C++ Exercises
Set8

Author(s): Ivan Yanakiev, Alex Swaters
–
22:47

January 31, 2023

57

```
The implementation of the class Semaphore
was followed using the Annotations.
```

Listing 1: `Semaphore.h`

```
#ifndef SEMAPHORE_H
#define SEMAPHONE_H

#include <mutex>
#include <condition_variable>

class Semaphore
{
    std::size_t d_available;
    std::mutex d_mut;
    std::condition_variable d_cond;

public:
    Semaphore(std::size_t available);

    void notify();
    void notify_all();

    std::size_t size() const;

    void wait();
};

inline std::size_t Semaphore::size() const
{
    return d_available;
}

#endif
```

Listing 2: `ctor.cc`

```
#include "Semaphore.h"

Semaphore::Semaphore(std::size_t size) : d_available(size)
{
}
```

Listing 3: `notify.cc`

```
#include "Semaphore.h"

void Semaphore::notify()
{
    std::lock_guard<std::mutex> lg(d_mut);
    if (d_available++ == 0)
        d_cond.notify_one();
}
```

Listing 4: notify_all.cc

```
#include "Semaphore.h"

void Semaphore::notify_all()
{
    std::lock_guard<std::mutex> lg(d_mut);
    if (d_available++ == 0)
        d_cond.notify_all();
}
```

Listing 5: wait.cc

```
#include "Semaphore.h"

void Semaphore::wait()
{

    std::unique_lock<std::mutex> ul(d_mut);
    while (d_available == 0)
        d_cond.wait(ul);

    --d_available;
}
```

58

```
First we create the matrices and fill them.
Next we init a 4x6 array of futures to store the results.
Then we loop over those dimensions, computing an inner
product at each position. This computation is done in a
packaged task containing a lambda which simply calls
the inner product on the relevant positions. The lambda
uses = as default capture because & causes the last one
or a few more to be uninitialized. Finally we display the
output once the futures finish. Example output:
    40 40 40 40 40 40
    40 40 40 40 40 40
    40 40 40 40 40 40
    40 40 40 40 40 40
Which is as expected since our first array is all twos and
the second is all fours (2*4=8 and the inner dimension =5).
```

Listing 6: main.cc

```
#include <algorithm>
#include <future>
#include <numeric>
#include <iostream>

int main()
{
    double lhs[4][5];
    double rhs[6][5];
    std::for_each(lhs, lhs + 4, [](auto &v){ std::fill(v, v + 5, 2); });
    std::for_each(rhs, rhs + 6, [](auto &v){ std::fill(v, v + 5, 4); });

    std::future<double> fut[4][6];
    for (size_t row = 0; row != 4; ++row)
    {
        for (size_t col = 0; col != 6; ++col)
        {
            std::packaged_task<double()> task([=]
            {
                return std::inner_product(
                    lhs[row], lhs[row] + 5, rhs[col], 0.0);
            });
            fut[row][col] = task.get_future();
            std::thread(std::move(task)).detach();
        }
    }

    for (size_t row = 0; row != 4; ++row)
    {
        for (size_t col = 0; col != 6; ++col)
            std::cout << fut[row][col].get() << ' ';
        std::cout << std::endl;
    }

}
```

59

In order to make use of std::async we
just need to issue each recursive call
to qsort to be done in a seperate thread.
At the end of the function we can just wait
for both sides to be sorted. Then in main
we create an array from 0 to 99 using iota.
The we shuffle it and print it so we validate
that the array is shuffled. After that we
proceed to sort it using the qsort and then
print it again to validate that it is indeed
sorted.

Listing 7: main.cc

```
#include "main.ih"

int main(int argc, char **argv)
{
    // 59 + function qsort_mt
    size_t const iaSize{100};
    int ia[iaSize];

    std::iota(ia, ia + iaSize, 0);
    std::random_shuffle(ia, ia + iaSize);

    std::cout << "Before sort:\n";
    std::copy(ia, ia + iaSize, std::ostream_iterator<int>(std::cout, " "));
    std::cout << std::endl;

    qsort_mt(ia, ia + iaSize);

    std::cout << "After sort:\n";
    std::copy(ia, ia + iaSize, std::ostream_iterator<int>(std::cout, " "));
    std::cout << std::endl;
}
```

60

In order to use multiple threads we can
create a vector<future<string>> which will
store the future results of each thread. Then
in the main while loop we will just need to
iterate over the vector and check if a thread
has finished if so we end the program.

Listing 8: main.cc

```
#include "main.ih"

```

```
int main(int argc, char **argv)
{
    // 60
    // Start the threadFun in a separate thread
    std::future<std::string> result = std::async(std::launch::async, threadFun);

    size_t count = 0;
    while (true)
    {
        // do the main-task
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::cerr << "inspecting: " << ++count << '\n';

        // inspect whether the thread indicates to end the program
        if (result.wait_for(std::chrono::seconds(0)) == std::future_status::
           ready)
        {
            std::cerr << "done\n";
            break;
        }
    }
    return 0;
}
```

62

```
The solution will revolve around the
following components.

1. A main thread that will be responsible
for parsing the comandline arguments,reading
list of source files, initializing threads and their
data structures.

2. A thread pool (workers) of configurable amount.
Each worker will be responsible for compiling
a single source file at a time. The number of workers
will be defined by the user (the default is the number of
cores on the computer). If the user enters 0 then 1 is
silently used.

3. A shared data structure (such as queue) that will be
used to pass the source files from main thread to
workers.

4. A shared data structure (such as set). This will keep
track of which source files have been compiled (and which
have failed).

5. Combination of lock_guard and mutex to synchronize the
data structures so that no race conditions occur.

6. Semaphore will be used to signal to the main thread
when a compilation fails. This will indicate to the main
thread to stop and display the error message, then end
the program.

The flow of the program would be as follows:

1. Main thread parses arguments and read the list of source files.

2. Main thread initializes data structures and thread pool.

3. Main thread adds source files to the shared queue

4. Worker threads take source files from the queue and
compile them.
```

5. If compilation fails , the worker signals the Semaphore
and adds the error output and failed source to the shared set .

6. Main waits for signal , when it is it stops and displays error
output if any .

7. Main thread removes the temp directory if specified

8. end the program

Classes :

1. BlockingQueue
2. BlockingSet

Both of these classes are independent
of one another and both of them will
be protected by mutexes to avoid races .
These classes will store the files that
have to be and have been compiled .

3. CompileWorker

This class will encapsulate a thread responsible
for compiling a source file .

4. Semaphore

This will a binary semaphore since even if 1 file
fails to compile we have problem . Used to signal to
main to end program and print error output .

5. FileFinder

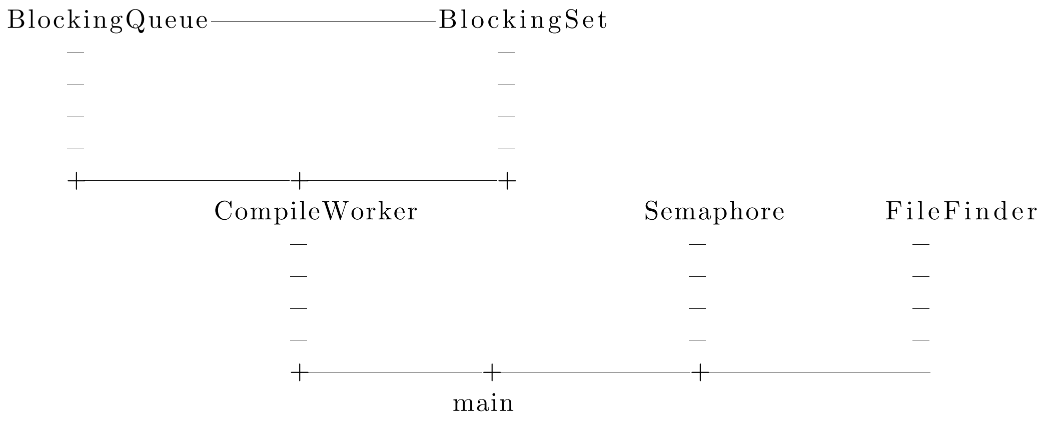
This class will handle finding the source file locations
and gathering them into a vector . The class will expect
the user to :

A. Specify a directory to search
B. Specify a file that contains the location of all sources
C. If none specified we will search the current directory for
source files .

Possible classes :

1. CompileFile (maybe if we need to store extra information
about the file it will be good to encapsulate it into its
own object)

Hierarchy :



This hierarchy shows how the classes that we create

```
will function between one another. We exclude other
classes that main will use like:
    vector
    queue
    thread (or async,packaged_task)
    etc
since these classes are already provided by the language
and we will not design them.
```

63

```
Implementing based on the plan in ex 62.
Still a WIP, hoping for feedback on what is
completed.
```

Listing 9: filefinder/FileFinder.ih

```
#include "FileFinder.h"
#include <iostream>
#include <filesystem>
```

Listing 10: filefinder/FileFinder.h

```
#ifndef FILE_FINDER_H
#define FILE_FINDER_H

#include <vector>
#include <string>

class FileFinder
{
public:
    static FileFinder *getInstance();
    static void deleteInstance();

    std::vector<std::string> getSourcesFromFile(std::string const &fileName);
    std::vector<std::string> getSourcesFromDir(std::string const &dirName);
    std::vector<std::string> getSourcesFromDirRec(std::string const &dirName);

private:
    static FileFinder *s_instance;
    FileFinder();
    FileFinder(FileFinder const &other) = delete;
    FileFinder &operator=(FileFinder const &other) = delete;

    // could be extended to support user input of vector of types to check
    // not needed for this program though
    bool checkExtension(std::string const &ext);
};

#endif
```

Listing 11: filefinder/check_ext.cc

```
#include "FileFinder.ih"

bool FileFinder::checkExtension(std::string const &ext)
{
    if (ext == "cpp" || ext == "cc")
        return true;
    return false;
}
```

Listing 12: filefinder/delete_instance.cc

```
#include "FileFinder.ih"

void FileFinder::deleteInstance()
```

```
{
    if (s_instance)
    {
        delete s_instance;
        s_instance = nullptr;
    }
}
```

Listing 13: filefinder/file_ctor.cc

```
#include "FileFinder.ih"

FileFinder::FileFinder()
{
}
```

Listing 14: filefinder/get_instance.cc

```
#include "FileFinder.ih"

FileFinder *FileFinder::getInstance()
{
    if (!s_instance)
        s_instance = new FileFinder;

    return s_instance;
}
```

Listing 15: filefinder/get_sources_dir.cc

```
#include "FileFinder.ih"

std::vector<std::string> FileFinder::getSourcesFromDir(std::string const &dir)
{
    std::vector<std::string> sourceFiles;

    for (auto const &entry : std::filesystem::directory_iterator(dir))
    {
        if (entry.is_regular_file())
        {
            std::string fileName = entry.path().string();
            std::string extension = fileName.substr(fileName.find_last_of(".") +
                1);
            if (checkExtension(extension))
                sourceFiles.push_back(fileName);
        }
    }
    return sourceFiles;
}
```

Listing 16: filefinder/get_sources_dir_rec.cc

```
#include "FileFinder.ih"

std::vector<std::string> FileFinder::getSourcesFromDirRec(std::string const &
   dirName)
{
    std::vector<std::string> sourceFiles;

    for (auto const &entry : std::filesystem::recursive_directory_iterator(
       dirName))
    {
        if (entry.is_regular_file())
        {
            std::string fileName = entry.path().string();
            std::string ext = fileName.substr(fileName.find_last_of(".") + 1);
            if (checkExtension(ext))
                sourceFiles.push_back(fileName);
        }
    }

    return sourceFiles;
}
```

Listing 17: filefinder/get_sources_file.cc

```
#include "FileFinder.ih"

std::vector<std::string> FileFinder::getSourcesFromFile(std::string const &
   fileName)
{
    std::vector<std::string> sourceFiles;

    // get filenames from file
    // call helper to find source
    // return set
    // throw error if cannot find all files

    return sourceFiles;
}
```

Listing 18: filefinder/instance.cc

```
#include "FileFinder.ih"

FileFinder *FileFinder::s_instance = nullptr;
```

Listing 19: queue/BlockQueue.ih

```
#include "BlockQueue.h"
```

Listing 20: queue/BlockQueue.h

```
#ifndef BLOCKING_QUEUE_H
#define BLOCKING_QUEUE_H

#include <queue>
#include <string>
#include <mutex>

class BlockingQueue
{
    std::queue<std::string> d_queue;
    std::mutex d_mut;
    bool b_finished;

public:
    BlockingQueue();
    void push(std::string const &item);
    void pop();
    std::string &front();
    bool empty();

    bool getFinished() const;
    void setFinished(bool finished);
};

inline bool BlockingQueue::getFinished() const
{
    return b_finished;
}

inline void BlockingQueue::setFinished(bool finished)
{
    b_finished = finished;
}

#endif
```

Listing 21: queue/ctor.cc

```
#include "BlockQueue.ih"

BlockingQueue::BlockingQueue()
    : b_finished(false)
{
}
```

Listing 22: queue/empty.cc

```
#include "BlockQueue.ih"

bool BlockingQueue::empty()
{
    std::lock_guard<std::mutex> lg(d_mut);
    return d_queue.empty();
}
```

Listing 23: queue/front.cc

```
#include "BlockQueue.ih"

std::string &BlockingQueue::front()
{
    std::lock_guard<std::mutex> lg(d_mut);

    return d_queue.front();
}
```

Listing 24: queue/pop.cc

```
#include "BlockQueue.ih"

void BlockingQueue::pop()
{
    std::lock_guard<std::mutex> lg(d_mut);
    d_queue.pop();
}
```

Listing 25: queue/push.cc

```
#include "BlockQueue.ih"

void BlockingQueue::push(std::string const &item)
{
    std::lock_guard<std::mutex> lg(d_mut);
    d_queue.push(item);
}
```

Listing 26: main.cc

```
#include "filefinder/FileFinder.h"

#include <string>
#include <iostream>
#include <thread>

int main(int argc, char **argv)
{
    std::string dir{"."};
    unsigned numThread{};

    // if no args then max threads + current dir
    if (argc < 2)
        numThread = std::thread::hardware_concurrency();

    if (argc == 2) // user defined thread num
    {
        if (std::stoi(argv[1]) == 0)
            numThread = 1;
        else
            numThread = std::stoi(argv[1]);
    }

    if (argc == 3) // user defined dir
    {
        // cod duplication mode it into a function
        if (std::stoi(argv[1]) == 0)
            numThread = 1;
        else
            numThread = std::stoi(argv[1]);

        dir = argv[2];
    }

    FileFinder *fileFinder = FileFinder::getInstance();
    std::vector<std::string> sources = fileFinder->getSourcesFromDirRec(dir);
    std::cout << numThread << '\n';

    // TODO: init blocking queue and workers, then start workers
}

// TODO make a CompileFlag class to store the name of the file that we compile
// also potentially the error output file
```

64

Listing 27: semaphore/semaphore.h

```
#ifndef SEMAPHORE_H
#define SEMAPHORE_H

#include <mutex>
#include <condition_variable>

class Semaphore
{
    std::mutex d_mutex;
    std::condition_variable d_condition;
    size_t d_nAvailable;
public:
    Semaphore(size_t nAvailable) : d_nAvailable(nAvailable) {}
    void wait();
    void wait_for(size_t seconds);
    void set(size_t nAvailable);
    std::size_t size() const;
};

inline size_t Semaphore::size() const
{
    return d_nAvailable;
}

#endif
```

Listing 28: semaphore/semaphore.ih

```
#include "semaphore.h"
#include <mutex>
#include <condition_variable>
```

Listing 29: semaphore/set.cc

```
#include "semaphore.ih"

void Semaphore::set(size_t nAvailable)
```

```
{
    std::unique_lock<std::mutex> lock(d_mutex);
    d_nAvailable = nAvailable;
    d_condition.notify_all();
}
```

Listing 30: `semaphore/wait_for.cc`

```
#include "semaphore.ih"

void Semaphore::wait_for(size_t seconds)
{
    std::unique_lock<std::mutex> lock(d_mutex);
    d_condition.wait_for(lock, std::chrono::seconds(seconds), [this]()
                            { return d_nAvailable > 0; });
    --d_nAvailable;
}
```

Listing 31: `semaphore/wait.cc`

```
#include "semaphore.ih"

void Semaphore::wait()
{
    std::unique_lock<std::mutex> lock(d_mutex);
    d_condition.wait(lock, [this]()
                        { return d_nAvailable > 0; });
    --d_nAvailable;
}
```

Listing 32: `run.cc`

```
#include "main.ih"

void run(pid_t pid, int argc, char **argv)
{
    Semaphore sem(1);
    std::thread waitThread([&]()
    {
        int status;
        waitpid(pid, &status, 0);
        sem.set(1);
    });

    if (argc == 4)
    {
        sem.wait_for(std::stoul(argv[3]));
        if (sem.size() == 0)
        {
            kill(pid, SIGKILL);
            std::cout << "Program ended at timeout\n";
        }
        else
        {
            std::cout << "Program ended normally\n";
        }
    }
    else
    {
        sem.wait();
        std::cout << "Program ended normally\n";
    }
}
```

Listing 33: `main.cc`

```
#include "main.ih"

int main(int argc, char **argv)
```

```
{
    if (argc < 3)
    {
        std::cout << "Usage: " << argv[0] << " <program> <lines> [timeout]\n";
        return 1;
    }

    pid_t pid = fork();
    if (pid <= 0)
    {
        if (pid == 0)
        {
            execl(argv[1], argv[1], argv[2], NULL);
            std::cout << "Failed to execute program\n";
            return 1;
        }
        std::cout << "Failed to fork process\n";
        return 1;
    }
    else
        run(pid, argc, argv);
}
```