LF live

MENTORSHIP SERIES

LF live MENTORSHIP SERIES

Debugging Linux Memory Management Subsystem

Khalid Aziz
Consulting Linux Kernel Engineer
Oracle

Agenda

- Look at a real customer problem
- Parts of mm that play a part in the problem
- Where is the problem
- How to solve it
- Tools available for mm and general kernel debugging

A Developer’s Investigative Toolbox

- When something does not work right, how do you determine where the problem is.
- First understand where to look by doing a failure analysis. Analysis helps determine the potential code paths in the failure.
- Add dynamic observability to the suspect code path: instrument the code (printk?), kernel debugger, trace points, add counters for events, use existing stats reported by kernel
- The big question is can the failure be reproduced on a test system? If not, what level of access do you have to the failing system?
- Access to customer systems tends to be extremely limited. If failure can not be reproduced on a test system, dynamic observability may not be an option.
- With lack of reproducible failure and no access to failing system, investigation depends upon being able to get static information from the failing system

Customer Problem

- Oracle X8-2 server, 96 cores, 256GB memory, Oracle enterprise kernel 5.4.17-2136.314.6
- System is kexec rebooted periodically
- After 90-100 kexec reboots, system fails to boot up with “Out of memory” messages.
- A system with 256GB memory that has booted up 100s of times successfully failing to boot because it ran out of memory is very odd

Stack Trace

- Stack trace on console looks like:

```
Out of memory: Killed process 1829 (dbus-daemon) total-vm:76664kB, anon-rss:0kB, file-rss:0kB, shmem-rss:0kB,
UID:81 pgtables:156kB oom_score_adj:-900
Kernel panic - not syncing: System is deadlocked on memory
CPU: 7 PID: 2066 Comm: kworker/u193:6 Tainted: G   OE    5.4.17-2136.314.6.el8uek.x86_64 #2
Hardware name: Oracle Corporation ORACLE SERVER X8-2/ASM,MB,X8-2, BIOS 51050300 07/01/2021
Call Trace:
 dump_stack+0x6d/0x8f
 panic+0x101/0x2f9
 out_of_memory.cold.38+0x5e/0x7e
 __alloc_pages_slowpath+0xd94/0xe78
 __alloc_pages_nodemask+0x2e3/0x32a
 alloc_pages_current+0x85/0xe2
 __get_free_pages+0x11/0x3a
 pgd_alloc+0x3a/0x233
 mm_init+0x1ab/0x295
 mm_alloc+0x4e/0x5a
………..snip………..
```

Digging through console log

- Kernel panics early on soon after root filesystem is mounted and systemd has been started. No login prompt yet.
- Without a login prompt and limited access to a customer system, console log is the only reasonable option to aid debugging.
- Command line in console log does not show any incorrect options that could cause this issue -
```
BOOT_IMAGE=(hd0,gpt2)/vmlinuz-5.4.17-2136.314.6.el8uek.x86_64 root=/dev/mapper/vg_lab77-lv_root ro
console=ttyS0,115200 crashkernel=256M@64M resume=/dev/mapper/vg_lab77-lv_swap
rd.lvm.lv=vg_lab77/lv_root rd.lvm.lv=vg_lab77/lv_swap biosdevname=1 net.ifnames=1
console=ttyS0,115200n8 pci=noaer pciehp.pciehp_surprise=1
```
- Console log shows “`total RAM covered: 262080M`” which verifies all 256GB of memory was discovered by the kernel

Memory information in console log

- What are possible points of failure that could cause out of memory condition:
  - Failure in physical memory modules
  - Kernel failed to detect all physical memory
  - Kernel fails to add all physical memory to its memory map
- Cold reboot restores system to working state and all 256GB of memory becomes available. That rules out physical memory module failure
- When kernel panics with out of memory condition, it still reports having discovered 256 GB of memory. That verifies kernel does detect all of memory

Checking physical memory detection

- Kernel maintains a list of all memory ranges available. As it discovers physical memory ranges, it adds them to its list of memory it is managing.
- Physical memory is managed in units of pages. Page size is dependent upon processor. architecture. Some processors can support multiple page sizes. In this specific case, we are working with x86-64 processor which has a base page size of 4K.
- Looking through the console log, we find the message "`last_pfn = 0x70000 max_arch_pfn = 0x400000000`". PFN stands for Physical Frame Number. Physical memory is divided into page frames. Depending upon how much of physical memory is installed, a page frame may or may not hold a physical page. Most often, a processor will have holes in physical page ranges. The last populated physical frame kernel found is 458752 which is at 1.75GB. Comparing that to a successful boot, last PFN on a good boot is 0x4080000 which is 258GB

Did kernel fail to add all physical memory?

- After kernel has initialized the memory range it is going to manage, it prints this range to console.
- Console log from a good bootup shows:

```
Initmem setup node 0 [mem 0x0000000000001000-0x000000207fffffff]
Initmem setup node 1 [mem 0x0000002080000000-0x000000407fffffff]
```

- Console log from a failed bootup shows:

```
Initmem setup node 0 [mem 0x0000000000001000-0x000000006fffffff]
Initmem setup node 1 [mem 0x0000000000000000-0x0000000000000000]
```

- Here we can see kernel’s view of memory is quite different between the good and failed cases. In the good case, kernel sees 128GB of memory each on nodes 0 and 1. In failed case, it only sees 1.75GB of memory on node 0 and no memory on node 1.
- With just 1.75GB of memory, kernel runs out of memory quickly as it attempts to start services that require significant amount of memory.

How does kernel detect physical memory?

- Firmware detects installed physical memory modules and their mappings in physical address range. It provides this information to the kernel.
- Kernel parses the physical memory map from firmware, sanitizes it and builds a final memory map for use by kernel.
- On x86-64 architecture, bootloader builds a boot parameters page (called the “zeropage”). It reads firmware provided memory map and populates an e820 table as part of zeropage. Structure of zeropage is defined as `struct boot_params` in `arch/x86/include/uapi/asm/bootparam.h` Each entry in e820 table is a starting physical address of memory range, size of the range and memory type. `e820_entries` in `struct boot_params` gives the number of entries in e820 table.
- Zeropage holds the first 128 memory entries. Remaining entries, if any, continue in a SETUP_E820_EXT node in `struct setup_data`

Kernel’s View of Memory Ranges

- Kernel prints the e820 memory map to the console at bootup. If extended memory ranges are present, those are printed as well preceded by “extended physical RAM map:” message. This memory map is also available after bootup as `/sys/firmware/memmap:`

```
# cat /sys/firmware/memmap/0/start
0x0
# cat /sys/firmware/memmap/0/end
0x9efff
# cat /sys/firmware/memmap/0/type
System RAM
```

Example e820 Memory Map

```
BIOS-provided physical RAM map:
BIOS-e820: [mem 0x0000000000000000-0x000000000009efff] usable
BIOS-e820: [mem 0x000000000009f000-0x00000000000fffff] reserved
BIOS-e820: [mem 0x0000000000100000-0x000000005f6c1fff] usable
BIOS-e820: [mem 0x000000005f6c2000-0x0000000063510fff] reserved
BIOS-e820: [mem 0x0000000063511000-0x0000000063d71fff] ACPI NVS
BIOS-e820: [mem 0x0000000063d72000-0x0000000063ffefff] ACPI data
BIOS-e820: [mem 0x0000000063fff000-0x0000000063ffffff] usable
BIOS-e820: [mem 0x0000000064000000-0x0000000067ffffff] reserved
BIOS-e820: [mem 0x0000000069400000-0x00000000695fffff] reserved
BIOS-e820: [mem 0x0000000069e00000-0x00000000707fffff] reserved
BIOS-e820: [mem 0x00000000c0000000-0x00000000cfffffff] reserved
BIOS-e820: [mem 0x00000000fed20000-0x00000000fed7ffff] reserved
BIOS-e820: [mem 0x00000000ff000000-0x00000000ffffffff] reserved
BIOS-e820: [mem 0x0000000100000000-0x000000088f7fffff] usable
```

Memory Map parsing

- Memory setup is done in `setup_arch()` defined in `arch/x86/kernel/setup.c`
- `e820__memory_setup()` does the initial parsing and sanitization of e820 memory map. It calls `e820__update_table()` to remove inconsistent, empty and overlapping memory ranges
- `e820__memory_setup_extended()` parses `struct setup_data` for SETUP_E820_EXT node and adds the memory entries it finds to e820 table.
- Kernel uses memblock early allocator. Once e820 memory ranges have been cleaned up, they are added to memblock allocator by a call to `e820__memblock_setup()` in `setup_arch()`. These memory ranges determine how much physical memory is available on the system and how it maps on the address map for the processor
- `/proc/iomem` shows the current physical memory map on the system
- For NUMA systems as part of initialization, kernel calls `numa_init()` to determine which memory ranges belong to each NUMA node on the system. This function uses either ACPI SRAT (System Resource Allocation Table) or northbridge to extract NUMA locality for memory.

What is kexec?

- Normal reboot sequence for a system is:

  kernel → reset → firmware → bootloader → kernel

- Each of these steps takes time to execute. Firmware can take significant amount of time to initialize all hardware for a complex system. Bootloader can add a little bit of time to this sequence as well.
- Kexec is a mechanism to shorten this reboot sequence.
- Kexec acts as a bootloader in the kernel that can load a kernel in memory, set up boot parameters and transfer control to the kernel it loaded upon system shutdown.
- Reboot sequence with kexec becomes:

  Kernel → kexec load → new kernel

Requirements from kexec

- Place system into the same state as it would be in upon handoff from firmware to bootloader
- Prepare the zeropage for the next kernel that would normally be done by the bootloader
- Zeropage must be populated with kernel command line parameters and e820 table for memory map
- Kernel builds three versions of e820 table from the table it gets from firmware:

  - `e820_table_firmware`: this is the original e820 table from firmware

  - `e820_table_kexec`: this is mostly a copy of `e820_table_firmware` with possible slight modifications

  - `e820_table`: Sanitized and cleaned up table used by currently running kernel to set up physical memory map

So where did memory go?

- Looking at the dmesg from failed kernel boot again, the message “`total RAM covered: 262080M`” indicates kernel could see all 256GB of RAM
- Customer had saved dmesg log from the failed kexec reboot as well as a few successful kexec reboots just before that. Customer had cycled power after last failed reboot and was able to provide dmesg log from boot up after power cycle as well. All of the logs showed kernel saw 256 GB of memory
- Looking at the BIOS e820 map as reported by the kernel, both spanned the same memory range but there was one significant difference between e820 map immediately after a power cycle and e820 map from the failed kexec reboot – After a power cycle, there were 15 entries in the map while the failed kexec reboot had 128 entries. This is odd and the number 128 is suspicious in that the maximum number of e820 memory entries on zeropage is 128

E820 memory ranges in dmesg

- From the dmesg log from boot up after power cycle, the last two e820 entries are:

```
BIOS-e820: [mem 0x00000000ff000000-0x00000000ffffffff] reserved

BIOS-e820: [mem 0x0000000100000000-0x000000407fffffff] usable
```

- From the dmesg log from failed kexec reboot, last e820 entry is:

```
BIOS-e820: [mem 0x00000000ff000000-0x00000000ffffffff] reserved
```

- The entry `0x0000000100000000-0x000000407fffffff` represents bulk of the RAM, 254GB. This entry would be entry #129 which has to be passed in the SETUP_E820_EXT node of setup data

- `setup_e820_entries()` sets up e820 table in zeropage for kexec kernel. This function has the comment - “TODO: Pass entries more than E820_MAX_ENTRIES_ZEROPAGE in bootparams setup data”. That explains why the entry #129 disappeared for kexec’d kernel!

The case of disappearing memory

- When the last e820 entry for the 254GB of memory is not passed to kexec’d kernel, it is working with just 1.75GB of memory which is not enough to start all the service system is configured to run.
- System starts out with just 15 entries in e820 table but has 129 entries after large number of kexecs.
- Comparing two successive kexec reboots, the number of e820 entries jumps by 2. This causes number of e820 entries to go from 15 to 129 with consecutive kexec reboots.
- Comparing e820 entries from two successive reboots, one of the entries is split every time into three entries. Following entry:

```
BIOS-e820: [mem 0x0000000000040000-0x000000000009ffff] usable
```

gets split into:

```
BIOS-e820: [mem 0x0000000000040000-0x000000000009e30f] usable

BIOS-e820: [mem 0x000000000009e310-0x000000000009e37f] usable

BIOS-e820: [mem 0x000000000009e380-0x000000000009ffff] usable
```

The case of disappearing memory contd.

- We start with 15 entries in e820 table and each kexec splits one of the entries into three. That means we add two entries every time kexec reboot happens. On 57th kexec reboot, e820 table goes from 127 to 129 entries. This triggers the bug/missing functionality in kexec that it does not set up an extended node for additional e820 table entries and 129th entry gets dropped.
- Next mystery is why is an e820 entry getting split into three every time kexec happens? To figure that out, we look at the function that can be used to split an e820 entry which is `__e820__range_update()`
- `__e820__range_update()` is called by `e820__range_update()` and `e820__range_update_kexec()`. Since it is the kexec copy of e820 table that gets passed to kexec'd kernel, `e820__range_update_kexec()` looks promising. Looking at the callers for this function, `e820__memblock_alloc_rese()` can be ruled out since it is used for faking an mptable which does not apply in this case.

Debugging splitting e820 Entry

- The other caller to `e820__range_update_kexec()` is `e820__reserve_setup_data()` which reserves memory ranges for setup data by looking at setup data passed in zeropage and splits e820 entries in kexec e820 table
- zeropage for kexec'd kernel was prepared by `setup_boot_parameters()` which builds and adds EFI setup data to zeropage. This means the EFI setup data given to a kernel can be discarded after processing and there is no need to keep the memory occupied by this data as reserved memory. Could `e820__reserve_setup_data()` be reserving memory range for data that will be synthesized again?
- Since this is an Oracle kernel based upon an older upstream kernel, it is time to check what changes may have been made to this function in upstream kernel

Confirming the problem

- Oracle kernel is based on upstream 5.4.17 kernel and `e820__reserve_setup_data()` iterates over `setup_data` reserving memory range for all setup data it finds.
- Upstream kernel at this time was 5.18 and there indeed had been a change in `e820__reserve_setup_data()` where unconditional call to `e820__range_update_kexec()` had been replaced with:

```
/*
 * SETUP_EFI is supplied by kexec and does not need to be
 * reserved.
 */
if (data->type != SETUP_EFI)
       e820__range_update_kexec(pa_data, sizeof(*data) + data->len,
                                 E820_TYPE_RAM, E820_TYPE_RESERVED_KERN);
```

- That confirms earlier suspicion about not needing to reserve memory range for EFI setup data

Fixing the problem

- The code change of interest was made by commit 8efbc518b884 “x86/kexec: Do not reserve EFI setup_data in the kexec e820 table”
- Commit log for this commit says:

```
SETUP_EFI types, however, are used by kexec itself to enable EFI in the
2nd kernel. Thus, it is pointless to add this type of setup_data to the
kexec e820 table as reserved.

IOW, what happens is this:

  -  1st physical boot: no SETUP_EFI.

  - kexec loads a new kernel and prepares a SETUP_EFI setup_data blob, then
  reboots the machine.

  - 2nd kernel sees SETUP_EFI, reserves it both in the e820 and in the
  kexec e820 table.

  - If another kexec load is executed, it prepares a new SETUP_EFI blob and
  then reboots the machine into the new kernel.

  5. The 3rd kexec-ed kernel has two SETUP_EFI ranges reserved. And so on...
```

- This sounds very much like the problem we are looking at. Backport this commit as fix.

Information available from Kernel

- Kernel makes significant amount of information available to help with debugging functional and performance problems
- Kernel information is available through procfs (mounted on `/proc`), sysfs (mounted on `/sys`) and debugfs (mounted on `/sys/kernel/debug`).
- Various files in these filesystems contain counters for events and objects that can provide insight into current state of mm subsystem
- Documentation in Documentation/filesystems/proc.txt, Documentation/filesystems/sysfs.txt and Documentation/admin-guide/mm
- Information in these files is updated dynamically

/proc/meminfo

- Data is populated by `meminfo_proc_show()` in `fs/proc/meminfo.c`
- Interesting counters:

  MemFree: Memory available on free list

  MemAvailable: Memory that can be made available through reclamation

  Cached: Memory consumed by pagecache

  Mlocked: Pages locked into memory through `mlock()` call

  Slab: Memory consumed by slab objects

  HugePages_Total: Memory pre-allocated for hugepages

/proc/vmstat

- Data is populated by `vmstat_show()` in `mm/vmstat.c`
- Hint: To find the counters shown in this file, change the name in `/proc/vmstat` to all capital letters and search in source code
- Interesting counters (too many to mention):

  allocstall_*: stalls doing direct reclamation

  compact_*: counters for various compaction events

  drop_pagecache: number of times pagecache was dropped through write to `/proc/sys/vm/drop_caches`

  thp_*: counters for Transparent HugePages (THP) related events

/proc/zoneinfo

- Data is populated by `zoneinfo_show()` in `mm/vmstat.c`
- Similar information as in `/proc/meminfo` but broken down by memory zones. **Important:** Unit for counters is pages
- Interesting counters:

  min, low, high: per-zone watermarks

  managed: number of managed pages in the zone

  cma: pages reserved for CMA allocations

Other interesting files

- /proc/iomem: current memory map for all addresses on the system
- /proc/kpagecount: binary file with an array of u64 representing number of mappings of each physical page, populated by `kpagecount_read()` in `fs/proc/page.c`
- /proc/kpageflags: binary file with an arracy of u64 representing flags for each physical page
- /proc/pagetypeinfo: count of pages of different types in each zone on each numa node
- /proc/slabinfo: memory use by various slab objects
- /proc/vmallocinfo: Shows vmalloced areas. Details in `Documentation/filesystems/proc.rst`
- /sys/kernel/debug/extfrag/extfrag_index: current external fragmentation for each zone. Look up extfrag_threshold in `Documentation/admin-guide/sysctl/vm.rst` for interpretation of values

Some debug tools

- ftrace: used for tracing events inside kernel. Kernel has a large number of event points that can be enabled dynamically without rebuilding the kernel. See Documentation/trace/ftrace.rst for details
- kprobetrace: kprobe based event tracer. This is more versatile than ftrace since it supports adding and removing probes on the fly. See Documentation/trace/kprobetrace.rst for details
- bpftrace: supports instrumentation and scripting using various probes like tracepoint, kprobe and others. More information at https://bpftrace.org
- drgn: drgn supports easy scriptability to debug kernel. More information at https://drgn.readthedocs.io/en/latest/
- Much more performance monitoring and tracing tools information at https://www.brendangregg.com

LF live MENTORSHIP SERIES

Thank you for joining us today!

We hope it will be helpful in your journey to learning more about effective and productive participation in open source projects. We will leave you with a few additional resources for your continued learning:

- The LF Mentoring Program is designed to help new developers with necessary skills and resources to experiment, learn and contribute effectively to open source communities.
- Outreachy remote internships program supports diversity in open source and free software
- Linux Foundation Training offers a wide range of free courses, webinars, tutorials and publications to help you explore the open source technology landscape.
- Linux Foundation Events also provide educational content across a range of skill levels and topics, as well as the chance to meet others in the community, to collaborate, exchange ideas, expand job opportunities and more. You can find all events at events.linuxfoundation.org.