use Adobe Reader to complete

(Type in fields)

Basics3 – Locks

Student Information

Integrity Policy: All university integrity and class syllabus policies have been followed. I have neither given, nor received, nor have I tolerated others' use of unauthorized aid.

I understand and followed these policies: ● Yes ○ No

Name: Joey Domino

Date: 9/29/2022

Submission Details

Final ***Changelist*** number: this one

Verified build: ● Yes ○ No

Number Tests Passed: 5

Required Configurations: 1: { Debug | x86 }

Discussion (What did you learn):

I learned how to lock threads using:
lock_guard
adopt_lock
unique_lock
defer_lock

and how to call a function only once when multiple threads can touch the call using call_once use Adobe Reader to complete

(Type in fields)

Verify Builds

- Follow the Piazza procedure on submission
  - Verify your submission compiles and works at the changelist number.
- Verify that only MINIMUM files are submitted
  - No – Generated files
    - *.pdb, *.suo, *.sdf, *.user, *.obj, *.exe, *.log, *.pdb, *.db
    - Anything that is generated by the compiler should not be included
  - No – Generated directories
    - /Debug, /Release, /Log, /ipch, /.vs
- Typical files project files that are required
  - *.sln, *.suo,
  - *.vcxproj, *.vcxproj.filters, *.vcxproj.user
  - *.cpp, *.h
  - CleanMe.bat

Standard Rules

Submit multiple times to Perforce

- Submit your work as you go to perforce several times (at least 5)
  - As soon as you get something working, submit to perforce
  - Have reasonable check-in comments
    - Seriously, I'm checking

Write all programs in cross-platform C++

- Optimize for execution speed and robustness
- Working code doesn't mean full credit

Submission Report

- Fill out the submission Report
  - No report, no grade

Code and project needs to compile and run

- Make sure that your program compiles and runs
  - Warning level ALL …
  - NO Warnings or ERRORS
    - Your code should be squeaky clean.
  - Code needs to work "as-is".
    - No modifications to files or deleting files necessary to compile or run.
  - All your code must compile from perforce with no modifications.
    - Otherwise it's a 0, no exceptions

Project needs to run to completion

- If it crashes for any reason…
  - It will not be graded and you get a 0

Leave Project Settings

- Do NOT change the project or warning level
  - Any changing of level or suppression of warnings is an integrity issue

Leaking Memory

- If the program leaks memory
  - There is a deduction of 20% of grade
- If a class creates an object using new/malloc
  - It is responsible for its deletion
- Any ***MEMORY*** dynamically allocated that isn't freed up is ***LEAKING***
  - Leaking is ***HORRIBLE***, so you lose points

No Debug code or files disabled

- Make sure the program is returned to the original state
  - If you added debug code, please return to original state
- If you disabled file, you need to re-enable the files
  - All files must be active to get credit.
  - Better to lose points for unit tests than to disable and lose all points
- Disable your debug printing otherwise you will lose points

Due Dates

- See Piazza for due date and time
- Submit program perforce in your student directory assignment supplied.
- Fill out your this ***Submission Report*** and commit to perforce
  - ***ONLY*** use Adobe Reader to fill out form, all others will be rejected.
  - Fill out the form and discussion for full credit.

Goals

- Learn
  - Protecting shared data
    - Mutex, Lock_Guard, Unique_Locks
    - Call_once

Assignments

1. ***Problem_1***
   - BACKGROUND
     - Several threads are spawned from a functor named ATTACK.
     - Attack functors send data to a common function call Problem_1::Add(int Val);
       - It does this in a loop… sending the data in a very quickly in a single thread
     - Several thread are launched at once… with different start and delta values
       - All of these thread are attacking one single Problem_1::Add(int Val) with a shared state.
     - Problem_1::Add(int Val)
       - Adds Value to a data structure
       - Prints each addition with a small delay to amplify the race conditions
       - You can see the tearing of prints in the output window
   - ACTION
     - Add protection to prevent tearing in Problem_1::Add(int Val)
       - Do not modify any sleeps, just add protection to the method
     - Add a ***mutex*** and use ***lock_guard***
     - Add data to the class as needed

2. ***Problem_2***
   - BACKGROUND
     - Cass Student that holds its score and name.
     - Several threads are spawn to add a constant value to 3 different students' score.
     - A single thread is launch taking a random ordering of the students as it argument.
       - Functor Problem_2 – contains the calling function
       - Problem_2::operator()(…) - is the entry point for the thread.
       - This function locks the input students mutexs and does the addition to each student
     - Data isn't be updated consistently
       - Since many threads are spawned all calling the same functor.
       - Students arguments are passed in a random order
       - There is locking of student's mutex that creates an order relative deadlock
   - ACTION
     - Add protection to prevent deadlock in Problem_2::operator()(…)
     - Use the existing mutexes, with ***lock_guard*** and ***adopt_lock***
     - Add data to the class as needed

3. ***Problem_3***
    - BACKGROUND
        - Cass Student that holds its score and name.
        - Several threads are spawn to add a constant value to 3 different students' score.
        - A single thread is launch taking a random ordering of the students as it argument.
            - Functor Problem_3 – contains the calling function
            - Problem_3::operator()(…) - is the entry point for the thread.
            - This function locks the input students mutes and does the addition to each student
        - Data isn't be updated consistently
            - Since many threads are spawned all calling the same functor.
            - Students arguments are passed in a random order
            - There is locking of student's mutex that creates an order relative deadlock
    - ACTION
        - Add protection to prevent deadlock in Problem_3::operator()(…)
        - Use the existing mutexes, with **unique_lock** and **defer_lock**
        - Add data to the class as needed

4. ***Problem_4***
    - BACKGROUND
        - Cass Student that holds its score and name.
        - Several threads are spawn to add a constant value to 3 different students' score.
        - A single thread is launch taking a random ordering of the students as it argument.
            - Functor Problem_4 – contains the calling function
            - Problem_4:operator()(…) - is the entry point for the thread.
            - This function locks the input students mutes and does the addition to each student
        - Data isn't be updated consistently
            - Since many threads are spawned all calling the same functor.
            - Students arguments are passed in a random order
            - There is locking of student's mutex that creates an order relative deadlock
    - ACTION
        - Add protection to prevent deadlock in Problem_4::operator()(…)
        - Use the existing mutexes, with ***unique_lock*** and ***adopt_lock***
        - Add data to the class as needed

5. ***Problem_5***
    - BACKGROUND
        - Class Dog that holds its name.
        - Dog has the calling function Dog::operator()(…)
            - In the calling function, a dog object is passed
            - It does a print then calls AlphaDog::SetAlphaDog()
        - Several Dog objects are launched in separate threads at once.
            - Each thread then calls AlphaDog::SetAlphaDog() at once
                a. Actually each thread calls this method
    - ACTION
        - Rework Dog::operator()(…)
            - To use ***call_once***() and any appropriate ***flags*** to insure that SetAlphaDog() is only called once not many times.
            - Remember threads are launched in random order, independent of the order declared in the unit test.

6. ***Make sure it builds for all configurations***
    - Suggestion: Implement and develop on Debug/x86
    - After that configuration works → verify all 1 configurations:
        - Debug x86

Validation

Simple checklist to make sure that everything is submitted correctly

- Is the project compiling and running without any errors or warnings?
- Does the project run ***ALL*** in all configurations without crashing?
- Is the submission report filled in and submitted to perforce?
- Follow the verification process for perforce
    - Is all the code there and compiles "as-is"?
    - No extra files
- Is the project leaking memory?

Hints

Most assignments will have hints in a section like this.

- Do many little check-ins
    - Iteration is easy and it helps.
    - Perforce is good at it.
- READ the book (chapter 3)
    - Many good ideas in there.

- I had to do a lot of googling and web searching
  - Not many examples out there.
  - Dig into it you’ll get it