第 0076 讲 5 互斥锁分析

1、互斥锁

互斥锁（mutex）只允许一个进程进入临界区，适合保护比较长的临界区，因素竞争互斥锁的进程可能睡眠和再次唤醒，代价相当高。互斥锁是在原子操作 API 基础之上实现信号量行为。互斥锁不能进行递归锁或解锁，能用于交互上下文但不能用中断上下文，同一时刻只有一个任务持有该锁，而且只有这个任务可以对互斥锁进行解锁。

当无法获取锁的时候，线程进入睡眠等待状态。尽管可以将二值信号量当作互斥锁使用，但是 Linux 内核单独实现互斥锁。互斥锁具体结构体源码分析如下：

include > linux > C mutex.h > mutex > wait_lock

```
58  struct mutex {
59      atomic_long_t       owner;
60      spinlock_t      wait_lock; // 等待获得互斥锁中使用的自旋锁
61  #ifdef CONFIG_MUTEX_SPIN_ON_OWNER
62      struct optimistic_spin_queue osq; /* Spinner MCS lock */
63  #endif
64      struct list_head    wait_list;
65  #ifdef CONFIG_DEBUG_MUTEXES
66      void            *magic;
67  #endif
68  #ifdef CONFIG_DEBUG_LOCK_ALLOC
69      struct lockdep_map  dep_map;
70  #endif
71  };
72
```

初始化静态互斥锁与运行时动态初始化互斥锁方法如下：

include > linux > C mutex.h > ...

```
144 #define DEFINE_MUTEX(mutexname) \
145     struct mutex mutexname = __MUTEX_INITIALIZER(mutexname)
146
147 extern void __mutex_init(struct mutex *lock, const char *name,
148                          struct lock_class_key *key);
149
```

2、互斥锁申请与释放

include > linux > C mutex.h > mutex_lock_killable(mutex *)

```
189
190 extern void mutex_lock(struct mutex *lock);
191 extern int __must_check mutex_lock_interruptible(struct mutex *lock);
192 extern int __must_check mutex_lock_killable(struct mutex *lock);
193
```

include > linux > C mutex.h > mutex_lock_killable(mutex *)

```
204 /*
205  * NOTE: mutex_trylock() follows the spin_trylock() convention,
206  *       not the down_trylock() convention!
207  *
208  * Returns 1 if the mutex has been acquired successfully, and 0 on contention.
209  */
210 extern int mutex_trylock(struct mutex *lock);
211 extern void mutex_unlock(struct mutex *lock);
212
```

3、总结 mutex 要注意以下几点：

- mutex 一次只能有一个进程能持有互斥锁，只有锁的持有者能进行解锁操作；
- 禁止多次解锁操作， 禁止递归加锁操作，mutex 结构只能通过 API 进行初始化；
- mutex 结构禁止通过 memset 或者拷贝来进行初始化；
- 已经被持有 mutex 锁禁止被再次初始化；
- mutex 不允许在硬件或软件上下文（tasklets, timer）中使用。

备注：有时间大家可以研究一下实时互斥锁。

拿锁场景：

- 快速路径 mutex_lock-->mutex_trylock_fast(...)owner(task field 为空，falgs field 为空）。
- 慢速路径 mutex_lock-->mutex_lock_slowpath(...) owner(task field 为空，falgs field 不为空。

二、互斥锁详解

1、互斥锁 mutex_lock()分析

```
kernel > locking > C mutex.c > ...
282
283  void __sched mutex_lock(struct mutex *lock)
284  {
285      might_sleep();
286
287      if (!__mutex_trylock_fast(lock))
288          __mutex_lock_slowpath(lock);
289  }
290  EXPORT_SYMBOL(mutex_lock);
291  #endif
292
```

2、互斥锁 mutex_unlock 分析

```
kernel > locking > C mutex.c > ...
739
740  void __sched mutex_unlock(struct mutex *lock)
741  {
742  #ifndef CONFIG_DEBUG_LOCK_ALLOC
743      if (__mutex_unlock_fast(lock))
744          return;
745  #endif
746      __mutex_unlock_slowpath(lock, _RET_IP_);
747  }
748  EXPORT_SYMBOL(mutex_unlock);
749
```

释放之前获取的 mutex；mutex 只有被获取，才能调用这个函数来释放。

```
kernel > locking > C mutex.c > __mutex_unlock_fast(mutex *)
179
180 static __always_inline bool __mutex_unlock_fast(struct mutex *lock)
181 {
182     unsigned long curr = (unsigned long)current;
183
184     if (atomic_long_cmpxchg_release(&lock->owner, curr, 0UL) == curr)
185         return true;
186
187     return false;
188 }
```

```
kernel > locking > C mutex.c > __mutex_unlock_slowpath(mutex *, unsigned long)
1227
1228 /*
1229  * Release the lock, slowpath:
1230  */
1231 static noinline void __sched __mutex_unlock_slowpath(struct mutex *lock, unsigned long ip)
1232 {
1233     struct task_struct *next = NULL;
1234     DEFINE_WAKE_Q(wake_q);
1235     unsigned long owner;
```

如果没有配置 debug 或者直接解锁失败：直接执行另一个分支。

申请互斥锁函数：

Mutex_lock(...)：申请互斥锁，如果锁被占有，进程尝试睡眠；

Mutex_lock_interruptible(...)：申请互斥锁，如果锁被占有，进程轻度睡眠。