

# Operation system assignment 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
./	<div><div></div></div> 66.7 %	3	2		<div><div></div></div> 50.0 %	2	1
./hits	<div><div></div></div> 78.4 %	721	565		<div><div></div></div> 84.6 %	259	219
./ext	<div><div></div></div> 100.0 %	9	9		<div><div></div></div> 100.0 %	4	4
./home/shifaa/Downloads/Operation_System_Het-main/OS_FK4	<div><div></div></div> 91.8 %	110	101		<div><div></div></div> 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:

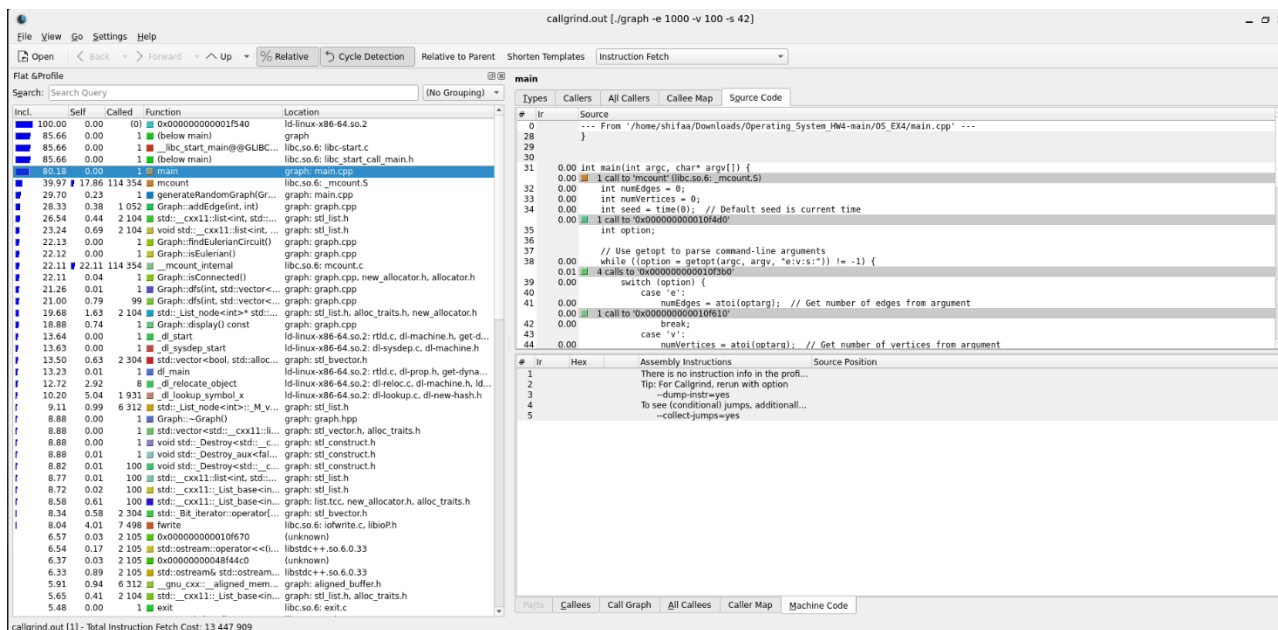
Run `$ make valgrind`, After running `make coverage`, the following line will be executed : `./graph -e 1000 -v 100 -s 42`

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:

**Left terminal** : 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.

**Second terminal**: 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:

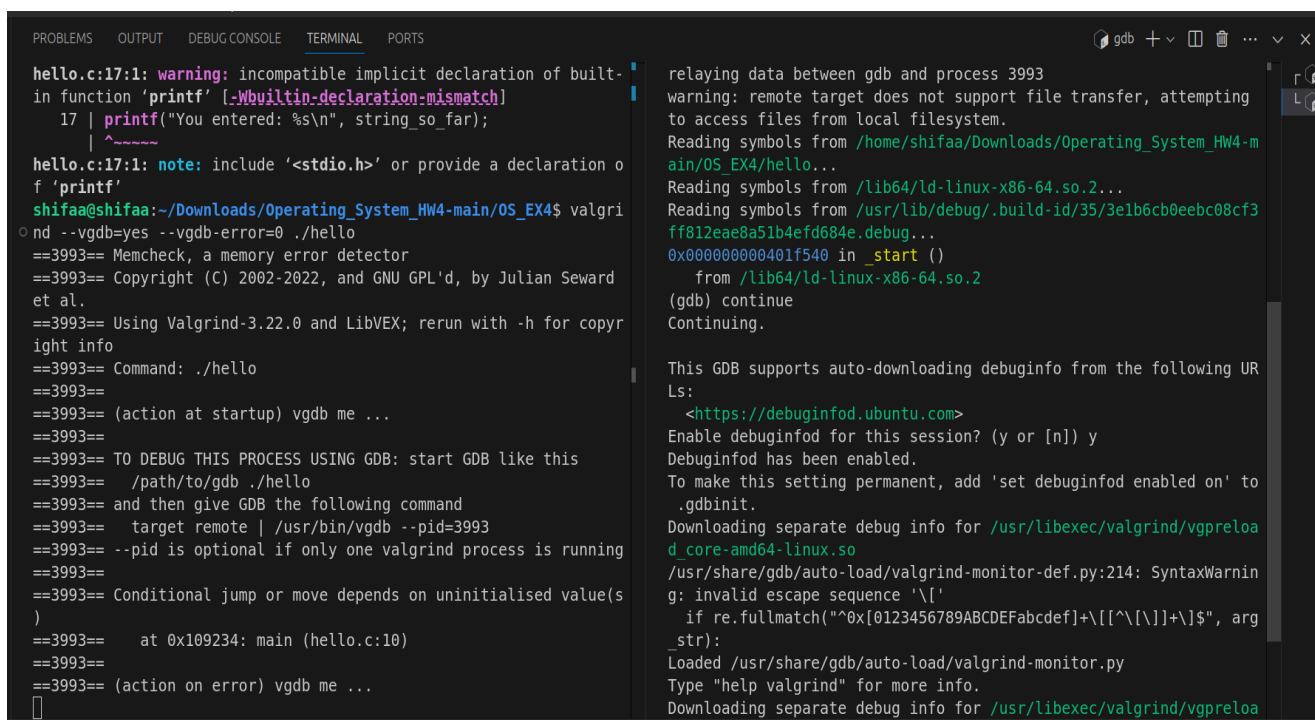
```
bt          # Get a backtrace of the current call stack
```

```
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
```



```
hello.c:17:1: warning: incompatible implicit declaration of built-in function 'printf' [-Wbuiltin-declaration-mismatch]
  17 | printf("You entered: %s\n", string_so_far);
      | ^~~~~~
hello.c:17:1: note: include '<stdio.h>' or provide a declaration of 'printf'
shifaa@shifaa:~/Downloads/Operating_System_HW4-main/OS_EX4$ valgrind --vgdb=yes --vgdb-error=0 ./hello
==3993== Memcheck, a memory error detector
==3993== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==3993== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==3993== Command: ./hello
==3993==
==3993== (action at startup) vgdb me ...
==3993==
==3993== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==3993== /path/to/gdb ./hello
==3993== and then give GDB the following command
==3993== target remote | /usr/bin/vgdb --pid=3993
==3993== --pid is optional if only one valgrind process is running
==3993==
==3993== Conditional jump or move depends on uninitialised value(s)
==3993==    at 0x109234: main (hello.c:10)
==3993==
==3993== (action on error) vgdb me ...
[ ]

relaying data between gdb and process 3993
warning: remote target does not support file transfer, attempting to access files from local filesystem.
Reading symbols from /home/shifaa/Downloads/Operating_System_HW4-main/OS_EX4/hello...
Reading symbols from /lib64/ld-linux-x86-64.so.2...
Reading symbols from /usr/lib/debug/.build-id/35/3e1b6cb0eebc08cf3ff812eae8a51b4efd684e.debug...
0x0000000000401f540 in _start ()
    from /lib64/ld-linux-x86-64.so.2
(gdb) continue
Continuing.

This GDB supports auto-downloading debuginfo from the following URLs:
<https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading separate debug info for /usr/libexec/valgrind/vgpreload_core-amd64-linux.so
/usr/share/gdb/auto-load/valgrind-monitor-def.py:214: SyntaxWarning: invalid escape sequence '\['
  if re.fullmatch("^0x[0123456789ABCDEFabcdef]+\[^[^\]]+\]$)", arg_str):
Loaded /usr/share/gdb/auto-load/valgrind-monitor.py
Type "help valgrind" for more info.
Downloading separate debug info for /usr/libexec/valgrind/vgpreload_core-amd64-linux.so
```

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.