- msgs.m：一个存储在主线程栈中的本地自动变量，被两个对等线程通过 ptr 间接地引用。
- myid.0 和 myid.1：一个本地自动变量的实例，分别驻留在对等线程 0 和线程 1 的栈中。

B. 变量 ptr、cnt 和 msgs 被多于一个线程引用，因此它们是共享的。

12.7　这里的重要思想是，你不能假设当内核调度你的线程时会如何选择顺序。

步骤	线程	指令	$\%edx_1$	$\%edx_2$	cnt
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	2	H_2	—	—	0
4	2	L_2	—	0	0
5	2	U_2	—	1	0
6	2	S_2	—	1	1
7	1	U_1	1	—	1
8	1	S_1	1	—	1
9	1	T_1	1	—	1
10	2	T_2	—	1	1

变量 cnt 最终有一个不正确的值 1。

12.8　这道题简单地测试你对进度图中安全和不安全轨迹线的理解。像 A 和 C 这样的轨迹线绕开了临界区，是安全的，会产生正确的结果。

A. H_1，L_1，U_1，S_1，H_2，L_2，U_2，S_2，T_2，T_1：安全的

B. H_2，L_2，H_1，L_1，U_1，S_1，T_1，U_2，S_2，T_2：不安全的

C. H_1，H_2，L_2，U_2，S_2，L_1，U_1，S_1，T_1，T_2：安全的

12.9　A. $p=1$，$c=1$，$n>1$：是，互斥锁是需要的，因为生产者和消费者会并发地访问缓冲区。

B. $p=1$，$c=1$，$n=1$：不是，在这种情况中不需要互斥锁信号量，因为一个非空的缓冲区就等于满的缓冲区。当缓冲区包含一个项目时，生产者就被阻塞了。当缓冲区为空时，消费者就被阻塞了。所以在任意时刻，只有一个线程可以访问缓冲区，因此不用互斥锁也能保证互斥。

C. $p>1$，$c>1$，$n=1$：不是，在这种情况中，也不需要互斥锁，原因与前面一种情况相同。

12.10　假设一个特殊的信号量实现为每一个信号量使用了一个 LIFO 的线程栈。当一个线程在 P 操作中阻塞在一个信号量上，它的 ID 就被压入栈中。类似地，V 操作从栈中弹出栈顶的线程 ID，并重启这个线程。根据这个栈的实现，一个在它的临界区中的竞争的写者会简单地等待，直到在它释放这个信号量之前另一个写者阻塞在这个信号量上。在这种场景中，当两个写者来回地传递控制权时，正在等待的读者可能会永远地等待下去。

注意，虽然用 FIFO 队列而不是用 LIFO 更符合直觉，但是使用 LIFO 的栈也是对的，而且也没有违反 P 和 V 操作的语义。

12.11　这道题简单地检查你对加速比和并行效率的理解：

线程 (t)	1	2	4
核 (p)	1	2	4
运行时间(T_p)	12	8	6
加速比(S_p)	1	1.5	2
效率 (E_p)	100%	75%	50%

12.12　ctime_ts 函数不是可重入函数，因为每次调用都共享相同的由 gethostbyname 函数返回的 static 变量。然而，它是线程安全的，因为对共享变量的访问是被 P 和 V 操作保护的，因此是互斥的。

12.13　如果在第 14 行调用了 pthread_create 之后，我们立即释放块，那么将引入一个新的竞争，这次竞争发生在主线程对 free 的调用和线程例程中第 24 行的赋值语句之间。

12.14　A. 另一种方法是直接传递整数 i，而不是传递一个指向 i 的指针：

```
for (i = 0; i < N; i++)
    Pthread_create(&tid[i], NULL, thread, (void *)i);
```