The Sentinel of Eternity

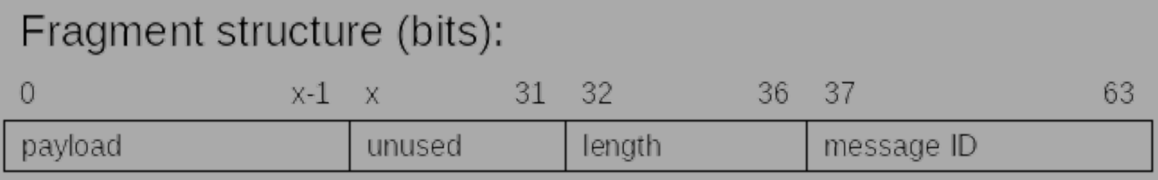
Submission deadline:	**2020-04-19 23:59:59**
Evaluation:	**30.0000**
Max. assessment:	**30.0000** (Without bonus points)
Submissions:	31 / 75
Advices:	0 / 0

The task is to develop a class which quickly receives, analyzes, and solves problems sent by an alien probe.

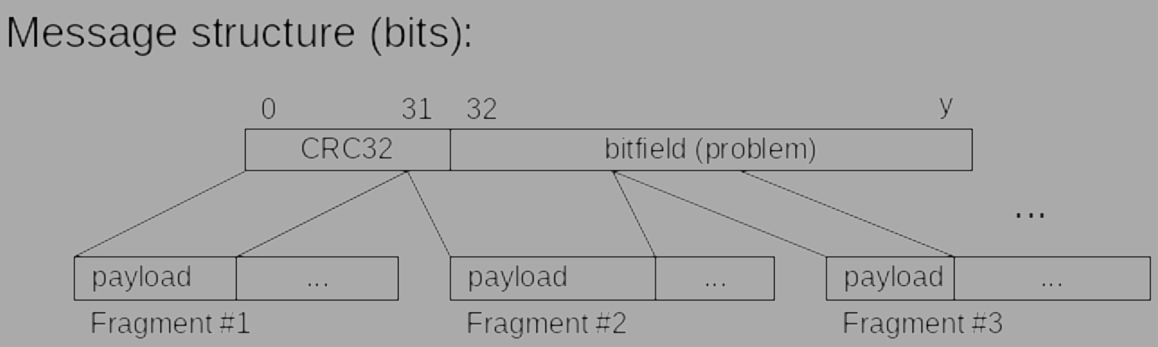
Sci-fi author A. C. Clarke wrote a story "The Sentinel of Eternity". An alien probe was found in the story. The probe was shielded by an unknown force field, preventing people from accessing the probe. Clearly, people were trying to disable the force field and access the probe. We assume a similar situation in this homework. An alien probe allows access only to a developed civilization. The civilization must be able to receive, analyze and solve problems transmitted from the probe. The software you have to develop must interface with the probe, accept the messages, and solve the problems the probe sends. A developed civilization is able to process and solve the problems quickly, therefore, your software has to use threads to speed up the processing.

The problem to solve is rather technical and requires a bit more programming. You may either develop everything by yourself, or you may use some classes and function that solve some steps in the entire problem. In either case, you will have to add the code that creates threads and synchronizes them:

- The alien probe sends problem instances to solve. One problem instance is described in one message. Each message has an unique identifier - a 27 bit long integer. A message is split into fragments before it is transmitted from the alien probe. Each fragment is 64 bit long; it contains message identifier (message id) and some bits of the message (payload, length). A fragment may deliver `x` bits of the message (e.g., `x = 17`). Then the field `length` is set to value `x-1` (i.e. 16 in the example). Bits 0, 1, 2, ... x-1 deliver a portion of the message, bits x, x+1, x+2, ..., 31 are not used (and may contain just anything). The structure of a fragment is depicted below:



- The probe may send several messages simultaneously. Moreover, the fragments are sent in a random order. Therefore, the receiver must use message identifier from the fragment to group the fragments that form the same message. Your implementation must take care of this, i.e., receive the fragments and group them by the message id.
- A message must be assembled from the grouped fragments. There are two problems to solve: the order of fragments is not known, moreover, there is not any explicit information on the number of fragments required to form the entire message. It had been discovered that the message starts with a CRC32 checksum. Therefore, the processing may try to assemble the fragments, compute the checksum and compare it with the CRC32 from the message. If there is not match, the order of fragments is incorrect, or there are some fragment(s) missing and the implementation must wait for the remaining fragments. The CRC32 checksum computation is implemented in the testing environment (and also in the attached library) - the function is named `CalculateCRC32`. Moreover, the testing environment (and the library) provides function `FindPermutations`. The function has an input parameter - the list of fragments. It tries all permutations of the input fragments and computes CRC32 checksum for each of them. If there is a match, the function reports a success - it calls the callback from the third parameter and passes it the assembled message. Theoretically, there may be more than one permutation of the input fragments that leads to a valid message (the situation is possible, but unlikely). Nevertheless, the callback interface of the `FindPermutations` function reports all such messages. Further details on `CalculateCRC32` and `FindPermutations` are described below.



- The message extracted from fragments contains the CRC32 and a bitfield. Once the message was assembled, the CRC32 information is useless and ma be ignored. The bitfield, on the other hand, defines the instance of the problem to solve. Each bit of the bitfield is seen as a Boolean value: true (bits set to 1) and false (bits set to 0). The bits are processes from LSB (least significant bit) to MSB (most significant bit). The number of bits in the bitfield is given by the length of the message (minus the length of CRC32). Operators AND, OR, and XOR may be placed in between the Boolean values such that the resulting expression is valid and results to true. There may exist many different ways to place the operators, the solution of the problem is the number of different ways how it may be done. To avoid get rid of possible associativity and priority problems, we may think of the problem in terms of the prefix notation. We are to form valid prefix expressions with operators AND, OR and XOR, the operands are given by the problem instance (the bitfield) and the expressions must evaluate to true. For example bitfield of length 3 bits formed by a single byte 0x06:

```
0x6 = 0b00000110, i.e. Boolean operands are (LSB to MSB): false true true
All valid prefix expressions:

  AND false AND true true
  AND false OR  true true
  AND false XOR true true

* OR  false AND true true
* OR  false OR  true true
  OR  false XOR true true

* XOR false AND true true
* XOR false OR  true true
  XOR false XOR true true

  AND AND false true true
* AND OR  false true true
* AND XOR false true true

* OR  AND false true true
* OR  OR  false true true
* OR  XOR false true true

* XOR AND false true true
  XOR OR  false true true
  XOR XOR false true true
Result: 10 (the lines with the asterisk)
```

The number of results grows very quickly with the increasing number of operands. Therefore, the result is stored into a `CBigInt` type, the object represents a 512 bit unsigned integer. The tasting environment (and the attached library) provides function `CountExpressions`. The function finds the result for the given problem instance (the bitfield). Further details on `CBigInt` and `CountExpressions` are described below.

- As stated previously, the fragments may be assembled into a valid message in more than one way (and it was the reason function `FindPermutations` reported its result by means of a callback). If there is more than one way to assemble the fragments, then the computation in the previous paragraph (the counts) must be applied to all bitfields returned from `FindPermutations`. The final result of the problem instance is the maximum.

The testing environment (and the attached library) provides the following:

- Class `CBigInt`. The class implements big unsigned integers using 512 bits. The implementation is not optimized for speed and the interface provides only few basic operations (addition, multiplication, and comparison). However, the interface is enough for the purposes of this homework. The exact interface and a doxygen documentation is available in header file `common.h`. We expect that you use the class in your implementation; there is no need to re-implement the class and we discourage you from doing so. The class is available in the progtest environment and works correctly in all tests, including the bonus tests.
- Function `CalculateCRC32` computes CRC32 checksum for the given message. There is an important advantage of the function: the function computes the result for an arbitrary number of bits(CRC32 implementations available on the internet are usually limited to byte granularity). Again, we expect you to use the existing function. The function is available in the progtest environment and works correctly in all tests, including the bonus tests.
- Function `FindPermutations` takes the fragments from its parameters, tries all their permutations, assembles messages and checks the CRC32. If the CRC32 matches, then the assembled message is passed to the callback (the third parameter). The callback is given the entire message including the CRC32, thus you will probably need to skip the first 32 bits. Function `FindPermutations` is available in the progtest testing environment, however, it is available only in the mandatory and optional tests. The function purposedly does not work in the bonus tests. If you plan to pass the bonus tests, you will have to implement the function yourself (the function design is not suitable for the bonus tests anyway).
- Function `CountExpressions` calculates the number of ways to place Boolean operators into the given Boolean values such that the result is true. The parameter is the bitfield representing the Boolean operands. Function `CountExpressions` is available in the progtest testing environment, however, it is available only in the mandatory and optional tests. The function purposedly does not work in the bonus tests (the results are invalid). If you plan to pass the bonus tests, you will have to implement the function yourself (the function design is not suitable for the bonus tests anyway).
- Class `CReceiver` is designed to receive the fragments from the alien probe. The class is a pure abstract class - it only defines the interface. The testing environment creates instances of `CReceiver` (its subclasses) and registers them in your implementation (AddReceiver, see below). Your implementation shall designate a thread that invokes method `CReceiver::Recv` in a loop. The method either results `true` (then there is another fragment received, the fragment is available via the output parameter), or `false`. If the result is false, then the receiver has no further fragments, the servicing thread must break the loop and terminate.
- Class `CTransmitter` is designed to pass the solved problems back to the alien probe. The class is a pure abstract class - it only defines the interface. The testing environment creates instances of `CTransmitter` (its subclasses) and registers them in your implementation (AddTransmitter, see below). The interface of `CTransmitter` consists of two methods:
  - `Send (id, cnt)`. The method is to be called when a problem instance was successfully solved. The parameters are `id` (message identifier) and `cnt` (the number of expressions found). Each successfully solved problem must be reported to the probe exactly once. However, there is no restriction on the transmitter instance, you may choose any transmitter available to report the solution.
  - `Incomplete (id)`. Call this method when you are about to terminate the computation and there are fragments that cannot be assembled into valid messages. This happens when all receivers have reported that no further fragments are available and the testing

environment called method `CSentinelHacker::Stop` (see below). When his happen, finish the computation for the assembled messages and report the results in the standard way (method `Send`). However, the messages that cannot be completed must be reported back using this method.

The instances of `CTransmitter` are neither reentrant nor thread-safe. One transmitter instance may send at most one result at a time. If there is more results to send / return as incomplete, your implementation must use all available transmitters or (if all transmitter instances are busy) wait until the previous result is sent.

Your solution needs to implement `CSentinelHacker` class, the implemented class must include the required interface below. Next, you may add further support classes and functions to your solution. `CSolutionHacker` class uses certain support classes that are implemented in the testing environment. A simplified implementation of these classes is provided in the attached archive. These are classes `CReceiver` and `CTransmitter`. The implementation of the last two classes is fixed, you cannot modify them in any way.

The program will use several threads:

- the main thread creates an instance of `CSentinelHacker`, registers receivers and transmitters, and starts the computation by invoking `CSentinelHacker::Start`. The `Start` method creates your threads, in particular:
- there will be a thread for each receiver. The receiver thread runs in a loop and invokes the corresponding `CReceiver::Recv` method. The receiver threads are not designed to do any time-intensive computation. Instead, the threads take the fragment received and pass it to the worker threads to process it. Your implementation is responsible for the receiver threads; it shall start them and wait for their completion,
- worker threads. The worker threads are designated to do all the expensive computation. Therefore, the number of worker threads may be controlled by a parameter passed to the `Start` method. Your implementation is responsible for these threads, i.e., it shall start and terminate them,
- transmitter threads. Each transmitter will have one dedicated thread that passes the result to the `Send` / `Incomplete` methods. The transmission of a result may take some time. Therefore, the transmitter interface must not be called directly from within worker threads (it would block them for a long time). Transmitter threads are similar to the receiver threads, they are not designed to do any time-intensive computation. Your implementation is responsible for the transmitter threads; it shall start them and wait for their completion.

Class `CSentinelHacker` encapsulates the computation and controls the execution of the threads. You have to implement the class. The required interface consists of methods:

`SeqSolve(fragments, cnt)`
: the method exists to simplify the testing of the sequential solver. Given a list of fragments, the methods sequentially solves the problem. Return value is either `false` if the input fragments do not form a valid message, or `true` if the fragments were assembled and the solution was computed. Output parameter `cnt` is set to the number of existing expressions in this case. Your implementation must provide this method. If it does not work correctly, the testing stops immediately. If you decide to use the built-in functions (`FindPermutations` and `CountExpressions`), this method is just a simple wrapper to pass the parameters.

`AddReceiver (recv)`
: the method registers a new receiver. The method itself does not do anything else (in particular, it does not start any thread). The receiver thread will be created after the computation actually starts (method `Start`).

`AddTransmitter (trans)`
: the method registers a new transmitter. The behavior is similar to `AddReceiver`, the method does not start the transmitter thread, the thread will be created after the computation actually starts (method `Start`).

`AddFragment (fragment)`
: the method is called asynchronously from thread(s) in the testing environment. The method must be thread-safe, it may be called from many threads simultaneously. The asynchronous input is another way the alien probe delivers message fragments to your implementation. You have to process all fragments, i.e., the fragments sent synchronously from receivers as well as the fragments delivered asynchronously. Method `AddFragment` is called only between the invocations of `CSentinelHacker::Start` and `CSentinelHacker::Stop` (i.e., is not called before Start and is not called after Stop).

`Start(thrCnt)`
: the method starts all required thread (receivers, transmitters and workers) and starts to process the incoming message fragments. The parameter is an integer `thrCnt`, it defines the number of worker threads to start. Once the threads are started, the method returns to the caller. It does not wait for the completion of the threads.

`Stop()`
: the method is called from the main thread to stop the computation. However, the computations are not terminated directly. Instead, the method waits until all fragments are read from the receivers (it is guaranteed that the asynchronous AddFragment is not be executed once the testing environment calls Stop). Next, it finished all computations (assembles all completed messages, solves the counts and passes the results to the transmitters). Incomplete messages must be reported to transmitters (see `CTransmitter::Incomplete`). Finally, the method waits until all your threads are finished and joined. Then it returns to the caller. Caution - do not invoke functions like `exit`, `pthread_exit` or similar. If `Stop` does not return to the caller, your program will be considered erroneous.

further
: you are free to add any other auxiliary methods and member variables to the `CSentinelHacker` class.

If you decide to implement your own solvers (replacements of `FindPermutations` and `CountExpressions`), consider the following:

- the algorithm that assembles fragments into the messages must try all permutations. Thus, it is definitely slow with $T(n) \in O(n!)$, where `n` is the number of fragments. The testing environment inputs reasonable number of fragments, thus the factorial complexity is acceptable,
- the algorithm that counts the expressions must be reasonably efficient. The expected time complexity is $T(n) \in O(n^3)$, where `n` is the length of the bitfield (in bits),
- both algorithms must be implemented with a reasonable overhead, both are time-critical to the overall run-time,
- the testing starts with a time calibration of your sequential solution. Based on the measured speed, the testing environment modifies the size of problems your solution receives. If the speed of your implementation is similar to the speed of the reference, the testing works fine. Therefore, check the speed of your implementation and compare it to the speed of the implementation in the attached library and make sure the speeds do not differ too much. The testing inputs problems where the number of fragments does not exceed 12 and the length of the bitfiels does not exceed 200 bits,
- your solution is included in a separate namespace in the testing environment. Thus the names of your functions do not conflict with the names in the testing environment. On the other hand, if you choose identifiers `FindPermutations` or `CountExpressions` for your implementation, you may get conflicting identifiers when compiling your code with the attached library. Therefore, it is recommended to use different names when implementing your own version of `FindPermutations` / `CountExpressions`.

Submit your source code containing the implementation of class `CSentinelHacker` with the required interface. You can add additional classes and functions, of course. Do not include function `main` nor "include" directives to your implementation. The function `main` and "include" directives can be included only if they are part of the conditional compile directive block (#ifdef / #ifndef / #endif).

Use the example implementation file included in the attached archive. Your whole implementation needs to be part of source file `solution.cpp`, the delivered file is only a stub. If you preserve compiler directives, you can submit file `solution.cpp` as a task solution.

You can use pthread or C++11 thread API for your implementation (see #include files). The Progtest uses g++ compiler version 8.3, this version handles most of the C++11, C++14, and C++17 constructs correctly.

Hints:

- Start with the threads and synchronization, use the function from the attached library to solve the algorithmic problems. Once your program works with `FindPermutations` and `CountExpressions`, you may replace these functions with your own implementation.
- To be able to use more CPU cores, process as many messages as possible, all in parallel. The message fragments are delivered from the registered instances of `CReceiver` and from the asynchronous method `AddFragment`. You must process all fragment from all these sources. You do not have any explicit information on the number of fragments that form a message. Therefore, you have to try to assemble the message with each new fragment received. The assembly of messages and the computation of expressions (e.g. in some other message) may run in parallel. Design your program this way.
- The instances of `CSentinelHacker` are created repeatedly for various inputs. Don't rely on global variable initialization. The global variable will have different values in the second, third, and further tests. An alternative is to initialize global variables always in constructor or `Start` method. Not to use global variables is even better.
- Don't use mutexes and conditional variables initialized by `PTHREAD_MUTEX_INITIALIZER`. There are the same reasons as in the paragraph above. Use `pthread_mutex_init()` instead. Or use C++11 API.
- The instances of receivers and transmitters are allocated by the testing environment when smart pointers are initialized. They are deallocated automatically when all references are destroyed. Don't free those instances; it is sufficient to forget all copies of the smart pointers. But, on the other hand, your program has to free all resources it allocates.
- The fragments must be received, processed, the problems must be solved, and counts must be transmitted simultaneously. Do not attempt to read all fragments into some data structure and then start the processing itself. The technique won't work - you will end up in a deadlock in the second test (two or more receiver threads). Moreover, you should simultaneously receive fragments requests from all receivers (and from the asynchronous AddFragment). If you try to receive fragments from fragment A, then B, etc., you will cause a deadlock too.
- Method `CTransmitter::Send` is not reentrant/thread safe. Your implementation must take care of the serialization of the calls. The transmission of a solution takes some time. You have to use all available transmitters to send all results in time.
- Don't use `exit`, `pthread_exit` or similar calls in `Stop` or in any other method. If `Stop` method does not return back to its caller, your program will be evaluated as wrong.
- Use sample data in the attached files. There you can find an example of API calls, several test data sets, and the corresponding results.
- The test environment uses STL. Be careful as the same STL container must not be accessed from multiple threads concurrently. You can find more information about STL parallel access in **C++ reference - thread safety.**
- Test environment has a limited amount of memory. There is no explicit limit, but the virtual machine, where tests are run has RAM size limited to 4GB. Your program is guaranteed at least 500MB of memory (i.e., data segment + stack + heap). The rest of the physical RAM is used by OS, and other processes.
- If you choose to solve all bonus tasks, be careful to use proper granularity of parallelism. The input problem must be divided into several subproblems to pass the bonus tests. On the other hand, if there are too many small problems, context switches induce a high overhead. The reference solution limits the maximum number of problems concurrently solved by the worker threads to avoid this overhead.
- Do not forget endianity if implementing your own `FindPermutations`. The CRC32 in the message is stored in the network byte order (big endian), function `CalculateCRC32` return CRC32 as a 32 bit integer, using platform byte order (x86-64, thus little-endian). Do the required conversions before the comparison.
- The time intensive computation (`FindPermutations`, `CountExpressions`) must be done in the worker threads. The number of worker threads is determined by the parameter of method `Start`. The testing environment rejects a solution that does time-intensive computation outside these threads (e.g. in the receiver/transmitter threads).

What do the particular tests mean:

Test algoritmu (sekvencni) [Algorithm test]
: The test environment calls methods `SeqSolve()` for various inputs and checks the computed results. The purpose of the test is to check your algorithm. No instance of `CSentinelHacker` is created, no `Start` method is called. You can check whether your implementation is fast enough with this test. The test data are randomly generated.

Základní test [Basic test]
: The test environment creates an instance of `CSentinelHacker` for different number of worker threads (W=xxx), receivers (R=xxx), and transmitters (T=xxx).

Test zahlcení [Flood test]
: The test environment generates a vast amount of fragments and it checks whether your implementation can handle it. If you do not control the number of started requests, you will probably exhaust memory limit.

Test zrychleni vypoctu [Speedup test]
: The test environment runs your implementation for the same input data with a various number of worker threads. The test measures the time required for the computation (wall and CPU times). As the number of worker threads increases, the wall time should decrease, and CPU time can slightly increase (the number of worker threads is below the number of physical CPU cores). If the wall time does not decrease or does not decrease enough, the test is failed. For example, the wall time shall drop to 0.5 of the sequential time if two worker threads are used. In reality, the speedup will not be 2. Therefore, there is some tolerance in the comparison.

Busy waiting (pomale vysilace) [Busy waiting - slow receivers]
: There is a sleep call inserted into the `CReceiver::Recv` calls (e.g. 100 ms sleep). Worker threads then do not have anything to do. If worker threads are not synchronized/blocked properly, CPU time increases and the test fails.

Busy waiting (pomale vysilace) [Busy waiting - slow transmitters]
: There is a sleep call inserted into the `CTransmitter::Send` calls (e.g. 100 ms sleep). If the worker threads are not synchronized/blocked properly, CPU time is increased, and the test fails.

Busy waiting (pomale vysilani i prijimani) [Busy waiting - slow receivers and transmitters]
: There are delays inserted in both `CReceiver::Recv` and `CTransmitter::Send` methods. If the worker threads are not synchronized/blocked properly, CPU time is increased, and the test fails.

Test rozlozeni zateze 1 [Load balance test 1]
: The test environment tries, whether the computation of a single problem engages more than one thread. There is just one message to process, the message consists of many fragments. The testing environment checks that the computation time decreases when the number of worker threads increases. This test is a bonus test. Functions `FindPermutations` and `CountExpressions` return invalid results in this test. You have to implement the functions yourself to pass the test.