```
**********************************************************
    9282 Mon Nov 19 23:33:39 2018
new/usr/src/lib/libc/port/threads/pthr_mutex.c
9959 pthread_mutex_init should initialize mutex appropriately for robust mutex_i
Reviewed by: Jason King <jason.king@joyent.com>
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
______unchanged_portion_omitted_
```

```
201 /*
202  * pthread_mutex_init: Initializes the mutex object.  It copies the
203  * various attributes into one type argument and calls mutex_init().
204  */
205 #pragma weak _pthread_mutex_init = pthread_mutex_init
206 int
207 pthread_mutex_init(pthread_mutex_t *_RESTRICT_KYWD mutex,
208     const pthread_mutexattr_t *_RESTRICT_KYWD attr)
209 {
210         mattr_t *ap;
211         int     type;
212         int     prioceiling = 0;

214         /*
215          * All of the pshared, type, protocol, robust attributes
216          * translate to bits in the mutex_type field.
217          */
218         if (attr != NULL) {
219                 if ((ap = attr->__pthread_mutexattrp) == NULL)
220                         return (EINVAL);
221                 type = ap->pshared | ap->type | ap->protocol | ap->robustness;
222                 if (ap->protocol == PTHREAD_PRIO_PROTECT)
223                         prioceiling = ap->prioceiling;
224         } else {
225                 type = PTHREAD_PROCESS_PRIVATE | PTHREAD_MUTEX_DEFAULT |
226                     PTHREAD_PRIO_NONE | PTHREAD_MUTEX_STALLED;
227         }

229         /*
230          * POSIX mutexes (this interface) make no guarantee about the state of
231          * the mutex before pthread_mutex_init(3C) is called.  Sun mutexes, upon
232          * which these are built and which mutex_init(3C) below represents
233          * require that a robust mutex be initialized to all 0s _prior_ to
234          * mutex_init() being called, and that mutex_init() of an initialized
235          * mutex return EBUSY.
236          *
237          * We respect both these behaviors by zeroing the mutex here in the
238          * POSIX implementation if and only if the mutex magic is incorrect,
239          * and the mutex is robust.
240          */
241         if (((type & PTHREAD_MUTEX_ROBUST) != 0) &&
242             (((mutex_t *)mutex)->mutex_magic != MUTEX_MAGIC)) {
243                 (void) memset(mutex, 0, sizeof (*mutex));
244         }

246 #endif /* ! codereview */
247         return (mutex_init((mutex_t *)mutex, type, &prioceiling));
248 }

250 /*
251  * pthread_mutex_setprioceiling: sets the prioceiling.
252  * From the SUSv3 (POSIX) specification for pthread_mutex_setprioceiling():
253  *      The process of locking the mutex need not
254  *      adhere to the priority protect protocol.
255  * We pass the MUTEX_NOCEIL flag to mutex_lock_internal() so that
256  * a non-realtime thread can successfully execute this operation.
257  */
```



```
258 int
259 pthread_mutex_setprioceiling(pthread_mutex_t *mutex, int ceil, int *oceil)
260 {
261         mutex_t *mp = (mutex_t *)mutex;
262         const pcclass_t *pccp = get_info_by_policy(SCHED_FIFO);
263         int error;

265         if (!(mp->mutex_type & PTHREAD_PRIO_PROTECT) ||
266             ceil < pccp->pcc_primin || ceil > pccp->pcc_primax)
267                 return (EINVAL);
268         error = mutex_lock_internal(mp, NULL, MUTEX_LOCK | MUTEX_NOCEIL);
269         if (error == 0 || error == EOWNERDEAD || error == ELOCKUNMAPPED) {
270                 if (oceil)
271                         *oceil = mp->mutex_ceiling;
272                 mp->mutex_ceiling = ceil;
273                 error = mutex_unlock_internal(mp, 1);
274         }
275         return (error);
276 }

278 /*
279  * pthread_mutex_getprioceiling: gets the prioceiling.
280  */
281 #pragma weak _pthread_mutex_getprioceiling = pthread_mutex_getprioceiling
282 int
283 pthread_mutex_getprioceiling(const pthread_mutex_t *mp, int *ceiling)
284 {
285         *ceiling = ((mutex_t *)mp)->mutex_ceiling;
286         return (0);
287 }

289 /*
290  * UNIX98
291  * pthread_mutexattr_settype: sets the type attribute
292  */
293 int
294 pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type)
295 {
296         mattr_t *ap;

298         if (attr == NULL || (ap = attr->__pthread_mutexattrp) == NULL)
299                 return (EINVAL);
300         switch (type) {
301         case PTHREAD_MUTEX_NORMAL:
302                 type = LOCK_NORMAL;
303                 break;
304         case PTHREAD_MUTEX_ERRORCHECK:
305                 type = LOCK_ERRORCHECK;
306                 break;
307         case PTHREAD_MUTEX_RECURSIVE:
308                 type = LOCK_RECURSIVE | LOCK_ERRORCHECK;
309                 break;
310         default:
311                 return (EINVAL);
312         }
313         ap->type = type;
314         return (0);
315 }

317 /*
318  * UNIX98
319  * pthread_mutexattr_gettype: gets the type attribute.
320  */
321 int
322 pthread_mutexattr_gettype(const pthread_mutexattr_t *attr, int *typep)
323 {
```

```
324         mattr_t *ap;
325         int type;

327         if (attr == NULL || (ap = attr->__pthread_mutexattrp) == NULL ||
328             typep == NULL)
329                 return (EINVAL);
330         switch (ap->type) {
331         case LOCK_NORMAL:
332                 type = PTHREAD_MUTEX_NORMAL;
333                 break;
334         case LOCK_ERRORCHECK:
335                 type = PTHREAD_MUTEX_ERRORCHECK;
336                 break;
337         case LOCK_RECURSIVE | LOCK_ERRORCHECK:
338                 type = PTHREAD_MUTEX_RECURSIVE;
339                 break;
340         default:
341                 return (EINVAL);
342         }
343         *typep = type;
344         return (0);
345 }
```