Operation system assigntment 4

Valgrind, Graph Data Structure, and Euler

Submitted by: Wasim shebalny : 323944280

Shifaa khatib : 324095702

Question 1-3: run $ make and then run

$ ./graph -e <num_edges> -v <num_vertices> -s <seed>

Replace <num_edges>, <num_vertices>, and <seed> with your specific values.

Question 4: Code Coverage and Profiling Reports, valgrind and Callgraph Report Generation.

-coverage:

Run $ make coverage , After running make coverage, the following line will be executed : ./graph -e 1000 -v 100 -s 42

After executing the program, the coverage data will be collected, and a coverage report will be generated.

The coverage tool used is lcov, which produces an HTML report.

To open the HTML coverage report, use the following command:

firefox coverage/index.html

LCOV - code coverage report

Current view:	top level		Coverage	Total	Hit
Test:	coverage.info	**Lines:**	80.3 %	843	677
Test Date:	2024-10-08 15:58:50	**Functions:**	85.1 %	275	234

Directory	Line Coverage			Function Coverage		
	Rate	Total	Hit	Rate	Total	Hit
13	66.7 %	3	2	50.0 %	2	1
13/bits	78.4 %	721	565	84.6 %	259	219
13/ext	100.0 %	9	9	100.0 %	4	4
/home/shifaa/Downloads/Operating_System_HW4-main/OS_EX4	91.8 %	110	101	100.0 %	10	10

Generated by: LCOV version 2.0-1

-profiling:

Run $ make profile , After running make coverage, the following line will be executed : ./graph -e 1000 -v 100 -s 42

And It will output gprof_report.txt.

Valgrind:

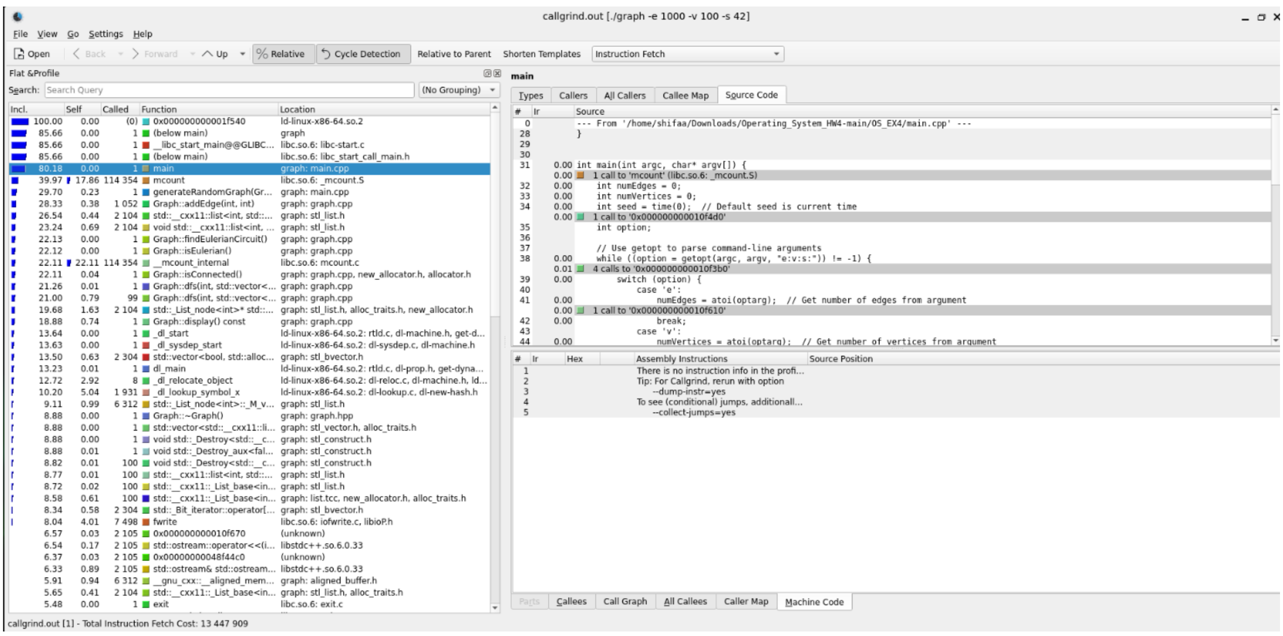
Run $ make valgrind , After running make coverage, the following line will be executed : ./graph -e 1000 -v 100 -s 42

And It will output valgrind_report.txt.

Callgraph:

we generate a **call graph** to visualize function calls within the program. This helps to understand the program flow and identify performance bottlenecks.

Run $ make callgraph it will output callgrind.out report , also:



This picture shows a **Call Graph report** generated by Valgrind's **Callgrind** tool and visualized using **KCacheGrind**. It displays how much processing time each function in the program took during execution. Key functions, such as main, are listed alongside their **inclusive cost** (time spent in the function and functions it calls) and **self cost** (time spent within the function itself). The source code on the right shows how often each line of code was executed, allowing for detailed performance analysis and identifying bottlenecks in the code execution.

Question 5: detect and report memory leaks using Valgrind on the hello.c program.

Run $ make valgrind_hello And It will output valgrind_hello_report.txt.

Question 6: demonstrate Valgrind attached to a debugger (such as gdb), you can follow these steps: Steps to Attach Valgrind to GDB:

Open two terminals:

<u>**Left terminal**</u> **:** enter this commands:

1) Compile the program**:**

$ gcc -g -o hello hello.c

2) Run Valgrind with GDB server:

$ valgrind --vgdb=yes --vgdb-error=0 ./hello

--vgdb=yes tells Valgrind to run with the gdb server.

--vgdb-error=0 tells Valgrind to pause on the first error and wait for gdb to attach.

<u>**Second terminal**</u>: enter this commands:

1) Attach GDB to Valgrind: $ gdb ./hello

2) connect to Valgrind by: $ target remote | vgdb

Once gdb is connected to Valgrind, it will allow you to debug and investigate the issue.

Example commands:

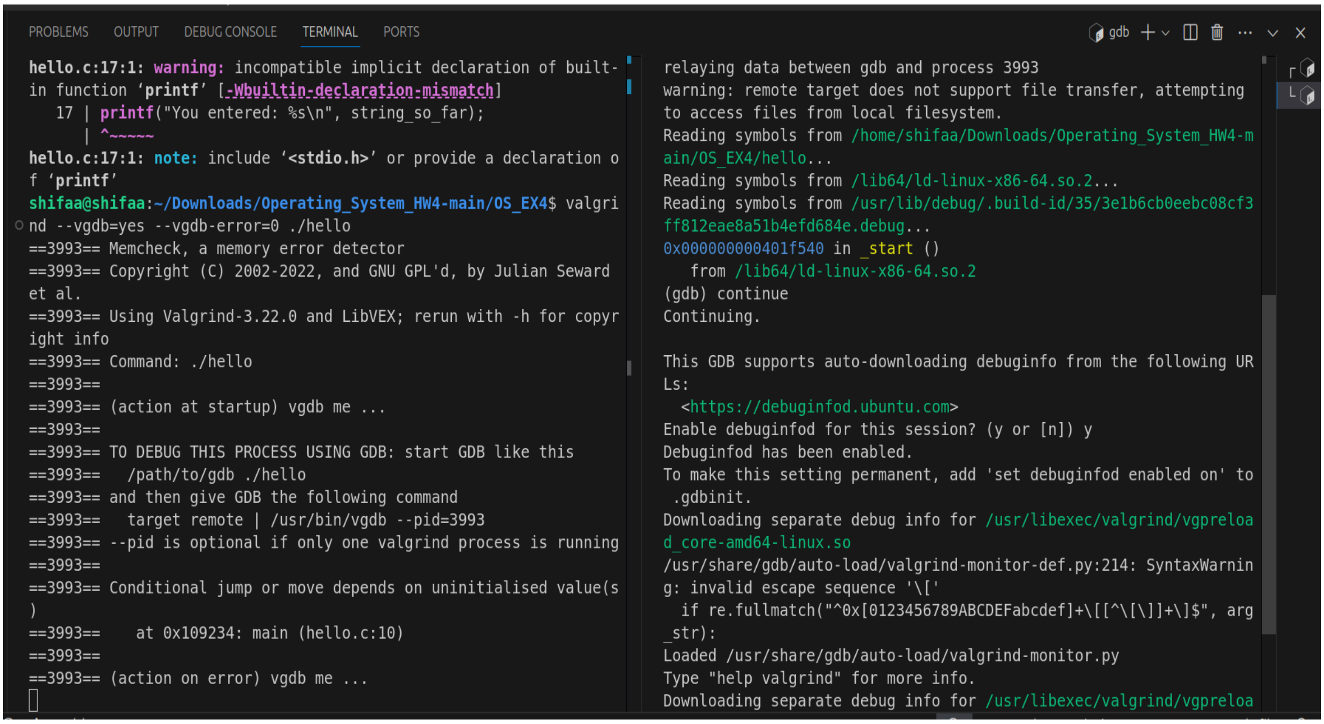
bt # Get a backtrace of the current call stack

info locals # Get information about local variables

step # Step into the current function

next # Move to the next line of code

continue # Continue running the program



On the left side,we are running the command valgrind --vgdb=yes --vgdb-error=0 ./hello to launch **Valgrind** and start the hello program. Valgrind is waiting for us to attach GDB for debugging.

On the right side, **GDB** is started and attached to the process using the instructions provided by Valgrind. we executed gdb ./hello and followed the instructions from Valgrind to attach **GDB** to the running process via target remote | /usr/bin/vgdb --pid=3993.

GDB has successfully connected to Valgrind and is ready to debug the program.

Question 7 : Detect race conditions using Valgrind/Helgrind.

Run $ make valgrind_race And It will output valgrind_race_report.txt.

Question 8 : Singleton Mutex and Guard Class Implementation

Files:

MutexBase.hpp:

- Defines an abstract base class for mutex operations (lock and unlock), which are implemented by the derived class.

MutexImpl.hpp:

- Implements the singleton pattern using pthread_mutex_t from POSIX. This file defines how the mutex is locked and unlocked, ensuring that only one instance of the mutex is used in the entire application.

Guard.hpp:

- A scope guard class that locks the mutex in its constructor and unlocks it in the destructor, ensuring the mutex is properly locked and unlocked within a specific scope.

main.cpp:

- The main file that tests and demonstrates the use of the Singleton Mutex and Guard classes. It simulates a multi-threaded environment where threads safely access shared resources.