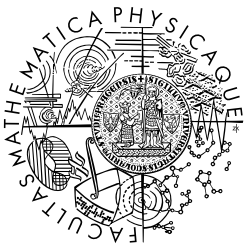

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Oliver Glitta

Debugging the SLUB allocator in the Linux kernel

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: RNDr. Vlastimil Babka, Ph.D.

Study programme: Computer Science

Study branch: IPSS

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date .....

.....

Author's signature

I would like to thank my supervisor, RNDr. Vlastimil Babka, Ph.D., for introducing me to the Linux development process and his guidance through this work. Also, for his many useful comments and suggestions about the thesis. I would like to thank prof. Petr Tůma, Dr., for his advice and endless ideas on the thesis. I am also grateful for the comments and suggestions on the code by Linux maintainers.

Title: Debugging the SLUB allocator in the Linux kernel

Author: Oliver Glitta

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Vlastimil Babka, Ph.D., SUSE

Abstract: This thesis is focused on the SLUB memory allocator and its debugging functionalities. The SLUB allocator implementation included several tests, but nobody was running them. It was because no automated kernel testing infrastructure existed for them. This project added one of the kernel testing frameworks, the KUnit testing, for SLUB. Existing tests were integrated into this framework with some minor changes. Also, new regression tests were added based on a systematic review of commits associated with SLUB. Another part of the thesis extends the SLUB debug options for tracking allocation events, including more efficient stack trace storage. The last part improves the virtual file system debugfs that provide information about kernel components to userspace by including stack traces in SLUB's virtual files. Also, this project helps to track the efficiency of the SLUB cache usage in terms of object size. All of these changes to the Linux kernel should help with SLUB debugging. Some of the changes were also submitted and even accepted by the community, they were added into Linux version 5.14.

Keywords: linux kernel allocator SLUB debugging

Contents

Introduction

The Linux kernel is an open-source operating system kernel. It is used in a large number of distributions. The kernel is the core of an operating system and has complete control over the system. It starts as one of the first components during system boot and handles the start of other parts of the system.

One of the tasks handled in the kernel is memory management. The kernel allocates memory pages for userspace processes and translates physical addresses into virtual ones. So, userspace processes operate only with virtual addresses. However, the kernel also needs memory for its processes. For this reason, it has its memory allocators that handle kernel memory.

The memory allocators in the Linux kernel are focused on smaller objects that are the most common. In the current Linux kernel version, three different allocators exist - SLAB, SLUB, and SLOB.

The default allocator for most Linux distro kernels these days is the SLUB allocator. It is the newest implementation, and its performance is still improving. Developers add new features and change existing code to make the allocator as effective and reliable as possible. With every change made, a developer needs to check if the allocator still works as it should. For this reason, it is beneficial to have some tests.

In the kernel, the testing is a little bit different than with typical programs. We can test a program simply by running it, and when the error occurs, we can fix it and rerun it. When we want to test some component of the kernel, we need to boot it first. It is bad to boot the kernel with experimental changes without having a separate system because we can lose control over the system. It is the reason why special tools for testing exists in the kernel. They are KUnit and self tests. They provide a framework for testing. Moreover, the advantage of using the testing framework is that it helps us with execution, data supplying, validation, and result reporting.

The SLUB allocator has included several tests in its implementation. Unfortunately, manual code modification is needed to enable their build and execution, and thus they are never used in practice. It would be helpful for developers to have some tests that can be run quickly and detect common classes of bugs in the system.

ANOTHER PART of software development is debugging when some bug or error occurs. During debugging, getting as much information about a system as possible is essential. However, storing and processing the debugging information also needs to be effective, so the system would not be overwhelmed. With the SLUB allocator, it is possible to specify during boot that some or all classes of objects will have some additional information stored. This information can help when some error occurs—for example, a stack trace that leads to the object allocation or freeing.

Also, the Linux kernel has many tools for debugging. For example, testing is one of them. Another way to debug kernel components is to use virtual file systems that provide information about selected kernel structures to userspace. Among file systems that SLUB uses are proc, sysfs, or debugfs. They provide information about SLUB caches used in the kernel.

THIS PROJECT focuses on the SLUB allocator debugging. It aims to implement one of the kernel testing frameworks for SLUB. Then to incorporate existing tests and create some new ones to be used during development to avoid reintroducing previously fixed bugs. Another task is to improve the implementation of debugging information. The last part improves the output of debugging file systems.

THE THESIS is divided into five chapters. The first two chapters are mostly theoretical and explain the Linux kernel and its SLUB memory allocator. Chapter 1 talks about the Linux community, testing, and building of the kernel. Chapter 2 is dedicated to the SLUB memory allocator, its structure, debugging options, and validation. After that, the third chapter sets goals for the thesis. The following Chapter 4 discussed problems and solutions in SLUB debugging. It has four parts. Two of them are focusing on testing, the third on stack trace, and the last on debugfs files for SLUB. The last, fifth chapter of the thesis overview the code added in this project.

Chapter 1: Linux kernel

1.1 Linux development

The Linux kernel is open-source software, and everybody can contribute to its development. Linus Torvalds created it, and he is now maintaining the mainline tree. A few branches exist in the development process -the mainline tree, stable trees with multiple major numbers, subsystem-specific trees, linux-next integration testing tree. [1] Each branch and every subsystem of Linux has its maintainers who verify new changes submitted by other contributors. Maintainers are responsible for keeping their subsystems working.

[1] For more detailed information about branches, see kernel documentation.

The main goal of every developer is to get his changes into the mainline tree. The author of the kernel himself is maintaining this tree. Every contributor can increase his chances of success by following some basic rules. First of all, the baseline of changes should be some of the current kernel branches, so it is easier for maintainers to apply them. Also, it is important to adequately describe changes and why it is good to add them into the kernel. In the description, the author needs to convince reviewers that his change makes sense. Another tip worth following is that patches should be small because fewer changes are easier to check and verify their correctness. It is good practice to divide a big patch with many changes into smaller patches with fewer changes. Also, the good idea is to respect the kernel coding style. We can check it with the style checker inside the kernel tree (scripts/checkpatch.pl). Before submitting the patch, the contributor should remove or justify all violations the checker reported.

With a lot of contributors and developers, it is crucial to keep track of who did what. For this purpose, the tag "Signed-off-by" was created. This tag is an alternative for a signed legal contract. It should be used by the author, but also by maintainers. It says that a maintainer has permission to submit the code. After the tag, the author should add his real name and email address. This tag must be at the end of the patch description. In the development process, more tags exist like Acked-by, Reported-by, Reviewed-by. Each one has its meaning and helps with verifying changes and fixing problems.

Another step in submitting patches is finding correct recipients and sending the patch to them. The contributor should send his

patch to every maintainer of the subsystems that is touching. With this, the script scripts/get_maintainer.pl can help. Besides maintainers, it is beneficial to send the patch to an appropriate mailing list. The best is to choose these lists which are related to the topic. On the mailing list, more people can review and test changes. One of the options for sending the patch is the command git send-email. [2]

After the submitting a patch, maintainers test it on different platforms with different configurations which they are using. The advantage of this approach is that many errors and bugs are detected. Furthermore, failures are revealed before changes are added to some of the main branches. So, we can fix errors and sometimes improve intended changes when someone else suggests a better solution for the problem.

[2] More rules and a more detailed description of these mentioned here are in the kernel documentation.

1.2 Kernel Testing

When we are developing some software, we wants to make it as reliable as possible. Many different strategies exist to achieve this. For example, we can start with choosing an appropriate programming language. Also, it is good to have someone else who can check the code. So, using a clean coding style can be very beneficial. Another method is testing.

We can divide most of the tests into three main categories. Tests that check only small isolated parts of the system are called unit tests. They are mostly fast and easy to run, so it is beneficial to use them when developing a program. After that, we may want to check how different parts are working together. So, we can take two or three parts and test their interaction, for example, between driver and hardware. We call these tests integration tests. When we checked different parts working together, we also want to check the entire system. These tests are called end-to-end tests.

Some testing methods can also include more extensive tests, mainly focused on performance when a program is overwhelmed. The goal of this intense testing is to determine the stability of the software. This method is called stress testing. It requires more time to perform, and developers do not use it very much when editing software. Common practice is that some external person, called a tester, does this kind of test.

Testing can be done manually or by creating some automatic tests. Testing a program is usually done by running it and experimenting with different inputs and paths in code. Commonly, the program crashes when some error occurs. In this case, we can rerun the program.

When we want to test some functionality in the kernel, we need to boot it first. It is not a good idea to boot the kernel with experimental changes because when an error occurs and the kernel crashes, we may lose control of the test system. For this reason, we have

unique frameworks for testing in the kernel. The same goal can be achieved by using virtualization, for example.

However, testing frameworks have an advantage over virtualization. They help us with the execution of the code. Moreover, they validate a test and report results in a simple and readable form. They also improve the speed and the accuracy of tests. Two main tools used in Linux for testing are KUnit tests and Linux kernel selftests.

KUNIT[3] is a framework for the unit testing method in the kernel. The inspiration behind it is mainly from JUnit, Python's unittest.mock, and Googletest/Googlemock for C++. It provides kunit_tool, which uses User Mode Linux (UML) [4] kernel to build, run and parse results. UML is open source virtualization platform. Its advantage is that it does not use any hardware virtualization features.

UML runs the kernel as a standard process running on Linux. UML is not the only way to run tests, but it is recommended. Primarily, all tests run on kernel startup using this method. On the other hand, they can also be built as a module and run when the test module is loaded. Test results are written to kernel log and have a simple TAP (Test Anything Protocol) [5] format.

[3] Kunit. https://kunit.dev/

[4] Anton Ivanov. Uml howto v2. https://www.kernel.org/doc/html/latest/virt/uml/user_mode_linux_howto_v2.html, Sep 2020

[5] More about TAP format here.

LINUX KERNEL SELFTESTS are meant to be small tests that test individual paths in code. This framework is commonly used for testing kernel components from userspace. For example, tests are created for tools like breakpoints, ftrace, mount. All self tests are in /tools/testing/selftests directory. When contributing new tests, we need to follow few rules, which are mentioned in the documentation: [6]

[6] Linux kernel selftests. https://www.kernel.org/doc/html/latest/dev-tools/kselftest.html

- *Do as much as you can if you're not root;*
- *Don't take too long;*
- *Don't break the build on any architecture, and*
- *Don't cause the top-level "make run_tests" to fail if your feature is unconfigured.*

Self tests can be used to test userspace. For this purpose, a test harness exists. On the other hand, with these tests, we can test tools within the kernel, too. A standard method for this is to create them as a module. To run tests as a module, we can use shell script kselftest/module.sh. Self tests are meant to run after building, installing, and booting a kernel.

THE BIGGEST DIFFERENCE between KUnit and self tests is how we run them and what they can test. They both can be used to test kernel components. However, in this case, it is better to use KUnit tests because running them is easier and quicker, for example, by

using kunit_tool. On the other hand, when we want to test a user space component, the self tests are more appropriate to use.

In this project, the focus is on improving debugging functionality for SLUB. Smaller tests are preferable in the kernel because they can run quickly and do not require much time. For this reason, only smaller KUnit tests are used and not, for example, stress testing.

1.3 Building kernel

When testing the Linux kernel, it is good to use virtualization rather than running the kernel directly on the developer machine. One of the reasons is that when an error occurs and causes panic (a special term for kernel emergency shutdown), there is no easy way to recover. On the other hand, when using a virtualized version of the kernel and the panic shutdown occurs, we can update the source code, build the kernel and rerun it.

A good tool for kernel debugging virtualization is Virtme. As its readme says:

"*Virtme is a set of simple tools to run a virtualized Linux kernel that uses the host Linux distribution or a simple rootfs instead of a whole disk image. Virtme is tiny, easy to use, and makes testing kernel changes quite simple.*"[7]

[7] Andrew Lutomirski. Virtme documentation. https://github.com/amluto/virtme, Feb 2014

Many Linux distributions have a package of this tool. If the package is not available, we can use the github repo of the author of this tool, Andy Lutomirski.

The tutorial on how to build, boot the Linux kernel and test the thesis results using virtualization is in the First Attachment: How to test the thesis results.

Chapter 2: SLUB memory allocator

2.1 Memory allocators

The memory of an executing program is roughly divided into two main parts. The first one is the stack. It contains things like local variables, function return address, function arguments. The job of the compiler is to generate code that performs the allocation and deallocation of this memory. In particular, it deallocates local variables when returning from a function. For this reason, it is sometimes called automatic memory.

However, sometimes, we need a variable whose lifetime is not bound to the scope of the allocating function. In this case, we use the second main part of the memory called the heap.

In C, functions that allocate and deallocate memory on the heap are called explicitly by the programmer. So, when we want to allocate some longer-living memory, we call, for example, the malloc() function, and the memory allocator finds a memory block and returns a pointer to it. When we want to free memory, we can call the free() function. In most other languages, allocation is still a programmer's job, but tools like garbage collectors exist for the deallocation.

Memory allocators' job is to handle allocation and deallocation operations by keeping track of individual memory blocks on the heap. Mainly, it manages the free blocks because keeping track of the allocated blocks is the programmer's responsibility. A basic technique is to keep free blocks in a linked list. Every free block also needs to have a header to store additional information, such as its size.

When allocating new memory, the allocator has to find a block with sufficient size or split a bigger one. Many strategies for finding the right block exists. For example, among the basic ones are the Best Fit, Worst Fit, and First Fit. More complex are Segregated Lists, Buddy Allocation. Every strategy has its positive and negative sides - after a few allocations and deallocations, the heap can probably be fragmented. [8]

In the Linux kernel, the default memory allocator is SLUB. This project is focused on improving this allocator's debugging functionality. The SLUB allocator is described in next section. It is one of the three allocators in the kernel. Others are SLAB and SLOB.

[8] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Aug, 2018

OTHER INTERESTING IMPLEMENTATIONS of the malloc function, outside of the kernel, are jemalloc, tcmalloc, ptmalloc, hoard. Each of them is optimized for a different type of memory use. For example, SLUB is primarily for smaller objects which are frequently used in the Linux kernel.

The jemalloc[9] allocator is used, for example, in the Firefox web browser. The comment inside the Firefox source code best describes the main focus of this allocator:

"*This allocator implementation is designed to provide scalable performance for multi-threaded programs on multiprocessor systems.*"[10]

Google's implementation of malloc() is called tcmalloc. [11] This allocator is best performing on a smaller number of threads, where it needs a smaller number of CPU cycles.

Ptmalloc [12] is the malloc() implementation in Unix systems. It is an extension of dlmalloc for multithreaded usage.

Another exciting memory allocator optimized for multithreaded and multiprocessor allocation is hoard[13]. It is multiplatform software, so we can also use it on Windows, Linux, or Mac OS X.

A performance scalability comparison of the four memory allocators - jemalloc, tcmalloc, ptmalloc and, hoard - has been done in 2018. It was performed on a system with this specifcation:

"*We run all our tests on a more or less typical 2-socket server box, with two E5645, each of E5645's having 6 cores with Hyper-threading (="2-way SMT"), 12M of L3 cache, and capable of running at 2.4GHz (2.67Ghz in turbo mode). The box has 32G of RAM. OS: Debian 9 "Stretch" (as of the moment of this writing, it is current Debian "stable"). Very briefly: it is a pretty typical (even if a bit outdated) real-world "workhorse" server box.*"[14]

The comparison results are available in the following figure:

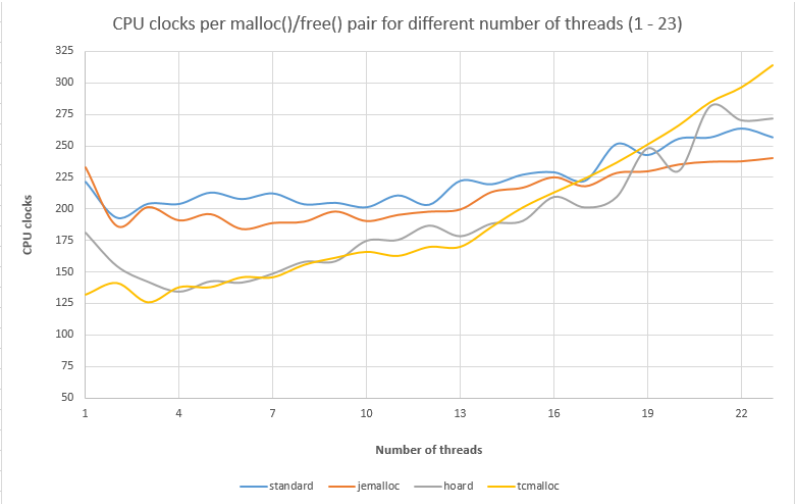

Figure 1: Performance scalability comparison of memory allocators http://ithare.com/testing-memory-allocators-ptmalloc2-tcmalloc-hoard-jemalloc-while-trying-to-simulate-real-world-loads/

[9] jemalloc. http://jemalloc.net/

[10] Firefox jemalloc source code. https://hg.mozilla.org/mozilla-central/file/e2b143e9609f/memory/jemalloc/jemalloc.c

[11] tcmalloc. https://github.com/google/tcmalloc

[12] ptmalloc. http://www.malloc.de/en/

[13] Hoard memory allocator. http://hoard.org/

[14] No Bugs Hare. Testing memory allocators: ptmalloc2 vs tcmalloc vs hoard vs jemalloc while trying to simulate real-world loads. http://ithare.com/testing-memory-allocators-ptmalloc2-tcmalloc-hoard-jemalloc-while-trying-to-simulate-real-world-loads/, July 2018

2.2 SLUB memory allocator

The SLUB allocator is a cousin of a conceptually similar but simpler SLAB allocator in the Linux kernel. SLAB was designed to reflect the fact that allocations commonly concern objects of few common types. It therefore minimizes the allocation time by working with pre-allocated objects with specific sizes, grouping objects of the same size into so-called slabs. The creation of slabs is handled during system initialization, (re)allocation of individual objects is dynamic, the allocator keeps a reserve of recently released objects to satisfy new allocations efficiently.

The main advantage of SLAB is the speed of the allocation. It simplifies the problem of memory fragmentation because many objects have the same size. Nevertheless, with larger systems, the users have complained about slab allocator storage overhead. As the SLAB allocator contributor, Christopher Lamater, said:

"*SLAB Object queues exist per node, per CPU. The alien cache queue even has a queue array that contain a queue for each processor on each node. For very large systems the number of queues and the number of objects that may be caught in those queues grows exponentially. On our systems with 1k nodes / processors we have several gigabytes just tied up for storing references to objects for those queues This does not include the objects that could be on those queues. One fears that the whole memory of the machine could one day be consumed by those queues.* "[15]

[15] Christopher Lameter. The unqueued slab allocator v6. https://lore.kernel.org/linux-mm/20070331193056.1800.68058.sendpatchset@schroeder.engr.sgi.com, Mar 2007

For this reason, Christopher Lamater created the SLUB memory allocator. In this new allocator, memory is divided into pages with objects of a given size. The author removed unnecessary metadata from the slab, except for the free pointer to the next object. Allocation is very fast because free objects are organized into a linked list, so the first object is returned during allocation. Each page stores two critical parameters—the pointer to the head of the freelist and the number of objects in use.

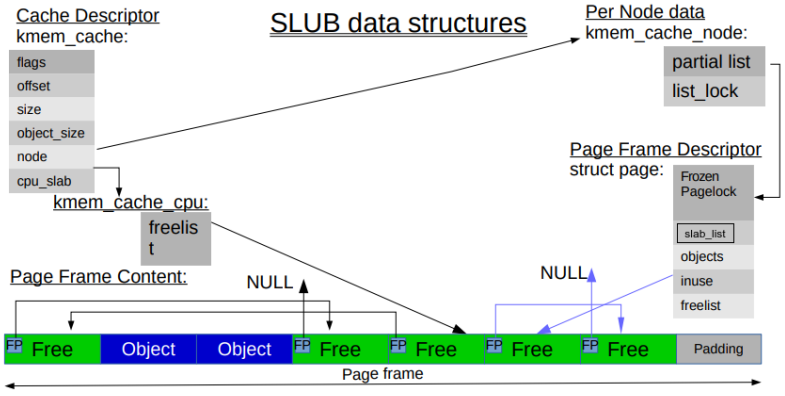

Figure 2: SLUB cache memory structure https://hammertux.github.io/slab-allocator

The SLUB cache structure saves a large set of information. Most important are its size with and without metadata, assigned flags, offset where the pointer to the next free object is stored. Each cache has an array of nodes assigned to it. These nodes have a partial list that contains pages with some free objects. During allocation, each cache takes free objects from its slabs. If cpu slab is full, free objects are taken from partial lists. [16]

[16] For a more detailed description of the allocation algorithm, the article by Matthew Ruffell can help.

2.3 SLUB Debugging

DURING the kernel development, we want to debug the software. The kernel has development tools that help with debugging. For example, developers use sparse for semantic check in C programs or KASAN, which detects memory errors looking for out-of-bound and use-after-free bugs. Also, we can detect a memory leak with a tool called Kernel Memory Leak Detector. Testing methods mentioned in the previous chapter also fall into this category.

In this project, the focus is on memory management, especially on the SLUB allocator. The allocator has some tools that can help with debugging. To enable debugging functionality in the SLUB, we need to set the SLUB_DEBUG option. SLUB also has the SLUB_DEBUG_ON option, which boots the kernel with debugging on by default for all SLUB caches. For this project, SLUB_DEBUG is enough.

Another method of switching debugging on is to add the slub_debug option to the kernel command line. The option alone will have the same effect as a SLUB_DEBUG_ON option. To enable SLUB debugging more specifically, we can give the option some parameters. We can choose from debug options, which the documentation describes: [17]

[17] Slub documentation. https://www.kernel.org/doc/html/latest/vm/slub.html

```
F       Sanity checks on (enables SLAB_DEBUG_
        CONSISTENCY_CHECKS Sorry SLAB legacy issues)
Z       Red zoning
P       Poisoning (object and padding)
U       User tracking (free and alloc)
T       Trace (please only use on single slabs)
A       Enable failslab filter mark for the cache
O       Switch debugging off for caches that would have
        caused higher minimum slab orders
-       Switch all debugging off (useful if the kernel is
        configured with CONFIG_SLUB_DEBUG_ON)
```

We can choose none, one, or more options and enable them in all SLUB caches. Also, we can choose for which cache we want to enable these options. Simply, we add names of slabs separated by commas after options.

Example:

```
slub_debug=ZP,kmalloc-64,kmalloc-32
```

This command will enable Red zoning and Poisoning for kmalloc-64 and kmalloc-32 caches.

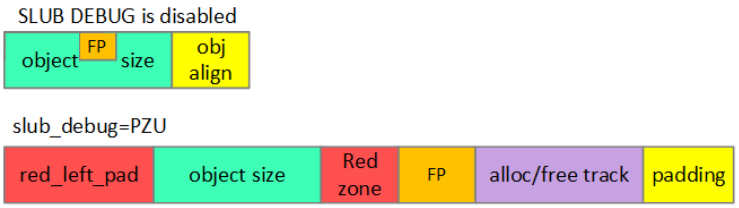

Figure 3: SLUB object structure https://ee-paper.com/working-principle-of-kasan-in-memory-management/

For this project, related options are Red zoning, Poisoning, and User tracking. The first two are used during testing, and the third when improving stack trace storing and creating DebugFS files.

2.3.1 Red zoning

We use the Z option (Red zoning) to detect writing just before or after the object. It adds few extra bytes before and after the object area. For these bytes, we use two different values. If the object is allocated, the value is 0xcc, and in the other case, if it is free, the value is 0xbb. These values have their macros in the kernel called SLUB_RED_ACTIVE and SLUB_RED_INACTIVE. When the program checks the cache and finds different values in these areas, it will report a bug. When creating a new cache, we can enable this option by using the
SLUB_RED_ZONE flag.

2.3.2 Poisoning

We can use the P option (Poisoning) to detect a use-after-free error. With this option, the allocator fills all free objects with bytes with specific values. These bytes have macros called POISON_FREE with value 0x6b, but the last byte has a specific macro POISON_END with value 0xa5. Poisoned is not only the object area but also the slab padding. In this case, the macro POISON_INUSE exists and has a value of 0x5a. When creating a new cache, we can enable this option by using the SLUB_POISON flag. When poisoning the object, a pointer to the next free object is stored after the object area.

2.3.3 User tracking (stack trace)

With debug option U (User tracking) to every object, metadata is added. It stores information about caller address, CPU, PID, a time when the operation occurs, and the stack trace with this option. This option is associated with a flag called SLAB_STORE_USER.

A stack trace is a helpful tool used during debugging. Most frequently, it is used when an exception is thrown. The stack trace is a list of methods called up to some point, for example, when we need to print it.

2.3.4 File systems for debugging

In the Linux kernel, we can use some of its pseudo file systems for debugging. These files can provide information about kernel subsystems, hardware, and other parts of the kernel, to the userspace. With the SLUB debugging, frequently three of them are used. They are proc, sysfs, and debugfs.

All these file systems are virtual. It means that they do not have actual files, only give system information at runtime. They work with kernel data structures. Typically, these files are read-only but also can be writable. It means that we can change some parameters of subsystems during runtime.

THE PROC file system[18] is focused mainly on providing information about processes. Commonly, this file system is mounted to /proc. For the SLUB cache statistics, it provides the slabinfo file.

[18] The /proc filesystem. https://www.kernel.org/doc/html/latest/filesystems/proc.html

Example of file /proc/slabinfo:

```
root@(none):/\# cat /proc/slabinfo
slabinfo - version: 2.1
# name <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit> <batchcount>
<sharedfactor> : slabdata <active_slabs> <num_slabs> <sharedavail>
9p-fcall-cache     0      3   8496    3    8 : tunables    0    0    0 : slabdata      1      1      0
9p-fcall-cache     0     15   8496    3    8 : tunables    0    0    0 : slabdata      5      5      0
fsverity_info      0      0    560   29    4 : tunables    0    0    0 : slabdata      0      0      0
fscrypt_info       0      0    432   18    2 : tunables    0    0    0 : slabdata      0      0      0
zswap_entry        0      0    360   22    2 : tunables    0    0    0 : slabdata      0      0      0
p9_req_t           0     32    504   16    2 : tunables    0    0    0 : slabdata      2      2      0
ip6-frags          0      0    592   27    4 : tunables    0    0    0 : slabdata      0      0      0
fib6_nodes         1     21    384   21    2 : tunables    0    0    0 : slabdata      1      1      0
ip6_dst_cache      0      0    576   28    4 : tunables    0    0    0 : slabdata      0      0      0
ip6_mrt_cache      0      0    512   16    2 : tunables    0    0    0 : slabdata      0      0      0
PINGv6             0      0   1920   17    8 : tunables    0    0    0 : slabdata      0      0      0
RAWv6              8     17   1920   17    8 : tunables    0    0    0 : slabdata      1      1      0
UDPLITEv6          0      0   2112   15    8 : tunables    0    0    0 : slabdata      0      0      0
UDPv6              0      0   2112   15    8 : tunables    0    0    0 : slabdata      0      0      0
```

On each line we can see infomration about one of SLUB caches. It includes statistics: the number of active objects, allocated objects, the objects size, the number of objects in each slab, pages in each slab. Tunables data: the maximum number of objects, the number of objects to transfer at one time on SMP systems, sharedfactor. And the slabdata: number of active slab, total number of slab, sharedavail. The sharedfactor and sharedavail are not documented fields yet.[19]

[19] slabinfo(5)-linux manual page. https://man7.org/linux/man-pages/man5/slabinfo.5.html, Mar 2021

THE MAIN CHARACTERISTIC of the sysfs file system[20] is that it allows only one value per file. This system is a good choice when we want to provide information about the attributes of subsystems.

[20] Mike Murphy Patrick Mochel. sysfs - _the_ filesystem for exporting kernel objectsfs. https://www.kernel.org/doc/html/latest/filesystems/sysfs.html, Jan 2003

Most frequently, it is mounted at /sys/. For SLUB caches, it exposes one directory per unique cache under /sys/kernel/slab.

Example for SLUB cache kmalloc-64:

```
root@(none):/# cd /sys/kernel/slab/kmalloc-64/ && grep . *
aliases:0
align:64
cache_dma:0
cpu_partial:0
cpu_slabs:0
destroy_by_rcu:0
hwcache_align:0
min_partial:5
object_size:64
objects:2648 N0=2648
objects_partial:86 N0=86
objs_per_slab:21
order:1
partial:5 N0=5
poison:0
reclaim_account:0
red_zone:0
remote_node_defrag_ratio:100
sanity_checks:0
slab_size:384
slabs:127 N0=127
slabs_cpu_partial:0(0)
store_user:1
total_objects:2667 N0=2667
trace:0
usersize:64
```

Another virtual file system in the kernel is debugfs[21]. Unlike sysfs, with one value per file rule, or proc, focused on processes, debugfs has no rules for its files. We use it mostly when we need to provide more than single value information to userspace.

For SLUB caches, the implementation of debugfs files was added just recently. It converts some of the sysfs files to debugfs. These files are alloc_traces and free_traces. They should provide information about allocation and freeing of objects. Files are created on location /sys/debug/slab/ for each cache.

[21] Jonathan Corbet. Debugfs. https://www.kernel.org/doc/html/latest/filesystems/debugfs.html, 2009

Example for kmalloc-64:

alloc_traces:

```
1085 populate_error_injection_list+0x97/0x110 age=166678/166680/166682 pid=1 cpus=1
```

free_traces:

```
51 acpi_ut_update_ref_count+0x6a6/0x782 age=236886/237027/237772 pid=1 cpus=1
```

We can see information about the number of uses, allocating/freeing function, minimal/average/maximal jiffies since alloc/free, PID of the process, allocated/freed by CPU.

2.4 SLUB Validation

SLUB HAS A FUNCTION for the validation of its content. The function is called validate_slab_cache(). It takes a cache and iterates through all nodes, pages, and objects in the cache. During the iterations, it checks the values of special bytes. For example, if Poisoning is enabled, the validation checks if in free objects the bytes have values 0x6b and the last one 0x5a. Also, it checks padding bytes or red zones if it is enabled.

When the value of a byte is not correct, it prints a bug report and tries to fix this byte.

Example of a bug report:

```
=============================================================
BUG TestSlub_RZ_alloc (Not tainted): Right Redzone overwritten
-------------------------------------------------------------------------

Disabling lock debugging due to kernel taint
0xffff97674419dcc0-0xffff97674419dcc0 @offset=3264. First byte 0x12 instead of 0xcc
Allocated in test_clobber_zone+0x4b/0x110 age=1 cpu=1 pid=138
 __slab_alloc+0x6d/0x90
 kmem_cache_alloc+0x2e7/0x300
 test_clobber_zone+0x4b/0x110
 kunit_try_run_case+0x56/0x80
 kunit_generic_run_threadfn_adapter+0x1a/0x30
 kthread+0x13c/0x160
 ret_from_fork+0x22/0x30
Slab 0xfffffe320c0106740 objects=26 used=1 fp=0xffff97674419d858 flags=0xfffffc0000201(locked|
slab|node=0|zone=1|lastcpupid=0x1fffff)
Object 0xffff97674419dc80 @offset=3200 fp=0xabf08ab5b9794938

Redzone  ffff97674419dc78: cc cc cc cc cc cc cc cc                         ........
Object   ffff97674419dc80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
Object   ffff97674419dc90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
Object   ffff97674419dca0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
Object   ffff97674419dcb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
Redzone  ffff97674419dcc0: 12 cc cc cc cc cc cc cc                         ........
Padding  ffff97674419dd08: 00 00 00 00 00 00 00 00                         ........
CPU: 1 PID: 138 Comm: kunit_try_catch Tainted: G    B             5.13.0-rc6-next-20210615+ #550
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.10.2-1ubuntu1 04/01/2014
Call Trace:
 dump_stack_lvl+0x85/0xaa
 dump_stack+0x10/0x12
 print_trailer+0x14c/0x155
 check_bytes_and_report.cold+0x70/0x93
 check_object+0x1d9/0x290
 validate_slab+0x144/0x1a0
 validate_slab_cache+0xa8/0x150
 test_clobber_zone+0x5a/0x110
 ? kunit_add_named_resource+0x15c/0x1e0
 kunit_try_run_case+0x56/0x80
 kunit_generic_run_threadfn_adapter+0x1a/0x30
 ? kunit_try_catch_throw+0x20/0x20
 kthread+0x13c/0x160
 ? set_kthread_struct+0x50/0x50
 ret_from_fork+0x22/0x30
FIX TestSlub_RZ_alloc: Restoring Right Redzone 0xffff97674419dcc0-0xffff97674419dcc0=0xcc
```

This bug report consists of two parts. The first includes information about the error, and the second is a fix that was done. On the first line of this report, we can see the SLUB cache name and bug

type. Then the bug description continues. In this case, the information about which byte is incorrect and information about the object are displayed. After that, the content of the object and more information about the object are printed. Also, the call trace is printed to see which functions were called before the error. The last line contains information about fixing the incorrect byte.

Chapter 3: Goals in detail

SLUB has its own set of tests wrapped in the resiliency_test() function. The function contains six tests for different types of errors. The first three are testing bug reporting on allocated objects, and the other three are testing bugs on freed objects. The test principle is that it corrupts specific bytes in memory, and then bugs are expected to be reported by the allocator when the specific cache is validated.

This testing method has few problems that limit the usage and correctness of these tests. First, reading and checking long bug reports, whether the expected bugs occurred, is time-consuming. We could not say quickly if a test failed or passed. Having fast testing can be beneficial because people will use it more often, and the code will be safer.

Second, these tests use normal kmalloc caches, which they share with other components in the kernel. This approach can be dangerous because it can corrupt data of a different process, especially in the test case when a random byte is being corrupted. For example, when a byte inside an allocated object is corrupted, it is not detected. This way, the test can overwrite data for a different process that can harm the kernel.

Moreover, having tests wrapped in the function inside the tested source code is problematic because they cannot run as a module. It means that these tests can only run when the function is called somewhere, which is not happening anywhere, or during the kernel's boot. With the test module, a user can call these tests anytime.

Another problem of these tests, and probably the biggest, is that the whole function is hidden behind #ifdef SLUB_RESILIENCY_−TEST. When we want to use some configuration option, it needs to be added to KConfig files, so everybody can set it on when needed. The SLUB_RESILIENCY_TEST option is not a part of these files, so there is no other way of running these tests than enabling this option by modifying the source code. Nobody is however doing that, and so these tests are never actually run, and thus the SLUB allocator is not being systematically tested.

The proper replacement for the resiliency_test () function and the type of testing used should be KUnit tests. This method of unit testing in the Linux kernel is suitable because we are testing only a small unit of the source code, the SLUB allocator. Also, the

argument for this testing method is that these tests do not require a userspace component. It could become limiting if we wanted to add some more complex new tests, but in such case, we could use the appropriate framework with userspace component only for those tests. In addition, KUnit tests are easy to run. They can be built and run as a module.

ANOTHER POINT of view on these tests is their coverage. How much are these tests covering? Is it enough? Can there be other test cases added?

These original six tests mainly cover the usage of debug flags RED_ZONE and POISON in the SLUB cache and the corruption of the list of free objects. There are many special cases, and it is hard to think about all of them every time something new is implemented or improves some tool's performance. So, for this reason, it would be good to have tests for as many special cases as possible. Nevertheless, how can we accomplish this goal?

One option is adding tests to try to find special cases by going over the source code. Another option is to do it more systematically. For example, create regression tests. As well, tests for other memory allocators may be a good source of new tests. In this project, the two last options are used. Creating regression tests with a systematic review of commits in the Linux kernel concerning the SLUB cache and the exploration of tests for some other allocators.

IN ADDITION, the goal of this project is to improve more SLUB debugging functionalities. The next improvement is associated with a stack trace. The stack trace is stored using an array of the fixed size of 16 items. SLUB_DEBUG is storing two stack traces, alloc and free, for all objects separately. Inside the Linux kernel, these stack traces repeat very frequently. So, instead of saving similar stack traces once, it saves them many times.

The next functionality worth improving is information in file systems used for debugging. Specifically, information displayed in existing debugfs files, alloc_traces, and free_traces. These files can provide developers with more information.

The last improvement in this project is to help using SLUB caches effectively in terms of object size. Kmalloc caches in the kernel have fixed sizes, and when someone want to allocate an object, it uses the first big enough. However, in the kernel, there may be many objects of a different size. So, maybe we can save memory by creating a cache of another popular size.

Chapter 4: Changes discussion

In this chapter, problems and solutions are discussed. The chapter consists of four parts. The first one is focusing on the conversion of existing tests from an unused function to KUnit tests. The second discussed ways of adding new tests for SLUB. The stack trace is analyzed in the third part. The last part explores improvements that can be made in the debugfs file system.

4.1 Integrate existing tests

[22]

[22] Some parts of tests description is based on the soruce code comments which I wrote when adjusting the tests. This documentation was submitted as a patch in linux mailing list.

The first tests to implement in KUnit for the SLUB were these from the resiliency_test () function. The basic idea behind tests stayed untouched, but few changes were needed to make it run as KUnit should.

First of all, we need to find a solution for evaluating tests, if the test passed or not. The solution used here is to count the number of errors, how many bugs and fixes are expected, and then compare it with the expected number of errors. It means that every time some bug should be reported or fixed, the counter for errors is incremented. This solution is not the most accurate one, but it is fast and does not need extra memory.

The error counting is implemented with structure kunit_resource, which is available in KUnit testing. For example, it is a better solution than adding a new field to the cache structure that would increase the size of the structure. It would not be a problem only with cache that the test uses, but every other in memory, too. So, using test API kunit_resource avoids overhead in cache structure. This solution uses only a global variable created in the test code. To count the number of errors during validation, the function called slab_add_kunit_errors() exists. When this function is called, it finds out if it was called from the test, and then it would try to increment the counter or return from the function.

The second bigger problem was with shared caches used in the existing tests. New tests use newly created cache instead of a shared kmalloc cache. If the test uses the kmalloc cache, more than one problem could be found during evaluation, and this type of evaluating test would be incorrect. Also, when the test is modifying free objects, and some other process allocates this memory before the

validation or the cache corruption, it would not find the error in the tested cache. In addition, the test can modify the memory of another subsystem and mess up the kernel. It is the reason why tests create new cache memory. Another advantage is that these caches can turn on only flags which they use for testing. In comparison, previous tests needed to set debug options by default or use the slub_debug option.

The following smaller problems that needed to be solved were the changed implementation of the structure and deterministic character of KUnit tests. The old test, which corrupts the pointer to the next free object, had a wrong computation of the pointer address. This address of the pointer was changed from the start of the object to the middle. It was done to minimize the damage when someone accidentally writes after the object. The offset to the pointer is always saved in the field offset in the structure representing cache. The second problem with determinism was because one of the original tests corrupts a random byte in memory. It has different results every time, and it cannot be evaluated with expected behavior. This test needed to be removed.

After submitting patches with tests and other developers' reviews, the problem on machines with the KASAN configuration option occurred. Because KASAN checks similar things as SLUB_DEBUG, but a little bit differently, some tests did not detect bugs the same way. After adding functions for disabling and enabling KASAN functionality into tests, still, only 2 out of 5 tests passed. It is a better solution to have at least some tests to run with this configuration than none. As Vlastimil Babka pointed out in a discussion about this:

"*I assume that once somebody opts for a full KASAN kernel build, they don't need the SLUB_DEBUG functionality at that point, as KASAN is more extensive (On the other hand SLUB_DEBUG kernels can be (and are) shipped as production distro kernels where specific targeted debugging can be enabled to help find bugs in production with minimal disruption). So, trying to make both cooperate can work only to some extent and for now we've chosen the safer ways.*"[23]

[23] Vlastimil Babka. Re: [patch v4 2/3] mm/slub, kunit: add a kunit test for slub debugging functionality. https://lore.kernel.org/linux-mm/23e27bc2-2f12-d65a-b3ac-8ecb7a37a8c1@suse.cz/, Apr 2021

THE DESCRIPTION of tests integrated from resiliency_test ():

test_clobber_zone

SLUB cache with SLAB_REDZONE flag can detect modifying memory locations after the object. This functionality is tested here on allocated memory. First, it creates the cache with the SLAB_REDZONE flag and allocates the object. Then the first byte after the allocated space is modified. The validation finds two errors. One is that there is a corrupted redzone, and the second due to the repair of the redzone. The bug report of this test should look like this when not silenced:

```
=============================================================
BUG TestSlub_RZ_alloc (Not tainted): Right Redzone overwritten
-------------------------------------------------------------------------------

Disabling lock debugging due to kernel taint
0xffff97674419dcc0-0xffff97674419dcc0 @offset=3264. First byte 0x12 instead of 0xcc
Allocated in test_clobber_zone+0x4b/0x110 age=1 cpu=1 pid=138
 __slab_alloc+0x6d/0x90
 kmem_cache_alloc+0x2e7/0x300
 test_clobber_zone+0x4b/0x110
 kunit_try_run_case+0x56/0x80
 kunit_generic_run_threadfn_adapter+0x1a/0x30
 kthread+0x13c/0x160
 ret_from_fork+0x22/0x30
Slab 0xffffe320c0106740 objects=26 used=1 fp=0xffff97674419d858 flags=0xfffffc0000201(locked|
slab|node=0|zone=1|lastcpupid=0x1fffff)
Object 0xffff97674419dc80 @offset=3200 fp=0xabf08ab5b9794938

Redzone  ffff97674419dc78: cc cc cc cc cc cc cc cc                          ........
Object   ffff97674419dc80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
Object   ffff97674419dc90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
Object   ffff97674419dca0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
Object   ffff97674419dcb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
Redzone  ffff97674419dcc0: 12 cc cc cc cc cc cc cc                          ........
Padding  ffff97674419dd08: 00 00 00 00 00 00 00 00                          ........
CPU: 1 PID: 138 Comm: kunit_try_catch Tainted: G    B             5.13.0-rc6-next-20210615+ #550
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.10.2-1ubuntu1 04/01/2014
Call Trace:
 dump_stack_lvl+0x85/0xaa
 dump_stack+0x10/0x12
 print_trailer+0x14c/0x155
 check_bytes_and_report.cold+0x70/0x93
 check_object+0x1d9/0x290
 validate_slab+0x144/0x1a0
 validate_slab_cache+0xa8/0x150
 test_clobber_zone+0x5a/0x110
 ? kunit_add_named_resource+0x15c/0x1e0
 kunit_try_run_case+0x56/0x80
 kunit_generic_run_threadfn_adapter+0x1a/0x30
 ? kunit_try_catch_throw+0x20/0x20
 kthread+0x13c/0x160
 ? set_kthread_struct+0x50/0x50
 ret_from_fork+0x22/0x30
FIX TestSlub_RZ_alloc: Restoring Right Redzone 0xffff97674419dcc0-0xffff97674419dcc0=0xcc
```

test_next_pointer

SLUB organizes free objects into a linked list, and the address to the next free object is always saved in a free object at the offset specified in variable offset in struct kmem_cache. This test tries to corrupt this freelist and then corrects it.

First, it allocates the memory and frees it to get a pointer to the free object. After that, the pointer to the next free object is corrupted. The first validation finds three errors. The first for a corrupted freechain. The second for the wrong count of objects in use, and the third for fixing the issue. This fix only set the number of objects in use to a number of all objects minus one because the first free object was corrupted.

The first validation should report these bugs and the fix:

```
=============================================================================
BUG TestSlub_next_ptr_free (Tainted: G               B            ): Freechain corrupt
-----------------------------------------------------------------------------
....
=============================================================================
BUG TestSlub_next_ptr_free (Tainted: G               B            ): Wrong object count. Counter is 0 but counted were 29
-----------------------------------------------------------------------------
....
FIX TestSlub_next_ptr_free: Object count adjusted
```

Then the free pointer is fixed to its previous value. The second validation finds two errors—one for the wrong count of objects in use and one for fixing this error.

The second validation should report these bugs and the fix:

```
=============================================================================
BUG TestSlub_next_ptr_free (Tainted: G               B            ): Wrong object count. Counter is 29 but counted were 0
-----------------------------------------------------------------------------
...
FIX TestSlub_next_ptr_free: Object count adjusted
```

The last validation is used to check if all errors were corrected, so no error is found. This validation should report nothing.

test_first_word

SLUB cache with the SLAB_POISON flag can detect corruption of free objects. This test checks this functionality. The test tries to corrupt the first byte in free memory.

First of all, it creates a cache with the SLAB_POISON flag and allocates the object. After that, it frees this object to get a pointer to it, and then the first byte is corrupted. After that, validation finds two errors, one for the bug and the other for the memory fix. Non-silenced bug and the fix reported:

```
=============================================================================
BUG TestSlub_next_ptr_free (Tainted: G               B            ): Poison overwritten
-----------------------------------------------------------------------------
...
FIX TestSlub_1th_word_free: Restoring Poison 0xffff9767443e27f8-0xffff9767443e27f8=0x6b
```

test_clobber_50th_byte

This test checks the SLAB_POISON functionality. The test tries to corrupt the 50th byte in freed memory.

First, it acquires the pointer to free space by allocating and freeing the object from the cache with the SLAB_POISON flag. Then the 50th byte is corrupted, and validation finds two errors for the bug and the memory fix.

Non-silenced bug and the fix reported:

```
=============================================================================
BUG TestSlub_next_ptr_free (Tainted: G               B            ): Poison overwritten
-----------------------------------------------------------------------------
...
FIX TestSlub_50th_word_free: Restoring Poison 0xffff9767443e6c6a-0xffff9767443e6c6a=0x6b
```

test_clobber_redzone_free

This test checks the SLAB_REDZONE functionality of the SLUB cache on a free object.

First, it creates a cache with the SLAB_REDZONE flag and allocates and frees the object to get a pointer to the free object. Then it corrupts the first byte after the free object. Validation finds two errors for the bug and the memory fix.

Non-silenced bug and the fix reported:

```
===============================================================================
BUG TestSlub_next_ptr_free (Tainted: G               B            ): Right Redzone overwritten
-------------------------------------------------------------------------------
...
FIX TestSlub_RZ_free: Restoring Right Redzone 0xffff9767443eb0e0-0xffff9767443eb0e0=0xbb
```

All of these tests are creating some bugs, and all these bugs are reported when they occur. For better readability of test results, all bug reports are silenced. It is achieved with the same function as for counting errors. Because fixes are reported, they are counted, too. When we run these tests, we do not want to read bug reports and look for the test result. Now we have only information whether the test passed or not. It is accomplished by returning bool from the previously mentioned function, slab_add_kunit_errors(). It returns true if a correct KUnit test calls the validation, so bug reports would not be printed.

4.2 New tests

ANOTHER TASK was to improve testing and add some more tests. In the Linux kernel, it is not very typical to test every path in the code. More common tests are focusing on some frequent errors in code. Tests that focus on previous mistakes are called regression tests.

The exploration of commits that were modifying the slub.c file in the Linux kernel tree was the first thing to do. The goal was to identify errors created and fixed during the SLUB's history to expose them with new tests.

This systematic review consists of 3 parts. The first was to look at all of the commits associated with the SLUB allocator, classify them by their properties, and identify the ones fixing some errors. The second part was to decide whether the identified errors are suitable for testing and if some tests can be made to catch them in the future. The last part was to create these tests, so they will help with preventing some regressions of the SLUB implementation.

THE FIRST THING to do was to iterate through all commits and try to get used to the style of their description. The focus was on looking for typical committed changes and finding categories that can be used for their classification. The categories are: merge, performance, feature, style fix, deadlock, memory leak, typos, and fix.

The Merge category includes basic merge commits, which occurred more frequently at the beginning of the SLUB existence when more people were contributing directly to git. These commits are not fixing errors, so no new tests came from here.

The second category, Performance, is for commits that make some smaller changes and improve the allocator's performance. For example, in this category, commits that removed unnecessary checks or changed the order of function calls were included. These commits are also not very beneficial for this project because they fix nothing.

Another type of commit is Feature. This type introduces or implements some new features in SLUB. For example, in recent years, KASAN was one of the more significant features added into the Linux kernel. KUnit tests added in this project can also be considered in this category. Once more, these changes do not fix anything, so no new tests can be made.

The category called Style fix is the biggest portion of the classification. It includes some smaller fixes to the code style or organization of tools and features inside the source code. Examples of changes that fall here: fix build error, rename or remove unused function, variable, file, convert int to unsigned int, make code safer or simplify it, move some feature somewhere else. Interestingly, since 2019 when KASAN was introduced, nine different commits only fixes adding the KASAN tag on a pointer. These commits fix something in the kernel, but these changes are minor and primarily focused on a good style of source code and the correct compilation of the code. So, no tests were created based on them.

Other fixes, which are handled quite commonly in commits are Deadlock errors. These errors occurred because of the wrong locking of objects. It is why they are not a good inspiration for unit tests.

Similarly, as deadlocks are common errors when working with memory, also
Memory leaks occur inside the SLUB cache. These errors are often theoretically uncovered, so they did not even have to happen and need precise settings and circumstances. As well, these errors were not good for testing.

Very often, mistakes fixed with commits are so-called Typos. These have nothing to do with the correct function of the kernel but only correct comments inside the source code. So, these fixes are untestable.

I called the last category Fix. Commits that repair some bigger bugs and look like candidates for tests fall into this section. The next part of the review is dealing with them more closely.

During next iterations through commits, the task was to classify all the commits. The overall number of commits was 929. The results can be seen in Table 1.

As we can see, the most frequent were small Fix style commits. The second most common change was introducing some new features. Also, commits improving performance were relatively frequent. It is a little bit surprising how many typos developers need to correct

Category	Count
Merge	80
Performance	73
Feature	252
Stylefix	420
Deadlock	7
Memoryleak	12
Typos	42
Fix	43

Table 1: Commits distribution in SLUB's history.

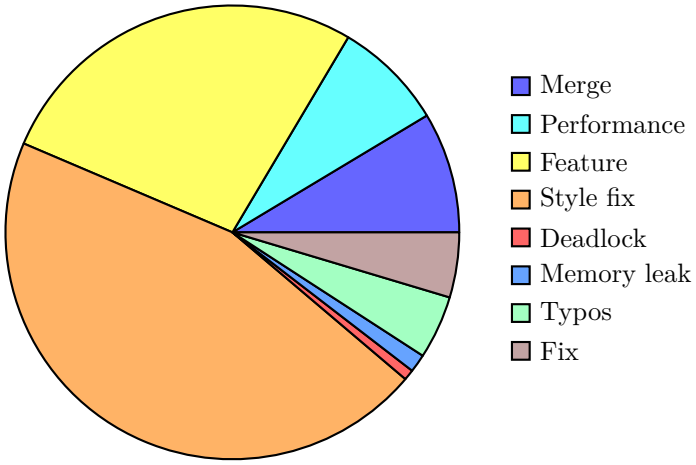

Figure 4: The pie chart for review results.

in commits.

With separated commits, the hunt for new tests started by looking at them and thinking about tests that can detect these errors. These commits were classified into the Fix category. The total number of them is 43 commits. It was pretty hard to find some suitable as a model for a test. Many fixes in this category are not very good for fast unit testing.

One of the common reasons the test cannot be created is that the fix is related to the user interface. For example, the error with the wrong number of slabs in commit '8afb147' (slub: Fix calculation of cpu slabs):

```
/sys/kernel/slab/:t-0000048 # cat cpu_slabs
 231 N0=16 N1=215
 /sys/kernel/slab/:t-0000048 # cat slabs
 145 N0=36 N1=109
```

The second testing method in the kernel, Linux kernel selftests, would be better for this problem.

Another type of problem that authors tried to fix with their commits was some set of abstract cases, for example, race conditions. This error type depends on the currently running software. It does not even have to happen. So, we cannot artificially create it as a test. Race conditions are primarily associated with readings and

writings on multithread processors. The fix for this problem is most frequently adding locks into the code. However, it was not only race conditions but also some abstract scenarios that were very hard to reproduce.

Having a multithread processor is good for performance, but it also can cause many errors. Example of bug report from commit '22e4663' (mm/slub: fix panic in slab_alloc_node()):

```
BUG: Kernel NULL pointer dereference on read at 0x00000007
Faulting instruction address: 0xc000000000456048
Oops: Kernel access of bad area, sig: 11 [#2]
LE PAGE_SIZE=64K MMU=Hash SMP NR_CPUS= 2048 NUMA pSeries
Modules linked in: rpadlpar_io rpaphp
CPU: 160 PID: 1 Comm: systemd Tainted: G        D           5.9.0 #1
NIP:  c000000000456048 LR: c000000000455fd4 CTR: c00000000047b350
REGS: c00006028d1b77a0 TRAP: 0300   Tainted: G        D              (5.9.0)
MSR:  8000000000009033 <SF,EE,ME,IR,DR,RI,LE>  CR: 24004228  XER: 00000000
CFAR: c00000000000f1b0 DAR: 0000000000000007 DSISR: 40000000 IRQMASK: 0
GPR00: c000000000455fd4 c00006028d1b7a30 c000000001bec800 0000000000000000
GPR04: 0000000000000dc0 0000000000000000 00000000000374ef c00007c53df99320
GPR08: 000007c53c980000 0000000000000000 000007c53c980000 0000000000000000
GPR12: 0000000000004400 c00000001e8e4400 0000000000000000 0000000000000f6a
GPR16: 0000000000000000 c000000001c25930 c000000001d62528 00000000000000c1
GPR20: c000000001d62538 c00006be469e9000 0000000fffffffe0 c0000000003c0ff8
GPR24: 0000000000000018 0000000000000000 00000000000000dc0 0000000000000000
GPR28: c00007c513755700 c000000001c236a4 c00007bc4001f800 0000000000000001
NIP [c000000000456048] __kmalloc_node+0x108/0x790
LR [c000000000455fd4] __kmalloc_node+0x94/0x790
Call Trace:
   kvmalloc_node+0x58/0x110
   mem_cgroup_css_online+0x10c/0x270
   online_css+0x48/0xd0
   cgroup_apply_control_enable+0x2c4/0x470
   cgroup_mkdir+0x408/0x5f0
   kernfs_iop_mkdir+0x90/0x100
   vfs_mkdir+0x138/0x250
   do_mkdirat+0x154/0x1c0
   system_call_exception+0xf8/0x200
   system_call_common+0xf0/0x27c
Instruction dump:
  e93e0000 e90d0030 39290008 7cc9402a e94d0030 e93e0000 7ce95214 7f89502a
  2fbc0000 419e0018 41920230 e9270010 <89290007> 7f994800 419e0220 7ee6bb78
```

In this case, with deep analysis, the author of the commit found the problem. This kernel panic was invoked because of a SLUB cache flush. It happens between the loading of a page and an object in the function slab_alloc_node(). So, the object was NULL, while the page was not. We cannot do some deterministic test for this type of error because we do not control the order in which threads are running.

Some commits were fixing some of the features that are no longer available, or their implementation changed, so it no longer works the same way. In one case, the commit fixed stack overrun. It is the type of error that can be caught with tests focused on system exhaustion.

AMONG ALL THESE commits, only five were suitable for tests. Four tests were added to help prevent these errors in the future.

test_slab_padding

This test was based on fixes from commits '8a3d271' (slub: fix slab_pad_check()) and '5d68268' (mm/slub.c: fix wrong address during slab padding restoration). The second one fixes the first one. In these commits, the error is associated with wrong padding restoration in the slab page. So, when padding was corrupted, the fix restored the wrong bytes. It tried to restore bytes inside padding but is also overwrites bytes in the object area.

Within this test, the address of padding is calculated and corrupted. Afterwards, the validation should find two errors. One for the padding corruption and the second for fixing it. The expected bug report, when non-silenced, should have this bug reported:

```
=============================================================================
BUG TestSlub_corrupt_padding (Tainted: G    B           ): Object padding overwritten
-----------------------------------------------------------------------------
...
FIX TestSlub_corrupt_padding: Restoring Object padding 0xffff98b6443ad0a8-0xffff98b6443ad0a8=0x5a
```

test_corrupt_padding

This test is not based on an error fixed in commits, but it is based on the previous one. It checks the corruption of padding inside the object.

First, its cache is created and allocated one object. Then one of the bytes in the page padding is corrupted. The validation finds three errors for a bug, a fix and a padding zone which should be silenced.

Non-silenced bug and the fix reported:

```
=============================================================================
BUG TestSlub_slab_padding (Tainted: G    B           ): Padding overwritten. 0xffff98b64446cfd0-0xffff98b64446cfd0
@offset=4048
-----------------------------------------------------------------------------
...
Padding ffff88c8c4421fd0: ab 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a  .ZZZZZZZZZZZZZZZ
Padding ffff88c8c4421fe0: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a  ZZZZZZZZZZZZZZZZ
Padding ffff88c8c4421ff0: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a  ZZZZZZZZZZZZZZZZ
FIX TestSlub_slab_padding: Restoring slab padding 0xffff98b64446cfd0-0xffff98b64446cfd0=0x5a
```

test_min_partial

The inspiration for this test is from an error fixed in commit '8a5b20a' (slub: fix off by one in a number of slab tests). One of the parameters in the SLUB cache is min_partial. It represents the minimal number of empty partial pages kept in the cache.

First, it creates the cache and allocates the object inside it. Allocation is needed to have at least one page assigned. Then the parameter min_partial is set to 0. After the deallocation of the object, the expected number of partial pages is 0. The test checks this value.

If the test passes, it should report nothing.

test_offset_position

This test is based on commits 'cbfc35a' (mm/slub: fix incorrect interpretation of s->offset) and 'e41a49f' (mm/slub: actually fix freelist pointer vs redzoning). The pointer to the next free object inside the SLUB cache should be saved in the middle of the free object. The offset from the beginning of the object to the pointer is stored in kmem_cache structure parameter offset.

This test checks if the calculation of the offset is correct. It tries to calculate the offset as expected and then compares it with the value in the kmem_cache structure.

If the test passes, it should report nothing.

ANOTHER METHOD used to add tests was looking at tests of other malloc implementations. There are many of them, and only three were picked – ptmalloc, jemalloc, tcmalloc. These implementations are focused on multithreaded and multiprocessors software. So, tests for them are also mainly focused on their performance with more threads, and they are trying to exhaust memory to see how it works. Also, there are some unit tests, but they did not test anything relevant to the SLUB implementation. After all, this method does not bring any new tests.

To sum up, with the help of all of these methods, four new tests were created. This result is not very impressive, but this systematic way is the best solution.

4.3 Stack trace

ANOTHER PART of the SLUB that can be improved is a way of saving a stack trace. The stack trace is implemented using an array of a fixed size. Moreover, SLUB_DEBUG is storing two stack traces, alloc and free, for all objects separately and these tracks are frequently repeating.

For this reason, in the Linux kernel, there exists a tool called stack depot. Its documentation best describes the solution of this tool:

"*Instead, stack depot maintains a hashtable of unique stacktraces. Since alloc and free stacks repeat a lot, we save about 100x space. Stacks are never removed from depot, so we store them contiguously one after another in a contiguos memory allocation.*"[24]

[24] Alexander Potapenko. Stack depot source code. https://github.com/torvalds/linux/blob/master/lib/stackdepot.c, 2016

Now, we need only a handler inside the track structure for the stack trace. Handler's type for stack depot is called depot_stack_handle_t in the kernel.

4.4 Debug files

THE NEXT DEBUGGING method which can be improved for SLUB is the debugfs file system. In the last version, 5.14, two sysfs files were moved to debugfs files. They were alloc_calls and free_calls files. The author of the patch described the reason for this change in the commit description:

"*alloc_calls and free_calls implementation in sysfs have two issues, one is PAGE_SIZE limitation of sysfs and other is it does not adhere to "one value per file" rule.*"[25]

[25] Faiyaz Mohammed. mm: slub: move sysfs slab alloc/free interfaces to debugfs. https://github.com/torvalds/linux/commit/64dd68497be76ab4e237cca06f5324e220d0f050, Jun 2021

During the discussion with Vlastimil Babka, one of the SLUB allocator maintainers, he suggested that it would be beneficial to expand these new files and their information. The original output, for one of the objects, looks like this:

alloc_traces:

```
1085 populate_error_injection_list+0x97/0x110 age=166678/166680/166682 pid=1 cpus=1
```

free_traces:

```
51 acpi_ut_update_ref_count+0x6a6/0x782 age=236886/237027/237772 pid=1 cpus=1
```

We can see information about the number of uses, allocating/freeing function, minimal/average/maximal jiffies since alloc/free, PID of the process, allocated/freed by CPU.

AS WE CAN SEE, the output includes some important information. One of the possible expansions is adding stack trace to the output, too. Another improvement to help with debugging is to aggregate these objects, not only by function address but also by the stack trace. The next improvement in these files was the order of objects sorted by the number of uses. It can help because we can see the most frequently used objects as the first, and we do not need to look for them in endless output.

The call to sorting function sort_r() was added after all information about objects has been collected to achieve this last improvement. It cannot be done during iteration through objects because it uses binary search to find the correct position in the array of objects.

After the changes, the output of debugfs files should look like this:

alloc_traces:

```
1085 populate_error_injection_list+0x97/0x110 age=166678/166680/166682 pid=1 cpus=1
                __slab_alloc+0x6d/0x90
                kmem_cache_alloc_trace+0x2eb/0x300
                populate_error_injection_list+0x97/0x110
                init_error_injection+0x1b/0x71
                do_one_initcall+0x5f/0x2d0
                kernel_init_freeable+0x26f/0x2d7
                kernel_init+0xe/0x118
                ret_from_fork+0x22/0x30
```

free_traces:

```
51 acpi_ut_update_ref_count+0x6a6/0x782 age=236886/237027/237772 pid=1 cpus=1
                kfree+0x2db/0x420
                acpi_ut_update_ref_count+0x6a6/0x782
                acpi_ut_update_object_reference+0x1ad/0x234
                acpi_ut_remove_reference+0x7d/0x84
                acpi_rs_get_prt_method_data+0x97/0xd6
                acpi_get_irq_routing_table+0x82/0xc4
                acpi_pci_irq_find_prt_entry+0x8e/0x2e0
                acpi_pci_irq_lookup+0x3a/0x1e0
                acpi_pci_irq_enable+0x77/0x240
                pcibios_enable_device+0x39/0x40
                do_pci_enable_device.part.0+0x5d/0xe0
                pci_enable_device_flags+0xfc/0x120
                pci_enable_device+0x13/0x20
                virtio_pci_probe+0x9e/0x170
                local_pci_probe+0x48/0x80
                pci_device_probe+0x105/0x1c0
```

The next suggestion for improving debugging was to obtain information about all objects in a particular cache. It means creating a new debugfs file. This new file is called all_objects.

Example of output for one object:

all_objects:

```
Object: 0000000042ee8b00 free
Last allocated: ima_queue_key+0x2f/0x1b0 age=247112 pid=1 cpu=1
        __slab_alloc+0x6d/0x90
        kmem_cache_alloc_trace+0x2eb/0x300
        ima_queue_key+0x2f/0x1b0
        ima_post_key_create_or_update+0x46/0x80
        key_create_or_update+0x383/0x5b0
        load_certificate_list+0x75/0xa0
        load_system_certificate_list+0x2f/0x31
        do_one_initcall+0x5f/0x2d0
        kernel_init_freeable+0x26f/0x2d7
        kernel_init+0xe/0x118
        ret_from_fork+0x22/0x30
Last free: ima_process_queued_keys.part.0+0x84/0xf0 age=170962 pid=137 cpu=1
        kfree+0x2db/0x420
        ima_process_queued_keys.part.0+0x84/0xf0
        ima_keys_handler+0x57/0x60
        process_one_work+0x2a5/0x590
        worker_thread+0x52/0x3f0
        kthread+0x140/0x160
        ret_from_fork+0x22/0x30
```

Description of the output:

a) Object: <address> allocated/free There is an object's address and information on whether the object is allocated or free

b) Last allocated: <address of the caller> age=<jiffies since alloc> pid=<pid of the process> cpu=<allocated by cpu>

c) Alloc stack trace

d) Last free: <address of caller> age=<jiffies since freed> pid=<pid of the process> cpu=<freed by cpu>

e) Free stack trace

The algorithm iterates through all nodes, their pages, and finally their objects in the selected SLUB cache. It uses the seq_file interface. This interface helps with iteration through the object for virtual file systems. This solution does not create the snapshot at some given moment because it would need to hold page locks for too long. In this solution, a lock is acquired only to obtain a bitmap for the page. After that, the page is unlocked. The whole cache is not printed at the exact moment, so information can change during iteration. This solution is the best effort, which tries to minimize holding pages locked.

THE LAST IMPROVEMENT in SLUB debugging added in this project is sorting objects by their sizes. This improvement is beneficial only with kmalloc caches. However, because we cannot easily determine if a cache is kmalloc at such a low level, the solution involves all caches. So, a new field was added to the struct track, where the original size of data should be stored. After that, the aggregation in alloc_traces and free_traces needs to aggregate by the object's original size, too. With this improvement, we can see the size of the most frequent caches.

For example, most frequently used object in kmalloc-64:

```
1083 populate_error_injection_list+0x97/0x110 age=44473/44475/44478 pid=1 cpus=0 object_size=48
```

The most frequent object has a size of 48 bytes. So, in each of these objects are 6 bytes unused in memory. It is not much, but in bigger systems, it can sum up to much larger overheads, and it might be beneficial to create a kmalloc-48 cache.

Chapter 5: Implementation overview

The project assignment was divided into three parts. All of them are trying to improve some of the SLUB cache debugging tools. The first should add KUnit tests for the SLUB cache. The second task should improve the implementation of the stack trace, and the third debug information obtained with debugfs files.

The solution for this project is in the form of patches that we can apply to the Linux kernel main tree. So, they follow the rules introduced in chapter 1.3, Submitting patches. The solution for each part contains small patches. Some of these patches were already submitted, reviewed, and tested by other maintainers and added to the Linux kernel.

The whole solution consists of 832 line changes to the Linux kernel - 708 insertions and 124 deletions. Changes are made in eight files.

KUnit

This part consists of 4 commits. Three of them were created as a series that changed the implementation of existing resiliency tests. The first patch change locks used in the testing interface. The second one creates a KUnit testing interface for SLUB cache and implements tests that already existed as resiliency tests. This patch added five tests. The third patch from the series removes the unused resiliency_test () function from the source code.

The overall number of line changes in the kernel done by this series was 302. Most of them are done with the creation of a new file with tests.

The series was already submitted and went through iteration with maintainers. It ended up with the fifth version. This version made its way into the mainline tree.

During the creation of this series, one error in the SLUB showed up. It was not precisely with running tests but the cache's creation. Debug flags in the cache did not work. These flags were only working when at least one of the caches in the memory have enabled some slub_debug options on the kernel command line. Interestingly, the cache with a given name in the slub_debug option did not even need to exist. [26]

[26] The solution for this problem is in commit '1f0723a' in the Linux kernel tree.

The remaining fourth commit from this part of the solution adds new tests created during the systematic review. This patch added four new tests. The number of line changes done with it is 129.

Tests added with the review were checked that they detect errors using commit revert. We can revert the fix commit that inspired it and is in the comment before the test. Then the test does not pass.

The tests mainly check bug reporting in SLUB with the corruption of specified byte. Also, they catch errors in basic functionality. For example, the cache is created and destroyed. The object is also allocated and freed in all tests. So, tests can detect errors in these parts of code.

Stack trace

THE STACK TRACE improvement is implemented in a single commit. This commit added 80 line changes in the kernel.

The solution of this part was also submitted and accepted by the maintainers. It was already committed to the mainline tree.

The solution needed two fixes which other maintainers revealed. They run the kernel on different platforms with different configurations. One problem was the collision in the name of the function for saving the stack trace. The function name save_stack_trace exists in the kernel without the config option CONFIG_ARCH_STACK−WALK. The second was with the type of cache created in stack depot implementation. The GFP_KERNEL type could be unsafe in some contexts, so it needs to use the GFP_NOWAIT type. The difference between these two types is that the allocation of the second would not sleep.

Debugfs

THE ADDITION of information in debugfs files is done in four steps, each in a separate commit. One adds the stack trace as a parameter in the aggregation of objects. The second changes the order of objects by sorting them by size. Another one adds a debugfs file with information about all objects. The last patch implements storing of original object size and aggregation by it in debugfs files.

The overall number of line changes in this part is 321. The implementation of the all_objects file was the largest.

The patches from this part of the solution were submitted only as RFC (Ready for comment) because patches with debugfs files for SLUB were not part of any Linux development tree. This RFC was reviewed only by the supervisor of this project and SLUB co-maintainer, Vlastimil Babka, so far.

Conclusion

As the thesis title says, the focus of this project was on improving SLUB allocator debugging functionality. It adds new ways how to obtain information that can help with debugging this allocator.

The first part implements the KUnit testing interface and adds new tests. Five tests are taken from the existing but unused tests in the SLUB implementation and check some debugging options and bug reporting. For example, use-after-free, write after object errors, or corruption of freelist. In tests, a specific byte in the memory is corrupted and checked if a memory validation finds the expected number of bugs and fixes. Some of the tests checks behavior in allocated objects and some in free objects.

The remaining four tests were created as a result of a systematic review of commits. These tests catch bugs that occur in the past and help to avoid them in the future. They focus on SLUB functionalities like a minimal number of partial pages in cache or an offset of pointer to the next object.

The following debug tool worth improving was the stack trace storage. The previous implementation used an array for each object to store the stack trace. However, many stack traces are similar, so the contents repeated frequently. For this reason, the kernel provides a structure called stack depot. It saves repeating stack traces only once. So, this project changes storing of the stack trace from the array to the much smaller stack depot handle.

Another improvement this project adds is the expansion of information in debugfs files. It adds the stack traces to aggregated object information and sorts them on output by frequency. So, the objects allocated with the most frequent allocation stack trace are reported as the first and can be easily found. Also, a new file was created. It prints information about all objects in the selected cache, without aggreggation.

The last helpful thing is checking is the efficiency of the SLUB cache sizes. It concerns shared kmalloc caches. The information about the original size of an allocated object is stored. It can help detect which object sizes are commonly used, but a kmalloc cache does not exist for this exact size. If such size was added, it could save some memory.

The solution for this project has a form of patches, which were

submitted into Linux kernel. KUnit tests from the original test function and implementation of stack trace already made their way into the mainline Linux kernel tree. Other parts of the project are ready to be or already submitted but not yet included in the mainline tree.

Bibliography

Firefox jemalloc source code. https://hg.mozilla.org/mozilla-central/file/e2b143e9609f/memory/jemalloc/jemalloc.c.

Hoard memory allocator. http://hoard.org/.

jemalloc. http://jemalloc.net/.

Linux kernel selftests. https://www.kernel.org/doc/html/latest/dev-tools/kselftest.html.

Kunit. https://kunit.dev/.

The /proc filesystem. https://www.kernel.org/doc/html/latest/filesystems/proc.html.

ptmalloc. http://www.malloc.de/en/.

Slub documentation. https://www.kernel.org/doc/html/latest/vm/slub.html.

tcmalloc. https://github.com/google/tcmalloc.

slabinfo(5)-linux manual page. https://man7.org/linux/man-pages/man5/slabinfo.5.html, Mar 2021.

Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Aug, 2018.

Vlastimil Babka. Re: [patch v4 2/3] mm/slub, kunit: add a kunit test for slub debugging functionality. https://lore.kernel.org/linux-mm/23e27bc2-2f12-d65a-b3ac-8ecb7a37a8c1@suse.cz/, Apr 2021.

Jonathan Corbet. Debugfs. https://www.kernel.org/doc/html/latest/filesystems/debugfs.html, 2009.

No Bugs Hare. Testing memory allocators: ptmalloc2 vs tcmalloc vs hoard vs jemalloc while trying to simulate real-world loads. http://ithare.com/testing-memory-allocators-ptmalloc2-tcmalloc-hoard-jemalloc-while-trying-to-simulate-real-world-loads/, July 2018.

Anton Ivanov. Uml howto v2. https://www.kernel.org/doc/html/latest/virt/uml/user_mode_linux_howto_v2.html, Sep 2020.

Christopher Lameter. The unqueued slab allocator v6. https://lore.kernel.org/linux-mm/20070331193056.1800.68058.sendpatchset@schroedinger.engr.sgi.com, Mar 2007.

Andrew Lutomirski. Virtme documentation. https://github.com/amluto/virtme, Feb 2014.

Faiyaz Mohammed. mm: slub: move sysfs slab alloc/free interfaces to debugfs. https://github.com/torvalds/linux/commit/64dd68497be76ab4e237cca06f5324e220d0f050, Jun 2021.

Mike Murphy Patrick Mochel. sysfs - _the_ filesystem for exporting kernel objectsfs. https://www.kernel.org/doc/html/latest/filesystems/sysfs.html, Jan 2003.

Alexander Potapenko. Stack depot source code. https://github.com/torvalds/linux/blob/master/lib/stackdepot.c, 2016.

List of Figures

List of Tables

Attachments

First Attachment: How to test the thesis results

BECAUSE THE THESIS involves changes in the Linux kernel, here is a quick tutorial on building, booting the kernel, and testing the job. The tutorial should work on most standard Linux distros. Packages needed for building the kernel are flex, bison, openssl, libelf-dev.

THE FIRST THING is to download the current Linux kernel tree and checkout to the Linux version 5.14-rc2, on which all patches are based.

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
cd linux
git checkout v5.14-rc2
```

From the thesis appendix in the zip file form, we need a .config file and the repository with patches. The best way is to copy these files to the linux repository.

```
cp -r <zip_file>/Task0* <zip_file>/.config .
```

This project is in the form of patches that are based on Linux kernel version 5.14-rc2. Patches are divided into three repositories. From the Task01 repository, with kernel version at least 5.14, we only need to apply patch 0001-slub-kunit-Add-new-tests-for-SLUB.patch. The other three patches should be part of the kernel already—the rest of the patches we need to apply.

```
git am Task01/0001-slub-kunit-Add-new-tests-for-SLUB.patch
git am Task02/0001-mm-slub-use-stackdepot-to-save-stack-trace-in-object.patch
git am Task03/v2-*.patch
git am Task03/0001-mm-slub-keep-track-of-original-object-size.patch
```

Now, we need to build the kernel. We can use parallel make, for example, with 8 processors:

```
make -j8
```

FOR DEBUGGING the kernel is beneficial to use virtualization—for example, a virtme tool. Virtme can be installed as a package in most distributions. If the package is not available, it can be installed from the github repository:

```
cd ..
git clone https://github.com/amluto/virtme
cd virtme
sudo ./setup.py install
cd ../linux
```

With the virtme tool, we can start a virtual machine:

```
virtme-run --mods=auto --kdir . --memory 2G --qemu-opts --smp 4
```

KUnit tests

We can choose how we want to run KUnit tests. To run tests during kernel boot, we need to set config option SLUB_KUNIT_TEST to y. With this option, we will see results on the boot log. Another option to run tests is to build it as a module and run it after the boot. For this, we need to set the config option to M. The test module can be executed with the command after the kernel boot:

```
modprobe slub_kunit
```

If all tests succeed, the output should be this:

```
# Subtest: slub_test
    1..9
    ok 1 - test_clobber_zone
    ok 2 - test_next_pointer
    ok 3 - test_first_word
    ok 4 - test_clobber_50th_byte
    ok 5 - test_offset_position
    ok 6 - test_clobber_redzone_free
    ok 7 - test_corrupt_padding
    ok 8 - test_slab_padding
    ok 9 - test_min_partial
ok 1 - slub_test
```

DebugFS

This part needs one more option set on. It is User tracking. We can do that by adding parameter (-a slub_debug=,U) to the command to run VM. The script for running a vitrual machine would be:

```
virtme-run --mods=auto --kdir . -a slub_debug=U --memory 2G --qemu-opts --smp 4
```

Debugfs files are mounted at /sys/kernel/debug/slab. Files for all SLUB caches are here. We can choose one of them. For example, to see files content of kmalloc-64 cache, we use commands after the kernel boot:

```
cd sys/kernel/debug/slab/kmalloc-64
grep .  alloc_traces
grep .  free_traces
grep .  all_objects
```

Live USB

When using other operating systems than Linux, the easiest way to start Linux is probably to use a Live USB. The Live USB contains a bootable operating system. All we need is a USB with a size of at least 4GB.

The tutorial on how to create a bootable Ubuntu USB stick can be found at this address https://ubuntu.com/tutorials/create-a-usb-stick-on-windows.