CS4414 Recitation 9

Performance(Gprof) & multi-threading

03/24/2023

Alicia Yang

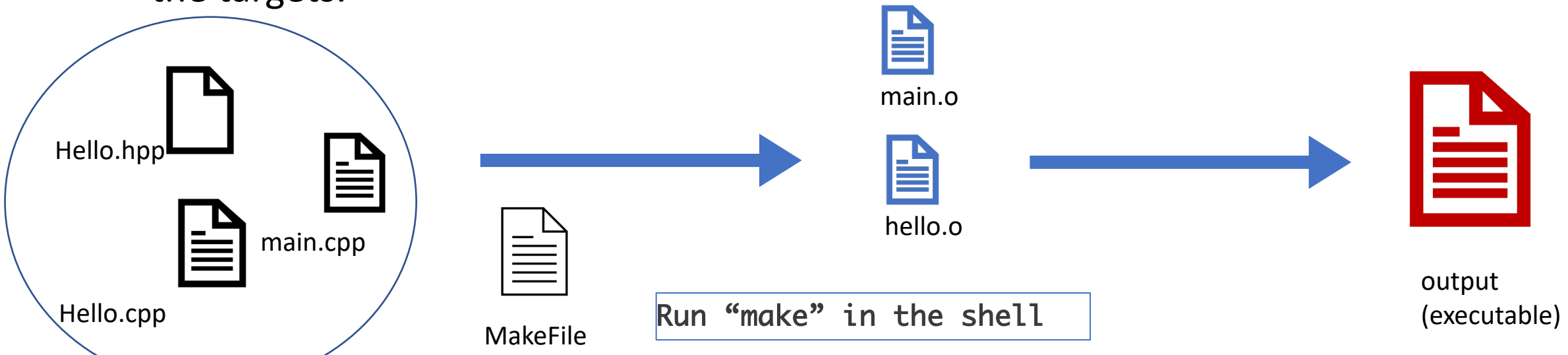
CMake

- What is CMake
- Simple CMake
- CMake with linked libraries
- CMake with flags

Code source:
https://github.com/aliciayuting/CS4414Demo.git

Build Files & Generate Executables --- MakeFile

- Makefile is just a text file that is used or referenced by the ‘make’ command to build the targets.



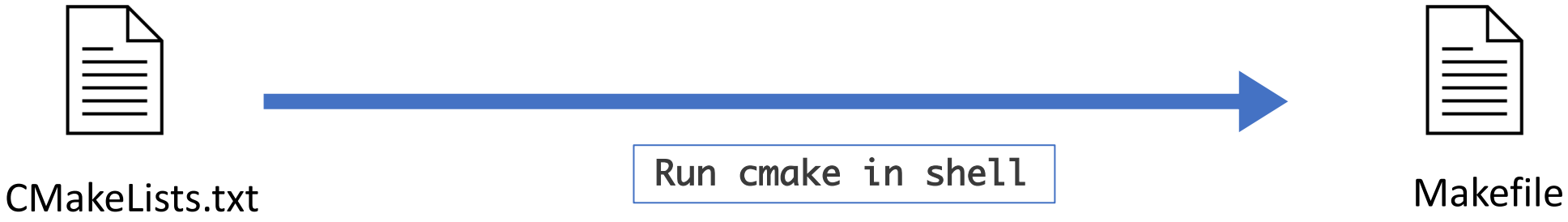

```
CC = g++
CFLAGS = -g -Wall
TARGET = output
all: $(TARGET)
$(TARGET): main.o hello.o
	$(CC) $(CFLAGS) -o $(TARGET) main.o hello.o
main.o: main.cpp hello.hpp
	$(CC) $(CFLAGS) -c main.cpp
hello.o: hello.hpp hello.cpp
	$(CC) $(CFLAGS) -c hello.cpp
```

CMake

- Why CMake?
  - Makefiles are low-level, clunky creatures
  - CMake is a higher level language to automatically generate Makefiles
  - CMake contains more features, such as finding library, files, header files; it makes the linking process easier, and gives readable errors
- What is CMake?
  - CMake is an extensible, open-source system that manages the build process in an operating system and in a compiler-independent manner.
- CMakeLists.txt files in each source directory are used to generate Makefiles

CMake

- Why CMake?
  - Makefiles are low-level, clunky creatures
  - CMake is a higher level language to automatically generate Makefiles
  - CMake contains more features, such as finding library, files, header files; it makes the linking process easier, and gives readable errors
- What is CMake?
  - CMake is an extensible, open-source system that manages the build process in an operating system and in a compiler-independent manner.
- CMakeLists.txt files in each source directory are used to generate Makefiles



Cmake

1.simple CMake

demo

- Helloworld demo example

cmakelists.txt

```
cmake_minimum_required(VERSION 3.12) # set the project
name project(MyProject) # add the executable
add_executable(output main.cpp)
```

- Build and Run
  - Navigate to the source directory, and create a build directory

    $ cd ./myproject & $ mkdir build
  - Navigate to the build directory, and run Cmake to configure the project and generate a build system

    $ cd build &. $ cmake ..
  - Call build system to compile/link the project

    either run. $ make

    or run. $ cmake –build .

Cmake

2. Cmake with libraries

- Why use library?
  - The C++ libraries are modular components of reusable code. Using class libraries, you can integrate blocks of code that have been previously built and tested.

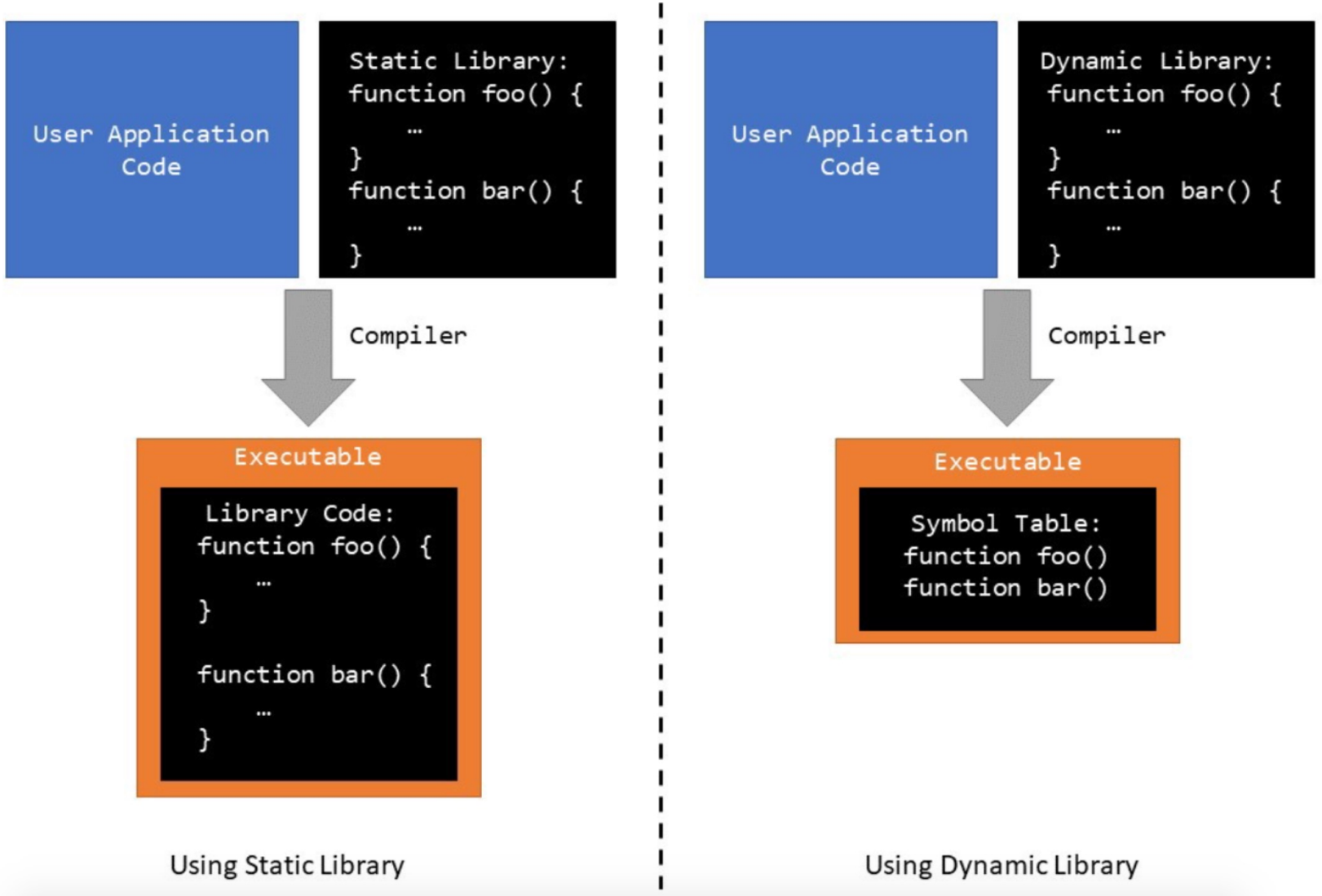
- What are in C++ library?
  - A C++ library consists of header files and an object library.
  - The header files provide class and other definitions that the library **exposes** (offers) to the programs using its.
  - The object library(precompiled binary) contains **compiled implementation** of functions and data that are linked with your program to produce an executable program.

Library Types in C++

- Static-linked library:
  - contains code that is linked to users' programs at compile time. (.a(archive) in linux, or .lib in windows)
  - consists of routines that are compiled and linked directly into your program
  - a copy of the library becomes part of every executable that uses it, this can cause a lot of wasted space. (Suppose building 100 executables, each one of them will contain the whole library code, which increases the code size overall)
- Dynamic(Shared) library:
  - contains code designed to be shared by multiple programs. (.so in linux, or .dll in wondows, .dylib in OS X files)
  - consists of routines that are loaded into your application at run time
  - many programs can share one copy, which saves space. (All the functions are in a certain place in memory space, and every program can access them, without having multiple copies of them)

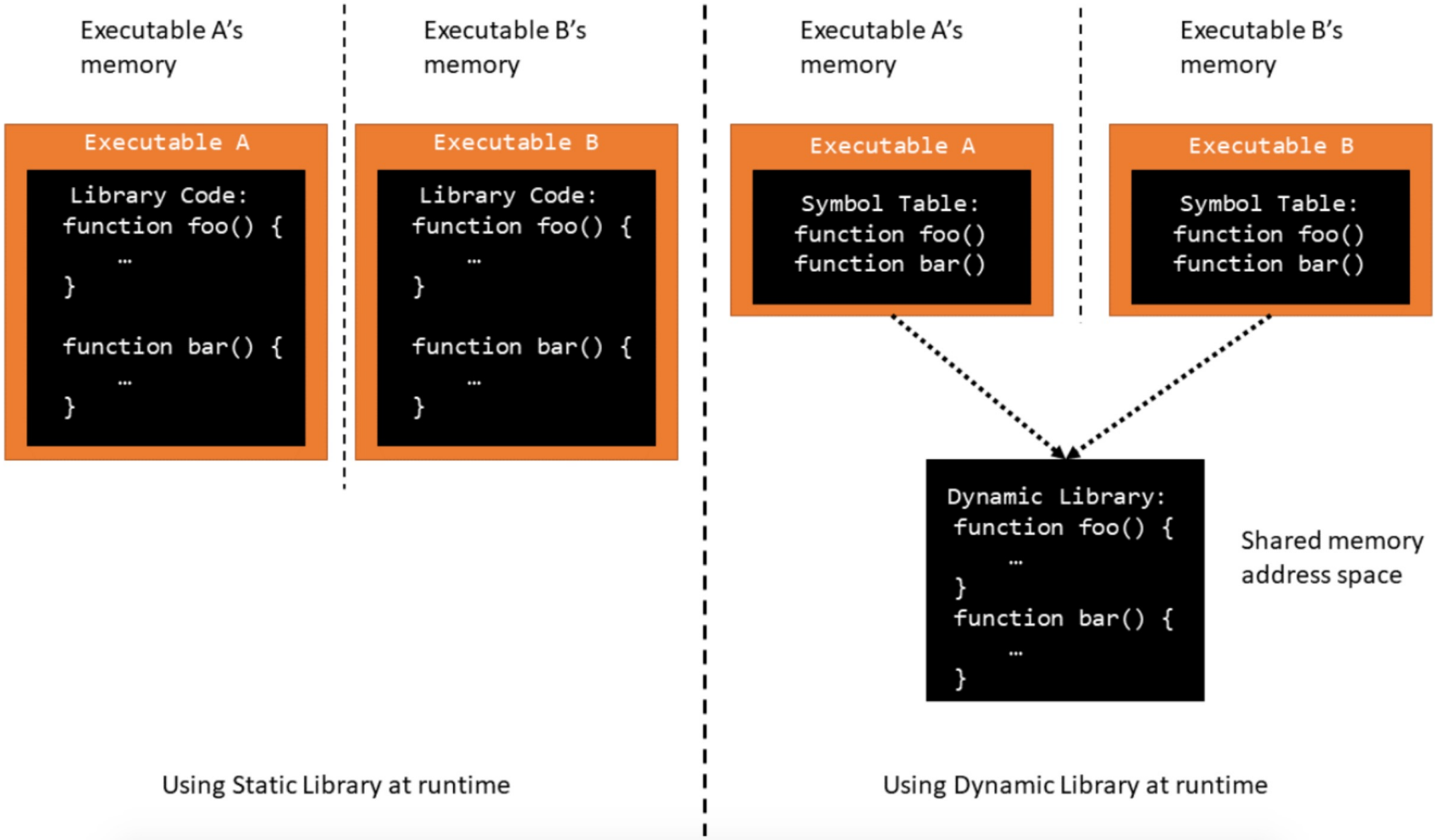
Library Types in C++

--- compile time



Library Types in C++

--- run time



Cmake

2. Cmake with libraries

- Demo: main.cpp with hello library

- Declare a new library
  - Library name : say-hello
  - Source files: hello.hpp, hello.cpp
  - Can add library type: STATIC (default), SHARED

- Tell cmake to link the library to the executable(output)
  - Private link
  - Public link
  - interface

cmakelists.txt

```
cmake_minimum_required(VERSION 3.12)
project(MyProject VERSION 1.0.0)

add_library{
        say-hello          [library type](optional)
        hello.hpp
        hello.cpp
}


add_executable(output main.cpp)

target_link_libraries(output PRIVATE say-hello)
```

Cmake

2. Cmake with libraries

demo

- Demo: main.cpp with hello library

- Declare a new library
  - Library name : say-hello
  - Source files: hello.hpp, hello.cpp
  - Can add library type: STATIC (default), SHARED

- Tell cmake to link the library to the executable(output)
  - Private link
  - Public link
  - interface

cmakelists.txt

```
cmake_minimum_required(VERSION 3.12)
project(MyProject VERSION 1.0.0)

add_library{
        say-hello          [library type](optional)
        hello.hpp
        hello.cpp
}


add_executable(output main.cpp)

target_link_libraries(output PRIVATE say-hello)
```

Cmake

--- Target_link_libraries/Target_include_directories

- target_link_libraries(<target>

  <PRIVATE|PUBLIC|INTERFACE> <lib> ...])

- The PUBLIC, PRIVATE and INTERFACE keywords can be used to specify both the link dependencies and the link interface in one command.
  - PUBLIC: Libraries and targets following PUBLIC are linked to, and are made part of the link interface.
  - PRIVATE: Libraries and targets following PRIVATE are linked to, but are not made part of the link interface.
  - INTERFACE: Libraries following INTERFACE are appended to the link interface and are not used for linking <target>.

Cmake

3. Cmake with Flags

- C++ standard (equivalent to -std=c++2a)

  CMAKE_CXX_STANDARD

cmakelists.txt

```
cmake_minimum_required(VERSION 3.12)
project(MyProject VERSION 1.0.0)

set(CMAKE_CXX_STANDARD 20)

set(CMAKE_BUILD_TYPE Release)
if(CMAKE_BUILD_TYPE STREQUAL "Release")

   set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} –O3")
   set(CMAKE_C_FLAGS_RELEASE "${CMAKE_C_FLAGS_RELEASE} –O3")

endif()

add_executable(output main.cpp)
```

Cmake

3. Cmake with Flags

demo

- Build Type

```
set(CMAKE_BUILD_TYPE Release)
set(CMAKE_BUILD_TYPE Debug) // gdb
```

- Optimization level

```
set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} -O1")
set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} -O3")
```

cmakelists.txt

```
cmake_minimum_required(VERSION 3.12)
project(MyProject VERSION 1.0.0)

set(CMAKE_CXX_STANDARD 20)

set(CMAKE_BUILD_TYPE Release)
if(CMAKE_BUILD_TYPE STREQUAL "Release")

  set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} –O3")
  set(CMAKE_C_FLAGS_RELEASE "${CMAKE_C_FLAGS_RELEASE} –O3")

endif()

add_executable(output main.cpp)
```

Performance Optimization

- 5 steps to improve runtime efficiency
- Time study
- How to use gprof
- Demo

Improve Execution Time Efficiency

1. Do timing studies
2. Identify hot spots
3. Use a better algorithm or data structure
4. Enable compiler speed optimization
5. Tune the code

Time the program

--- Unix ‘time’ command

- Run $ time ./output

```
real    0m12.977s
user    0m12.860s
sys     0m0.010s
```

- Real: Wall-clock time between program invocation and termination
- User: CPU time spent executing the program
- System: CPU time spent within the OS on the program’s behalf

Identify hot spots

- Gather statistics about your program's execution
- Runtime profiler: gprof (GNU Performance Profiler)
- How does gprof work?
    - By randomly sampling the code as it runs, gprof check what line is running, and what function it's in

Gprof

- Compile the code with flag –pg
  - g++ –pg helloworld.cpp –o output
- Run the program
  - $ ./output
  - Running the application produce a profiling result called gmon.out
- Create the report file
  - gprof output > myreport
- Read the report
  - vim myreport

Gprof by CMake

- Compile the code with flag –pg set in CMakeLists

- Run the program
  - $ ./output
- Create the report file
  - gprof output > myreport
- Read the report
  - vim myreport

cmakelists.txt

```
cmake_minimum_required(VERSION 3.12)
project(MyProject VERSION 1.0.0)

# Enable gprof profiling
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -pg")
set(CMAKE_EXE_LINKER_FLAGS
"${CMAKE_EXE_LINKER_FLAGS} -pg")

add_executable(output main.cpp)
```

Flat Profile

```
Each sample counts as 0.01 seconds.
  %      cumulative   self              self     total
 time     seconds   seconds  calls  us/call  us/call  name
 13.22      0.28     0.28  50045000     0.01     0.01  void std::__cxx11::basic_string<char, std::char_traits<char>, …
 10.39      0.50     0.22  100000000    0.00     0.00  std::vector<Entity, std::allocator<Entity> >::operator[](unsigned long)
 6.85       0.65     0.15  50005000     0.00     0.00  __gnu_cxx::__normal_iterator<Entity const*,std::vector<Entity,…
 5.67       0.77     0.12  100030000    0.00     0.00  __gnu_cxx::__normal_iterator<Entity const*, std::vector<Entity, …
 5.67       0.89     0.12  50045000     0.00     0.01  std::iterator_traits<char*>::difference_type std::distance<char*>(char*,…
 5.43       1.00     0.12  50005000     0.00     0.00  __gnu_cxx::__normal_iterator<Entity const*,std::vector<Entity, …
…
…
```

- name: name of the function
- %time: percentage of time spent executing this function
- cumulative seconds: This is the cumulative total number of seconds the computer spent executing this functions, plus the time spent in all the functions above this one in this table.
- self seconds: time spent executing this function
- calls: number of times function was called (excluding recursive)
- self s/call: average time per execution (excluding descendents)
- total s/call: average time per execution (including descendents)

Improve Execution Time Efficiency

demo

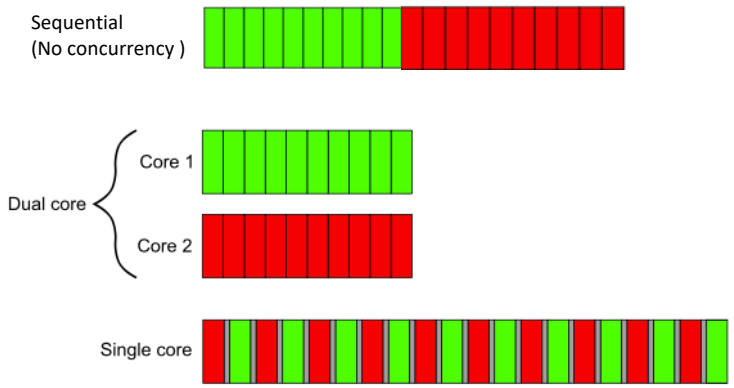
1. Do timing studies
2. Identify hot spots
3. Use a better algorithm or data structure
4. Enable compiler speed optimization. (compile flag with -O3)
5. Tune the code

Multithreading

- What is concurrency
- Multithreading
- Threads Management

Concurrency

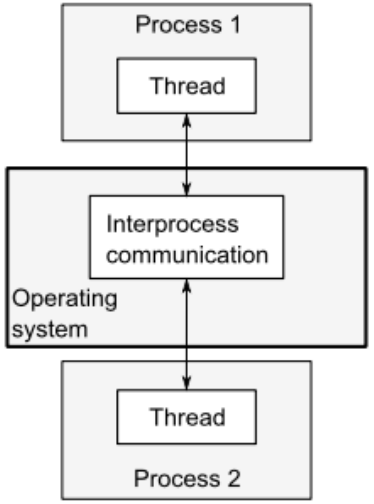
- What is concurrency?
  - a single system performs multiple independent activities in parallel



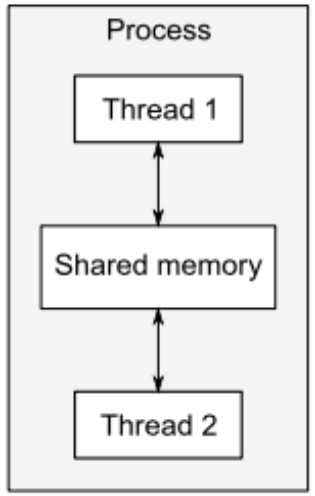
- Why use concurrency?
  - Separation of concerns
  - Performance

Concurrency

- Types of concurrency:
  - Concurrent Processes
  - Concurrent Threads

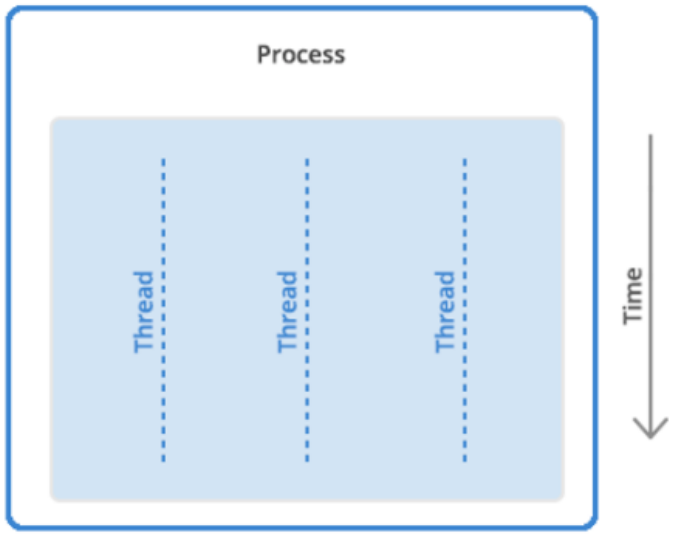


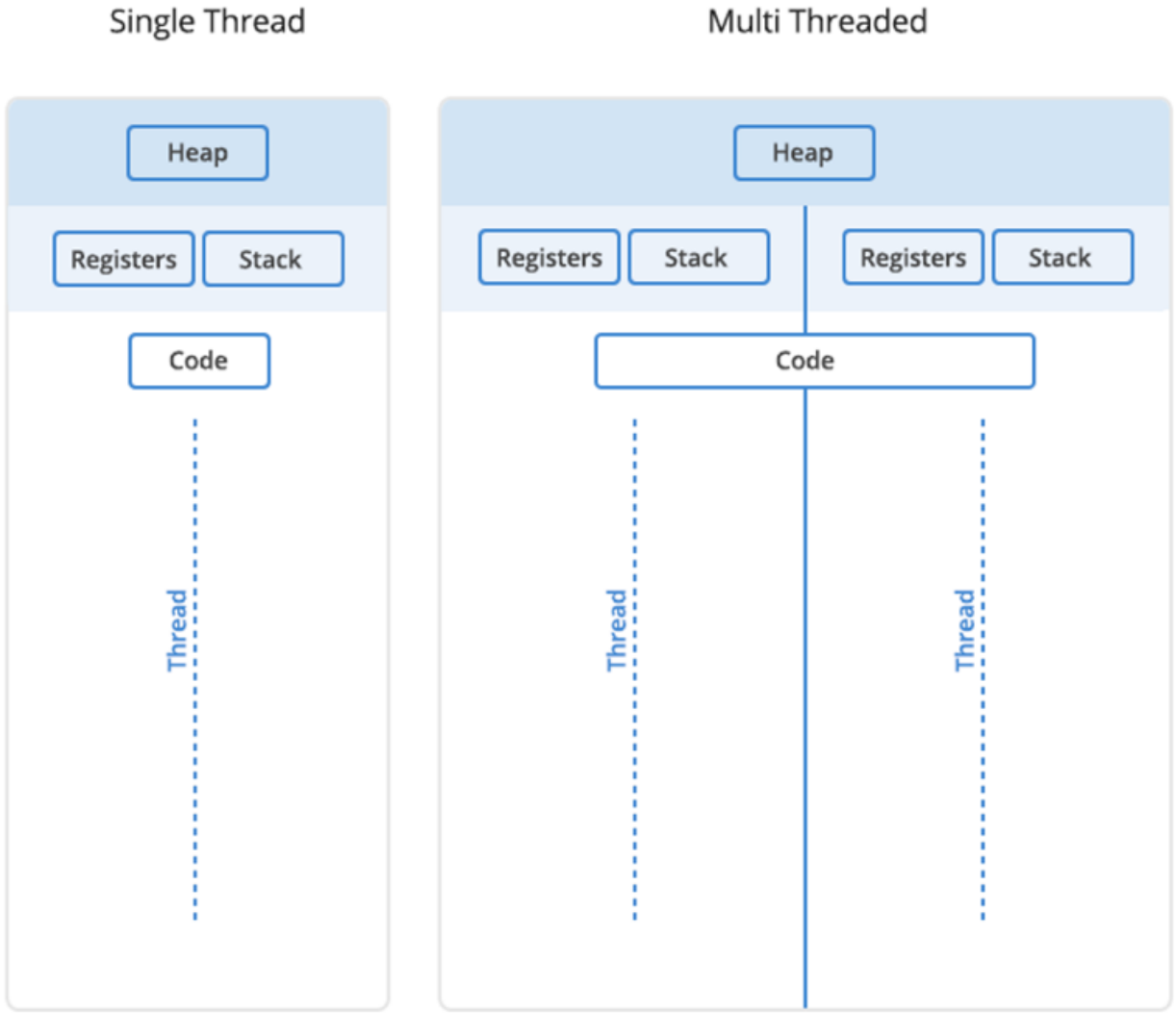
Multiple Processes



Multiple Threads

Concurrency





https://www.backblaze.com/blog/whats-the-diff-programs-processes-and-threads/

Multithreading

- Threads:
  - Threads are lightweight executions: each thread runs independently of the others and may run a different sequence of instructions.
  - All threads in a process share the same address space, and most of the data can be accessed directly from all threads—global variables remain global, and pointers or references to objects or data can be passed around among threads.
- Example:

```
#include <iostream>
#include <thread>

void hello() {
    std::cout<<"Hello Concurrent World\n";
}

int main() {
    std::thread t(hello);
    t.join();
}
```

Compile with –lpthread flag

Multithreading

--- managing thread

- Launching a thread (std::thread)
  - Create a new thread object.
  - Pass the executing code to be called (i.e, a callable object) into the constructor of the thread object.
  - Once the object is created a new thread is launched, it will execute the code specified in callable.
- A callable types:
  - A function pointer
  - A function object
  - A lambda expression

Multithreading

--- managing thread

- Launching a thread (std::thread)
  - Create a new thread object.
  - Pass the executing code to be called (i.e, a callable object) into the constructor of the thread object.
  - Once the object is created a new thread is launched, it will execute the code specified in callable.
- A callable types:
  - **A function pointer**
  - A function object
  - A lambda expression

Multithreading

--- Launching thread with function pointer

- Launching a thread using function pointers and function parameters

```
void func(params)
{
   // Do something
}

std::thread thread_obj(func, args);
```

- Example1: function takes one argument

```
#include <thread>

void hello(std::string to)
{
     std::cout << "Hello Concurrent World to " << to << "\n";
}
int main()
{
     std::thread t1(hello, "alicia");
     std::thread t2(hello, ”ricky");
     t1.join();
     t2.join();
}
```

Multithreading

--- Launching thread with function pointer

- Launching a thread using function pointers and function parameters

```
void func(params)
{
   // Do something
}

std::thread thread_obj(func, params);
```

- Example2: function takes multiple arguments (passing by values, and passing by reference)
  - std::ref for reference arguments

```
#include <thread>

void hello_count(std::string to, int &x){
     x++;
     std::cout << "Hello to " << to << x << std::endl;
}

int main(){
     int x = 0;
     std::thread threadObj(hello_count, "alicia", std::ref(x));
     threadObj.join();
     std::cout << "After thread x=" << x << std::endl;
}
```

Calling function of class on an object in a new thread

- First: How does calling a function on a class object work in C++?
- Suppose I have a class with an attribute x, a function print() that prints x.
- All objects of the class have their own copy of the non-static data members, but they share the class functions.
- When I call print() on different objects, why is there behavior different?

```
Class myClass{
public:
    int x;
    void print(){
        std::cout << x << std::endl;
    }
};
```

```
int main(){
    myClass obj;
    obj.print();
}
```

Calling function of class on an object in a new thread

Solution to the puzzle:

- All class functions automatically receive a pointer to the class object as their first argument
- For example, myClass::print() behaves as if it's written as myClass::print(myClass* obj_ptr)
- All references to x in the function resolve as obj_ptr->x

```
Class myClass{
public:
    int x;
    void print(){
        std::cout << x << std::endl;
    }
};
```

```
int main(){
    myClass obj;
    obj.print();
}
```

Multithreading

--- Launching thread with member function

- Launching a thread using member function

```
class FunClass {
   void func()(params) {
      // Do Something
   }
};
FunClass x;
std::thread thread_object(&FunClass::func, &x, params);
```

- Example3: launching thread with member function

```
class Hello
{
public:
     void greeting(std::string const &message) const{
          std::cout << message << std::endl;
     }
};

int main(){
     Hello x;
     std::thread t(&Hello::greeting, &x, "hello");
     t.join();
}
```

Multithreading

--- managing thread

- Launching a thread (std::thread)
  - Create a new thread object.
  - Pass the executing code to be called (i.e, a callable object) into the constructor of the thread object.
  - Once the object is created a new thread is launched, it will execute the code specified in callable.

- A callable types:
  - A function pointer
  - **A function object**
  - A lambda expression

Multithreading

--- Launching thread with function object

- Launching a thread using function object and taking function parameters

```
class fn_object_class {
    // Overload () operator
   void operator()(params) {
      // Do Something
   }
}
std::thread thread_object(fn_object_class(), params)
```

- Example: launching thread with function object
  - Create a callable object using the constructor
  - The thread calls the function call operator on the object

```
#include <thread>
#include <string>

class Hello{
public:
      void operator()(std::string name)
      {
            std::cout << "Hello to " << name << std::endl;
      }
};

int main(){
      std::thread t(Hello(), "alicia");
      t.join();
}
```

Multithreading

--- managing thread

- Launching a thread (std::thread)
  - Create a new thread object.
  - Pass the executing code to be called (i.e, a callable object) into the constructor of the thread object.
  - Once the object is created a new thread is launched, it will execute the code specified in callable.
- A callable types:
  - A function pointer
  - A function object
  - **A lambda expression**

Multithreading

--- Launching thread with lambda function

- Launching a thread using lambda function

```
std::thread thread_object([](params) {
    // Do Something
}, params);
```

- Example1:

basic lambda function

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

int main()
{
    std::thread t([](std::string name){
        std::cout << "Hello World ! " << name <<" \n";
    }, “Alicia”);
    t.join();
}
```

Lambda function

- Lambda expression

```
[ capture clause ] (parameters) -> return-type
{
  definition of method
}
```

- Capture variables:
  - [&] : capture all external variables by reference
  - [=] : capture all external variables by value
  - [a, &b] : capture a by value and b by reference

```
int main()
{
    std::vector<int> v1 = {3, 1, 7, 9};
    std::vector<int> v2 = {10, 2, 7, 16, 9};
    //  access v1 and v2 by reference
    auto pushinto = [&] (int m){
        v1.push_back(m);
        v2.push_back(m);
    };
    pushinto(100);
 …
}
```

& can access all the variables that are in scope.

Multithreading

--- managing threads

- Joining threads with std::thread
  - Wait for a thread to complete
  - Ensure that the thread was finished before the function was exited and thus before the local variables were destroyed.
  - Clean up any storage associated with the thread, so the std::thread object is no longer associated with the now- finished thread
  - join() can be called only once for a given thread

```
std::thread thread_obj(func, params);
Thread_obj.join();
```

demo

Code source:
https://github.com/aliciayuting/CS4414Demo.git

Multithreading

--- managing threads

- Detach threads with std::thread
  - Run thread in the background, with no direct means of communicating with it. Ownership and control are passed over to the C++ Runtime Library
  - Detached threads are also called daemon / Background threads.
  - Such threads are typically long-running; they may well run for almost the entire lifetime of the application, performing a background task
  - If neither join or detach is called with a std::thread object that has associated executing thread then during that object's destruct, it will terminate the program.

```
std::thread thread_obj(func, params);
thread_obj.detach();
```

demo

Code source:
https://github.com/aliciayuting/CS4414Demo.git

Recap Multithreading

- Launching a thread:
  - Function pointer
  - Function object
  - Lambda function
- Managing threads
  - Join()
  - Detach()

Where to find the resources?

- Concurrency programing:
  - Book: C++Concurrency in Action Practice Multithreading
- Multithreading and mutex:
  - https://www.geeksforgeeks.org/multithreading-in-cpp/
  - https://thispointer.com/c11-multithreading-part-2-joining-and-detaching-threads/
  - https://www.youtube.com/watch?v=q6dVKMgeEkk [helpful tutorial to understand RAII]
- Notes:
  - https://thispointer.com/c11-multithreading-part-3-carefully-pass-arguments-to-threads/