NWI-IBC019: Operating systems / Assignment: memory / 2022 v0

In this assignment, you will learn more about paging and it's impact on turnaround time. In other courses you have learned about complexity analysis and the big-O notation, e.g. $42 * n^2$ is abstracted to $O(n^2)$. However, in the trade-offs made in operating systems the constant that is hidden in the big-O annotation has a large impact. So $65536 * n$ (which whould be $O(n)$) is worse than $0.5 * n^2$ (which would be $O(n^2)$). The reason is that in operating system the numbers tend to be small, e.g. you will never have millions of processes running at the same time. You will read about this in an article written by Kamp. You are going to adjust some sample code so it will run faster.

NOTE *This is a group assignment (groups of 2).*

START TIME *Start after you have read the chapter about basic memory management.*

DURATION *Keep in mind the time you spend on this assignment: on average it will take **8 hours**.*

OBJECTIVES
- *gaining insight into complexity and timing in operating systems;*
- *understanding performance pitfalls of the memory system;*
- *using kernel support to gather statistics useful for tuning software implementations;*
- *using these statistics to optimize your implementation.;*

DEADLINE *You should hand it in before Sunday 30 October, 23:59. Send it to us using Brightspace.*

Background

On modern operating systems, not all memory referenced by a process is loaded in physical memory when the process is running. The remaining memory can be still on disk (e.g. swapped out or memory mapped), This is why these systems are called virtual memory systems. Data is only loaded when actually needed, this is when **page faults** can occur.

Page faults happen when a valid (assigned) memory location is accessed of which the contents are currently not in physical memory. The memory management unit (MMU) that is nowadays part of the processor will issue a page fault. This page fault will trigger the kernel to run. The virtual memory (VM) system (part of the kernel) steps in and checks if the memory location is supposed to be a valid memory location. If not, the program is aborted with a segmentation fault (SEGFAULT).

If the memory location is valid, the operating system will look up the data (probably from disk), fill the frames in physical memory, and update the page table of the process. The thread can resume afterwards. This process is blocking, inherently.

Due to pressure on the virtual memory system, caching tactics, interference between processes, etc, it is common that a couple of pages per process are not located in physical memory. However, handling these page faults and retrieving these pages can take a long time, and causes unresponsive applications, latency in database operations, etc. Page faults need to be avoided if possible.

However, page faults are hard to simulate and optimize for. Because the application need to be longer running, and you need to come up with realistic usage for other programs on your computer (or use a modified kernel to force certain behaviour). If you optimize for caching, you will also optimize for paging. Although they are not the same, similar techniques can be used. That is what this assignment is about.

Method

Using the `getrusage()` system call ('get resource usage'), we can get information about a number of important metrics related to the memory system. These are printed after the program executes. The full data structure as returned by `getrusage()` is (from the Linux man page):

```
struct rusage {
  struct timeval ru_utime; /* user CPU time used */
  struct timeval ru_stime; /* system CPU time used */
  long   ru_maxrss;        /* maximum resident set size */
  long   ru_ixrss;         /* integral shared memory size */
  long   ru_idrss;         /* integral unshared data size */
  long   ru_isrss;         /* integral unshared stack size */
  long   ru_minflt;        /* page reclaims (soft page faults) */
  long   ru_majflt;        /* page faults (hard page faults) */
  long   ru_nswap;         /* swaps */
  long   ru_inblock;       /* block input operations */
  long   ru_oublock;       /* block output operations */
  long   ru_msgsnd;        /* IPC messages sent */
  long   ru_msgrcv;        /* IPC messages received */
  long   ru_nsignals;      /* signals received */
  long   ru_nvcsw;         /* voluntary context switches */
  long   ru_nivcsw;        /* involuntary context switches */
};
```

listing 1 Data structure as returned by `getrusage()`.

At the end of the `main()` function, you can print these statistics. By executing your program for a number of scenarios, you can increase your understanding of its performance.

Another tool you can use is the `perf` tool on Linux. This tool uses the kernel to retrieve metrics about the cache hits and misses. On FreeBSD you can use `pmcstat`. These tools are a bit more involved, because they can depend on your hardware, as specific hardware registers are used. For `perf`, the following command might work (depending on your CPU): `sudo perf stat -B -e cache-references,cache-misses,cycles,instructions,branches,faults,L1-dcache-load-misses,L1-dcache-loads,L1-dcache-stores ./memory`.

As is the case with all performance measurements, be careful that you choose representative cases. A common pitfall is to choose a non-realistic case, and start micro-optimising for that case, but degrading performance for other cases.

Be advised that on a different system, or on your current system after a kernel upgrade, the program may behave differently because details (in this case) in the memory system has been changed. This happened for example with the Meltdown and Spectre patches (large security problem originating in the interplay between hardware and operating systems, more on that in the course Operating Systems Security).

The above is the reason it is important to understand the concepts and organization of memory, and not blindly measure and optimize.

Problem

STEP 0 Download the new project in your current development environment of the shell assignment at `https://gitlab.science.ru.nl/operatingsystems/assignment3`. If you want to use the `perf` tool on Linux, you might need to install it on Debian/Ubuntu systems with:
`sudo apt install linux-tools-common linux-tools-generic`

STEP 1 Look at the sample code that is given. We are going to optimize this code. Have a clear understanding of what this code does. Play with the input parameters, and look what the impact on the number of paging operations, CPU usage, and turnaround time is. Choose a `SIZE` that fit your memory, and a `REPEAT` so that running the example will take at least 20 seconds. For the remainder of the assignment, these numbers are fixed. Document these values, turn-around and CPU times, and `getrusage()` values as **your baseline**. Add a thorough description of your test environment/setup (processor, number of cores, amount of memory, operating system, both guest and host, emulation, virtual machines, etc). List also the amount of cache (which can be found using `sudo lshw`).

STEP 2 Read the article by Kamp [Kam10]. Write a **short** summary (max 250 words) of the issue that is raised in general (hint: not about paging).

STEP 3 Explain the **reason behind** the design choice that the first generation of elements in a B-Heap page do not expand (copying/quoting '... going to compare them both with their parent' from the text is not an answer). In your answer relate to the concepts in the book and the lectures. Does this work for **all page sizes**? Note that page sizes are always a power of 2, starting from 4096 bytes.

STEP 4 Now try to adjust the algorithm (given the boundaries as indicated in the source). Your goal is to reduce the turnaround time **and** reduce the number of CPU cycles used. The means to this goal is to reduce the number of paging operations, or as a proxy for that (as explained above) improve the caching. Try to keep the dummy value the same. Explain all your modifications, and document this, supported by measurements and explanation why the modification should prevent page faults in worst case scenarios.

STEP 5 Now take your final measurements. Important are the number of paging operations, CPU usage, and turnaround time. Compare this with your baseline in step 1. Explain the remaining page faults.

NOTE *If you use a virtual machine for this assignment, give this VM multiple cores to work with.*

HAND IN
- `main.cpp`, *with the comments added in step 3.*
- *a report adhering to template 2.*

```
Report

Names & and student numbers: ... ..
Your report should be in .pdf, and it should contain your names and student numbers.
In addition, the following sections are required

1. Summary of the article by Kamp
2. An explanation why the first generation in a B-heap does not expand,
   and an explanation of whether this works for all page sizes.
3. Baseline measurements and a description of the machine on which you test
4. A list of all the changes you made and for every change, why you think
   it will improve the performance. Support your claims with measurements.
5. The final measurements and a comparison with the baseline measurements
```

template 2 Report template

Bibliography

[Kam10] Poul-Henning Kamp. "**You're doing it wrong**". In: *Communications of the acm* 53.7 (2010). http://queue.acm.org/detail.cfm?id=1814327, pages 55–59. DOI: 10.1145/1785414.1785434 (cited on page 3).