

 **Steve French** smb3: add dynamic trace points for tree disconnect ... History

 16 contributors  +4

1048 lines (934 sloc) | 30.3 KB

1

// SPDX-License-Identifier: GPL-2.0-or-later

2

/*

3

*

4

* Copyright (C) International Business Machines Corp., 2000,2005

5

*

6

* Modified by Steve French (sfrench@us.ibm.com)

7

*/

8

#include <linux/fs.h>

9

#include <linux/string.h>

10

#include <linux/ctype.h>

11

#include <linux/module.h>

12

#include <linux/proc_fs.h>

13

#include <linux/uaccess.h>

14

#include "cifspdu.h"

15

#include "cifsglob.h"

16

#include "cifsproto.h"

17

#include "cifs_debug.h"

18

#include "cifsfs.h"

19

#include "fs_context.h"

20


#ifdef CONFIG_CIFS_DFS_UPCALL


21

#include "dfs_cache.h"

22

#endif

 **JoePerches** Follow

 Committed to this repository

```
30 {
31     pr_debug("%s: dump of %d bytes of data at 0x%p\n", label, length, data);
32     print_hex_dump(KERN_DEBUG, "", DUMP_PREFIX_OFFSET, 16, 4,
33                     data, length, true);
34 }
35
36 void cifs_dump_detail(void *buf, struct TCP_Server_Info *server)
37 {
38     #ifdef CONFIG_CIFS_DEBUG2
39         struct smb_hdr *smb = buf;
40
41         cifs_dbg(VFS, "Cmd: %d Err: 0x%x Flags: 0x%x Flgs2: 0x%x Mid: %d Pid: %d\n",
42                 smb->Command, smb->Status.CifsError,
43                 smb->Flags, smb->Flags2, smb->Mid, smb->Pid);
44         cifs_dbg(VFS, "smb buf %p len %u\n", smb,
45                 server->ops->calc_smb_size(smb));
46     #endif /* CONFIG_CIFS_DEBUG2 */
47 }
48
49 void cifs_dump_mids(struct TCP_Server_Info *server)
50 {
51     #ifdef CONFIG_CIFS_DEBUG2
52         struct mid_q_entry *mid_entry;
53
54         if (server == NULL)
55             return;
56
57         cifs_dbg(VFS, "Dump pending requests:\n");
58         spin_lock(&server->mid_lock);
59         list_for_each_entry(mid_entry, &server->pending_mid_q, qhead) {
60             cifs_dbg(VFS, "State: %d Cmd: %d Pid: %d Cbdata: %p Mid %llu\n",
61                     mid_entry->mid_state,
62                     le16_to_cpu(mid_entry->command),
63                     mid_entry->pid,
64                     mid_entry->callback_data,
65                     mid_entry->mid);
66             #ifdef CONFIG_CIFS_STATS2
67                 cifs_dbg(VFS, "IsLarge: %d buf: %p time rcv: %ld now: %ld\n",
68                         mid_entry->large_buf,
69                         mid_entry->resp_buf,
70                         mid_entry->when_received,
71                         jiffies);
72             #endif /* STATS2 */
73             cifs_dbg(VFS, "IsMult: %d IsEnd: %d\n",
74                     mid_entry->multiRsp, mid_entry->multiEnd);
75             if (mid_entry->resp_buf) {
76                 cifs_dump_detail(mid_entry->resp_buf, server);
77                 cifs_dump_mem("existing buf: ",
78                             mid_entry->resp_buf, 62);
79             }
80         }
81     #endif
82 }
```

```
79         }
80     }
81     spin_unlock(&server->mid_lock);
82 #endif /* CONFIG_CIFS_DEBUG2 */
83 }
84
85 #ifdef CONFIG_PROC_FS
86 static void cifs_debug_tcon(struct seq_file *m, struct cifs_tcon *tcon)
87 {
88     __u32 dev_type = le32_to_cpu(tcon->fsDevInfo.DeviceType);
89
90     seq_printf(m, "%s Mounts: %d ", tcon->tree_name, tcon->tc_count);
91     if (tcon->nativeFileSystem)
92         seq_printf(m, "Type: %s ", tcon->nativeFileSystem);
93     seq_printf(m, "DevInfo: 0x%x Attributes: 0x%x\n\tPathComponentMax: %d Status: %d",
94               le32_to_cpu(tcon->fsDevInfo.DeviceCharacteristics),
95               le32_to_cpu(tcon->fsAttrInfo.Attributes),
96               le32_to_cpu(tcon->fsAttrInfo.MaxPathNameComponentLength),
97               tcon->status);
98     if (dev_type == FILE_DEVICE_DISK)
99         seq_puts(m, " type: DISK ");
100     else if (dev_type == FILE_DEVICE_CD_ROM)
101         seq_puts(m, " type: CDROM ");
102     else
103         seq_printf(m, " type: %d ", dev_type);
104
105     seq_printf(m, "Serial Number: 0x%x", tcon->vol_serial_number);
106
107     if ((tcon->seal) ||
108         (tcon->ses->session_flags & SMB2_SESSION_FLAG_ENCRYPT_DATA) ||
109         (tcon->share_flags & SHI1005_FLAGS_ENCRYPT_DATA))
110         seq_printf(m, " Encrypted");
111     if (tcon->nocase)
112         seq_printf(m, " nocase");
113     if (tcon->unix_ext)
114         seq_printf(m, " POSIX Extensions");
115     if (tcon->ses->server->ops->dump_share_caps)
116         tcon->ses->server->ops->dump_share_caps(m, tcon);
117     if (tcon->use_witness)
118         seq_puts(m, " Witness");
119     if (tcon->broken_sparse_sup)
120         seq_puts(m, " nosparse");
121     if (tcon->need_reconnect)
122         seq_puts(m, "\tDISCONNECTED ");
123     seq_putc(m, '\n');
124 }
125
126 static void
127 cifs_dump_channel(struct seq_file *m, int i, struct cifs_chan *chan)
```

```
128 {
129     struct TCP_Server_Info *server = chan->server;
130
131     seq_printf(m, "\n\n\t\tChannel: %d ConnectionId: 0x%llx"
132               "\n\t\tNumber of credits: %d Dialect 0x%x"
133               "\n\t\tTCP status: %d Instance: %d"
134               "\n\t\tLocal Users To Server: %d SecMode: 0x%x Req On Wire: %d"
135               "\n\t\tIn Send: %d In MaxReq Wait: %d",
136               i+1, server->conn_id,
137               server->credits,
138               server->dialect,
139               server->tcpStatus,
140               server->reconnect_instance,
141               server->srv_count,
142               server->sec_mode,
143               in_flight(server),
144               atomic_read(&server->in_send),
145               atomic_read(&server->num_waiters));
146 }
147
148 static void
149 cifs_dump_iface(struct seq_file *m, struct cifs_server_iface *iface)
150 {
151     struct sockaddr_in *ipv4 = (struct sockaddr_in *)&iface->sockaddr;
152     struct sockaddr_in6 *ipv6 = (struct sockaddr_in6 *)&iface->sockaddr;
153
154     seq_printf(m, "\tSpeed: %zu bps\n", iface->speed);
155     seq_puts(m, "\t\tCapabilities: ");
156     if (iface->rdma_capable)
157         seq_puts(m, "rdma ");
158     if (iface->rss_capable)
159         seq_puts(m, "rss ");
160     seq_putc(m, '\n');
161     if (iface->sockaddr.ss_family == AF_INET)
162         seq_printf(m, "\t\tIPv4: %pI4\n", &ipv4->sin_addr);
163     else if (iface->sockaddr.ss_family == AF_INET6)
164         seq_printf(m, "\t\tIPv6: %pI6\n", &ipv6->sin6_addr);
165     if (!iface->is_active)
166         seq_puts(m, "\t\t[for-cleanup]\n");
167 }
168
169 static int cifs_debug_files_proc_show(struct seq_file *m, void *v)
170 {
171     struct TCP_Server_Info *server;
172     struct cifs_ses *ses;
173     struct cifs_tcon *tcon;
174     struct cifsFileInfo *cfile;
175
176     seq_puts(m, "# Version:1\n");
```

```

177     seq_puts(m, "# Format:\n");
178     seq_puts(m, "# <tree id> <persistent fid> <flags> <count> <pid> <uid>");
179 #ifdef CONFIG_CIFS_DEBUG2
180     seq_printf(m, " <filename> <mid>\n");
181 #else
182     seq_printf(m, " <filename>\n");
183 #endif /* CIFS_DEBUG2 */
184     spin_lock(&cifs_tcp_ses_lock);
185     list_for_each_entry(server, &cifs_tcp_ses_list, tcp_ses_list) {
186         list_for_each_entry(ses, &server->smb_ses_list, smb_ses_list) {
187             list_for_each_entry(tcon, &ses->tcon_list, tcon_list) {
188                 spin_lock(&tcon->open_file_lock);
189                 list_for_each_entry(cfile, &tcon->openFileList, tlist) {
190                     seq_printf(m,
191                             "0x%x 0x%llx 0x%x %d %d %d %pd",
192                             tcon->tid,
193                             cfile->fid.persistent_fid,
194                             cfile->f_flags,
195                             cfile->count,
196                             cfile->pid,
197                             from_kuid(&init_user_ns, cfile->uid),
198                             cfile->dentry);
199 #ifdef CONFIG_CIFS_DEBUG2
200                     seq_printf(m, " %llu\n", cfile->fid.mid);
201 #else
202                     seq_printf(m, "\n");
203 #endif /* CIFS_DEBUG2 */
204                 }
205                 spin_unlock(&tcon->open_file_lock);
206             }
207         }
208     }
209     spin_unlock(&cifs_tcp_ses_lock);
210     seq_putc(m, '\n');
211     return 0;
212 }
213
214 static int cifs_debug_data_proc_show(struct seq_file *m, void *v)
215 {
216     struct mid_q_entry *mid_entry;
217     struct TCP_Server_Info *server;
218     struct cifs_ses *ses;
219     struct cifs_tcon *tcon;
220     struct cifs_server_iface *iface;
221     int c, i, j;
222
223     seq_puts(m,
224             "Display Internal CIFS Data Structures for Debugging\n"
225             "-----\n");

```

```
226     seq_printf(m, "CIFS Version %s\n", CIFS_VERSION);
227     seq_printf(m, "Features:");
228 #ifdef CONFIG_CIFS_DFS_UPCALL
229     seq_printf(m, " DFS");
230 #endif
231 #ifdef CONFIG_CIFS_FSCACHE
232     seq_printf(m, ",FSCACHE");
233 #endif
234 #ifdef CONFIG_CIFS_SMB_DIRECT
235     seq_printf(m, ",SMB_DIRECT");
236 #endif
237 #ifdef CONFIG_CIFS_STATS2
238     seq_printf(m, ",STATS2");
239 #else
240     seq_printf(m, ",STATS");
241 #endif
242 #ifdef CONFIG_CIFS_DEBUG2
243     seq_printf(m, ",DEBUG2");
244 #elif defined(CONFIG_CIFS_DEBUG)
245     seq_printf(m, ",DEBUG");
246 #endif
247 #ifdef CONFIG_CIFS_ALLOW_INSECURE_LEGACY
248     seq_printf(m, ",ALLOW_INSECURE_LEGACY");
249 #endif
250 #ifdef CONFIG_CIFS_POSIX
251     seq_printf(m, ",CIFS_POSIX");
252 #endif
253 #ifdef CONFIG_CIFS_UPCALL
254     seq_printf(m, ",UPCALL(SPNEGO)");
255 #endif
256 #ifdef CONFIG_CIFS_XATTR
257     seq_printf(m, ",XATTR");
258 #endif
259     seq_printf(m, ",ACL");
260 #ifdef CONFIG_CIFS_SWN_UPCALL
261     seq_puts(m, ",WITNESS");
262 #endif
263     seq_putc(m, '\n');
264     seq_printf(m, "CIFSMaxBufSize: %d\n", CIFSMaxBufSize);
265     seq_printf(m, "Active VFS Requests: %d\n", GlobalTotalActiveXid);
266
267     seq_printf(m, "\nServers: ");
268
269     c = 0;
270     spin_lock(&cifs_tcp_ses_lock);
271     list_for_each_entry(server, &cifs_tcp_ses_list, tcp_ses_list) {
272         /* channel info will be printed as a part of sessions below */
273         if (CIFS_SERVER_IS_CHAN(server))
274             continue;
```

```
275
276         c++;
277         seq_printf(m, "\n%d) ConnectionId: 0x%llx ",
278                 c, server->conn_id);
279
280         if (server->hostname)
281             seq_printf(m, "Hostname: %s ", server->hostname);
282 #ifdef CONFIG_CIFS_SMB_DIRECT
283         if (!server->rdma)
284             goto skip_rdma;
285
286         if (!server->smbd_conn) {
287             seq_printf(m, "\nSMBDirect transport not available");
288             goto skip_rdma;
289         }
290
291         seq_printf(m, "\nSMBDirect (in hex) protocol version: %x "
292                 "transport status: %x",
293                 server->smbd_conn->protocol,
294                 server->smbd_conn->transport_status);
295         seq_printf(m, "\nConn receive_credit_max: %x "
296                 "send_credit_target: %x max_send_size: %x",
297                 server->smbd_conn->receive_credit_max,
298                 server->smbd_conn->send_credit_target,
299                 server->smbd_conn->max_send_size);
300         seq_printf(m, "\nConn max_fragmented_recv_size: %x "
301                 "max_fragmented_send_size: %x max_receive_size: %x",
302                 server->smbd_conn->max_fragmented_recv_size,
303                 server->smbd_conn->max_fragmented_send_size,
304                 server->smbd_conn->max_receive_size);
305         seq_printf(m, "\nConn keep_alive_interval: %x "
306                 "max_readwrite_size: %x rdma_readwrite_threshold: %x",
307                 server->smbd_conn->keep_alive_interval,
308                 server->smbd_conn->max_readwrite_size,
309                 server->smbd_conn->rdma_readwrite_threshold);
310         seq_printf(m, "\nDebug count_get_receive_buffer: %x "
311                 "count_put_receive_buffer: %x count_send_empty: %x",
312                 server->smbd_conn->count_get_receive_buffer,
313                 server->smbd_conn->count_put_receive_buffer,
314                 server->smbd_conn->count_send_empty);
315         seq_printf(m, "\nRead Queue count_reassembly_queue: %x "
316                 "count_enqueue_reassembly_queue: %x "
317                 "count_dequeue_reassembly_queue: %x "
318                 "fragment_reassembly_remaining: %x "
319                 "reassembly_data_length: %x "
320                 "reassembly_queue_length: %x",
321                 server->smbd_conn->count_reassembly_queue,
322                 server->smbd_conn->count_enqueue_reassembly_queue,
323                 server->smbd_conn->count_dequeue_reassembly_queue,
```

```

324         server->smbd_conn->fragment_reassembly_remaining,
325         server->smbd_conn->reassembly_data_length,
326         server->smbd_conn->reassembly_queue_length);
327     seq_printf(m, "\nCurrent Credits send_credits: %x "
328               "receive_credits: %x receive_credit_target: %x",
329               atomic_read(&server->smbd_conn->send_credits),
330               atomic_read(&server->smbd_conn->receive_credits),
331               server->smbd_conn->receive_credit_target);
332     seq_printf(m, "\nPending send_pending: %x ",
333               atomic_read(&server->smbd_conn->send_pending));
334     seq_printf(m, "\nReceive buffers count_receive_queue: %x "
335               "count_empty_packet_queue: %x",
336               server->smbd_conn->count_receive_queue,
337               server->smbd_conn->count_empty_packet_queue);
338     seq_printf(m, "\nMR responder_resources: %x "
339               "max_frmr_depth: %x mr_type: %x",
340               server->smbd_conn->responder_resources,
341               server->smbd_conn->max_frmr_depth,
342               server->smbd_conn->mr_type);
343     seq_printf(m, "\nMR mr_ready_count: %x mr_used_count: %x",
344               atomic_read(&server->smbd_conn->mr_ready_count),
345               atomic_read(&server->smbd_conn->mr_used_count));
346     skip_rdma:
347     #endif
348     seq_printf(m, "\nNumber of credits: %d Dialect 0x%x",
349               server->credits, server->dialect);
350     if (server->compress_algorithm == SMB3_COMPRESS_LZNT1)
351         seq_printf(m, " COMPRESS_LZNT1");
352     else if (server->compress_algorithm == SMB3_COMPRESS_LZ77)
353         seq_printf(m, " COMPRESS_LZ77");
354     else if (server->compress_algorithm == SMB3_COMPRESS_LZ77_HUFF)
355         seq_printf(m, " COMPRESS_LZ77_HUFF");
356     if (server->sign)
357         seq_printf(m, " signed");
358     if (server->posix_ext_supported)
359         seq_printf(m, " posix");
360     if (server->nosharesock)
361         seq_printf(m, " nosharesock");
362
363     if (server->rdma)
364         seq_printf(m, "\nRDMA ");
365     seq_printf(m, "\nTCP status: %d Instance: %d"
366               "\nLocal Users To Server: %d SecMode: 0x%x Req On Wire: %d",
367               server->tcpStatus,
368               server->reconnect_instance,
369               server->srv_count,
370               server->sec_mode, in_flight(server));
371
372     seq_printf(m, "\nIn Send: %d In MaxReq Wait: %d",

```



```

373         atomic_read(&server->in_send),
374         atomic_read(&server->num_waiters));
375
376     seq_printf(m, "\n\n\tSessions: ");
377     i = 0;
378     list_for_each_entry(ses, &server->smb_ses_list, smb_ses_list) {
379         i++;
380         if ((ses->serverDomain == NULL) ||
381             (ses->serverOS == NULL) ||
382             (ses->serverNOS == NULL)) {
383             seq_printf(m, "\n\t%d) Address: %s Uses: %d Capability: 0x%x\tSess",
384                 i, ses->ip_addr, ses->ses_count,
385                 ses->capabilities, ses->ses_status);
386             if (ses->session_flags & SMB2_SESSION_FLAG_IS_GUEST)
387                 seq_printf(m, "Guest ");
388             else if (ses->session_flags & SMB2_SESSION_FLAG_IS_NULL)
389                 seq_printf(m, "Anonymous ");
390         } else {
391             seq_printf(m,
392                 "\n\t%d) Name: %s Domain: %s Uses: %d OS: %s "
393                 "\n\tNOS: %s\tCapability: 0x%x"
394                 "\n\tSMB session status: %d ",
395                 i, ses->ip_addr, ses->serverDomain,
396                 ses->ses_count, ses->serverOS, ses->serverNOS,
397                 ses->capabilities, ses->ses_status);
398         }
399
400         seq_printf(m, "\n\tSecurity type: %s ",
401             get_security_type_str(server->ops->select_sectype(server, ses->sec
402
403         /* dump session id helpful for use with network trace */
404         seq_printf(m, " SessionId: 0x%llx", ses->Suid);
405         if (ses->session_flags & SMB2_SESSION_FLAG_ENCRYPT_DATA)
406             seq_puts(m, " encrypted");
407         if (ses->sign)
408             seq_puts(m, " signed");
409
410         seq_printf(m, "\n\tUser: %d Cred User: %d",
411             from_kuid(&init_user_ns, ses->linux_uid),
412             from_kuid(&init_user_ns, ses->cred_uid));
413
414         spin_lock(&ses->chan_lock);
415         if (CIFS_CHAN_NEEDS_RECONNECT(ses, 0))
416             seq_puts(m, "\tPrimary channel: DISCONNECTED ");
417         if (CIFS_CHAN_IN_RECONNECT(ses, 0))
418             seq_puts(m, "\t[RECONNECTING] ");
419
420         if (ses->chan_count > 1) {
421             seq_printf(m, "\n\n\tExtra Channels: %zu ",

```

```

422         ses->chan_count-1);
423     for (j = 1; j < ses->chan_count; j++) {
424         cifs_dump_channel(m, j, &ses->chans[j]);
425         if (CIFS_CHAN_NEEDS_RECONNECT(ses, j))
426             seq_puts(m, "\tDISCONNECTED ");
427         if (CIFS_CHAN_IN_RECONNECT(ses, j))
428             seq_puts(m, "\t[RECONNECTING] ");
429     }
430 }
431 spin_unlock(&ses->chan_lock);
432
433 seq_puts(m, "\n\n\tShares: ");
434 j = 0;
435
436 seq_printf(m, "\n\t%d) IPC: ", j);
437 if (ses->tcon_ipc)
438     cifs_debug_tcon(m, ses->tcon_ipc);
439 else
440     seq_puts(m, "none\n");
441
442 list_for_each_entry(tcon, &ses->tcon_list, tcon_list) {
443     ++j;
444     seq_printf(m, "\n\t%d) ", j);
445     cifs_debug_tcon(m, tcon);
446 }
447
448 spin_lock(&ses->iface_lock);
449 if (ses->iface_count)
450     seq_printf(m, "\n\n\tServer interfaces: %zu",
451         ses->iface_count);
452 j = 0;
453 list_for_each_entry(iface, &ses->iface_list,
454     iface_head) {
455     seq_printf(m, "\n\t%d)", ++j);
456     cifs_dump_iface(m, iface);
457     if (is_ses_using_iface(ses, iface))
458         seq_puts(m, "\t\t[CONNECTED]\n");
459 }
460 spin_unlock(&ses->iface_lock);
461 }
462 if (i == 0)
463     seq_printf(m, "\n\t\t[NONE]");
464
465 seq_puts(m, "\n\n\tMIDs: ");
466 spin_lock(&server->mid_lock);
467 list_for_each_entry(mid_entry, &server->pending_mid_q, qhead) {
468     seq_printf(m, "\n\tState: %d com: %d pid:"
469         " %d cbdata: %p mid %llu\n",
470         mid_entry->mid_state,

```

```

471         le16_to_cpu(mid_entry->command),
472         mid_entry->pid,
473         mid_entry->callback_data,
474         mid_entry->mid);
475     }
476     spin_unlock(&server->mid_lock);
477     seq_printf(m, "\n--\n");
478 }
479 if (c == 0)
480     seq_printf(m, "\n\t[NONE]");
481
482     spin_unlock(&cifs_tcp_ses_lock);
483     seq_putc(m, '\n');
484     cifs_swn_dump(m);
485
486     /* BB add code to dump additional info such as TCP session info now */
487     return 0;
488 }
489
490 static ssize_t cifs_stats_proc_write(struct file *file,
491                                     const char __user *buffer, size_t count, loff_t *ppos)
492 {
493     bool bv;
494     int rc;
495     struct TCP_Server_Info *server;
496     struct cifs_ses *ses;
497     struct cifs_tcon *tcon;
498
499     rc = kstrtobool_from_user(buffer, count, &bv);
500     if (rc == 0) {
501 #ifdef CONFIG_CIFS_STATS2
502         int i;
503
504         atomic_set(&total_buf_alloc_count, 0);
505         atomic_set(&total_small_buf_alloc_count, 0);
506 #endif /* CONFIG_CIFS_STATS2 */
507         atomic_set(&tcpSesReconnectCount, 0);
508         atomic_set(&tconInfoReconnectCount, 0);
509
510         spin_lock(&GlobalMid_Lock);
511         GlobalMaxActiveXid = 0;
512         GlobalCurrentXid = 0;
513         spin_unlock(&GlobalMid_Lock);
514         spin_lock(&cifs_tcp_ses_lock);
515         list_for_each_entry(server, &cifs_tcp_ses_list, tcp_ses_list) {
516             server->max_in_flight = 0;
517 #ifdef CONFIG_CIFS_STATS2
518             for (i = 0; i < NUMBER_OF_SMB2_COMMANDS; i++) {
519                 atomic_set(&server->num_cmds[i], 0);

```

```

520         atomic_set(&server->smb2slowcmd[i], 0);
521         server->time_per_cmd[i] = 0;
522         server->slowest_cmd[i] = 0;
523         server->fastest_cmd[0] = 0;
524     }
525 #endif /* CONFIG_CIFS_STATS2 */
526     list_for_each_entry(ses, &server->smb_ses_list, smb_ses_list) {
527         list_for_each_entry(tcon, &ses->tcon_list, tcon_list) {
528             atomic_set(&tcon->num_smb_ses, 0);
529             spin_lock(&tcon->stat_lock);
530             tcon->bytes_read = 0;
531             tcon->bytes_written = 0;
532             spin_unlock(&tcon->stat_lock);
533             if (server->ops->clear_stats)
534                 server->ops->clear_stats(tcon);
535         }
536     }
537 }
538 spin_unlock(&cifs_tcp_ses_lock);
539 } else {
540     return rc;
541 }
542
543 return count;
544 }
545
546 static int cifs_stats_proc_show(struct seq_file *m, void *v)
547 {
548     int i;
549 #ifdef CONFIG_CIFS_STATS2
550     int j;
551 #endif /* STATS2 */
552     struct TCP_Server_Info *server;
553     struct cifs_ses *ses;
554     struct cifs_tcon *tcon;
555
556     seq_printf(m, "Resources in use\nCIFS Session: %d\n",
557               sesInfoAllocCount.counter);
558     seq_printf(m, "Share (unique mount targets): %d\n",
559               tconInfoAllocCount.counter);
560     seq_printf(m, "SMB Request/Response Buffer: %d Pool size: %d\n",
561               buf_alloc_count.counter,
562               cifs_min_rcv + tcpSesAllocCount.counter);
563     seq_printf(m, "SMB Small Req/Resp Buffer: %d Pool size: %d\n",
564               small_buf_alloc_count.counter, cifs_min_small);
565 #ifdef CONFIG_CIFS_STATS2
566     seq_printf(m, "Total Large %d Small %d Allocations\n",
567               atomic_read(&total_buf_alloc_count),
568               atomic_read(&total_small_buf_alloc_count));

```

```

569 #endif /* CONFIG_CIFS_STATS2 */
570
571 seq_printf(m, "Operations (MIDs): %d\n", atomic_read(&mid_count));
572 seq_printf(m,
573     "\n%d session %d share reconnects\n",
574     tcpSesReconnectCount.counter, tconInfoReconnectCount.counter);
575
576 seq_printf(m,
577     "Total vfs operations: %d maximum at one time: %d\n",
578     GlobalCurrentXid, GlobalMaxActiveXid);
579
580 i = 0;
581 spin_lock(&cifs_tcp_ses_lock);
582 list_for_each_entry(server, &cifs_tcp_ses_list, tcp_ses_list) {
583     seq_printf(m, "\nMax requests in flight: %d", server->max_in_flight);
584 #ifdef CONFIG_CIFS_STATS2
585     seq_puts(m, "\nTotal time spent processing by command. Time ");
586     seq_printf(m, "units are jiffies (%d per second)\n", HZ);
587     seq_puts(m, "  SMB3 CMD\tNumber\tTotal Time\tFastest\tSlowest\n");
588     seq_puts(m, "  -----\t-----\t-----\t-----\t-----\n");
589     for (j = 0; j < NUMBER_OF_SMB2_COMMANDS; j++)
590         seq_printf(m, "  %d\t\t%d\t\t%llu\t\t%u\t\t%u\n", j,
591             atomic_read(&server->num_cmds[j]),
592             server->time_per_cmd[j],
593             server->fastest_cmd[j],
594             server->slowest_cmd[j]);
595     for (j = 0; j < NUMBER_OF_SMB2_COMMANDS; j++)
596         if (atomic_read(&server->smb2slowcmd[j]))
597             seq_printf(m, "  %d slow responses from %s for command %d\n",
598                 atomic_read(&server->smb2slowcmd[j]),
599                 server->hostname, j);
600 #endif /* STATS2 */
601     list_for_each_entry(ses, &server->smb_ses_list, smb_ses_list) {
602         list_for_each_entry(tcon, &ses->tcon_list, tcon_list) {
603             i++;
604             seq_printf(m, "\n%d) %s", i, tcon->tree_name);
605             if (tcon->need_reconnect)
606                 seq_puts(m, "\tDISCONNECTED ");
607             seq_printf(m, "\nSMBs: %d",
608                 atomic_read(&tcon->num_smbs_sent));
609             if (server->ops->print_stats)
610                 server->ops->print_stats(m, tcon);
611         }
612     }
613 }
614 spin_unlock(&cifs_tcp_ses_lock);
615
616 seq_putc(m, '\n');
617 return 0;

```

```
618 }
619
620 static int cifs_stats_proc_open(struct inode *inode, struct file *file)
621 {
622     return single_open(file, cifs_stats_proc_show, NULL);
623 }
624
625 static const struct proc_ops cifs_stats_proc_ops = {
626     .proc_open      = cifs_stats_proc_open,
627     .proc_read      = seq_read,
628     .proc_lseek     = seq_lseek,
629     .proc_release   = single_release,
630     .proc_write     = cifs_stats_proc_write,
631 };
632
633 #ifdef CONFIG_CIFS_SMB_DIRECT
634 #define PROC_FILE_DEFINE(name) \
635 static ssize_t name##_write(struct file *file, const char __user *buffer, \
636     size_t count, loff_t *ppos) \
637 { \
638     int rc; \
639     rc = kstrtoint_from_user(buffer, count, 10, & name); \
640     if (rc) \
641         return rc; \
642     return count; \
643 } \
644 static int name##_proc_show(struct seq_file *m, void *v) \
645 { \
646     seq_printf(m, "%d\n", name ); \
647     return 0; \
648 } \
649 static int name##_open(struct inode *inode, struct file *file) \
650 { \
651     return single_open(file, name##_proc_show, NULL); \
652 } \
653 \
654 static const struct proc_ops cifs_##name##_proc_fops = { \
655     .proc_open      = name##_open, \
656     .proc_read      = seq_read, \
657     .proc_lseek     = seq_lseek, \
658     .proc_release   = single_release, \
659     .proc_write     = name##_write, \
660 }
661
662 PROC_FILE_DEFINE(rdma_readwrite_threshold);
663 PROC_FILE_DEFINE(smbd_max_frmr_depth);
664 PROC_FILE_DEFINE(smbd_keep_alive_interval);
665 PROC_FILE_DEFINE(smbd_max_receive_size);
666 PROC_FILE_DEFINE(smbd_max_fragmented_recv_size);
```

```
667 PROC_FILE_DEFINE(smbd_max_send_size);
668 PROC_FILE_DEFINE(smbd_send_credit_target);
669 PROC_FILE_DEFINE(smbd_receive_credit_max);
670 #endif
671
672 static struct proc_dir_entry *proc_fs_cifs;
673 static const struct proc_ops cifsFYI_proc_ops;
674 static const struct proc_ops cifs_lookup_cache_proc_ops;
675 static const struct proc_ops traceSMB_proc_ops;
676 static const struct proc_ops cifs_security_flags_proc_ops;
677 static const struct proc_ops cifs_linux_ext_proc_ops;
678 static const struct proc_ops cifs_mount_params_proc_ops;
679
680 void
681 cifs_proc_init(void)
682 {
683     proc_fs_cifs = proc_mkdir("fs/cifs", NULL);
684     if (proc_fs_cifs == NULL)
685         return;
686
687     proc_create_single("DebugData", 0, proc_fs_cifs,
688                       cifs_debug_data_proc_show);
689
690     proc_create_single("open_files", 0400, proc_fs_cifs,
691                       cifs_debug_files_proc_show);
692
693     proc_create("Stats", 0644, proc_fs_cifs, &cifs_stats_proc_ops);
694     proc_create("cifsFYI", 0644, proc_fs_cifs, &cifsFYI_proc_ops);
695     proc_create("traceSMB", 0644, proc_fs_cifs, &traceSMB_proc_ops);
696     proc_create("LinuxExtensionsEnabled", 0644, proc_fs_cifs,
697               &cifs_linux_ext_proc_ops);
698     proc_create("SecurityFlags", 0644, proc_fs_cifs,
699               &cifs_security_flags_proc_ops);
700     proc_create("LookupCacheEnabled", 0644, proc_fs_cifs,
701               &cifs_lookup_cache_proc_ops);
702
703     proc_create("mount_params", 0444, proc_fs_cifs, &cifs_mount_params_proc_ops);
704
705 #ifdef CONFIG_CIFS_DFS_UPCALL
706     proc_create("dfscache", 0644, proc_fs_cifs, &dfscache_proc_ops);
707 #endif
708
709 #ifdef CONFIG_CIFS_SMB_DIRECT
710     proc_create("rdma_readwrite_threshold", 0644, proc_fs_cifs,
711               &cifs_rdma_readwrite_threshold_proc_ops);
712     proc_create("smbd_max_frmr_depth", 0644, proc_fs_cifs,
713               &cifs_smbd_max_frmr_depth_proc_ops);
714     proc_create("smbd_keep_alive_interval", 0644, proc_fs_cifs,
715               &cifs_smbd_keep_alive_interval_proc_ops);
```

```
716     proc_create("smbd_max_receive_size", 0644, proc_fs_cifs,  
717                 &cifs_smbd_max_receive_size_proc_fops);  
718     proc_create("smbd_max_fragmented_recv_size", 0644, proc_fs_cifs,  
719                 &cifs_smbd_max_fragmented_recv_size_proc_fops);  
720     proc_create("smbd_max_send_size", 0644, proc_fs_cifs,  
721                 &cifs_smbd_max_send_size_proc_fops);  
722     proc_create("smbd_send_credit_target", 0644, proc_fs_cifs,  
723                 &cifs_smbd_send_credit_target_proc_fops);  
724     proc_create("smbd_receive_credit_max", 0644, proc_fs_cifs,  
725                 &cifs_smbd_receive_credit_max_proc_fops);  
726 #endif  
727 }  
728  
729 void  
730 cifs_proc_clean(void)  
731 {  
732     if (proc_fs_cifs == NULL)  
733         return;  
734  
735     remove_proc_entry("DebugData", proc_fs_cifs);  
736     remove_proc_entry("open_files", proc_fs_cifs);  
737     remove_proc_entry("cifsFYI", proc_fs_cifs);  
738     remove_proc_entry("traceSMB", proc_fs_cifs);  
739     remove_proc_entry("Stats", proc_fs_cifs);  
740     remove_proc_entry("SecurityFlags", proc_fs_cifs);  
741     remove_proc_entry("LinuxExtensionsEnabled", proc_fs_cifs);  
742     remove_proc_entry("LookupCacheEnabled", proc_fs_cifs);  
743     remove_proc_entry("mount_params", proc_fs_cifs);  
744  
745 #ifdef CONFIG_CIFS_DFS_UPCALL  
746     remove_proc_entry("dfscache", proc_fs_cifs);  
747 #endif  
748 #ifdef CONFIG_CIFS_SMB_DIRECT  
749     remove_proc_entry("rdma_readwrite_threshold", proc_fs_cifs);  
750     remove_proc_entry("smbd_max_frmr_depth", proc_fs_cifs);  
751     remove_proc_entry("smbd_keep_alive_interval", proc_fs_cifs);  
752     remove_proc_entry("smbd_max_receive_size", proc_fs_cifs);  
753     remove_proc_entry("smbd_max_fragmented_recv_size", proc_fs_cifs);  
754     remove_proc_entry("smbd_max_send_size", proc_fs_cifs);  
755     remove_proc_entry("smbd_send_credit_target", proc_fs_cifs);  
756     remove_proc_entry("smbd_receive_credit_max", proc_fs_cifs);  
757 #endif  
758     remove_proc_entry("fs/cifs", NULL);  
759 }  
760  
761 static int cifsFYI_proc_show(struct seq_file *m, void *v)  
762 {  
763     seq_printf(m, "%d\n", cifsFYI);  
764     return 0;  
}
```



```
765 }
766
767 static int cifsFYI_proc_open(struct inode *inode, struct file *file)
768 {
769     return single_open(file, cifsFYI_proc_show, NULL);
770 }
771
772 static ssize_t cifsFYI_proc_write(struct file *file, const char __user *buffer,
773     size_t count, loff_t *ppos)
774 {
775     char c[2] = { '\0' };
776     bool bv;
777     int rc;
778
779     rc = get_user(c[0], buffer);
780     if (rc)
781         return rc;
782     if (strtobool(c, &bv) == 0)
783         cifsFYI = bv;
784     else if ((c[0] > '1') && (c[0] <= '9'))
785         cifsFYI = (int) (c[0] - '0'); /* see cifs_debug.h for meanings */
786     else
787         return -EINVAL;
788
789     return count;
790 }
791
792 static const struct proc_ops cifsFYI_proc_ops = {
793     .proc_open      = cifsFYI_proc_open,
794     .proc_read      = seq_read,
795     .proc_lseek     = seq_lseek,
796     .proc_release   = single_release,
797     .proc_write     = cifsFYI_proc_write,
798 };
799
800 static int cifs_linux_ext_proc_show(struct seq_file *m, void *v)
801 {
802     seq_printf(m, "%d\n", linuxExtEnabled);
803     return 0;
804 }
805
806 static int cifs_linux_ext_proc_open(struct inode *inode, struct file *file)
807 {
808     return single_open(file, cifs_linux_ext_proc_show, NULL);
809 }
810
811 static ssize_t cifs_linux_ext_proc_write(struct file *file,
812     const char __user *buffer, size_t count, loff_t *ppos)
813 {
```

```
814     int rc;
815
816     rc = kstrtobool_from_user(buffer, count, &linuxExtEnabled);
817     if (rc)
818         return rc;
819
820     return count;
821 }
822
823 static const struct proc_ops cifs_linux_ext_proc_ops = {
824     .proc_open      = cifs_linux_ext_proc_open,
825     .proc_read      = seq_read,
826     .proc_lseek     = seq_lseek,
827     .proc_release   = single_release,
828     .proc_write     = cifs_linux_ext_proc_write,
829 };
830
831 static int cifs_lookup_cache_proc_show(struct seq_file *m, void *v)
832 {
833     seq_printf(m, "%d\n", lookupCacheEnabled);
834     return 0;
835 }
836
837 static int cifs_lookup_cache_proc_open(struct inode *inode, struct file *file)
838 {
839     return single_open(file, cifs_lookup_cache_proc_show, NULL);
840 }
841
842 static ssize_t cifs_lookup_cache_proc_write(struct file *file,
843     const char __user *buffer, size_t count, loff_t *ppos)
844 {
845     int rc;
846
847     rc = kstrtobool_from_user(buffer, count, &lookupCacheEnabled);
848     if (rc)
849         return rc;
850
851     return count;
852 }
853
854 static const struct proc_ops cifs_lookup_cache_proc_ops = {
855     .proc_open      = cifs_lookup_cache_proc_open,
856     .proc_read      = seq_read,
857     .proc_lseek     = seq_lseek,
858     .proc_release   = single_release,
859     .proc_write     = cifs_lookup_cache_proc_write,
860 };
861
862 static int traceSMB_proc_show(struct seq_file *m, void *v)
```

```
863 {
864     seq_printf(m, "%d\n", traceSMB);
865     return 0;
866 }
867
868 static int traceSMB_proc_open(struct inode *inode, struct file *file)
869 {
870     return single_open(file, traceSMB_proc_show, NULL);
871 }
872
873 static ssize_t traceSMB_proc_write(struct file *file, const char __user *buffer,
874     size_t count, loff_t *ppos)
875 {
876     int rc;
877
878     rc = kstrtobool_from_user(buffer, count, &traceSMB);
879     if (rc)
880         return rc;
881
882     return count;
883 }
884
885 static const struct proc_ops traceSMB_proc_ops = {
886     .proc_open      = traceSMB_proc_open,
887     .proc_read      = seq_read,
888     .proc_lseek     = seq_lseek,
889     .proc_release   = single_release,
890     .proc_write     = traceSMB_proc_write,
891 };
892
893 static int cifs_security_flags_proc_show(struct seq_file *m, void *v)
894 {
895     seq_printf(m, "0x%x\n", global_secflags);
896     return 0;
897 }
898
899 static int cifs_security_flags_proc_open(struct inode *inode, struct file *file)
900 {
901     return single_open(file, cifs_security_flags_proc_show, NULL);
902 }
903
904 /*
905  * Ensure that if someone sets a MUST flag, that we disable all other MAY
906  * flags except for the ones corresponding to the given MUST flag. If there are
907  * multiple MUST flags, then try to prefer more secure ones.
908  */
909 static void
910 cifs_security_flags_handle_must_flags(unsigned int *flags)
911 {
```

```
912     unsigned int signflags = *flags & CIFSSEC_MUST_SIGN;
913
914     if ((*flags & CIFSSEC_MUST_KRB5) == CIFSSEC_MUST_KRB5)
915         *flags = CIFSSEC_MUST_KRB5;
916     else if ((*flags & CIFSSEC_MUST_NTLMSSP) == CIFSSEC_MUST_NTLMSSP)
917         *flags = CIFSSEC_MUST_NTLMSSP;
918     else if ((*flags & CIFSSEC_MUST_NTLMV2) == CIFSSEC_MUST_NTLMV2)
919         *flags = CIFSSEC_MUST_NTLMV2;
920
921     *flags |= signflags;
922 }
923
924 static ssize_t cifs_security_flags_proc_write(struct file *file,
925     const char __user *buffer, size_t count, loff_t *ppos)
926 {
927     int rc;
928     unsigned int flags;
929     char flags_string[12];
930     bool bv;
931
932     if ((count < 1) || (count > 11))
933         return -EINVAL;
934
935     memset(flags_string, 0, 12);
936
937     if (copy_from_user(flags_string, buffer, count))
938         return -EFAULT;
939
940     if (count < 3) {
941         /* single char or single char followed by null */
942         if (strtobool(flags_string, &bv) == 0) {
943             global_secflags = bv ? CIFSSEC_MAX : CIFSSEC_DEF;
944             return count;
945         } else if (!isdigit(flags_string[0])) {
946             cifs_dbg(VFS, "Invalid SecurityFlags: %s\n",
947                 flags_string);
948             return -EINVAL;
949         }
950     }
951
952     /* else we have a number */
953     rc = kstrtouint(flags_string, 0, &flags);
954     if (rc) {
955         cifs_dbg(VFS, "Invalid SecurityFlags: %s\n",
956             flags_string);
957         return rc;
958     }
959
960     cifs_dbg(FYI, "sec flags 0x%x\n", flags);
```

```
961
962     if (flags == 0) {
963         cifs_dbg(VFS, "Invalid SecurityFlags: %s\n", flags_string);
964         return -EINVAL;
965     }
966
967     if (flags & ~CIFSSSEC_MASK) {
968         cifs_dbg(VFS, "Unsupported security flags: 0x%x\n",
969             flags & ~CIFSSSEC_MASK);
970         return -EINVAL;
971     }
972
973     cifs_security_flags_handle_must_flags(&flags);
974
975     /* flags look ok - update the global security flags for cifs module */
976     global_secflags = flags;
977     if (global_secflags & CIFSSSEC_MUST_SIGN) {
978         /* requiring signing implies signing is allowed */
979         global_secflags |= CIFSSSEC_MAY_SIGN;
980         cifs_dbg(FYI, "packet signing now required\n");
981     } else if ((global_secflags & CIFSSSEC_MAY_SIGN) == 0) {
982         cifs_dbg(FYI, "packet signing disabled\n");
983     }
984     /* BB should we turn on MAY flags for other MUST options? */
985     return count;
986 }
987
988 static const struct proc_ops cifs_security_flags_proc_ops = {
989     .proc_open      = cifs_security_flags_proc_open,
990     .proc_read      = seq_read,
991     .proc_lseek     = seq_lseek,
992     .proc_release   = single_release,
993     .proc_write     = cifs_security_flags_proc_write,
994 };
995
996 /* To make it easier to debug, can help to show mount params */
997 static int cifs_mount_params_proc_show(struct seq_file *m, void *v)
998 {
999     const struct fs_parameter_spec *p;
1000     const char *type;
1001
1002     for (p = smb3_fs_parameters; p->name; p++) {
1003         /* cannot use switch with pointers... */
1004         if (!p->type) {
1005             if (p->flags == fs_param_neg_with_no)
1006                 type = "noflag";
1007             else
1008                 type = "flag";
1009         } else if (p->type == fs_param_is_bool)
```

```
1010         type = "bool";
1011     else if (p->type == fs_param_is_u32)
1012         type = "u32";
1013     else if (p->type == fs_param_is_u64)
1014         type = "u64";
1015     else if (p->type == fs_param_is_string)
1016         type = "string";
1017     else
1018         type = "unknown";
1019
1020     seq_printf(m, "%s:%s\n", p->name, type);
1021 }
1022
1023 return 0;
1024 }
1025
1026 static int cifs_mount_params_proc_open(struct inode *inode, struct file *file)
1027 {
1028     return single_open(file, cifs_mount_params_proc_show, NULL);
1029 }
1030
1031 static const struct proc_ops cifs_mount_params_proc_ops = {
1032     .proc_open      = cifs_mount_params_proc_open,
1033     .proc_read      = seq_read,
1034     .proc_lseek     = seq_lseek,
1035     .proc_release   = single_release,
1036     /* No need for write for now */
1037     /* .proc_write   = cifs_mount_params_proc_write, */
1038 };
1039
1040 #else
1041 inline void cifs_proc_init(void)
1042 {
1043 }
1044
1045 inline void cifs_proc_clean(void)
1046 {
1047 }
1048 #endif /* PROC_FS */
```