Single Thread-Contained Dispatcher as Middleware with Command as Request

1 Abstract

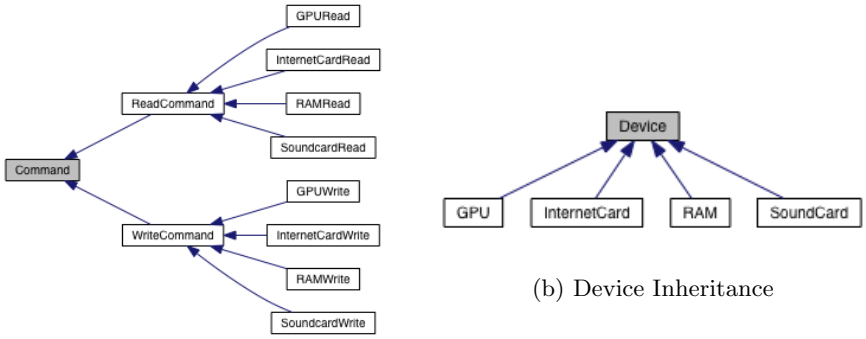
The analysis of this labs could be roughly summarised as: make device-type-generalized thread safe executions with *Command Pattern* while maintaining growth-allowed, read-write-specied and dynamic threads sets with *Singleton Pattern*.

In this work, we implemented tasks **case 2** with **C++**. Problems were deconstructed into two stages. Stage of the rst was required to wrap the device action from concrete characterised intefaces to a more unied read-write-distinguishable command object. Then threads with uniformed name of access method were poured into queues as threads pool, with r/w accordingly. With this design, an execution of a command will be covered by a life cycles of a single thread.

2 Introduction

Implementing in a OOP language, we rst extract the **Device** in include/abstract_devices.h. This module is found crucial in the manner of data-racing-avoiding design. Features of C++17[1] (best practice for RW lock nowadays with multiple readers and a single writer) (14 and 11 also included) are adopted in partial parts of this project[2] including but not limited to RAII[3] lock and RTTI[4] (sounds a sign of poor design). Considering implementaions of the **Device**, any of the devices resembling **InternetCard** or **GPU** will expose thread safe interfaces.

Bridging in between the devices and the commands, **ReadCommand** and **WriteCommand** are created for later types inferring with *dynamic_cast*. We rst mount devices, then create concrete commands, after which read-write-specied threads with **Command** member function *execute()* will be pushed into a thread container correspondingly. We may try to stop the execution following *Dispatcher::start()*, still the **Dispatcher** wait for all threadsexit, however. The dispatch man not only maintain queue of commands and pools of threads but also undertake responsibilities for threads spawning and converging.



(a) Command Inheritance

(b) Device Inheritance

Figure 1: Design of Abstract Command and Device

[1] https://en.cppreference.com/w/cpp/thread/shared_mutex
[2] https://github.com/ro-n/design-pattern
[3] https://en.cppreference.com/w/cpp/language/raii
[4] https://en.cppreference.com/w/cpp/language/dynamic_cast

3 Model

Basically our model could be covered by figure1 [5], through which1a wrapped command are capable of being inferred of their types to push into queues maintain by the dispatcher. Devices1b describe the inner design (structure) of the hardware, inside which we implement the shared read/write lock in case of concurrenet reading and writing.

(a) Dispatcher in Main()

(b) Dispatcher Start

(c) Dispatcher Terminate

As is illustrated in figure2a and figure2b, at the main thread will the static class **Dispatcher** call to *execute()* directly after the *start()*. Then comes the *stop()*2c, through which the dispatcher loops through the correspondent container (R/W in this) and try block the main thread with *std::thread::join()*, through which solves the early-end issue (that is, main thread exits earlier than the distributed work threads).

4 Details

In this section4, we hope to demostrate with Q&A.

- `By which means have we implemented the` ***`Command Pattern`***`?` First we implement **Command** with pure *virtual* functions as behaving like a preliminary interface. Then insert a tight-coupling class to provide the concrete command with certain types. The grad-level class do write the *execute()*.
- `By which means have we implemented the` ***`Singleton Pattern`***`?` The **Dispatcher** behaves the singleton. There roughly exists two ways for us to implement it in C++, which can be *static Dispatcher D; return D;* or converting all members to static. The latter was adopted.
- `How do we solve case 2?` Case 2 simply asks: more generalized, dynamic threads creating and removing which are solved by extraction of the abstraction of the device and two containers of threads respectively.
- `How do we implement the multi-reader lock?` We present the lock with the C++17 *shared_mutex*, which provides advantages of mutual access in between users within and without the same interfaces for the same resources.
- `How to ensure that lock works?` Hard to tell. One should consider a concurrenet reading from the resources followed by a writing request and the results of reading should always be the same as long as control the write to be the last thread to run with a conditional variable. Since our project only logs, it's hard to check.
- `Any improvements of this design?` Sure, more dynamic features should be added to the dispatcher. The *Dispatcher::start()* is not a read thread safe loop, we can concatenate the queue with a new queue of commands or maintain a thread continuously pour the threads from commands queue into the pools.
- `Why the output be undefined?` Though *std::cout* is a thread safe method, the **stdout** is shared due to the outprint mechanism. One should search all and replace them with a logging functions with a global mutex.

[5] Demo diagrams are fully generated by Doxygen at http://www.doxygen.nl/

5 Conclusion

In this task, we implemented tasks mainly focused on solving "command as object divided into threads". *Command Pattern* and *Singleton Pattern* were utilized for this design. Hopefully we've presented our thoughts clearly.

6 Appendix

```
mounting devices...
dispatch starting...
send in net card
write to gpu
sound card recording
write to ram
read from gpu
read from gpu
dispatch stopping...
read from ram
dispatch stopped
unmounting devices...
```