Checkpoint 1 Writeup

My name: 卢郡然

My SUNet ID: 502024330034

This lab took me about 4 hours to do.

Implementation of Reassembler

Program Structure and Design

This task is easily solved by modeling it to a sliding window. Noting a `head_index` and see if the current buffer reaches this `head_index` is an easy way to implement a reassembler.

But when it comes to counting `bytes_pending`, things get hard. The core problem is how to handle overlapping properly and efficiently, which is effort-paying.

I finally decide to convert the question into a interval union problem.

The core idea is to maintaining non-overlapping intervals along the index axis. This is be implemented by leveraging `std:set`'s ability to maintain ordered array and find elements, erase elements in $O(\log |S|)$.

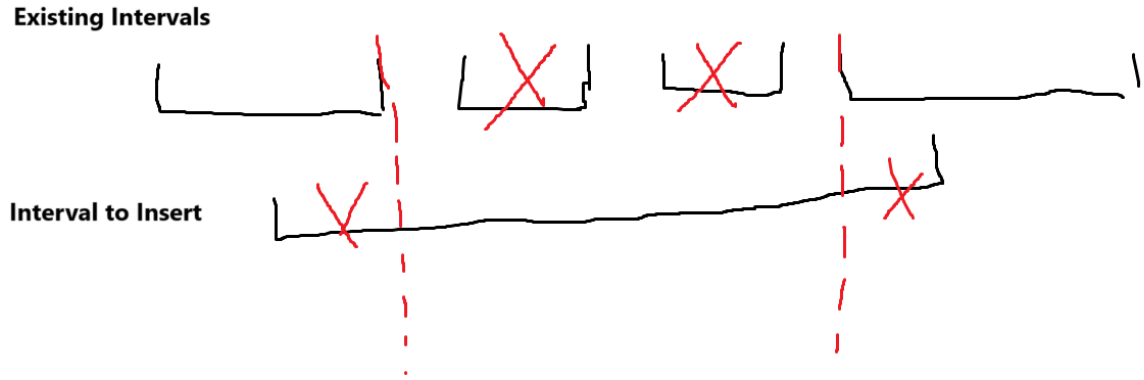
Specifically, the elements are

```
struct Node{
  uint64_t index;
  string data;

  Node(uint64_t i, string d): index(i), data(d) {}

  bool operator < (const Node& other) const{
    return index < other.index || (index == other.index && data.length() >
other.data.length());
  }
};
```

which is ordered by `index`.

When a new interval x is trying to insert into the set. It is required to maintain the non-overlapping state. The detailed process is shown below:

1. Check intervals which have small left end to x. Shrink the left bound of x if overlapping happens.
2. Check intervals which have equal or greater left end to x. Delete existing intervals that fully included by x. Shrink right bound of x if the right-most interval has overlaps.



The `btytes_pending` is counted during the process.

All operations above is supported by `std::set`, which contains:1. `set::lower_bound()` to find the corresponding intervals. `set::erase()` to delete intervals.

The find and erase operation has a logarithmic complexity, and the iteration to erase fully included intervals happens to each interval once at most. So the overall time complexity is $O(\log |S|)$.

Challenges and Difficulties

The definition of `is_last_substring` is ambitious, which is not mentioned in the document or startup codes.

It really costs me a large mount of time to check the test case to find out the true meaning of the "last string".

The method to handle "last string " is to find the last index. The index of the last character is fixed if the inputs are reasonable, which can be calculated by inputs labeling `is_last_substring = True` :

```
if(is_last_substring){
    last_index = first_index + data.length();
  }
```

When pushing index over this `last_index` the true ending is set and the output is closed.

```
if(head_index >= last_index){
    writer.close();
}
```

Experimental Results and Performance.

```
[main] Building folder: /home/max/minnow/build check1
[build] Starting build
[proc] Executing command: /home/max/miniconda3/bin/cmake --build /home/max/minnow/build --config Debug --target check1 --
[build] [1/1 100% :: 3.432] cd /home/max/minnow/build && /home/max/miniconda3/lib/python3.10/site-packages/cmake/data/bin/
[build] Test project /home/max/minnow/build
[build]       Start  1: compile with bug-checkers
[build]  1/17 Test  #1: compile with bug-checkers ........   Passed    2.17 sec
[build]       Start  3: byte_stream_basics
[build]  2/17 Test  #3: byte_stream_basics ...............   Passed    0.01 sec
[build]       Start  4: byte_stream_capacity
[build]  3/17 Test  #4: byte_stream_capacity .............   Passed    0.01 sec
[build]       Start  5: byte_stream_one_write
[build]  4/17 Test  #5: byte_stream_one_write ............   Passed    0.01 sec
[build]       Start  6: byte_stream_two_writes
[build]  5/17 Test  #6: byte_stream_two_writes ...........   Passed    0.01 sec
[build]       Start  7: byte_stream_many_writes
[build]  6/17 Test  #7: byte_stream_many_writes ..........   Passed    0.06 sec
[build]       Start  8: byte_stream_stress_test
[build]  7/17 Test  #8: byte_stream_stress_test ..........   Passed    0.05 sec
[build]       Start  9: reassembler_single
[build]  8/17 Test  #9: reassembler_single ...............   Passed    0.01 sec
[build]       Start 10: reassembler_cap
[build]  9/17 Test #10: reassembler_cap ..................   Passed    0.01 sec
[build]       Start 11: reassembler_seq
[build] 10/17 Test #11: reassembler_seq ..................   Passed    0.01 sec
[build]       Start 12: reassembler_dup
[build] 11/17 Test #12: reassembler_dup ..................   Passed    0.05 sec
[build]       Start 13: reassembler_holes
[build] 12/17 Test #13: reassembler_holes ................   Passed    0.01 sec
[build]       Start 14: reassembler_overlapping
[build] 13/17 Test #14: reassembler_overlapping ..........   Passed    0.01 sec
[build]       Start 15: reassembler_win
[build] 14/17 Test #15: reassembler_win ..................   Passed    0.20 sec
[build]       Start 37: compile with optimization
[build] 15/17 Test #37: compile with optimization ........   Passed    0.67 sec
[build]       Start 38: byte_stream_speed_test
[build] 16/17 Test #38: byte_stream_speed_test ...........   Passed    0.04 sec
[build]       Start 39: reassembler_speed_test
[build] 17/17 Test #39: reassembler_speed_test ...........   Passed    0.09 sec
[build]
[build] 100% tests passed, 0 tests failed out of 17
[build]
[build] Total Test time (real) =   3.40 sec
[driver] Build completed: 00:00:03.549
[build] Build finished with exit code 0
```

I think the performance may be reduced due to the frequent use of `string:substr()` which may lead to high copy cost.