HIGH PERFORMANCE PROGRAMMING
UPPSALA UNIVERSITY
SPRING 2018
LAB 7: MEMORY SPACE & COMPILER OPTIMIZATION

In this lab, we are going to work on conserving the memory footprint of programs and examine compiler optimization in a bit more detail.

The amount of memory needed by a program is often referred to as the *memory footprint* of that program. A large memory requirement is problematic in several ways; it may lead to performance problems and/or take away resources that other programs and processes may need, and may even prevent the program from running at all. Therefore it is generally good to try to keep the memory footprint as small as possible.

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.

In this lab we will also look at the size (memory space used) for structs and union datatypes in C, and what can be done to reduce the size.

The lab also includes a part about compiler optimization options, and a part about the effects of using uninitialized memory in C programs.

1. Main Tasks

Task 0

The code for this task is in the `Task-0` directory.

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

In the `Task-0` 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
```

Date: February 3, 2018.

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 1

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

Extra part: how does the memory profile look when you remove all memory allocation from the merge sort routine and instead use a work buffer allocated in the main function?

Task 2

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

This task is about *structure packing*, or minimizing the size of structs. When variables are stored in memory, they are always stored at an address that is a multiple of the size of the variable. This is called *natural alignment.* For example, a pointer on a 64-bit machine will be 8-byte aligned and can be stored in addresses ending in `0x0` or `0x8`.

Structs are stored in a similar way. Struct members are naturally aligned, and invisible "padding" bytes are inserted into the struct by the compiler to ensure this alignment. This means that the size of a struct is not necessarily equal to the sum of the size of each member; the struct size may be larger due to padding bytes. Moreover, the natural alignment of a struct is the same as the largest natural alignment of its members. For example, a struct containing a pointer and a char (9 bytes of data) will be 8-byte aligned.

You'll find four structs declared in `structs.c`. Without running anything, write down the size of each struct. Then execute the code to check your answers.

Gcc provides a way of removing the padding automatically. Add `__attribute__((__packed__))` to the structure declarations to see the packed size. Pay extra attention to `foo3`!

The compiler expects the attribute to be placed after the closing bracket of the struct. Example:

```
struct xy {
  char x;
  int y;
};
struct xy_packed {
  char x;
  int y;
} __attribute__((__packed__)) ;
```

The downside of using the packed attribute is that accessing the data will be slower (requiring extra instructions). Reordering the members of a struct so that large members come first will remove padding between members, although any padding in the end will remain.

Adjust the order of members in `foo3` and `foo4` and remove the packed attribute.

Extra part: write a loop that accesses foo3 and/or foo4 and compare the speed performance between the reordered versions and the packed attribute versions.

The `Task-2` directory also contains a small test program called `more_structs.c`. Look at that code and try to understand what it is doing.

The idea with `more_structs.c` is to explicitly look at how data in a struct is ordered and how padding bytes are added. A char buffer is filled with zeros, then that memory is used as a struct of the type C. The struct is filled with some data.

Run the code and look at the output, and make sure you understand what is happening and why the output looks the way it does.

Task 3

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

Bitfields provide a way to declare structure fields smaller than one byte. In C, this is done with the following syntax:

```
struct {
   type member_name : width;
};
```

The `type` of the field determines how the bitfield's value is interpreted. You'll usually want `unsigned int`, but `int` is also allowed. `member_name` is the name of the field, as usual. The `width` is the length of the bitfield in bits and can be in the range `[1,sizeof(type)*8]`. You can combine bitfields with ordinary types in a struct.

In the `Task-3` directory, you will find a program that "reads in" information about a series of button presses and stores them in an array. Then the information is copied to another array, using a different kind of struct.

Look at the two struct types that are declared in `buttonread.h` and write down the sizes you expect them to have. Then run the program and look at the sizes printed, to check if you were correct.

In this case, we know that the numbers stored are small, in the range 0-3 for "leftright" and "updown", and in the range 0-7 for "held". Therefore it should be possible to use the datatype "unsigned char" instead of "unsigned int". Change this in both structs, and then run the program again. What are the sizes of the structs now? Except for the sizes, the program should work in the same way as before.

Look in `main.c` where there is a loop copying values from `presses_org` to `presses_2`. As you can see, there is an outer loop repeating this 100000 times, just to give us a larger time to measure. Inside the inner loop, values are copied into the `presses_2` array. The bitfield accesses there may be a little slow. It can be faster to compose a bitfield by using `<<` and `|` (left shift and bitwise OR) than writing to the field members individually.

C will not let us assign a char directly to a `buttonpress_2` type, so we create a `char*` pointer that points to the same address as `presses_2[i]` — as you can see, a line setting such a pointer `p` already exists in the code. Now, your job is to do some left shift and bitwise OR operations to achieve the same effect as using the bitfield accesses. Remember to check that the result is still correct! Are you able to make it work? Does the code run faster after this change?

If you need a hint, consult Section 7.27 in the book by Agner Fog.

Unions

A *union* is a structure where members share the same memory space. This can be used to access the same data in different ways without pointer casting, but it can also be used to save memory space. *Caveats:* Because the use of unions prevents the use of register variables, it is not recommended to put simple variables into a union. Beware also that the compiler does not know if you use the union improperly (e.g. accidentally corrupt the data in one member by using the other).

A union is declared simply in a way that is analogous to a struct:

```
typedef union {
   float lookThisIsAFloat;
   int lookThisIsAnInt;
} myUnionType;

myUnionType foo;
foo.lookThisIsAFloat = 1.2345;  // valid!
foo.lookThisIsAnInt = 300;      // works!
```

It is possible to create a union of a struct and e.g. an int, for example:

```
 typedef union {
   struct {
     int a:23;
     int b:5;
   };
   int ab;
 } abType;
```

Note that the struct with members `a` and `b` then shares the same memory space as the int `ab`, so modifying `ab` can change `a` and/or `b`, and vice versa.

One example of how a union datatype can be used is given in the `Task-3/union_test.c` program. There, a union is used as a convenient way to extract and/or manipulate the individual bits of a `char`. Look at the code to figure out what it is doing, then try running it and modifying it to see the bit values for different `char` values.

Some other examples of using bitfields and unions can be found in section 14.9 in the book by Agner Fog. There, the sign bit, exponent, and mantissa parts of a floating-point number are extracted using bitfields.

Task 4

As you already know, the compiler is the best optimisation tool you have. This task will explore the GCC optimization options in more detail.

The main optimization flag is `-O` or (equivalently) `-O1`. With this setting, the compiler tries to reduce code size and execution time without performing any optimizations that take a great deal of compilation time.

With `-O2`, every optimization is turned on that does not result in a bigger executable. This option exists because the size of the executable affects the efficiency of the instruction cache.

Option `-O3` turns on additional optimization options that may increase the executable size or can take a very long time to compile.

Option `-Os` is like `-O2` but includes additional options that can shrink the executable size.

The `-Ofast` option is like `-O3`, but disregards strict IEEE standards for floating point math (it turns on `-ffast-math`).

(On some systems, `-O4` performs optimization at link time, enabling optimization across compilation units.)

This is only the beginning, however, because GCC will by default assume very little about the processor architecture and even the instruction set. By specifying the CPU architecture with the `-march` option, you let GCC use a wider, architecture-dependent instruction set. You can compile to any one of a number of target architectures, but you usually want to use `-march=native`, which automatically selects the architecture to match your system. The `-mtune` option tells GCC to just tune the compilation to suit a particular machine, while still using a smaller and more generic instruction set. `-march` implies the same tuning steps as `-mtune`.

Important: if you specify `-march`, the binary output will be less portable to other machines. If you want to use an executable or library on another machine, use `-mtune` instead.

In the `Task-4` directory, you'll find the code for a multigrid solver, a makefile, and a data file. The makefile will attempt to compile ten different versions of the solver as well as output the assembly for each. Your task is to evaluate the performance of each executable. Be sure to note the size as well. If you can, try doing this on a couple of different systems to see how the results differ.

Extra part: Use massif to compare the memory footprint of the original program in 64-bit mode and in 32-bit mode (compile with -m32). Note that to be able to do this you need the 32-bit dev library to be installed on the computer you are using.

Extra part: Consult http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html and specify optimization options manually to see which actually make a difference for this program.

Task 5

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

Here we will look at what can happen when a code uses uninitialized memory. Using uninitialized memory is a common programming error that can be tricky since it leads to unpredictable results, and different things may happen on different runs of the same program.

The code in `main.c` is calling a function `f()` several times. Inside the function `f()` an array "v" is placed on the stack, but then only some of the values in that array are initialized, while some of them are left uninitialized. Then all values are printed.

Look at the `f()` implementation code in `funcs.c` and make sure you understand what it is doing. Note that to initialize all values in the array, the loop on line 10 in `funcs.c` should have included all i-values, but the stepping `i+=2` means that every second value is left uninitialized.

Run the code and look at the output. Note that since the program used uninitialized memory for odd-numbered i the values printed there can be anything that happens to exist in those memory locations.

Next, add a call to `otherfunc(777777)` between the second and third call to `f()`. Does this affect the output from the third call to `f()`? Look at the `otherfunc()` implementation in `funcs.c` and try to understand what is happening.

Also try moving the `otherfunc(777777)` call to between the first and second call to `f()`, and changing the argument 777777 to some other number. How is the output from the program affected by those changes?

Earlier in this lab we saw how `valgrind` can be used to check a program for memory-access related errors. Try checking the program we have in this task using `valgrind` — does it work? Is `valgrind` able to detect that uninitialized data is being used?

Finally, fix the problem in the `f()` implementation by changing `i+=2` to `i+=1` on line 10 in `funcs.c`, so that no unititialized memory is used anymore. Run the code again to verify that there is no longer any unpredictable behavior, and that any previous calls to `otherfunc()` no longer affect the output from `f()`.

Extra part: Write a program that performs a similar experiment showing effects of uninitialized memory for heap memory usage. Perhaps something like this: allocate a large array on the heap and fill it with some data, then free the array, then allocate another buffer that you do not initialize. Can you see the old data from the previous array still lying there in the newly allocated buffer? Note that there is no guarantee that you will get the same memory for the second allocation, but the chances of that happening should increase if you use large buffers.