Implementation

I used classes for the Logger and buffer.

1. `Logger` Class:
    - Manages a log of messages, which is a vector of strings. One can read and write (=append) to the log.
    - Read-only functions such as read(size_t), length() use locks: They may not have modifying operations, but this does not exclude the possibility of racing conditions. We don't know the implementation of the std::vector backing this log, so we cannot be certain that it is in a valid state during a write operation. If we interleave a read() with a write then, this may give unexpected results. As an example, if we don't make the `read(size_t)` lock, consider the following trace:

```
{Suppose vector<int> is implemented as a pointer int* elems to an array on
the heap a size_t sz representing the size }

Thread 2: calls add(), add() locks buf.mutex and calls buf.push_back(),
which performs only one operation: sz++; sz is now 4.
Thread 1: calls read with line_nr = 3, does not block because read() uses
no lock checks whether line_nr < size() == sz, so it continues
Thread 1: reads from unassigned memory *(elems + 3)
Thread 2: writes msg to *(elems + 3).
```

Thread 1 would have undefined behaviour.

And there are way, way more ways for this to fail. For example, if the vector is added to beyond its current capacity, it will reallocate `int* elems`, potentially needing to move elements from one location in the heap to another. This gives plenty of time for read(line_nr) operations to read from all sorts of invalidated locations in memory. Conclusion: reading and writing must occur exclusively, and we need locks to demarcate critical sections in reading functions as well.

- The write() function is thread-safe by using a mutex: this mutex is released before returning from the function. So no hold-and-wait is possible when calling this function.
- I provided some extra functionality: one can not only read the oldest line, but also at an arbitrary line_nr using `read(line_nr)`. However, to ensure line_nr < log.size(), one first needs to check the length of the logger, and for that I introduced Logger::length(). There is no risk of a check-then-act error in the code

  ```
  if (line < logger.length()) cout << logger.read(line);
  ```

  because the logger cannot shrink in between the check and the act (lines once written are never removed from the log).

2. `Buffer` Class:

- Represents a buffer for storing integers. I use a `std::deque` for reasons mentioned in `Discord, OSC#assignments, Oct 5 + Oct 6`, and instead of a `std::queue` I use a deque because only deque provides the member function `resize()` which is needed to truncate the buffer in case the bound is set to some count lower than `size()`.
- It allows setting a bound on the maximum number of elements in the buffer using `set_bound()`. This function has 1 or 2 (in case of truncation) modifying operations that need to happen atomically. Hence a lock is acquired upon entering and released only after this critical section. A thread does not hold any locks after calling (so no hold-and-wait possible).
- Provides a function `add(int)` to add integers to the buffer, `int remove(void)` to remove integers from the buffer. Both are modifying operations and their critical sections are covered by mutex calls. I made both throw an exception: this happens when adding to a full buffer (or when deque throws an exception due to its size) or when removing from an empty one. Why exceptions? Well, remove does not allow for a boolean indicator return value, since it already returns an `int`, and it seemed strange to me to let remove return a boolean indicator of success/failure but to let `remove` throw in case of a failure.
- last, a function `set_unbounded()` to set the buffer to unbounded state. Again use a mutex to ensure mutual exclusion of modifications to the size_t bound.
- Could I have used separate mutexes for `bound` and `buf`? Maybe. `add()` and `set_bound()` have check-then-act situations on `bound` and `buf` simultaneously. Overall, I thought it would be a bit overengineered to do this. So I use one mutex to represent the entire class as one resource. This also makes it much easier to reason about deadlocks and other synchronization details.
- One extra detail: `std::deque` may throw all sorts of exceptions if one tries to `push_back()` beyond `max_size()`. This is something that could happen if the buffer is set unbounded. I thought it would be good to let this not terminate the program, but rather to catch and log this. So that is what I did in `add()`: it features a try-catch block doing exactly this. Notice how I have to keep track of acquired locks in every branch of the function.

I am not very happy with the specification's way full or empty buffers should be handled. I would have used condition variables to let threads release and wait until elements are added/removed, rather than to log failure. Logging failure can lead to livelocks, for example if we let a thread repeat trying to add/remove an element if it failed (this is what some of my test tasks do).

I would have used condition variables instead, together with `std::unique_lock<std::mutex> lock(std::mutex&)` to adhere to modern c++'s RAII principles and not have the issue of having to release locks in every branch of an if-statement or every try-catch block, which is already quite a tedious task for such a simple implementation as this (but who am I?). In fact, I made another, more satisfying implementation that I will include in my submission (`buffer_modern.cpp`).

Further, you may ask why I chose to add a mutex to the Logger, since access by the buffer is already guaranteed to happen in exlusion due to `buf_mutex`'s acquirement everywhere a `Logger::write()` is called. The reason is that there is still a possibility for a user of my class to obtain a reference to Buffer's Logger via `Logger& Buffer::logger()`. Why this function, then? Well, we may want to read out the log

after program execution, so I found it necessary to include it (otherwise, what is the point of keeping a log if you encapsulate it and not give a handle to the user?)

3. Test Functions:
    - test_work_1, test_work_2, and test_work_3 are thread functions that simulate different scenarios of using the buffer and logger.
    - These functions demonstrate multi-threaded operations on the buffer and logger. They are further discussed under **2. Tests**.
4. Main Function:

- Executes multiple tests:
- threads te and to are allowed to execute concurrently, then joined. These add even and odd number in the range 0-500, smaller to greater, to the buffer. Then, the buffer is printed. We can check whether indeed all even number and odd numbers are inserted, where the even numbers are in order among themselves and the odd numbers are in order among themselves.
- (t1, t2, t3) that execute the test functions that add and remove elements in parallel and t3 that sets the bound size of the buffer multiple times. Our expected behaviour of this is not very specifically specifiable, but we at least expect this not to terminate due to exceptions.
- After joining t1,t2,t3, main creates a thread to test test_length, which is a stress test trying to let the queue throw a length_error, which should be handled appropriately with as expected behaviour th message to stdout shown in the comment line.
- After all threads finish, it obtains a reference to the (global) buffer `b`'s `Logger` and reads it to stdout.

Note that I added an option `DEBUG` (at the top of the file), so that if the program terminated prematurely, we can rerun it and see the lines written to the log printed during program execution (featured option in `Logger::write()`). This has been very helpful while debugging. It made me among others trace back the cause of a `std::length_error` thrown by `deque`.

2. Tests

I made some test functions in `main.cpp`, which at least cover all the implemented functions for `Buffer` (i.e. `set_bound`, `set_unbounded`, `remove`, `add`). I wanted to see some failures of the remove/add operations. In order to do this, I made:

Behaviour tests

- Small tests to ensure correct behaviour of the buffer.
- `test_even` sets the buffer bound to 200, then writes all even numbers between 0 and 500 to the buffer.
- `test_odd` writes all odd numbers between 0 and 500 to the buffer. We interleave these to see if the buffer indeed contains 20 numbers (i.e. stops at bound) and whether, if k is in buffer, then also k - 2 is (i.e. no adds got skipped for some strange reason). We check this by printing the buffer's state. It seems to work. 200 is also big enough to see interleaving in some of the program runs (20 was not, so I increased the number).

Stress tests

- `test_work_1` only add 20 initial elements to the buffer, which is not much. t1 and t2 will together perform many remove+add pairs of operations, which will certainly at some point cause an underflow of the buffer. Then, we have a remove fail case
- `test_work_3` will set the bounds to unbounded or 0-30 exclusive randomly. So we will certainly see some truncation happen.
- I am not sure whether we will see any `std::length_error`s being caught in this way. That is why I made a `test_length`, which is executed after joining t1, t2, t3, t4, to see whether whatever the deque will throw gets handled correctly and logged. Unfortunately, std::deque is able to handle up to 10000000 elements without any `length_error` thrown, so I could not in fact see this behaviour.

There is no other way to check whether this all works than to turn on DEBUG and look for unexpected behaviour. In defense of this rather handwavy testing,

- Creating well-defined tests for multiple threads that *will* have nondeterminism in their traces, is very hard.
- Unit testing is a non-exhaustive game anyway.

Deadlocks and Starvation

Deadlocks: We can prove that there can be no deadlock in our program.

If there is a situation with a deadlock, this means that we have a circular wait, and this can only involve two mutexes because there are but two mutexes in the entire program, `buf_mutex` and `log_mutex`. Now, this means: some thread holding a the buffer mutex is waiting on another thread holding the logger mutex.

BUT a thread holding the logger mutex never has to wait on the logger's mutex, because there is no hold-and-wait possible in any function where the logger's mutex is being held (check this, it is only in Logger::write, and this function never tries to access a buffer.)

=> conclusion: there can be no hold-and-wait involving the logger's mutex being held, so that excludes one of the necessary conditions of a deadlock and hence rules out the existence of deadlocks.

Starvation: Starvation happens when some thread is waiting indefinitely. There are but 6 kinds of threads, because we can distinguish them by the action that they want to perform: `Buffer::add`, `Buffer::remove`, `Buffer::set_bound`, `Buffer::set_unbounded`, `Logger::read`, `Logger::write`. The other function that I implemented are for testing purposes or construction and I will assume threads never call these.

In that case, starvation happens if a thread is preempted continuously. But our code does not imply any priorities on the different functions: unlike in the reader-writer problem, we don't have multiple readers, so we don't have to lock when reading, nor give priority to them (or to the writers), so these cannot starve w.r.t. each other. By "no priority is given", I mean that there is no control flow in the code where knowledge of a certain type of thread waiting for the Buffer/Logger will lead to that thread being given first access after release by the current thread.

Finally, because of failing operations, no thread will be waiting indefinitely before an empty or full buffer condition. Whether you consider this indefinite waiting a deadlock or starvation, it cannot happen anyway.