Lock free and Wait Free. (Далее лф и вф — траблы с клавой) [from Wikipedia]
Являются противопоставлением алгоритмам, которые для разделения доступа к данным используют блокировку. Структура таких алгоритмов создана специально, что бы уберечь систему от разрушения, в случае доступа многих потоков к общим данным. ЛФ в данном случае подразумевает тот факт, что никакой поток не будет стоять в блокировке: каждый шаг вернет управление. То есть никакие примитивы синхронизации (мьютексы, семафоры) не могут буть использованы, так как в случае сбоя в потоке удерживающем блокировку может произойти общая остановка системы (не будет больше прогресса). Вф означает, что любой поток сможет завершить свою операцию за конечное число шагов, вне зависимости от действий других потоков. Все вф алгоритмы являются лф, но в обратную сторону — не верно. Лф — один из типов не блокирующей синхронизации.

Как правило ЛФ и ВФ имлементируются базируясь на атомарных примитивах (примитивы операции на которых не могут быть перекрыты по времени). Самый важный из данных примитивов есть CAS(compareAndSwap). Хотя существуют алгоритмы, которые обходят и то условие, например, алгоритм Деккера для синхронизации в системе двух потоков.

```
f0 := false
f1 := false
turn := 0   // or 1

p0:                                  p1:
    f0 := true                           f1 := true
    while f1 {                           while f0 {
        if turn ≠ 0 {                        if turn ≠ 1 {
            f0 := false                          f1 := false
            while turn ≠ 0 {                     while turn ≠ 1 {
            }                                    }
            f0 := true                           f1 := true
        }                                    }
    }                                    }

   // critical section                   // critical section
   ...                                   ...
   // remainder section                  // remainder section
   turn := 1                             turn := 0
   f0 := false                           f1 := false
```

One advantage of this algorithm is that it doesn't require special Test-and-set (atomic read/modify/write) instructions and is therefore highly portable between languages and machine architectures. One disadvantage is that it is limited to two processes and makes use of Busy waiting instead of process suspension. (The use of busy waiting suggests that processes should spend a minimum of time inside the critical section.)

Modern operating systems provide mutual exclusion primitives that are more general and flexible than Dekker's algorithm. However, it should be noted that in the absence of actual contention between the two processes, the entry and exit from critical section is extremely efficient when Dekker's algorithm is used.

Many modern CPUs execute their instructions in an out-of-order fashion. This algorithm won't work on SMP machines equipped with these CPUs without the use of memory barriers.

Additionally, many optimizing compilers can perform transformations that will cause this algorithm to fail regardless of the platform. In many languages, it is legal for a compiler to detect that the flag variables *f0* and *f1* are never accessed in the loop. It can then remove the writes to those variables from the loop, using a process called Loop-invariant code motion. It would also be possible for many compilers to detect that the *turn* variable is never modified by the inner loop, and perform a similar transformation, resulting in a potential infinite loop. If either of these transformations is

performed, the algorithm will fail, regardless of architecture.

TO ALLEVIATE THIS PROBLEM, VOLATILE VARIABLES SHOULD BE MARKED AS MODIFIABLE OUTSIDE THE SCOPE OF THE CURRENTLY EXECUTING CONTEXT. FOR EXAMPLE, IN JAVA, ONE WOULD ANNOTATE THESE VARIABLES AS 'VOLATILE'. NOTE HOWEVER THAT THE C/C++ "VOLATILE" ATTRIBUTE ONLY GUARANTEES THAT THE COMPILER GENERATES CODE WITH THE PROPER ORDERING; IT DOES NOT INCLUDE THE NECESSARY MEMORY BARRIERS TO GUARANTEE IN-ORDER *EXECUTION* OF THAT CODE.

Задача о консенсусе, способы её решения.

Задача о консенсусе — разрулить работу нескольких потоков. Используются специальные алгоритмы.
Herlihy [9] presented a hierarchy of nonblocking objects that also applies to atomic primitives. A primitive is at level n of the hierarchy if it can provide a nonblocking solution to a consensus problem for up to n processors. Primitives at higher levels of the hierarchy can provide nonblocking implementations of those at lower levels, but not conversely. Compare-and-swap and the pair load-linked and store-conditional are universal primitives as they are at level _ of the hierarchy. Widely supported primitives such as test-and-set, fetch-and-add, and fetch-and-store are at level 2.

Compare-and-swap, introduced on the IBM System 370, takes as arguments the address of a shared memory location, an expected value, and a new value. If the shared location currently holds the expected value, it is assigned the new value atomically. A Boolean return value indicates whether the replacement occurred. Compare-and-swap is supported on the Intel Pentium Pro and Sparc V9 architectures.

Load-linked and store-conditional, proposed by Jensen et al. [15], must be used together to read, modify, and write a shared location. Load-linked returns the value stored at the shared location. Store-conditional checks if any other processor has since written to that location. If not then the location is updated and the operation returns success, otherwise it returns failure. Load-linked _store-conditional is supported by the MIPS II, PowerPC, and Alpha architectures.

A wait-free implementation of a concurrent data object is one that guarantees that any process can complete any operation in a finite number of steps, regardless of the execution speeds on the other processes. The wait-free condition provides fault-tolerance: no process can be prevented from completing an operation by undetected halting failures of other processes, or by arbitrary variations in their speed. The fundamental problem of wait-free synchronization can be phrased as follows:
Given two concurrent objects X and Y, does there exist a wait-free implementation of X by Y?
It is clear how to show that a wait-free implementation exists: one displays it. Most of the current literature takes this approach. Examples include “atomic” registers from nonatomic “safe” registers [18], complex atomic registers from simpler atomic registers [4, 5, 16, 23, 25, 26, 29, 31], read-modify-write operations from combining networks [11, 15], and typed objects such as queues or sets from simpler objects [14, 19, 20].
It is less clear how to show that such an implementation does not exkt. In the first part of this paper, we propose a simple new technique for proving statements of the form “there is no wait-free implementation of X by Y.” We derive a hierarchy of objects such that no object at one level can implement any object at higher levels (see Figure 1). The basic idea is the following each object has an associated consensus number, which is the maximum number of processes for which the object can solve a simple consensus problem. In a system of n or more concurrent processes, we show that it is impossible to construct a wait-free implementation of an object with consensus number n from an object with a lower consensus number.
These impossibility results do not by any means imply that wait-free synchronization is impossible or infeasible. In the second part of this paper, we show that there exist universal objects from which one can construct a wait-free implementation of any object. We give a simple test for universality, showing that an object is universal in a system of n processes if and only if it has a consensus number greater than or equal to n. In Figure 1, each object at level n is universal for a system of n processes. A machine architecture or programming language is computationally powerful enough to support arbitrary wait-free synchronization if and only if it provides a universal object as a primitive.
Most recent work on wait-free synchronization has focused on the construction of atomic read/write registers [4, 5, 16, 18, 23, 25, 26, 29, 31]. Our results address a basic question: what are these registers good for? Can they be used to construct wait-free implementations of more complex data structures? We show that atomic registers have few, if any, interesting applications in this area. From a set of atomic registers, we show that it is impossible to construct a wait-free implementation of (1) common data types such as sets, queues, stacks, priority queues, or lists, (2) most if not all the classical synchronization primitives, such as test&set,

Число	Объект
1	read/write registers
2	test&set, swap, fetch&add, queue, stack
...	...
2n-2	N-register ssignment
...	...
Infinity	augmented ,memory move and swap-to-memory ,queuecompare&swap, fetch&cons, sticky byte

COMPARE&SWAP, AND FETCHD-ADD, AND (3) SUCH SIMPLE MEMORY-TO-MEMORY OPERATIONS AS MOVE OR MEMORY-TO-MEMORY SWAP. THESE RESULTS SUGGEST THAT FURTHER PROGRESS IN UNDERSTANDING WAIT-FREE SYNCHRONIZATION REQUIRES TURNING OUR ATTENTION FROM THE CONVENTIONAL READ AND WRITE OPERATIONS TO MORE FUNDAMENTAL PRIMITIVES.

УНИВЕРСАЛЬНЫЕ КОНСТРУКЦИИ: COMPAREANDSWAP(CAS), LOADLINKED-STORECONDITIONAL (LLSC)

ПРИМЕР ИСПОЛЬЗОВАНИЯ — LLSC ДЛЯ QUEUE.

БАЗОВЫЙ ФОРМАЛИЗМ ЛАМПОРТА

ОПИСЫВАЕТ ДВА ПОНЯТИЯ В СИСТЕМЕ — ПРОИЗОШЛО ДО, И МОГЛО ПОВЛИЯТЬ НА.
В БОЛЬШИХ РАСПРЕДЕЛЕННЫХ СИСТЕМАХ, КАК УТВЕЖДАЛ ЛАМПОРТ МОЖЕТ СУЩЕСТВОВАТ Ь ЛИШЬ ЧАСТИЧНАЯ СИСТЕМА HAPPEND BERFORE:

1.