GWDG
WG Computing
Jack Ogaja

Tutorial 1 / 03.04.2024
Practical Course High-Performance Computing / SoSe 2024
45 Minutes Total

# Debugging Using GDB and Valgrind

## Learning Objectives

The learning objectives in this tutorial are:

- To study and debug programs line-by-line and/or instruction-by-instruction
- To simulate programs' interaction with the machine's memory and identify bad and good memory access patterns.

## Tools

- GDB
- Valgrind

# Contents



In this tutorial we use GNU debugger (gdb) and Valgrind to study the state of the given programs during execution. The programs are simple line count and matrix initialization and bad memory allocation codes written in C. The matrix initialization programs do the initializations in rows and columns. The first *matrix_init_rw.c*, initializes each row, and the second, *matrix_init_cn.c* initializes each column. Due to the differences in the initializations, the programs will use the memory differently resulting to different run-times. The line count program *linecount.c* count the number of lines in a given file.

# Debugging with GDB 1: Tutorial (20 min)

Use GDB to introduce breakpoints and print the local variables for the functions being executed. Additionally use GDB to view the assembly code of the two given programs and print some of the registers used.

## Steps

1. Compile the two source codes using gcc with -g option using the prepared *Makefile*
   `$ make`

2. Launch GDB in the commandline
   `$ gdb`
3. Use GDB's help command to open manual in an interactive debugger session. e.g. Find and use different Text User Interface (TUI) layouts
   `(gdb) help`
4. Change the default GDB's assembly syntax to Intel (there are "att" and "intel" flavours)
   `(gdb) show disassembly-flavor`

   To switch the flavours use the commands
   `(gdb) set disassembly-flavor {intel, att}`
5. load the binary file or executable
   `(gdb) file executableFile`
6. Print the line numbers in the source code
   `(gdb) list <LINE_NUMBER>`
7. Run the program (with arguments *ARGS* )within the debugger context
   `(gdb) run <ARGS>`
8. disassemble the source code to see the assembly code
   `(gdb) disassemble <FUNCTION>`
9. Introduce and remove breakpoints at various positions and run the programs
   `(gdb) break <LINE_NUMBER>`
   `(gdb) delete <BREAKPOINT_NUMBER>`
10. Continue execution until the next break-point is reached
    `(gdb) continue`
11. Print the stack of the function executions
    `(gdb) bt | backtrace`
12. Examine the Stack Frames
    `(gdb) info frame | f`: Print a verbose description of the selected stack frame
    `(gdb) info args`: Print the arguments of the selected frame, each on a separate line.
    `(gdb) info locals`: Print the local variables of the selected frame, each on a separate line.

## Debugging memory issues with Valgrind 2: Tutorial (10 min)

Pointers are an extremely powerful tool, but it is easy to make mistakes and cause errors while using them. For example in the following program *pp1.c*, a pointer can be declared using the `*` symbol. It is possible to access the data at the pointer rather than the pointer itself by putting `*` before the name of the pointer variable. The address of the memory location to which a variable is stored (the pointer points to) can also be obtained by putting `&` before the variable.

```c
#include "stdio.h"
int main(void) {
    int x = 8;
    int *y = &x;
    printf("%p\n", y);
    printf("%d\n", *y);
```

Pointers can also be created using the `malloc()` function as shown in the following program, *pp2.c*.

```c
#include "stdio.h"
```

```c
int main(void) {
    int *y = malloc(sizeof(int));;
    *y = 8;
    printf("%p\n", y);
    printf("%d\n", *y);
```

Many tools exist to help programmers with resolving errors with pointers. One of these is valgrind. If you run valgrind on the above program, you'll see that it detects no errors.

```
$ valgrind ./pp2
==383== Memcheck, a memory error detector
==383== Copyright (C) 2002-2015, and GNU GPLd, by Julian Seward et al.
==383== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==383== Command: ./pp2
==383==
==383== error calling PR_SET_PTRACER, vgdb might block 0x5204040
8
==383== HEAP SUMMARY:
==383== in use at exit: 4 bytes in 1 blocks
==383== total heap usage: 2 allocs, 1 frees, 4,100 bytes allocated
==383==
==383== LEAK SUMMARY:
==383== definitely lost: 4 bytes in 1 blocks
==383== indirectly lost: 0 bytes in 0 blocks
==383== possibly lost: 0 bytes in 0 blocks
==383== still reachable: 0 bytes in 0 blocks
==383== suppressed: 0 bytes in 0 blocks
==383== Rerun with --leak-check=full to see details of leaked memory
==383==
==383== For counts of detected and suppressed errors, rerun with: -v
==383== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

As you can see from the above, a memory leak of 4 bytes was found (and cleaned up automatically when the program terminated). This is because we never called free on the memory we allocated with malloc. Try editing the code to correctly free the allocated memory and see if valgrind reports any memory leaks.

The provided program *badmem.c* contains instructions about different kinds of memory errors. Explore how these appear in valgrind by looking at the different errors presented and altering the code to resolve them.

### Steps

1. Open *badmem.c* and visually explore the source code and comments.
2. Run valgrind on the compiled code, and explore the errors it gives
3. Assess and compare cache misses in each program using Valgrind
4. Make alterations to the code to resolve the errors hilighted by valgrind, until you are satisfied you understand the types of error it might give.

## Assessing memory access patterns with Valgrind 3: Tutorial (10 min)

Analyse the memory patterns when the two given programs *matrix_init_rw.c* and *matrix_init_cn.c* are executed. Check whether cache is used effectively by the programs.

### Steps

1. Compile the two source codes using gcc with -g option

2. Run and time their executions using shell built-in `time` utility
3. Assess and compare cache misses in each program using Valgrind
4. Write results to output files "cache_misses_1.txt" and "cache_misses_2.txt"

## Optional: Assembly Instructions 4: Tutorial (5 min)

Using Intel Architecture Software Developer's Manual, find out the meaning of different instructions and registers shown in tutorial 1 above.

### Hints

- Create `$HOME/.gdbinit` file and add the following line, `set disassembly-flavor intel`
- Use Valgrind's `Cachegrind` tool to simulate the program's interaction with the machine's memory hierarchy.
- Write output files using the `--cachegrind-out-file` option.

### Further Reading

- I. Zhirkov (2017) Low-Level Programming, DOI 10.1007/978-1-4842-2403-8_11
- Intel Architecture Software Developer's Manual - https://www.intel.com/content/www/us/en/developer/articles/
sdm.html
- Valgrind User Manual - https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
- GDB User Manual - https://sourceware.org/gdb/current/onlinedocs/gdb/