

```

*****
51582 Fri Dec 18 10:33:31 2015
new/usr/src/uts/common/fs/fifofs/fifovnops.c
6474 getupeercred causes spurious event port wakeups on FIFOs
*****
_____unchanged_portion_omitted_____

1129 static inline int
1130 fifo_ioctl_getpeercred(fifonode_t *fnp, intptr_t arg, int mode)
1131 {
1132     k_peercred_t *kp = (k_peercred_t *)arg;
1133
1134     if (mode == FKIOCTL && fnp->fn_pcredp != NULL) {
1135         crhold(fnp->fn_pcredp);
1136         kp->pc_cr = fnp->fn_pcredp;
1137         kp->pc_cpuid = fnp->fn_cpuid;
1138         return (0);
1139     } else {
1140         return (ENOTSUP);
1141     }
1142 }

1144 #endif /* ! codereview */
1145 static int
1146 fifo_fastioctl(vnode_t *vp, int cmd, intptr_t arg, int mode,
1147               cred_t *cr, int *rvalp)
1148 {
1149     fifonode_t *fnp          = VTOF(vp);
1150     fifonode_t *fn_dest;
1151     int error                = 0;
1152     fifolock_t *fn_lock     = fnp->fn_lock;
1153     int cnt;

1155     /*
1156      * tty operations not allowed
1157      */
1158     if (((cmd & IOCTYPE) == LDIOC) ||
1159         ((cmd & IOCTYPE) == tIOC) ||
1160         ((cmd & IOCTYPE) == TIOC)) {
1161         return (EINVAL);
1162     }

1164     mutex_enter(&fn_lock->flk_lock);

1166     if (!(fnp->fn_flag & FIFOFAST)) {
1167         goto stream_mode;
1168     }

1170     switch (cmd) {

1172     /*
1173      * Things we can't handle
1174      * These will switch us to streams mode.
1175      */
1176     default:
1177     case I_STR:
1178     case I_SRDOPT:
1179     case I_PUSH:
1180     case I_FDINSERT:
1181     case I_SENDFD:
1182     case I_RECVFD:
1183     case I_E_RECVFD:
1184     case I_ATMARK:
1185     case I_CKBAND:
1186     case I_GETBAND:
1187     case I_SWROPT:

```

```

1188         goto turn_fastoff;

1190     /*
1191      * Things that don't do damage
1192      * These things don't adjust the state of the
1193      * stream head (i_setcltime does, but we don't care)
1194      */
1195     case I_FIND:
1196     case I_GETSIG:
1197     case FIONBIO:
1198     case FIOASYNC:
1199     case I_GRDOPT: /* probably should not get this, but no harm */
1200     case I_GWROPT:
1201     case I_LIST:
1202     case I_SETCLTIME:
1203     case I_GETCLTIME:
1204         mutex_exit(&fn_lock->flk_lock);
1205         return (striocctl(vp, cmd, arg, mode, U_TO_K, cr, rvalp));

1207     case I_CANPUT:
1208         /*
1209          * We can only handle normal band canputs.
1210          * XXX: We could just always go to stream mode; after all
1211          * canput is a streams semantics type thing
1212          */
1213         if (arg != 0) {
1214             goto turn_fastoff;
1215         }
1216         *rvalp = (fnp->fn_dest->fn_count < Fifohiwat) ? 1 : 0;
1217         mutex_exit(&fn_lock->flk_lock);
1218         return (0);

1220     case I_NREAD:
1221         /*
1222          * This may seem a bit silly for non-streams semantics,
1223          * (After all, if they really want a message, they'll
1224          * probably use getmsg() anyway). but it doesn't hurt
1225          */
1226         error = copyout((caddr_t)&fnp->fn_count, (caddr_t)arg,
1227                       sizeof (cnt));
1228         if (error == 0) {
1229             *rvalp = (fnp->fn_count == 0) ? 0 : 1;
1230         }
1231         break;

1233     case FIORDCHK:
1234         *rvalp = fnp->fn_count;
1235         break;

1237     case I_PEEK:
1238     {
1239         STRUCT_DECL(strpeek, strpeek);
1240         struct uio uio;
1241         struct iovec iov;
1242         int count;
1243         mblk_t *bp;
1244         int len;

1246         STRUCT_INIT(strpeek, mode);

1248         if (fnp->fn_count == 0) {
1249             *rvalp = 0;
1250             break;
1251         }

1253         error = copyin((caddr_t)arg, STRUCT_BUF(strpeek),

```

```

1254     STRUCT_SIZE(strpeek));
1255     if (error)
1256         break;
1257
1258     /*
1259     * can't have any high priority message when in fast mode
1260     */
1261     if (STRUCT_FGET(strpeek, flags) & RS_HIPRI) {
1262         *rvalp = 0;
1263         break;
1264     }
1265
1266     len = STRUCT_FGET(strpeek, databuf.maxlen);
1267     if (len <= 0) {
1268         STRUCT_FSET(strpeek, databuf.len, len);
1269     } else {
1270         iov.iov_base = STRUCT_FGETP(strpeek, databuf.buf);
1271         iov.iov_len = len;
1272         uio.uio_iov = &iov;
1273         uio.uio_iovcnt = 1;
1274         uio.uio_loffset = 0;
1275         uio.uio_segflg = UIO_USERSPACE;
1276         uio.uio_fmode = 0;
1277         /* For pipes copy should not bypass cache */
1278         uio.uio_extflg = UIO_COPY_CACHED;
1279         uio.uio_resid = iov.iov_len;
1280         count = fnp->fn_count;
1281         bp = fnp->fn_mp;
1282         while (count > 0 && uio.uio_resid) {
1283             cnt = MIN(uio.uio_resid, MBLKL(bp));
1284             if ((error = uiomove((char *)bp->b_rptr, cnt,
1285                 UIO_READ, &uio)) != 0) {
1286                 break;
1287             }
1288             count -= cnt;
1289             bp = bp->b_cont;
1290         }
1291         STRUCT_FSET(strpeek, databuf.len, len - uio.uio_resid);
1292     }
1293     STRUCT_FSET(strpeek, flags, 0);
1294     STRUCT_FSET(strpeek, ctlbuf.len, -1);
1295
1296     error = copyout(STRUCT_BUF(strpeek), (caddr_t)arg,
1297         STRUCT_SIZE(strpeek));
1298     if (error == 0 && len >= 0)
1299         *rvalp = 1;
1300     break;
1301 }
1302
1303 case FIONREAD:
1304     /*
1305     * let user know total number of bytes in message queue
1306     */
1307     error = copyout((caddr_t)&fnp->fn_count, (caddr_t)arg,
1308         sizeof(fnp->fn_count));
1309     if (error == 0)
1310         *rvalp = 0;
1311     break;
1312
1313 case I_SETSIG:
1314     /*
1315     * let streams set up the signal masking for us
1316     * we just check to see if it's set
1317     * XXX : this interface should not be visible
1318     * i.e. STREAM's framework is exposed.
1319     */

```

```

1320     error = strioctl(vp, cmd, arg, mode, U_TO_K, cr, rvalp);
1321     if (vp->v_stream->sd_sigflags & (S_INPUT|S_RDNORM|S_WRNORM))
1322         fnp->fn_flag |= FIFOSETSIG;
1323     else
1324         fnp->fn_flag &= ~FIFOSETSIG;
1325     break;
1326
1327 case I_FLUSH:
1328     /*
1329     * flush them message queues
1330     */
1331     if (arg & ~FLUSHRW) {
1332         error = EINVAL;
1333         break;
1334     }
1335     if (arg & FLUSHR) {
1336         fifo_fastflush(fnp);
1337     }
1338     fn_dest = fnp->fn_dest;
1339     if ((arg & FLUSHW)) {
1340         fifo_fastflush(fn_dest);
1341     }
1342     /*
1343     * wake up any sleeping readers or writers
1344     * (waking readers probably doesn't make sense, but it
1345     * doesn't hurt; i.e. we just got rid of all the data
1346     * what's to read ?)
1347     */
1348     if (fn_dest->fn_flag & (FIFOWANTW | FIFOWANTR)) {
1349         fn_dest->fn_flag &= ~(FIFOWANTW | FIFOWANTR);
1350         cv_broadcast(&fn_dest->fn_wait_cv);
1351     }
1352     *rvalp = 0;
1353     break;
1354
1355     /*
1356     * Since no band data can ever get on a fifo in fast mode
1357     * just return 0.
1358     */
1359 case I_FLUSHBAND:
1360     error = 0;
1361     *rvalp = 0;
1362     break;
1363
1364 case _I_GETPEERCRED:
1365     error = fifo_ioctl_getpeercred(fnp, arg, mode);
1366     break;
1367
1368 #endif /* ! codereview */
1369 /*
1370 * invalid calls for stream head or fifos
1371 */
1372
1373 case I_POP:
1374 case I_LOOK:
1375 case I_LINK:
1376 case I_PLINK:
1377 case I_UNLINK:
1378 case I_PUNLINK:
1379
1380     /*
1381     * more invalid tty type of ioctls
1382     */
1383
1384 case SRIOCSREDIR:
1385 case SRIOCISREDIR:

```

```

1386         error = EINVAL;
1387         break;
1389     }
1390     mutex_exit(&fn_lock->flk_lock);
1391     return (error);
1393 turn_fastoff:
1394     fifo_fastoff(fnp);
1396 stream_mode:
1397     /*
1398      * streams mode
1399      */
1400     mutex_exit(&fn_lock->flk_lock);
1401     return (fifo_striocctl(vp, cmd, arg, mode, cr, rvalp));
1403 }
1405 /*
1406  * FIFO is in STREAMS mode; STREAMS framework does most of the work.
1407  */
1408 static int
1409 fifo_striocctl(vnode_t *vp, int cmd, intptr_t arg, int mode,
1410              cred_t *cr, int *rvalp)
1411 {
1412     fifonode_t    *fnp = VTOF(vp);
1413     int           error;
1414     fifolock_t    *fn_lock;
1416     if (cmd == _I_GETPEERCRED)
1417         return (fifo_ioctl_getpeercred(fnp, arg, mode));
1419     if (cmd == _I_GETPEERCRED) {
1420         if (mode == FKIOCTL && fnp->fn_pcredp != NULL) {
1421             k_peercred_t *kp = (k_peercred_t *)arg;
1422             crhold(fnp->fn_pcredp);
1423             kp->pc_cr = fnp->fn_pcredp;
1424             kp->pc_cpuid = fnp->fn_cpuid;
1425             return (0);
1426         } else {
1427             return (ENOTSUP);
1428         }
1429     }
1430     error = striocctl(vp, cmd, arg, mode, U_TO_K, cr, rvalp);
1432     switch (cmd) {
1433     /*
1434      * The FIFOSEND flag is set to inform other processes that a file
1435      * descriptor is pending at the stream head of this pipe.
1436      * The flag is cleared and the sending process is awoken when
1437      * this process has completed receiving the file descriptor.
1438      * XXX This could become out of sync if the process does I_SENDFDs
1439      * and opens on connld attached to the same pipe.
1440      */
1441     case I_RECVFD:
1442     case I_E_RECVFD:
1443         if (error == 0) {
1444             fn_lock = fnp->fn_lock;
1445             mutex_enter(&fn_lock->flk_lock);
1446             if (fnp->fn_flag & FIFOSEND) {
1447                 fnp->fn_flag &= ~FIFOSEND;
1448                 cv_broadcast(&fnp->fn_dest->fn_wait_cv);
1449             }
1450             mutex_exit(&fn_lock->flk_lock);
1451         }
1452     }

```

```

1441         break;
1442     default:
1443         break;
1444     }
1446     return (error);
1447 }
    unchanged_portion_omitted

```