Programming Project 2: Reader/Writer Simulation
CS415 - Operating Systems

Codey Sivley

3/1/2022

For Dr. Lewis Spring 2022

Intro:

This assignment is an exercise in multithreading and mutual exclusion.

Much of the code was provided by Dr. Lewis, so changes and my personal input are highlighted below.

Major Functions:

`void simulateInput(int rate)`

This code was provided by Dr. Lewis.

Changes:

- Changed uniform distribution lower bound from 0 to 1, to align with thread numbering.
- Chrono
  - Where there was previously an infinite loop, replaced condition with a chrono check.
  - Each loop updates the time difference between the start and the current time.
  - When time exceeds user input time, function returns and thread joins.
- Exit commands
  - Since `processData()` expected 999 in the device field to return from function, added in a loop that pushes 999s to the inputQueue right before closing.
- cout <<
  - Printed each input to the console for debugging purposes. Commented out for now.

`void processData()`

This code was provided by Dr. Lewis.

Changes:

- Fstream
  - Ran into trouble with fstream. Since the thread buffers are only writing to this file, changed to ofstream. This cleared up odd issues I was having with the file output.
- FileName
  - Added ".txt" to filename generation, so I could view the files without having to open notepad++.
- Scoped_lock
  - Created a scoped_block instead of using the standard lock_guard.
  - Had to change project C++ code to C++17 (default was C++14)
  - This block surrounds the `if(!inputQueue.empty())` check
    - I was getting race conditions when this lock was after the check
    - Code was crashing on `.back()` on an empty deque.

        - The scoped_lock has it's own scope just inside the `while(keepProcessing)` loop. I think that scoping the entire loop was causing it to never unlock. I'll need to do some more checking on that.
- dataStream.close() added at the end of the function call so file closes before thread joins.

`void multithreadDriver()`

This code was written by me. The matrix multiplication example source code was used as reference.

Notes:

- Ran into issues with thread numbers aligning. Adjusted uniform distribution above to resolve this.
- Threads call function by reference. I am unsure if it is better to call by reference or not.
- Otherwise, standard procedure:
    - Generate array of thread objects.
    - The first thread is assigned to run the input generator, pushing to the deque.
    - The next 10 threads are assigned to run readers, each which pop matching device numbered inputs to the thread output file.
        - Was getting deadlock issues waiting for a file with device number 0.
    - After 5 minutes, input generator returns and joins.
    - Delay for 5 seconds, then close and join reader threads.
    - Nice little cout messages display updates so we can keep up.

Source code:

```
// OpSysMultithreadingBuffers.cpp : This file contains the 'main' function.
Program execution begins and ends there.
// codey sivley CS415 OpSys Spring 2022

#include <iostream>
#include <random>
#include <thread>
#include <string>
#include <sstream>
#include <deque>
#include <chrono>
#include <fstream>
#include <mutex>

std::deque<std::string> inputQueue; //global shared between read and write
threads

std::mutex readMTX;
std::mutex writeMTX;

static const int RUNTIME = 300; //runtime in seconds.
static const int RATE = 3; //poisson rate

//input sim
void simulateInput(int rate)
```

```
{
    std::random_device rd{};
    std::mt19937 gen{ rd() };
    int count = 0;

    std::poisson_distribution<> pD(rate);
    std::normal_distribution<> nD(5.0, 3.0);
    std::uniform_int_distribution<int> uD(1, 10);

    auto startClock = std::chrono::steady_clock::now();
    auto currentClock = std::chrono::steady_clock::now();
    std::chrono::duration<double> delta = currentClock - startClock;
    while (delta.count() < RUNTIME) //instead of infinite loop, runs for user
set runtime
    {
        int numberEvents = pD(gen);
        if (numberEvents > 0)
        {
            for (int i = 0; i < numberEvents; ++i)
            {
                std::stringstream sampleStream;
                int deviceNumber = uD(gen);
                float sample = nD(gen);
                count++;

                {//mutex lock
                    std::lock_guard<std::mutex> writeLock(writeMTX);
                    sampleStream << count << " " << deviceNumber << " " <<
sample;
                    //std::cout << sampleStream.str() << std::endl; //for
debugging
                    inputQueue.push_front(sampleStream.str()); //critical
section, write to queue
                }//mutex unlock
            }
            std::this_thread::sleep_for(std::chrono::milliseconds(1000));
        }
        currentClock = std::chrono::steady_clock::now();
        delta = currentClock - startClock; //update time passed
    }
    //send exit to processData by stuffing buffer with 999s
    for (int i = 0; i < 11; ++i)
    {
        std::stringstream sampleStream;
        {//mutex lock
            std::lock_guard<std::mutex> writeLock(writeMTX);
            sampleStream << count << " " << 999 << " " << 999;
            inputQueue.push_front(sampleStream.str()); //critical section,
write to queue
        }//mutex unlock
```

```
    }
}

//for processData, referenced:
https://en.cppreference.com/w/cpp/thread/scoped_lock
// and changed project C++ standard from C++14 to C++17
void processData(int devNum) {
    std::ofstream dataStream; //no need to read, just output
    bool keepProcessing = true;
    std::string fileName = "device" + std::to_string(devNum) + ".txt";
    int count;
    int device;
    float value;
    dataStream.open(fileName);
    while (keepProcessing) {
        { //scope lock
            std::scoped_lock bothLocks{ readMTX, writeMTX }; //lock outside,
locking inside causes race condition on back()
            if (!inputQueue.empty()) {
                std::string sample = inputQueue.back();
                std::stringstream sampleStringStream(sample);
                sampleStringStream >> count >> device >> value;
                if (device != 999) {
                    if (devNum == device) {
                        inputQueue.pop_back();
                        dataStream << sample << std::endl;
                    }
                }
                else {
                    std::cout << "Device " << devNum << " has seen exit
message." << std::endl;
                    keepProcessing = false;
                }
            }
        }//end scope lock to check for updates.
    }
    dataStream.close();
}

void multithreadDriver() {

    std::thread threads[11]; //1 simulate input, 10 readers
    threads[0] = std::thread(&simulateInput, RATE);
    for (int i = 1; i < 11; ++i) {
        threads[i] = std::thread(&processData, i);
    }
    threads[0].join(); //input thread ends
    std::cout << "Input thread ended." << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(5000)); //wait for
buffers to clear.
```

```
    std::cout << "Attempt close buffers." << std::endl;
    for (int i = 1; i < 11; ++i) {
        threads[i].join();
        std::cout << "Thread " << i << " closed." << std::endl;
    }
}
int main()
{

    std::cout << "Multithread start: " << std::endl;
    multithreadDriver();
    std::cout << "Multithread end." << std::endl;

    return 0;
}
```

Output file example:

device1.txt:

1 1 2.80989

13 1 5.45452

14 1 7.96567

16 1 5.62436

31 1 5.15506

33 1 5.74692

43 1 5.43173

53 1 8.61345

60 1 1.76718

61 1 8.12361

73 1 5.00635

84 1 -0.216402

101 1 7.43333

104 1 5.15983

114 1 4.84564

123 1 7.88437

161 1 0.696633

162 1 3.55331

186 1 3.6182

189 1 10.1194