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、Mutex

Linux中的互斥锁（Mutex）是一种用于保护共享资源的同步原语。其底层实现原理主要基于内核提供的原子操作和自旋锁机制。 互斥锁的实现主要包括以下几个关键部分： 1、原子操作：Linux内核提供了一组原子操作函数，这些函数可以在多线程环境中安全地执行一些基本的操作，如增加、减少、设置等。这些原子操作可以确保在多线程环境中对共享资源的操作是原子的，不会被其他线程打断。 2、自旋锁：自旋锁是一种特殊的锁，当一个线程试图获取自旋锁但该锁已经被其他线程持有时，该线程会一直循环等待（自旋），直到获得锁为止。Linux中的自旋锁与互斥锁紧密相关，互斥锁的实现通常会使用自旋锁来确保 线程安全地获取和释放锁。 3、互斥锁结构体：内核定义了一个互斥锁的数据结构，该结构体包含了与互斥锁相关的各种信息，如指向自旋锁的指针、指向等待队列的指针等。 总的来说，Linux中互斥锁的底层实现原理主要是基于原子操作和自旋锁机制，通过这些机制来确保在多线程环境中对共享资源的互斥访问。 部分源码截图如下： mutex结构体：

```
struct __pthread_mutex_s
{
  int __lock __LOCK_ALIGNMENT;  //锁
  unsigned int __count;         //递归类型互斥锁相关，记录本线程锁住次数
  int __owner;                  //所属线程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;                   //互斥锁的类型
  __PTHREAD_COMPAT_PADDING_MID
#if __PTHREAD_MUTEX_NUSERS_AFTER_KIND
  unsigned int __nusers;
#endif
#if !__PTHREAD_MUTEX_USE_UNION
  __PTHREAD_SPINS_DATA;
  __pthread_list_t __list;
# 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
};
```

mutex的lock和unlock：

```
/* Wait on address PTR, without blocking if its contents
 * are different from VAL.  */
//1、将等待项添加到等待队列中。
//2、将当前进程（或线程）状态设置为等待状态，并将其加入到调度器中进行调度。
//3、释放当前进程（或线程）的CPU资源，使其可以被其他进程使用。
//4、当等待条件成立时，通过唤醒机制将等待项从等待队列中移除，并将等待进程（或线程）重新调度到CPU上继续执行。
#define lll_wait(ptr, val, flags)   \
  __gsync_wait (__mach_task_self (),   \
    (vm_offset_t)(ptr), (val), 0, 0, (flags))

/* Wake one or more threads waiting on address PTR.  */
#define lll_wake(ptr, flags)   \
  __gsync_wake (__mach_task_self (), (vm_offset_t)(ptr), 0, (flags))

/* Acquire the lock at PTR.  */
#define lll_lock(ptr, flags)   \
  ({   \
    int *__iptr = (int *)(ptr);   \
    int __flags = (flags);   \
    if (*__iptr != 0 ||   \
        atomic_compare_and_exchange_bool_acq (__iptr, 1, 0) != 0)   \
      while (1)   \
        {   \
          if (atomic_exchange_acq (__iptr, 2) == 0)   \
            break;   \
          lll_wait (__iptr, 2, __flags);   \ //挂起线程
        }   \
    (void)0;   \
  })

/* Try to acquire the lock at PTR, without blocking.
   Evaluates to zero on success.  */
#define lll_trylock(ptr)   \
  ({   \
    int *__iptr = (int *)(ptr);   \
    *__iptr == 0 &&   \
      atomic_compare_and_exchange_bool_acq (__iptr, 1, 0) == 0 ? 0 : -1;   \
  })

/* Release the lock at PTR.  */
#define lll_unlock(ptr, flags)   \
  ({   \
    int *__iptr = (int *)(ptr);   \
    if (atomic_exchange_rel (__iptr, 0) == 2)   \
      lll_wake (__iptr, (flags));   \
    (void)0;   \
  })

#endif
```

3、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。所以单核CPU不能使用自旋锁(当然现在应该也没有单核的CPU)。

4、条件变量

pthread_cond_wait：

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

  //unlock mutex
  int err = __pthread_mutex_unlock_usercnt (mutex, 0);
  if (__glibc_unlikely (err != 0))
  {
    __condvar_cancel_waiting (cond, seq, g, private);
    __condvar_confirm_wakeup (cond, private);
    return err;
  }

  ...

  //非原子操作，可能会造成虚假唤醒
  do
  {
    while (1)
    {
      ...

      //挂起线程，futex 是一种轻量级的锁机制，用于快速地进行互斥操作。大概流程如下
      //1、将当前线程添加到指定的等待队列中。
      //2、检查 futex 的值是否为预期的值（通常为0）。如果是，则表示可以立即获得锁，函数返回成功。
      //3、如果 futex 的值不等于预期的值，表示需要等待。设置一个定时器或检查点，以便在指定的时间或条件满足时重新评估 futex 的值。
      //4、将当前线程置于等待状态，并释放 CPU 资源。
      //5、当 futex 的值发生变化或定时器/检查点触发时，函数会重新评估 futex 的值。如果此时 futex 的值已经发生了变化，说明有其他线程释放了锁，函数返回成功。
      //6、如果在等待期间收到取消请求，函数会检查取消状态并相应地处理取消操作。
      int err = futex_wait_cancelable (
      cond->__data.__g_signals + g, 0, private);

      ...
    }
  }
  while (!atomic_compare_exchange_weak_acquire (cond->__data.__g_signals + g,
                                                &signals, signals - 2));

  ...

  //lock mutex
  err = __pthread_mutex_cond_lock (mutex);
  return (err != 0) ? err : result;
}
```

5、信号量

sem_post：

```
int __new_sem_post (sem_t *sem)
{
  ...

  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 ((d >> SEM_NWAITERS_SHIFT) > 0)
    futex_wake (((unsigned int *) &isem->data) + SEM_VALUE_OFFSET, 1, private);

  return 0;
}
```

sem_wait：

```
static int
__attribute__ ((noinline))
__new_sem_wait_slow (struct new_sem *sem, const struct timespec *abstime)
{
  int err = 0;

  ...

  uint64_t d = atomic_fetch_add_relaxed (&sem->data,
      (uint64_t) 1 << SEM_NWAITERS_SHIFT);

  for (;;)
  {
    if ((d & SEM_VALUE_MASK) == 0)
    {
      err = do_futex_wait (sem, abstime);

      if (err == ETIMEDOUT || err == EINTR)
      {
        __set_errno (err);
        err = -1;
        atomic_fetch_add_relaxed (&sem->data,
       -((uint64_t) 1 << SEM_NWAITERS_SHIFT));
        break;
      }
      d = atomic_load_relaxed (&sem->data);
    }
    else
    {
      if (atomic_compare_exchange_weak_acquire (&sem->data,
        &d, d - 1 - ((uint64_t) 1 << SEM_NWAITERS_SHIFT)))
      {
        err = 0;
        break;
      }
    }
  }

  ...

  return err;
}
```

6、条件变量和信号量比较

- 1、条件变量存在虚假唤醒而信号量不会。条件变量和信号量虽然底层都使用了 futex 机制来实现，但它们在使用方式和语义上存在一些差异，这导致了它们在处理虚假唤醒方面的不同表现。 在 Linux 内核中，条件变量的实现涉及更多的调度和检查操作，因此容易出现虚假唤醒的情况；而信号量的实现更加简单和原子性，因此不会出现虚假唤醒的情况。
- 2、条件变量和信号量在性能方面没有绝对的优劣之分，因为它们的使用场景和适用范围不同，而且在实际应用中，性能的差异也取决于具体的实现和场景。 条件变量主要用于实现线程之间的同步和通信。当

> 线程调用条件变量的等待函数时，它会将自身放入等待队列，并释放互斥锁，然后进入睡眠状态。当其他线程修改了条件并调用通知函数时，一个或多个等待线程将被唤醒。条件变量的典型应用场景包括生产者-消费者问题、线程间的数据共享等。在某些情况下，条件变量的性能可能优于信号量，因为它们减少了不必要的阻塞和上下文切换。然而，条件变量的实现通常比信号量更复杂，因为需要处理虚假唤醒和其他同步问题。信号量是一种更通用的同步工具，用于控制对共享资源的访问。信号量的值表示可用资源的数量，当线程需要获取资源时，它会尝试减少信号量的值。如果信号量的值为零，线程将会阻塞或等待。当其他线程释放资源时，它会增加信号量的值并唤醒一个或多个等待线程。在某些情况下，信号量的性能可能优于条件变量，因为它们避免了虚假唤醒和其他同步问题。然而，信号量的实现通常比条件变量更复杂，因为需要处理资源计数和死锁等问题。

综上所述，虽然条件变量和信号量底层都使用 futex 实现，但由于它们的使用方式和语义不同，导致了在处理虚假唤醒方面的不同表现。条件变量主要用于同步和通信，而信号量主要用于资源访问控制。条件变量和信号量在性能方面没有绝对的优劣之分，选择使用哪个同步工具应该根据具体的场景和需求来决定。在某些场景下，条件变量的性能可能优于信号量，而在其他场景下，信号量的性能可能更优。