


 [torvalds / linux](#) Public[Code](#) [Pull requests](#) 312 [Actions](#) [Projects](#) [Security](#) [Insights](#) master ▾

...

[linux](#) / [include](#) / [linux](#) / [spinlock.h](#)

Sebastian Andrzej Siewior locking: Detect includes rwlock.h outside of spinlock.h ...

 History 51 contributors

+18

497 lines (426 sloc) | 13.8 KB

...

```
1  /* SPDX-License-Identifier: GPL-2.0 */
2  #ifndef __LINUX_SPINLOCK_H
3  #define __LINUX_SPINLOCK_H
4  #define __LINUX_INSIDE_SPINLOCK_H
5
6  /*
7   * include/linux/spinlock.h - generic spinlock/rwlock declarations
8   *
9   * here's the role of the various spinlock/rwlock related include files:
10  *
11  * on SMP builds:
12  *
13  *  asm/spinlock_types.h: contains the arch_spinlock_t/arch_rwlock_t and the
14  *                        initializers
15  *
16  *  linux/spinlock_types_raw:
17  *                        The raw types and initializers
18  *  linux/spinlock_types.h:
19  *                        defines the generic type and initializers
20  *
21  *  asm/spinlock.h:      contains the arch_spin_*/etc. lowlevel
22  *                        implementations, mostly inline assembly code
23  *
24  *  (also included on UP-debug builds:)
25  *
26  *  linux/spinlock_api_smp.h:
27  *                        contains the prototypes for the _spin_() APIs.
28  *
29  *  linux/spinlock.h:    builds the final spin_() APIs.
```

```
30  *
31  * on UP builds:
32  *
33  * linux/spinlock_type_up.h:
34  *             contains the generic, simplified UP spinlock type.
35  *             (which is an empty structure on non-debug builds)
36  *
37  * linux/spinlock_types_raw:
38  *             The raw RT types and initializers
39  * linux/spinlock_types.h:
40  *             defines the generic type and initializers
41  *
42  * linux/spinlock_up.h:
43  *             contains the arch_spin_*()/etc. version of UP
44  *             builds. (which are NOPs on non-debug, non-preempt
45  *             builds)
46  *
47  * (included on UP-non-debug builds:)
48  *
49  * linux/spinlock_api_up.h:
50  *             builds the _spin_*() APIs.
51  *
52  * linux/spinlock.h:    builds the final spin_*() APIs.
53  */
54
55 #include <linux/typecheck.h>
56 #include <linux/preempt.h>
57 #include <linux/linkage.h>
58 #include <linux/compiler.h>
59 #include <linux/irqflags.h>
60 #include <linux/thread_info.h>
61 #include <linux/stringify.h>
62 #include <linux/bottom_half.h>
63 #include <linux/lockdep.h>
64 #include <asm/barrier.h>
65 #include <asm/mmiowb.h>
66
67
68 /*
69  * Must define these before including other files, inline functions need them
70  */
71 #define LOCK_SECTION_NAME ".text..lock."KBUILD_BASENAME
72
73 #define LOCK_SECTION_START(extra)          \
74     ".subsection 1\n\t"                  \
75     extra                                \
76     ".ifndef " LOCK_SECTION_NAME "\n\t"  \
77     LOCK_SECTION_NAME ": \n\t"          \
78     ".endif\n\t"
```

```

79
80 #define LOCK_SECTION_END          \
81     ".previous\n\t"
82
83 #define __lockfunc __section(".spinlock.text")
84
85 /*
86  * Pull the arch_spinlock_t and arch_rwlock_t definitions:
87  */
88 #include <linux/spinlock_types.h>
89
90 /*
91  * Pull the arch_spin*() functions/declarations (UP-nondebug doesn't need them):
92  */
93 #ifdef CONFIG_SMP
94 # include <asm/spinlock.h>
95 #else
96 # include <linux/spinlock_up.h>
97 #endif
98
99 #ifdef CONFIG_DEBUG_SPINLOCK
100     extern void __raw_spin_lock_init(raw_spinlock_t *lock, const char *name,
101                                     struct lock_class_key *key, short inner);
102
103 # define raw_spin_lock_init(lock)          \
104 do {                                       \
105     static struct lock_class_key __key;   \
106                                           \
107     __raw_spin_lock_init((lock), #lock, &__key, LD_WAIT_SPIN); \
108 } while (0)
109
110 #else
111 # define raw_spin_lock_init(lock)          \
112 do { *(lock) = __RAW_SPIN_LOCK_UNLOCKED(lock); } while (0)
113 #endif
114
115 #define raw_spin_is_locked(lock)    arch_spin_is_locked(&(lock)->raw_lock)
116
117 #ifdef arch_spin_is_contended
118 #define raw_spin_is_contended(lock)    arch_spin_is_contended(&(lock)->raw_lock)
119 #else
120 #define raw_spin_is_contended(lock)    (((void)(lock), 0))
121 #endif /*arch_spin_is_contended*/
122
123 /*
124  * smp_mb__after_spinlock() provides the equivalent of a full memory barrier
125  * between program-order earlier lock acquisitions and program-order later
126  * memory accesses.
127  */

```

```

128 * This guarantees that the following two properties hold:
129 *
130 * 1) Given the snippet:
131 *
132 *     { X = 0;  Y = 0; }
133 *
134 *     CPU0                      CPU1
135 *
136 *     WRITE_ONCE(X, 1);          WRITE_ONCE(Y, 1);
137 *     spin_lock(S);              smp_mb();
138 *     smp_mb__after_spinlock();  r1 = READ_ONCE(X);
139 *     r0 = READ_ONCE(Y);
140 *     spin_unlock(S);
141 *
142 *     it is forbidden that CPU0 does not observe CPU1's store to Y (r0 = 0)
143 *     and CPU1 does not observe CPU0's store to X (r1 = 0); see the comments
144 *     preceding the call to smp_mb__after_spinlock() in __schedule() and in
145 *     try_to_wake_up().
146 *
147 * 2) Given the snippet:
148 *
149 *     { X = 0;  Y = 0; }
150 *
151 *     CPU0                      CPU1                      CPU2
152 *
153 *     spin_lock(S);              spin_lock(S);              r1 = READ_ONCE(Y);
154 *     WRITE_ONCE(X, 1);          smp_mb__after_spinlock();    smp_rmb();
155 *     spin_unlock(S);            r0 = READ_ONCE(X);            r2 = READ_ONCE(X);
156 *                                WRITE_ONCE(Y, 1);
157 *                                spin_unlock(S);
158 *
159 *     it is forbidden that CPU0's critical section executes before CPU1's
160 *     critical section (r0 = 1), CPU2 observes CPU1's store to Y (r1 = 1)
161 *     and CPU2 does not observe CPU0's store to X (r2 = 0); see the comments
162 *     preceding the calls to smp_rmb() in try_to_wake_up() for similar
163 *     snippets but "projected" onto two CPUs.
164 *
165 * Property (2) upgrades the lock to an RCsc lock.
166 *
167 * Since most load-store architectures implement ACQUIRE with an smp_mb() after
168 * the LL/SC loop, they need no further barriers. Similarly all our TSO
169 * architectures imply an smp_mb() for each atomic instruction and equally don't
170 * need more.
171 *
172 * Architectures that can implement ACQUIRE better need to take care.
173 */
174 #ifndef smp_mb__after_spinlock
175 #define smp_mb__after_spinlock()      kcsan_mb()
176 #endif

```

```
177
178 #ifdef CONFIG_DEBUG_SPINLOCK
179     extern void do_raw_spin_lock(raw_spinlock_t *lock) __acquires(lock);
180     extern int do_raw_spin_trylock(raw_spinlock_t *lock);
181     extern void do_raw_spin_unlock(raw_spinlock_t *lock) __releases(lock);
182 #else
183 static inline void do_raw_spin_lock(raw_spinlock_t *lock) __acquires(lock)
184 {
185     __acquire(lock);
186     arch_spin_lock(&lock->raw_lock);
187     mmio_wb_spin_lock();
188 }
189
190 static inline int do_raw_spin_trylock(raw_spinlock_t *lock)
191 {
192     int ret = arch_spin_trylock(&(lock)->raw_lock);
193
194     if (ret)
195         mmio_wb_spin_lock();
196
197     return ret;
198 }
199
200 static inline void do_raw_spin_unlock(raw_spinlock_t *lock) __releases(lock)
201 {
202     mmio_wb_spin_unlock();
203     arch_spin_unlock(&lock->raw_lock);
204     __release(lock);
205 }
206 #endif
207
208 /*
209  * Define the various spin_lock methods. Note we define these
210  * regardless of whether CONFIG_SMP or CONFIG_PREEMPTION are set. The
211  * various methods are defined as nops in the case they are not
212  * required.
213  */
214 #define raw_spin_trylock(lock) __cond_lock(lock, _raw_spin_trylock(lock))
215
216 #define raw_spin_lock(lock) _raw_spin_lock(lock)
217
218 #ifdef CONFIG_DEBUG_LOCK_ALLOC
219 # define raw_spin_lock_nested(lock, subclass) \
220     _raw_spin_lock_nested(lock, subclass)
221
222 # define raw_spin_lock_nest_lock(lock, nest_lock) \
223     do { \
224         typecheck(struct lockdep_map *, &(nest_lock)->dep_map); \
225         _raw_spin_lock_nest_lock(lock, &(nest_lock)->dep_map); \
```

```

226         } while (0)
227 #else
228 /*
229  * Always evaluate the 'subclass' argument to avoid that the compiler
230  * warns about set-but-not-used variables when building with
231  * CONFIG_DEBUG_LOCK_ALLOC=n and with W=1.
232  */
233 # define raw_spin_lock_nested(lock, subclass)          \
234     _raw_spin_lock(((void)(subclass), (lock)))
235 # define raw_spin_lock_nest_lock(lock, nest_lock)      _raw_spin_lock(lock)
236 #endif
237
238 #if defined(CONFIG_SMP) || defined(CONFIG_DEBUG_SPINLOCK)
239
240 #define raw_spin_lock_irqsave(lock, flags)              \
241     do {                                                  \
242         typecheck(unsigned long, flags);                 \
243         flags = _raw_spin_lock_irqsave(lock);            \
244     } while (0)
245
246 #ifndef CONFIG_DEBUG_LOCK_ALLOC
247 #define raw_spin_lock_irqsave_nested(lock, flags, subclass) \
248     do {                                                  \
249         typecheck(unsigned long, flags);                 \
250         flags = _raw_spin_lock_irqsave_nested(lock, subclass); \
251     } while (0)
252 #else
253 #define raw_spin_lock_irqsave_nested(lock, flags, subclass) \
254     do {                                                  \
255         typecheck(unsigned long, flags);                 \
256         flags = _raw_spin_lock_irqsave(lock);             \
257     } while (0)
258 #endif
259
260 #else
261
262 #define raw_spin_lock_irqsave(lock, flags)              \
263     do {                                                  \
264         typecheck(unsigned long, flags);                 \
265         _raw_spin_lock_irqsave(lock, flags);             \
266     } while (0)
267
268 #define raw_spin_lock_irqsave_nested(lock, flags, subclass) \
269     raw_spin_lock_irqsave(lock, flags)
270
271 #endif
272
273 #define raw_spin_lock_irq(lock)        _raw_spin_lock_irq(lock)
274 #define raw_spin_lock_bh(lock)         _raw_spin_lock_bh(lock)

```

```

275 #define raw_spin_unlock(lock)          _raw_spin_unlock(lock)
276 #define raw_spin_unlock_irq(lock)      _raw_spin_unlock_irq(lock)
277
278 #define raw_spin_unlock_irqrestore(lock, flags) \
279     do { \
280         typecheck(unsigned long, flags); \
281         _raw_spin_unlock_irqrestore(lock, flags); \
282     } while (0)
283 #define raw_spin_unlock_bh(lock)        _raw_spin_unlock_bh(lock)
284
285 #define raw_spin_trylock_bh(lock) \
286     __cond_lock(lock, _raw_spin_trylock_bh(lock))
287
288 #define raw_spin_trylock_irq(lock) \
289 ({ \
290     local_irq_disable(); \
291     raw_spin_trylock(lock) ? \
292     1 : ({ local_irq_enable(); 0; }); \
293 })
294
295 #define raw_spin_trylock_irqsave(lock, flags) \
296 ({ \
297     local_irq_save(flags); \
298     raw_spin_trylock(lock) ? \
299     1 : ({ local_irq_restore(flags); 0; }); \
300 })
301
302 #ifndef CONFIG_PREEMPT_RT
303 /* Include rwlock functions for !RT */
304 #include <linux/rwlock.h>
305 #endif
306
307 /*
308  * Pull the _spin_*()/_read_*()/_write_*() functions/declarations:
309  */
310 #if defined(CONFIG_SMP) || defined(CONFIG_DEBUG_SPINLOCK)
311 # include <linux/spinlock_api_smp.h>
312 #else
313 # include <linux/spinlock_api_up.h>
314 #endif
315
316 /* Non PREEMPT_RT kernel, map to raw spinlocks: */
317 #ifndef CONFIG_PREEMPT_RT
318
319 /*
320  * Map the spin_lock functions to the raw variants for PREEMPT_RT=n
321  */
322
323 static __always_inline raw_spinlock_t *spinlock_check(spinlock_t *lock)

```

```
324 {
325     return &lock->rlock;
326 }
327
328 #ifdef CONFIG_DEBUG_SPINLOCK
329
330 # define spin_lock_init(lock) \
331 do { \
332     static struct lock_class_key __key; \
333     \
334     __raw_spin_lock_init(spinlock_check(lock), \
335                          #lock, &__key, LD_WAIT_CONFIG); \
336 } while (0)
337
338 #else
339
340 # define spin_lock_init(_lock) \
341 do { \
342     spinlock_check(_lock); \
343     *(_lock) = __SPIN_LOCK_UNLOCKED(_lock); \
344 } while (0)
345
346 #endif
347
348 static __always_inline void spin_lock(spinlock_t *lock)
349 {
350     raw_spin_lock(&lock->rlock);
351 }
352
353 static __always_inline void spin_lock_bh(spinlock_t *lock)
354 {
355     raw_spin_lock_bh(&lock->rlock);
356 }
357
358 static __always_inline int spin_trylock(spinlock_t *lock)
359 {
360     return raw_spin_trylock(&lock->rlock);
361 }
362
363 #define spin_lock_nested(lock, subclass) \
364 do { \
365     raw_spin_lock_nested(spinlock_check(lock), subclass); \
366 } while (0)
367
368 #define spin_lock_nest_lock(lock, nest_lock) \
369 do { \
370     raw_spin_lock_nest_lock(spinlock_check(lock), nest_lock); \
371 } while (0)
372
```



```
373 static __always_inline void spin_lock_irq(spinlock_t *lock)
374 {
375     raw_spin_lock_irq(&lock->rlock);
376 }
377
378 #define spin_lock_irqsave(lock, flags) \
379 do { \
380     raw_spin_lock_irqsave(spinlock_check(lock), flags); \
381 } while (0)
382
383 #define spin_lock_irqsave_nested(lock, flags, subclass) \
384 do { \
385     raw_spin_lock_irqsave_nested(spinlock_check(lock), flags, subclass); \
386 } while (0)
387
388 static __always_inline void spin_unlock(spinlock_t *lock)
389 {
390     raw_spin_unlock(&lock->rlock);
391 }
392
393 static __always_inline void spin_unlock_bh(spinlock_t *lock)
394 {
395     raw_spin_unlock_bh(&lock->rlock);
396 }
397
398 static __always_inline void spin_unlock_irq(spinlock_t *lock)
399 {
400     raw_spin_unlock_irq(&lock->rlock);
401 }
402
403 static __always_inline void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)
404 {
405     raw_spin_unlock_irqrestore(&lock->rlock, flags);
406 }
407
408 static __always_inline int spin_trylock_bh(spinlock_t *lock)
409 {
410     return raw_spin_trylock_bh(&lock->rlock);
411 }
412
413 static __always_inline int spin_trylock_irq(spinlock_t *lock)
414 {
415     return raw_spin_trylock_irq(&lock->rlock);
416 }
417
418 #define spin_trylock_irqsave(lock, flags) \
419 ({ \
420     raw_spin_trylock_irqsave(spinlock_check(lock), flags); \
421 })
```

```
422
423 /**
424  * spin_is_locked() - Check whether a spinlock is locked.
425  * @lock: Pointer to the spinlock.
426  *
427  * This function is NOT required to provide any memory ordering
428  * guarantees; it could be used for debugging purposes or, when
429  * additional synchronization is needed, accompanied with other
430  * constructs (memory barriers) enforcing the synchronization.
431  *
432  * Returns: 1 if @lock is locked, 0 otherwise.
433  *
434  * Note that the function only tells you that the spinlock is
435  * seen to be locked, not that it is locked on your CPU.
436  *
437  * Further, on CONFIG_SMP=n builds with CONFIG_DEBUG_SPINLOCK=n,
438  * the return value is always 0 (see include/linux/spinlock_up.h).
439  * Therefore you should not rely heavily on the return value.
440  */
441 static __always_inline int spin_is_locked(spinlock_t *lock)
442 {
443     return raw_spin_is_locked(&lock->rlock);
444 }
445
446 static __always_inline int spin_is_contended(spinlock_t *lock)
447 {
448     return raw_spin_is_contended(&lock->rlock);
449 }
450
451 #define assert_spin_locked(lock)      assert_raw_spin_locked(&(lock)->rlock)
452
453 #else /* !CONFIG_PREEMPT_RT */
454 # include <linux/spinlock_rt.h>
455 #endif /* CONFIG_PREEMPT_RT */
456
457 /*
458  * Pull the atomic_t declaration:
459  * (asm-mips/atomic.h needs above definitions)
460  */
461 #include <linux/atomic.h>
462 /**
463  * atomic_dec_and_lock - lock on reaching reference count zero
464  * @atomic: the atomic counter
465  * @lock: the spinlock in question
466  *
467  * Decrements @atomic by 1. If the result is 0, returns true and locks
468  * @lock. Returns false for all other cases.
469  */
470 extern int _atomic_dec_and_lock(atomic_t *atomic, spinlock_t *lock);
```

```
471 #define atomic_dec_and_lock(atomic, lock) \
472     __cond_lock(lock, _atomic_dec_and_lock(atomic, lock))
473
474 extern int _atomic_dec_and_lock_irqsave(atomic_t *atomic, spinlock_t *lock,
475     unsigned long *flags);
476 #define atomic_dec_and_lock_irqsave(atomic, lock, flags) \
477     __cond_lock(lock, _atomic_dec_and_lock_irqsave(atomic, lock, &(flags)))
478
479 int __alloc_bucket_spinlocks(spinlock_t **locks, unsigned int *lock_mask,
480     size_t max_size, unsigned int cpu_mult,
481     gfp_t gfp, const char *name,
482     struct lock_class_key *key);
483
484 #define alloc_bucket_spinlocks(locks, lock_mask, max_size, cpu_mult, gfp) \
485     ({ \
486         static struct lock_class_key key; \
487         int ret; \
488         \
489         ret = __alloc_bucket_spinlocks(locks, lock_mask, max_size, \
490             cpu_mult, gfp, #locks, &key); \
491         ret; \
492     })
493
494 void free_bucket_spinlocks(spinlock_t *locks);
495
496 #undef __LINUX_INSIDE_SPINLOCK_H
497 #endif /* __LINUX_SPINLOCK_H */
```