1、atomic（原子类型）

std::atomic自定义类型的有可能不是lock-free的原因： 1、对齐问题：为了获得最佳的性能，数据需要对齐到特定的地址。不是所有的数据类型都可以保证这种对齐。当一个原子对象跨越多个缓存行时，这可能导致额外的开销，因为CPU需要检查多个缓存行来获取完整的数据。 2、大小问题：如果自定义类型的大小超过一个缓存行的大小，那么在执行原子操作时可能需要访问多个缓存行。这增加了操作的复杂性和开销。 3、实现复杂性：对于内置类型，编译器和标准库可以很容易地利用硬件指令来实现高效的原子操作。但对于复杂的自定义类型，实现高效的原子操作更为复杂。 4、硬件限制：不同的处理器和硬件平台可能有不同的原子操作支持和限制。这可能导致为某些自定义类型实现lock-free原子操作的困难。

2、spin lock(自旋锁)

Linux中的spin_lock其实就是一个int类型的变量，使用CPU的原子指令实现自旋锁，不会挂起线程，一直循环等待。

3、mutex（glibc版本为2.29）

__pthread_mutex_s:

```
struct __pthread_mutex_s
{
    int __lock __LOCK_ALIGNMENT;    //锁
    unsigned int __count;           //递归类型加锁次数
    int __owner;                    //当前持有锁的线程ID
    #if !__PTHREAD_MUTEX_NUSERS_AFTER_KIND
    unsigned int __nusers;
    #endif
    /* KIND must stay at this position in the structure to maintain
       binary compatibility with static initializers.  */
    int __kind;                     //锁的类型
    __PTHREAD_COMPAT_PADDING_MID
    #if __PTHREAD_MUTEX_NUSERS_AFTER_KIND
    unsigned int __nusers;          //当前等待和加锁的线程数
    #endif
    #if !__PTHREAD_MUTEX_USE_UNION
    __PTHREAD_SPINS_DATA;           //PTHREAD_MUTEX_ADAPTIVE_NP类型相关，用来计算最大自旋次数
    __pthread_list_t __list;        //线程列表
    # define __PTHREAD_MUTEX_HAVE_PREV      1
    #else
    __extension__ union
    {
        __PTHREAD_SPINS_DATA;
        __pthread_slist_t __list;
    };
    # define __PTHREAD_MUTEX_HAVE_PREV      0
```

```
    #endif
    __PTHREAD_COMPAT_PADDING_END
};
```

首先需要说明的是lock mutex并不是一定就会有用户态和内核态的切换（具体的实现再写），然后mutex的底层唤醒机制用的是futex。__pthread_mutex_lock:

```
int __pthread_mutex_lock (pthread_mutex_t *mutex)
{
    //获取mutex的类型
    unsigned int type = PTHREAD_MUTEX_TYPE_ELISION (mutex);

    //PTHREAD_MUTEX_ELISION_FLAGS_NP
    if (__builtin_expect (type & ~(PTHREAD_MUTEX_KIND_MASK_NP
                | PTHREAD_MUTEX_ELISION_FLAGS_NP), 0))
        return __pthread_mutex_lock_full (mutex);

    //PTHREAD_MUTEX_TIMED_NP 默认类型
    if (__glibc_likely (type == PTHREAD_MUTEX_TIMED_NP))
    {
        //调用lll_lock加锁
        ...
    }
    //PTHREAD_MUTEX_RECURSIVE_NP 递归类型
    else if (__builtin_expect (PTHREAD_MUTEX_TYPE (mutex)
                == PTHREAD_MUTEX_RECURSIVE_NP, 1))
    {
        //获取当前的线程ID，如果当前的线程ID等于mutex->__data.__owner，则累加mutex->__data.__count并返回
        //否则调用lll_lock加锁
        ...
    }
    //PTHREAD_MUTEX_ADAPTIVE_NP 这个类型为了避免用户态和内核态的切换，会做短暂的自旋操作
    else if (__builtin_expect (PTHREAD_MUTEX_TYPE (mutex)
            == PTHREAD_MUTEX_ADAPTIVE_NP, 1))
    {
        //尝试加锁，如果被锁则短暂的自旋
        if (...)
        {
            //短暂的自旋，超过最大次数则调用lll_lock加锁
            ...
        }
    }
    //PTHREAD_MUTEX_ERRORCHECK_NP类型,
    else
    {
        //如果mutex->__data.__owner和当前线程ID相等，则返回EDEADLK,
        //否则调用lll_lock (mutex)
        ...
    }
```

```
    //赋值mutex->__data.__owner等字段
    ...

    return 0;
}
```

详细代码截图如下所示：

```
int
__pthread_mutex_lock (pthread_mutex_t *mutex)
{
  unsigned int type = PTHREAD_MUTEX_TYPE_ELISION (mutex);

  LIBC_PROBE (mutex_entry, 1, mutex);

  if (__builtin_expect (type & ~(PTHREAD_MUTEX_KIND_MASK_NP
                                 | PTHREAD_MUTEX_ELISION_FLAGS_NP), 0))
    return __pthread_mutex_lock_full (mutex);

  if (__glibc_likely (type == PTHREAD_MUTEX_TIMED_NP))
    {
      FORCE_ELISION (mutex, goto elision);
    simple:
      /* Normal mutex.  */
      LLL_MUTEX_LOCK (mutex);
      assert (mutex->__data.__owner == 0);
    }
#ifdef HAVE_ELISION
  else if (__glibc_likely (type == PTHREAD_MUTEX_TIMED_ELISION_NP))
    {
  elision: __attribute__((unused))
      /* This case can never happen on a system without elision,
         as the mutex type initialization functions will not
     allow to set the elision flags.  */
      /* Don't record owner or users for elision case.  This is a
         tail call.  */
      return LLL_MUTEX_LOCK_ELISION (mutex);
    }
#endif
  else if (__builtin_expect (PTHREAD_MUTEX_TYPE (mutex)
                             == PTHREAD_MUTEX_RECURSIVE_NP, 1))
    {
      /* Recursive mutex.  */
      pid_t id = THREAD_GETMEM (THREAD_SELF, tid);

      /* Check whether we already hold the mutex.  */
      if (mutex->__data.__owner == id)
    {
      /* Just bump the counter.  */
      if (__glibc_unlikely (mutex->__data.__count + 1 == 0))
        /* Overflow of the counter.  */
        return EAGAIN;

      ++mutex->__data.__count;

      return 0;
```

```c
      return 0;
    }

    /* We have to get the mutex.  */
    LLL_MUTEX_LOCK (mutex);

    assert (mutex->__data.__owner == 0);
    mutex->__data.__count = 1;
  }
  else if (__builtin_expect (PTHREAD_MUTEX_TYPE (mutex)
              == PTHREAD_MUTEX_ADAPTIVE_NP, 1))
  {
    if (! __is_smp)
  goto simple;

    if (LLL_MUTEX_TRYLOCK (mutex) != 0)
  {
    int cnt = 0;
    int max_cnt = MIN (MAX_ADAPTIVE_COUNT,
                 mutex->__data.__spins * 2 + 10);
    do
      {
      if (cnt++ >= max_cnt)
      {
      LLL_MUTEX_LOCK (mutex);
      break;
      }
      atomic_spin_nop ();
      }
    while (LLL_MUTEX_TRYLOCK (mutex) != 0);

    mutex->__data.__spins += (cnt - mutex->__data.__spins) / 8;
  }
    assert (mutex->__data.__owner == 0);
  }
  else
  {
    pid_t id = THREAD_GETMEM (THREAD_SELF, tid);
    assert (PTHREAD_MUTEX_TYPE (mutex) == PTHREAD_MUTEX_ERRORCHECK_NP);
    /* Check whether we already hold the mutex.  */
    if (__glibc_unlikely (mutex->__data.__owner == id))
  return EDEADLK;
    goto simple;
  }

  pid_t id = THREAD_GETMEM (THREAD_SELF, tid);

  /* Record the ownership.  */
  mutex->__data.__owner = id;
#ifndef NO_INCR
  ++mutex->__data.__nusers;
#endif

  LIBC_PROBE (mutex_acquired, 1, mutex);

  return 0;
}
```

```
#define __lll_lock(futex, private)                                    \
  ((void)                                                             \
   ({                                                                 \
     int *__futex = (futex);                                          \
     if (__glibc_unlikely                                             \
         (atomic_compare_and_exchange_bool_acq (__futex, 1, 0)))     \
       {                                                              \
         if (__builtin_constant_p (private) && (private) == LLL_PRIVATE) \
           __lll_lock_wait_private (__futex);                         \
         else                                                         \
           __lll_lock_wait (__futex, private);                        \
       }                                                              \
   }))
#define lll_lock(futex, private)  \
  __lll_lock (&(futex), private)
```

__lll_lock_wait_private汇编代码中，当futex被唤醒后，还会有CAS操作来避免内核的虚假唤醒。

__pthread_mutex_unlock:

```
int
attribute_hidden
__pthread_mutex_unlock_usercnt (pthread_mutex_t *mutex, int decr)
{
    int type = PTHREAD_MUTEX_TYPE_ELISION (mutex);
    if (__builtin_expect (type &
            ~(PTHREAD_MUTEX_KIND_MASK_NP|PTHREAD_MUTEX_ELISION_FLAGS_NP), 0))
        return __pthread_mutex_unlock_full (mutex, decr);

    //PTHREAD_MUTEX_TIMED_NP 默认类型
    if (__builtin_expect (type, PTHREAD_MUTEX_TIMED_NP)
        == PTHREAD_MUTEX_TIMED_NP)
    {
        //重置mutex->__data.__owner, mutex->__data.__nusers字段减1，并且调用
lll_unlock解锁
        ...
    }
    else if (__glibc_likely (type == PTHREAD_MUTEX_TIMED_ELISION_NP))
        {
        /* Don't reset the owner/users fields for elision.  */
        return lll_unlock_elision (mutex->__data.__lock, mutex->__data.__elision,
                                PTHREAD_MUTEX_PSHARED (mutex));
        }
    //PTHREAD_MUTEX_RECURSIVE_NP 递归类型
    else if (__builtin_expect (PTHREAD_MUTEX_TYPE (mutex)
                            == PTHREAD_MUTEX_RECURSIVE_NP, 1))
    {
        //如果mutex->__data.__owner不是当前线程，返回EPERM
        ...

        //mutex->__data.__count做减1操作，如果结果不为0直接返回。某一线程重复加锁
        ...
```

```
        //重置mutex->__data.__owner，mutex->__data.__nusers字段减1，并且调用
lll_unlock解锁
        ...
    }
    //PTHREAD_MUTEX_ADAPTIVE_NP
    else if (__builtin_expect (PTHREAD_MUTEX_TYPE (mutex)
                == PTHREAD_MUTEX_ADAPTIVE_NP, 1))
    {
        //重置mutex->__data.__owner，mutex->__data.__nusers字段减1，并且调用
lll_unlock解锁
        ...
    }
    //PTHREAD_MUTEX_ERRORCHECK_NP
    else
    {
        //检查mutex->__data.__owner和当前线程ID相等，以及当前是否被锁，失败则返回EPERM
        ...

        //重置mutex->__data.__owner，mutex->__data.__nusers字段减1，并且调用
lll_unlock解锁
        ...
    }
}
```

详细代码截图如下所示：

```c
int
attribute_hidden
__pthread_mutex_unlock_usercnt (pthread_mutex_t *mutex, int decr)
{
  int type = PTHREAD_MUTEX_TYPE_ELISION (mutex);
  if (__builtin_expect (type &
    ~(PTHREAD_MUTEX_KIND_MASK_NP|PTHREAD_MUTEX_ELISION_FLAGS_NP), 0))
    return __pthread_mutex_unlock_full (mutex, decr);

  if (__builtin_expect (type, PTHREAD_MUTEX_TIMED_NP)
      == PTHREAD_MUTEX_TIMED_NP)
    {
      /* Always reset the owner field.  */
    normal:
      mutex->__data.__owner = 0;
      if (decr)
  /* One less user.  */
  --mutex->__data.__nusers;

      /* Unlock.  */
      lll_unlock (mutex->__data.__lock, PTHREAD_MUTEX_PSHARED (mutex));

      LIBC_PROBE (mutex_release, 1, mutex);

      return 0;
    }
  else if (__glibc_likely (type == PTHREAD_MUTEX_TIMED_ELISION_NP))
    {
      /* Don't reset the owner/users fields for elision.  */
      return lll_unlock_elision (mutex->__data.__lock, mutex->__data.__elision,
              PTHREAD_MUTEX_PSHARED (mutex));
    }
  else if (__builtin_expect (PTHREAD_MUTEX_TYPE (mutex)
            == PTHREAD_MUTEX_RECURSIVE_NP, 1))
    {
      /* Recursive mutex.  */
      if (mutex->__data.__owner != THREAD_GETMEM (THREAD_SELF, tid))
  return EPERM;

      if (--mutex->__data.__count != 0)
  /* We still hold the mutex.  */
  return 0;
      goto normal;
    }
  else if (__builtin_expect (PTHREAD_MUTEX_TYPE (mutex)
            == PTHREAD_MUTEX_ADAPTIVE_NP, 1))
    goto normal;
  else
    {
      /* Error checking mutex.  */
      assert (type == PTHREAD_MUTEX_ERRORCHECK_NP);
      if (mutex->__data.__owner != THREAD_GETMEM (THREAD_SELF, tid)
    || ! lll_islocked (mutex->__data.__lock))
  return EPERM;
      goto normal;
    }
}
```

```
#define __lll_unlock(futex, private)                    \
  ((void)                                               \
   ({                                                   \
     int *__futex = (futex);                            \
     int __private = (private);                         \
     int __oldval = atomic_exchange_rel (__futex, 0);   \
     if (__glibc_unlikely (__oldval > 1))               \
       lll_futex_wake (__futex, 1, __private);          \
   }))
#define lll_unlock(futex, private)  \
  __lll_unlock (&(futex), private)
```

PTHREAD_MUTEX_TIMED_NP(默认类型)、PTHREAD_MUTEX_ADAPTIVE_NP在解锁的时候没有判断锁的持有线程是否为当前线程，所以需要注意加锁解锁相对应，也可以用PTHREAD_MUTEX_ERRORCHECK_NP，会做检查，只是性能稍差。

4、条件变量

```
static __always_inline int
__pthread_cond_wait_common (pthread_cond_t *cond, pthread_mutex_t *mutex,
    const struct timespec *abstime)
{
    //释放mutex
    ...

    //调用futex(FUTEX_WAIT, …)
    ...

    //被唤醒后重新尝试获取mutex
    ...
}
```

```
int
__pthread_cond_signal (pthread_cond_t *cond)
{
    //调用futex(FUTEX_WAKE, …)
    ...
}
```

5、信号量

new_sem:

```
struct new_sem
{
```

```
    #if __HAVE_64B_ATOMICS
        /* The data field holds both value (in the least-significant 32 bits) and
           nwaiters.  */
        # if __BYTE_ORDER == __LITTLE_ENDIAN
        #  define SEM_VALUE_OFFSET 0
        # elif __BYTE_ORDER == __BIG_ENDIAN
        #  define SEM_VALUE_OFFSET 1
        # else
        # error Unsupported byte order.
        # endif
        # define SEM_NWAITERS_SHIFT 32
        # define SEM_VALUE_MASK (~(unsigned int)0)
        uint64_t data;      //futex监听的地址
        int private;        //是否单个进程私有
        int pad;
    #else
        # define SEM_VALUE_SHIFT 1
        # define SEM_NWAITERS_MASK ((unsigned int)1)
        unsigned int value;
        int private;
        int pad;
        unsigned int nwaiters;
    #endif
};
```

__new_sem_post:

```
int __new_sem_post (sem_t *sem)
{
    struct new_sem *isem = (struct new_sem *) sem;
    int private = isem->private;

    //当前值
    uint64_t d = atomic_load_relaxed (&isem->data);
    do
    {
        //是否超过最大值SEM_VALUE_MAX（int类型的最大值）
        if ((d & SEM_VALUE_MASK) == SEM_VALUE_MAX)
        {
            __set_errno (EOVERFLOW);
            return -1;
        }
    }
    //CAS操作，data字段+1
    while (!atomic_compare_exchange_weak_release (&isem->data, &d, d + 1));

    //如果大于0，则唤醒一个线程
    if ((d >> SEM_NWAITERS_SHIFT) > 0)
        futex_wake (((unsigned int *) &isem->data) + SEM_VALUE_OFFSET, 1,
private);
```

```
        return 0;
    }
```

详细代码截图如下所示：

```
/* See sem_wait for an explanation of the algorithm.  */
int
__new_sem_post (sem_t *sem)
{
  struct new_sem *isem = (struct new_sem *) sem;
  int private = isem->private;

#if __HAVE_64B_ATOMICS
  /* Add a token to the semaphore.  We use release MO to make sure that a
     thread acquiring this token synchronizes with us and other threads that
     added tokens before (the release sequence includes atomic RMW operations
     by other threads).  */
  /* TODO Use atomic_fetch_add to make it scale better than a CAS loop?  */
  uint64_t d = atomic_load_relaxed (&isem->data);
  do
    {
      if ((d & SEM_VALUE_MASK) == SEM_VALUE_MAX)
  {
    __set_errno (EOVERFLOW);
    return -1;
  }
    }
  while (!atomic_compare_exchange_weak_release (&isem->data, &d, d + 1));

  /* If there is any potentially blocked waiter, wake one of them.  */
  if ((d >> SEM_NWAITERS_SHIFT) > 0)
    futex_wake (((unsigned int *) &isem->data) + SEM_VALUE_OFFSET, 1, private);
#else
  /* Add a token to the semaphore.  Similar to 64b version.  */
  unsigned int v = atomic_load_relaxed (&isem->value);
  do
    {
      if ((v >> SEM_VALUE_SHIFT) == SEM_VALUE_MAX)
  {
    __set_errno (EOVERFLOW);
    return -1;
  }
    }
  while (!atomic_compare_exchange_weak_release
   (&isem->value, &v, v + (1 << SEM_VALUE_SHIFT)));

  /* If there is any potentially blocked waiter, wake one of them.  */
  if ((v & SEM_NWAITERS_MASK) != 0)
    futex_wake (&isem->value, 1, private);
#endif

  return 0;
}
```

__new_sem_wait:

```
int
__new_sem_wait (sem_t *sem)
{
    //如果data字段大于0，并且CAS data字段做减1操作成功，则直接返回0
    if (...)
        return 0;
    else
    {
        //如果data字段等于0，则调用futex wait，被唤醒后原子操作data字段做减1操作
        if (data == 0)
        {
            futex(wait);
            cas(data -1);
        }
        //
        else
        {
            cas(data-1);
        }
    }
}
```

详细代码截图如下所示：

```
int
__new_sem_wait (sem_t *sem)
{
  /* We need to check whether we need to act upon a cancellation request here
     because POSIX specifies that cancellation points "shall occur" in
     sem_wait and sem_timedwait, which also means that they need to check
     this regardless whether they block or not (unlike "may occur"
     functions).  See the POSIX Rationale for this requirement: Section
     "Thread Cancellation Overview" [1] and austin group issue #1076 [2]
     for thoughs on why this may be a suboptimal design.

     [1] http://pubs.opengroup.org/onlinepubs/9699919799/xrat/V4_xsh_chap02.html
     [2] http://austingroupbugs.net/view.php?id=1076 for thoughts on why this
   */
  __pthread_testcancel ();

  if (__new_sem_wait_fast ((struct new_sem *) sem, 0) == 0)
    return 0;
  else
    return __new_sem_wait_slow((struct new_sem *) sem, NULL);
}
```

```
static int
__new_sem_wait_fast (struct new_sem *sem, int definitive_result)
{
  /* We need acquire MO if we actually grab a token, so that this
     synchronizes with all token providers (i.e., the RMW operation we read
     from or all those before it in modification order; also see sem_post).
     We do not need to guarantee any ordering if we observed that there is
     no token (POSIX leaves it unspecified whether functions that fail
     synchronize memory); thus, relaxed MO is sufficient for the initial load
     and the failure path of the CAS.  If the weak CAS fails and we need a
     definitive result, retry.  */
#if __HAVE_64B_ATOMICS
  uint64_t d = atomic_load_relaxed (&sem->data);
  do
    {
      if ((d & SEM_VALUE_MASK) == 0)
  break;
      if (atomic_compare_exchange_weak_acquire (&sem->data, &d, d - 1))
  return 0;
    }
  while (definitive_result);
  return -1;
#else
  unsigned int v = atomic_load_relaxed (&sem->value);
  do
    {
      if ((v >> SEM_VALUE_SHIFT) == 0)
  break;
      if (atomic_compare_exchange_weak_acquire (&sem->value,
    &v, v - (1 << SEM_VALUE_SHIFT)))
  return 0;
    }
  while (definitive_result);
  return -1;
#endif
}
```

```
/* Slow path that blocks.  */
static int
__attribute__ ((noinline))
__new_sem_wait_slow (struct new_sem *sem, const struct timespec *abstime)
{
  int err = 0;

#if __HAVE_64B_ATOMICS
  /* Add a waiter.  Relaxed MO is sufficient because we can rely on the
     ordering provided by the RMW operations we use.  */
  uint64_t d = atomic_fetch_add_relaxed (&sem->data,
      (uint64_t) 1 << SEM_NWAITERS_SHIFT);

  pthread_cleanup_push (__sem_wait_cleanup, sem);

  /* Wait for a token to be available.  Retry until we can grab one.  */
  for (;;)
    {
      /* If there is no token available, sleep until there is.  */
      if ((d & SEM_VALUE_MASK) == 0)
  {
    err = do_futex_wait (sem, abstime);
    /* A futex return value of 0 or EAGAIN is due to a real or spurious
```

```
	     wake-up, or due to a change in the number of tokens.  We retry in
	     these cases.
	     If we timed out, forward this to the caller.
	     EINTR is returned if we are interrupted by a signal; we
	     forward this to the caller.  (See futex_wait and related
	     documentation.  Before Linux 2.6.22, EINTR was also returned on
	     spurious wake-ups; we only support more recent Linux versions,
	     so do not need to consider this here.)  */
	  if (err == ETIMEDOUT || err == EINTR)
	    {
	      __set_errno (err);
	      err = -1;
	      /* Stop being registered as a waiter.  */
	      atomic_fetch_add_relaxed (&sem->data,
		  -((uint64_t) 1 << SEM_NWAITERS_SHIFT));
	      break;
	    }
	  /* Relaxed MO is sufficient; see below.  */
	  d = atomic_load_relaxed (&sem->data);
	}
      else
	{
	  /* Try to grab both a token and stop being a waiter.  We need
	     acquire MO so this synchronizes with all token providers (i.e.,
	     the RMW operation we read from or all those before it in
	     modification order; also see sem_post).  On the failure path,
	     relaxed MO is sufficient because we only eventually need the
	     up-to-date value; the futex_wait or the CAS perform the real
	     work.  */
	  if (atomic_compare_exchange_weak_acquire (&sem->data,
	      &d, d - 1 - ((uint64_t) 1 << SEM_NWAITERS_SHIFT)))
	    {
	      err = 0;
	      break;
	    }
	}
    }

  pthread_cleanup_pop (0);
#else
  /* The main difference to the 64b-atomics implementation is that we need to
     access value and nwaiters in separate steps, and that the nwaiters bit
     in the value can temporarily not be set even if nwaiters is nonzero.
     We work around incorrectly unsetting the nwaiters bit by letting sem_wait
     set the bit again and waking the number of waiters that could grab a
     token.  There are two additional properties we need to ensure:
     (1) We make sure that whenever unsetting the bit, we see the increment of
     nwaiters by the other thread that set the bit.  IOW, we will notice if
     we make a mistake.
     (2) When setting the nwaiters bit, we make sure that we see the unsetting
     of the bit by another waiter that happened before us.  This avoids having
     to blindly set the bit whenever we need to block on it.  We set/unset
     the bit while having incremented nwaiters (i.e., are a registered
     waiter), and the problematic case only happens when one waiter indeed
     followed another (i.e., nwaiters was never larger than 1); thus, this
     works similarly as with a critical section using nwaiters (see the MOs
     and related comments below).

     An alternative approach would be to unset the bit after decrementing
     nwaiters; however, that would result in needing Dekker-like
     synchronization and thus full memory barriers.  We also would not be able
     to prevent misspeculation, so this alternative scheme does not seem
     beneficial.  */
  unsigned int v;

  /* Add a waiter.  We need acquire MO so this synchronizes with the release
     MO we use when decrementing nwaiters below; it ensures that if another
```

```
      waiter unset the bit before us, we see that and set it again.  Also see
      property (2) above.  */
  atomic_fetch_add_acquire (&sem->nwaiters, 1);

  pthread_cleanup_push (__sem_wait_cleanup, sem);

  /* Wait for a token to be available.  Retry until we can grab one.  */
  /* We do not need any ordering wrt. to this load's reads-from, so relaxed
     MO is sufficient.  The acquire MO above ensures that in the problematic
     case, we do see the unsetting of the bit by another waiter.  */
  v = atomic_load_relaxed (&sem->value);
  do
    {
      do
  {
    /* We are about to block, so make sure that the nwaiters bit is
       set.  We need release MO on the CAS to ensure that when another
       waiter unsets the nwaiters bit, it will also observe that we
       incremented nwaiters in the meantime (also see the unsetting of
       the bit below).  Relaxed MO on CAS failure is sufficient (see
       above).  */
    do
      {
        if ((v & SEM_NWAITERS_MASK) != 0)
    break;
      }
    while (!atomic_compare_exchange_weak_release (&sem->value,
        &v, v | SEM_NWAITERS_MASK));
    /* If there is no token, wait.  */
    if ((v >> SEM_VALUE_SHIFT) == 0)
      {
        /* See __HAVE_64B_ATOMICS variant.  */
        err = do_futex_wait(sem, abstime);
        if (err == ETIMEDOUT || err == EINTR)
    {
      __set_errno (err);
      err = -1;
      goto error;
    }
        err = 0;
        /* We blocked, so there might be a token now.  Relaxed MO is
     sufficient (see above).  */
        v = atomic_load_relaxed (&sem->value);
      }
  }
      /* If there is no token, we must not try to grab one.  */
      while ((v >> SEM_VALUE_SHIFT) == 0);
    }
  /* Try to grab a token.  We need acquire MO so this synchronizes with
     all token providers (i.e., the RMW operation we read from or all those
     before it in modification order; also see sem_post).  */
  while (!atomic_compare_exchange_weak_acquire (&sem->value,
      &v, v - (1 << SEM_VALUE_SHIFT)));

error:
  pthread_cleanup_pop (0);

  __sem_wait_32_finish (sem);
#endif

  return err;
}
```

通过源码的注释也能知道当futex被唤醒后，还会有CAS操作来避免内核的虚假唤醒。

6、三者比较

- 1、关于条件变量的虚假唤醒
  首先需要说明的是三种底层唤醒机制都是用的futex，但只有条件变量存在虚假唤醒。这是有两个方面导致的，条件变量的实现以及内核的原因，解释如下：
  实现方面：futex唤醒和mutex加锁之间的竞态，上面代码中有介绍pthread_cond_wait内部的行为大致为：1、释放mutex。2、调用futex(FUTEX_WAIT, ...)进入休眠。3、被pthread_cond_signal或pthread_cond_broadcast唤醒。4、重新尝试获取 mutex。由于步骤3和4不是原子的，多个被唤醒的线程可能会竞争mutex，导致一些线程在真正获取mutex之前再次进入等待。
  内核方面：futex_wait()可能会被虚假唤醒（例如：信号、调度、内核 Bug 等因素）。
  而互斥锁和信号量不会因为Linux内核可能出现的spurious wakeup导致虚假唤醒，这是因为互斥锁和信号量futex被唤醒后，还会通过原子操作检查监控的值是否满足条件。