scp linux-5.19.17.tgz spate@192.168.64.9:<choose-path>/.
```

You will need to untar it on the client VM:

```
$ tar xvfz linux-5.19.17.tgz
```

On the kgdb client VM you'll need to install `gdb` if it's not there already:

```
$ sudo apt install gdb
```

Note that you should be at the top level of the Linux source tree that was copied over from the target VM since access to symbol information will be relative from the directory in which you start `gdb`. Then just run `gdb` specifying the name of the binary (XXX) and then connect to the target VM:

```
$ cd linux-5.19.17
$ gdb vmlinux
(gdb) target remote tcp:192.168.64.1:1234
```

You can also load the kernel symbols as follows from within gdb if you just run gdb and press enter:

```
(gdb) symbol-file vmlinux
```

The "target remote" command here is UTM-specific. If you are using a different hypervisor or physical machines, you will need to change this command. For example, on VMware Fusion, you would have: **XXX - why did I say this?**

```
(gdb) target remote 8864 # using localhost on Apple Mac
(gdb) target remote 192.168.192.131:8864 # connect to a remote VM
```

9.1.4 gdb Command Reference

There are many things you can do with gdb and the most common operations have been used throughout the book. Just type "h" from within gdb to get help. There are also several crib sheets online that can be helpful as a reference.

Here is a summary of the commands that have been used throughout the book: **XXX—look at my old FS book for examples**

- set print pretty on – when you print structures they will be nicely formatted. "off" will turn it off.
- set pagination off – if something is being displayed which takes up more than a page, you'll get a message to continue/quit etc. You can turn this off so everything is displayed in one go. "on" will turn it back on again.
- bt – stack backtrace. If you supply an optional number 'n', only 'n' stack frames will be displayed.
- print – print structures. You can use "p".
- break (br) – set a breakpoint
- tbreak – set a temporary breakpoint that will only be hit once and then cleared???
- c – continue running
- next (n) – similar to step but don't enter functions.
- step (s) – step through the program until it reaches a different source line.
- delete - delete a breakpoint. Without any arguments you are prompted as to whether you want to delete all breakpoints.
- clear – clear a breakpoint

- `x` – dump contents of memory in different formats
- RETURN KEY – repeat last command
- `info` –
- `add-symbol-file` – load debugging symbols from a specified file and also included where the text and other sections of the binary can be found.
- `source` – execute commands from the specified file
- `q` – quit `gdb`. You should reply "y" to have the kernel continue running.

You can abbreviate each command as long as the command is still unique. To know the extent to which you can abbreviate a command, type the first letter and hit the TAB key twice as the following example shows:

```
(gdb) l # followed by TAB-TAB
layout          lx-device-list-bus    lx-list-check
lconn          lx-device-list-class  lx-lsmod
list           lx-device-list-tree   lx-mounts
load           lx-dmesg            lx-ps
lx-clk-summary  lx-fdtdump        lx-symbols
lx-cmdline     lx-genpd-summary   lx-timerlist
lx-configdump   lx-iomem          lx-version
lx-cpus         lx-ioports
(gdb) list # followed by TAB-TAB
```

Just typing "l" followed by TAB-TAB shows multiple options but "li" followed by TAB-TAB shows only one. Therefore you are safe to use "li" in place of "list". For example you can use `p` for `print` since there are no other commands beginning with `p`.

The official `gdb` documentation can be found here:



URL 33 <https://tinyurl.com/yc49ustp>

The user manual is over 900 pages. I've found it most useful to search for *gdb cheat sheet*. You will find plenty to choose from.

9.1.5 Linux Kernel Helper Functions

There are several `gdb` built-in functions that can help with Linux kernel debugging. These commands start with `lx_` and an abbreviated list is:

```
(gdb) apropos lx
function lx_current -- Return current task.
function lx_module -- Find module by name
function lx_task_by_pid -- Find Linux task by PID
lx-cmdline -- Commandline used in the current kernel.
```

```

lx-cpus -- List CPU status arrays
lx-dmesg -- Print Linux kernel log buffer.
lx-lsmod -- List currently loaded modules.
...
lx-ps -- Dump Linux tasks.
lx-version -- Report the Linux Version of the current kernel.

```

For these scripts to be available, you will need to add the following to your `/.gdbinit` file. Note that you will need to change the path to reference your specific Linux kernel source tree.

```

add-auto-load-safe-path \
/home/spate/linux-5.19.17/scripts/gdb/vmlinux-gdb.py

define lconn
    target remote tcp:192.168.64.1:1234
end

```

Just change the path to reference the Linux source tree that you're using. Note that I've also defined a new function called `lconn` which saves typing the `"target remote ..."` line which I can never remember or wish to type that often.

NOTE – the first two lines shown should be a single line. they've been separated because the pathname doesn't fit in the width of a page! Simply combine them and remove the "\".

macros are in kernel debug book that I have on page 568

9.1.6 Basic gdb Examples

Here are some examples of using these commands that you will find helpful when analyzing file/filesystem structures.

1. Dump the process list and search for a process containing the string "spate".

```
(gdb) pipe lx-ps | grep spate
```

2. Print the contents of the `task_struct` for the process found above.

```
(gdb) p *(struct task_struct *)0xfffff0000ed8ba180
```

3. Assign the address of the `(task_struct *)` above to a convenience variable:

```
(gdb) set $ts = (struct task_struct *)0xfffff0000ed8ba180
```

4. Print out the above `task_struct` using the convenience variable:

```
(gdb) p *$ts
```

5. Running `lx-dmesg` to search for messages which is the same as running `dmesg (8)`.

When I was debugging SPFS, I wanted to look for times that I'd set the `i_private` field of the `inode` structure. I used `printk` to print them out. Obviously you can't run `dmesg` while sitting in the debugger. That's where the `lx-dmesg` command comes in useful. Knowing that messages were printed out containing "`i_private`", the `lx-dmesg` command can be run as follows:

```
(gdb) pipe lx-dmesg | grep i_private
[ 125.063346] *** set i_private to ffff0000c98dd470
[ 1200.525141] *** set i_private to ffff0000c98a6e70
[ 1510.511176] *** set i_private to ffff0000c99bd470
...
```

Sometimes it can be useful to display messages to get information (such as structure addresses) that can be used within `gdb` to display the contents of structures and so on.

9.1.7 Walking Lists

Not sure where this should go but ...

The `lx-list-check` command can be used to check a list for consistency. In the following example, we have the dentry for the root directory of the Linux tree. We display the `d_subdirs` field which is a linked list running through the child dentries' `d_child` fields. Using the same pointer, we ask `lx-list-check` to check the list for consistency.

```
(gdb) p nd->path->dentry->d_name.name
$71 = (const unsigned char *) 0xfffff888101c54878 "/"
(gdb) p nd->path->dentry->d_subdirs
$72 = {
    next = 0xfffff88800da08c90,
    prev = 0xfffff888130616150
}
(gdb) lx-list-check nd->path->dentry->d_subdirs
Starting with: {
    next = 0xfffff88800da08c90,
    prev = 0xfffff888130616150
}
list is consistent: 283 node(s)
```

XXX - build a macro?

9.1.8 Setting Breakpoints and Stepping Through Code

The `break` (or just `br`) command is used to set breakpoints. The easiest usage is to just specify the function you want to set a breakpoint on:

```
(gdb) br vfs_open
Breakpoint 6 at 0xffffffff813e250a: vfs_open. (3 locations)
```

If you want to set a breakpoint in `do_filp_open()`, but only want to hit the breakpoint if the `file` argument is called "lorem-ipsum", you can set the breakpoint as follows:

```
(gdb) br do_filp_open if $_streq(pathname->name, "lorem-ipsum")
```

If you know the PID of the process you want to trace you can use the Linux helper function to located the PID of the current process:

```
(gdb) p $lx_current()->pid
$11 = 1095
```

so could also breakpoint on "pid"

```
(gdb) br do_filp_open if $lx_current()->pid == 1095
```

The current list of breakpoints can be displayed as follows:

```
(gdb) info b
Num Type Disp Enb Address What
4 breakpoint keep y 0xffffffff813fe8c0 in do_filp_open
at fs/namei.c:3678
stop only if $_streq(pathname->name, "lorem-ipsum")
breakpoint already hit 10 times
6 breakpoint keep y <MULTIPLE>
6.1 y 0xf...a in vfs_open at fs/open.c:977
6.2 y 0xf...2 in vfs_open at fs/open.c:977
6.3 y 0xf...0 in vfs_open at fs/open.c:976
```

After the kernel has hit a breakpoint and you see the (gdb) prompt, you can move through the kernel source one line at a time, or execute multiple lines in one go, for example by having the kernel execute a function and stop on return. To execute a single line of code, type the "step" command or just "s". If the line to be executed is a function then gdb will step into the function and start executing its code and you can keep entering "s" to execute one line at a time. If you want to execute the function with one keypress, type the "next" command or just "n".

The following example shows stepping through filp_open(). The "n" command is used to skip functions until we're about to call path_openat() which we would like to enter and thus, using the "s" command.

```
(gdb) n
3679 struct nameidata nd;
(gdb) n
3683 set_nameidata(&nd, dfd, pathname, NULL);
(gdb) n
610 p->state = 0;
(gdb) n
3683 set_nameidata(&nd, dfd, pathname, NULL);
(gdb) n
3684 filp = path_openat(&nd, op, flags | LOOKUP_RCU);
(gdb) s
path_openat (nd=nd@entry=0xfffffc90000ed3d30,
op=op@entry=0xfffffc90000ed3e54, flags=flags@entry=65)
at fs/namei.c:3639
3639 {
```

Any time you're are in gdb and want to resume normal execution, type the "continue" command or just "c". The kernel will then continue running until another breakpoint is hit or you click "ctrl-c".

9.1.9 Calling `container_of()`

We have a directory that contains two files. The dentry for the directory is assigned to the convenience variable `$c2`. Let's print out the `d_subdirs` field:

```
(gdb) p $c2
$177 = (struct dentry *) 0xfffff888102eb19c0
(gdb) p $c2->d_subdirs
$178 = {
    next = 0xfffff888103050bd0,
    prev = 0xfffff8881030502d0
}
```

The linked list of children goes through the `d_child` field which is part way through the dentry structure. Using the address of `$c2` above, we can locate the address of this field:

```
(gdb) p &$c2->d_child
$180 = (struct list_head *) 0xfffff888102eb1a50
```

Grab a hex calculator and you can see that:

```
0xfffff888102eb1a50 - 0xfffff888102eb19c0 = 0x90
```

so 144 bytes from the start of the dentry structure. You certainly don't want to be doing arithmetic everything you want to see a dentry. Therefore you can use the `next` and `prev` pointers shown above and from within gdb we can call `container_of()` to get to the start of each dentry. Just specify the starting address, the type of the structure and the field within it.

```
(gdb) set $f1 = $container_of(0xfffff888103050bd0, \
                           "struct dentry", "d_child")
(gdb) set $f2 = $container_of(0xfffff8881030502d0, \
                           "struct dentry", "d_child")
```

and from here we have our dentry structures so we can display the file names:

```
(gdb) p $f1->d_iname
$160 = "hello", '\000' <repeats 26 times>
(gdb) p $f2->d_iname
$162 = "lorem-ipsum", '\000' <repeats 20 times>
```

See section 9.3.6 to see how to achieve the same using the `crash(1)` command.

9.1.10 Listing Source Code

When inside `gdb` you can view part of the source code for your program by using the `"list"` command or just `"l"`.

```
(gdb) bt
#0 link_path_walk (nd=0xfffffc90000ed3d30,
                   name=0xfffff8881009a3020 "lorem-ipsum") at fs/namei.c:2266
#1 path_openat (nd=nd@entry=0xfffffc90000ed3d30,
                op=op@entry=0xfffffc90000ed3e54, flags=flags@entry=65)
```

```

at fs/namei.c:3653
...
(gdb) list
list
2261     static int link_path_walk(const char *name,
                                struct nameidata *nd)
2262 {
2263     int depth = 0; // depth <= nd->depth
2264     int err;
2265
2266     nd->last_type = LAST_ROOT;
2267     nd->flags |= LOOKUP_PARENT;
2268     if (IS_ERR(name))
2269         return PTR_ERR(name);
2270     while (*name=='/')

```

In response `gdb` will print out source code for the lines around the current line of code to be executed. To view following lines, just press "enter". To view different lines of code, type "list [linenumber]". My preference is to have the source code visible in an editor in another window so I can see more lines of code. I can also use tag stacking to see what code is in functions about to be executed. I then know whether to use "n" or "s" to continue stepping through code.

9.1.11 Getting Symbols From Loadable Modules

If you're experimenting with kernel modules, it's frustrating to have `gdb` running, displaying structures, setting breakpoints and all the other great `gdb` features, only to find that none of them work on your loadable module. That's not to say that they won't work on your loadable module but first, you need to tell `gdb` where the module is loaded in memory.

This is something we ran into many years ago when building Linux filesystem modules when I worked at a small startup in the San Francisco Bay Area and also something we needed at VERITAS when porting VxFS, the VERITAS filesystem to Linux in the late 90s to early 2000s. Amit Kale was in our team and he was one of the people who got kgdb support into the Linux kernel around that time. He also wrote a nice script to do all of the following work for us. The script will come later but for now, this section will show what needs to be done.

The module needs to be compiled with symbol table information for `gdb`. After this, the next step is to load the module:

```
$ sudo insmod spfs.ko
```

Kernel modules, together with general application binaries, are stored in the ELF format, the *Executable and Linkable Format*, a binary format that was first introduced with SVR4 UNIX. If you're interested in details, the Wikipedia page has a very nice figure that is worth looking at.

Once upon a time, Mary Lou Nohr's book "*Unix System V: Understanding Elf Object Files and Debugging Tools*" [7] occupied space on my books shelf. At the time, I was working on an ELF loader in the SVR4 subsystem on the Chorus microkernel.

URL 34 <https://tinyurl.com/5e657c77>

Once the module is loaded we need to locate the addresses in memory of various ELF sections of the module (text, read-only data, data and BSS). This can be done as follows:

```
$ cd /sys/module/spfs/sections
$ sudo cat .text .rodata .data .bss
0xfffffffffa067b000
0xfffffffffa067d080
0xfffffffffa067e040
0xfffffffffa067ef80
```

After that, it's just a matter of providing this information to gdb. Note that if you have compiled your module on the target machine, you will need to copy the source/binaries over to the client where gdb is running:

```
(gdb) add-symbol-file /home/spate/spfs/kern/spfs.ko \
    0xfffffffffa067b000 -s .rodata 0xfffffffffa067d080 \
    -s .data 0xfffffffffa067e040 -s .bss 0xfffffffffa067ef80
add symbol table from file "/home/spate/spfs/kern/spfs.ko" at
    .text_addr = 0xfffffffffa067b000
    .rodata_addr = 0xfffffffffa067d080
    .data_addr = 0xfffffffffa067e040
    .bss_addr = 0xfffffffffa067ef80
(y or n) y
Reading symbols from /home/spate/spfs/kern/spfs.ko...
```

NOTE – jumping ahead a little here but you may find that breakpoints don't work on all functions. For example, you may set a breakpoint but the address does not look correct. And this was with correct addresses in /proc/kallsyms. It took me a while to find a solution based on the following answer to a question on StackOverflow:

The linux kernel using unlikely/likely macro to optimize the compilation, generate .text and .text.unlikely section correspondingly.

The probe function is compiled into .text.unlikely, and add-symbol-file only add .text section to the address you assigned, which is not contains the probe function.

To solve the problem, add .text.unlikely manually in the gdb shell

If you see this problem then also dump the contents of the file .text.unlikely and from within gdb call:

```
(gdb) add-symbol-file /home/spate/spfs/kern/spfs.ko \
    0xfffffffffa0673000 -s .rodata 0xfffffffffa0675080 \
    -s .data 0xfffffffffa0676040 \
    -s .bss 0xfffffffffa0676f80 \
    -s .text.unlikely 0xfffffffffa06733d8
```

XXX – it would be good to come back and explain what is happening here and look at the Linux likely/unlikely paths

Now that everything is in place, check to see that the module source code can be viewed in gdb:

```
(gdb) list sp_find_entry
23      * If found the inode number is returned.
24      */
25
26      int
27      sp_find_entry(struct inode *dip, char *name)
28      {
29          struct sp_inode_info *spi = ITOSPI(dip);
30          struct super_block    *sb = dip->i_sb;
31          struct buffer_head    *bh;
32          struct sp_dirent     *dirent;
```

The next step is to set a breakpoint. The function `spfs_fill_super()` is called when the filesystem is mounted so let's set a breakpoint there and mount a filesystem to check that the breakpoint is hit:

```
(gdb) br spfs_fill_super
Breakpoint 1 at 0xfffffffffa0674bb8:
file /home/spate/spfs/kern/sp_inode.c, line 321.
```

Next is to mount an SPFS filesystem:

```
(gdb) mount -t spfs /dev/loop7 /mnt
```

If everything is working correctly, the breakpoint will be hit and a stack backtrace can be displayed as follows:

```
#0  spfs_fill_super (sb=sb@entry=0xfffff88810adb7000,
                     data=data@entry=0x0 <fixed_percpu_data>,
                     silent=silent@entry=0)
    at /home/spate/spfs/kern/sp_inode.c:321
#1  0xffffffff813ec86e in mount_bdev (
    fs_type=fs_type@entry=0xfffffffffa0676aa0 <spfs_fs_type>,
    flags=flags@entry=0,
    dev_name=dev_name@entry=0xfffff88811ade5350 "/dev/loop7",
    data=data@entry=0x0 <fixed_percpu_data>,
    fill_super=fill_super@entry=0xfffffffffa0674bb8
    <spfs_fill_super>) at fs/super.c:1367
#2  0xfffffffffa067479d in spfs_mount (
    fs_type=0xfffffffffa0676aa0 <spfs_fs_type>, flags=0,
    dev_name=0xfffff88811ade5350 "/dev/loop7",
    data=0x0 <fixed_percpu_data>)
    at /home/spate/spfs/kern/sp_inode.c:420
...

```

Assuming you've added all ELF sections for the module, you should be able to view global variables, for example:

```
(gdb) p spfs_inode_cache
$2 = (struct kmem_cache *) 0xfffff888130798f00
```

There should be no restrictions with using `gdb` on loadable modules compared with the kernel. The only difference is if you need to debug early initialization code during module load time. There are several articles on the web that cover this topic.

Note that if you are iteratively developing your module and unloading/loading it, you will need to rerun the `add-symbol-file` command each time the module is loaded.

9.1.12 A Script to Load Modules

It's a laborious task to load a module, grab all the required ELF sections and provide that information to `gdb`. The best thing is to combine it in a script so that when you run the script, you get the following output:

```
$ modcopy
spfs.ko                      100% 177KB 10.9MB/s 00:00
add-symbol-file /home/spate/spfs/kern/spfs.ko 0xfffffffffa06aa000
-s .rodata 0xfffffffffa06ac080 -s .data 0xfffffffffa06ad040
-s .bss 0xfffffffffa06adf80 -s .text.unlikely 0xfffffffffa06aa3d8
```

Then you can just cut and paste the `add-symbol-file` line to `gdb`. It could also be written to a file and you can run the `source` command. Perhaps it could be places in `.gdbinit` but if you're constantly changing the module so unloading/loading often, that won't work too well.

Here is the script.

```
MOD=/home/spate/spfs/kern/spfs.ko
MODD=/sys/module/spfs/sections
USR=spate@192.168.64.11

scp $MOD $USR:/home/spate/module

ssh $USR sudo insmod /home/spate/module/spfs.ko

text='ssh $USR sudo cat $MODD/.text'
rodata='ssh $USR sudo cat $MODD/.rodata'
data='ssh $USR sudo cat $MODD/.data'
bss='ssh $USR sudo cat $MODD/.bss'
tunlikely='ssh $USR sudo cat $MODD/.text.unlikely'

echo "add-symbol-file /home/spate/spfs/kern/spfs.ko $text \
-s .rodata $rodata -s .data $data -s .bss $bss \
-s .text.unlikely $tunlikely"
```

Note that the last line should be one line only. It's been formatted to fit the page. Remove the backslashes and combine all three lines. Or just download the script from [XXX - github](#)

In order to get the script to work you need to be able to SSH without a password and `/etc/sudoers` on the target has been modified to prevent user "spate" from needing to type a password when running `sudo`. The following line was added:

```
spate ALL=(ALL) NOPASSWD:ALL
```

As a final note, the script doesn't check to see if the module is already loaded. If it is, you'll get a nice big warning anyway!

9.1.13 gdb User-Defined Commands

Show how to do gdb functions and put them in .gdbinit - need some good examples

There will be times when you want to develop new `gdb` functions and test them while still inside an existing `gdb` session. To achieve this, add/modify the functions in your `.gdbinit` file then run the following inside `gdb`:

```
(gdb) source /.gdbinit
```

This will execute the commands in the specified file. It will not undo or reset any commands that were in the file prior to your making changes. It will update any definitions or settings the current commands make, but it will not erase or undo any old definitions or settings.

Alternatively, you can put commands/functions in another file and run the `source` command to load/execute them from that file.

9.1.14 I Want to Edit the Command History Using vi

Just as with editing the command line history in the shell, I like to do the same in `gdb`. To do this, simply add "set editing-mode vi" in the file `.inputrc` in your home directory. Emacs users can use "set editing-mode emacs" although `gdb` operates in emacs mode by default. You can also type "esc-ctrl-j" from within `gdb` to enable `vi` mode.

9.2 Exploring File/Filesystem Information with eBPF

We've covered several tools for exploring file/filesystem related information up to this point. Another method is to use eBPF, the enhanced Berkeley Packet Filter technology that has been in the Linux kernel since 2014.

You'll find everything you need to know about eBPF on Brendan Gregg's website and I recommend his book "*BPF Performance Tools*".



URL 35 <https://tinyurl.com/yc2y7kv>

To use `bpftrace` you'll need to install it first:

```
$ sudo apt install bpftrace
```

Once installed you'll have access to `bpftrace` you will see about 35 scripts in `/usr/sbin/*.bt`. There are 35 different scripts here. Those related to filesystems are:

- `opensnoop.bt` – trace open (2) system calls displaying the filenames that are being opened.
- `vfscount.bt` – count the different VFS calls. **XXX - what does it define as a VFS call?**
- `vfsstat.bt` – count some VFS calls, with per-second summaries. **some?**
- `writeback.bt` – trace file system writeback events with details.???

at time of writing these fail on Ubuntu 22.04 and 22.10 - but I have had them working in the past. Revisit

9.2.1 Tracing Kernel Functions with eBPF

When implementing symlinks in SPFS, I had no end of issues and couldn't see why my symlink-related operations were not getting called. The obvious method was to use `kgdb` and step through the kernel code but that required a lot of work I hadn't started at that time. I'd been looking at eBPF for another project so knew that it had the ability to trace kernel functions and look at return values. Therefore, I could follow the path where I walked through the kernel code, seeing which functions were getting called and which ones were not. The first step was to setup a simple script that put a kernel probe (**XXX**) on the function I wanted to see whether it got called and if it did, print out the return value. Here is the script:

<https://peteralmgren.com/tracing-linux-kernel-bpftrace/>

TBD

```
#include <linux/path.h>
#include <linux/dcache.h>

kretprobe:vfs_readlink /pid == 11756/
{
    printf("returned: %d\n", retval);
}
```

Before running it, I had a small program which tried to read the symlink:

```
#include <stdio.h>
#include <unistd.h>

int
main()
{
    char    buf[256];
    size_t  sz;
```

```

pid_t pid;

pid = getpid();
printf("pid = %d\n", pid);
sleep(20);

sz = readlink("/mnt/bar", buf, sizeof(buf));
printf("sz from readlink = %ld\n", sz);
}

```

I then run the program, take note of the PID displayed, quickly modify my script and then run as follows:

```
# bpftrace ./test/watch.bt
Attaching 1 probe...
```

I can see that the probe is attached but nothing gets displayed after the program completes `sleep()` and calls `readlink()`. This tells me that the kernel `vfs_readlink()` function never gets called.

9.2.2 Warning

XXX—be very careful when attaching ops to different file types (inodes). I had `page_get_link` in inode operations that were attached to a regular file. I couldn't figure out why I was getting a panic in `page_get_link()` when dealing with a regular file. The stack trace was:

```
[ 198.626623] kernel BUG at fs/namei.c:5027!
[ 198.627489] Internal error: Oops - BUG: 0 [#1] SMP
...
[ 198.646732] page_get_link+0x130/0x140
[ 198.646875] pick_link+0x340/0x400
[ 198.646996] step_into+0x290/0x3b0
[ 198.647115] open_last_lookup+0xc4/0x41c
[ 198.647256] path_openat+0x90/0x2c4
[ 198.647379] do_filp_open+0xb0/0x184
```

`nohighmem` was not set for IFREG

9.3 Debugging With The `crash(8)` Command

The `crash(1)` command has been one of my favorite tools for exploring UNIX kernels since the early 1990s and I used it extensively for describing the SCO UNIX kernel in my first book [10]. The Linux version has the same feel as the older UNIX variants but has been greatly enhanced by the integration of `crash` with `gdb`.

First of all, here's a little history of `crash` provided by David Anderson:

The Linux operating system originally lacked a built-in, traditional UNIX-like kernel crash dump facility. This was initially addressed by the Mission Critical Linux Mcore kernel patch and the LKCD (Linux Kernel Crash Dump) kernel

patch from SGI in 1999, and later by the Red Hat Netdump facility in 2002, and the Red hat Diskdump facility in 2004. The upstream Linux community finally settled upon the adoption of the Kdump crash dump facility in 2006.

However, the creation of a kernel crash dump file is only half of the picture; a utility is required to be able to recognize the dumpfile format in order to read it, and to offer a useful set of commands to make sense of it.

Furthermore, to examine the contents of a live system's kernel internals from user space, the only readily available option has been to use gdb on /proc/kcore. While gdb is an incredibly powerful tool, it is designed to debug user programs, and is not at all "kernel-aware". Consequently, using gdb alone has limited usefulness when looking at kernel memory, essentially constrained to the printing of kernel data structures if the vmlinux file was built with the -g C flag, the disassembly of kernel text, and raw data dumps. Furthermore, distributions such as Red Hat Enterprise Linux have limited the access to /proc/kcore, making it unusable as a kernel memory source.

As far as kernel crash dump files are concerned, the Red Hat Netdump and uncompressed Diskdump facilities, and the Kdump facility create dump files that are readable by gdb, but aside from giving it the capability of displaying the panicking task's stack trace, it has the same constraints as when reading /proc/kcore. However, gdb cannot read LKCD, Mcore, Xen or s390/s390x dump files.

That being the state of things, the crash utility was developed as a convenient means to cover all bases, i.e., all listed dumpfile formats as well as live systems. Moreover, it is also designed to be easily enhanced to suit the specific needs of the kernel developers or analysts using it; the builtin command set can easily be extended or enhanced, and external command modules may be written and dynamically attached.

This section will be focusing on using `crash` to analyze a live system but if you have need to debug a Linux crash dump, the `crash` command will be invaluable.

9.3.1 Installing `crash` and Linux Debugging Information

Installing `crash` itself is very straightforward but having the necessary debugging information available can take a little more work and isn't always straightforward on different versions of each distribution. Here are the list of steps that were followed for Ubuntu 22.04. I suggest looking at **XXX - my website** - I will endeavor to provide information for different distributions and for different versions for each distribution. There is nothing more frustrating than trying to use a tool for learning and then to find that the tool can't be installed or at least not without a lot of effort and searching.

The first step is to **XXX what? Why didn't I write down instructions in the first place?**. Note that some lines are truncated so you will need to go to XXX for complete instructions.

```
# apt-key adv --keyserver keyserver.ubuntu.com \
--recv-keys C8CAB6595FDFF622
# codename=$(lsb_release -c | awk 'print $2')
# sudo tee /etc/apt/sources.list.d/ddebs.list << EOF
%deb http://ddebs.ubuntu.com/ ${codename} main ...
%deb http://ddebs.ubuntu.com/ ${codename}-security main ...
%deb http://ddebs.ubuntu.com/ ${codename}-updates main ...
%deb http://ddebs.ubuntu.com/ ${codename}-proposed main ...
EOF
# apt-get update
# apt-get install linux-image-$(uname -r)-dbgsym
```

On Ubuntu 22.04 you need to XXX

```
$ echo "deb http://ddebs.ubuntu.com $(lsb_release -cs) main \
restricted universe multiverse" | sudo tee -a \
/etc/apt/sources.list.d/ddebs.list

$ echo "deb http://ddebs.ubuntu.com $(lsb_release -cs)-updates \
main restricted universe multiverse" | sudo tee -a \
/etc/apt/sources.list.d/ddebs.list
```

and then XXX

```
$ sudo apt-get update # refresh repository info - will fail
# first time and display C8CAB6595FDFF622]

- fix cert issue: - the numbers will be displayed

$ sudo apt-key adv --keyserver keyserver.ubuntu.com \
--recv-keys C8CAB6595FDFF622

- refresh repository info again

$ sudo apt-key adv --keyserver keyserver.ubuntu.com \
--recv-keys C8CAB6595FDFF622
```

Install the debug symbols package

```
$ sudo apt-get install linux-image-$(uname -r)-dbgsym
```

The final step is to install `crash` itself:

```
$ sudo apt install crash
```

All of the above will take up a lot of space so make sure you have at least **XXX GB of disk space available. You also need to install gcc/make etc if you want to monitor your own programs**

9.3.2 Upgrading the Kernel

Your kernel will get automatically upgraded from time to time so the debug version that you use with `crash` will no longer work. For example, as will be shown soon, `crash` is launched against ...

```
$ sudo crash /usr/lib/debug/boot/vmlinux-5.15.0-58-generic
...
WARNING: /usr/lib/debug/boot/vmlinux-5.15.0-58-generic
        and /proc/version do not match!

WARNING: /proc/version indicates kernel version: 5.15.0-60-generic

crash: please use the vmlinux file for that kernel version, or
      try using the System.map for that kernel version as an
      additional argument.
```

In this case, the correct debug symbols package is needed. You can follow the procedure in section 9.1.2 to disable automatic updates or you can upgrade your debug symbols package. If that is the path you choose, you can use the following command to install the package for the currently running kernel:

```
$ sudo apt-get install linux-image-$(uname -r)-dbgsym
```

you can then relaunch `crash` using the correct version:

```
$ sudo crash /usr/lib/debug/boot/vmlinux-5.15.0-60-generic
```

or simply:

```
$ sudo crash /usr/lib/debug/boot/vmlinux-$(uname -r)
```

At this point, `crash` should work as expected.

9.3.3 Kernel Compilation Issues

My kernel had been compiled to have split files (???). When running `crash` I got the following error:

```
Dwarf Error: wrong unit_type in compilation unit header
(is DW_UT_split_compile (0x05), should be DW_UT_type
(0x02)) [in module /home/spate/linux-5.19.17/init/main.dwo]

crash: vmlinux: no debugging data available
```

No idea how to fix this.

9.3.4 Running `crash`

Now that everything is installed you run `crash` on a live system by specifying a single argument containing the path to the debugging version of the kernel. This will differ based on which distribution you are using. IBM has a nice page that covers everything that is needed to install `crash` on RHEL, SLES and Ubuntu. For specific distributions, documentation can be spotty and may not always work. At the time of writing I couldn't get `crash` to work on Ubuntu 22.10 so went back to 22.02.

For Ubuntu the path is as follows. Once inside, "?" displays commands available. There is quite a bit of information displayed on first entry.



URL 36 – <https://tinyurl.com/mrx4d9k8>

```
$ sudo crash /usr/lib/debug/boot/vmlinux-5.15.0-58-generic

crash 8.0.0
Copyright (C) 2002-2021 Red Hat, Inc.
Copyright (C) 2004, 2005, 2006, 2010 IBM Corporation
Copyright (C) 1999-2006 Hewlett-Packard Co
...
For help, type "help".
Type "apropos word" to search for commands related to "word"...

        KERNEL: /usr/lib/debug/boot/vmlinux-5.15.0-58-generic
        DUMPFILE: /proc/kcore
...
crash> ?

*      extend      log      rd      task
alias   files       mach    repeat   timer
ascii   foreach     mod     runq     tree
bpf    fuser       mount   search   union
bt     gdb         net     set      vm
btop   help        p       sig     vtop
dev    ipcs       ps      struct  waitq
dis    irq        pte     swap    whatis
eval   kmem       ptob   sym     wr
exit   list        ptov   sys     q

crash version: 8.0.0      gdb version: 10.2
For help on any command above, enter "help <command>".
For help on input options, enter "help input".
For help on output options, enter "help output".
```

Since many examples throughout the book look at analyzing structures of processes that are currently running, let's start with find a process using the `ps` command and displaying a few fields from the `task_struct`:

Since many examples throughout the book look at analyzing structures of processes that are currently running, let's start with find a process using the `ps` command and displaying a few fields from the `task_struct`: Since many examples throughout the book look at analyzing structures of processes that are currently running, let's start with find a process using the `ps` command and displaying a few fields from the `task_struct`:

```
crash> ps | head -3
  PID PPID CPU      TASK          ST %MEM  VSZ  RSS COMM
    0    0   0  ffffbaa703da88c0  RU  0.0    0    0 [swapper/0]
    0    0   1  ffff5a09c0304ec0  RU  0.0    0    0 [swapper/1]
```

```
crash> ps | grep spate
    7703 5214      3  ffff5a09d824af40 IN    0.0 2056 968  open-spate
crash> task ffff5a09d824af40
PID: 7703. TASK: ffff5a09d824af40 CPU: 3. COMMAND: "open-spate"
struct task_struct {
    .....
    pid = 7703,
    .....
    fs = 0xffff5a09d0764740,
    files = 0xffff5a09c08f7c80,
}
```

The `files` field is of type `files_struct` so can be displayed using the `struct` command or in most cases by just typing the name of the structure and giving the address. So both of these commands produce the same result:

```
crash> struct files_struct 0xffff5a09c08f7c80
crash> files_struct 0xffff5a09c08f7c80
```

Memory can be displayed using the `rd` command. This example displays an array of pointers:

```
crash> rd 0xffff5a09d4292800 4
ffff5a09d4292800: ffff5a09d97a1200 ffff5a09d97a1200
ffff5a09d4292810: ffff5a09d97a1200 ffff5a09b2bf7200
```

XXX—check on the above to see what other formats there are.

There is interaction between `crash` and `gdb` so you can do things like:

```
crash> p file_systems
file_systems = $1 = (struct file_system_type *)0xfffffbbaa703f3bd50
                <sysfs_fs_type>
```

The `sys` command displays information about the running system:

```
crash> sys
    KERNEL: /usr/lib/debug/boot/vmlinux-5.15.0-58-generic
    DUMPFILE: /proc/kcore
    CPUS: 4
    DATE: Wed Feb 1 00:05:14 UTC 2023
    UPTIME: 10:24:44
    LOAD AVERAGE: 0.06, 0.02, 0.00
    TASKS: 175
    NODENAME: 2204-crash
    RELEASE: 5.15.0-58-generic
    VERSION: #64-Ubuntu SMP Thu Jan 5 12:06:43 UTC 2023
    MACHINE: aarch64 (unknown Mhz)
    MEMORY: 4 GB
```

Many similar commands aren't necessarily useful if you're using `crash` or `gdb` for looking at running kernels for educational purposes but come in very useful when debugging core dumps to understand what was running at the time of a panic.

There is a reasonable amount of information on the web about `crash`. Igor Ljubuncic (aka Dedoimedo) has written a 182 page book about debugging with `crash` which you can find here:



URL 37 – <https://tinyurl.com/363bbmf>

It was written over ten years ago but much of the material is still relevant. **XXX – it's mostly about crash dumps and has little analysis. Find better material**

9.3.5 Exploring Lists With `crash`

TBD - store the /proc stuff in SPFS with lists that can be seen in crash. Not needed - just walk the list in the SB or the inode list

The `crash` command has the ability to walk Linux kernel linked lists and display all elements in a list. For example, each `super_block` structure has a list of VFS inodes for this mount point in a doubly linked list as the following shows:

```
crash> super_block 0xfffff5a09d2fc000
...
s_inodes = {
    next = 0xfffff5a09d3ffb950,
    prev = 0xfffff5a09d3ff86c8
},
...
...
```

Using the head of the list (shown in bold), the `list` command is called to display all entries. The highlighted address will be used in the next section.

```
crash> list 0xfffff5a09c1f27318
fffff5a09d3ffb950
fffff5a09c0ef8ff8
fffff5a09c0eff070
fffff5a09197f79a0
fffff5a09d3ffa6f0
fffff5a09d3ff9dc0
fffff5a09d3ff86c8
fffff5a09d2fc0548
```

Section 7.4.1 provides an example of walking inode and dentry lists using `gdb` and `crash`.

9.3.6 Equivalent of `container_of()` With `crash`

Section 5.5 describes how the `container_of()` inline function is used to located a structure given an element within the containing structure. In the example above, we are displaying linked list information in an inode structure, correspond to this field:

```
struct inode {  
    ..  
    struct list_head    i_sb_list;  
    ..  
}
```

We can print out the contents of the corresponding `inode` structure as follows.

```
crash> inode -l inode.i_sb_list ffff5a09c0ef8ff8
```

The address passed as an argument is the address highlighted in bold above.

See section 9.1.9 to see how to achieve the same using the `gdb`.

9.3.7 Exploring Trees With `crash`

TBD - "help tree" to look at something other than rbtrees?

XXX—come back here when I've played more and figured out what's useful - specifically "list" and "tree"

9.4 Conclusion

If you want to program in the Linux kernel, you need to get good at debugging. It used to be much harder than it is now as there are so many good tools available and there's a good chance someone has already hit a problem you've run into, asked on line and got several good responses.

This chapter showed how to build a kernel with kgdb support and produce symbols such that the kernel can be debugged remotely using `gdb`. If you're serious about learning the Linux kernel, this is the best method to follow so you can set breakpoints and view structures in the kernel in real time. It's also worth becoming familiar with what `gdb` can do so try as many examples as you can. Try making some changes to SPFS, load the module, set breakpoints and see your changes in action. It's quite a bit of work to get there but is very rewarding.

The integration of `gdb` with `crash(1)` has produced another powerful debugging tool extending the `crash` command which has been available in UNIX for several decades. Both are used throughout the book to explore kernel structures and set breakpoints to understand various code paths.

But `gdb` is only one such tool so explore what's possible with eBPF and **XXX—TBD**

Chapter 10

Filesystem Security

It's all about the data! Whether hackers infiltrate a network and encrypt data for ransomware or whether data is stolen for its value (secrets, credit card numbers etc), there are an endless number of attacks against systems, often with the goal of getting to the data at the heart of an organization.

This chapter explores various security aspects surrounding the data that filesystems store. Encryption and key management are discussed in detail together with the standards that demand their deployment and govern their use. Some unusual topics like how to exploit previously allocated filesystem blocks to potentially locate sensitive data will be described together with fixes that may or may not be applied depending on the filesystem and performance characteristics needed.

Two of the main Linux security subsystems, SELinux and AppArmor will be highlighted, specifically as both apply to filesystem access.

10.1 It Starts With File and Directory Permissions

We've already discussed how the standard UNIX file permissions model is supposed to work. Let's discuss its weaknesses and what can be done to alleviate these issues.

find issues - setuid bit etc etc - NFS?

10.2 Encryption and Key Management

Our encrypting FUSE filesystem was a fun exercise and although file names and data are encrypted on disk, it is not a secure system. If someone got hold of the embedded encryption key, they have access to data. Most encryption systems use published encryption algorithms so if you have the key and know the encryption algorithm, you will be able to decrypt the data.

Furthermore, in our example of encryption in FUSE, we used a *hard-coded encryption key* which is not at all secure. One key for all mounted filesystems and everywhere? Traditional laptop encryption used a password scheme where the encryption key is derived

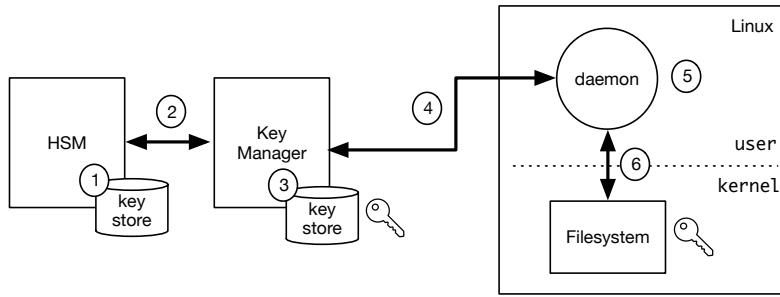


Figure 10.1: Components in an enterprise encryption solution

from the password. We could enhance the FUSE solution to do something similar but most enterprise encryption solutions require operating in a hands-off mode.

Figure 10.1 shows the components that may be used in an enterprise key management solution for providing filesystem encryption for Linux servers.

Here are the paths followed for key management:

1. An HSM may be used to generate the key(s) used to encrypt filesystem data. These keys are called DEKs (Data Encryption Keys). They may be generated on demand from the key manager (KM). Alternatively, the KM may generate its own keys using a local random source (either software or hardware).
2. Whether a DEK or a wrapping key, the key manager will request such keys from the HSM.
3. A wrapping key may be used to unwrap (decrypt) one or more keys that are used to encrypt a key store on the KM.
4. DEKs are sent to the encryption agent over TLS and typically with an additional layer of *wrapping* (extra layer of encryption). This will usually be in response to an event such as mounting a filesystem.
5. The DEK is delivered to a filesystem module. There is likely additional wrapping between the daemon and the filesystem module.
6. The DEK will be held in memory and used to encrypt / decrypt data as it passes to / from disk. The key may be protected in memory when not in use but will be decrypted (in the clear) in memory when in use.

Many commercially available, enterprise encryption solutions will have a KM deployed but many will not use an HSM. There is certainly a cost factor involved but the extra level of integration is typically only done for the most secure environments.

10.2.1 Password-based Encryption

UNIX passwords used to reside in the file /etc/passwd which each line looked like the following:

```
root:Ep6mckrOLChF:0:1:System Operator::/bin/ksh
daemon:*:1:1::/tmp:
uucp:POE9oN9NE:4:4::/var/spool/uucppublic:/usr/lib/uucp/uucico
dennis:eH5/.jN7B3mdc:181:100:Dennis Ritchie:/u/dennis:/bin/ksh
ken:fwkf5j1Tif4o.:182:100:Ken Thompson:/u/ken:/bin/csh
```

The odd thing was that this file was readable by everyone. I had great fun in the late 1980s by running the `crack` program on this file and showing our system administrator what the root password was. Each password was hashed using `crypt(3)` so all the `crack` program needed to do was brute force the password by trying one guess after another.

I can't find crack anywhere. sshcrack perhaps?. Nowadays john the ripper does something similar

The 1979 paper *Password Security: A Case History* by Robert Morris and Ken Thompson is a very interesting read as they discuss issues with the original password scheme.



URL 38 <https://tinyurl.com/2je3dd9e>

As a side note, I was super happy to find out a few years ago that Ken Thompson is still going strong and was one of the creators of Golang, a new language developed by Thompson together with Rob Pike and Robert Griesemer, all of whom now work for and Google.

There was a move away from using /etc/passwd to store passwords in the mid 1980s but took time to roll out and be adopted by other vendors and of course Linux.

10.2.2 Encryption Algorithms and Key Sizes

There are many different encryption algorithms available today but by far, the most commonly used encryption algorithm used for file and disk encryption is AES, the *Advanced Encryption Standard* with typical key sizes of 128 and 256 bytes. Most encryption systems have 256-byte keys as default. The next sessions discusses Intel hardware support for AES for key sizes and 128, 192 and 256 bits.

CBC, XTS, ESSIV etc ...

10.2.3 Hardware Acceleration of AES

Crypto is time-consuming especially when using software algorithms. When standing in front of customers, the question of performance would always come up and we would always talk around it and show the best numbers we had. Of course, if a databases has an extensive cache, the overhead is less depending on the benchmark. But software-based encryption generally has a high overhead and performance can be dreadful.

There have been many hardware based solutions over the years typically some form of card that is plugged into the motherboard. HSMs can be used to offload some crypto functions but not for file I/O operations. The trip over the write would add an unbearable overhead.

AES is by far the most popular algorithm used for symmetric encryption. Intel introduced hardware support for AES with the introduction of AES-NI (Advanced Encryption Standard New Instructions) in 2008. It is an extension to the x86 instruction set architecture and has been enhanced many times over successive generations of Intel processors. Furthermore, the instructions are supported by the guest operating systems of all major hypervisor vendors. Support for AES-NI was added to the Linux kernel many years ago and can be used by any component inside the kernel.

For implementing encryption at the filesystem level, the performance gains are huge. How much gain is obtained is very depending on the type of I/O that applications are performing (as with all performance tests) but have reduced the encryption overhead to *noise* rather than having a significant impact when using only software-based encryption.

10.2.4 Access Controls

UNIX/Linux has provided permissions at the file-level since UNIX first made its appearing in the late 1960s. While the user/group/other level of permissions is fine for general usage, it's possible for privileged users to be able to gain access to data for general users.

Encryption and enterprise key management, which will be covered in the next section, goes a long way to protect sensitive data but concerns still exist around ensuring only the right users and applications are accessing the decrypted data. This is particularly problematic on Linux since the root administrator can access non-root owned data. Security frameworks such as SELinux which will be described in section 10.5 can help with preventing root from accessing data but the implementation is quite cumbersome. There are commercial products on the market such as Thales CTE (Ciphertrust Transparent Encryption), the old Vormetric products I worked on, which provide additional access controls that have capabilities such as:

- Prevent root users from accessing specified paths under any circumstance.
- Zero-downtime encryption (for encrypting data that already exists).
- Only allow specific binaries to access data and ensure that these binaries haven't been tampered with through use of checksums.

The encryption suite comes with FIPS 140-2 validated key management and highly-secure HSMs (Hardware Security Modules).

10.2.5 EKM – Enterprise Key Management and FIPS

When selling key management into federal government and highly-regulated industries such as banking, obtaining FIPS 140-2/3 certification is a must.

It is generally a requirement for key managers to be at least FIPS 140-2 Level 1 certified since many of the key managers available today are run as virtual machines. Level 1 is

generally the highest level of certification for a software component. The component that completes FIPS certification is called the *cryptographic module*. This could be a single program or could be the whole operating system shipped to customers in different formats but could be a virtual appliance.

With level 1, at least one *approved* cryptographic algorithm must be validated. This is achieved by running *Known Answer Tests* (KAT) against the algorithm where the inputs and outputs are known ahead of testing. KAT are used to validate the algorithm in order to obtain a valid certificate for the algorithm and also performed during system initialization to make sure nothing has been tampered with and the algorithm still performs as expected. There is also a lot of scrutiny around which random number generator is used since that is the seed used for key generation. There are also other aspects of level 1 including taking checksums of components in the module and validating the components against the checksums on system initialization.

Above level 1, physical hardware must be used since level 2 above require tamper detection through hardware mechanisms and if an attempt is made to break into a physical FIPS appliance, keys will be shredded rendering the appliance useless.

FIPS 140-2 only validates the module once development is completed and yes, changes to reach certification, may result in a new version of the product being produced. At the time of writing, FIPS 140-2 is coming to an end and being replaced by FIPS 140-3 requires evaluation of security requirements during all stages of the cryptographic module creation including design, implementation, and final operational deployment.

All modules certified by NIST have a *Security Policy* that describes the module including algorithms certificates and all security policies can be found on the NIST website. I've gone through this process several times. An example of one of my security policies can be located on the NIST website here:



39 <https://tinyurl.com/3vzee5xe>

You can see the algorithms validated and a link to the security policy.

10.2.6 Common Criteria

Most encryption and key management vendors cringe when the topic of Common Criteria is brought up by customers. FIPS 140-2 can be a costly and time-consuming exercise but nothing when it comes to the Common Criteria (CC) standard.

Where the FIPS component being evaluated is called the cryptographic module, in CC, it's the *Target of Evaluation* (TOE). CC comprises several different *Protection Profiles* (PP) and it's often not a trivial task to determine which CC a solution will fit under.

It will be left as an exercise to the reader to discover more about CC although I suspect most will only do so if required.

10.2.7 HSMs – Hardware Security Modules

Providing the ultimate secure storage of secrets (encryption keys, certificates and so on), HSMs have been around for many years and have higher-levels of FIPS compliance since they will generally come in a highly secured physical appliance that is tamper proof.

There are also specialized HSMs for different industries including payment HSMs which provide high levels of protection for cryptographic keys and customer PINs used during the issuance of magnetic stripe and EMV (Europay, Mastercard, and Visa) chip cards (and their mobile application equivalents) and the subsequent processing of credit and debit card payment transactions.

The functions of an HSM are:

- Creation of crypto keys.
- Secure storage for crypto keys. Note that not all crypto keys are stored in the HSM but generally speaking, the most sensitive keys such as *master keys* and *wrapping keys* are typically stored in an HSM.
- Key management functions including digital signature functions, offloading application servers for complete asymmetric and symmetric cryptography.
- Manage transparent data encryption keys for databases and keys for storage devices such as disk or tape.

In the latter case, an HSM may be used to create and manage database keys but for filesystems and disks, a key manager will typically be used since it generally has more context about the keys. In this case, a higher-level key that comes from the HSM may be used for the key store inside a key manager.

HSMs are typically hardware appliances that are tamper proof including tamper detection and shredding of keys if seals are broken, wires tripped etc. One way of doing this is to use a physical HSM card inside the box that is wired to the chassis. If the chassis is tampered with (an attempt to open it), the wire will be pulled and keys will be shredded.

They also have specialized hardware for key generation, crypto algorithms etc. But even with high-performing crypto acceleration, HSMs are not used for inline encryption of data through filesystems, the disk layer or for databases. It would be too much of a hit on performance to ship data back and forth across the wire between say a Linux server and an HSM. Instead, HSMs are often used in conjunction with a key manager (KM) where the KM may hold encrypted keys a filesystem might use but these keys are wrapped (encrypted themselves) with a key that is generated in the HSM and stored in the HSM.

10.2.8 Protecting Keys Over the Wire

For step 4 in figure 10.1, an encryption key will come out of secure storage and be transported to the server where it will be used to encrypt/decrypt files and devices. TLS (Transport Layer Security) which was built on the now-deprecated SSL (Secure Sockets Layer) is used as a base layer of security by providing encryption of data over the wire. Various

standards including Common Criteria will dictate the minimum version of TLS to use to avoid known vulnerabilities in earlier versions.

But TLS is not enough by itself and is frowned upon in security circles if that is the only method used to transport encryption keys. Some standards provide secure key exchange protocols and some encryption vendors also go beyond such standards with some rather elaborate schemes. In all cases asymmetric encryption is used to assist with the key exchange.

The Diffie–Hellman key exchange protocol is one such method of providing secure exchange of keys between two parties. If you like mathematics, you are welcome to read about the algorithm on Wikipedia or view the NIST special publication here:



40 <https://tinyurl.com/2r93e6s5>

10.2.9 Protecting Encryption Keys in Memory

The previous sections have covered how keys can be securely generated, transported and stored *at rest* (on disk storage for example) with hardware support and standardized protocols to ensure their safety. But with all of this, a key needs to be present when encryption/decryption takes place and will therefore be present in memory (in the clear). Some encryption products provided additional layers of security by unwrapping (decrypting) keys in memory only when needed and then wrapping (encrypting) them once the crypto operation has finished. But there is still a window when the keys are in clear-text and vulnerable to attack.

I recall many years ago when we demonstrated issues with encryption inside a VM to VMware and showed that, when we took a snapshot of the VM, we pulled the key out of the snapshot file. The issue is still there today.

This has been a major source of frustration for me for the last 15 years. There have been so many advances in memory protection by hardware vendors such as Intel but software support has been sorely lacking. And more specifically, we have been living in a virtualized world for many years but the hypervisor vendors have been slow to adopt some of these technologies. Furthermore, the hardware vendors have been designing in a vacuum with little consideration as to how such features will work through the hypervisor. Take VM live migration as an example. If CPUs are different XXX

Looking at some of the available technologies that can help with memory protection:

- **Intel SGX** — Intel Software Guard Extensions (SGX) allow applications to have protected private regions of memory, called enclaves. Enclaves are areas of memory that are encrypted which prevent them from being read or examined by other programs. While promising, early versions of SGX had very limited space for enclaves and the technology only applies to user-level applications so it can't be deployed for filesystem or disk-level encryption.

- **Intel Total Memory Encryption (TME)** — this provides encryption of memory which protects against physical attacks to exfiltrate data. The CPU and RAM modules communicate over a bus on the motherboard, it's possible to tap into this data bus allowing an attacker to read the contents of memory.
- **AMD Secure Encrypted Virtualization (SEV)** — Like Intel TME, SEV provides encryption of all of the memory of a virtual machine (VM) using a key per VM.

While such technologies don't directly help with encryption of filesystem data, their main use is in protection of memory where encryption keys are held. With some encryption technologies, keys themselves are wrapped (encrypted) in memory and only decrypted when in use but that still leaves a window when the key is visible.

10.2.10 NIST Standards

The United States National Institute of Standards and Technology (NIST) has many standards for encryption and key management including a set of public facing documents call special publications. Each special publication that has been written to address and support the security and privacy needs of U.S. Federal Government information and information systems is prefixed with "SP 800". Just run a web search for "nist sp 800" and you should see the following website:



41 <https://csrc.nist.gov/publications/sp800>

There are a lot of documents on this page. Some examples include:

- SP 800-12 – An Introduction to Information Security
- SP 800-133 – Recommendation for Cryptographic Key Generation
- SP 800-131A – Transitioning the Use of Cryptographic Algorithms and Key Lengths

10.2.11 Keep it Simple — Encrypt it Yourself

If you wish to transport encrypted files to a friend, family member or colleague, there are many commercial solutions available which are well beyond the scope of this book but there are open-source tools that allow you to achieve this with the `openssl` (SSL) command being one of the top choices. For information on how to do this, I recommend the following article:



42 <https://tinyurl.com/3b9d3t3y>

As with most things encryption related, the mathematics is complicated but most of the engineers who implement encryption solutions rarely have to delve that deep. Most encryption algorithms have publicly available implementations and most languages and support for hardware-based encryption is available everywhere including OpenSSL and inside the Linux kernel.

10.3 Zero Blocks / Extents on Allocation or Free

Data allocated to a file is in blocks or extents which are general multiples of the disk sector size. To remove a file, each of these blocks/extents must be removed from the filesystem inode and added back to the free list. Generally speaking these blocks are not zeroed which is a timely operation. This means that when such blocks are allocated to a new file, the old contents remain and are potentially visible by the new owner.

Generally speaking, most applications won't see this stale data. If you create a file and write a file as follows:

```
int
main()
{
    char *buf = "hello world\n";
    int fd = open("/mnt/foo", O_CREAT|O_WRONLY, 0700);

    write(fd, buf, strlen(buf));
}
```

You will see a file of the size determined by the amount of data written (12 bytes):

```
$ ls -l foo
-rw----- 1 spate spate 12 Dec 22 16:55 foo
```

If you try to read beyond the file, you will get an error XXX. Furthermore, the data in the block is zeroed by the kernel so you won't see old data. **What about mmap?????? - need to find where the kernel zeroes blocks.**

But a problem can exist. To demonstrate this, I'll use the disk-based SPFS which does not zero blocks that have been deallocated and reused. I have a file with 200 lines of fake credit card names and numbers.

```
American Express, Scott Davis, 47289...2804, 03/2026, CID: 3379
American Express, Hailey Jones, 34357...6656, 03/2026, CID: 6275
American Express, Maria Miller, 34784...6300, 03/2026, CID: 1391
...
...
```

and simply copy the file into an SPFS-mounted filesystem:

```
# mount -t spfs /dev/sda /mnt
# cp ./test/credit-card-nos /mnt
# umount /mnt
```

Inside the SPFS fsdb we can locate the file, the allocated blocks inside the file and then read the contents of each block:

```
spfsdb > i4
inode number 4
  i_mode      = 81a4
  i_nlink     = 1
  i_atime     = Wed Dec 21 18:06:19 2022
  i_mtime     = Wed Dec 21 18:06:19 2022
  i_ctime     = Wed Dec 21 18:06:19 2022
  i_uid       = 0
  i_gid       = 0
  i_size      = 13774
  i_blocks    = 7
  i_addr[ 0] = 131   i_addr[ 1] = 132   i_addr[ 2] = 133
  i_addr[ 3] = 134   i_addr[ 4] = 135   i_addr[ 5] = 136
  i_addr[ 6] = 137
spfsdb > b131
0000  416d 6572 6963 616e 2045 7870 7265 7373 American Express
0020  3437 3238 3339 3838 3837 3238 3034 2c20 47283988872804,
0040  3739 0a41 6d65 7269 6361 6e20 4578 7072 79.American Expr
0060  732c 2033 3433 3537 3436 3936 3133 3536 s, 3435746961356
0080  3a20 3632 3735 0a41 6d65 7269 6361 6e20 : 6275.American
00a0  696c 6c65 722c 2033 3437 3834 3831 3831 iller, 347848181
....
```

This alone presents a security issue to many given that the root administrator can access sensitive data through the device interface.

But let's get back to the example in hand. We're going to remove this file and create a new file with the following program:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int
main(int argc, char *argv[])
{
    char *buf = "hello world\n";
    int fd = open("/mnt/foo", O_CREAT|O_WRONLY, 0700);

    lseek(fd, 2030, SEEK_SET);
    write(fd, buf, strlen(buf));
}
```

Let's give it a run and see what happens:

```
# mount -t spfs /dev/sda /mnt
# rm /mnt/credit-card-nos
# ./seeker
# cat /mnt/foo
American Express, Scott Davis, 3472...72804, 03/2026, CID: 3379
American Express, Hailey Jones, 3435...35656, 03/2026, CID: 6275
```

```
American Express, Maria Miller, 3478...46300, 03/2026, CID: 1391
...
American Express, Bradley Henry, 3768...5450, 03/2026, CID: 5787
American Express, Frank Robinsonhello world
```

Voila! There are some of the contents of the old file. Our "hello world" string is there where we wrote it (highlighted in bold).

You would not see the same thing if we run the following:

```
# cp ./test/credit-card-nos /mnt
# rm /mnt/credit-card-nos
# echo hello > /mnt/foo
# cat /mnt/foo
hello
```

and looking at the allocated block using fsdb:

```
spfsdb > b131
0000 6865 6c6c 6f0a 0000 0000 0000 0000 0000 hello.....
0020 0000 0000 0000 0000 0000 0000 0000 0000 .....
0040 0000 0000 0000 0000 0000 0000 0000 0000 .....
```

XXX—need to say why. Page allocated and zeroed

10.3.1 Fixing the Issue

There are several ways to fix this issue or a fix may not be needed at all if there is no sensitive data in the filesystem. The first way is to zero a block every time it's allocated. If you look at `sp_block_alloc()` it simply looks through the in-core SPFS superblock to see if a block is available. Here is the snippet of code with the check highlighted:

```
mutex_lock(&sbi->s_lock);
for (i = 1 ; i < SP_MAXBLOCKS ; i++) {
    if (sbi->s_block[i] == SP_BLOCK_FREE)
        sbi->s_block[i] = SP_BLOCK_INUSE;
        sbi->s_nbfree--;
        mutex_unlock(&sbi->s_lock);
        return SP_FIRST_DATA_BLOCK + i;
}
```

There is no need to go to disk at this point although the superblock will be marked dirty and flushed to disk at some point. If we wish to zeroize the block, we don't want to read it from disk so we should call `getblk`, zero the block and mark it as dirty so it will be flushed at some point. Hmm! Won't work with the page cache or there may be issues. Need to look into this.

Also show quick fix in SPFS by zeroing blocks when freed

10.3.2 A DIY Solution

The `wipe(1)` command is an interesting option. The manpage gives reasons as to why taking matters into your own hands may be a good solution, especially when decommissioning disk drives. It highlights issues with newer filesystems where data may temporarily be stored in the filesystem journal (something we'll be covering in Volume 2) and even if deletion, how it's possible to recover data that was previously erased. And they state that the best way to sanitize a storage medium is to subject it to temperatures exceeding 1500K. I doubt anyone has a BBQ that can go to those temperatures. And furthermore, I have to quote this from the manpage:

I strongly recommend to call wipe directly on the corresponding block device with the appropriate options. However THIS IS AN EXTREMELY DANGEROUS THING TO DO. Be sure to be sober.

There are several options available for the command and it repeatedly writes over the data:

... wipe repeatedly overwrites special patterns to the files to be destroyed, using the `fsync()` call and/or the `O_SYNC` bit to force disk access. In normal mode, 34 patterns are used (of which 8 are random)

Here is the `wipe(1)` running to overwrite the contents of the credit card file:

```
$ wipe /mnt/dir/credit-card-nos
Okay to WIPE 1 regular file ? (Yes/No) Yes
Wiping /mnt/dir/credit-card-nos, pass 34 (34)
Operation finished.
1 file wiped and 0 special files ignored in 0 directories,
0 symlinks removed but not followed, 0 errors occurred.
```

Running the SPFS `fsdb` command the contents of what was the first block of the file are displayed. This and all other blocks now have random data throughout.

```
spfsdb > b131
0000 047f 5e0c fc36 96ab 049f e200 7727 708f ..^..6.....w'p.
0020 54aa aff9 c1c9 84b1 d6ac 9632 7e92 d01f T.....2~...
0040 ac26 c946 b034 90d8 50ef ed21 87a6 6e78 .&.F.4..P...!..nx
```

It's safe to say that at this point, the old data has ceased to be.

If I was that concerned about the security of data I would encrypt the filesystem or disk, add appropriate access controls and make sure I had a good key management strategy. I recall a conversation with a Wall Street CISO several years ago. His CTO was concerned about encryption overheard. The CSO responded by asking how much it would cost for CPUs with slightly greater performance to compensate. If you want to be secure, don't be cheap! Data Breaches are very costly.

We added a block zeroize capability to VxFS many years ago in response to customer requests. This still left the problem of regular file data being present at times in the intent log. We'll come back to this topic on Volume 2.

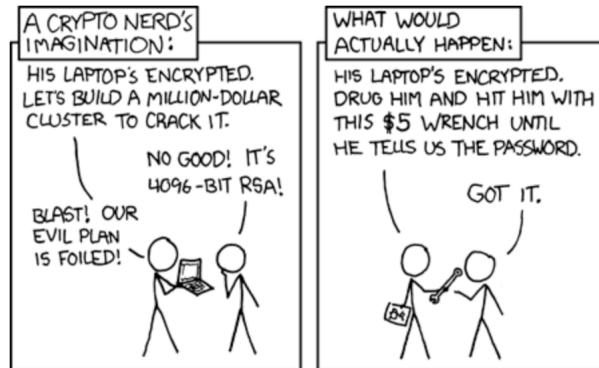


Figure 10.2: How good is your security?

10.4 Security is a Multi-Layered Approach

I remember many years when I first heard my friend and former colleague Mike Fleck talking about the M&M model of security. *Hard on the outside and soft on the inside*. In other words, an organization builds an impenetrable shell to the organization's infrastructure, taking the view "*No one gets in unless we let them*". But once you're on (or allowed to be in), you generally have free rein throughout the organization's infrastructure. This always reminds me of figure 10.2.

Security vendors and customers are becoming more interested in the *Zero Trust* model of security whereby they assume that they are operating in an environment where there is no traditional network edge. In other words, the M&M model of security no longer applies.

In the zero trust model, trust is never granted implicitly but must be continually evaluated. In other words "*never trust, always verify*". Zero Trust was designed to protect modern environments and enable digital transformation by using strong authentication methods, leveraging network segmentation, preventing lateral movement, providing Layer 7 threat prevention, and simplifying granular, "least access" policies.

The NIST Special Publication *Zero Trust Architecture* (SP 800-207) is being adopted by some security vendors. It can be used as a base to understand the zero trust model and then see what the security vendors say in terms of their products. For many of them, they have multiple products which don't work well, or at all, together. A zero trust model should bring these products together to operate as a whole and give enterprise-level visibility and security.

10.5 SELinux

Main benefits - su detection
out of the box still for RHEL as well as OpenShift.

lots of customers disable it straightaway

Fedora Core 2 adopted SELinux back in 2004. It was then added to RHEL (Red Hat Enterprise Linux) in 2005 although the aims were for maximum ease of use and therefore wasn't as restrictive as it could be. It was added to Ubuntu 8.04 (2008) and was in OpenSUSE 11.1 as a technical preview in XXX?

SE Linux and labeling files.

10.6 AppArmor

TBD

10.7 SELinux vs AppArmor

TBD

In general, SELinux is a more complex technology than AppArmor in that it controls more operations and *separates containers by default*???. It's not possible to get this level of control using AppArmor due to the lack of MCS. In addition, not having MLS means that AppArmor cannot be used in highly secure environments. <- rewrite and understand.

from wikipedia

SELinux represents one of several possible approaches to the problem of restricting the actions that installed software can take. Another popular alternative is called AppArmor and is available on SUSE Linux Enterprise Server (SLES), openSUSE, and Debian-based platforms. AppArmor was developed as a component to the now-defunct Immunix Linux platform. Because AppArmor and SELinux differ radically from one another, they form distinct alternatives for software control. Whereas SELinux re-invents certain concepts to provide access to a more expressive set of policy choices, AppArmor was designed to be simple by extending the same administrative semantics used for DAC up to the mandatory access control level.

There are several key differences:

One important difference is that AppArmor identifies file system objects by path name instead of inode. This means that, for example, a file that is inaccessible may become accessible under AppArmor when a hard link is created to it, while SELinux would deny access through the newly created hard link.

As a result, AppArmor can be said not to be a type enforcement system, as files are not assigned a type; instead, they are merely referenced in a configuration file.

SELinux and AppArmor also differ significantly in how they are administered and how they integrate into the system.[34] Since it endeavors to recreate traditional DAC controls with MAC-level enforcement, AppArmor's set of operations is also considerably smaller than those available under most SELinux implementations. For example, AppArmor's set of operations consist of: read,

write, append, execute, lock, and link.[35] Most SELinux implementations will support numbers of operations orders of magnitude more than that. For example, SELinux will usually support those same permissions, but also includes controls for mknod, binding to network sockets, implicit use of POSIX capabilities, loading and unloading kernel modules, various means of accessing shared memory, etc.

There are no controls in AppArmor for categorically bounding POSIX capabilities. Since the current implementation of capabilities contains no notion of a subject for the operation (only the actor and the operation) it is usually the job of the MAC layer to prevent privileged operations on files outside the actor's enforced realm of control (i.e. "Sandbox"). AppArmor can prevent its own policy from being altered, and prevent file systems from being mounted/unmounted, but does nothing to prevent users from stepping outside their approved realms of control.

For example, it may be deemed beneficial for help desk employees to change ownership or permissions on certain files even if they don't own them (for example, on a departmental file share). The administrator does not want to give the user(s) root access on the box so they give them CAP_FOWNER or CAP_DAC_OVERRIDE. Under SELinux the administrator (or platform vendor) can configure SELinux to deny all capabilities to otherwise unconfined users, then create confined domains for the employee to be able to transition into after logging in, one that can exercise those capabilities, but only upon files of the appropriate type.[citation needed] There is no notion of multilevel security with AppArmor, thus there is no hard BLP or Biba enforcement available.[citation needed]. AppArmor configuration is done using solely regular flat files. SELinux (by default in most implementations) uses a combination of flat files (used by administrators and developers to write human readable policy before it's compiled) and extended attributes. SELinux supports the concept of a "remote policy server" (configurable via /etc/selinux/semanage.conf) as an alternative source for policy configuration. Central management of AppArmor is usually complicated considerably since administrators must decide between configuration deployment tools being run as root (to allow policy updates) or configured manually on each server.

10.8 Security Standards

There are many standards out there that deal with security. I suspect that readers will have heard of some but certainly not all. For example, we all hear about HIPPA (the *Health Insurance Portability and Accountability Act*) as we come across it each time we visit the doctor, dentist or optometrist. If you deal with government organizations you'll certainly have heard of FIPS. Of course GDPR (General Data Protection Regulation) and its California equivalent CCPA (California Consumer Protection Act) together with huge fines that organizations are having to pay are often in the news

In the banking space, the Payment Card Industry PCI DSS (*Data Security Standard*) have a set of guidelines that must be followed by vendors who are storing credit cards. You may have seen the term PCI 2.0.

At the operating system level, there are a lot of different standards. On the Red Hat website, this is what they list that they support just for the government space including FIPS and CC which were discussed earlier.

1. COMMON CRITERIA
2. FIPS 140-2 and FIPS 140-3
3. Secure Technical Implementation Guidelines (STIG)
4. Criminal Justice Information Services (CJIS)
5. US Government Configuration Baseline (USGCB)
6. USGv6-r1 TESTED PRODUCT LIST
7. USGv6 TESTED PRODUCT LIST
8. SECTION 508
9. US ARMY CERTIFICATE OF NETWORTHINESS
10. FISMA
11. FedRAMP
12. NISPOM CHAPTER 8
13. HIPAA Overview

Not all are filesystem-related but many many have some implications on filesystem activity. Most people working on Linux, including the kernel may not come across many of these standards. It largely depends on the field in which you work. For engineers working on products in the security space, many will be familiar. Encryption and key management plays a re large part in several of these standards, some times as recommendations (HIPPA) and sometimes as mandates (PCI).

10.9 Secure Programming

Need to think about whether to include this or not.

10.10 Conclusion

Security is a large topic and anyone who has been to the RSA show in San Francisco held in winter each year will know how big a field this is and how much it has grown. While it is not always imperative for engineers to be aware of what standards are out there, it's important to have some level of awareness of the issues and what can be done to protect against vulnerabilities or attackers. Hopefully this chapter provided insight into the most important topics.

Chapter 11

Filesystem Performance

Performance is a fascinating but complex subject with a lot of "depends on" as part of the conversation. Since file activity is such a critical part of making a system *perform* there have been many changes to the kernel over several decades to get the best performance possible. In addition to the kernel, filesystems and underlying storage subsystems have made huge advances

Whole books have been written about operating system performance. This chapter can only go so far but will explain areas of the kernel that affect performance, cover performance studies of different filesystems and describes the tools available that can help analyze performance bottlenecks.

Brendan Gregg's book *Systems Performance – Enterprise and the Cloud* [2] is XXX

11.1 Performance of Different Filesystems

Linux supports over 80 different filesystem types although less than 10 of them could be considered high-performance filesystems. Which one to use is going to depend both on the type of I/O that the application is performing as well as the underlying storage subsystem.

This section describes the performance characteristics of the major filesystems referencing various studies that have taken place over recent years.

11.1.1 The Phoronix Study

The Phoronix study in 2019 tested the performance of btrfs, ext4, XFS and F2FS on aSATA 3.0 solid-state drive, USB SSD, and an NVMe SSD.

The conclusion of the tests were that F2FS and XFS are usually fastest on the consumer solid-state storage devices with ext4 not too far behind. Btrfs has a lot of features but at least in its out-of-the-box behavior generally being a fair amount slower than EXT4/F2FS/XFS. Perhaps most interesting from today's results were the startup-time application results where the Flash-Friendly File-System easily won across all of those tests.
XXX—needs to be rewritten - copied from page 4

You can find the study here:



URL 43 https://tinyurl.com/4yu4zef8

-

11.2 Mount Options

TBD

11.3 Block Sizes

TBD

11.4 Extend-based Allocation

TBD

11.5 The Effects of Filesystem Fragmentation

TBD

11.6 Open Flags etc etc

TBD - depends on which FS

11.7 Volume Management / Disk Subsystems

things have changed a lot - cloud???

SSDs vs ...

11.8 Buffer Cache and Page Cache Tuning

free command to show buf/cache usage

11.9 Other commands???

11.10 The sysstat Package

The sysstat package contains various utilities, common to many commercial Unixes, to monitor system performance and usage activity:



URL 44 https://tinyurl.com/5auz66fj

Here are the standalone commands that are part of the package. I'm the description text from the website

- **iostat** – reports CPU statistics and input/output statistics for block devices and partitions.
- **mpstat** – reports individual or combined processor related statistics.
- **pidstat** – reports statistics for Linux tasks (processes) : I/O, CPU, memory, etc.
- **tapestat** – reports statistics for tape drives connected to the system.
- **cifsiostat** – reports CIFS statistics.

There are also tools in the package that can be scheduled via `cron` or `systemd` to collect and historize performance and activity data:

- **sar** – collects, reports and saves system activity information (see below a list of metrics collected by sar).
- **sadc** – is the system activity data collector, used as a backend for sar.
- **sal** – collects and stores binary data in the system activity daily data file. It is a front end to sadc designed to be run from cron or systemd.
- **sa2** – writes a summarized daily activity report. It is a front end to sar designed to be run from cron or systemd.
- **sadf** – displays data collected by sar in multiple formats (CSV, XML, JSON, etc.) and can be used for data exchange with other programs. This command can also be used to draw graphs for the various activities collected by sar using SVG (Scalable Vector Graphics) format.

The default sampling interval for these tools is 10 minutes but this value can be changed and be as low as 1 second.

After installing the package, if you try to run one of these tools you may see:

```
$ sudo sar
[sudo] password for spate:
Cannot open /var/log/sysstat/sa26: No such file or directory
Please check if data collecting is enabled
```

To enable collection of data you need to edit the file `/etc/default/sysstat` and change `ENABLED="false"` to `ENABLED="true"` and then run:

```
$ sudo service sysstat restart
```

11.11 eBPF – a New World of Opportunities

The *Berkeley Packet Filter (BPF)* is a technology that originated in BSD UNIX in 1992 for programs that need to, among other things, analyze network traffic. It provided a raw interface to data link layers in a protocol independent fashion. All packets on the network, even those destined for other hosts, are accessible through this mechanism.

eBPF, the *extended Berkeley Packet Filter*, is a technology. It has a lot of promise as performance expert Brendan Gregg has been quoted as saying.

"super powers have finally come to Linux" – Brendan Gregg

Gregg's book *BPF Performance Tools* [3] provides in-depth information about eBPF covering everything from XXX to YYY. There is also a significant amount of information in his website:

eBPF is a technology that allows programs to be run in the kernel in a sandboxed environment. It does this XXX

best description here - <https://ebpf.io/what-is-ebpf/>



URL 45 – <https://tinyurl.com/4hvpc35j>



URL 46 – <https://tinyurl.com/yc2y7kvy>

Some aspects of eBPF have already been covered including how it can be used to enhance FUSE filesystem performance (section 9.2.1) and debugging to understand whether specific Linux kernel functions are invoked or not (section 8.9.1).

Figure 11.1 shows how it all works XXX

11.11.1 BCC – BPF Compiler Collection

TBD

The source code and information about BCC can be found on github here:



URL 47 – <https://tinyurl.com/fmfe8498>

There are many examples shown at the bottom of the home page including tracing a kernel functions and printing all kernel stack traces and VFS read latency distribution (**XXX—explain or remove**).

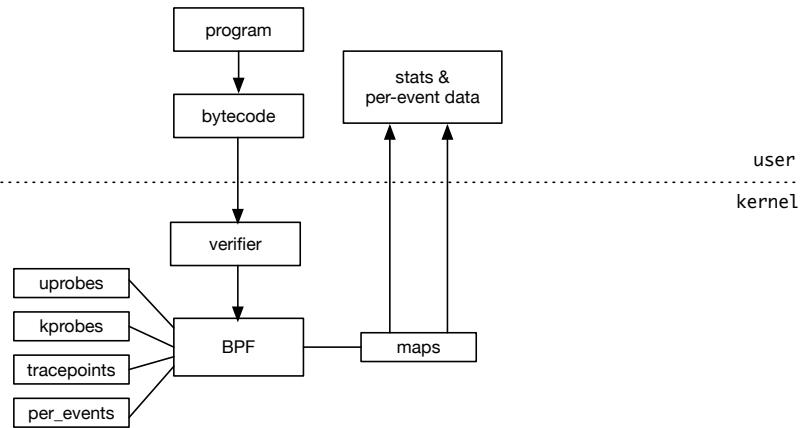


Figure 11.1: eBPF Flow - XXX

11.11.2 The `bpftrace` Command

BPFtrace is a high level tracing language which is built on top of LLVM and BCC. BPFtrace provides an easier way of writing BPF programs for interacting with Linux tracing capabilities, such as kprobes, uprobes, kernel tracepoints, USDT and hardware events.

`bpftrace` is part of the `iovisor` project. Source code and information can be found here:



URL 48 – <https://tinyurl.com/4hvpc35j>

```
$ sudo apt install bpftrace
```

A great example to show the power of eBPF is with the following example that uses kprobes to XXX

```
#include <linux/path.h>
#include <linux/dcache.h>

kprobe:vfs_open
{
    printf("Path: %s\n",
           str(((struct path *)arg0)->dentry->d_name.name));
}
```

The program is run as follows:

```
$ sudo bpftrace open-path.bt
Attaching 1 probe...
```

```
Path: cgroup
Path: cat
Path: ld-linux-aarch64.so.1
Path: ld.so.cache
Path: libc.so.6
Path: locale-archive
Path: open-path.bt
...
```

There will be a lot of output even on what seems like a Linux system with little activity. The first and last entry shown corresponded to running:

```
$ cat /home/spate/open-path.bt
```

So very powerful XXX

11.11.3 The Annual eBPF summit

Each year there is an eBPF summit which is free to attend and covers many aspects of eBPF including new innovations. The summit started in 2020 so at the time of writing, there have been three conferences to date. Their website can be found here:



URL 49 – <https://ebpf.io>

You can watch all of the videos and download slides for many of them. Example talks from the 2022 summit are:

- The future of eBPF in the Linux Kernel – vision from Alexei Starovoitov who is the co-creator and co-maintainer of eBPF.
- eBPF - The Capital Market's Perspective – VCs are showing interest in companies developing eBPF-based technology.
- XRP: In-Kernel Storage Functions with eBPF

If you're interested in learning more about eBPF, these short videos can help you gain a good understanding of the technology and its applications.

11.12 Conclusion

TBD

Chapter 12

Filesystem Case Studies

Chapter xx provided information about features of the main Linux filesystems. This chapter goes a level deeper by describing the internal operation of some of the more important features of each of the most widely used filesystems in the kernel today.

It's not possible to cover everything and some features make good blog posts but topics like ext4 journaling and XXX are popular features and will be described in more detail. I have endeavored to choose interesting topics that I think people would find interesting but have not already been covered. For the latter case, I'll provide references. But for popular and well-documented topics such as ext4 journaling, I will provide a different view and, as always, a way to get as hands-on as possible and see for yourself how it works.

For each filesystem, if the contents of on-disk structures for super-blocks, inodes, journals and so on were listed in the book, that would add another hundred pages or more and not be particularly useful. But it's also hard to understand each filesystem without having the information at hand. **xxx — need to put it on my website somewhere or reference it if there is good info out there**

A forensics tool would be nice to display a bunch of stuff. If it could do multiple filesystems, that would be great

12.1 The Extended Filesystems

It's hard to ignore the *Extended Filesystem* which has had a rich and varied history in Linux from the introduction of ext2 in 1993 through to today's journaling ext4.

12.1.1 The ext4 Disk Layout

XXX

Extents

xxx — Another major feature in Ext4 is the use of extents rather than the previously described block mapping method shown in Fig. 2. Extents are more efficient at map-

ping data blocks of large contiguous files as their structure generally consists of the address of the first physical data block followed by a length.

xxx

12.1.2 ext4 Superblock

The ext4 superblock is just under 1024 bytes and contains a lot of fields.

There are several tools to help with analyzing ext4 filesystems:

- dumpe2fs –
- e2fsck –
- tune2fs –
- debugfs –
- mactime – take a look???

xxx

The The Sleuth Kit (TSK) is a set of useful tools that is used to facilitate the forensic analysis of computer systems. It supports ext2, ext3 and ext4.

12.1.3 ext4 Reserved Inodes

The following inodes are reserved by ext4:

- 0 – not used.
- 1 – list of defective blocks.
- 2 – root directory.
- 3 – user quota.
- 4 – group quota.
- 5 – boot loader.
- 6 – undelete directory.
- 7 – reserved group descriptors inode. ("resize inode")
- 8 – journal inode.
- 9 – "exclude" inode, for snapshots(?)
- 10 – replica inode, used for some non-upstream feature?
- 11 – traditional first non-reserved inode. Usually this is the `lost+found` directory.
See `s_first_ino` in the superblock

xxx

12.1.4 Exploring on-disk ext4 Data Structures

hard to do all but show how fsdb works so people can look with confidence.

12.1.5 ext4 Journaling (jbd2)

The ext4 filesystem implements a *journal* to protect the filesystem against corruption in the case of a system crash. The journal is small and continuous region of disk reserved inside the filesystem as a place to land "important" data writes on-disk as quickly as possible <- copied. The default size of the journal is 128 MB.

The following is interesting but copied from the web.

```
# dumpe2fs /dev/sda2 | grep -i journal
Journal inode: 8
Journal backup: inode blocks
Journal features: journal_incompat_revoke
Journal size: 32M
Journal length: 8192
Journal sequence: 0x00000662
Journal start: 1
```

According to the official documentation:

There are 3 different data modes:

- **writeback mode** – In data=writeback mode, ext4 does not journal data at all. This mode provides a similar level of journaling as that of XFS, JFS, and ReiserFS in its default mode - metadata journaling. A crash+recovery can cause incorrect data to appear in files which were written shortly before the crash. This mode will typically provide the best ext4 performance.
- **ordered mode** – In data=ordered mode, ext4 only officially journals metadata, but it logically groups metadata information related to data changes with the data blocks into a single unit called a transaction. When it's time to write the new metadata out to disk, the associated data blocks are written first. In general, this mode performs slightly slower than writeback but significantly faster than journal mode.
- **journal mode** – data=journal mode provides full data and metadata journaling. All new data is written to the journal first, and then to its final location. In the event of a crash, the journal can be replayed, bringing both data and metadata into a consistent state. This mode is the slowest except when data needs to be read from and written to disk at the same time where it outperforms all others modes. Enabling this mode will disable delayed allocation and O_DIRECT support.

xxx
xxx



The XXX header file can be found at `fs/ext4/ext4_jbd2.h` and also at `include/linux/jbd2.h` — XXX they are different!!!

12.1.6 Analyzing the ext4 Journal

There's a packet in debian called sleuthkit, in which you have some tools like `jls` or `jcat`. The `jls` tool can list all journal entries of an ext4 file system, for example:

```
# jls grafi.img

JBlk      Description
0:        Superblock (seq: 0)
sb version: 4
sb version: 4
sb feature_compat flags 0x00000000
sb feature_incompat flags 0x00000000
sb feature_ro_incompat flags 0x00000000
1:        Allocated Descriptor Block (seq: 2)
2:        Allocated FS Block 161
3:        Allocated Commit Block (seq: 2, sec: 1448889478.49360128)
...
```

Need to figure out what we can show

12.2 XFS

There are approximately 77,000 LOC in `fs/xfs` with the remainder of the tools on

12.2.1 The XFS Disk Layout

XXX

Each Allocation Group is divided into four different structures:

- Superblock
- List of free blocks
- Information on allocated and free inodes
- Blocks allocated for extension of B-trees

XXX

The XFS superblock can be seen by ... figure 12.2 - i like this. find a way to incorporate some of this with tools for getting to all the main components.

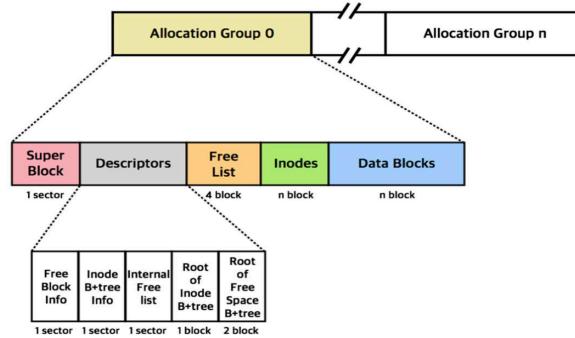


Figure 12.1: XFS Allocation Groups

Offset(h)	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	
00000000	58 46 53 42 00 00 10 00 00 00 00 00 03 E8 00	XFSB.....è.
00000010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000020	7D B8 68 B2 C5 64 49 D1 88 13 79 46 E1 9F D8 D3),h*ÀdIN.yFaYØO
00000030	00 00 00 00 00 02 00 04 00 00 00 00 00 00 00 40ø
00000040	00 00 00 00 00 00 00 41 00 00 00 00 00 00 00 42A.....B
00000050	00 00 00 01 00 00 FA 00 00 00 00 04 00 00 00 00ü.....
00000060	00 00 03 57 E4 B5 02 00 02 00 00 08 00 00 00 00 00W'ü.....
00000070	00 00 00 00 00 00 00 00 00 0C 09 09 03 10 00 00 19@.....;
00000080	00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 3BA'.....
00000090	00 00 00 00 00 03 C4 91 00 00 00 00 00 00 00 00A'.....
000000A0	FF	yyyyyyyyyyyyyyyy
000000B0	00 00 00 00 00 00 04 00 00 00 00 00 00 00 00 00
000000C0	00 00 00 00 00 00 01 00 00 01 8A 00 00 01 8A§..§
000000D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00

Figure 12.2: XFS Superblock

12.2.2 XFS Journaling

XXX

COPIED — From the information recorded in superblock, the first block of journal can be reached. Log record begins with a 512 bytes header to document general information of this log record, and it starts with a magic number “0xfeedbabe” to help make sure at the start location of this log record.

XXX

12.2.3 Analyzing the XFS Journal

Need to figure out what we can show

12.3 The btrfs Filesystems

TBD

12.3.1 The btrfs Disk Layout

XXX

12.3.2 btrfs COW vs Journaling

XXX

12.4 NFS

NFS has gone through several different versions. Rather than dig deep on older versions, this section will focus on version 4 (v4) and make references to earlier versions as needed.

There are about 80,000 LOC in `fs/nfs` so more than ext4's 60,000 LOC and about half as much code as btrfs 140,000 LOC.

First of all, need a good architecture diagram - can also refer to one in fs chapter.

12.4.1 Standardization and Protocols

XXX

12.4.2 Client-side NFS Handling

XXX

12.4.3 Server-side NFS Handling

XXX

12.4.4 NFS Locking

XXX

12.4.5 NFS File Delegation

XXX

12.4.6 Other NFS Implementations

XXX

12.5 Conclusion

This chapter described some of the major filesystems in more detail covering topics such as ext4 journaling, NFS client and server side implementation and newer filesystems such as btrfs. It would be possible to write short books on each filesystem so I've been very selective in choosing what to present with the goals of being able to get hands-on to understand the material or to be able to located where relevant information is on the web.

XXX

Chapter 13

Filesystem Forensics

Don't know what to add or whether it's worth it?
there are articles online about recovery

13.1 Conclusion

TBD

Bibliography

- [1] Callaghan, Brent (1999) NFS Illustrated, Addison Wesley Professional Computing Series
- [2] Gregg, Brendan (2020) Systems Performance – Enterprise and the Cloud, Addison Wesley Professional Computing Series
- [3] Gregg, Brendan (2019) – BPF Performance Tools, Addison Wesley Professional Computing Series
- [4] Kernighan, Brian W. and Pike, R. (1984) – The UNIX Programming Language, Prentice Hall Professional Technical Reference
- [5] Kernighan, Brian W. and Ritchie, Dennis M. (1988) – The C Programming Language, 2nd Edition, Prentice Hall Professional Technical Reference
- [6] Lions, J (1976) – A Commentary on the UNIX Operating System, Peer to Peer Communications/ Annabook
- [7] Nohr, Mary Lou (1994) – Unix System V: Understanding Elf Object Files and Debugging Tools, Prentice Hall Open Systems Library
- [8] McKenney, Paul E. (2004) – Scaling deache with RCU, Linux Journal
- [9] Pate, Steve D. (2003) – UNIX Filesystems: Evolution, Design, and Implementation, Wiley
- [10] Pate, Steve D. (1996) – UNIX Internals: A Practical Approach, Addison Wesley
- [11] Von Hagen, William (2002) – Linux Filesystems, Sams

Index

address_space_operations, xxx
Async I/O xxx
 example of, xxx
 kernel implementation, xxx
Backup and restore, xxx
Boot process, xxx
 run levels, xxx
bpftrace, see eBPF
Buffer cache
 flushing, xxx
 tuning, xxx-xxx
Buffered I/O, xxx
Caching
 Buffer cache, xxx
 Directory cache, xxx
 Inode cache, xxx
 Page cache, xxx
crash command, xxx
 installing, xxx
 example, xxx
 example, xxx
 example, xxx
 example, xxx
dentry, xxx
 definition, xxx
 interfaces, xxx
 implementation, xxx
Direct I/O, xxx
Directories, xxx
 reading, xxx
 kernel implementation, xxx
 SPFS implementation, xxx
Disks
 partitioning, xxx
 GPT, xxx
 MBR, xxx
 volume management, xxx
 performance, xxx
eBPF, xxx
 bpftrace, xxx
 installing, xxx
 tracking kernel functions, xxx
errno handling, xxx
Extended attributes, xxx
file
 descriptors, xxx
 allocation of, xxx
 hierarchy, xxx
 locking, xxx
 table, xxx
 types, xxx
 undelete, xxx
File descriptor, xxx
 allocation of, xxx
 table, xxx
File open flags, xxx
 open(2) system call, xxx
 fcntl(2) system call, xxx

File table, xxx
 Allocation of, xxx
 file_table structure, xxx
 KGDB example, xxx
 file structure, xxx
 FILE structure, xxx
 file_operations, xxx

Filesystem
 block sizes, xxx
 comparison between filesystems, xxx
 different types of, xxx
 ext2 / ext3 / ext4, xxx
 extents, xxx
 FS-Cache, xxx
 hierarchy, xxx
 jfs, xxx
 mounting, xxx
 NFS, xxx
 proc, xxx
 ramfs, xxx
 SPFS, see "SPFS"
 statfs(2), xxx
 kernel implementaiton, xxx
 sync, xxx
 sysfs, xxx
 tmpfs, xxx
 unmounting, xx
 VxFS, xxx
 XFS, xxx

FS-Cache, xxx

fsck command, xxx
 SPFS, xxx

fsdb, xxx
 ext4, xxx
 SPFS, xxx

FUSE, xxx
 architecture, xxx
 installing, xxx
 examples of, xxx

encryption, xxx
 performance with eBPF, xxx

gdb, xxx
 building a kernel with kdgb support, xxx
 reference guide, xxx

HSM applications, xxx

Inode
 Incore vs disk, xxx
 inode structure, xxx
 inode_operations, xxx
 SPFS, xxx
 inode_operations, xxx

ioctl, xxx

KGDB examples, xxx
 analyzing dentries, xxx
 mounting filesystems, xxx
 pathname resolution, xxx
 Per-File Kernel Structures, xxx

indexspace

Kernel
 vs user-space, xxx
 source code walkthrough, xxx
 nubmer of LOC, xxx

Kernel Tools
 crash, xxx
 Elixir source code reference, xxx
 vi tag stacking, xxx

ls command implementation, xxx

LSB (Linux Standards Base), xxx

Locking, xxx
 file record locking, xxx
 file-private locking, xxx
 kernel implementation, xxx
 mandatory locks, xxx

Manpages, xxx

Mandatory file locking, xxx

Memory mapped files, xxx
example of, xxx
user interfaces, xxx
kernel implementation, xxx

mmap, see Memory mapped files

Mounting filesystems, xxx
mount command, xxx
kernel implementation, xxx

Multi-threading, xxx

Named pipes, xxx

NFS, xxx

Page cache, xxx
Compound pages, xxx
Folios, xxx
Process address space, xxx
SunOS implementation, xxx

Pathname resolution, xxx

Performance, xxx
async I/O, see Async I/O
block sizes, xxx
different filesystems, xxx
direct I/O, see direct I/O
eBPF tools, xxx
sysstat package, xxx

Pipes, xxx
named pipes, xxx

POSIX, xxx
threads, xxx

Pseudo filesystems, xxx
examples of, see filesystems
proc, xxx
Ramfs, xxx
sysfs, xxx

Quotas, xxx
example of using quotas, xxx

Rename, xxx
kernel implementation, xxx

Security, chapter, xxx
Common Criteria, xxx
Encryption, xxx
FIPS, xxx
Key management, xxx
File permissions, xxx

Sparse files, xxx
example of, xxx
virtual machines, xxx
QCOW, xxx
swap files, xxx

Superblock
super_block structure, xxx
SPFS handling of, xxx

SPFS filesystem, xxx
chapter, xxx
create, xxx
fsdb, xxx
rename, xxx
undelete, xxx

Standard I/O library, xxx
source code overview, xxx
standard file descriptors, xxx
performance of, xxx

super_operations, xxx

Synchronous writes, xxx

Systems calls, xxx
source code, xxx
vs library functions, xxx

tags for vi etc, xxx

Vectored reads/writes, xxx

vi, xxx
tag stacking, xxx
the -t option, xxx

VFS interface, xxx

vim, see vi

Writing, xxx

Synchronous writes, xxx

xattr, see Extended attributes