Lab 2 Report – Double Buffering, Producer/Consumer, and Benchmarking

Compiled by Student

February 16, 2026

Abstract

This report documents Tasks 1, 2, and 3 from Lab 2. Each task implements and demonstrates approaches for buffered character input, a producer/consumer reading pattern, and a benchmark comparing single- and double-buffered streams. The following sections summarize the implementation, execution observations, and rationale for the chosen approaches.

Task 1 – Double Buffering

Objective:

- Implement a simple double-buffered reader that fills two fixed-size buffers and uses a sentinel to detect buffer boundaries.

Implementation Steps:

- Created `BufferManager` (in `task1.py`) with two buffers: `buffer1` and `buffer2`.
- On initialization, both buffers are filled from the input file and a sentinel is appended to mark the end of each buffer.
- `getNextChar()` reads from the current active buffer; when the sentinel is reached, it either stops (EOF) or switches the active buffer and fills the emptied buffer from the file.
- `switch_buffer()` resets the `forward` pointer, fills the other buffer, updates `active`, and increments `switch_count`.

Observations:

- Program prints contents of each buffer when filled and shows per-character reads as it advances through the active buffer.
- Sentinel detection triggers buffer switching; switch counts are printed at the end.

Rationale:

- Double buffering allows continuous character consumption without blocking on I/O.
- Using a sentinel simplifies boundary detection: only comparison with sentinel value is needed.

Task 2 – Producer / Consumer with Double Buffers

Objective:

- Demonstrate threading with a producer that reads the file and a consumer that scans characters concurrently using the `BufferManager`.

Implementation Steps:

- Wrote `task2.py`, creating a `BufferManager` and two threads: `producer` and `consumer`.
- `producer` repeatedly calls `getNextChar()` until EOF to fill buffers from a dedicated thread.
- `consumer` simulates a scanner: takes characters (protected by a `lock` around `getNextChar()` calls) and processes them with a small sleep to emulate work.
- A global `done` flag signals EOF to the consumer once the producer finishes reading.

Observations:

- Producer completes reading and prints a completion message.
- Consumer concurrently prints scanned characters; locking ensures safe buffer access.
- Final output includes total buffer switches recorded in the buffer manager.

Rationale:

- Producer/consumer pattern decouples I/O from processing, improving throughput.
- Using a lock keeps access simple and safe; higher-performance alternatives could use lock-free or finer-grained synchronization.

Task 3 – CharStream, SingleBufferStream, and Benchmarking

Objective:

- Implement `CharStream` (double buffer) with diagnostics, `SingleBufferStream` for comparison, and a `benchmark()` to measure performance and buffer fills.

Implementation Steps:

- Implemented `CharStream` with explicit buffer offsets, `absolute_pos` tracking, `lexemeBegin` support, and a `transitions` log capturing buffer switch metadata.
- Implemented `SingleBufferStream` that refills a single buffer at sentinel detection and measures fill durations.
- Wrote `benchmark()` to run both streams over the same file and record processing times, average fill times, character counts, and buffer switch counts.
- `print_report()` prints a summary and shows a buffer-transition demo highlighting bytes around each buffer switch.

Observations:

- Benchmarking quantifies single vs. double buffering performance.
- `transitions` verify lexeme continuity across buffer boundaries.
- Sample tokens show that scanning works correctly on the read content.

Rationale:

- Comparing single vs. double buffering quantifies the benefit of overlapping I/O and processing.

- Recording fill durations and transition snapshots aids debugging and performance analysis.
- Keeping both implementations together allows direct, repeatable comparison.

Conclusions and Possible Improvements

- Double buffering reduces processing pauses caused by I/O latency.
- Producer/consumer pattern decouples reading from scanning; correct synchronization is crucial for performance and correctness.
- Possible improvements: use lock-free queues, increase buffer size, or employ asynchronous I/O for large files.
- Add unit tests for boundary conditions (small files, exact buffer divisions) to ensure sentinel and switching logic robustness.

Files Referenced

- `task1.py` – BufferManager (double buffering example)
- `task2.py` – Producer/consumer demonstration using BufferManager
- `task3.py` – CharStream, SingleBufferStream, and benchmark/report code
- `tasksample.cpp` – Sample input used for tests and benchmarks