多线程线程锁设置有问题 若节点释放却没有没有解链，会导致node->nd\_item.ci\_parent变量为空造成空指针解引用

1. **static** **int** configfs\_mkdir(**struct** inode \*dir, **struct** dentry \*dentry, umode\_t mode)
2. {
3. **int** ret = 0;
4. **int** module\_got = 0;
5. **struct** config\_group \*group = NULL;
6. **struct** config\_item \*item = NULL;
7. **struct** config\_item \*parent\_item;
8. **struct** configfs\_subsystem \*subsys;
9. **struct** configfs\_dirent \*sd;
10. **const** **struct** config\_item\_type \*type;
11. **struct** module \*subsys\_owner = NULL, \*new\_item\_owner = NULL;
12. **char** \*name;
14. sd = dentry->d\_parent->d\_fsdata;
16. /\*
17. \* Fake invisibility if dir belongs to a group/default groups hierarchy
18. \* being attached
19. \*/
20. **if** (!configfs\_dirent\_is\_ready(sd)) {
21. ret = -ENOENT;
22. **goto** out;
23. }
25. **if** (!(sd->s\_type & CONFIGFS\_USET\_DIR)) {
26. ret = -EPERM;
27. **goto** out;
28. }
30. /\* Get a working ref for the duration of this function \*/
31. parent\_item = configfs\_get\_config\_item(dentry->d\_parent);
32. type = parent\_item->ci\_type;
33. subsys = to\_config\_group(parent\_item)->cg\_subsys;
34. BUG\_ON(!subsys);
36. **if** (!type || !type->ct\_group\_ops ||
37. (!type->ct\_group\_ops->make\_group &&
38. !type->ct\_group\_ops->make\_item)) {
39. ret = -EPERM;  /\* Lack-of-mkdir returns -EPERM \*/
40. **goto** out\_put;
41. }
43. /\*
44. \* The subsystem may belong to a different module than the item
45. \* being created.  We don't want to safely pin the new item but
46. \* fail to pin the subsystem it sits under.
47. \*/
48. **if** (!subsys->su\_group.cg\_item.ci\_type) {
49. ret = -EINVAL;
50. **goto** out\_put;
51. }
52. subsys\_owner = subsys->su\_group.cg\_item.ci\_type->ct\_owner;
53. **if** (!try\_module\_get(subsys\_owner)) {
54. ret = -EINVAL;
55. **goto** out\_put;
56. }
58. name = kmalloc(dentry->d\_name.len + 1, GFP\_KERNEL);
59. **if** (!name) {
60. ret = -ENOMEM;
61. **goto** out\_subsys\_put;
62. }
64. snprintf(name, dentry->d\_name.len + 1, "%s", dentry->d\_name.name);
66. mutex\_lock(&subsys->su\_mutex);
67. **if** (type->ct\_group\_ops->make\_group) {
68. group = type->ct\_group\_ops->make\_group(to\_config\_group(parent\_item), name);
69. **if** (!group)
70. group = ERR\_PTR(-ENOMEM);
71. **if** (!IS\_ERR(group)) {
72. link\_group(to\_config\_group(parent\_item), group);
73. item = &group->cg\_item;
74. } **else**
75. ret = PTR\_ERR(group);
76. } **else** {
77. item = type->ct\_group\_ops->make\_item(to\_config\_group(parent\_item), name);
78. **if** (!item)
79. item = ERR\_PTR(-ENOMEM);
80. **if** (!IS\_ERR(item))
81. link\_obj(parent\_item, item);
82. **else**
83. ret = PTR\_ERR(item);
84. }
85. mutex\_unlock(&subsys->su\_mutex);
87. kfree(name);
88. **if** (ret) {
89. /\*
90. \* If ret != 0, then link\_obj() was never called.
91. \* There are no extra references to clean up.
92. \*/
93. **goto** out\_subsys\_put;
94. }
96. /\*
97. \* link\_obj() has been called (via link\_group() for groups).
98. \* From here on out, errors must clean that up.
99. \*/
101. type = item->ci\_type;
102. **if** (!type) {
103. ret = -EINVAL;
104. **goto** out\_unlink;
105. }
107. new\_item\_owner = type->ct\_owner;
108. **if** (!try\_module\_get(new\_item\_owner)) {
109. ret = -EINVAL;
110. **goto** out\_unlink;
111. }
113. /\*
114. \* I hate doing it this way, but if there is
115. \* an error,  module\_put() probably should
116. \* happen after any cleanup.
117. \*/
118. module\_got = 1;
120. /\*
121. \* Make racing rmdir() fail if it did not tag parent with
122. \* CONFIGFS\_USET\_DROPPING
123. \* Note: if CONFIGFS\_USET\_DROPPING is already set, attach\_group() will
124. \* fail and let rmdir() terminate correctly
125. \*/
126. spin\_lock(&configfs\_dirent\_lock);
127. /\* This will make configfs\_detach\_prep() fail \*/
128. sd->s\_type |= CONFIGFS\_USET\_IN\_MKDIR;
129. spin\_unlock(&configfs\_dirent\_lock);
131. **if** (group)
132. ret = configfs\_attach\_group(parent\_item, item, dentry);
133. **else**
134. ret = configfs\_attach\_item(parent\_item, item, dentry);
136. spin\_lock(&configfs\_dirent\_lock);
137. sd->s\_type &= ~CONFIGFS\_USET\_IN\_MKDIR;
138. **if** (!ret)
139. configfs\_dir\_set\_ready(dentry->d\_fsdata);
140. spin\_unlock(&configfs\_dirent\_lock);
142. out\_unlink:
143. **if** (ret) {
144. /\* Tear down everything we built up \*/
145. mutex\_lock(&subsys->su\_mutex);
147. client\_disconnect\_notify(parent\_item, item);
148. **if** (group)
149. unlink\_group(group);
150. **else**
151. unlink\_obj(item);
152. client\_drop\_item(parent\_item, item);
154. mutex\_unlock(&subsys->su\_mutex);
156. **if** (module\_got)
157. module\_put(new\_item\_owner);
158. }
160. out\_subsys\_put:
161. **if** (ret)
162. module\_put(subsys\_owner);
164. out\_put:
165. /\*
166. \* link\_obj()/link\_group() took a reference from child->parent,
167. \* so the parent is safely pinned.  We can drop our working
168. \* reference.
169. \*/
170. config\_item\_put(parent\_item);
172. out:
173. **return** ret;
174. }
175. **static** **void** unlink\_obj(**struct** config\_item \*item)
176. {
177. **struct** config\_group \*group;
179. group = item->ci\_group;
180. **if** (group) {
181. list\_del\_init(&item->ci\_entry);
183. item->ci\_group = NULL;
184. item->ci\_parent = NULL;
186. /\* Drop the reference for ci\_entry \*/
187. config\_item\_put(item);
189. /\* Drop the reference for ci\_parent \*/
190. config\_group\_put(group);
191. }
192. }
193. **static** ssize\_t o2nm\_node\_num\_store(**struct** config\_item \*item, **const** **char** \*page,
194. **size\_t** count)
195. {
196. **struct** o2nm\_node \*node = to\_o2nm\_node(item);
197. **struct** o2nm\_cluster \*cluster = to\_o2nm\_cluster\_from\_node(node);
198. unsigned **long** tmp;
199. **char** \*p = (**char** \*)page;
200. **int** ret = 0;
202. tmp = simple\_strtoul(p, &p, 0);
203. **if** (!p || (\*p && (\*p != '\n')))
204. **return** -EINVAL;
206. **if** (tmp >= O2NM\_MAX\_NODES)
207. **return** -ERANGE;
209. /\* once we're in the cl\_nodes tree networking can look us up by
210. \* node number and try to use our address and port attributes
211. \* to connect to this node.. make sure that they've been set
212. \* before writing the node attribute? \*/
213. **if** (!test\_bit(O2NM\_NODE\_ATTR\_ADDRESS, &node->nd\_set\_attributes) ||
214. !test\_bit(O2NM\_NODE\_ATTR\_PORT, &node->nd\_set\_attributes))
215. **return** -EINVAL; /\* XXX \*/
217. write\_lock(&cluster->cl\_nodes\_lock);
218. **if** (cluster->cl\_nodes[tmp])
219. ret = -EEXIST;
220. **else** **if** (test\_and\_set\_bit(O2NM\_NODE\_ATTR\_NUM,
221. &node->nd\_set\_attributes))
222. ret = -EBUSY;
223. **else**  {
224. cluster->cl\_nodes[tmp] = node;
225. node->nd\_num = tmp;
226. set\_bit(tmp, cluster->cl\_nodes\_bitmap);
227. }
228. write\_unlock(&cluster->cl\_nodes\_lock);
229. **if** (ret)
230. **return** ret;
232. **return** count;
233. }
234. **static** **struct** o2nm\_cluster \*to\_o2nm\_cluster\_from\_node(**struct** o2nm\_node \*node)
235. {
236. /\* through the first node\_set .parent
237. \* mycluster/nodes/mynode == o2nm\_cluster->o2nm\_node\_group->o2nm\_node \*/
238. **return** to\_o2nm\_cluster(node->nd\_item.ci\_parent->ci\_parent);
239. }