HIGH PERFORMANCE PROGRAMMING
UPPSALA UNIVERSITY
SPRING 2022
LAB 4: DEBUGGING AND PROFILING

Profiling is a process of looking for slow parts of the program and optimizing them. The aim of this lab is to give hands-on experience of different profiling and debugging tools.

This lab includes the following parts:

(1) Using the GDB debugger

(2) Using valgrind memory checker

(3) Using performance analyzer gprof and operf

Note: `valgrind` is usually available or at least easy to install on Linux computers. It is available on the lab computers. If you're using your own Mac, it may happen that `gdb, valgrind, gprof` and/or `operf` is not available. It should be possible to install them, but it may be easier for you to use the Linux lab computers for the tasks involving `gdb, valgrind, gprof` and `operf`. Alternatively, you can use ssh to login to one of the university's Linux servers, see the list of available Linux hosts here: http://www.it.uu.se/datordrift/maskinpark/linux.

Start by unpacking the `Lab04_Debug.tar.gz` file which contains the files needed for the tasks in this lab.

1. The GDB debugger

Task 1:

The code for this task is in the `Task-1` directory.

In Lab 1 we saw how GDB can be used to detect where in your code a "segmentation fault" error happens. Now we will look at some other ways of using gdb.

We will use the program `littlecode.c` as our example here. Suppose that we are interested in the state of the program when a certain function is called. We can then use gdb to set a so-called *breakpoint* in that place in the code. Then gdb will run the code until it reaches the breakpoint, and then stop, giving us the opportunity of investigating e.g. what values different variables have at that point of our program's run.

To test this, first compile the program using the `-g` compiler option to include the debugging information that gdb needs:

```
gcc -g littlecode.c -o littlecode
```

Date: January 18, 2026.

Run the program to see what it does; it computes a number x that is printed. Now run gdb giving out program as input argument, like this:

```
gdb ./littlecode
```

Now we get a gdb prompt "`(gdb)`" where we can give different commands. Suppose we are interested in the program's behavior when the `hh()` function is called. Then we can set a breakpoint there, like this:

```
(gdb) break hh
```

Then tell gdb to start our program by giving the command "run". Now gdb will run our program until it reaches the breakpoint, and then it will stop and show the "`(gdb)`" prompt again. So now we are at the breakpoint. Try the "`bt`" (backtrace) command to show the current call stack; the functions that have been called to reach this point. Compare to the code in `littlecode.c`. Does the call stack look as you expect?

When we have stopped at a breakpoint we can also check the current values of different variables using the "`print`" command. Try this, for example to check the values of the `global1` and `global2` variables. Does it work? We can use the command "`c`" (continue) to continue execution of the program. In this case, since the `hh()` function is called in a loop, the program will again stop at the same breakpoint in the next loop iteration. Try doing "`c`" several times and look at the output this gives. You can see that it stops at the breakpoint in the `hh()` function, with different values of the `a` input argument. Do the values of `a` shown match your expectations, considering how the code in `littlecode.c` is written?

Another useful thing we can do with gdb is to find out why a program seems to hang. This is a common type of bugs that can be difficult to track down – if your program just appears to freeze it is hard to know what went wrong. In this situation gdb can help.

To see how gdb can help us when our program seems to hang, start by introducing a bug in the `littlecode.c` code: change the loop condition in the loop in the `gg()` function from `c<100` to `c<200`. Since `c` is of type `char` this means that the loop will never end. Now we will pretend as if we do not know that, we just have a program with a bug that causes it to seemingly freeze while it is running.

Run the program:

```
./littlecode
```

So now the program just seems to hang, it never finishes. We can see the process ID (PID) of the program if we run `top` in another terminal window. Do that, and note the PID of the `littlecode` process. Knowing the PID we can run gdb and attach it to the process that is already running. If the PID is 1234 then this is done as follows:

```
gdb -p 1234
```

Try that. What happens?

If you get an error message saying "Could not attach to process", see the alternative approach under "Alternative" below.

When attaching gdb to a running process like this, gdb will stop that program and let us check what the program was actually doing. In this case we will directly see that the program is currently inside the gg() function. We can use the "bt" command to see which other functions were called to reach this point, and we can print values of variables that we are interested in. Hopefully this will help us understand what had happened, in this case that we have an infinite loop due to the c<200 code change.

Alternative: On some systems, depending on security settings in Linux, gdb may not be allowed to attach to another process in the way described above. In that case, if you have a program that appears to hang and you want to stop it and find out where and why it hangs, you will have to start the program using gdb, then when it hangs enter <ctrl>C (where <ctrl> means the ctrl key on your keyboard) to interrupt the program. Then gdb will detect the interrupt and stop the program, and you will then see the gdb command prompt and be able to check what is going on in the same way as if you had attached gdb to a running process.

There is much more you can do with gdb – look at man gdb and/or type "help" within gdb to get more information. For example, you can look up the gdb commands "next" and "step", and try using them.

2. Valgrind memory checker

He we will be using valgrind — a suite of tools for debugging and profiling programs. It is an extremely useful tool that can be used in several ways; we will first see how it can detect memory errors, and then how it can be used to examine the amount of heap and stack memory used by our programs.

Task 2:

The code for this task is in the Task-2 directory.

In this task we look at how valgrind can be used to find memory errors in our programs.

In the Task-2 directory you find a small code that does some kind of small computation. Use the makefile to build it, and try running it.

Now, it we want to use valgrind on our program, we simply run valgrind and give it our program's executable name as input, like this:

```
valgrind ./prog
```

Then valgrind will run our program. We will get the regular output from our program, plus additional output from valgrind.

The default behavior of valgrind is that it is running its so-called "memcheck" tool, meaning that it is trying to check the program for memory-access-related errors. If it found no errors, valgrind should say "ERROR SUMMARY: 0 errors" on the last line of its output.

To see how the memcheck functionality works, now introduce a bug in the code: add a line of code where you try to access memory outside of what was allocated. For example, in fun1() you could add a line trying to modify something outside what was allocated for the vector tmp, e.g. "tmp[k]=0.2;". Such a bug is not detected by the compiler, and may lead to wrong results and/or crashing your code. Valgrind can help us detect such bugs.

Introduce such a bug, build the program again and then run it using valgrind. Does valgrind detect the error? Are you able to see from the valgrind error message on what line in the code the error appears? What happens if you run the program normally, without using valgrind?

When using valgrind it is important to compile the code using the `-g` compiler option, to include debugging information when compiling. To see why this is important, remove the `-g` option in the makefile, then do "make clean" and then "make" again. If you run it with valgrind now, do you still get information about on what line in the code the error appears? After testing that, put the `-g` flag back again.

You can also use valgrind to detect memory leaks. To see this, introduce a memory leak by removing one of the `free()` calls. Then run the program using valgrind, and look for the "LEAK SUMMARY" output. There you see that there was some memory leak, and it says "Rerun with –leak-check=full to see details of leaked memory". That means you can run it like this:

```
valgrind --leak-check=full ./prog
```

Then valgrind will tell you where the problematic allocation was made. Note that the allocation in itself is not necessarily a problem, the problem is just that it was never freed.

When there are no memory leaks, valgrind should give you the message "All heap blocks were freed – no leaks are possible". Make a habit of always checking your programs with valgrind, and verify that you get the "All heap blocks were freed" message!

Running a program with valgrind usually takes a lot longer time than running it in the normal way, because of all the checking and extra work that valgrind is doing. To see this, increase the amount of work done in the program so that it takes e.g. 1 second to run normally, and then check how long time it takes to run it using valgrind. Does it become 2 times slower? 5 times? more?

Task 3:

The code for this task is in the `Task-3` directory.

The program for this task is doing some operations with an array. The array is initialised with random integers and a result is printed out.

Try to compile and run the `ex1.c` program.

The program may seem to run without problems but it contains some memory errors. Code with memory errors can be tricky to debug since it can appear to work for small tests but if applied to large problems or if routines are called many times, memory errors will likely cause the program to crash or to become extremely slow. Your tasks is to find memory errors using `valgrind` and fix them.

Look at the output from valgrind. Every row has prefix like `==12345==`, where the number is an identifier of the working process. The memory errors are described after the line

```
==12345== Command: ./ex1
```

Here you should see lines similar to:

```
==12345== Invalid write of size 4
==12345==    at 0x400872: main (ex1.c:35)
==12345==  Address 0x520307c is 0 bytes after a block of size 60 alloc'd
==12345==    at 0x4C2DB8F: malloc (vg_replace_malloc.c:299)
==12345==    by 0x40081B: main (ex1.c:29)
```

This output indicates that there is no storage beyond the end of the array of 60 bytes, which means that program is trying to reach a location outside the bounds of the allocated memory. The `(ex1.c:35)` tells you the location in the code: line 35 in `ex1.c`. Valgrind also shows where the allocated memory buffer came from: memory was allocated using `malloc` in the main function (in `ex1.c` line 29).

Look also under LEAK SUMMARY.

If there is something definitely lost, that means the program has memory leaks.

Rerun with `--leak-check=full` to see details of leaked memory. If you fix all memory leaks you should see the following message:

```
All heap blocks were freed -- no leaks are possible
```

Next, uncomment the line with the call to `function66(N)` and again use valgrind to check for memory errors. Now you will see a few more lines for each error. This is because now there are errors inside some function calls and valgrind is showing the call stack which lead to each error, e.g. `main` called `function66` which in turn called `function77` where the error occurred.

To find more information about options for valgrind tools type `valgrind --help`

Valgrind can detect when your program tries to access memory outside what was allocated, giving "invalid read" or "invalid write" messages. However, this does not mean that valgrind can detect all errors where a program is going outside of array bounds. To see this, use the example code `ex2.c`. Look at the code to see what it does: it works with a struct type A that includes an array `arr` followed by a long int `x`.

Compile the program and run it with valgrind to check for errors. Then introduce a bug by changing from `i < 8` to `i < 9` in the loop where `a->arr` is set. This means that the program is going beyond the array size. Compile and run it with valgrind. Is valgrind able to detect this error? What happens with the `a->x` output value? Can you understand why?

Then introduce the same bug for `b`, changing from `i < 8` to `i < 9` in the loop where `b->arr` is set. Is valgrind able to detect that error? Can you understand why?

Task 4:

The code for this task is in the `Task-4` directory.

We will use the `valgrind` tool “`massif`” to examine heap and stack usage. By default, `massif` measures only heap memory, i.e. memory allocated with malloc and similar functions. This is because stack profiling takes even more time.

The output from `massif` is placed in a file named `massif.out.<pid>`, where `<pid>` is a number representing the process id of the process you have run. This output file is read with the program `ms_print`.

You will here again use the `-g` compiler option, telling the compiler to add debugging information in the generated object code. Such debugging information includes for example line numbers, making it possible for debugging and profiling tools to determine which line in the original source code corresponds to a particular place in the generated object code.

In this task, we return to the merge sort code that you worked with in a previous lab. In order to use `massif`, make sure to compile the code with `-g`, then execute (for example):

```
valgrind --tool=massif ./sort_test 20000
```

After that, use the `ls` command to check the filename of the output file from `massif` — there should be a file called something like `ms_print massif.out.1234` (where 1234 is just an example, it will be another number depending on the process id).

Examine the profiling results with the following command:

```
ms_print massif.out.1234
```

The above command may give many lines of output, perhaps so much that it does not fit in your terminal window, and it may be inconvenient to try to scroll up to see all the output. Then it can be helpful to redirect the output to a file, like this:

```
ms_print massif.out.1234 > tmp.txt
```

In Linux, you can redirect the output of any command in that way, using the “`>`” symbol. Having done that, you can then look at the contents of the `tmp.txt` file in any way you choose, for example you can open it using your favourite text editor. Try this: run `ms_print` with and without redirecting the output to a file, and verify that the file then contains the same text that is otherwise printed directly in the terminal window. Does it work?

About the `ms_print` output: First you see a graph of memory usage. Each column of the graph corresponds to one memory snapshot. Most snapshots are normal snapshots, denoted with a column of `:` symbols. Some snapshots will contain more detailed information and are denoted by a column of `@` symbols. One snapshot is recognized as the peak and is also detailed, denoted by `#`. This graph is followed by information for each snapshot:

- the snapshot number
- the time the snapshot was taken in instructions (default) or bytes (if `--time-unit=B` was specified)

- the total memory consumption at that point
- the number of useful (asked-for) heap bytes allocated
- the number of extra heap bytes (for administration and alignment)
- and the size of the stack (if `--stacks=yes` was specified to valgrind).

When the printout reaches a detailed snapshot, an allocation tree is printed. The first line shows the allocation function used to create the heap allocation (e.g. malloc). What follows is a call tree showing on which lines in the code these malloc calls were made, which gives you a complete picture of how and why all heap memory was allocated at the time of the snapshot.

When using `--time-unit=B` the "time" is given in terms of the total number of bytes that has so far been allocated/deallocated on the heap and/or stack, so it is not really a time unit but it can be useful since it gives the total sum of all allocations. You can read more about the different massif options by doing `man valgrind` and looking for "MASSIF OPTIONS" there.

Run `massif` with `--time-unit=B`. Use the graph to determine the peak heap usage and the total sum of the allocations.

Change `main.c` to use `bubble_sort()` instead of `merge_sort()`. How does this affect the memory footprint?

Reverse the change and go back to using `merge_sort()`. Replace the two `malloc` calls by allocating a single buffer and setting `list1` and `list2` by pointing into that buffer. This reduces the number of calls to malloc/free to half. Do this optimization and run the program again. Is the memory footprint affected? How is the output of `massif` different?

An alternative approach to get rid of the memory allocations in our merge sort implementation would be to place the `list1` and `list2` buffers on the stack instead of calling `malloc`. Do this change and check how it works. Use `--stacks=yes` to include stack usage in the profile.

Finally, you can cut the recursion in `merge_sort` short by calling `bubble_sort` when the list of elements is smaller than some number n. Perform this optimization and experiment with different values of n, and check how this affects the output from `massif`.

Task 5:

The code for this task is in the `Task-5` directory.

The program matmul.c multiplies two dense square matrices. Three versions of the loop ordering (ijk, kij, jik) for multiplication are implemented.

Here we will use two of `valgrind`'s tools for profiling: `callgrind` and `cachegrind`.

Valgrind cachegrind simulates a CPU with a 2-level cache. For modern machines it simulates the first and the last level (LL) caches (if cachegrind can detect the cache configuration). Usually the LL cache is the L2 or L3 cache. Valgrind cachegrind gives detailed cache and branch profiling.

Compile matmul.c using the -g flag.

Run valgrind on the executable using the --tool=cachegrind, --branch-sim=yes and --cache-sim=yes options.

Set matrix size to 200.

You will see a table with data collected by valgrind during execution of the program. In the first lines cachegrind reports information about instruction caches. In the second part are presented counters for cache accesses for data (rd = reads, wr = writes). At the end one can see combined information for LL cache and branch prediction statistics.

Cachegrind also writes a file, cachegrind.out.<pid> (where <pid> is the ID of the process), in the current directory. This file contains all the data.

The program cg_annotate can produce a detailed presentation of this data. Run e.g. "cg_annotate cachegrind.out.1234" (replace 1234 with the process ID in your case). The program cg_merge can be used to combine data from different cachegrind files.

In the output from cg_annotate, in the first three rows, the cache configuration is presented. It looks something like this:

```
I1 cache:         32768 B, 64 B, 8-way associative
D1 cache:         32768 B, 64 B, 8-way associative
LL cache:         6291456 B, 64 B, 12-way associative
```

In the above example, "I1 cache" is the instruction cache and "D1 cache" is the L1 data cache. So the size of the L1 data cache is in the above example 32768 bytes, and the "64 B" means that the cache line size is 64 bytes. If you want, you can compare the cache sizes listed by cg_annotate to the cache sizes given by the lscpu command; you should normally see the same cache sizes there. In that way you can also see if the "LL cache" corresponds to the L2 or L3 cache on the computer you are using.

After that comes different counters, both for the entire program and separately for each function. The meaning of all the counters can be found in table 2.1.

Ir	nr. of instructions executed	Dr	nr. of memory reads
I1mr	L1 instruction cache read misses	D1mr	L1 data cache read misses
ILmr	LL cache instruction read misses	DLmr	LL cache data read misses
Dw	nr. of memory writes	Bc	conditional branches executed
D1mw	L1 data cache write misses	Bcm	conditional branches mispredicted
DLmw	LL data cache write misses	Bi	indirect branches execute
		Bim	indirect branches mispredicted

TABLE 2.1. Cachegrind counters and their meaning.

Use cg_annotate to look at data cache read misses for the mul_xyz functions. According to this information, which version of the matrix multiplication algorithm is the most cache efficient regarding the L1 data cache (D1mr and D1mw numbers) for the matrix size yo uwere testing now?

Look at the source code in `matmul.c` to see how the three different matrix-matrix multiplication variants are implemented. From looking a the code, can you understand the D1mr and D1mw numbers given by cachegrind?

Note that for efficient cache usage it is in general best to have "stride 1" memory access in the inner loop, as in the `mul_kij` case in `matmul.c` (here, "stride 1" means moving by one step in memory, thereby directly using all the contents of each cache line). However, depending on the size of the cache and the matrix size, the cache usage can sometimes be OK even if we do not have "stride 1" memory access in the inner loop. Look at the `mul_ijk` case — there the inner loop variable is `k` so the memory access `b[k][j]` does not look good. However, if `n` is small, it may happen that the `b[k][j]` value is anyway in cache because it was loaded into the cache line earlier, for the previous `j`. Based on that, we would expect the `mul_ijk` code to have OK L1 cache usage for small matrices, but to become drastically worse when the matrix becomes too large so that the `b[k][j]` value no longer remains in cache. Based on the size of the L1 data cache and the size of each cache line, can you predict at what matrix size this should happen? Experiment with some different matrix sizes and try to see this happening. At approximately what matrix size does the `mul_ijk` code run into trouble with the L1 cache usage? How much worse does it become compared to the `mul_kij` variant?

The `callgrind` tool extends the functionality of Cachegrind by recording the function call history.

Run `valgrind --tool=callgrind ./a.out`

Use the program `callgrind_annotate` to post-process the file generated by `callgrind`. You can read about the differences between Cachegrind and Callgrind here:

http://valgrind.org/docs/manual/cl-manual.html

3. Performance analyzer

Task 6:

The code for this task is in the `Task-6` directory.

This task is about profiling using `gprof`.

Note about `gprof` and compiler optimization flags: Unfortunately, when some more advanced compiler optimization are turned on that often leads to less information from `gprof`. For this reason, first do this task using only the `-O1` compiler optimization flag, to see what information `gprof` gives then. Then you can try also with other optimization flags.

For a description of `gprof` and its options type

```
man gprof
```

In order to profile with `gprof` add the flag `-pg` when compiling:

```
gcc -O1 -g -pg matmul.c
```

Run the program for matrix with size 900

```
./a.out
```

Then list all the files in the current directory using `ls -l`:

```
ls -l
```

Note the new file gmon.out where profiling data has been created. Note that the profiler shows results corresponding to the actual run of the program. If in this run some function was not called, then this function will not be included in the analysis. To include all the functions, even those that were never called, use the `-z` option.

Now run the profiler and examine the output:

```
gprof a.out gmon.out
```

The output information is divided into two parts. The *flat profile* shows how much time your program spent in each function, and how many times that function was called. The *call graph* shows relations between functions. Which functions called the current function, which other functions it called, and how many times. There is also an estimate of how much time was spent in the subroutines of each function.

Rerun the program and compare results of gprof for different matrix sizes.

- What are the most time consuming operations?
- Which loop ordering is more efficient?
- What is the most called function?
- Does the profiler output time include time while waiting for the user input?
- Which metric is used to order function in the call graph?
- What are the self and children columns representing?

Note that the `-pg` compiler option that is needed for `gprof` profiling means that extra instrumentation code is added in the program, and the performance may be affected by this; your program will probably be slower when compiled with the `-pg` option.

Task 7:

On the Scientific Linux machines we another performance analyzer installed *Operformance analyzer*. Here you can see timings on each function but also on code line level. Log in to `vitsippa.it.uu.se`.

Compile the program `matmul.c` with `gcc -g -o matmul matmul.c`.

Run the performance analyzer with `operf ./matmul`.

Analyze code with `opannotate --source`.

This should give a detailed report of timings on line level. If you want to see timings on function level, do `opreport --callgraph`.