Debugging performance issues in kernel space: minor fault and major faults

July 6, 2021Kernel Space and Performance, PerformanceLeave a Reply

*We at **Johnny's Software Lab LLC** are experts in performance. If performance is in any way concern in your software project, feel free to contact us.*

In the previous post we talked about debugging performance issues in kernel space, more specifically, how to find out which system calls are taking the most time and we explored some ideas on how to fix it. But the system calls are not the only reason why your program might be spending a lot of time in kernel space.

In this post **we talk about the second reason why your program might be spending a lot of time in kernel space, and these are *traps***. The official definition of *trap is a type of synchronous interrupt caused by an exceptional condition (e.g., breakpoint, division by zero, invalid memory access, etc). A trap usually results in a switch to kernel mode, wherein the operating system performs some action before returning control to the originating process*[1].

There are many ways why your program might switch into kernel space, some of which are completely harmless and expected; others that can make your program much slower than it actually needs to be. **In this post we will explore two types of traps to the kernel and their effect on performance: major faults and minor faults**. We will also show you how to check if your code has a problem with trapping to kernel and how to find the spots that cause the traps.

Types of traps

As already said, there are several types of traps that are more or less severe. In this post we are covering only minor faults and major faults, in the follow-up post we will cover other types as well: emulation faults, alignment faults, context switches and cpu migrations.

Minor-faults

When your program requests memory from the operating system (e.g. system allocator requests large blocks of memory from the operating system), **the operating system doesn't immediately allocate all the requested memory**. Many times, programs ask for a lot of memory without actually using it later on and it would be a waste to allocate such a memory block to the program. What happens instead, is that **the operating system allocates the requested amount of virtual memory, but the virtual memory is not backed up by physical memory**.

When your program tries to access a memory page that is not backed up by physical memory, a minor fault occurs and the operating system takes over the control of your program. It

allocates a free physical memory page, assigns it to the virtual page and returns the execution back to your program.

Minor faults are completely normal and fine, and rarely do the programs suffer from performance problems due to minor faults. Once a memory page is allocated, there will be no further performance price for access to the same page. Minor faults are known to cause problems in low latency environments since the program will be spending time in kernel space allocating memory instead of doing useful work.

Decreasing the number of minor-faults

The simplest way to reduce the number of minor faults is to use less memory for your data. In the performance contest we organized with Denis Bakhvalov from EasyPerf.net and which dealt with image processing, the original program allocated several temporary buffers the size of the whole image. Several contestants made an optimization to use a temporary buffer which is the size of only one row. This decreased the number of minor faults drastically.

If this approach doesn't work for you, there are others. In Linux, you can allocate a large piece of memory using `mmap` system call. **One of the parameters to `mmap` system call is `MAP_POPULATE`, which when provided, will force the operating system to back up the allocated virtual memory with physical memory**. This means that there will be no minor faults when the program is accessing this block of memory later. Bear in mind that `mmap` with `MAP_POPULATE` will take a longer time to complete, but overall, the speed of your program will be slightly increased.

Here is an example of the allocate that allocates preallocated memory using `mmap`. Be careful not to use `mmap` to allocate memory smaller than the page size (which is in most systems 4 kB), since `mmap` allocates memory at multiple of the memory page size.

```
void* malloc_large(size_t size) {
    return mmap(0, size, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_POPULATE, -1, 0);
}
void free_large(void* p, size_t size) {
   munmap(p, size);
}
```

In our measurement with an image processing algorithm from the performance challenge 4, when the algorithm was used on a very large picture, the runtime with memory blocks allocated using regular `malloc` was 3.398 s (standard deviation 0.021, ten runs). When we replaced six `malloc` calls that allocate the same amount of memory as the image size with `malloc_large`, the runtime fell down to 3.15 s (standard deviation 0.055). The original program had about 486,270 minor faults. After the change, this number fell down to 97,460.

A large block of memory that is allocated, used completely a small number of times, and then disposed of makes a good candidate for preallocation. Note, however, the time saved using this technique will most of the time be small, as rarely the bottleneck will be touching the block of unallocated virtual memory for the first time.

Major faults

Major faults are similar to minor faults in the sense that the virtual memory page is not backed up by a physical page. But the similarity stops there. In the case of a minor fault, the operating system just had to allocate a new physical page. **In the case of a major fault, the data needed to fill the page is on the disk, and the operating system needs to access the disk to fetch it**.

Fetching a page from the hard disk is much more expensive than allocating a memory page, and ideally, you would want to decrease the number of major faults as much as possible. There are two main reasons why major faults happen:

- **The program is accessing binary code for the first time**. On many operating systems, including Linux, the operating system doesn't load all the binary code before the program start. Instead, it creates a mapping between the virtual memory and the code in the libraries on disk. When your program tries to access binary code for the first time, a major fault happens. The operating system then loads the missing binary code from the disk to the operating system, and the program can resume.
- **The operating system ran out of memory for all the running processes in the system, so**, to be able to continue running, **it swapped out some physical pages from the memory to the hard drive**. When the program tries to access the virtual page corresponding to the swapped physical page, a major fault happens. This causes the operating system to take over the control of the execution and load the missing page back to the memory.

For the case where the program accesses binary code for the first time, major faults are completely expected. This approach saves memory as unused parts of the program are not loaded into memory. The number of major faults will roughly be proportional to the program size – larger programs typically have more major faults. This type of major fault has an effect on loading time: decreasing the number of major faults typically results in faster loading time.

When you run your program for the first time since the OS booted, it loads slower and you can expect some major faults. The second time you run the same program, however, the operating system has already cached parts of the program or the whole program in the main memory. The number of major faults will decrease significantly or disappear altogether. Here is the number of major faults for a simple `hello-world.js` program executed through V8 JavaScript engine:

```
$ perf stat -e major-faults,minor-faults d8 hello_world.js
 ...
 Performance counter stats for 'd8 hello_world.js':

               69      major-faults
             3460      minor-faults
 ...

$ perf stat -e major-faults,minor-faults d8 hello_world.js
 ...
 Performance counter stats for 'd8 hello_world.js':

                0      major-faults
             3490      minor-faults
...
```

The first time that the program executed it caused 69 major faults. The second time, it didn't cause any major faults, since all the binaries were already present in the main memory from the previous run.

Decreasing the number of major faults

If major faults are responsible for your program loading slowly, the problem can be solved by decreasing the binary size or increasing the code locality.

Decrease binary size

One of the ways to decrease binary size is through compiler switches. Most compilers offer a compiler switch that specifically targets binary size, e.g. GCC and CLANG offer a compiler switch `-Os` that instructs the compiler to turn off all optimizations that result in binary size increase. In our experiment, V8 running on OCTANE benchmark suite experienced 303 major faults when compiled with `-O3` switch, in contrast with 265 major faults when compiled with `-Os` switch.

Note however, compiling with `-Os` instead of `-O3` can also cause general performance degradation, since many useful optimizations are disabled with `-Os`.

Increase code locality

Another option to decrease the number of major faults is the usage of tools that increase code locality. **By increasing code locality, what we mean is moving the functions that often call one another close to each other in memory**. It also means moving parts of the functions that are rarely executed out of the functions into separate memory locations. Both approaches can in principle decrease the number of major faults since the code will reside more compact in memory and also increase instruction cache hit rate.

However, the primary goal of increasing code locality is speeding up the overall program execution, not just the program load time. So there is no guarantee that this approach will result in a decrease in the number of major faults. Actually, in one of our measurements, the number of major faults actually increased. Nevertheless, we are including it on the list, with a remark that you will need to test and confirm its behavior on your specific program.

There are a few tools that you can use to increase the code locality of your program. All of them work by **first observing your program's behavior to understand how functions call one another, and then they modify the function memory layout accordingly.**

- **Profile Guided Optimizations (PGO) with instrumentation**: tool that is used to perform the PGO is the compiler. In the first step, the compiler compiles your program with PGO instrumentation in place. In the second step, you run your PGO instrumented program on a real-world load to collect the information needed for code locality optimizations. And lastly, you run the compiler again, this time feeding it the information collected in the previous step. The compilers can now generate a binary with an optimized memory layout, among others. Supported by GCC and CLANG. We talked about PGO in a previous post in more details.

- **Profile Guided Optimization (PGO) with runtime information from an external profiler**: very similar to the previous case. The difference is that, instead of compiler generating PGO instrumented binary to collect the information, we use an external profiler to do that, for example `perf`.
- **BOLT (Binary Optimization and Layout Tool)**: a LLVM based tool that uses information collected by `perf` profiler to optimize the code layout in order to increase code locality. Conceptually very similar to the previous bullet, except that you don't need to recompile the whole program, you just need to relink your existing program.

Tips to avoid swapping

For the case where the piece of memory was swapped to the hard disk because of lack of memory, there are a few ways to fix it:

- **Get more system memory**
- **Decrease the number of parallel running processes** – decreasing the number of running processes typically results in smaller memory usage and decreases the chance of memory pages being swapped out to disk
- **Decrease the memory consumption of your program** – there are many ways to do it, here are a few: decrease the use of temporary buffers, reuse buffers, recompile your program for a 32-bit system, use more efficient data structures, decrease memory fragmentation, use system allocators with lower memory usage, use custom allocator for data structures that consume a lot of memory, etc.

Nevertheless, if you are running in a constrained environment where you can expect some memory swapping, **you can use system call `mlock` to prevent a memory block from being swapped to disk**. Originally, this call was introduced to prevent a memory block from being swapped to disk for security reasons. But as a side effect, it can make your program faster if you expect the system it is running on to run out of physical memory.

Finding code that causes major and minor faults

The simplest way to find the code that causes major and minor faults is to use perf. The following commands will record the program's execution for you to be able to find the source of major and minor faults:

```
$ perf record --call-graph dwarf -e major-faults,minor-faults ./my_program
$ perf report
```

The first command record the program execution, the second command displays the results. If you are unfamiliar with `perf`, I suggest starting at Perf Wiki.

Conclusion

In this post we talked about major and minor faults and their impact on performance. Although they can be very expensive, in most applications (with an exception of the low-latency domain) they rarely pose a problem and you should not expect a speedup that is higher than a few percent.

In the follow-up post we will talk about other types of traps that can happen: emulation faults, alignment faults, context switches and cpu migrations.

Additional Information

Runtime Performance Optimization Blueprint: Intel Architecture Optimization with Last Branch Record (LBR)