Operating Systems (67808)

Exercise 1

Due: 10/04/2025

Introduction

This exercise consists of two parts:

1. Usage of `strace` command for understanding program behavior. In this part, you will trace the system calls of a program using `strace`.

   This part worth 20 points.

2. Measurement of memory level access latency. This part focuses on measuring memory access latency across various cache levels. You will measure latencies for both random and sequential memory access patterns, and analyze the results. This part consists also a bonus part of 10 points.

   This part worth 80 points.

Notes:

- Before starting, ensure you read the course guidelines carefully to understand all requirements and expectations, especially the instructions regarding the README format.
- Start Early! Environment installation and code execution may take time.

1 First Assignment: Understanding Program Behavior using `strace`

The goal of this assignment is to help you become familiar with the `strace` command. `strace` is a Linux command that traces system calls and signals from a program. It is an important tool for debugging your programs in advanced exercises.

In this assignment, you should follow the `strace` of a program to understand what it does. You can assume that the program does only what you can see using `strace`.

To run the program, do the following:

- Download WhatIDo into an empty folder in your login on the CS-computers.
- Run the program using `strace`.
- Follow `strace` output.

Tip: Many lines at the beginning are part of the load of the program. The first "interesting" lines come immediately after the following lines:

```
execve("./WhatIDo", ["WhatIDo"], 0x7ffd6bf0efd0 /* 73 vars */) = 0
brk(NULL)                               = 0x564f0ac61000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=413196, ...}) = 0
mmap(NULL, 413196, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f5ce9581000
```

```
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260A\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1824496, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f5ce957f000
mmap(NULL, 1837056, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f5ce93be000
mprotect(0x7f5ce93e0000, 1658880, PROT_NONE) = 0
mmap(0x7f5ce93e0000, 1343488, PROT_READ|PROT_EXEC, MAP_PRIVATE ...
mmap(0x7f5ce9528000, 311296, PROT_READ, MAP_PRIVATE|MAP_FIXED ...
mmap(0x7f5ce9575000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE ...
mmap(0x7f5ce957b000, 14336, PROT_READ|PROT_WRITE, MAP_PRIVATE ...
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7f5ce9580500) = 0
mprotect(0x7f5ce9575000, 16384, PROT_READ) = 0
mprotect(0x564f093e2000, 4096, PROT_READ) = 0
mprotect(0x7f5ce960d000, 4096, PROT_READ) = 0
munmap(0x7f5ce9581000, 413196)          = 0
```

Your assignment is to provide a brief description of what the program does in the README file, with the title "Assignment 1". Tip: Google and the command `man` may be useful for this assignment.

2 Second Assignment: Memory-level access latency

Background: In this assignment, we will measure the memory read latency for the cache levels and RAM access in the target machine. To do that, we provide you with a partially implemented C code that is designed to measure memory access times as accurately as possible.

- We measure access to memory by allocating an array with the desired memory size, and accessing (reading) it single-threaded. The larger the array is, the less (relatively) it fits into the faster (and smaller) cache banks, and so we measure access to slower (and bigger) cache banks, until, for large enough arrays, we exhaust the largest cache and hit the RAM itself.
- To provide a baseline for our measurement, we also measure the same logic (access pattern) without memory access. This allows us to estimate the exact latency of a single read operation.
- Random access to memory: sequential access (or any access with a simple, predetermined pattern) may cause excessive cache hits, and thus skew the measurements. Therefore, we randomize the access to the memory by using a "random number generator" based on a 64-bit Galois LFSR whose output is taken modulo the memory size we measure.
- The modulo operation is implemented (in x64 architecture) as a single 64-bit unsigned divide (`DIVQ`). It turns out that on some old machines, `DIVQ`'s run time depends on the divisor — the larger the divisor is, the faster `DIVQ` runs (it makes sense, if you think about it). Therefore, we need to run the baseline measurement for each array size we measure.
- Another phenomenon we need to take into account is out-of-order execution. A modern CPU may re-order the execution of instructions in order to start a memory load (potentially a slow operation) earlier than its "native" timing. This will happen if the address to load from is known, and if there is no memory store in between. Thus, our code makes sure (to the extent possible) that all inner-loop instructions depend on previous instructions so that they cannot be re-ordered.

Assignment: This assignment has six parts:

1. Complete the missing parts in the `memory_latency` program provided to you.

2. Open one of the CSE machines of the university (not possible when connecting remotely). An alternative way would be to set up a virtual machine (VM) on your PC, although this method is inferior in measurement quality and therefore less preferred. Then, use the `memory_latency` program to measure memory access latency in the CSE computer terminal (or inside the VM).

3. Plot the results in a graph showing the cache level sizes, and attach it to your homework submission.

4. Explain the results in the README file under the title "Assignment 2".

2.1 The `measure` Library

The `measure` library is provided to you. It measures the average access time to a given array together with the baseline measurement (the same operation without memory access), averaged over *repeat* memory accesses. The library consists of one function that measures the access latency for an allocated array, and it's signature is:

```
struct measurement measure_latency(uint64_t repeat, array_element_t* arr, uint64_t arr_size,
                                   uint64_t zero)
```

Where:

- `struct measurement` is a structure containing the fields:
  - `double baseline` represents the average time (ns) taken to preform the measured operation without memory access.
  - `double per_cycle` represents the average time (ns) taken to preform the measured operation with memory access.
  - `uint64_t rnd` the variable that was used to randomly access the array. This is returned so that the compiler won't preform unnecessary (for us) optimizations.
- `uint64_t repeat`: the number of times that the function should repeat the measurement.
- `array_element_t* arr`: the allocated array to preform measurements on.

  The array needs to be initialized with non-repeating values (e.g. 1,2,3,...).
- `uint64_t arr_size`: the length of the array.
- `uint64_t zero` a variable containing zero but in a way that the compiler doesn't know it in compilation time. You are provided with a line that generates this variable. This is necessary to prevent the compiler from making unwanted optimizations.

We encourage you to look at the code and figure out how it operates. Be careful not to change anything in this code, it is very sensitive due to the delicacy of preventing the compiler from making unwanted optimizations. We also note that you will be tested only based on the contents of `memory_latency.cpp`, which needs to work with our version of the header files and the `measure.cpp` file.

2.2 The `memory_latency` Program

Your assignment is to complete the missing code in the `memory_latency.cpp` program. This program measures the average memory access latency of the different caching levels.

To measure the latency, we have provided you with the library `measure` as described in subsection 2.1. Be aware that the `measure_latency()` function was checked on computers or VMs running Debian 11 (more details in subsection 2.3), we cannot guarantee it to work on other versions of Unix-based 64-bit operating systems.

2.2.1 The `measure_sequential_latency()` function

In addition to the provided `measure_latency()` function that measures the access latency when a random access pattern is used, you will implement the `measure_sequential_latency()` function which measures the access latency when a sequential access pattern is used.

The signature of the `measure_sequential_latency()` function is the following:

```
struct measurement measure_sequential_latency(uint64_t repeat, array_element_t* arr,
                                              uint64_t arr_size, uint64_t zero)
```

Note that the signature of this function is the same as the signature of the `measure_latency` function.

Recommendations:

- Use the `register` keyword in order to indicate to the compiler that the variable you are using should be stored in the CPU register.
- Use variables (like `rnd` and `zero`) to prevent the compiler from preforming unwanted optimizations (by making sure each line is dependent on the previous line).
- Make the measured operations (in the inner loop) as similar as possible regarding the the number and type of operation you preform so that the measurements will be as comparable as possible.

You will need to make sure that the number of times you repeat the measurement is larger then the array size.

HINT: A good implementation of this function will differ only in two lines when compared to the provided `measure_latency()` function.

2.2.2 The input and output of `memory_latency`

When running the `memory_latency` program in the terminal with the following command:

```
./memory_latency max_size factor repeat
```

The program should run the measurement for arrays of sizes from 100 bytes up to *max_size* bytes and **repeat** the measurement *repeat* times for each array size.

The array sizes should be behave like a geometric series with *factor* as the ratio of the series. For example, running the command:

```
./memory_latency 800 1.5 200
```

would measure the latency for the arrays sizes $[100, 150, 225, 338, 507, 761]$ and repeat each measurement 200 times. This will be done both for random array access and sequential array access patterns. The program should deliver textual output to `stdout` where each line contains the size of the allocated array and the average latency measurements for accessing it ($offset = access\_time - baseline$), separated by a comma. Example:

```
mem_size1(bytes),offset1(random access latency, ns),offset1(sequential access latency, ns)
mem_size2(bytes),offset2(random access latency, ns),offset2(sequential access latency, ns)
           ...
           ...
           ...
```

2.2.3 Verify your code

Before proceeding, make sure that your code actually measures what you think it does. Your code should call the `measure_latency()` and `measure_sequential_latency()` functions once for each memory size to check. You should allocate a new array using `malloc()` (and remember to `free()` the array after measurement); the manual pages for these functions are available via the `man` command: `man malloc` and `man free`.

2.3 Setup

We highly recommend using the SCE computers for this program, because, as mentioned, running through a VM is inferior and the measurements will be less accurate. You can run your code on the SCE computers using the terminal (due to running time limitations, you can't run it when connecting remotely).

If you still wish to use virtual machine on your PC, the following steps should result in a machine suitable to the needs in this exercise. You will use the Debian 11 Linux-based OS that runs on a VM. The OS you will be running is the Debian 11.9 version available online.

1. Download and install VirtualBox from here.
2. Download the netinst CD image here (Most computers will need the amd64 version. Macs with M chipsets will need the arm64 version).
3. Inside VirtualBox create a new virtual machine with at least 6.4 GB of RAM (or 2 GB less then your machine has) and at least 2 cores using the downloaded image. Mark the **Guest Additions** option and set a password and username you will remember while setting-up the VM.
4. Start the VM and enable Drag and Drop (or a shared folder with the host) in the **devices** menu bar.

You may want to run as root so you can access sheared folders, this can be done with the command `su root` followed by your password. More information on installing VirtualBox and creating a new VM can be found here.

2.4 Measurement

Now that everything is ready, you can measure the average latency time on the SCE computers (or inside the VM on your PC). After gathering the data, you will create a graph of the results and submit it as an image file bundled together with your code. To measure the times inside the VM you will have to copy your source code into the VM and copy the results out of the VM. You may want to use a shared folder or the Drag and Drop function in VirtualBox on your PC.

2.4.1 Compilation

You should compile the program using the following command:

```
g++ -std=c++11 -O3 -Wall memory_latency.cpp measure.cpp -o memory_latency
```

IMPORTANT: The code should compile without any warnings.

Good values to start with for *max_size*, *factor* and *repeat* are 6000000000 for *max_size* (around 5.5GiB), 1.1 for *factor* and 100000000 for *repeat*.

Make sure that neither your host nor your VM goes to sleep while running the code. Also, to increase the accuracy of the measurements, it is recommended that you do not perform other operations at the same time.

2.4.2 Plotting

Obtaining Hardware Specs: First, extract the size of L1d, L2 and L3 cache levels using the command `lscpu` in the terminal (note that for cache levels that are not shared – L1d and L2 – we want the per-core values, more specifically, you will need to divide the total cache size reported by number of instances, i.e. the number of cores). More details about cache levels can be extracted by adding the `-C` flag to the `lscpu` command.

If your host is Windows, run the `lscpu` command inside the VM.

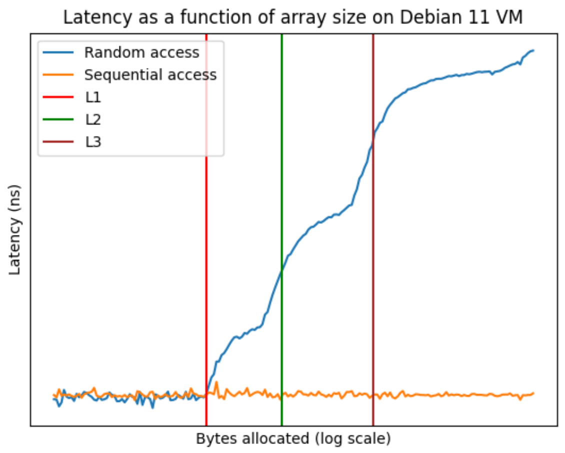

Figure 1: An example for the graph with measurements

Plotting Instructions:

- Include the name of the CPU model in the title of your graph (can be found in the `lscpu` command).
- Add a screenshot of the `lscpu` output (without the `-C` flag) with the file name `lscpu.png`.

Now that you have the cache sizes, plot a graph using the output values (one line for random array pattern and one line for sequential array access pattern) . You can use python/Matlab or any other program. The x-axis should have the memory size values in bytes and the y-axis the offset for the corresponding memory size in nanoseconds. Use a logarithmic scale on the y-axis and x-axis if necessary. You are provided with `plot_example.py` as an example of using Python's Matplotlib library for plotting. If you wish, you can also use Excel or LaTeX to do this.

Add the cache level sizes as vertical lines to the graph, mention the size of each cache level. Make sure that the graph is self-explanatory, namely:

1. It has a legend.
2. It has a caption.
3. Its axes have labels and units where applicable.
4. Each vertical line has a different color.
5. Each plot line has a different color.

An example for a graph without axis ticks and cache level sizes (**your graph should have them**) can be found in Figure 1.

2.4.3 Results Discussion

The README file should include an explanation of the graph's results. In your discussion, please address the following points:

- What trends or patterns do you observe in the graph?
- Do the results meet your expectations? If not, what discrepancies do you notice?
- Explain the differences in latency between random and sequential array access patterns.
- Discuss how the various cache levels influence the measured latency.

For additional details and guidelines, refer to section 4.

2.5 What to expect

Let's analyze the results and the reasons for the differences between the two measurements.

2.5.1 Random Access

At first, all memory reads will hit the L1d cache, which is the fastest (and smallest) cache. Each core has its own L1 cache (split into instruction cache – L1i, and data cache – L1d, not necessarily of the same size). The L1d cache is typically several $10KiBs$ (per core), with access time of ≈ 1 nano-second.

When the memory size exceeds L1d's size, it'll start hitting L2 cache, which is slower and bigger. On modern mid-to-high end machines, L2 is still per-core, and has a size of several hundred $KiBs$ to few $MiBs$, with access time of around 3 nano-seconds. When the accessed memory is small enough, a significant part of it will still hit L1d. For example, when the array is twice the size of L1d, half the array reads will still be served from L1d (and half will be served from L2). So we will get an average read latency as the average of that of L1d and L2. This is why the graph is expected to have a "knee" rather than a "step" at the cache cutoff point.

The same happens with the transition from L2 to L3 (for machines that have L3 cache).

ADVANCED NOTE: in the L2-to-L3 transition (as well as the L3-to-RAM), observe that latency begins to accumulate a bit before actually hitting the L2 size. This is because L2 (and higher caches) is keyed by the physical address of its data (as opposed to L1d which is keyed by the virtual address of its data). This means our contiguous block of data in virtual memory (the array) is not necessarily contiguous in physical memory. This in turn means that the cache sets may not be uniformly filled, and so we can experience L2 cache misses for full cache sets even before the array size hits the L2 cache size.

Please consult this stackoverflow question, and this caching explanation, to get an idea of typical cache size and latency figures (it's OK if your machine is a bit slower or faster than these).

2.5.2 Sequential Access

For small arrays (byte count < L1d size), the array completely fits into the L1d cache, and thus all access is from L1, with $\approx 1ns$ latency, just like the random access case.

For larger size arrays, the sequential case enjoys the fact that basic unit of memory access called "cache line" (or "cache block") is actually 64 bytes in modern CPUs. This means that when the CPU loads the first 64-bit array cell from memory, the actual data transferred to L1d is the 64 bytes of cells 0-7. Thus the next 7 loads are guaranteed to hit L1d. In other words, even if the array is much bigger than L1d, only one in 8 loads will generate a cache miss on L1d.

But it gets better. Modern CPUs usually have a "hardware prefetching" feature, wherein the CPU tries to predict which cache lines will be needed even before they are requested. For example, in modern Intel CPUs, the "DCU hardware prefetcher" will load the cache line starting at address $a + 64$ to L1d if three hits are made to a cache line representing addresses $a, \dots, a + 63$, as described in section 3.7 of this manual. There is a similar mechanism ("Streamer") to load data into L2. Similar optimizations are available in AMD CPUs. The net result is that due to the processing time of the inner loop, by the time the memory will be needed, it will already be present in L1d, due to the said optimizations, regardless of the array size. Thus we expect L1d-level latency for all array sizes.

Note that depending on your implementation for the `measure_sequential_latency()` function, the measured latency for sequential access can be below the L1d latency due to subtle changes un the compiled code.

Side-note: in our experiment, sequential access runs at L1d speed, but that's only because our code makes sure there are dependencies in data access. If we remove the dependencies, then memory access can be even faster than L1d speed, because the CPU can prefetch future-used data from the memory to its registers (not via the cache prefetcher – due to instruction re-ordering). In addition, our code makes sure that the compiler doesn't know we access the memory sequentially which prevents it from making additional optimizations like loop unrolling and software level prefetching.

2.6 Take-away points

In this exercise we measured the access latency for the different cache levels and with different access patterns. Here are some points we want you to take from this exercise:

- Memory access is expensive (time-wise), but caching reduces the cost when the memory referenced is small enough – the smaller the data set referenced, the smaller and faster cache it fits in. Particularly L1d cache is very fast (and small), so there's very little latency there.
- Sequential access hugely outperforms random access for large arrays, which demonstrates that the cache is optimized for sequential access. Indeed, modern CPUs employ cache optimizations (fetching and caching an entire cache line, prefetching) that make code that references memory sequentially (or nearly so) very efficient (de-facto hitting L1d all the time).
- Measuring can be quite tricky and requires good understanding of the phenomenon involved (cache levels, cache optimizations, CPU out-of-order execution).

To summarize, in order to decrease the run time of your code try to use as small objects as possible and/or access the data sequentially (this is of course relevant only to the performance-critical part of your code).

3 Bonus – Page Table Eviction Threshold (10 Points)

Paging is a mechanism that enables the operating system to manage physical memory efficiently by providing each process with a contiguous virtual memory space. Paging works by translating virtual addresses to physical addresses using a page table, and this translation incurs some memory overhead – the page table itself. In this section, we examine how the page table's footprint in the cache can impact performance when its entries no longer fit entirely in the cache. We encourage you to revisit this section later in the course, after studying virtual memory and paging in more detail.

3.1 The Assignment

In this section, we aim to estimate the page table eviction threshold—the point where page table entries start getting evicted from the last-level cache, leading to increased memory access latency. We will determine this threshold based on system parameters and visualize it on our latency graph, using data from the random access experiments. To estimate the threshold, you will need:

- The word (address) size of your system.
- The page size of your system.
- The L3 cache size (which you already have).

To obtain the first two values, perform the following: To get the size of the memory page, run the following command inside the virtual machine or on the CSE computer terminal:

```
getconf PAGE_SIZE
```

To get the word (address) size in your system, you will need your system type. The word size is 8 bytes on `64-bit` systems (and usually 4 bytes on `32-bit` systems). Your system type (`64-bit` or `32-bit`) also appears in the output of the `lscpu` command and can be found under **Settings**>**System**>**About**>**Device specifications**>**System type** if you are using Windows.

Now that you have all the data you need, calculate the following value which represents the estimated threshold for page table eviction[1]:

$$\textit{page table eviction threshold} \approx \frac{1}{2} \cdot \frac{\textit{page\_size}}{\textit{address\_size}} \cdot L3$$

3.2 Submission

Add the estimated threshold as another vertical line to your graph (and specify the threshold you calculated). An example for the graph with measurements as well as the estimated threshold can be found in Figure 2. Add a screenshot of the output of the `getconf PAGE_SIZE` command under the name `page_size.png`. Add an explanation to what happens between the L3 cache size and the estimated threshold and to what happens after it under the title 'Bonus' in the README file. You can find the full submission instructions in section 4.

3.3 What to expect

A surge in the memory access latency due to reading more data from the bigger and slower memory – the transition from L1 to L2 and from L2 to L3 – also happens with the transition from L3 to RAM. Naively we'd expect this to be the last transition, since after it, (almost) all data is read from the RAM. As you'll see, this is not so.

An interesting transition happens when the array size hits around few hundreds times the last-line cache (LLC) size (the last-line cache is the slowest, largest cache, usually shared among all cores – typically L3). Since each virtual memory access requires translation from a virtual address to a real address using the page table,[2] then each array read

[1] See Appendix A for proof

[2] The translation goes through the Page Table Entry (PTE) for the (virtual) page of the array cell; PTE is stored in a word, so the PTE size is word size.

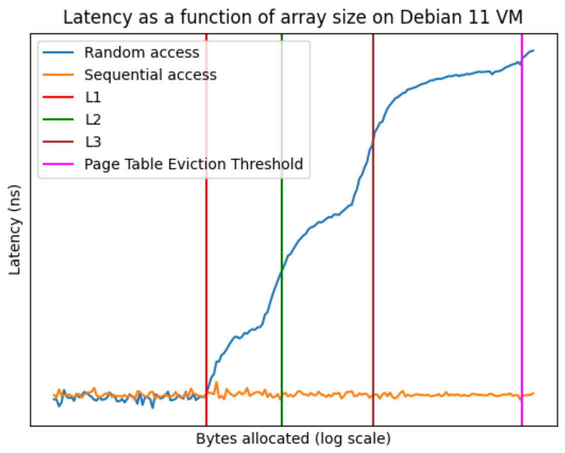

Figure 2: An example for the graph with measurements together with the estimated page eviction threshold

access entails two memory reads - a first one for the page table to calculate the physical address, followed by the actual data read. When reading the relevant entries in the page table start resulting in L3 cache misses, we will start seeing two RAM accesses per a read operation. Since each leaf page in the page table represents 512 real "data" pages[3], the behavior should be observed for memory access of at least 512 times the L3 cache. But in fact, we can expect it even before that. We set the threshold to be half of that, i.e. at 256 times the L3 cache size (details in Appendix A). Note that technically we could observe this behavior when the page table is not entirely contained in L2 (or L1d) but since the cache sizes grow by a factor of few dozen (and not few hundreds), when the page table "overflows" L2 and starts hitting L3, the array data is already fetched from RAM which is much slower than L3, thus we do not see a major slow-down. Likewise, when the page table overflows L1d into L2, the main array read is already from L3 which is much slower than L2 and thus we do not see a major slow down.

[3]Assuming word size = address size = 8 bytes, and a page size of 4096 bytes

4 Guidelines for the Entire Exercise

4.1 Coding

- Do not change the header files and the `measure` library. Your solution should work with our version of `memory_latency.h`, `measure.h` and `measure.cpp`.
- Ensure your code is readable (proper indentation, meaningful function names, etc.) and compliant with **the course guidelines**.
- The programming in this exercise is straightforward, but you must look at the results and think how to make them reasonably accurate. This may take time. Setting up the virtual machine can also take time.
- There is no single solution, and multiple ideas may work.
- Your code should check validity of input arguments. **Valid arguments are**:
  - $max\_size \geq 100$ (64-bit integer – natural number)
  - $factor > 1$ (float)
  - $repeat > 0$ (integer - natural number)
- In case of an error (system call error or invalid arguments) you should print an error massage and exit the program with -1.
- Make sure to check the exit status of all system calls you use (you can assume `timespec_get` will run successfully).

4.2 Measurement

- Note that running the code with the arguments suggested takes around half an hour (on CSE computers).
- You are encouraged to play with the arguments. Larger *repeat* value will yield more accurate results but will take longer to run. Set the allocated VM memory to at least $2GiB$ more than the *max_size* value you run the program with. For the bonus part you should set *max_size* to be larger by several applications of the *factor* than the estimated page table eviction threshold ($256 \cdot L3$ for most machines).
- How can you tell if your measurements are good enough? It is hard to get exact measurements. Approximate measurements are good enough for this exercise. You should check the following:
  - When you run the measurements several times on the same machine, you get similar results.
  - Above a certain threshold, the number of repetitions should not affect the average time of accessing the array.
  - You get results that you can explain and that are similar to the provided examples.
- Run your code on a computer **as idle as possible**. That is, make sure that as few other apps run concurrently with your program as possible. For example, it is better to run the program through the terminal, and not through some IDE such as CLion. Any CPU load may affect the results.

4.3 Submission

Submit a tar file containing the following:

- A `README` file. The `README` file should be structured according to the course guidelines and contains the required information described in both of the assignments (and the bonus part if you solved it). In order to be compliant with the guidelines, please use the `README` template that we provided.
- The source file for your implementation of the program described in Assignment 2.

- Your `Makefile` for Assignment 2. Running `make` with no arguments should generate the `memory_latency` executable file. You can use the provided `Makefile_Example` as an example.
- An image file with the graph of the results. (in `.png` format)
- The screenshot of the output for the `lscpu` command. (in `.png` format)
- If you solved the bonus part – add the screenshot of the output for the `getconf PAGE_SIZE` command (in `.png` format).

A A Proof that Exceeding the Page Eviction Threshold Results in Page Table Evictions

Denote by A the array size, P – the page size (typically $P = 4KiB$), W – the word/pointer size (on $64 - bit$ CPUs, typically $W = 8B$) and L – the LLC size.

CLAIM: Page table eviction occurs when $A > \frac{1}{2}\frac{P}{W}L$

PROOF: Assume to the contrary that $A = (\frac{1}{2} + \varepsilon)\frac{P}{W}L$ For some $\varepsilon > 0$ and no page table evictions occur. Since by assumption there is no page table eviction, all the leaf (last level) pages in the page table are in the LLC, thereby occupying $\frac{A}{P/W}$ space in the LLC. This leaves $(\frac{1}{2} - \varepsilon)L$ space in LLC for the array data, i.e. $(\frac{1}{2} - \varepsilon)\frac{L}{P}$ pages of array data. Since the access pattern is random, and the array is huge, each page of the array in LLC corresponds to a single array access. By the same argument, the average number of array accesses between access of a leaf page in the page table, is $\frac{A/P}{P/W} = (\frac{1}{2} + \varepsilon)\frac{L}{P}$. In order for the page table leaf pages not to be evicted, they cannot be accessed in intervals longer than the number of array pages in the LLC, i.e. $(\frac{1}{2} + \varepsilon)\frac{L}{P} < (\frac{1}{2} - \varepsilon)\frac{L}{P}$, but this contradicts $\varepsilon > 0$. QED.

To summarize: at around $256 \cdot L$ we expect to start seeing another bump, due to eviction of page table leaf pages from LLC, which asymptotically doubles the access time (as eventually every array data access is preceded by a memory fetch for the leaf page in the page table).