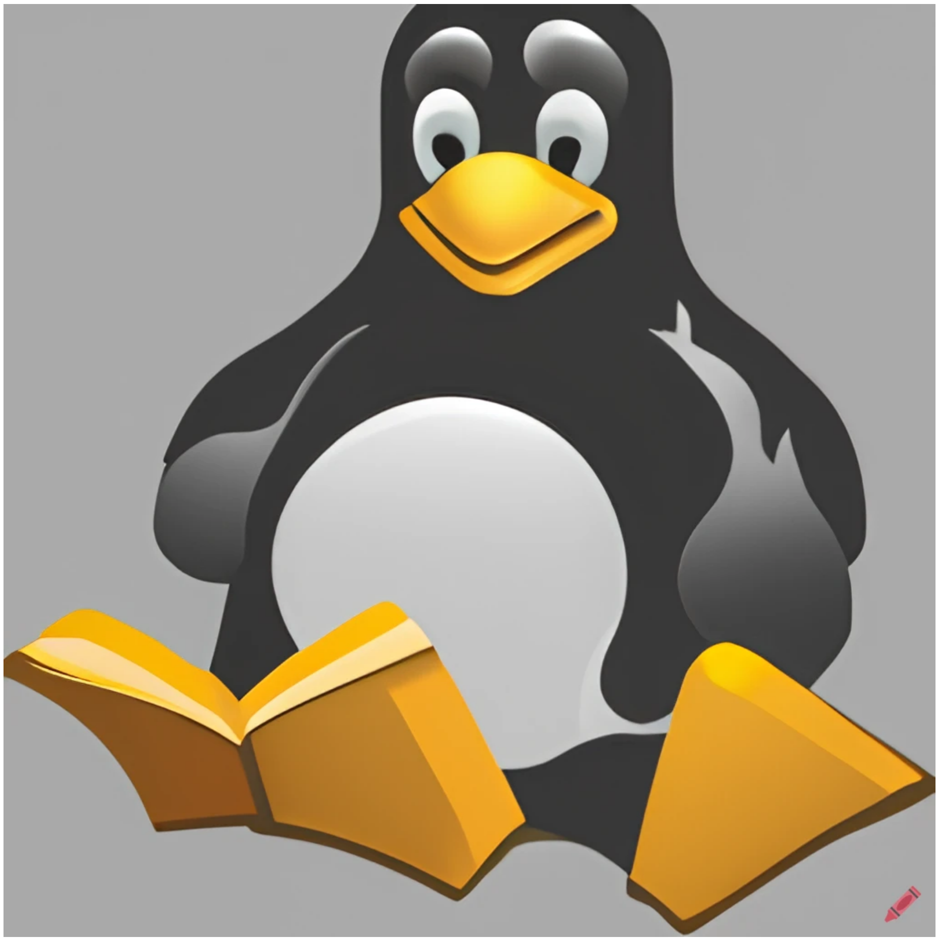
The Linux Process Journey

version 6.0

September-2023

By Dr. Shlomi Boutnaru



Created using Craiyon, AI Image Generator

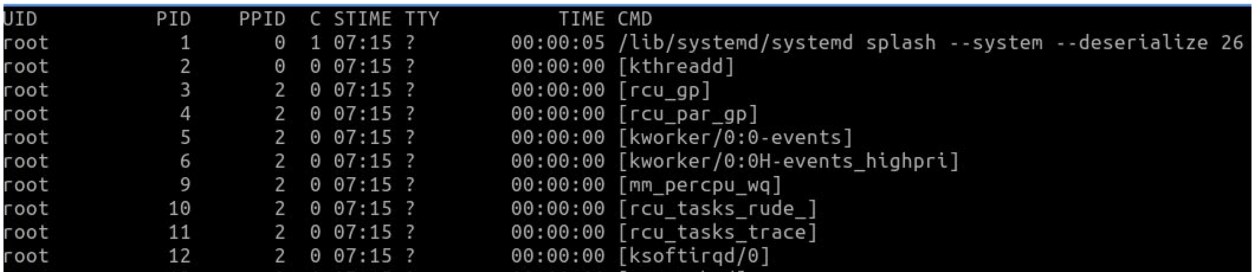
Table of Contents

Introduction

When starting to learn OS internals I believe that we must understand the default processes executing (roles, tasks, etc). Because of that I have decided to write a series of short writeups named "Process ID Card" (aimed at providing the OS vocabulary).

Overall, I wanted to create something that will improve the overall knowledge of Linux in writeups that can be read in 1-3 mins. I hope you are going to enjoy the ride.

In order to create the list of processes I want to explain, I have installed a clean Ubuntu 22.10 VM (Desktop version) and executed ps (as can be seen in the following image - not all the output was included).



```
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  1 07:15 ?        00:00:05 /lib/systemd/systemd splash --system --deserialize 26
root           2       0  0 07:15 ?        00:00:00 [kthreadd]
root           3       2  0 07:15 ?        00:00:00 [rcu_gp]
root           4       2  0 07:15 ?        00:00:00 [rcu_par_gp]
root           5       2  0 07:15 ?        00:00:00 [kworker/0:0-events]
root           6       2  0 07:15 ?        00:00:00 [kworker/0:0H-events_highpri]
root           9       2  0 07:15 ?        00:00:00 [mm_percpu_wq]
root          10       2  0 07:15 ?        00:00:00 [rcu_tasks_rude_]
root          11       2  0 07:15 ?        00:00:00 [rcu_tasks_trace]
root          12       2  0 07:15 ?        00:00:00 [ksoftirqd/0]
```

Probably the best way to do it is to go over the processes by the order of their PID value.
The first one I want to talk about is the one we can't see on the list, that is PID 0 (we can see it is the PPID for PID 1 and PID 2 - on them in the next posts).

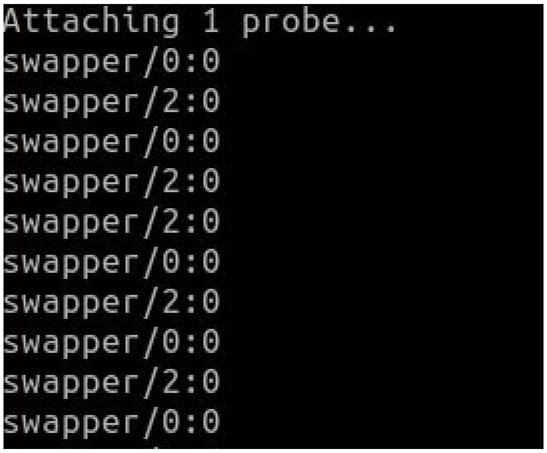
Lastly, you can follow me on twitter - @boutnaru (https://twitter.com/boutnaru). Also, you can read my other writeups on medium - https://medium.com/@boutnaru.

Lets GO!!!!!!

swapper (PID 0)

Historically, old Unix systems used swapping and not demand paging. So, swapper was responsible for the “Swap Process” - moving all pages of a specific process from/to memory/backing store (including related process’ kernel data structures). In the case of Linux PID 0 was used as the “idle process”, simply does not do anything (like nops). It was there so Linux will always have something that a CPU can execute (for cases that a CPU can’t be stopped to save power). By the way, the idle syscall is not supported since kernel 2.3.13 (for more info check out “man 2 idle”). So what is the current purpose of swapper today? helping with pageout ? cache flushes? idling? buffer zeroning? I promise we will answer it in more detail while going through the other processes and explaining the relationship between them.

But how can you believe that swapper (PID 0) even exists? if you can’t see it using ps. I am going to use “bpftrace” for demonstrating that (if you don’t know about bpftrace, I strongly encourage you to read about it). In the demo I am going to trace the kernel function “hrtimer_wakeup” which is responsible for waking up a process and move it to the set of runnable processes. During the trace I am going to print the pid of the calling process (BTW, in the kernel everything is called a task - more on that in future posts) and the executable name (the comm field of the task_struct [/include/linux/sched.h]). Here is the command: sudo bpftrace -e 'kfunc:hrtimer_wakeup { printf("%s:%d\n",curtask->comm,curtask->pid); }'.



```
Attaching 1 probe...
swapper/0:0
swapper/2:0
swapper/0:0
swapper/2:0
swapper/2:0
swapper/0:0
swapper/2:0
swapper/0:0
swapper/2:0
swapper/0:0
```

From the output we can see we have 3 instances of swapper: swapper/0, swapper/1 and swapper/2 all of them with PID 0. The reason we have three is because my VM has 3 virtual CPUs and there is a swapper process for each one of them - see the output of the command in the following image.

init (PID 1)

After explaining about PID 0, now we are going to talk about PID 1. Mostly known as “init”. init is the first Linux user-mode process created, which runs until the system shuts down. init manages the services (called demons under Linux, more on them in a future post). Also, if we check the process tree of a Linux machine we will find that the root of the tree is init.

There are multiple implementations for init, each of them provide different advantages among them are: SysVinit, launched, systemd, runit, upstart, busybox-init and OpenRC (those are examples only and not a full list). Thus, based on the implementation specific configuration files are read (such as /etc/inittab - SysVinit), different command/tools to manage demons (such as service - SysVinit and systemctl - systemd), and different scripts/profiles might be executed during the boot process (runlevels of SysVinit vs targets in systemd).

The creation of init is done by the kernel function “rest_init”[1]. In the code we can see the call to “user_mode_thread” which spawns init, later in the function there is a call to “kernel_thread” which creates PID 2 (more information about it in the upcoming pages ;-).

Now we will go over a couple of fun facts about init. First, in case a parent process exits before all of its children process, init adopts those child processes. Second, only the signals which have been explicitly installed with a handler can be sent to init. Thus, sending “kill -9 1” won’t do anything in most distributions (try it and see nothing happens). Remember that different init implementations handle signals in different ways.

Because they are multiple init implementations (as we stated before) we can determine the one installed in the following manner. We can perform “ls -l /sbin/init”. If it is not a symlink it is probably SysVinit, else if it points to “/lib/systemd/systmed” than systemd is in use (and of course they are other symlinks to the other implementation - you can read about it in the documentation of each init implementation). As you can see in the attached screenshot Ubuntu 22.10 uses systemd.

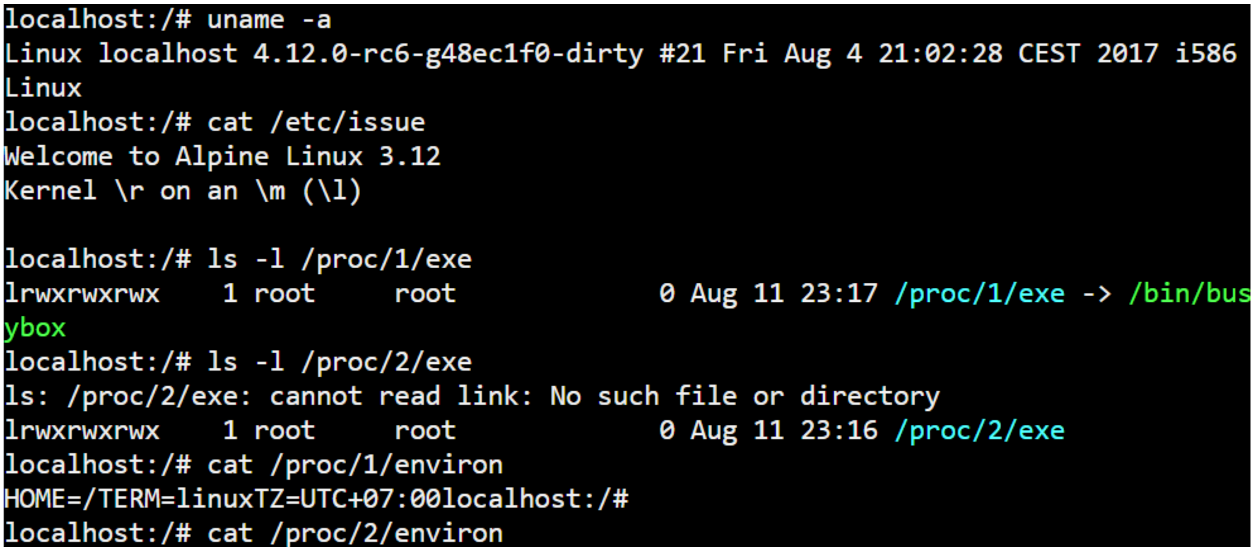
[1] https://elixir.bootlin.com/linux/v6.1.8/source/init/main.c#L683

Kernel Threads

Before we will go over kthreadd I have decided to write a short post about kernel threads (due to the fact kthreadd is a kernel thread). We will go over some characteristics of kernel threads. First, kernel threads always execute in Kernel mode and never in User mode. Thus, kernel threads have basically all privileges and have no userspace address associated with them.

Second, both user mode process and kernel threads are represented by a task_struct inside the Linux kernel. As with all other user tasks, kernel threads are also part of the OS scheduling flow and can be executed on any CPU (there are cases in which there is a specific kernel thread for each CPU, we have seen it with swapper in the first post). Third, all kernel threads are descendants of kthreadd - Why is that? We will explain it in the next post focused on kthreadd.

Lastly, let’s investigate kernel threads using /proc and see the difference in information retrieved from a regular user process (aka user task). There are multiple file entries in “/proc/pid” that contain information in case of a user mode process but are empty in case of a kernel thread, such as: “maps”, “environ”, “auxv”, “cmdline” (I suggest reading “man proc” to get more info about them). Also, the fd and fdinfo directories are empty and the link “exe” does not point to any executable. In the attached screenshot we can see some of the difference between PID 1 [example of a regular user mode process] and PID 2 [example for a kernel thread]. BTW, the screenshot below was taken from an online/browser based Linux implementation called JSLinux - https://bellard.org/jslinux.



```
localhost:/# uname -a
Linux localhost 4.12.0-rc6-g48ec1f0-dirty #21 Fri Aug 4 21:02:28 CEST 2017 i586
Linux
localhost:/# cat /etc/issue
Welcome to Alpine Linux 3.12
Kernel \r on an \m (\l)

localhost:/# ls -l /proc/1/exe
lrwxrwxrwx    1 root     root             0 Aug 11 23:17 /proc/1/exe -> /bin/bus
ybox
localhost:/# ls -l /proc/2/exe
ls: /proc/2/exe: cannot read link: No such file or directory
lrwxrwxrwx    1 root     root             0 Aug 11 23:16 /proc/2/exe
localhost:/# cat /proc/1/environ
HOME=/TERM=linuxTZ=UTC+07:00localhost:/#
localhost:/# cat /proc/2/environ
```

kthreadd (PID 2)

After explaining about PID 1, now we are going to talk about PID 2.Basically, kthreadd is the "kernel thread daemon". Creation of a new kernel thread is done using kthreadd (We will go over the entire flow). Thus, the PPID of all kernel threads is 2 (checkout ps to verify this). As explained in the post about PID 1 (init) the creation of "kthreadd" is done by the kernel function "rest_init"[2]. There is a call to the function "kernel_thread" (after the creation of init).

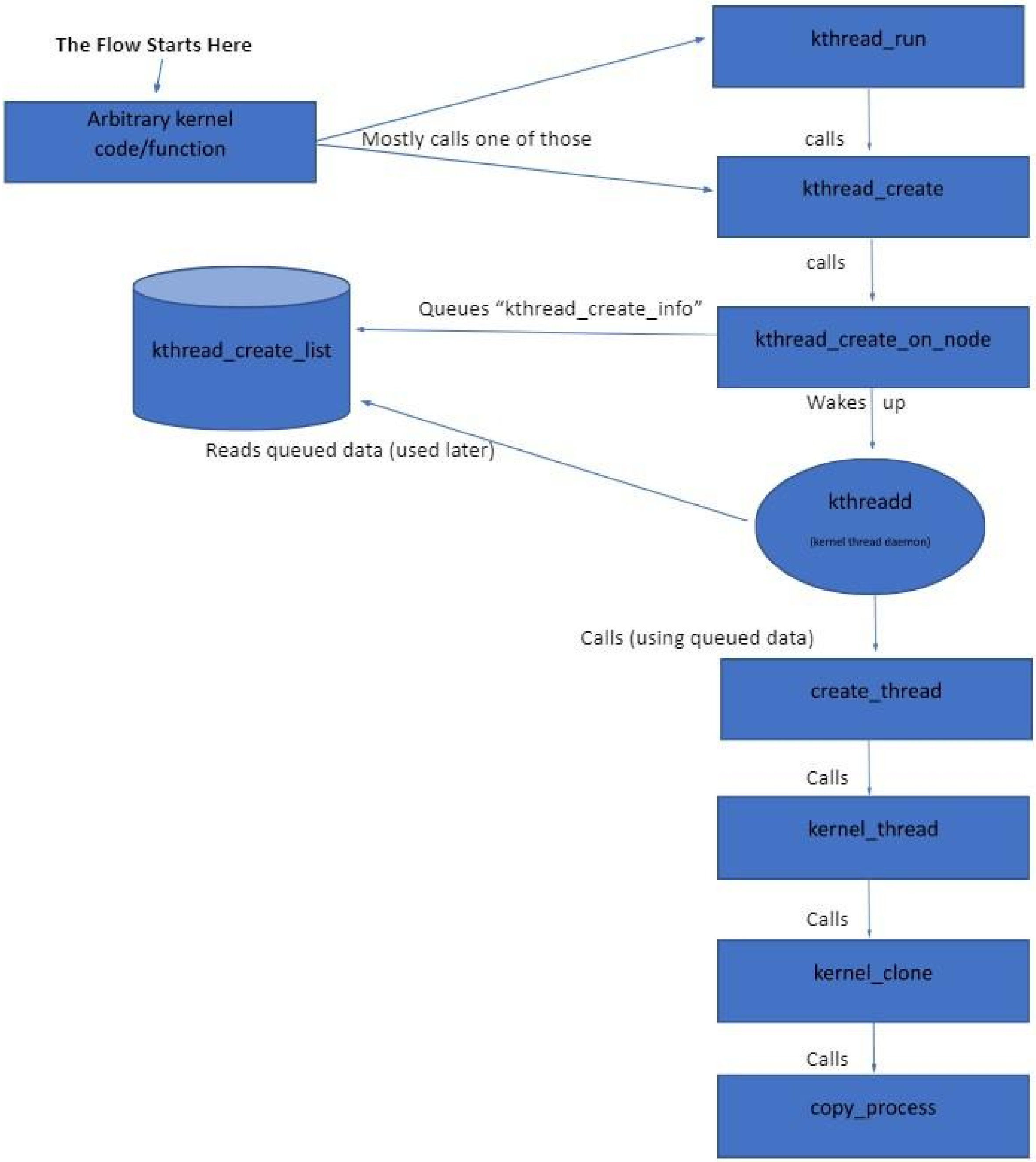
Basically, the kernel uses "kernel threads" (kthreads from now on) in order to run background operations. Thus, it is not surprising that multiple kernel subsystems are leveraging kthreads in order to execute async operations and/or periodic operations. In summary, the goal of kthreadd is to make available an interface in which the kernel can dynamically spawn new kthreads when needed.

Overall, kthreadd continuously runs (infinite loop[3]) and checks "kthread_create_list" for new kthreads to be created. In order to create a kthread the function "kthread_create"[4] is used, which is a helper macro for "kthread_create_on_node"[5]. We can also call "kthread_run"[6] could also be used, it is just a wrapper for "kthread_create". The arguments passed to the creating function includes: the function to run in the thread, args to the function and a name.

While going over the source code we have seen that "kthread_create" calls "kthread_create_on_node", which instantiates a "kthread_create_info" structure (based on the args of the function). After that, that structure is queued at the tail of "kthread_create_list" and "kthreadd" is awakened (and it waits until the kthread is created, this is done by "__kthread_create_on_node"[7]). What "kthreadd" does is to call "create_thread" based on the information queued. "create_thread" calls "kernel_thread", which then calls "kernel_clone". "kernel_clone" executes "copy_process", which creates a new process as a copy of an old one - the caller needs to kick-off the created process (or thread in our case). By the way, the flow of creating a new task (recall every process/thread under Linux is called task and represented by "struct task_struct") from user mode also gets to "copy_process".

For the sake of simplicity, I have created a flow graph which showcases the flow of creating a kthread, not all the calls are there, only those I thought are important enough. Also, in both cases of macros/functions I used the verb "calls". The diagram appears at the end of the post. Let me know if it is clear enough or do you think I should change something.

[2] https://elixir.bootlin.com/linux/v6.1.8/source/init/main.c#L683
[3] https://elixir.bootlin.com/linux/v6.1.12/source/kernel/kthread.c#L731
[4] https://elixir.bootlin.com/linux/v6.1.12/source/include/linux/kthread.h#L27
[5] https://elixir.bootlin.com/linux/v6.1.12/source/kernel/kthread.c#L503
[6] https://elixir.bootlin.com/linux/v6.1.12/source/include/linux/kthread.h#L51
[7] https://elixir.bootlin.com/linux/v6.1.12/source/kernel/kthread.c#L414


The Flow Starts Here
Arbitrary kernel code/function
Mostly calls one of those
kthread_run
calls
kthread_create
calls
kthread_create_list
Queues "kthread_create_info"
kthread_create_on_node
Wakes up
Reads queued data (used later)
kthreadd
(kernel thread daemon)
Calls (using queued data)
create_thread
Calls
kernel_thread
Calls
kernel_clone
Calls
copy_process

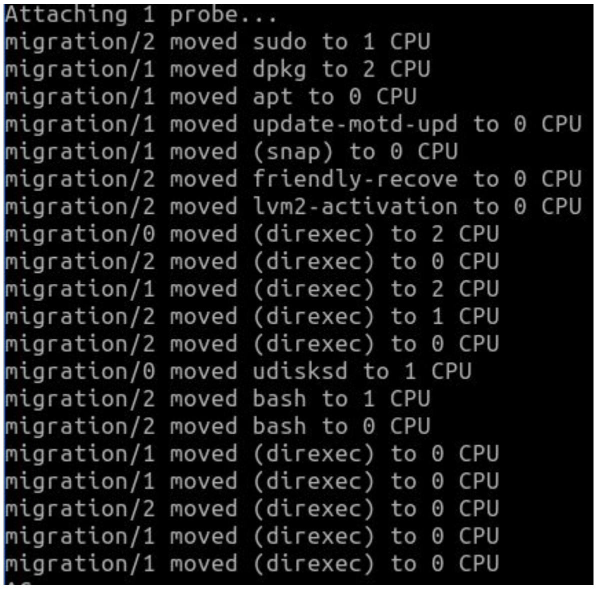
migration

One of the goals of an operating system is to handle and balance resources across the hardware of the compute entity. In order to do that, Linux has a kernel thread named “migration” which has an instance on every vCPU. By the way, the naming format is “migration/N” where N is the id of the vCPU.

By default threads are not constrained to a vCPU and can be migrated between them in the next call to “schedule()” (which calls the main scheduler function, which is “__scheduler()”[8]). It is done mainly in case the scheduler identifies an unbalanced across the runqueues (the queue in which processes which are in ready/runnable state are waiting to use the processor) of the vCPUs.

It is important to state that we can influence this flow by setting the affinity of a thread (for more read “man 2 sched_setaffinity”. We will talk about that in a future post). There could be performance, cache and other impacts for doing that (but that is also a topic for a different writeup).

I have created a small demo which shows the working of “migration”. For that I have created a VM running Ubuntu 22.04 with 3 vCPUs. In order to trace the usage of “move_queue_task” I have used bpftrace with the following command: **sudo bpftrace -e 'kfunc:move_queued_task { printf("%s moved %s to %d CPU\n",curtask->comm,args->p->comm,args->new_cpu); }'**. The output of the command is shown below. The one-liner prints: the name of the task calling “move_queued_task”, the name of the task which is moved and id the vCPU which the task is moved to.



```
Attaching 1 probe...
migration/2 moved sudo to 1 CPU
migration/1 moved dpkg to 2 CPU
migration/1 moved apt to 0 CPU
migration/1 moved update-motd-upd to 0 CPU
migration/1 moved (snap) to 0 CPU
migration/2 moved friendly-recove to 0 CPU
migration/2 moved lvm2-activation to 0 CPU
migration/0 moved (direxec) to 2 CPU
migration/2 moved (direxec) to 0 CPU
migration/1 moved (direxec) to 2 CPU
migration/2 moved (direxec) to 1 CPU
migration/2 moved (direxec) to 0 CPU
migration/0 moved udisksd to 1 CPU
migration/2 moved bash to 1 CPU
migration/2 moved bash to 0 CPU
migration/1 moved (direxec) to 0 CPU
migration/1 moved (direxec) to 0 CPU
migration/2 moved (direxec) to 0 CPU
migration/1 moved (direxec) to 0 CPU
migration/1 moved (direxec) to 0 CPU
```

[8] https://elixir.bootlin.com/linux/latest/source/kernel/sched/core.c#L6544

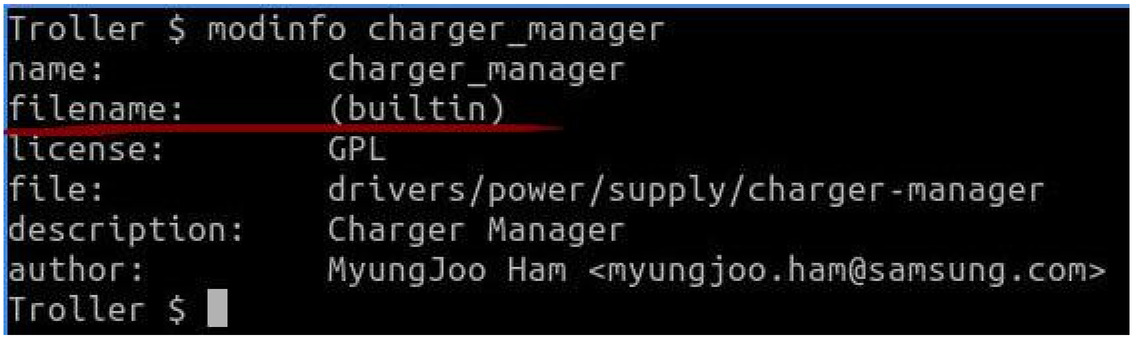
In summary, what the kernel thread “migration” does is to move threads from highly loaded vCPUs to others which are less crowded (by inserting them to a different run-queue). A function which is used by “migration” in order to move a task to a new run-queue is “move_queued_task” (https://elixir.bootlin.com/linux/latest/source/kernel/sched/core.c#L2325).

charger_manager

The “charger_manager” kernel thread is created by a freezable workqueue[9]. Freezable workqueues are basically frozen when the system is moved to a suspend state[10]. Based on the kernel source code “charger_manager” is responsible for monitoring the health (like temperature monitoring) of the battery and controlling the charger while the system is suspended to memory[11]. The “Charger Manager” kernel module is written by MyungJoo Ham[12].

Moreover, the kernel documentation states that the “Charger Manager" also helps in giving an aggregated view to user-space in case there are multiple chargers for a battery. In case they are multiple batteries with different chargers on a system, that system would need multiple instances of “Charger Manager”[13] .

On my Ubuntu VM (22.04.1 LTS) this kernel module is not compiled as a separate “*.ko” file. It is compiled into the kernel itself (builtin), as you can see in the output of “modinfo” in the screenshot below.



```
Troller $ modinfo charger_manager
name:           charger_manager
filename:       (builtin)
license:        GPL
file:           drivers/power/supply/charger-manager
description:    Charger Manager
author:         MyungJoo Ham <myungjoo.ham@samsung.com>
Troller $ 
```

[9] https://elixir.bootlin.com/linux/latest/source/drivers/power/supply/charger-manager.c#L1749
[10] https://lwn.net/Articles/403891/
[11] https://elixir.bootlin.com/linux/latest/source/drivers/power/supply/charger-manager.c
[12] https://elixir.bootlin.com/linux/latest/source/drivers/power/supply/charger-manager.c#L1768
[13] https://www.kernel.org/doc/html/v5.3/power/charger-manager.html

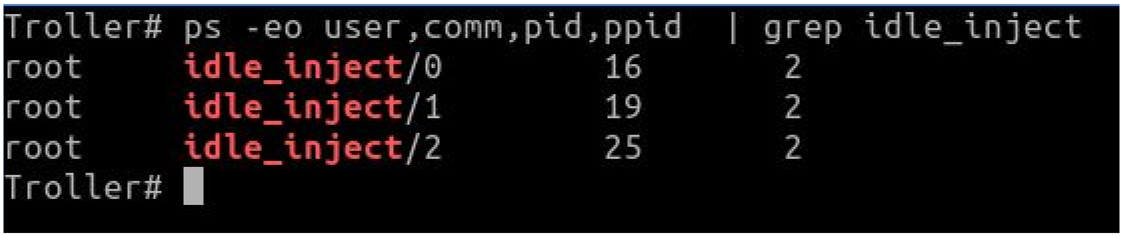
idle_inject

On our plate this time we are going to talk about the kernel thread “idle_inject”, which was merged to the kernel in about 2009. The goal of “idle_inject” is forcing idle time on a CPU in order to avoid overheating.

If we think about it, “idle_inject” adds latency, thus it should be considered only if CPUFreq (CPU Frequency scaling) is not supported. Due to the fact the majority of modern CPUs are capable of running a different clock frequency and voltage configuration we can use CPUFreq in order to avoid overheating.

Overall, there is one “idle_inject” kernel thread per processor (with the name pattern “idle_inject/N”, where N is the id of the processor) - as shown in the screenshot below. Also, all of them are created at init time.

The “idle_inject” kernel threads will call “idle_inject_fn()”->”play_idle_precise()” to inject a specified amount of idle time. After all of the kernel threads are woken up, the OS sets a timer for the next cycle. When the timer interrupt handler wakes the threads for all processors based on a defined “cpu-mask” (affected by idle injection). By the way, when I set a kprobe on “idle_inject_fn()” for 3 hours on my VM it was never called ;-)

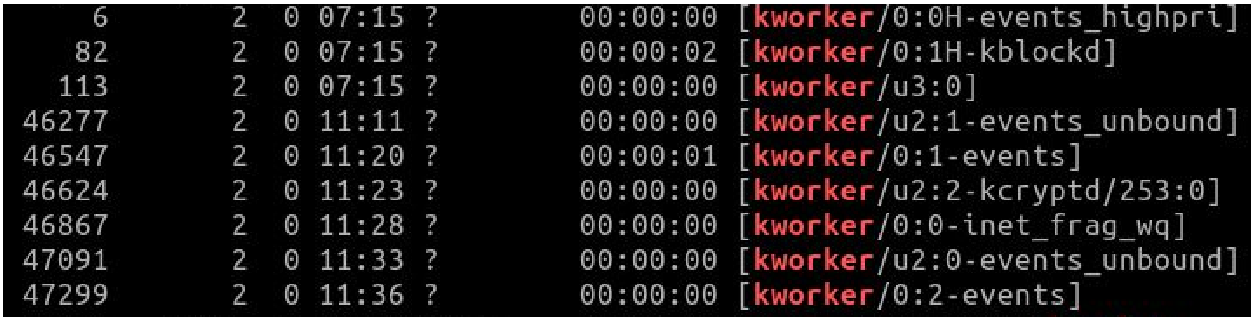


```
Troller# ps -eo user,comm,pid,ppid  | grep idle_inject
root     idle_inject/0       16       2
root     idle_inject/1       19       2
root     idle_inject/2       25       2
Troller#
```

kworker

A kworker is a kernel thread that performs processing as part of the kernel, especially in the case of interrupts, timers, I/O, etc. It is based on workqueues which are async execution mechanisms, that execute in “process context” (I will post on workqueus in more details separately, for now it is all that you need to know).

Overall, there are a couple of kworkers running on a Linux machine. The naming pattern of kworkers includes: the number of the core on which it is executed, the id of the thread and can contain also string that hints what the kworker does (check the output of ‘ps -ef | grep kworker’).

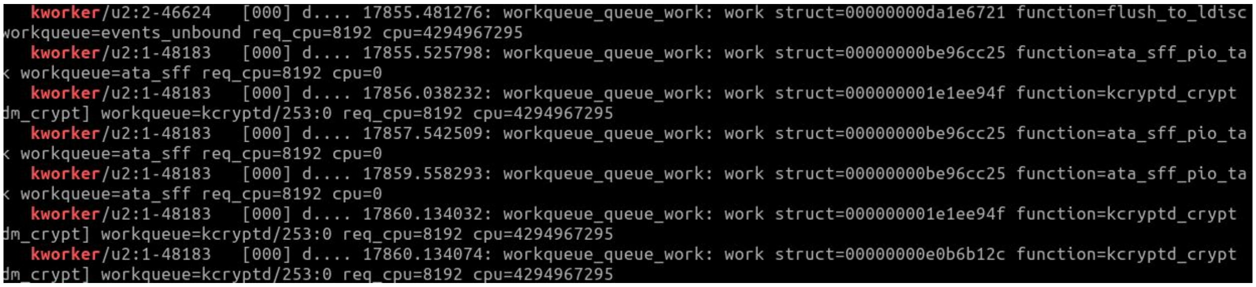


```
      6        2  0 07:15 ?        00:00:00 [kworker/0:0H-events_highpri]
     82        2  0 07:15 ?        00:00:02 [kworker/0:1H-kblockd]
    113        2  0 07:15 ?        00:00:00 [kworker/u3:0]
  46277        2  0 11:11 ?        00:00:00 [kworker/u2:1-events_unbound]
  46547        2  0 11:20 ?        00:00:01 [kworker/0:1-events]
  46624        2  0 11:23 ?        00:00:00 [kworker/u2:2-kcryptd/253:0]
  46867        2  0 11:28 ?        00:00:00 [kworker/0:0-inet_frag_wq]
  47091        2  0 11:33 ?        00:00:00 [kworker/u2:0-events_unbound]
  47299        2  0 11:36 ?        00:00:00 [kworker/0:2-events]
```

The big question is - “How do we know what each kwoker is doing?”. It’s a great question, the way in which we are going to answer it is by using ftrace (function tracing inside the kernel - I suggest reading more about that - https://www.kernel.org/doc/Documentation/trace/ftrace.txt). The command we are going to use are:

echo workqueue:workqueue_queue_work > /sys/kernel/debug/tracing/set_event
cat /sys/kernel/debug/tracing/trace_pipe > /tmp/trace.log

The first one enables the tracing regarding workqueus. The second reads the tracing data and saves it to a file. We can also run “cat /sys/kernel/debug/tracing/trace_pipe | grep kworker” and change the grep filter to a specific kworker process. In the trace we will see the function name that each kworker thread is going to execute.



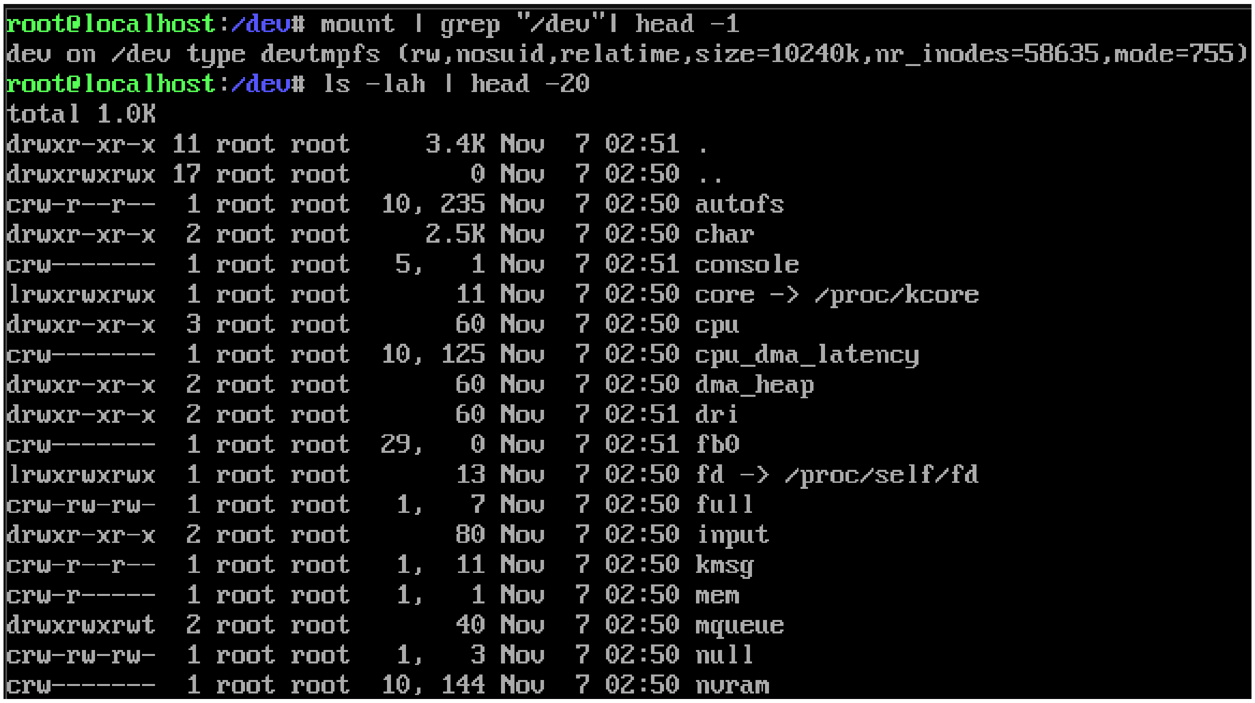
```
   kworker/u2:2-46624   [000] d.... 17855.481276: workqueue_queue_work: work struct=00000000da1e6721 function=flush_to_ldisc
orkqueue=events_unbound req_cpu=8192 cpu=4294967295
   kworker/u2:1-48183   [000] d.... 17855.525798: workqueue_queue_work: work struct=00000000be96cc25 function=ata_sff_pio_ta
 workqueue=ata_sff req_cpu=8192 cpu=0
   kworker/u2:1-48183   [000] d.... 17856.038232: workqueue_queue_work: work struct=000000001e1ee94f function=kcryptd_crypt
m_crypt] workqueue=kcryptd/253:0 req_cpu=8192 cpu=4294967295
   kworker/u2:1-48183   [000] d.... 17857.542509: workqueue_queue_work: work struct=00000000be96cc25 function=ata_sff_pio_ta
 workqueue=ata_sff req_cpu=8192 cpu=0
   kworker/u2:1-48183   [000] d.... 17859.558293: workqueue_queue_work: work struct=00000000be96cc25 function=ata_sff_pio_ta
 workqueue=ata_sff req_cpu=8192 cpu=0
   kworker/u2:1-48183   [000] d.... 17860.134032: workqueue_queue_work: work struct=000000001e1ee94f function=kcryptd_crypt
m_crypt] workqueue=kcryptd/253:0 req_cpu=8192 cpu=4294967295
   kworker/u2:1-48183   [000] d.... 17860.134074: workqueue_queue_work: work struct=00000000e0b6b12c function=kcryptd_crypt
m_crypt] workqueue=kcryptd/253:0 req_cpu=8192 cpu=4294967295
```

kdevtmpfs

“kdevtmpfs” is a kernel thread which was created using the “kthread_run” function[14]. “kdevtmpfs” creates a devtmpfs which is a tmpfs-based filesystem (/dev). The filesystem is created during bootup of the system, before any driver code is registered. In case a driver-core requests a device node it will result in a node added to this filesystem[15].

We can see the specific line of code that is used in order to create the mounting point “/dev”[16]. The mountpoint is created using the function “init_mount”[17]. A nice fact is that it is part of “init_*” functions which are routines that mimic syscalls but don’t use file descriptors or the user address space. They are commonly used by early init code[18].

Thus, we can say the “kdevtmpfs” is responsible for managing the “Linux Device Tree”. Also, by default the name created for nodes under the filesystem is based on the device name (and owned by root) - as shown in the screenshot below (taken from copy.sh based Linux). By the way, not all devices have a node in “/dev” think about network devices ;-)



```
root@localhost:/dev# mount | grep "/dev"| head -1
dev on /dev type devtmpfs (rw,nosuid,relatime,size=10240k,nr_inodes=58635,mode=755)
root@localhost:/dev# ls -lah | head -20
total 1.0K
drwxr-xr-x 11 root root     3.4K Nov  7 02:51 .
drwxrwxrwx 17 root root        0 Nov  7 02:50 ..
crw-r--r--  1 root root  10, 235 Nov  7 02:50 autofs
drwxr-xr-x  2 root root     2.5K Nov  7 02:50 char
crw-------  1 root root   5,   1 Nov  7 02:51 console
lrwxrwxrwx  1 root root       11 Nov  7 02:50 core -> /proc/kcore
drwxr-xr-x  3 root root       60 Nov  7 02:50 cpu
crw-------  1 root root  10, 125 Nov  7 02:50 cpu_dma_latency
drwxr-xr-x  2 root root       60 Nov  7 02:50 dma_heap
drwxr-xr-x  2 root root       60 Nov  7 02:51 dri
crw-------  1 root root  29,   0 Nov  7 02:51 fb0
lrwxrwxrwx  1 root root       13 Nov  7 02:50 fd -> /proc/self/fd
crw-rw-rw-  1 root root   1,   7 Nov  7 02:50 full
drwxr-xr-x  2 root root       80 Nov  7 02:50 input
crw-r--r--  1 root root   1,  11 Nov  7 02:50 kmsg
crw-r-----  1 root root   1,   1 Nov  7 02:50 mem
drwxrwxrwt  2 root root       40 Nov  7 02:50 mqueue
crw-rw-rw-  1 root root   1,   3 Nov  7 02:50 null
crw-------  1 root root  10, 144 Nov  7 02:50 nvram
```

[14] https://elixir.bootlin.com/linux/v6.2-rc1/source/drivers/base/devtmpfs.c#L474
[15] https://elixir.bootlin.com/linux/v6.2-rc1/source/drivers/base/devtmpfs.c#L3
[16] https://elixir.bootlin.com/linux/v6.2-rc1/source/drivers/base/devtmpfs.c#L377
[17] https://elixir.bootlin.com/linux/v6.2-rc1/source/fs/init.c#L16
[18] https://elixir.bootlin.com/linux/v6.2-rc1/source/fs/init.c#L3

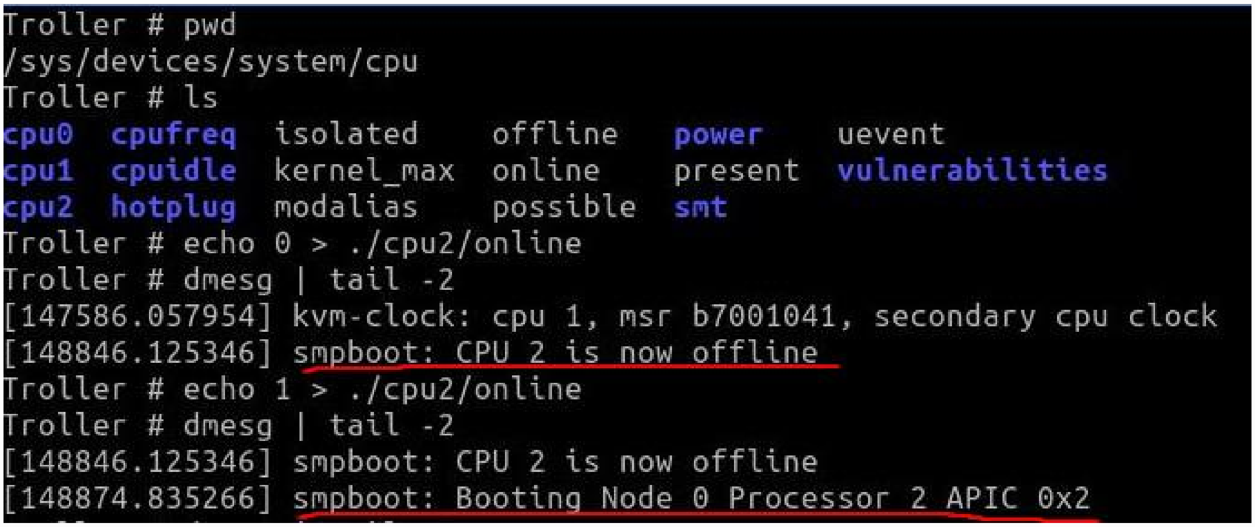
cpuhp

This kernel thread is part of the CPU hotplug support. It enables physically removing/adding CPUs on a specific system. There is one kernel thread per vCPU, and the pattern of the thread's name is "cpuhp/N" (where N is the id of the vCPU) - as can be seen in the screenshot below. Also, today the CPU hotplug can be used to resume/suspend support for SMP (Symmetric Multiprocessing).

If we want our kernel to support CPU hotplug the CONFIG_HOTPLUG_CPU should be enabled (it's supported on a couple of architectures such as: MIPS, ARM, x86 and PowerPC). The kernel holds the current state for each CPU by leveraging "struct cpuhp_cpu_state"[19].

We can configure the CPU hotplug mechanism using sysfs (/sys/devices/system/cpu). For example we can shut down and bring up a CPU by writing "0" and "1" respectively to the "online" file in the directory representing the CPU (for which we want to change the status) - checkout the screenshot below (the Linux VM I am testing on has 3 vCPUs).

In order to bring the CPU down the function "cpu_device_down"[20] is called. In order to bring up a CPU function "cpu_device_up"[21] is called.



```
Troller # pwd
/sys/devices/system/cpu
Troller # ls
cpu0  cpufreq  isolated    offline   power    uevent
cpu1  cpuidle  kernel_max  online    present  vulnerabilities
cpu2  hotplug  modalias    possible  smt
Troller # echo 0 > ./cpu2/online
Troller # dmesg | tail -2
[147586.057954] kvm-clock: cpu 1, msr b7001041, secondary cpu clock
[148846.125346] smpboot: CPU 2 is now offline
Troller # echo 1 > ./cpu2/online
Troller # dmesg | tail -2
[148846.125346] smpboot: CPU 2 is now offline
[148874.835266] smpboot: Booting Node 0 Processor 2 APIC 0x2
```

[19] https://elixir.bootlin.com/linux/latest/source/kernel/cpu.c#L65
[20] https://elixir.bootlin.com/linux/latest/source/kernel/cpu.c#L1225
[21] https://elixir.bootlin.com/linux/latest/source/kernel/cpu.c#L1439

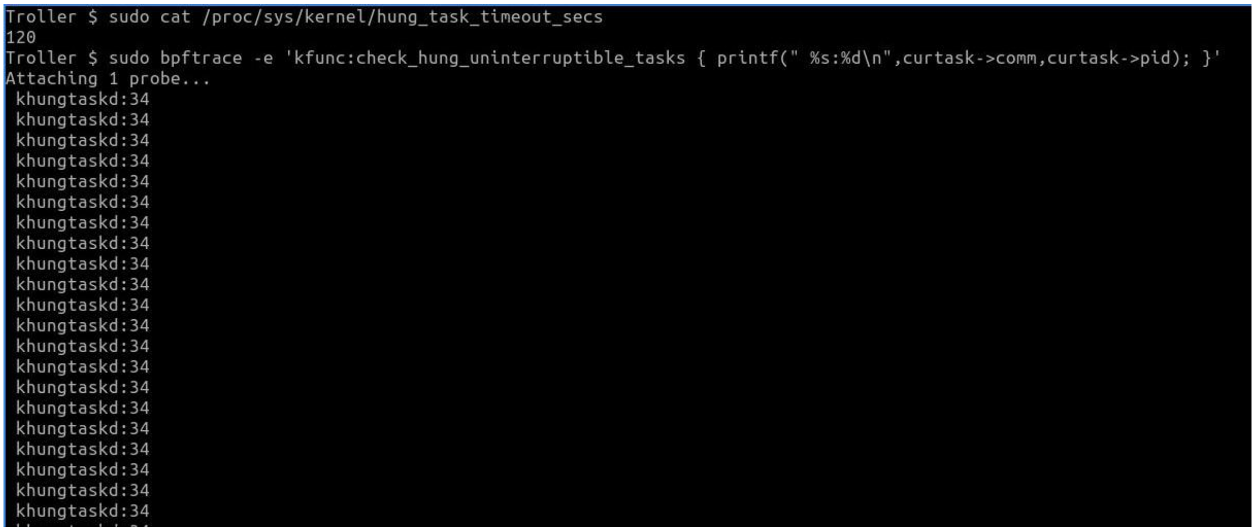
khungtaskd

This kernel thread “khungtaskd” is used in order to help with identifying and debugging “Hung Tasks”. This kernel thread is scheduled every 120 seconds (that is the default value). We can say “khungtaskd” is used for detecting tasks which are stuck in uninterruptible sleep (state “D” in ps output). We can also go over the code of the kernel thread as part of the Linux kernel source code[22].

The basic algorithm of “khungtaskd” is as follows: Iterate over all running tasks on the system and if there are ones marked as TASK_UNINTERRUPTIBLE and it was scheduled at least once in the last 120 seconds it is considered as hung. When a task is considered hung it’s “call stack” is dumped and if the CONFIG_LOCKDEP is also enabled then all of the locks held by the tasks are outpted also.

If we want we can change the sampling interval using the sysctl interface, “/proc/sys/kernel/hung_task_timeout_secs” .We can also verify that the default is 120 seconds by reading it - as shown in the screenshot below.

In order to demonstrate the operation of “khungtaskd” I have executed the following bpftrace one liner - “sudo bpftrace -e 'kfunc:check_hung_uninterruptible_tasks { printf(" %s:%d\n",curtask->comm,curtask->pid); }'”. The trace prints the name of the task and it’s pid when the function “check_hung_uninterruptible_tasks” is called[23] - You can see the output in the screenshot below.



```
Troller $ sudo cat /proc/sys/kernel/hung_task_timeout_secs
120
Troller $ sudo bpftrace -e 'kfunc:check_hung_uninterruptible_tasks { printf(" %s:%d\n",curtask->comm,curtask->pid); }'
Attaching 1 probe...
 khungtaskd:34
 khungtaskd:34
 khungtaskd:34
 khungtaskd:34
 khungtaskd:34
 khungtaskd:34
 khungtaskd:34
 khungtaskd:34
 khungtaskd:34
 khungtaskd:34
 khungtaskd:34
 khungtaskd:34
 khungtaskd:34
 khungtaskd:34
 khungtaskd:34
 khungtaskd:34
 khungtaskd:34
 khungtaskd:34
 khungtaskd:34
 khungtaskd:34
 khungtaskd:34
```

[22] https://elixir.bootlin.com/linux/latest/source/kernel/hung_task.c
[23] https://elixir.bootlin.com/linux/latest/source/kernel/hung_task.c#L178

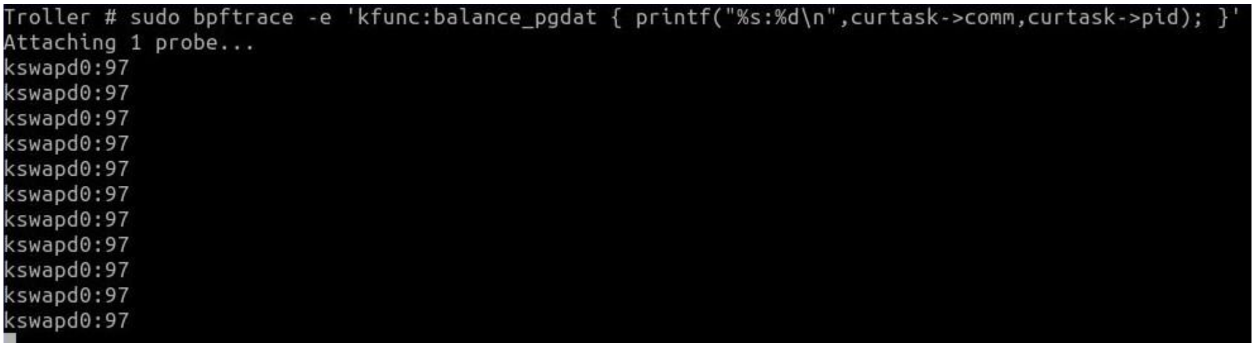
]

dedicated instance of “kswapd” is created for each NUMA zone (on my Ubuntu 22.10 VM I have only “kswapd0” - as shown in the screenshot below).

Overall, the goal of the “kswapd” is to reclaim pages when memory is running low. In the old days, the “kswapd” was woken every 10 seconds but today it is only wakened by the page allocator, by calling “wakeup_kswapd”[24]. The code of the page allocator is located at “mm/page_alloc.c”[25].

Basically, “kswapd” trickles out pages so the system has some free memory even if no other activity frees up anything (like by shrinking cache). Think about cases in which operations work in asynchronous contexts that cannot page things out.

The major function which is called by “kswapd” is “balance_pgdat()”[26]. In order to see that process happening we can use the following bpftrace one-liner: “**sudo bpftrace -e 'kfunc:balance_pgdat { printf("%s:%d\n",curtask->comm,curtask->pid); }'**” - You can see “kswapd0” calling it in the screenshot below. The flow of “kswapd” is based on limits, when to start shirking and “until when” to shrink (low and high limits).



```
Troller # sudo bpftrace -e 'kfunc:balance_pgdat { printf("%s:%d\n",curtask->comm,curtask->pid); }'
Attaching 1 probe...
kswapd0:97
kswapd0:97
kswapd0:97
kswapd0:97
kswapd0:97
kswapd0:97
kswapd0:97
kswapd0:97
kswapd0:97
kswapd0:97
kswapd0:97
```

[24] https://elixir.bootlin.com/linux/latest/source/mm/vmscan.c#L4555
[25] https://elixir.bootlin.com/linux/latest/source/mm/page_alloc.c
[26] https://elixir.bootlin.com/linux/latest/source/mm/vmscan.c#L4146

kcompactd

When a Linux system is up and running, memory pages of different processes/tasks are scattered and thus are not physically-contiguous (even if they are contiguous in their virtual address). We can move to bigger pages size (like from 4K to 4M) but it still has its limitations like: waste of space in case of regions with small sizes and the need for multiple pages in case of large regions that can still be fragmented. Due to that, the need for memory compaction was born[27].

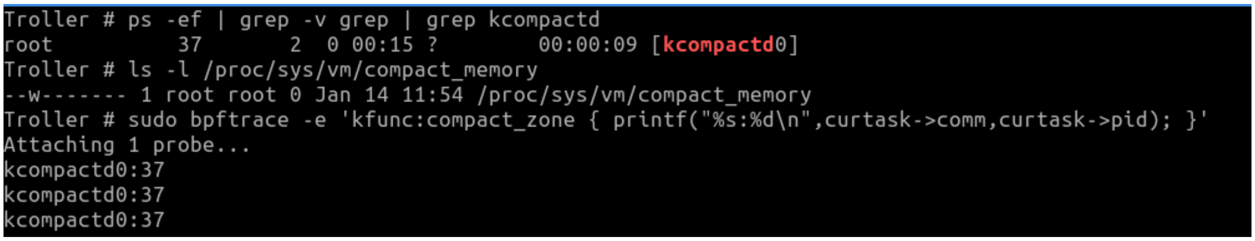
“kcompatd” is performing in the background the memory compaction flow. The goal of memory compaction is to reduce external fragmentation. This procedure is heavily dependent on page migration[28] to do all the heavy lifting[29]. In order for “kcompactd” to work we should compile the kernel with “CONFIG_COMPACTION” enabled. Also, when a Linux system identifies that it is tight low in available memory the “kcompactd” won’t perform memory compaction memory[30].

Overall, the “kcompactd” kernel thread is created in “kcompactd_run” function[31] which is called by “kcompactd_init”[32].. The function “kcompactd_init” is started by “subsys_initcall”[33], which is responsible for initializing a subsystem.

The kernel thread starts the function “static int kcompactd(void *p)”[34].. An instance of the kernel thread is created for each node (like vCPU) on the system[35].. The pattern of the kernel thread name is “kcompactd[IndexOfNode]” for example “kcompactd0” as we can see in the screenshot below.

“kcompactd” can be called in one of two ways: woken up or by using a timeout. It can be woken up by kswapd[36].. Also, we can configure it using modification of the filesystem (“/proc/sys/vm/compact_memroy” for example). By the way, in the memory compaction flow of the function “compact_zone”[37] is executed in the context of “kcompactd”. In order to demonstrate that we can use the following one-liner using bpftrace: **sudo bpftrace -e 'kfunc:compact_zone { printf("%s:%d\n",curtask->comm,curtask->pid); }'** - The output can be seen in the screenshot below.

[27] https://lwn.net/Articles/368869/
[28] https://lwn.net/Articles/157066/
[29] https://elixir.bootlin.com/linux/v6.2-rc3/source/mm/compaction.c#L5
[30] https://www.linux-magazine.com/Issues/2015/179/Kernel-News
[31] https://elixir.bootlin.com/linux/v6.2-rc3/source/mm/compaction.c#L2996
[32] https://elixir.bootlin.com/linux/v6.2-rc3/source/mm/compaction.c#L3048
[33] https://elixir.bootlin.com/linux/v6.2-rc3/source/mm/compaction.c#L3065
[34] https://elixir.bootlin.com/linux/v6.2-rc3/source/mm/compaction.c#L2921
[35] https://elixir.bootlin.com/linux/v6.2-rc3/source/mm/compaction.c#L3061
[36] https://www.slideshare.net/AdrianHuang/memory-compaction-in-linux-kernelpdf
[37] https://elixir.bootlin.com/linux/v6.2-rc3/source/mm/compaction.c#L2289



```
Troller # ps -ef | grep -v grep | grep kcompactd
root          37       2  0 00:15 ?        00:00:09 [kcompactd0]
Troller # ls -l /proc/sys/vm/compact_memory
--w------- 1 root root 0 Jan 14 11:54 /proc/sys/vm/compact_memory
Troller # sudo bpftrace -e 'kfunc:compact_zone { printf("%s:%d\n",curtask->comm,curtask->pid); }'
Attaching 1 probe...
kcompactd0:37
kcompactd0:37
kcompactd0:37
```

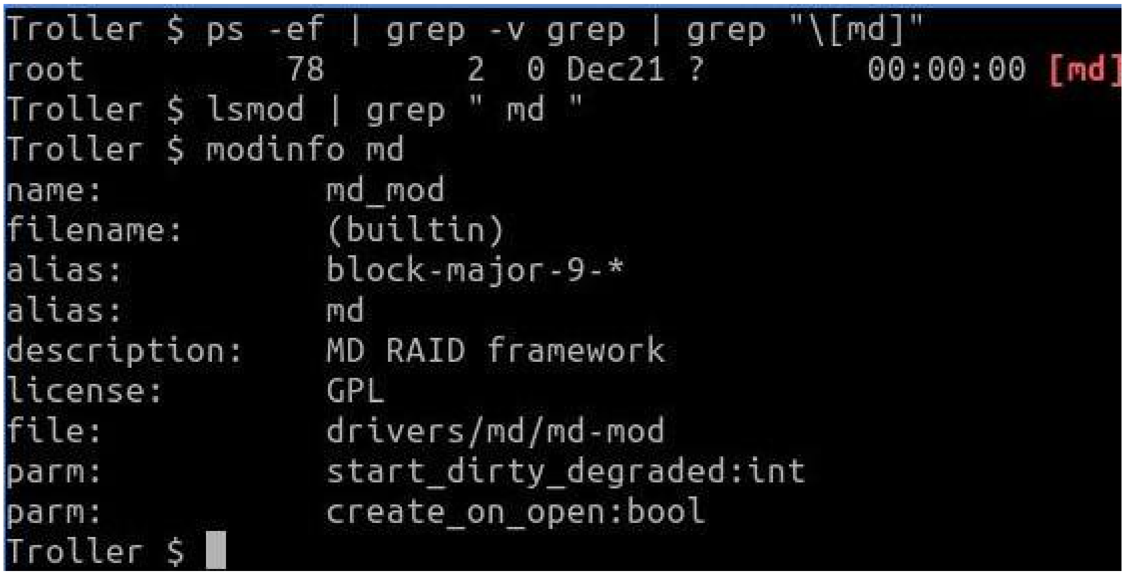
md (Multiple Device Driver)

“md” is a kernel thread which is based on a workqueue[38]. It is responsible for managing the Linux md (multiple device) driver which is also known as the “Linux software RAID”. RAID devices are virtual devices (created from two or more real block devices). This allows multiple devices (typically disk drives or partitions thereof) to be combined into a single device to hold (for example) a single filesystem[39].

By using the “md” driver we can create from one/more physical devices (like disk drivers) a virtual device(s). By the use of an array of devices we can achieve redundancy, which is also known as RAID (Redundant Array of Independent Disks). For more information I suggest reading https://man7.org/linux/man-pages/man4/md.4.html.

Overall, “md” supports different RAID types: RAID 1 (mirroring), RAID 4, RAID 5, RAID 6 and RAID 10. For more information about RAID types I suggest reading the following link https://www.prepressure.com/library/technology/raid. Besides that, “md” also supports pseudo RAID technologies like: RAID 0, LINAR, MULTIPATH and FAULTY[40].

The code of “md” is included as a driver/kernel module in the source code of Linux. Thus, it can be compiled directly into the kernel or as a separate “*.ko” file. In my VM (Ubuntu 22.04) it is compiled directly into the kernel image as shown in the screenshot below.



```
Troller $ ps -ef | grep -v grep | grep "\[md]"
root          78       2  0 Dec21 ?        00:00:00 [md]
Troller $ lsmod | grep " md "
Troller $ modinfo md
name:           md_mod
filename:       (builtin)
alias:          block-major-9-*
alias:          md
description:    MD RAID framework
license:        GPL
file:           drivers/md/md-mod
parm:           start_dirty_degraded:int
parm:           create_on_open:bool
Troller $ 
```

[38] https://elixir.bootlin.com/linux/v6.1/source/drivers/md/md.c#L9615
[39] https://linux.die.net/man/8/mdadm
[40] https://doxfer.webmin.com/Webmin/Linux_RAID

The block devices that can be used in order to access the software RAID on Linux are in the pattern “/dev/mdN” (where N is a number [0–255])[41]. It can also be configured to allow access using “/dev/md/N” or “/dev/md/name”. If we want information about the current state of “md” we can query the file “/proc/mdstat” — for more information you can read https://raid.wiki.kernel.org/index.php/Mdstat. There is also the command line utility “mdadm” that can help with managing those devices[42].

Lastly, the init function is declared using “subsys_initcall” (and not the “module_init”) which ensures that it will run before the device drivers that needs it (if they are using “module_init”) — https://elixir.bootlin.com/linux/v6.1/source/drivers/md/md.c#L9947. More information about initcalls will be included on a future writeup.

[41] https://www.oreilly.com/library/view/managing-raid-on/9780596802035/ch01s03.html
[42] https://linux.die.net/man/8/mdadm

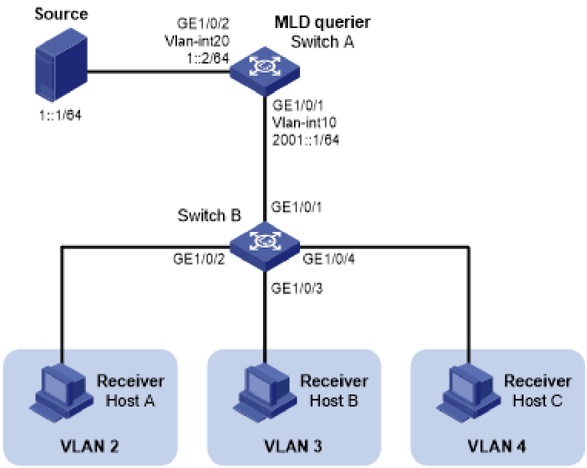
mld (Multicast Listener Discovery)

“mld” is a kernel thread which was created using a workqueue[43]. It is the Linux implementation for the multicast listener (MLD) protocol. This protocol is used by IPv6 based routers in order to discover multicast listeners on the local network and identify which multicast addresses are of interest to those listeners. MLD is supported on different operating systems such as Windows[44] and Linux[45].

We can think about it like IGMP[46] which is used on IPv4 based networks (MLDv1 is derived from IGMPv2 and MLDv2 is similar to IGMPv3). One important difference is that MLD uses ICMPv6 message types, rather than IGMP message types[47].

Overall, MLD has three major message types: “Multicast Listener Query”, “Multicast Listener Report” and “Multicast Done”. For more information about them I suggest reading the following link[48]. Also, a more detailed explanation about the different MLD operations can be found in https://ipcisco.com/lesson/mld-operations/.

What “mld” does is to send MLD report messages[49] which are sent by an MLD host (see the diagram below[50]) and processes messages[51]. From the source code we can see that there are definitions for structs representing both MLDv1 and MLDv2 headers.



[43] https://elixir.bootlin.com/linux/latest/source/net/ipv6/mcast.c#L3185
[44] https://learn.microsoft.com/en-us/windows/win32/winsock/igmp-and-windows-sockets
[45] https://lwn.net/Articles/29489/
[46] https://www.cloudflare.com/learning/network-layer/what-is-igmp/
[47] https://www.ibm.com/docs/en/zos/2.2.0?topic=protocol-multicast-listener-discovery
[48] https://community.cisco.com/t5/networking-knowledge-base/multicast-listener-discovery-mld/ta-p/3112082
[49] https://elixir.bootlin.com/linux/latest/source/net/ipv6/mcast.c#L3185
[50] https://techhub.hpe.com/eginfolib/networking/docs/switches/5130ei/5200-3944_ip-multi_cg/content/images/image33.png
[51] https://elixir.bootlin.com/linux/latest/source/net/ipv6/mcast.c#L1359

ksmd (Kernel Same Page Merging)

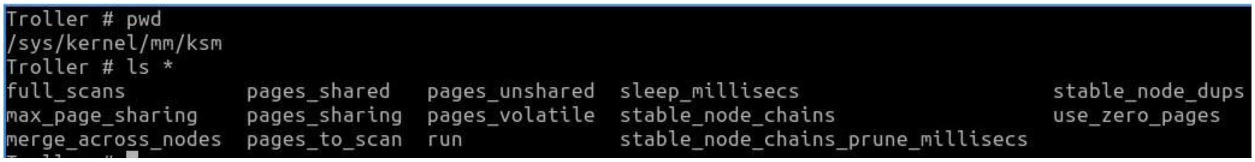
The kernel thread "ksm" is also known as "Kernel Same Page Merging" (and "ksmd" is ksm demon). It is used by the KVM hypervisor to share identical memory pages (supported since kernel 2.6.32) Those shared pages could be common libraries or even user data which is identical. By doing so KVM (Kernel-based Virtual Machine) can avoid memory duplication and enable more VMs to run on a single node.

In order for "ksmd" to save memory due to de-duplication we should compile the kernel with "CONFIG_KSM=y". It is important to understand that the sharing of identical pages is done even if they are not shared by fork(). If you want to go over "ksmd" source code you can use the following link - https://elixir.bootlin.com/linux/latest/source/mm/ksm.c.

The way "ksmd" works is as follows. Scanning main memory for frames ("physical pages") holding identical data and collectes the virtual memory address that they are mapped. "ksmd" leaves one of those frames and remaps each duplicate one to point to the same frame. Lastly, "ksmd" frees the other frames. All of the merge pages are marked as COW (Copy-on-Write) for cases in which one of the processes using them will want to write to the page. There is a concern that even if the memory usage is reduced the CPU usage is increased.

The kernel thread "ksmd" is created using the function kthread_run[52]. We can see from the code that the function which is the entry point of the thread is "ksm_scan_therad()" which is calling "ksm_do_scan()" which is the ksm's scanner main worker function (it gets as input the number of pages to scan before returning). "ksmd" only merges anonymous private pages and not pagecache. Historically, the merged pages were pinned into kernel memory. Today they can be swapped like any other pages.

"ksmd" can be controlled by a sysfs interface ("/sys/kernel/mm/ksm") - as can be seen in the screenshot below. One of the files exported by sysfs is "run" that can react to one of the following values 0/1/2. "0" means stop "ksmd" from running but keep the merged pages. "1" means run "ksmd". "2" means stop "ksmd" from running and unmerge all currently merge pages (however leave the mergeable areas registered for next time).



```
Troller # pwd
/sys/kernel/mm/ksm
Troller # ls *
full_scans          pages_shared   pages_unshared  sleep_millisecs                    stable_node_dups
max_page_sharing    pages_sharing  pages_volatile  stable_node_chains                 use_zero_pages
merge_across_nodes  pages_to_scan  run             stable_node_chains_prune_millisecs
```

[52] https://elixir.bootlin.com/linux/v6.0/source/mm/ksm.c#L3188

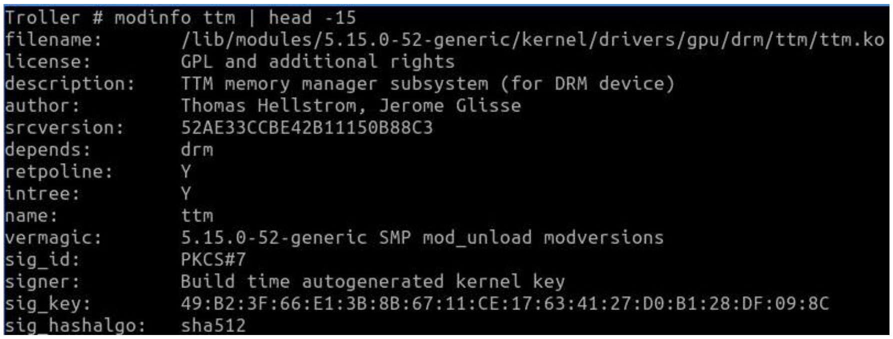
ttm_swap

The kernel thread "ttm_swap" is responsible for swapping GPU's (Graphical Processing Unit) memory. Overall, TTM (Translation-Table Maps) is a memory manager that is used to accelerate devices with dedicated memory. Basically, all the resources are grouped together by objects of buffers in different sizes. TTM then handles the lifetime, the movements and the CPU mapping of those objects[53].

Based on the kernel documentation, each DRM (Direct Rendering Manager) driver needs a memory manager. There are two memory managers supported by DRM: TTM and GEM (Graphics Execution Manager). I am not going to talk about GEM, if you want you can start reading about in the following link - https://docs.kernel.org/gpu/drm-internals.html.

Moreover, "ttm_swap" is a single threaded workqueue as seen in the Linux source code[54].
Also, the man pages describe TTM as a generic memory-manager provided by the kernel, which does not provide a user-space interface (API). In case we want to use it you should checkout the interface of each driver[55].

TTM is at the end a kernel module, you can find the source code and the Makefile in the kernel source tree[56]. Based on the module source code it is written by Thomas Hellstrom and Jerome Glisse[57]. Also, it is described as "TTM memory manager subsystem (for DRM device)"[58]. As you can see it is part of the "drivers/gpu/drm" subdirectory, which holds the code and Makefile of the drm device driver, which provides support for DRI (Direct Rendering Infrastructure) in XFee86 4.1.0+. Lastly, on my VM (Ubuntu 22.04.01) it is compiled as a separate "*.ko" file (/lib/modules/[KernelVersion]/kernel/drivers/gpu/drm/ttm.ko) - as shown in the screenshot below.



```
Troller # modinfo ttm | head -15
filename:       /lib/modules/5.15.0-52-generic/kernel/drivers/gpu/drm/ttm/ttm.ko
license:        GPL and additional rights
description:    TTM memory manager subsystem (for DRM device)
author:         Thomas Hellstrom, Jerome Glisse
srcversion:     52AE33CCBE42B11150B88C3
depends:        drm
retpoline:      Y
intree:         Y
name:           ttm
vermagic:       5.15.0-52-generic SMP mod_unload modversions
sig_id:         PKCS#7
signer:         Build time autogenerated kernel key
sig_key:        49:B2:3F:66:E1:3B:8B:67:11:CE:17:63:41:27:D0:B1:28:DF:09:8C
sig_hashalgo:   sha512
```

[53] https://docs.kernel.org/gpu/drm-mm.html
[54] https://elixir.bootlin.com/linux/v5.12.19/source/drivers/gpu/drm/ttm/ttm_memory.c#L424
[55] https://www.systutorials.com/docs/linux/man/7-drm-ttm/
[56] https://elixir.bootlin.com/linux/v6.1-rc2/source/drivers/gpu/drm/ttm
[57] https://elixir.bootlin.com/linux/v6.1-rc2/source/drivers/gpu/drm/ttm/ttm_module.c#L89
[58] https://elixir.bootlin.com/linux/v6.1-rc2/source/drivers/gpu/drm/ttm/ttm_module.c#L89

watchdogd

This kernel thread "watchdogd" is used in order to let the kernel know that a serious problem has occurred so the kernel can restart the system. It is sometimes called COP (Computer Operating Properly). The way it is implemented is by opening "/dev/watchdog", then writing at least once a minute. Every time there is a write the restart of the system is delayed.

In case of inactivity for a minute the watchdog should restart the system. Due to the fact we are not talking about a hardware watchdog the compilation of the operation depends on the state of the machine. You should know that the watchdog implementation could be software only (there are cases in which it won't restart the machine due to failure) or using a driver/module in case of hardware support[59].

If we are talking about hardware support then the watchdog module is specific for a chip or a device hardware. It is most relevant to systems that need the ability to restart themself without any human intervention (as opposed to a PC we can reboot easily) - think about an unmanned aircraft. We need to be careful because a problem in the watchdog configuration can lead to unpredictable reboot, reboot loops and even file corruption due to hard restart[60].

The relationship between the hardware and software is as follows: the hardware is responsible to set up the timer and the software is responsible to reset the timer. When the timer gets to a specific value (configured ahead) and it is not elapsed by the software the hardware will restart the system. For an example of using hardware for this functionality you can read the following link https://developer.toradex.com/linux-bsp/how-to/linux-features/watchdog-linux/.

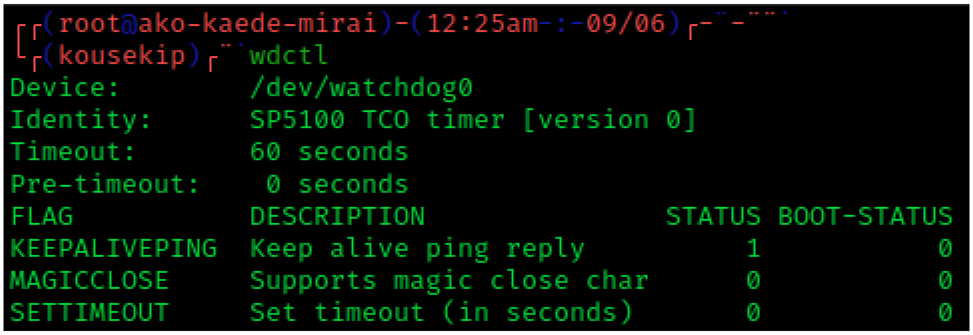
The software part is being conducted by the "watchdogd" (the software watchdog daemon) which opens "/dev/watchdog" and writes to it in order to postpone the restart of the system by the hardware - for more information you can read https://linux.die.net/man/8/watchdog. Examples for different watchdog drives/modules for specific chips can be found in the source tree of linux here https://elixir.bootlin.com/linux/v6.0.11/source/drivers/watchdog. Some example are apple_wdt (Apple's SOC), ath79_wdt (Atheros AR71XX/AR724X/AR913X) and w83977f_wdt (Winbond W83977F I/O Chip).

We can stop the watchdog without restarting the system by closing "/dev/watchdog". It is not possible if the kernel was compiled with "CONFIG_WATCHDOG_NOWAYOUT" enabled.

[59] https://github.com/torvalds/linux/blob/master/Documentation/watchdog/watchdog-api.rst
[60] https://linuxhint.com/linux-kernel-watchdog-explained/

Overall, in order for the watchdog to operate the kernel needs to be compiled with CONFIG_WATCHDOG=y and "/dev/watchdog" character device should be created (with major number of 10 and minor number of 130 - checkout "man mknod" if you want to create it).

Lastly, if you want to see the status of the watchdog you can use the command "wdctl"[61] - As can be seen in the screenshot below[62]. For more information about the concept I suggest reading https://en.wikipedia.org/wiki/Watchdog_timer.



```
┌┌(root@ako-kaede-mirai)-(12:25am-:-09/06)┌-¨-¨¨˙
└┌(kousekip)┌¨˙wdctl
Device:        /dev/watchdog0
Identity:      SP5100 TCO timer [version 0]
Timeout:       60 seconds
Pre-timeout:    0 seconds
FLAG           DESCRIPTION               STATUS BOOT-STATUS
KEEPALIVEPING  Keep alive ping reply          1           0
MAGICCLOSE     Supports magic close char      0           0
SETTIMEOUT     Set timeout (in seconds)       0           0
```

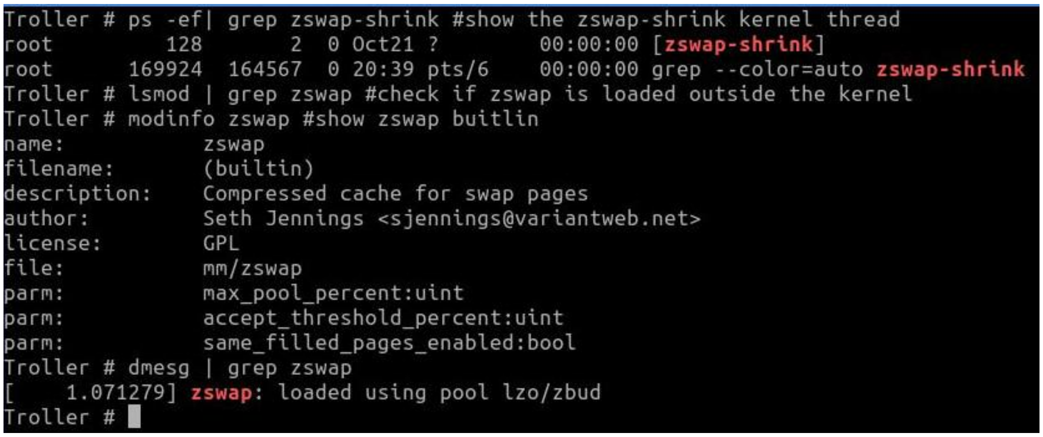
[61] https://man7.org/linux/man-pages/man8/wdctl.8.html
[62] https://en.wikipedia.org/wiki/Watchdog_timer#/media/File:Wdctl_screenshot.png

zswap-shrink

Based on the kernel source code zswap is a backend for frontswap. Frontswap provides a "transcendent memory" interface for swap pages. In some cases we can get increased performance by saving swapped pages in RAM (or a RAM-like device) and not on disk as swap partition\swapfile[63]. The frontends are usually implemented in the kernel while the backend is implemented as a kernel module (as we will show soon). Zswap takes pages that are in the process of being swapped out and attempts to compress and store them in a RAM-based memory pool[64].

We can say that zswap trades CPU cycles for potentially reduced swap I/O. A significant performance improvement can happen in case the reads from the swap device are much slower than the reads from the compressed cache[65]. The "zswap_frontswap_store" is the function that attempts to compress and store a single page[66].

The kernel thread "zswap-shrink" is created created based on a workqueue[67]. On my VM (Ubuntu 22.04.1) zswap is compiled part of the kernel itself and not as a separate "*.ko" (kernel module). You can see in the screenshot below that it does not appear in the output of "lsmod" and is marked as builtin (look at the filename field) in the output of "modinfo".



```
Troller # ps -ef| grep zswap-shrink #show the zswap-shrink kernel thread
root         128       2  0 Oct21 ?        00:00:00 [zswap-shrink]
root      169924  164567  0 20:39 pts/6    00:00:00 grep --color=auto zswap-shrink
Troller # lsmod | grep zswap #check if zswap is loaded outside the kernel
Troller # modinfo zswap #show zswap buitlin
name:           zswap
filename:       (builtin)
description:    Compressed cache for swap pages
author:         Seth Jennings <sjennings@variantweb.net>
license:        GPL
file:           mm/zswap
parm:           max_pool_percent:uint
parm:           accept_threshold_percent:uint
parm:           same_filled_pages_enabled:bool
Troller # dmesg | grep zswap
[    1.071279] zswap: loaded using pool lzo/zbud
Troller # 
```

For more information like the compression used by zswap (the default one is lzo) and other parameters that can be configured for zswap I suggest reading the following link https://wiki.archlinux.org/title/zswap. You can also read the parameter ons "/sys/module/zswap/parameters".

[63] https://www.kernel.org/doc/html/v4.18/vm/frontswap.html

[64] https://elixir.bootlin.com/linux/latest/source/mm/zswap.c

[65] https://www.kernel.org/doc/html/v4.18/vm/zswap.html

[66] https://elixir.bootlin.com/linux/v6.1-rc2/source/mm/zswap.c#L1097

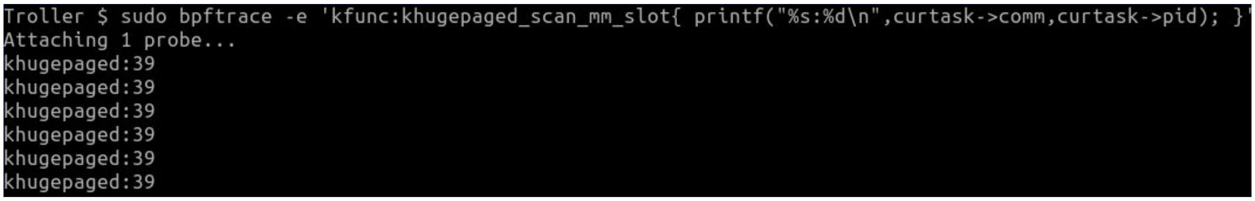
[67] https://elixir.bootlin.com/linux/v6.1-rc2/source/mm/zswap.c#L1511

khugepaged

The kernel thread “kugepaged” is created using the “kthread_run()” function[68]. It is responsible for the “Transparent Hugepage Support” (aka THP). “kugepaged” scans memory and collapses sequences of basic pages into huge pages[69].

We can manage and configure TPH using sysfs[70] or by using the syscalls “madvise”[71] and “prctl”[72]. The scan of memory is done by calling “khugepaged_do_scan()”[73] which in turn calls “khugepaged_scan_mm_slot()”[74]. In order to demonstrate that I have used the following bpftrace oneliner “**sudo bpftrace -e 'kfunc:khugepaged_scan_mm_slot{ printf("%s:%d\n",curtask->comm,curtask->pid); }'**”. The output is shown in the screenshot below.

Lastly, we can also monitor the modifications made by “khugepaged” by checking the information on “/proc”. For example we can check the “AnonHugePages”/”ShmemPmdMapped”/”ShmemHugePages” in “/proc/meminfo”, which is global for the entire system. If we want information regarding a specific process/task we can use “/proc/[PID]/smaps” and count “AnonHugePages”/”FileHugeMapped” for each mapping (https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html).



```
Troller $ sudo bpftrace -e 'kfunc:khugepaged_scan_mm_slot{ printf("%s:%d\n",curtask->comm,curtask->pid); }'
Attaching 1 probe...
khugepaged:39
khugepaged:39
khugepaged:39
khugepaged:39
khugepaged:39
khugepaged:39
```

[68] https://elixir.bootlin.com/linux/latest/source/mm/khugepaged.c#L2551
[69] https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html
[70] https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html#thp-sysfs
[71] https://man7.org/linux/man-pages/man2/madvise.2.html
[72] https://man7.org/linux/man-pages/man2/prctl.2.html
[73] https://elixir.bootlin.com/linux/latest/source/mm/khugepaged.c#L2404
[74] https://elixir.bootlin.com/linux/v6.1.12/source/mm/khugepaged.c#L2250

krfcommd

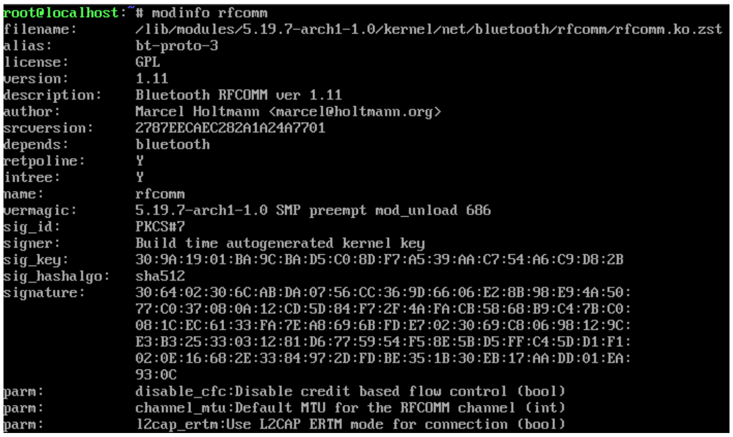
“krfcommd” is a kernel which is started by executing “kthread_run()” function[75]. The kernel thread executes the “rfcomm_run()” function[76]. Thus, we can say that “krfcommd” is responsible for RFCOMM connections[77].

RFCOMM (Radio Frequency Communication) is a set of transport protocols on top of L2CAP which provides emulated RS-232 serial ports. It provides a simple reliable data stream (like TCP). It is used directly by many telephony related profiles as a carrier for AT commands, as well as being a transport layer for OBEX over Bluetooth[78].

Moreover, there is also an “rfcomm” cli tool in Linux. It is used to inspect and maintain RFCOMM configuration[79]. For more information about RFCOMM I suggest reading https://www.btframework.com/rfcomm.htm. You can also go over the protocol specification[80].

Also, RFCOMM protocol supports up to 60 simultaneous connections between two Bluetooth devices. The number of connections that can be used simultaneously is implementation-specific. For the purposes of RFCOMM, a complete communication path involves two applications running on different devices (the communication endpoints) with a communication segment between them[81].

Lastly, RFCOMM is implemented as a kernel module. Thus, it can be compiled directly to the kernel or separate kernel module - in the screenshot below we can see it compiled as a separate file.



```
root@localhost:~# modinfo rfcomm
filename:       /lib/modules/5.19.7-arch1-1.0/kernel/net/bluetooth/rfcomm/rfcomm.ko.zst
alias:          bt-proto-3
license:        GPL
version:        1.11
description:    Bluetooth RFCOMM ver 1.11
author:         Marcel Holtmann <marcel@holtmann.org>
srcversion:     2787EECAEC282A1A24A7701
depends:        bluetooth
retpoline:      Y
intree:         Y
name:           rfcomm
vermagic:       5.19.7-arch1-1.0 SMP preempt mod_unload 686
sig_id:         PKCS#7
signer:         Build time autogenerated kernel key
sig_key:        30:9A:19:01:BA:9C:BA:D5:C0:8D:F7:A5:39:AA:C7:54:A6:C9:D8:2B
sig_hashalgo:   sha512
signature:      30:64:02:30:6C:AB:DA:07:56:CC:36:9D:66:06:E2:8B:98:E9:4A:50:
                77:C0:37:08:0A:12:CD:5D:84:F7:2F:4A:FA:CB:58:68:B9:C4:7B:C0:
                08:1C:EC:61:33:FA:7E:A8:69:6B:FD:E7:02:30:69:C8:06:98:12:9C:
                E3:B3:25:33:03:12:81:D6:77:59:54:F5:8E:5B:D5:FF:C4:5D:D1:F1:
                02:0E:16:68:2E:33:84:97:2D:FD:BE:35:1B:30:EB:17:AA:DD:01:EA:
                93:0C
parm:           disable_cfc:Disable credit based flow control (bool)
parm:           channel_mtu:Default MTU for the RFCOMM channel (int)
parm:           l2cap_ertm:Use L2CAP ERTM mode for connection (bool)
```

[75] https://elixir.bootlin.com/linux/latest/source/net/bluetooth/rfcomm/core.c#L2215
[76] https://elixir.bootlin.com/linux/latest/source/net/bluetooth/rfcomm/core.c#L2109
[77] https://stackoverflow.com/questions/57152408/what-is-the-internal-mechanics-of-socket-function
[78] https://en.wikipedia.org/wiki/List_of_Bluetooth_protocols
[79] https://linux.die.net/man/1/rfcomm
[80] https://www.bluetooth.com/specifications/specs/rfcomm-1-1/
[81] https://www.amd.e-technik.uni-rostock.de/ma/gol/lectures/wirlec/bluetooth_info/rfcomm.html

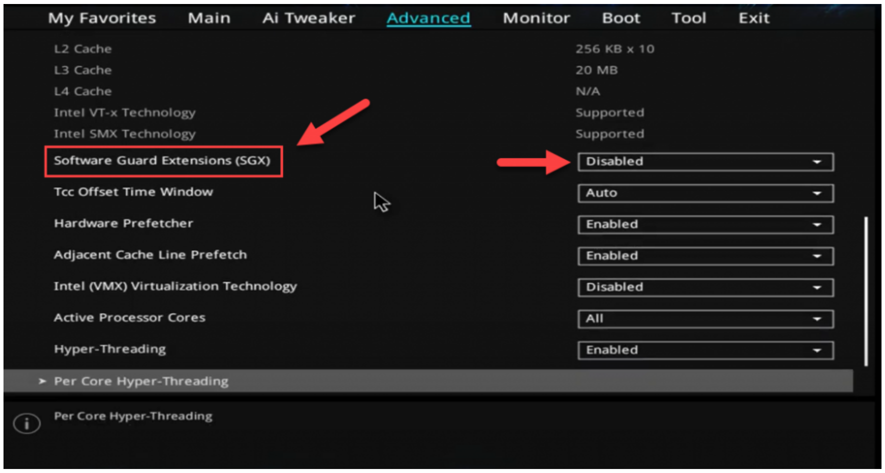
ksgxd

The kernel thread “ksgxd” is part of the Linux support for SGX (Software Guard eXtensions). Overall, SGX is a hardware security feature of Intel’s CPU that enables applications to allocate private memory regions for data and code. There is a privilege opcode “ENCLS” which allows creation of regions and “ENCLU” which is a privilege opcode that allows entering and executing code inside the regions[82]. For more information about SGX you can read my writeup about it[83].

“ksgxd” is a kernel which is started by executing “kthread_run()” function[84]. The kernel thread executes the “ksgxd” function[85]. “ksgxd” is started while SGX is initializing and at boot time it re-initializes all enclave pages. In case of over commitment “ksgxd” is also responsible for swapping enclave memory[86] like “kswapd”[87].

If you want to know if your CPU supports SGX you can use the following command: “cat /proc/cpuinfo | grep sgx” (you can also use lscpu). You can also check your UEFI (legacy BIOS) configuration to check if you - check out the screenshot below[88].

Lastly, there is a great guide for an example SGX app using a Linux VM on Azure that I encourage you to read[89]. For more information about the Linux stack for SGX I suggest reading https://download.01.org/intelsgxstack/2021-12-08/Getting_Started.pdf and going over the following github repo https://github.com/intel/linux-sgx.



L2 Cache	256 KB x 10
L3 Cache	20 MB
L4 Cache	N/A
Intel VT-x Technology	Supported
Intel SMX Technology	Supported
Software Guard Extensions (SGX)	Disabled
Tcc Offset Time Window	Auto
Hardware Prefetcher	Enabled
Adjacent Cache Line Prefetch	Enabled
Intel (VMX) Virtualization Technology	Disabled
Active Processor Cores	All
Hyper-Threading	Enabled

> Per Core Hyper-Threading

Per Core Hyper-Threading

[82] https://docs.kernel.org/x86/sgx.htmlhttps://docs.kernel.org/x86/sgx.html
[83] https://medium.com/@boutnaru/security-sgx-software-guard-extension-695cab7dbcb2
[84] https://elixir.bootlin.com/linux/v6.1.10/source/arch/x86/kernel/cpu/sgx/main.c#L427
[85] https://elixir.bootlin.com/linux/v6.1.10/source/arch/x86/kernel/cpu/sgx/main.c#L395
[86] https://elixir.bootlin.com/linux/v6.1.10/source/arch/x86/kernel/cpu/sgx/main.c#L188
[87] https://medium.com/@boutnaru/the-linux-process-journey-kswapd-22754e783901
[88] https://phoenixnap.com/kb/intel-sgx
[89] https://tsmatz.wordpress.com/2022/05/17/confidential-computing-intel-sgx-enclave-getting-started/

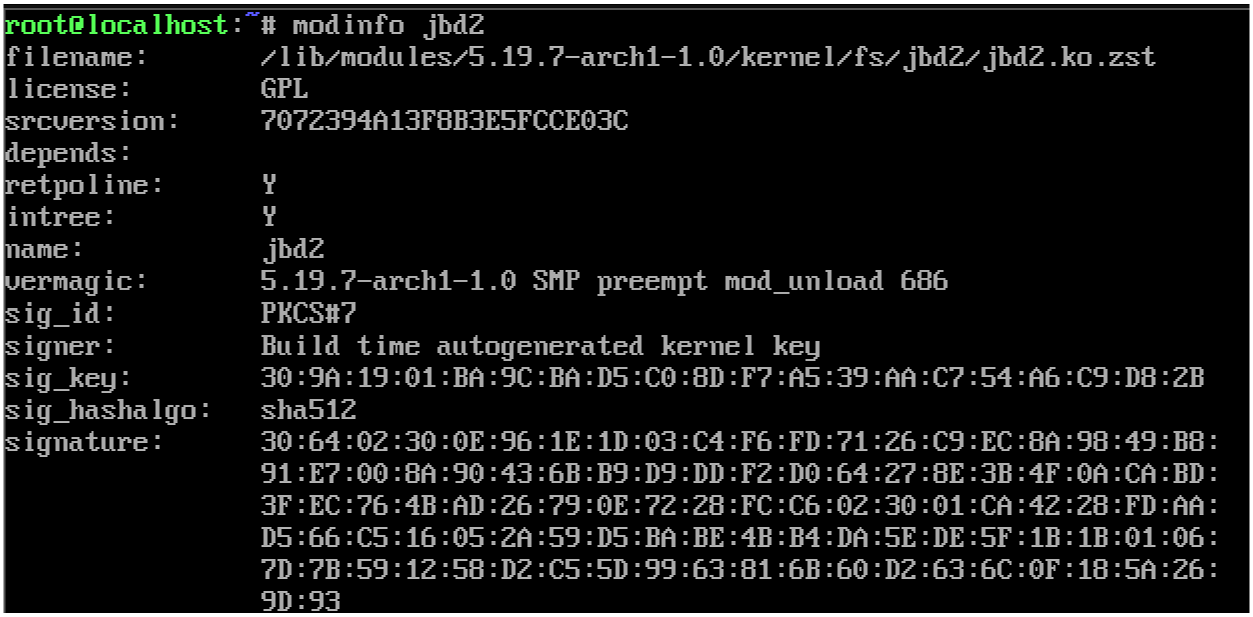
jbd2 (Journal Block Device 2)

“JBD” stands for “Journal Block Device”[90]. “jbd2” is a kernel which is started by executing “kthread_run()” function[91]. The name of the kernel thread has the following pattern “jbd2/[DeviceName]”. The code is part of a kernel module - as you can see in the screenshot below.

Moreover, as we can see from the code it is a file system journal-writing code (part of the ext2fs journaling system). The journal is an area of reserved disk space used for logging transactional updates. The goal of “jbd2” is to schedule updates to that log[92].

The kernel thread executes the “kjournald2()” function[93]. This main thread function is used to manage a logging device journal. Overall, the thread has two main responsibilities: commit and checkpoint. Commit is writing all metadata buffers of the journal. Checkpoint means flushing old buffers in order to reuse an “unused section” of the log file[94].

Lastly, JBD was written by Stephen Tweedie and it is filesystem independent. There are different filesystems that are using it like etx3,etx4 and OCFS2. There are two versions: JBD created in 1998 with ext3 and JBD2 forked from JBD in 2006 with ext4[95].



```
root@localhost:~# modinfo jbd2
filename:       /lib/modules/5.19.7-arch1-1.0/kernel/fs/jbd2/jbd2.ko.zst
license:        GPL
srcversion:     7072394A13F8B3E5FCCE03C
depends:
retpoline:      Y
intree:         Y
name:           jbd2
vermagic:       5.19.7-arch1-1.0 SMP preempt mod_unload 686
sig_id:         PKCS#7
signer:         Build time autogenerated kernel key
sig_key:        30:9A:19:01:BA:9C:BA:D5:C0:8D:F7:A5:39:AA:C7:54:A6:C9:D8:2B
sig_hashalgo:   sha512
signature:      30:64:02:30:0E:96:1E:1D:03:C4:F6:FD:71:26:C9:EC:8A:98:49:B8:
                91:E7:00:8A:90:43:6B:B9:D9:DD:F2:D0:64:27:8E:3B:4F:0A:CA:BD:
                3F:EC:76:4B:AD:26:79:0E:72:28:FC:C6:02:30:01:CA:42:28:FD:AA:
                D5:66:C5:16:05:2A:59:D5:BA:BE:4B:B4:DA:5E:DE:5F:1B:1B:01:06:
                7D:7B:59:12:58:D2:C5:5D:99:63:81:6B:60:D2:63:6C:0F:18:5A:26:
                9D:93
```

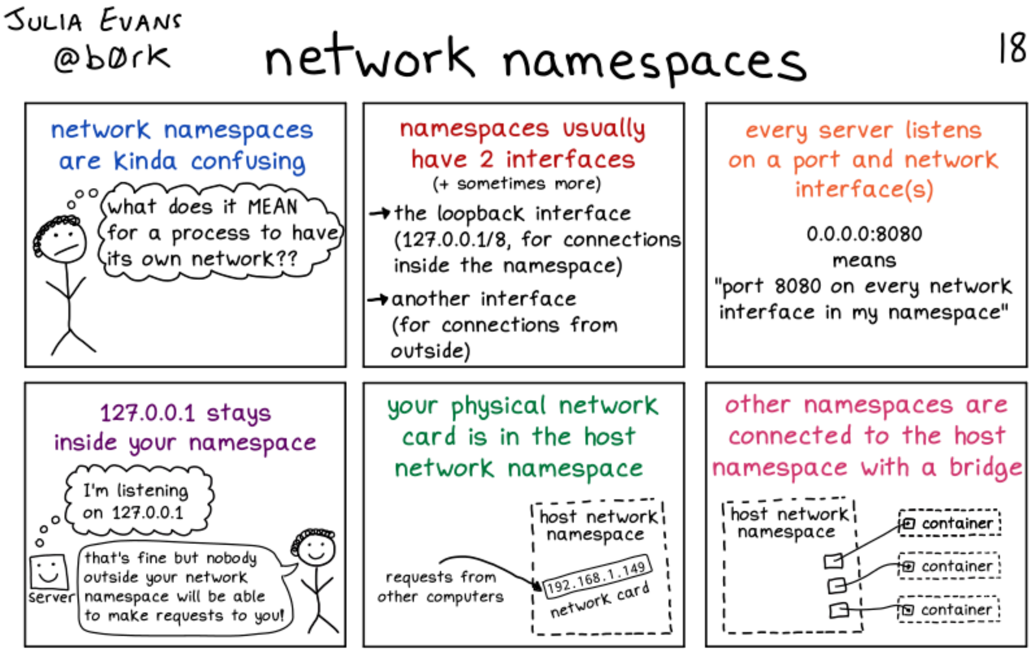
[90] https://manpages.ubuntu.com/manpages/jammy/man1/pmdajbd2.1.html
[91] https://elixir.bootlin.com/linux/v6.2.1/source/fs/jbd2/journal.c#L277
[92] https://elixir.bootlin.com/linux/v6.2.1/source/fs/jbd2/journal.c
[93] https://elixir.bootlin.com/linux/v6.2.1/source/fs/jbd2/journal.c#L169
[94] https://elixir.bootlin.com/linux/v6.2.1/source/fs/jbd2/journal.c#L152
[95] https://en.wikipedia.org/wiki/Journaling_block_device

netns

The kernel thread “netns” is based on a single threaded workqueue[96], which is created when the network namespace is initialized (net_ns_init()). If you want to read more about “network namespaces” you can use the following link https://medium.com/@boutnaru/linux-namespaces-network-namespace-part-3-7f8f8e06fef3. Also, for a reminder you can also check out the diagram below[97].

“netns” is responsible for cleaning up network namespaces. When a namespace is destroyed the kernel adds it to a cleanup list. The kernel thread “netns” goes over the list and performs the cleanup process using the “cleanup_net()” function[98].

If you want to see where all the magic happens is in “__put_net()” which queues the work on the “netns” to execute “cleanup_net()” function[99].


JULIA EVANS
@b0rk
network namespaces
18
network namespaces are kinda confusing
what does it MEAN for a process to have its own network??
namespaces usually have 2 interfaces
(+ sometimes more)
→ the loopback interface (127.0.0.1/8, for connections inside the namespace)
→ another interface (for connections from outside)
every server listens on a port and network interface(s)
0.0.0.0:8080 means "port 8080 on every network interface in my namespace"
127.0.0.1 stays inside your namespace
I'm listening on 127.0.0.1
server
that's fine but nobody outside your network namespace will be able to make requests to you!
your physical network card is in the host network namespace
host network namespace
requests from other computers
192.168.1.149
network card
other namespaces are connected to the host namespace with a bridge
host network namespace
container
container
container

[96] https://elixir.bootlin.com/linux/v6.2-rc4/source/net/core/net_namespace.c#L1106
[97] https://wizardzines.com/comics/network-namespaces/
[98] https://elixir.bootlin.com/linux/v6.2.3/source/net/core/net_namespace.c#L565
[99] https://elixir.bootlin.com/linux/v6.2-rc4/source/net/core/net_namespace.c#L649

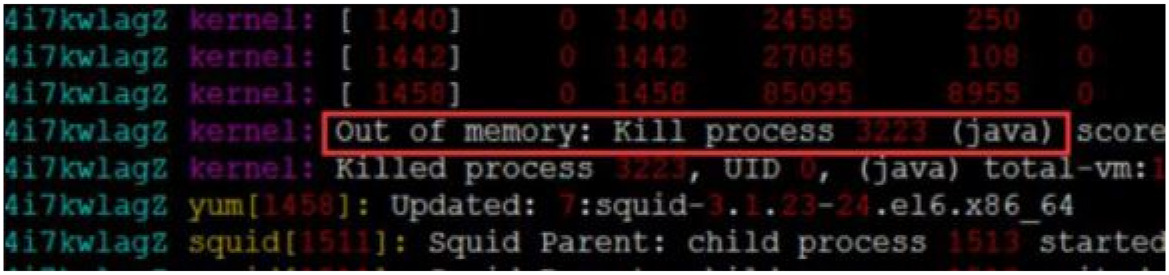
oom_reaper

“oom_reaper” is a kernel thread which was created using the “kthread_run” function[100]. Basically, it is the implementation of the OMM (Out–of-Memory) killer function of the Linux kernel - for more information about it I encourage you to read the following link https://medium.com/@boutnaru/linux-out-of-memory-killer-oom-killer-bb2523da15fc.

The function which is executed by the thread is “oom_reaper”[101] which calls “oom_reap_task”[102].

Based on the documentation the goal of the “oom_reaper” kernel thread is to try and reap the memory used by the OOM victim[103]. “oom_reaper” sleeps until it is waked up [104] which is after OOM kills the process[105].

After killing the process the victim is queued so the “oom_reaper” can release the resources[106]. You can see an example of the log created by OOM after killing a process[107].



```
4i7kwlagZ kernel: [ 1440]      0  1440    24585      250   0
4i7kwlagZ kernel: [ 1442]      0  1442    27085      108   0
4i7kwlagZ kernel: [ 1458]      0  1458    85095     8955   0
4i7kwlagZ kernel: Out of memory: Kill process 3223 (java) score
4i7kwlagZ kernel: Killed process 3223, UID 0, (java) total-vm:
4i7kwlagZ yum[1458]: Updated: 7:squid-3.1.23-24.el6.x86_64
4i7kwlagZ squid[1511]: Squid Parent: child process 1513 started
```

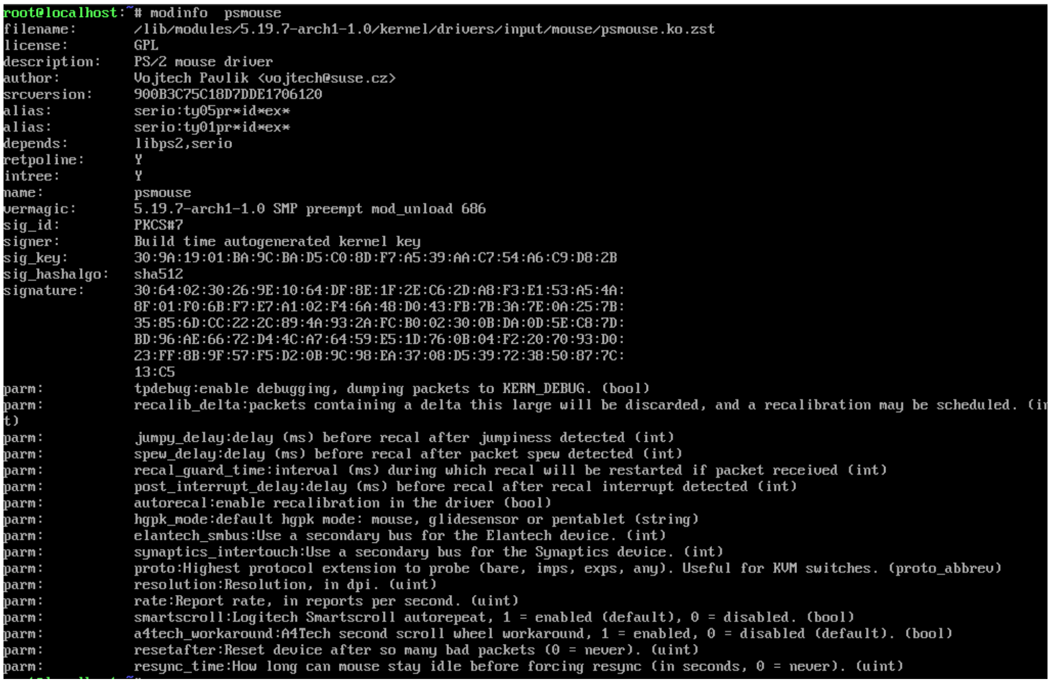
[100] https://elixir.bootlin.com/linux/v6.2.5/source/mm/oom_kill.c#L735
[101] https://elixir.bootlin.com/linux/v6.2.5/source/mm/oom_kill.c#L640
[102] https://elixir.bootlin.com/linux/v6.2.5/source/mm/oom_kill.c#L609
[103] https://elixir.bootlin.com/linux/v6.2.5/source/mm/oom_kill.c#L504
[104] https://elixir.bootlin.com/linux/v6.2.5/source/mm/oom_kill.c#L680
[105] https://elixir.bootlin.com/linux/v6.2.5/source/mm/oom_kill.c#L947
[106] https://elixir.bootlin.com/linux/v6.2.5/source/mm/oom_kill.c#L992
[107] https://blog.capdata.fr/index.php/linux-out-of-memory-killer-oom-killer-pour-un-serveur-base-de-donnees-postgresql/

kpsmoused

“kpsmoused” is a kernel thread which based on an ordered workqueue[108] which is allocated inside the “pmouse_init” function. “kpsmoused” is responsible for handling the input from PS/2 mouse devices.

Thus, “kpsmoused” transforms the raw data to high level event of mouse movements that be can consume from “/dev/input/mice”, “/dev/input/mouseX”, or “/dev/input/eventX”[109].

The kernel thread is created by the “psmouse” kernel module which is described as “PS/2 mouse driver” - as shown in the screenshot below (which was created using copy.sh). By the way, the “kpsmoused” is created as part of “/drivers/input/mouse/psmouse-base.c” since kernel 2.5.72[110].



```
root@localhost:~# modinfo  psmouse
filename:       /lib/modules/5.19.7-arch1-1.0/kernel/drivers/input/mouse/psmouse.ko.zst
license:        GPL
description:    PS/2 mouse driver
author:         Vojtech Pavlik <vojtech@suse.cz>
srcversion:     900B3C75C18D7DDE1706120
alias:          serio:ty05pr*id*ex*
alias:          serio:ty01pr*id*ex*
depends:        libps2,serio
retpoline:      Y
intree:         Y
name:           psmouse
vermagic:       5.19.7-arch1-1.0 SMP preempt mod_unload 686
sig_id:         PKCS#7
signer:         Build time autogenerated kernel key
sig_key:        30:9A:19:01:BA:9C:BA:D5:C0:8D:F7:A5:39:AA:C7:54:A6:C9:D8:2B
sig_hashalgo:   sha512
signature:      30:64:02:30:26:9E:10:64:DF:8E:1F:2E:C6:2D:A8:F3:E1:53:A5:4A:
                8F:01:F0:6B:F7:E7:A1:02:F4:6A:48:D0:43:FB:7B:3A:7E:0A:25:7B:
                35:85:6D:CC:22:2C:89:4A:93:2A:FC:B0:02:30:0B:DA:0D:5E:C8:7D:
                BD:96:AE:66:72:D4:4C:A7:64:59:E5:1D:76:0B:04:F2:20:70:93:D0:
                23:FF:8B:9F:57:F5:D2:0B:9C:98:EA:37:08:D5:39:72:38:50:87:7C:
                13:C5
parm:           tpdebug:enable debugging, dumping packets to KERN_DEBUG. (bool)
parm:           recalib_delta:packets containing a delta this large will be discarded, and a recalibration may be scheduled. (in
t)
parm:           jumpy_delay:delay (ms) before recal after jumpiness detected (int)
parm:           spew_delay:delay (ms) before recal after packet spew detected (int)
parm:           recal_guard_time:interval (ms) during which recal will be restarted if packet received (int)
parm:           post_interrupt_delay:delay (ms) before recal after recal interrupt detected (int)
parm:           autorecal:enable recalibration in the driver (bool)
parm:           hgpk_mode:default hgpk mode: mouse, glidesensor or pentablet (string)
parm:           elantech_smbus:Use a secondary bus for the Elantech device. (int)
parm:           synaptics_intertouch:Use a secondary bus for the Synaptics device. (int)
parm:           proto:Highest protocol extension to probe (bare, imps, exps, any). Useful for KVM switches. (proto_abbrev)
parm:           resolution:Resolution, in dpi. (uint)
parm:           rate:Report rate, in reports per second. (uint)
parm:           smartscroll:Logitech Smartscroll autorepeat, 1 = enabled (default), 0 = disabled. (bool)
parm:           a4tech_workaround:A4Tech second scroll wheel workaround, 1 = enabled, 0 = disabled (default). (bool)
parm:           resetafter:Reset device after so many bad packets (0 = never). (uint)
parm:           resync_time:How long can mouse stay idle before forcing resync (in seconds, 0 = never). (uint)
```

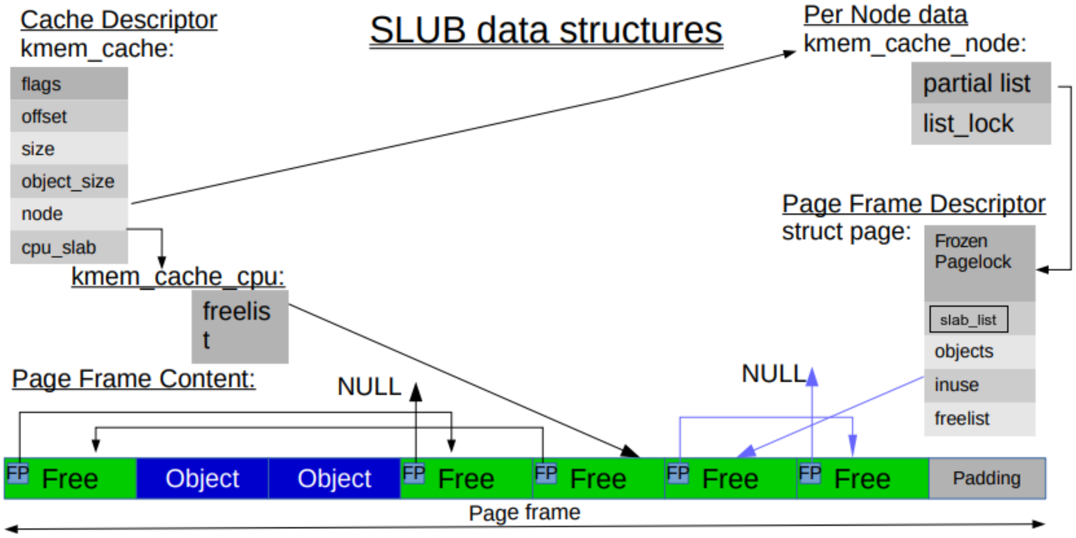
[108] https://elixir.bootlin.com/linux/v6.2.6/source/drivers/input/mouse/psmouse-base.c#L2046
[109] https://www.kernel.org/doc/html/v5.5/input/input.html
[110] https://elixir.bootlin.com/linux/v2.5.72/source/drivers/input/mouse/psmouse-base.c

slub_flushwq

“slub_flushwq” is a kernel thread which based on a workqueue[111] which is allocated inside the “kmem_cache_init_late” function. Based on the source code the allocation is done only if “CONFIG_SLUB_TINY” is enabled[112]. From the documentation “CONFIG_SLUB_TINY” is for configuring SLUB allocation in order to achieve minimal memory footprint, it is not recommended for systems with more than 16 GB of RAM[113]. The queuing of work is done inside the “flush_all_cpus_locked” function[114].

SLUB is also known as the “Unqueued Slab Allocator”[115]. Slab allocation is a memory management mechanism which allows efficient memory allocation of objects. It is done using reduction of fragmentation that is caused due to allocations/deallocations[116]. For more information about slab allocation I suggest reading the following link https://hammertux.github.io/slab-allocator.

Thus, SLUB is a slab allocator that limits the use of cache lines instead of using queued object per cpu/per node list[117]. So, it is less complicated because it does not keep queues (like for each CPU). The only queue is a linked list for all the objects in each of the slub pages[118]. The interplay between the three main data structures (kmem_cache, kmem_cache_cpu, kmem_cache_node) used by the SLUB allocator is shown in the diagram below[119] .



[111] https://elixir.bootlin.com/linux/v6.2.6/source/mm/slub.c#L5057
[112] https://elixir.bootlin.com/linux/v6.2.6/source/mm/slub.c#L5056
[113] https://cateee.net/lkddb/web-lkddb/SLUB_TINY.html
[114] https://elixir.bootlin.com/linux/v6.2.6/source/mm/slub.c#L2822
[115] https://lwn.net/Articles/229096/
[116] https://en.wikipedia.org/wiki/Slab_allocation
[117] https://elixir.bootlin.com/linux/v6.2.6/source/mm/slub.c#L3
[118] https://hammertux.github.io/slab-allocator
[119] https://hammertux.github.io/img/SLUB-DS.png

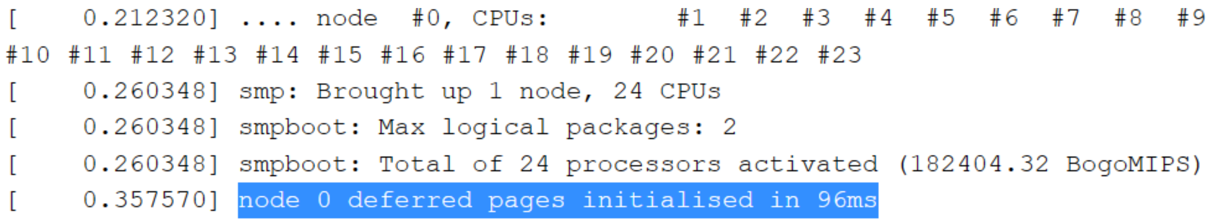
pgdatinit

“pgdatinit” is a kernel which is started by executing the “kthread_run()” function[120]. The kernel thread executes the “deferred_init_memmap()” function[121].

Thus, “pgdatinit” is responsible for initializing memory on every node of the system. For each node a dedicated kernel thread is created with the name pattern “pgdatinit[NodeNumber]”[122].

Overall, the kernel thread is created in case CONFIG_DEFERRED_STRUCT_PAGE_INIT is enabled when compiling the kernel. Which states that initialization of struct pages is deferred to kernel threads[123].

Lastly, after the initialization flow is finished an information message is sent to the kernel ring buffer[124] - as you can see in the image below[125].



```
[    0.212320] .... node  #0, CPUs:        #1   #2   #3   #4   #5   #6   #7   #8   #9
#10 #11 #12 #13 #14 #15 #16 #17 #18 #19 #20 #21 #22 #23
[    0.260348] smp: Brought up 1 node, 24 CPUs
[    0.260348] smpboot: Max logical packages: 2
[    0.260348] smpboot: Total of 24 processors activated (182404.32 BogoMIPS)
[    0.357570] node 0 deferred pages initialised in 96ms
```

[120] https://elixir.bootlin.com/linux/v6.3-rc4/source/mm/page_alloc.c#L2284
[121] https://elixir.bootlin.com/linux/v6.3-rc4/source/mm/page_alloc.c#L2108
[122] https://elixir.bootlin.com/linux/v6.3-rc4/source/mm/page_alloc.c#L2283
[123] https://cateee.net/lkddb/web-lkddb/DEFERRED_STRUCT_PAGE_INIT.html
[124] https://elixir.bootlin.com/linux/v6.3-rc4/source/mm/page_alloc.c#L2177
[125] https://www.mail-archive.com/debian-bugs-dist@lists.debian.org/msg1822096.html

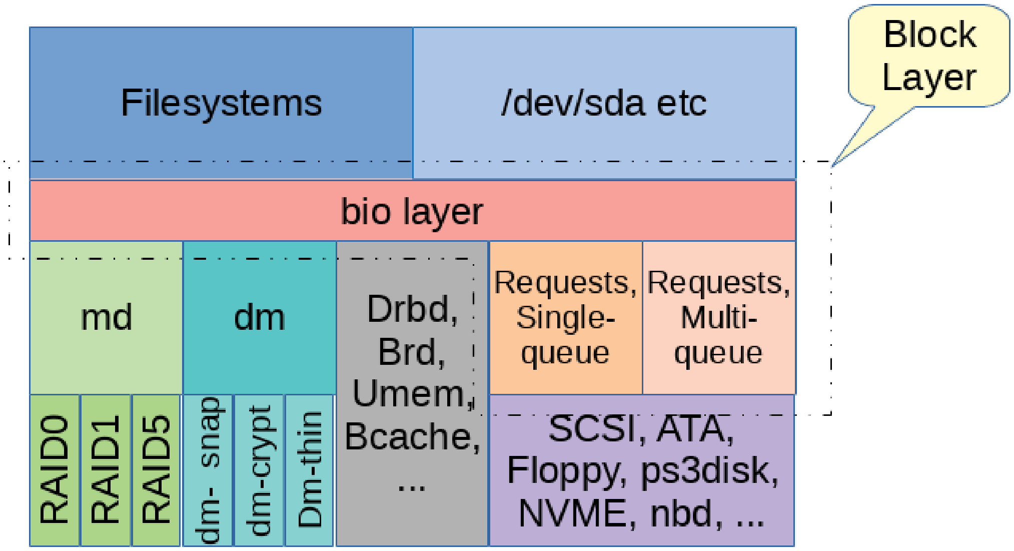
kblockd

“kblockd” is a kernel thread based on a workqueue[126] which is marked with high priority and that it can be used for memory reclaim. It is used for performing I/O disk operations.

Moreover, we can deduct based on the location of the file in the Linux source tree (/block) that “kblockd” is part of the “Block Layer” (which is responsible for managing block devices) - as shown in the diagram below[127].

Overall, one might think that we can use keventd[128] for performing I/O operations. However, because they can get blocked on disk I/O. Due to that, “kblockd” was created to run low-level disk operations like calling relevant block device drivers[129].

Thus, “kblockd” must never block on disk I/O so all the memory allocations should be GFP_NOIO. We can sum up that it is used to handle all read/writes requests to block devices[130].



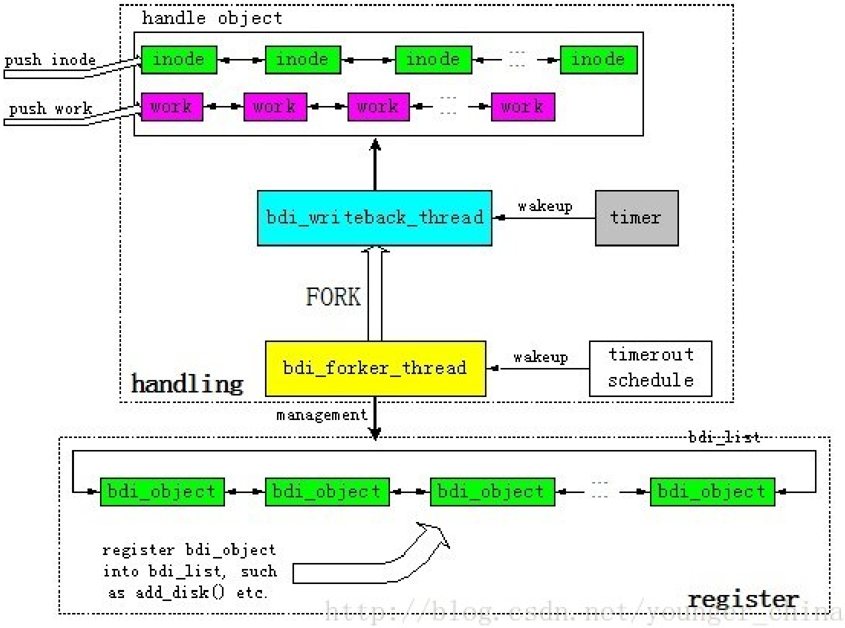
[126] https://elixir.bootlin.com/linux/v6.2.9/source/block/blk-core.c#L1191
[127] https://lwn.net/Articles/736534/
[128] https://lwn.net/Articles/11351/
[129] https://mirrors.edge.kernel.org/pub/linux/kernel/people/akpm/patches/2.5/2.5.70/2.5.70-mm8/broken-out/kblockd.patch
[130] https://elixir.bootlin.com/linux/v6.3-rc4/source/block/blk-core.c#L13

writeback

The kernel thread “writeback” is based on a workqueue[131]. The goal of the kernel thread is to serve all async writeback tasks[132]. Thus, “writeback” is flushing dirty information from the page cache (aka disk cache) to disks. The page cache is the main disk cache used by the kernel. The kernel references the page cache when reading from/writing to disk[133].

Overall, they are two ways of flushing dirty pages using writeback. The first is in case of an explicit writeback request - like syncing inode pages of a superblock. Thus, the “wb_start_writeback()” is called with the superblock information and the number of pages to flush. The second one is when there is no specific writeback request, in this case there is a timer that wakes up the thread periodically to flush dirty data[134].

Moreover, from kernel 3.2 the original mechanism of “pdflush” was changed to “bdi_writeback”. By doing so it solves one of the biggest limitations of “pdflush” in a multi-disk environment. In that case “pdflush” manages the buffer/page cache of all the disks which creates an IO bottleneck. On the other hand, “bdi_writeback” creates a thread for each disk[135]. By the way, “bdi” stands for “Backing Device Information”[136]. Lastly, to get an overview of the “writeback” mechanism you can checkout the diagram below[137].



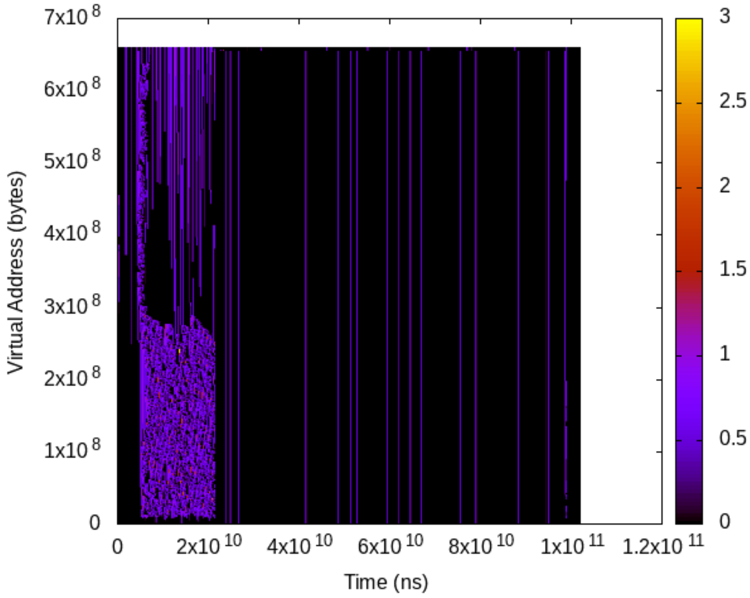
[131] https://elixir.bootlin.com/linux/v6.2.5/source/mm/backing-dev.c#L363
[132] https://elixir.bootlin.com/linux/v6.2.5/source/mm/backing-dev.c#L35
[133] https://www.oreilly.com/library/view/understanding-the-linux/0596005652/ch15s01.html
[134] https://lwn.net/Articles/326552/
[135] https://blog.csdn.net/younger_china/article/details/55187057
[136] https://lwn.net/Articles/326552/
[137] https://blog.csdn.net/younger_china/article/details/55187057

kdamond (Data Access MONitor)

“kdamond” is a kernel thread which is created using the “kthread_run()” function[138] which is part of the DAMON (Data Access MONitor) subsystem. The kernel thread executes the “kdamon_fn()” function[139].Overall, DAMON provides a lightweight data access monitoring facility that can help users in analyzing the memory access patterns of their systems[140]. Based on the documentation DAMON increases the memory usage by 0.12% and slows the workloads down by 1.39%[141].

Also, DAMON has an API for kernel programs[142]. Moreover, there is also DAMOS (DAMon-Based Operations Schemas). Using that, users can develop and run access-aware memory management with no code and just using configurations[143].

Probably the best way to go over DAMON data is by using visualization. A great demonstration for that has been done by SeongJae Park using the PARSEC3/SPLASH-2X benchmarks[144]. The output was heatmaps of the dynamic access patterns for heap area, mmap()ed area and the stack area. One example is shown in the image below, it visualizes the data access pattern of the stack area when running the parsec3-blackscholes[145]. Lastly, there are also other mechanisms in Linux that can help with data access monitoring such as “Perf Mem” and “Idle Page Tracking”



[138] https://elixir.bootlin.com/linux/v6.3-rc5/source/mm/damon/core.c#L632
[139] https://elixir.bootlin.com/linux/v6.3-rc5/source/mm/damon/core.c#L1304
[140] https://www.kernel.org/doc/html/latest/admin-guide/mm/damon/index.html
[141] https://damonitor.github.io/doc/html/v20/vm/damon/eval.html
[142] https://www.kernel.org/doc/html/v5.17/vm/damon/api.html#functions
[143] https://sjp38.github.io/post/damon/
[144] https://parsec.cs.princeton.edu/parsec3-doc.htm
[145] https://lwn.net/Articles/813108/

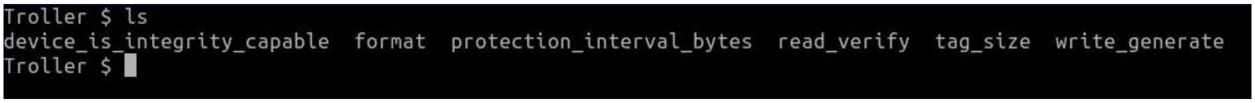
kintegrityd

“kintegrityd” is a kernel thread based on a workqueue[146] which is responsible for verifying the integrity of block devices by reading/writing data from/to them. The function which is executed by the workqueue is “bio_integrity_verify_fn”[147]. The function is called to complete a read request by verifying the transferred integrity metadata and then calls the original bio end_io function[148].

This procedure is done to ensure that the data was not changed by mistake (like in a case of a bug or an hardware failure[149]. This mechanism is also called “bio data integrity extensions“. And it allows the user to get protection for the entire flow: from the application to storage device. The implementation is transparent to the application itself and it is part of the block layer[150].

Moreover, in order for it to work we should enable CONFIG_BLK_DEV_INTEGRITY, which is defined as “Block layer data integrity support”[151]. The filesystem does not have to be aware that the block device can include integrity metadata. The metadata is generated as part of the block layer when calling the submit_bio() function[152]. We can toggle the writing of metadata using “/sys/block/<BlockDevice>/integrity/write_generate“ and the verification of the metadata using “/sys/block/<BlockDevice>/integrity/read_verify” - as shown in the screenshot below.

Lastly, there are also file systems which are integrity aware (and they will generate/verify the metadata). There are also options for sending the metadata information from userspace, for more information I suggest reading the following Linux’s kernel documentation https://www.kernel.org/doc/Documentation/block/data-integrity.txt.



```
Troller $ ls
device_is_integrity_capable  format  protection_interval_bytes  read_verify  tag_size  write_generate
Troller $ █
```

[146] https://elixir.bootlin.com/linux/v6.1/source/block/bio-integrity.c#L455
[147] https://elixir.bootlin.com/linux/v6.1/source/block/bio-integrity.c#L317
[148] https://elixir.bootlin.com/linux/v6.1/source/block/bio-integrity.c#L313
[149] https://www.quora.com/What-is-the-purpose-of-kintegrityd-Linux-Kernel-Daemon/answer/Liran-Ben-Haim
[150] https://www.kernel.org/doc/Documentation/block/data-integrity.txt
[151] https://elixir.bootlin.com/linux/v6.1/source/block/Kconfig#L60
[152] https://www.kernel.org/doc/Documentation/block/data-integrity.txt

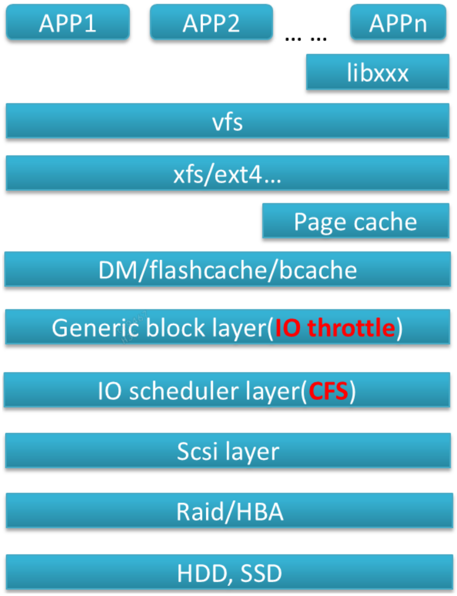
kthrotld

“kthrotld” is a kernel thread which was created using an workqueue[153] which acts as an interface for controlling IO bandwidth on request queues (throttling requests). Overall, read and write requests to block devices are placed on request queues[154].

In order to understand how request queues are used the best way is to check the source code of the kernel. The first step is going over the definition of “struct request_queue”[155] and then where is it referenced[156]. By the way, in kernel version 6.1.1 it is referenced in 199 files. We can summarize that a request queue holds I/O requests in a linked list. Also, it is a best practice to create a separate request queue for every device[157].

Thus, we can say that “kthrotld” acts as a block throttle, which provides block QoS (Quality of Service). It is used to limit IOPS (I/O per second)/BPS (Bits per second) per cgroup (control group)[158].

Overall, IO throttling is done as part of the generic block layer and before the IO scheduler as seen in the diagram below[159]. For more information on “Block Throttling” I suggest reading https://developer.aliyun.com/article/789736.



[153] https://elixir.bootlin.com/linux/v6.1.1/source/block/blk-throttle.c#L2470
[154] https://www.halolinux.us/kernel-architecture/request-queues.html
[155] https://elixir.bootlin.com/linux/v6.1.1/source/include/linux/blkdev.h#L395
[156] https://elixir.bootlin.com/linux/v6.1.1/C/ident/request_queue
[157] https://www.oreilly.com/library/view/linux-device-drivers/0596000081/ch12s04.html
[158] https://developer.aliyun.com/article/789736
[159] https://blog.csdn.net/yiyeguzhou100/article/details/104044419

scsi_eh (Small Computer System Interface Error Handling)

The kernel thread “scsi_eh” is executed using the “kthread_run” function. The name pattern of the kernel thread is “scsi_eh_<SCSI_HOST_NUMBER>”[160]. It is the “SCSI error handler” which is responsible for all of the error handling targeting every SCSI host[161]. The kernel thread is executing the “scsi_error_handler” function[162].

Moreover, a SCSI controller which coordinates between other devices on the SCSI bus is called a “host adapter”. It can be a card connected to a slot or part of the motherboard. You can see an example of a SCSI connector in the image below[163].

Lastly, SCSI stands for “Small Computer System Interface”. It is a set of standards (from ANSI) for electronic interfaces in order to communicate with peripheral hardware like CD-ROM drives, tap drivers, printers, disk drives and more[164].. For more information about SCSI I suggest going over https://hackaday.com/2023/03/02/scsi-the-disk-bus-for-everything/.

© 2006 HowStuffWorks

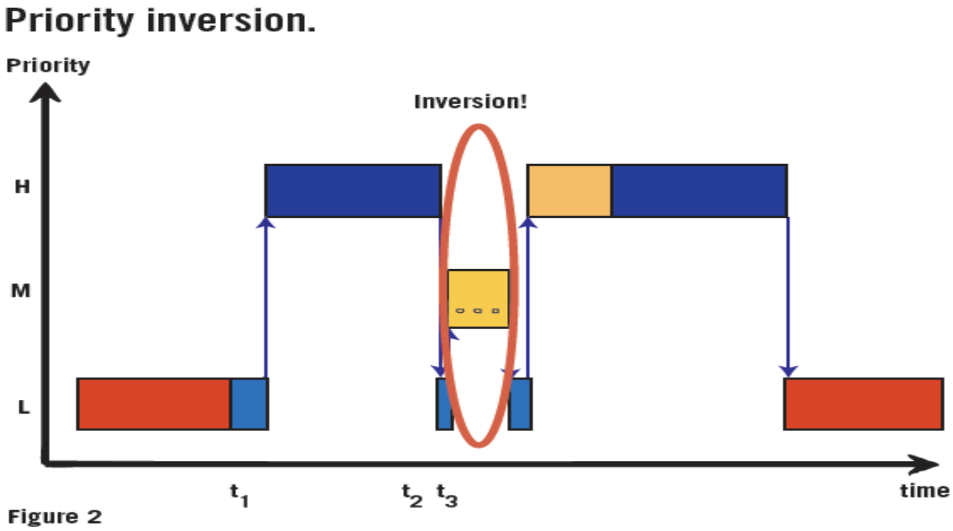
[160] https://elixir.bootlin.com/linux/v6.4-rc1/source/drivers/scsi/hosts.c#L504
[161] https://elixir.bootlin.com/linux/v6.4-rc1/source/drivers/scsi/scsi_error.c#L2230
[162] https://elixir.bootlin.com/linux/v6.4-rc1/source/drivers/scsi/scsi_error.c#L2233
[163] https://computer.howstuffworks.com/scsi.htm
[164] https://www.techtarget.com/searchstorage/definition/SCSI

blkcg_punt_bio

“blkcg_punt_bio” is a kernel thread based on a workqueue. The workqueue itself is created in the “blkcg_init” function[165]. It is part of the common block controller cgroup interface[166].

Overall, when a shared kernel thread tries to issue a synchronized block I/O (bio) request for a specific cgroup it can lead to a priority inversion. It can happen if the kernel thread is blocked waiting for that cgroup[167]. An example of priority inversion is shown in the diagram below[168].

Thus, to avoid the problem mentioned above the function “submit_bio”[169] punts the issuing of the bio request to a dedicated work item (per-block cgroup).
It calls “blkcg_punt_bio_submit”[170], which will call “__blkcg_punt_bio_submit”[171].

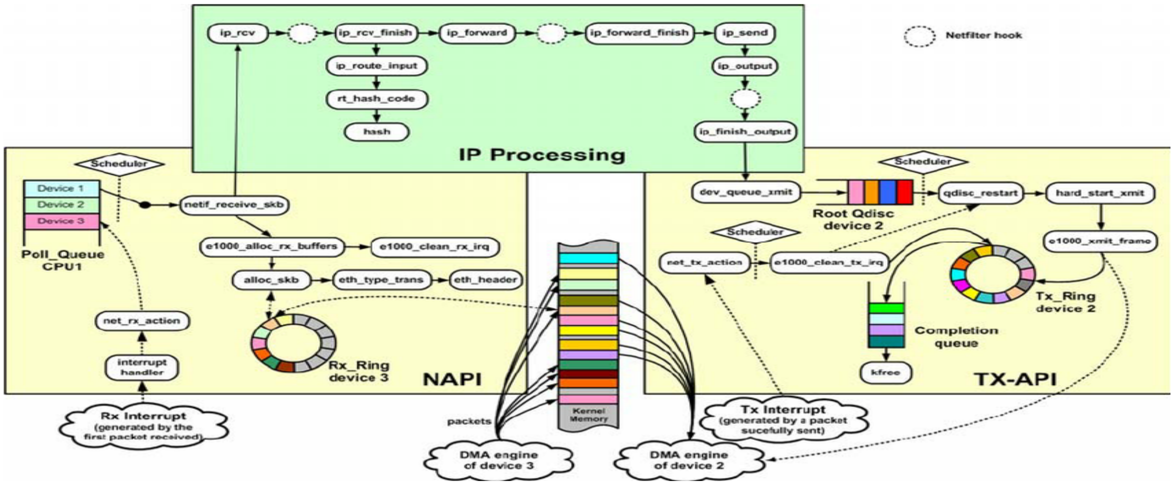


[165] https://elixir.bootlin.com/linux/v6.2.5/source/block/blk-cgroup.c#L2058
[166] https://elixir.bootlin.com/linux/v6.2.5/source/block/blk-cgroup.c#L3
[167] https://patchwork.kernel.org/project/linux-block/patch/20190627203952.386785-6-tj@kernel.org/
[168] https://embeddedgurus.com/barr-code/2010/11/firmware-specific-bug-8-priority-inversion/
[169] https://elixir.bootlin.com/linux/v6.2.5/source/block/blk-core.c#L829
[170] https://elixir.bootlin.com/linux/v6.2.5/source/block/blk-cgroup.h#L380
[171] https://elixir.bootlin.com/linux/v6.2.5/source/block/blk-cgroup.c#L1657

napi (New API)

NAPI stands for “New API” which is used to reduce the number of received interrupts. Think about cases in which the network driver receives a large number of packets at a fast pace[172]. If we think about it in the case of a Gigabit network card and an MTU of 1500 the CPU will get about 90K of interrupt per second. Thus, we can say that NAPI is an extension to the Linux packet processing framework, which is done for improving performance for high speed networking. This is performed using interrupt mitigation and packet throttling. It is important to say that the addition of NAPI does not break backward compatibility[173]. “napi” is a kernel thread which is created using the “kthread_run()”[174] function which is part of the NAPI (New API) subsystem. The name of the kernel thread is based on the pattern “napi[DeviceName]-[NAPI-ID]”. It executes the “napi_threaded_poll”[175] function.

Due to that, drivers that support NAPI can disable hardware interrupts as a mechanism for packet reception. In that case the network stack relies on polling for new packets at a specific interval. It might seem that polling is less efficient but in case the network device is busy any time the kernel will poll for a packet it will get something[176]. Lastly, the way NAPI does that is by combining hardware interrupts and polling. When a hardware interrupt is received, the driver disables it and notifies the kernel to read the packets.Then a kernel software interrupt polls the network device for a specific time. When the time runs out/there is no more data the kernel will enable the hardware interrupt again[177]. A detailed diagram of the NAPI flow is shown in the diagram below[178].



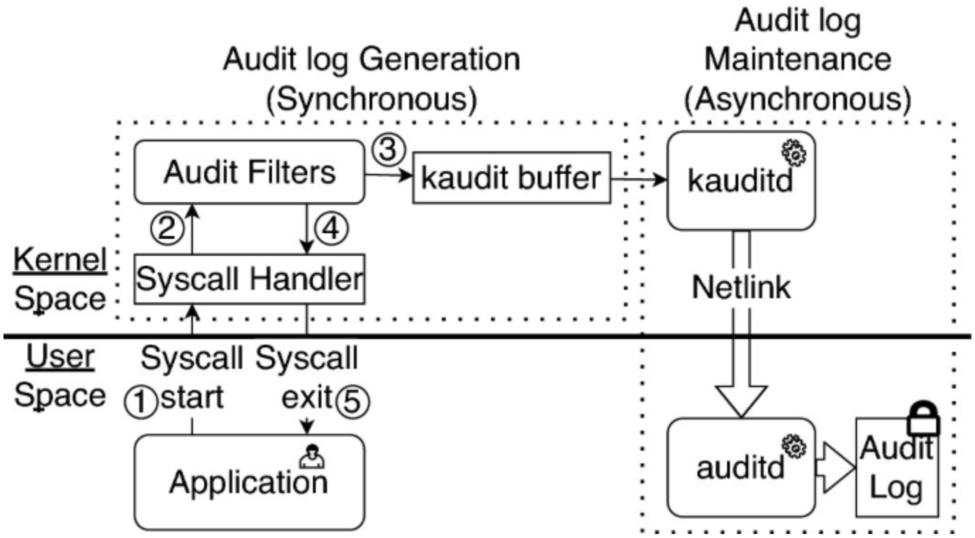
[172] https://www.hitchhikersguidetolearning.com/2023/04/09/handling-receive-packets-via-napi/
[173] https://wiki.linuxfoundation.org/networking/napi
[174] https://elixir.bootlin.com/linux/v6.4-rc4/source/net/core/dev.c#L1371
[175] https://elixir.bootlin.com/linux/v6.4-rc4/source/net/core/dev.c#L662
[176] https://lwn.net/Articles/833840/
[177] https://www.jianshu.com/p/7d4e36c0abe8
[178] https://www.researchgate.net/profile/Roberto-Bruschi-2/publication/228624515/figure/fig4/AS:301797211164675@1448965470134/Detailed-scheme-of-the-forwarding-operations-in-26-kernel-NAPI.png

kauditd (Kernel Audit Daemon)

“kauditd” is a kernel thread which is started using the “kthread_run” function[179]. The kernel thread is calling the “kauditd_thread” function, this function is responsible for sending audit logs to userspace[180].Overall, the kernel mechanism in the Linux kernel has a couple of goals: integrate fully with LSMs[181], minimal run-time overhead when performing auditing, ability to disable system call auditing at boot time, allow to be used by other parts of the kernel for auditing, netlink interface to userspace and support for filtering to minimize the information sent to user-mode[182].

Thus, we can say “kauditd” is the kernel component of the “Linux Auditing System” which handles the audit events - as shown in the diagram below[183] . In order to configure which set of rules are going to be loaded in the kernel audit system we can use the “/etc/audit/audit.rules” file. This file can hold configuration in one of three categories: control (configuring the audit system), file system rules monitoring rules and system call monitoring rules[184].

Lastly, by using the “Linux Auditing System” the system administrator can investigate what happens in the system for the purpose of debugging or in case of a security incident. We can also use the “auditctl” utility get/add/delete rules as part of Linux's kernel audit system[185]. Also, there are great examples for “audit.rules” in GitHub (one example is https://github.com/Neo23x0/auditd/blob/master/audit.rules).



[179] https://elixir.bootlin.com/linux/v6.4-rc4/source/kernel/audit.c#L1700
[180] https://elixir.bootlin.com/linux/v6.4-rc4/source/kernel/audit.c#L828
[181] https://medium.com/@boutnaru/linux-security-lsm-linux-security-modules-907bbcf8c8b4
[182] https://elixir.bootlin.com/linux/v6.4-rc4/source/kernel/audit.c#L11
[183] https://link.springer.com/chapter/10.1007/978-3-031-17143-7_30
[184] https://manpages.debian.org/unstable/auditd/audit.rules.7.en.html
[185] https://linux.die.net/man/8/auditctl

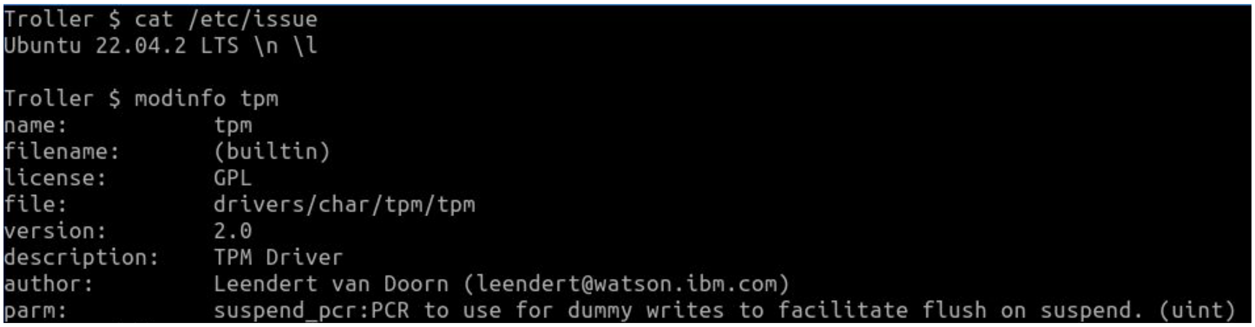
tpm_dev_ wq

“tpm_dev_wq” is a kernel thread base on a workqueue[186]. It belongs a device file system interface for “Trusted Platform Module” aka TPM[187].

Overall, TPM is an international standard for secure cryptoprocessors. Those are microprocessors which are used for a variety of security applications such as secure boot, random number generating and crypto key storage[188].

Moreover, a work is queued for “tpm_dev_wq” as part of the function “tpm_common_write”[189]. In case we are working in non-blocking mode an async job for sending the command is scheduled[190].

Lastly, “tpm-dev-common.c” is compiled as part of the kernel TPM device drivers as shown in the Makefile[191]. The information about the TPM module is shown in the screenshot below. I am using Ubuntu “22.04.2”, in which the TPM module is compiled directly into the kernel itself.



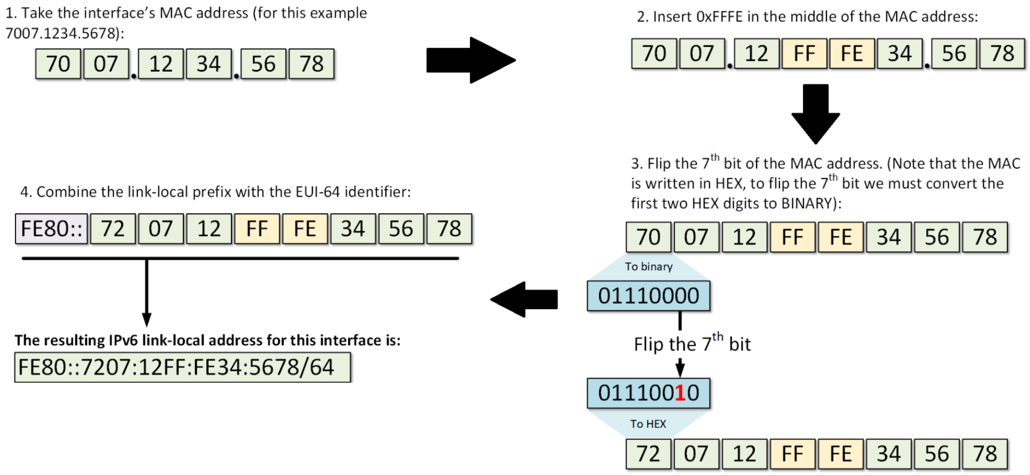
```
Troller $ cat /etc/issue
Ubuntu 22.04.2 LTS \n \l

Troller $ modinfo tpm
name:           tpm
filename:       (builtin)
license:        GPL
file:           drivers/char/tpm/tpm
version:        2.0
description:    TPM Driver
author:         Leendert van Doorn (leendert@watson.ibm.com)
parm:           suspend_pcr:PCR to use for dummy writes to facilitate flush on suspend. (uint)
```

[186] https://elixir.bootlin.com/linux/v6.3-rc7/source/drivers/char/tpm/tpm-dev-common.c#L273
[187] https://elixir.bootlin.com/linux/v6.3-rc7/source/drivers/char/tpm/tpm-dev-common.c#L13
[188] https://wiki.archlinux.org/title/Trusted_Platform_Module
[189] https://elixir.bootlin.com/linux/v6.3-rc7/source/drivers/char/tpm/tpm-dev-common.c#L209
[190] https://elixir.bootlin.com/linux/v6.3-rc7/source/drivers/char/tpm/tpm-dev-common.c#L202
[191] https://elixir.bootlin.com/linux/v6.3-rc7/source/drivers/char/tpm/Makefile

ipv6_addrconf

“ipv6_addrconf” is a kernel thread which is based on a workqueue[192]. This code is part of the Linux INET6 implementation and is responsible for the IPv6 Address auto configuration[193]. Overall, each IPv6 entity in the network needs a globally unique address for communicating outside of the local segment. In order to get such an address there are a few options: manual assignment of an address, DHCPv6 (Dynamic Host Configuration Protocol version 6) and SLAAC (Stateless Address Autoconfiguration). When talking about stateless and stateful it means if there is a server/device that keep tracks of a state for each address assignment[194].

Moreover, the stateless address autoconfiguration has the following phases. The node configures itself with a link-local address. The most known way for doing that is using the link-local prefix “FE80::/64” and combining that with the EUI-64 identifier generated from the MAC address - as shown in the diagram below.



1. Take the interface’s MAC address (for this example 7007.1234.5678):
70 07 . 12 34 . 56 78
2. Insert 0xFFFE in the middle of the MAC address:
70 07 . 12 FF FE 34 . 56 78
3. Flip the 7th bit of the MAC address. (Note that the MAC is written in HEX, to flip the 7th bit we must convert the first two HEX digits to BINARY):
70 07 12 FF FE 34 56 78
To binary
01110000
Flip the 7th bit
01110010
To HEX
72 07 12 FF FE 34 56 78
4. Combine the link-local prefix with the EUI-64 identifier:
FE80:: 72 07 12 FF FE 34 56 78
The resulting IPv6 link-local address for this interface is:
FE80::7207:12FF:FE34:5678/64

The flow above It is done by the function “addrconf_addr_gen”[195]. We can see there the link-local prefix[196] and the call for generating the EUI-64 identifier by the function “ipv6_generate_eui64”[197]. After that, the node performs DAD (Duplicate Address Detection) in order to ensure that the address is unique in the local segment. It is done using NDP (Neighbor Discovery Protocol), which defines 5 new packets types to ICMPv6 that allows to provide different functionality like DAD and others like parameter discovery, next hop determination and

[192] https://elixir.bootlin.com/linux/v6.2.11/source/net/ipv6/addrconf.c#L7292
[193] https://elixir.bootlin.com/linux/v6.2.11/source/net/ipv6/addrconf.c#L3
[194] https://www.networkacademy.io/ccna/ipv6/stateless-address-autoconfiguration-slaac
[195] https://elixir.bootlin.com/linux/v6.2.11/source/net/ipv6/addrconf.c#L3314
[196] https://elixir.bootlin.com/linux/v6.2.11/source/net/ipv6/addrconf.c#L3326
[197] https://elixir.bootlin.com/linux/v6.2.11/source/net/ipv6/addrconf.c#L3345

more[198]. If there are no issues with the link-local address it is assigned to the specific device. The DAD operation is performed by the function “addrconf_dad_work”[199].

Lastly, there is also a similar flow for configuring a global unicast address. The difference is that there is also a need for sending a “Router Solicitation” message for getting the global prefix of the segment, I will leave the details of that for a future writeup.

[198] https://datatracker.ietf.org/doc/html/rfc4862

[199] https://elixir.bootlin.com/linux/v6.2.11/source/net/ipv6/addrconf.c#L4058

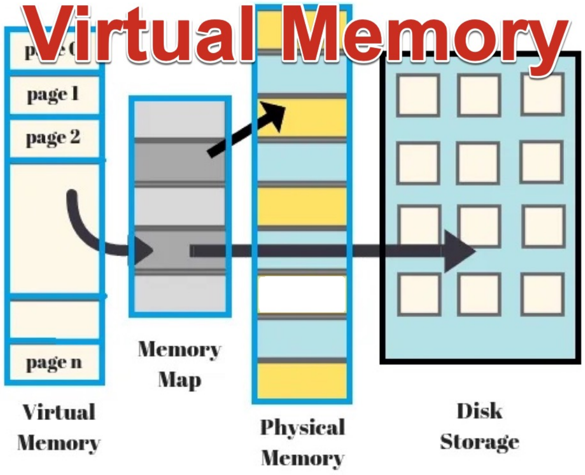
mm_percpu_wq

“mm_percpu_wq” is a kernel thread based on a workqueue which is created in the “init_mm_internals” function[200]. It is part of the the statistics management regarding virtual memory[201]. An overview diagram of virtual memory is shown below[202].

Overall, “mm_percpu_wq” is the worker thread which updates different counters about the virtual memory of a Linux system. It is also called the “vmstat worker”[203]. “vmstat” stands for “Virtual Memory Statistics” which includes information such as: number of free pages, number of mapped pages, number or dirty pages, amount of memory allocated to kernel stacks and more (there are more than 150 different counters).

The statistics can be read from the file “/proc/vmstat”[204]. This proc entry is created with others (“buddyinfo, “pagetypeinfo” and “zoneinfo”) in the same file in which “mm_percpu_mm” is allocated[205]. We can see the list of the metric counters in the source code[206].

As it names suggested the kernel thread is responsible for accumulating the vm events among all CPUs[207]. It is done by going over all the “online” CPUs[208]. Lastly, we can use different cli tools to review the different statistic counters. One of those tools is “vmstat”[209].



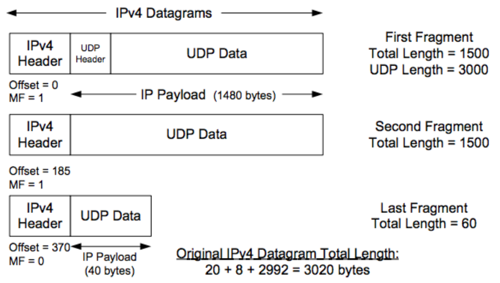
[200] https://elixir.bootlin.com/linux/v6.4-rc5/source/mm/vmstat.c#L2100
[201] https://elixir.bootlin.com/linux/v6.4-rc5/source/mm/vmstat.c#L5
[202] https://iboysoft.com/wiki/virtual-memory.html
[203] https://elixir.bootlin.com/linux/v6.4-rc5/source/mm/vmstat.c#L2021
[204] https://man7.org/linux/man-pages/man5/proc.5.html
[205] https://elixir.bootlin.com/linux/v6.4-rc5/source/mm/vmstat.c#L2123
[206] https://elixir.bootlin.com/linux/v6.4-rc5/source/mm/vmstat.c#L1168
[207] https://elixir.bootlin.com/linux/v6.4-rc5/source/mm/vmstat.c#L126
[208] https://elixir.bootlin.com/linux/v6.4-rc5/source/mm/vmstat.c#L117
[209] https://linux.die.net/man/8/vmstat

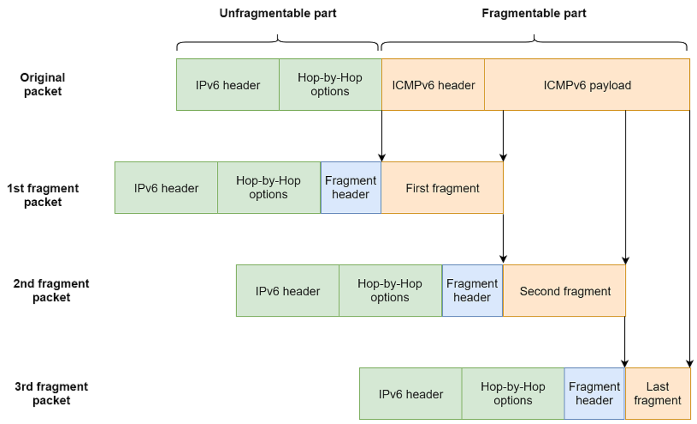
inet_frag_wq

The kernel thread "inet_frag_wq" is created using a workqueue[210], we could have guessed it based on a workqueue do to the "wq" suffix. It is used for fragment management of IP packets. Thus the goal of "inet_frag_wq" is to reassemble fragmented IPv4/IPv6 packets[211].

Overall, the goal of IP fragmentation is to split packets into smaller chunks in order to allow them to meet the MTU (Maximum Transmission Unit) requirement of a specific network. There is an implementation difference between IP fragmentation in IPv4 and IPv6. On IPv4 the information needed for fragmentation is part of the IPv4 header which in IPv6 there is a specific "Fragmentation Header"[212]. An illustration of the flow is shown in the diagram below both for IPv4[213] and IPv6[214].

Thus, "inet_frag_wq" is relevant when a fragmented IP packet arrives at a specific system. The OS stores the fragmented packets in a queue and reassembles them before they are passing the data to the upper layers of the network stack. The fragment queue is represented by "struct inet_frag_queue"[215]. Moreover, we can see in the source code the function "ip_frag_reasm" which is responsible for building a new IP datagram from all of its fragments[216].





[210] https://elixir.bootlin.com/linux/v6.2-rc1/source/net/ipv4/inet_fragment.c#L211
[211] https://elixir.bootlin.com/linux/v6.2-rc1/source/net/ipv4/inet_fragment.c#L6
[212] https://www.geeksforgeeks.org/ipv6-fragmentation-header/
[213] https://notes.shichao.io/tcpv1/ch10/
[214] https://blog.quarkslab.com/analysis-of-a-windows-ipv6-fragmentation-vulnerability-cve-2021-24086.html
[215] https://elixir.bootlin.com/linux/v6.2-rc1/source/include/net/inet_frag.h#L66
[216] https://elixir.bootlin.com/linux/v6.2-rc1/source/net/ipv4/ip_fragment.c#L411

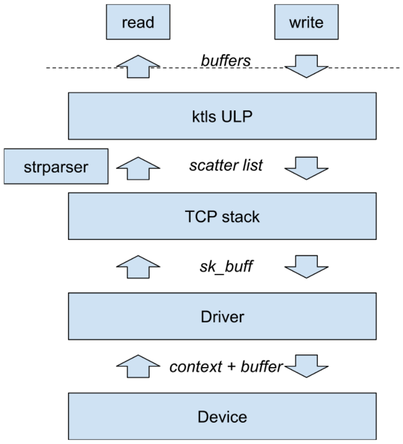
kstrp (Stream Parser)

“kstrp” is based on a single threaded workqueue[217]. Form the source code documentation we can see that “strparser” means “Stream Parser”[218]. A stream parser is a utility that gets data streams and parsers the application layer protocol over those streams. A stream parser can work in one of two modes: general mode or receive callback mode.

In general mode, a sequence of socket buffers (skbs) are given to the stream parser from an outside source. Messages are parsed and delivered as the sequence is processed. This mode allows a stream parser to be applied to any arbitrary stream of data. In receive callback mode, the stream parser is called from the data_ready callback of the TCP socket. Messages are parsed and delivered as they are received on the socket[219].

Thus, we can say that we can parse application layer protocol messages in TCP. It is basically a generalization of KCM (Kernel Connection Multiplexor)[220].
KMC provides a message based interface over TCP for generic application protocols. With the use of KMC applications can send/receive application messages efficiently over TCP[221].

Lastly, “strparser” allows intercepting packets on TCP connections. This is done at the kernel level which provides the ability to perform custom processing. The processing can be done using the BPF/Kernel module[222]. One example for that is the implementation of KTLS[223] (a Linux TLS/DTLS kernel module). An illustration of the flow is shown below[224].



[217] https://elixir.bootlin.com/linux/v6.1.1/source/net/strparser/strparser.c#L539
[218] https://elixir.bootlin.com/linux/v6.1.1/source/net/strparser/strparser.c#L3
[219] https://www.kernel.org/doc/html/v5.10/networking/strparser.html
[220] https://lwn.net/Articles/695982/
[221] https://www.kernel.org/doc/html/latest/networking/kcm.html
[222] https://zhuanlan.zhihu.com/p/543663512
[223] https://github.com/ktls/af_ktls
[224] https://docs.kernel.org/networking/tls-offload.html

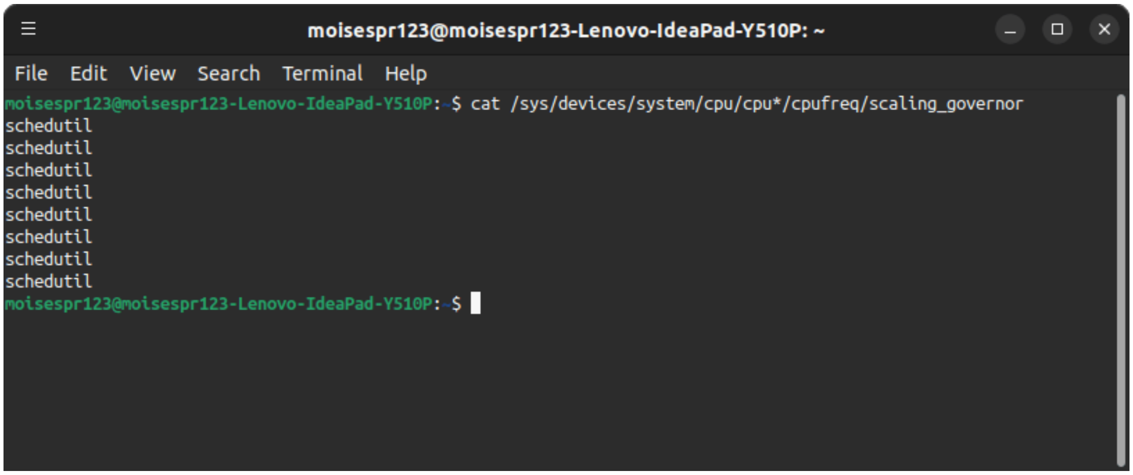
devfreq_wq

“devfreq_wq” is a kernel thread which is based on a freezable workqueue[225]. It is part of the Generic Dynamic Voltage and Frequency Scaling (DVFS) Framework for Non-CPU devices[226].

Overall, DVFS enables Linux to scale the CPU frequency in order to minimize the power usage. It is mostly done when the full performance of the CPU is not needed. By using DVFS the system can set min/max CPU frequency. There is also the ability to set a “scaling governor” which monitors the performance requirements and decides what CPU frequency to use each time[227].

Moreover, based on the Linux documentation there are 6 governors: “Performance”, “Powersave”, “Userspace”, “Ondemand”, “Conservative” and “Schedutil”[228]. We can also develop our own governor as a kernel module, we just need to register it using the function “cpufreq_register_governor”[229].

Lastly, we can use the sysfs filesystem to configure/read information regarding “cpufeq”. An example of a file path for the first cpu is “/sys/devices/system/cpu/cpu0/cpufreq/” (if sysfs is mounted at “/sys”). It might contain the information like (but not limited to): current frequency of the CPU, the time it takes the CPU to switch frequencies (in nanosecs) and more[230]. An example of reading the current configure governor is shown below[231].


moisespr123@moisespr123-Lenovo-IdeaPad-Y510P: ~
File Edit View Search Terminal Help

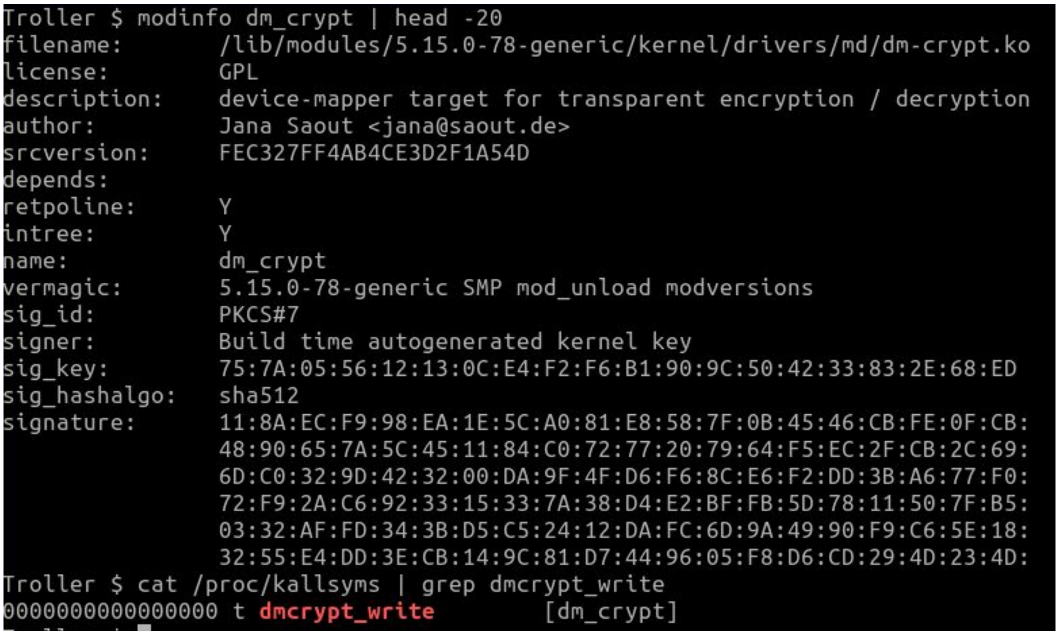
```
moisespr123@moisespr123-Lenovo-IdeaPad-Y510P:~$ cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
schedutil
schedutil
schedutil
schedutil
schedutil
schedutil
schedutil
schedutil
moisespr123@moisespr123-Lenovo-IdeaPad-Y510P:~$
```

[225] https://elixir.bootlin.com/linux/v6.2.5/source/drivers/devfreq/devfreq.c#L1997
[226] https://elixir.bootlin.com/linux/v6.2.5/source/drivers/devfreq/devfreq.c#L3
[227] https://wiki.somlabs.com/index.php/How_to_scale_CPU_frequency_with_DVFS_framework
[228] https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt
[229] https://elixir.bootlin.com/linux/v6.5-rc2/source/drivers/cpufreq/cpufreq.c#L2443
[230] https://www.kernel.org/doc/Documentation/cpu-freq/user-guide.txt
[231] https://moisescardona.me/changing-the-cpu-governor-to-performance-in-linux/

dmcrypt_write

“dmcrypt_write” is a kernel thread which is created using the “kthread_run” function[232]. The name of the kernel thread is in the pattern of "dmcrypt_write/%s", where the added string represents the device name.Overall, “dm-crypt” is a device-mapper target[233] supported from kernel version 2.6.4[234]. It is responsible for transparent (aka real-time/on-the-fly encryption) block device encryption while using the kernel crypto API[235] .

This means the data is encrypted/decrypted while it is read/written. To enable the “dm-crypt” support we need to enable “CONFIG_DM_CRYPT” in the compilation config of the kernel[236]. Moreover, the function that is executed as part of the kernel thread is “dmcrypt_write” function[237]. This function is part of the kernel module “dm_crypt” - as shown in the screenshot below. We can use “modinfo dm_crypt” for more information, also shown in the screenshot below.



```
Troller $ modinfo dm_crypt | head -20
filename:       /lib/modules/5.15.0-78-generic/kernel/drivers/md/dm-crypt.ko
license:        GPL
description:    device-mapper target for transparent encryption / decryption
author:         Jana Saout <jana@saout.de>
srcversion:     FEC327FF4AB4CE3D2F1A54D
depends:
retpoline:      Y
intree:         Y
name:           dm_crypt
vermagic:       5.15.0-78-generic SMP mod_unload modversions
sig_id:         PKCS#7
signer:         Build time autogenerated kernel key
sig_key:        75:7A:05:56:12:13:0C:E4:F2:F6:B1:90:9C:50:42:33:83:2E:68:ED
sig_hashalgo:   sha512
signature:      11:8A:EC:F9:98:EA:1E:5C:A0:81:E8:58:7F:0B:45:46:CB:FE:0F:CB:
                48:90:65:7A:5C:45:11:84:C0:72:77:20:79:64:F5:EC:2F:CB:2C:69:
                6D:C0:32:9D:42:32:00:DA:9F:4F:D6:F6:8C:E6:F2:DD:3B:A6:77:F0:
                72:F9:2A:C6:92:33:15:33:7A:38:D4:E2:BF:FB:5D:78:11:50:7F:B5:
                03:32:AF:FD:34:3B:D5:C5:24:12:DA:FC:6D:9A:49:90:F9:C6:5E:18:
                32:55:E4:DD:3E:CB:14:9C:81:D7:44:96:05:F8:D6:CD:29:4D:23:4D:
Troller $ cat /proc/kallsyms | grep dmcrypt_write
0000000000000000 t dmcrypt_write        [dm_crypt]
```

[232] https://elixir.bootlin.com/linux/v6.5-rc3/source/drivers/md/dm-crypt.c#L3388
[233] https://elixir.bootlin.com/linux/v6.5-rc3/source/drivers/md/dm-crypt.c#L3689
[234] https://elixir.bootlin.com/linux/v2.6.4/source/drivers/md/dm-crypt.c
[235] https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMCrypt
[236] https://elixir.bootlin.com/linux/v6.5-rc3/source/drivers/md/Makefile#L59
[237] https://elixir.bootlin.com/linux/v6.5-rc3/source/drivers/md/dm-crypt.c#L1922

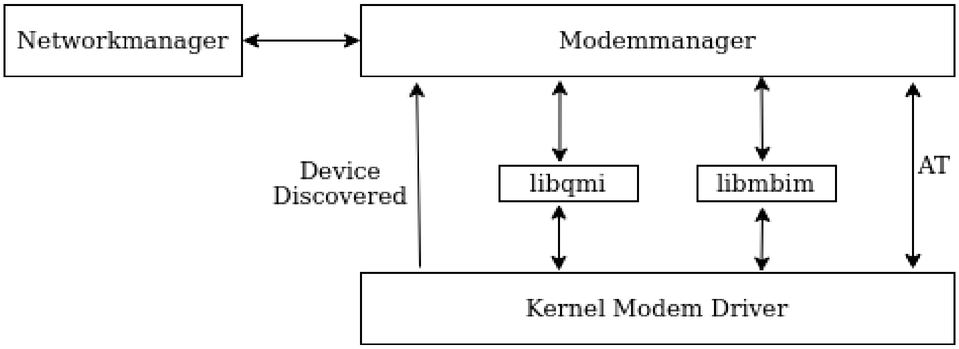
ModemManager (Modem Management Daemon)

"ModemManager" is an ELF binary located by default at "/usr/sbin/ModemManager" which is used to provide a unified high level API for communication with mobile broadband modems[238]. Alos, it is started by PID 1 (init/systemd) with the permission of the root user.

Overall, it is a DBus-powered[239] Linux daemon which acts as a standard RIL (Radio Interface Layer). "ModemManager" can be used by different connection managers (think about "NetworkManager" for example). Moreover, if we want to control and manage "ModemManager" we can use the CLI tool "mmcli". By using it we can list all available modems, connect to a modem, get/set properties of the modem and more[240].

Thus, we can summarize "ModemManager" as a system daemon that controls WWAN (2G/3G/4G/5G) devices and connections. It is the default mobile broadband management system in most Linux distributions (like Ubuntu, Debian, Fedora and Arch). By the way, it is also used by routers running OpenWRT[241].

It is important to understand that "ModemManager" leverages "libqmi"[242] and "libmbim"[243] to communicate over QMI (Qualcomm MSM Interface) and MBIM (Mobile Interface Broadband Model) for setting connection to to the cellular network[244]. It does not matter if the modem is builtin, USB connected or bluetooth-paired. A diagram of the architecture is shown below. Lastly, if we want to go over the source code on "ModemManager" or contribute we can use its repo[245]. I also suggest going over the documentation site of "ModemManager" and the relevant libraries: libmbim, libqmi and libqrtr-glib[246].



[238] https://manpages.ubuntu.com/manpages/trusty/man8/ModemManager.8.html
[239] https://www.freedesktop.org/software/ModemManager/api/latest/
[240] https://manpages.ubuntu.com/manpages/trusty/man8/mmcli.8.html
[241] https://modemmanager.org/
[242] https://github.com/linux-mobile-broadband/libqmi
[243] https://github.com/linux-mobile-broadband/libmbim
[244] https://developer.toradex.com/software/connectivity/modem-support/
[245] https://gitlab.freedesktop.org/mobile-broadband/ModemManager
[246] https://modemmanager.org/docs/

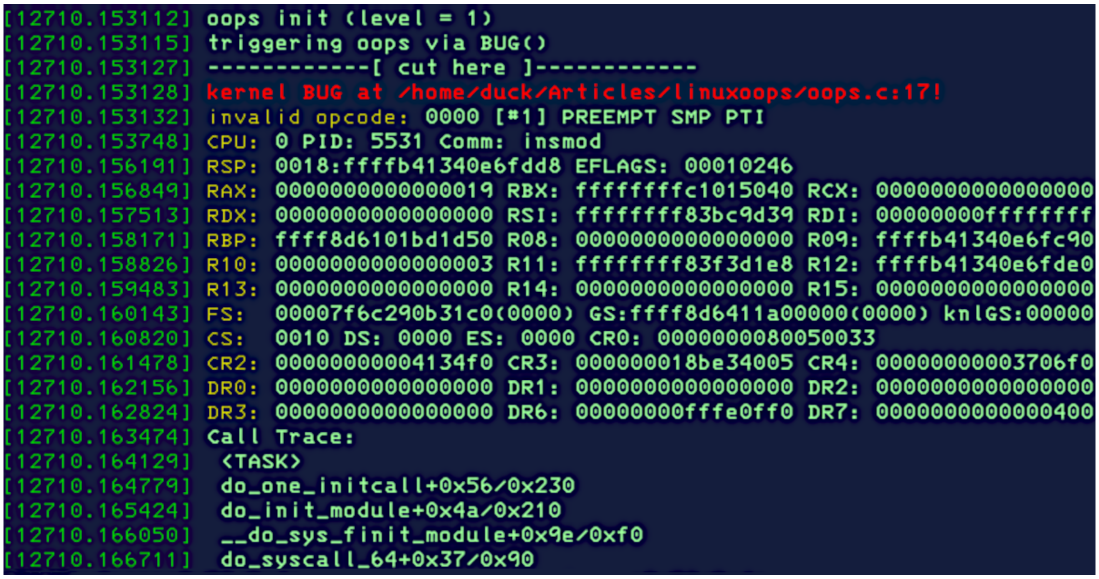
kerneloops

“kerneloops” is an ELF binary located at “”. It is used to collect kernel crash information (as part of a kernel oops) and submit them to kerneloops.org[247]. An example of such oops is shown in the screenshot below[248]. By the way, they are also known as “soft panic”[249].

Overall, a kernel oops is a serious but non-fatal error in the Linux kernel. It is a way for the kernel to signal that it has found a problem that could potentially cause the system to crash. However, the kernel will continue to run after an oops, although it may be unstable and can lead to a kernel panic. This helps in debugging the error in order to find a solution for the problem[250].

Moreover, if we want to debug the kernel with gdb it is suggested to compile it with “CONFIG_DEBUG_INFO” enabled, which causes the kernel to be built with full debugging information[251]. Also, I recommend also enabling “CONFIG_FRAME_POINTER”, which gives very useful debugging information in case of kernel bugs - precise oopses/stacktraces/warnings[252].

Lastly, there is also a setting called “oops_limit” which states after what number of oops should cause a panic. The default value by the way is 10000[253].



```
[12710.153112] oops init (level = 1)
[12710.153115] triggering oops via BUG()
[12710.153127] ------------[ cut here ]------------
[12710.153128] kernel BUG at /home/duck/Articles/linuxoops/oops.c:17!
[12710.153132] invalid opcode: 0000 [#1] PREEMPT SMP PTI
[12710.153748] CPU: 0 PID: 5531 Comm: insmod
[12710.156191] RSP: 0018:ffffb41340e6fdd8 EFLAGS: 00010246
[12710.156849] RAX: 0000000000000019 RBX: ffffffffc1015040 RCX: 0000000000000000
[12710.157513] RDX: 0000000000000000 RSI: ffffffff83bc9d39 RDI: 00000000ffffffff
[12710.158171] RBP: ffff8d6101bd1d50 R08: 0000000000000000 R09: ffffb41340e6fc90
[12710.158826] R10: 0000000000000003 R11: ffffffff83f3d1e8 R12: ffffb41340e6fde0
[12710.159483] R13: 0000000000000000 R14: 0000000000000000 R15: 0000000000000000
[12710.160143] FS:  00007f6c290b31c0(0000) GS:ffff8d6411a00000(0000) knlGS:00000
[12710.160820] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[12710.161478] CR2: 00000000004134f0 CR3: 000000018be34005 CR4: 00000000003706f0
[12710.162156] DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
[12710.162824] DR3: 0000000000000000 DR6: 00000000fffe0ff0 DR7: 0000000000000400
[12710.163474] Call Trace:
[12710.164129]  <TASK>
[12710.164779]  do_one_initcall+0x56/0x230
[12710.165424]  do_init_module+0x4a/0x210
[12710.166050]  __do_sys_finit_module+0x9e/0xf0
[12710.166711]  do_syscall_64+0x37/0x90
```

[247] https://linux.die.net/man/8/kerneloops
[248] https://nakedsecurity.sophos.com/2023/03/13/linux-gets-double-quick-double-update-to-fix-kernel-oops/
[249] https://www.opensourceforu.com/2011/01/understanding-a-kernel-oops/
[250] https://en.wikipedia.org/wiki/Linux_kernel_oops
[251] https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch04.html
[252] https://cateee.net/lkddb/web-lkddb/FRAME_POINTER.html
[253] https://docs.kernel.org/admin-guide/sysctl/kernel.html#oops-limit