Advanced Operating Systems and Virtualization - A.A. 2019/2020

Final Project Report

Student Full Name: Alessio Papi

Student ID: 1761063

Table of Contents

Introduction **3**

Building process 4

Kernel-Level Data Structures **5**

struct group_dev_t 5

struct delayed_msg_t 6

struct msg_t 7

Kernel-Level Subsystem Implementation **8**

Lookaside Caches 8

Garbage Collection 8

Global Variables 9

Module Initialization: initfn 9

Group Initialization: gdev_init 9

Module Finalization: exitfn 9

File Operations: group_fops 10

Open: group_open 10

Write: group_write 10

Read: group_read 11

Ioctl: group_ioctl 11

INSTALL_GROUP 12

SLEEP_ON_BARRIER 12

AWAKE_BARRIER 12

SET_SEND_DELAY 13

REVOKE_DELAYED_MESSAGES 13

Flush: group_flush 13

Sysfs: max_message_size, max_storage_size 14

Udev rules: 83-groups.rules 15

User Space Library 16

struct lgroup_t 16

lgroup_init, lgroup_destroy 16

API 16

Control Operations 17

Messaging Operations 17

Barrier Operations 18

Testcase 19

unit_tests.c 19

test_install_group.c 19

test_rw_fifo.c 19

test_sysfs.c 20

test_barrier.c 20

test_delay.c 20

test_revoke.c 21

test_flush.c 21

test_stress.c 21

test_max_install.c 22

test.c 22

Benchmark 23

rw_tps.c 23

Introduction

Provide here a general picture of what the subsystem does and how it does. Don't go too much down in technical details.

The subsystem allows threads (from any process) to (1) exchange messages and (2) synchronize with each other, within the context of groups.

The subsystem, implemented as a Linux LKM (Loadable Kernel Module), offers the following services:

- **Installing a new group**
  Before starting any communication or synchronization operation, client threads have to register within some group. The group involved in the registration is distinguished via a **group id**, which univocally identifies the group within the subsystem.
  For this purpose, the subsystem exposes to userspace a header file (src/include/kernel/groups.h) containing the definition of a descriptor for a group: **struct group_t**.
  It consists of two fields: (I) **id** and (II) **devname**. The initialized group_t shall be encapsulated in the installation request to the kernel, which will correctly set the **devname** field.
  With the *group_t->devname* in hand, client threads can interact with the group's device file, whose symbolic link is located on disk at *"/dev/synch/$(group_t->devname)"* by the udev daemon.
  Interactions on the group's device file happen via standard POSIX I/O primitives (open, close, read, write and ioctl), which eventually boil down to the file operations implemented within the LKM.

Once a group is installed and opened, a client thread can:

- **Send a message**
  At the abstract level, sending a message means publishing a unit of information in a group-shared queue of messages. The publication semantics vary depending on the group's **operating mode**, which can be either **immediate** or **delayed**.
  A group is installed with *immediate operating mode* on, according to which messages are immediately and synchronously published on the group's communication channel. The operating mode can be changed via a dedicated ioctl command: SET_SEND_DELAY.
  While sending a message, some restrictions apply. Each group has reconfigurable size limits, both locally (on the size of the single messages) and globally (at the granularity of the whole group-shared queue).

- **Deliver a message**
  It allows dequeuing a message from the group-shared queue, with **FIFO delivery** semantics.
  Being POSIX compliant, the invoking thread has to specify the amount of bytes to read. If such an amount is lesser than the size of the message to be dequeued, the message is truncated.
  Also, the delivery happens with an **exactly once within group** semantics: once a member of a group delivers a (possibly truncated) message, no other member will ever be able to deliver that message again.

- **Sleep-on** / **Awake barrier**
  The first command allows a thread to be descheduled and never rescheduled until explicitly woken up by some other thread in the group (via the second command). However, the sleep of a sleeping client thread can be **interrupted** (EG.: by means of a SIGINT signal).
  At the high level, the barrier service is implemented by relying on a group-shared condition variable, which is initially unset. As long as the variable is unset, incoming sleepers will block on

the group’s barrier… only the AWAKE_BARRIER ioctl command will set the condition variable, awaking them all. The last awaken sleeper is the one in charge of resetting the condition variable.

- **Set send delay**
  This command allows client threads to modify the write operating mode.
  The new **delay** (expressed in milliseconds) shall be passed as parameter: when it is zero, the immediate operating mode is restored for future write operations; otherwise, the group enters the delayed operating mode, according to which messages are **asynchronously** stored in the shared queue, in a **delayed** fashion.

- **Revoke delayed messages**
  This command allows retracting all delayed messages whose delay has not elapsed yet.

Concurrent I/O sessions on every group are supported. Indeed, all modifications to shared data structures happen within the context of properly designed critical sections.

Building process

The project is made up of several software artifacts: kernel module, userspace library, unit tests, general purpose userspace testing application, benchmark application and bash scripts.

Artifacts shall be compiled in a precise order, due to possible linking dependencies among them.

Hence, the building process was implemented using the Makefile building tool.
To build the entire project, you simply have to download the repository in a folder and run **make build**. All compiled artifacts will appear, organized, into the **out** folder, within the current working directory. To *undo* the building step, simply run **make clean**.

The **out** folder can be finally moved to the testing environment to play around with compiled binaries and scripts (gathered under the out/bin and out/scripts directory).
It’s important not to delete nor moving files from the output directory, because of the dependencies leveraged among the output files via relative paths. For instance:

- The out/script/benchmark.sh script references the out/bin/test/benchmark/rw_tps.c application.
- The out/script/install.sh script references both the out/obj/kernel/groups.ko and out/udev/83-groups.rules files.

Kernel-Level Data Structures

Provide here a description of the most important kernel-level data structures that you have implemented. They will be referenced in the next section.

I defined 3 main data structures within the LKM: group_dev_t, delayed_msg_t, msg_t.

struct group_dev_t

This is the main data structure of the LKM, because it defines the physical representation of a group within the subsystem. It is composed of the following fields:

- unsigned long **size** : accumulator variable keeping track of the size of all messages handled by the group, I.E.: both published and delayed.
- unsigned long **max_msg_size** : message size limits for the group.
  The field: "struct kobj_attribute max_msg_size_attr" represents the associated kobject attribute data structure for exposition within sysfs.
- unsigned long **max_strg_size** : upper bound for the size field.
  The field: "struct kobj_attribute max_strg_size_attr" represents the associated kobject attribute data structure for exposition within sysfs.
- spinlock_t **size_lock** : access to the aforementioned 3 fields (size, max_msg_size and max_strg_size) always happen within the critical section implemented on this lock.

- atomic_t **delay** : current delay for the group. Discriminates its operating mode.

- struct list_head **published_list** : first list_head node of the group-shared queue of published messages. Successive nodes will be contained within struct msg_t objects.
- spinlock_t **published_list_lock** : protects access to published_list.
- struct list_head **delayed_list** : first list_head node of the group-shared queue of delayed messages. Successive nodes will be contained within struct delayed_msg_t objects.
- spinlock_t **delayed_list_lock** : protects access to delayed_list.

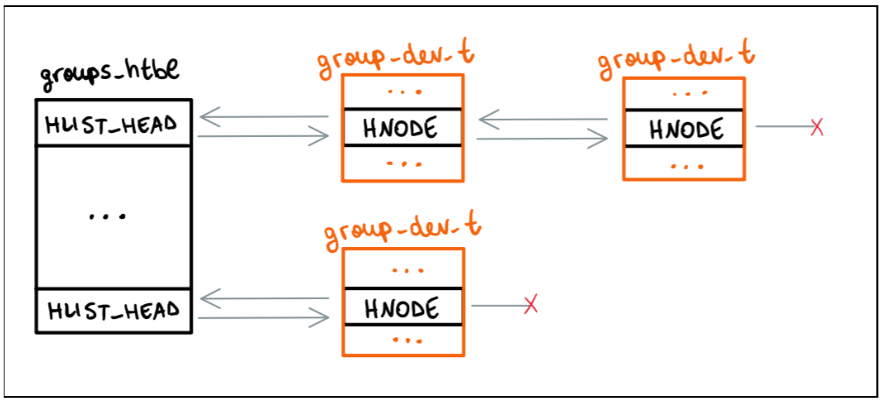
- int **sleepers** : updated amount of client threads sleeping on the group's barrier.
- int **wakeup** : sleeping condition variable.
- spinlock_t **barrier_lock** : access to the aforementioned 2 fields (sleepers, wakeup) always happen within the critical section implemented on this lock.
- wait_queue_head_t **sleeping_wq** : wait queue to optimize sleep/awake barrier operations.

- struct device ***dev** : Linux kernel data structure retaining information about the physical device node associated with the group, and its sysfs representation.

- struct cdev **cdev** : Linux kernel data structure representing a char device.

- struct group_t ***group** : custom-defined data structure holding identification information about the group. It is composed of two character buffer fields:
    - char ***id** : the identifier of the group.
    - char ***devname** : the device filename of an installed group. After a group's installation, threads expect to find the group's device file published at /dev/synch/$(devname). *devname* has the format 'group%d', where %d = serial number of the group within the subsystem.

- struct hlist_node **hnode** : Linux kernel implementation of hashtables. They are used to gather all group_dev_t(s) within a unique data structure: **groups_htbl**, hashed by group id.
  Thanks to groups_htbl, we'll efficiently be able to find out whether, during an installation request, some group is already present within the subsystem, or it has to be installed from scratch.



struct delayed_msg_t

This data structure defines the physical representation of a delayed unit of information:

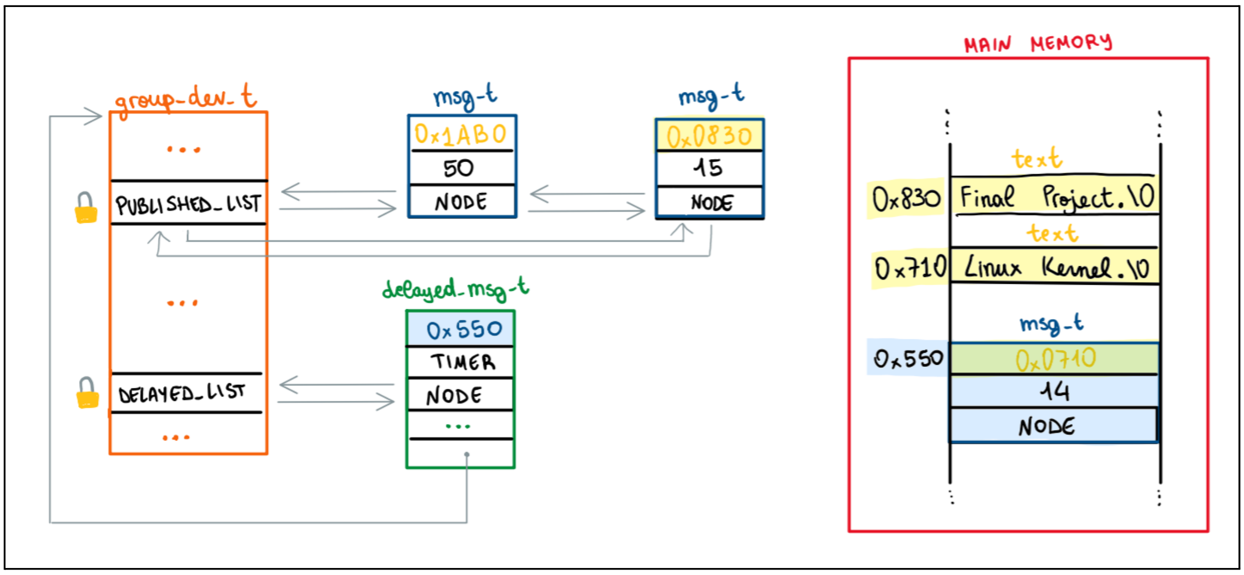
- struct group_dev_t ***gdev** : reference to the struct *group_dev_t* whose *delayed_list* this *delayed_msg_t* belongs to (needed within the *timer_callback* context).
- struct list_head **node** : list_head node linking this delayed_msg_t to its group's delayed list.
- struct msg_t ***msg** : the actual unit of information embedded in this delayed_msg.
- int **is_revoked, is_flushed** : boolean flags used to implement the REVOKE_DELAYED_MESSAGES ioctl and the flush syscall respectively.
- struct timer_list **timer** : Linux kernel timer associated with this delayed_msg.

struct msg_t

This data structure defines the physical representation of a unit of information within the subsystem, I.E.: a message ready to be delivered by the next reader thread.

- char ***text** : pointer to the memory buffer containing the message payload.
- size_t **size** : size of the *text* memory buffer.
- struct list_head **node** : list_head node linking this msg_t to its group's published list.

Below is the global picture of the subsystem's data structures:



Kernel-Level Subsystem Implementation

Describe here the implementation of your subsystem. You can insert code snippets, add as many subsections as you want. Try to be as clear as possible.

My subsystem implementation targets (and was implemented on) the Linux kernel version **5.4.0-70-generic** (my machine runs the Ubuntu 18.04.5 LTS distribution).

The whole LKM was implemented within a single C source file: *src/kernel/groups.c* , explained in the following paragraphs. Also, of relevant importance for this chapter are: (I) the exported kernel header (src/include/kernel/groups.h) and (II) the udev rules file (src/udev/83-groups.rules).

Lookaside Caches

The subsystem maintains three different lookaside caches, each in charge of managing a pool of cached objects (or slabs) targeting each distinct kernel-level data structure: **msg_cache**, **delayed_msg_cache** and **group_dev_cache**.

Lookaside caches were introduced to optimize the, very frequent, memory allocations and deallocations for these data types. Caches are created[destroyed] at module initialization[finalization] with kmem_cache_create[destroy], and used to serve memory operations with kmem_cache_alloc[free].

Garbage Collection

Despite employing lookaside caches, memory allocations and deallocations still remain a criticality when reasoning about performances and responsiveness. On this purpose, deallocations are not directly performed within I/O operations, but deferred to the workqueue: **groups_wq**.

Every time a deallocation is required by some user context (EG.: after a reader has delivered a message from a group), the kernel level data structure is atomically appended to a pool of objects whose deallocation is deferred in time. Those pools of objects are:

- ***next_published_list**, for the garbage collection of msg_t objects by the routine: **published_work_fn(..)**, associated with the work_struct **published_work**.
  The pool is protected by **published_work_lock**, to avoid inconsistencies.
- ***next_delayed_list**, for the garbage collection of delayed_msg_t objects by the routine: **delayed_work_fn(..)**, associated with the work_struct **delayed_work**.
  The pool is protected by **delayed_work_lock**, to avoid inconsistencies.

Consider the example of the reader… before returning to userspace, the read msg_t is atomically appended to next_published_list, and published_work is queued on the groups_wq.

Sooner or later, published_work_fn will be scheduled, and it will:

- Atomically swap the global *list_head* pointers of next_published_list and **active_published_list**.
  As a result, next_published_list is reset and prepared to accept new deallocations deferrals.
- Start iterating over active_published_list, actually releasing memory.

The same happens with delayed_work_fn.

The architectural decision of employing a garbage collection facility originated by having the subsystem allocating message payloads' buffers with **vmalloc**. As opposed to **kfree**, the call to **vfree** is a much slower kernel facility, which can sleep. Workqueues allow to delegate batches of such deallocations to user context workers, which are allowed to sleep, preserving the subsystem's throughput and responsiveness.

Global Variables

- Groups are organized within the hashtable **groups_htbl**, hashed by group id. Modifications to this hashtable are protected by the spinlock **htbl_lock**.
- unsigned long **groups** : keeps track of the amount of currently installed groups. Since the subsystem dynamically allocates a new major starting from baseminor 0, this variable:
    a. Represents the minor number assigned to new groups' character devices.
    b. Can never overpass the amount of pre-reserved minors : unsigned long **range**.
- Finally, **major** and **class**, which are standard global variables for a device driver.

Module Initialization: initfn

The initialization function (**initfn**) is straightforward. It deals with the initialization of everything discussed so far: lookaside caches, garbage collection facilities and global variables.

A major number is dynamically allocated, starting from baseminor 0. In the allocation request, the subsystem pre-reserves a range of minors (by default 2000) representing the availability of new groups. Also, the device class "groups" is created, to whom future group devices will belong to.

Finally, the first subsystem's group is initialized and assigned id "groups" and devname "group0".

Group Initialization: gdev_init

It's the *group_dev_t* initialization routine, and it does the following:

1. Allocates and zeroes a new group_dev_t from the group_dev_cache.
2. Creates a new device node within /dev, via *device_create(..)*, with device name "*group$(groups)*" (IE.: group0, group1…). Such struct device is stored into gdev->dev.
3. Initializes and adds the char device gdev->cdev, passing a reference to the global struct file_operations group_fops.
4. Initializes and exposes the sysfs kobject attributes for max_msg_size and max_strg_size.
5. Initializes all remaining group_dev_t fields: group_t, size, delay, sleepers, wakeup, spinlock(s), sleeping_wq, list_head(s)…
6. Atomically publishes this group_dev_t into groups_htbl, hashed on group->id by means of the xxh64 hash function from include/linux/xxhash.h.

Module Finalization: exitfn

The finalization function (**exitfn**) removes all the traces left by the LKM on the system:

- Flushes and destroys groups_wq for trashing pending objects.

- Iterates every hlist_node within groups_htbl, retrieving a reference to the corresponding group_dev_t with *hlist_entry_safe*, and:
    - All unread payloads are deallocated, and their msg_t(s) are returned to the msg_cache.
    - Sysfs attributes are destroyed, as well as the group's struct device and struct cdev.
    - Eventually, the group_dev_t is returned to the group_dev_cache.
- Finally, lookaside caches are destroyed as well as the device class, and the dynamically allocated chrdev_region is unregistered, together with all pre-registered minors.

File Operations: group_fops

While initializing a new group's character device, a **struct file_operations** shall be specified. It contains the routines that are eventually invoked to interact with the device.

Open: group_open

group_open simply stores a reference to the target group_dev_t (associated with the character device inode) within the **private_data** field of the new struct file. As a result, successive file operations will easily retrieve the target group_dev_t from private_data.

Write: group_write

In the Introduction, we discussed the different write semantics depending on the value of *gdev->delay*. I also presented the *group_dev_t* data structure, whose **published_list** and **delayed_list** fields already hints at how the write mechanism works. In the following, I'll first present my implementation of the immediate operating mode write, then I'll move to the delayed one.

Immediate Operating Mode

1. A check is done to assert that the message is not empty (terminator char only messages are not accepted by the subsystem).
2. A struct msg_t is allocated and initialized, and a *copy_from_user* is invoked to copy *count* bytes from the user buffer to the kernel buffer *msg->text*, allocated with vmalloc.
3. Additional checks are atomically made to make sure that the size limits (w.r.t. both the message and the whole storage) are honoured. Note:
    - These checks are performed **atomically**, while grabbing gdev->size_lock.
    - If no violation happened, gdev->size is incremented by *msg->size* before leaving the c.s. .

At this point, depending on the value of gdev->delay*,* the execution path deviates.
In the immediate mode (delay = *0*), the thread starts spinning on gdev->published_list_lock, and eventually appends the new msg_t to the group-shared queue of published messages (*list_add_tail*).

Of course, both memory allocations and user/kernel data transfers are performed outside any critical sections, for the sake of efficiency. This is a non-written rule characterizing every critical section in the module: critical sections are left as soon as possible, and they never include calls which may sleep.

Delayed Operating Mode

If gdev->delay was set (via some previous SET_SEND_DELAY ioctl command), the delayed execution path is activated. In this case:

4. A struct delayed_msg_t is allocated and initialized, encapsulating the msg_t crafted at step 2.
5. The timer_list delayed_msg->timer **is configured** to trigger (after gdev->delay ms) the timer_callback routine, but the timer is not activated yet.
6. The thread starts spinning on gdev->delayed_list_lock, and eventually appends the new delayed_msg_t to the group-shared queue of delayed messages (*list_add_tail*).
7. Finally, the timer is activated.

Timer Callback

The timer_callback routine is extremely simple:

1. A reference to the delayed_msg containing the expired timer is retrieved with **from_timer** (a *container_of* under the hood).
2. Thanks to the delayed_msg->gdev reference, we atomically enqueue the delayed_msg->msg into gdev->published_list (as an immediate write would do).
3. At this point, the delayed_msg_t is atomically unlinked from gdev->delayed_list and deallocated (this step is skipped if the delayed message is involved in a flush operation).

Note: importantly enough, the two locks (gdev->published_list_lock and gdev->delayed_list_lock) shall always be locked with the *spin_lock_bh* primitive, since they are grabbed from the timer_callback too. Indeed, since a timer callback is executed in the context of a software interrupt (softirq), and it shares data with user context, it could happen that a user thread is interrupted by the timer callback right in the middle of the critical section. *spin_lock_bh* disables softirqs on the running CPU, then grabs the lock.

Cost Analysis

Thanks to the use of doubly linked lists, the write operation (independently of the operating mode) has **constant computational cost** w.r.t. gdev->published_list and gdev->delayed_list cardinalities.

Read: group_read

Once analyzed the write logic, read is straightforward:

1. The thread spins on gdev->published_list_lock, to enter the “published critical section”.
2. If the message queue is empty, leave the c.s. and return EOF. Otherwise, get the first message from the list (*list_first_entry*) and unlink its node from the queue (*list_del*). Leave the c.s. .
3. Copy msg->text into the user buffer (*copy_to_user*).
   - If some error occurs, recover (I) re-entering the critical section and (II) pushing the message at *the beginning* of the queue. It will be the next message to be delivered.
4. Atomically decrement gdev->size by msg->size.
5. Defer memory deallocations by (I) atomically appending the msg_t to next_published_list and (II) queue published_work on groups_wq (if it was already queued, nothing happens).

Cost Analysis

Just like the write, the computational cost of a read is **constant** w.r.t. gdev->published_list cardinality.

Ioctl: group_ioctl

The unlocked ioctl file operation allows the execution of device-specific I/O operations. Its implementation is based on a switch construct, built on the command parameter. Also, it accepts an additional argument, which is decoded within the context of the dispatched command.

INSTALL_GROUP

This ioctl command highly resembles what is done by initfn for the initialization of group0. The main difference lies in the fact that the struct group_t reference for the new group is received as input by the invoking thread. Therefore, the target group **could be already present** within the subsystem.

1. The struct group_t is retrieved with copy_from_user from the input argument.
2. The group->id is hashed and used as search key within groups_htbl, to **atomically** check whether the associated group_dev_t is already installed within the subsystem.
   If that is the case, group_ioctl returns **0**.
   - Note: the search is performed in a critical section (IE.: while grabbing the htbl_lock), to avoid inconsistencies in case a concurrent installation is modifying the hashtable.
3. If no group_dev_t is found, group_ioctl will return value **1**. The group must be installed:
   a. The **install_lock** is taken, and the "install critical section" is entered: only one thread at a time is allowed to modify the **groups** variable and to initialize a new group_dev_t.
   b. Once in the c.s., "groups" is incremented and gdev_init is called. This time, the address of a local *group_dev_t placeholder is passed as parameter to gdev_init, that will populate it with the memory address of the initialized group_dev_t.
4. Finally, *gdev->group->devname* is copied into the corresponding field of the userspace struct group_t (ioctl input parameter).

Corner case: it could happen that two concurrent threads are trying to install the same (fresh new) group. Thus, once grabbed the install_lock, a second hashtable search is performed, to avoid installing the group twice. This second search is not atomic, because (while grabbing the install_lock) we are sure that no other thread can be modifying the hashtable in the meantime.

Cost Analysis

While assuming that the xxh64 hash function is capable of uniformly distributing group id hashes among the buckets… the computational cost of INSTALL_GROUP is $O(\frac{groups}{\#buckets})$ on average.

SLEEP_ON_BARRIER

The sleep/awake implementation logic is based on a set of group-shared **barrier variables**:

1. As soon as a sleep is invoked, the thread spins on gdev->barrier_lock. Indeed, the barrier variables (sleepers, wakeup) are always used within the barrier critical section.
2. Once in the c.s., the sleeper (I) increments gdev->sleepers and (II) checks the gdev->wakeup:
   - If it is **unset**, the thread is descheduled until the next AWAKE_BARRIER will be invoked. This behavior is achieved via *wait_event_interruptible*, which sets the sleeper's PID state as TASK_INTERRUPTIBLE until either a signal is received or gdev->wakeup is set.[1]
   - If [When] gdev->wakeup is **set**, the thread won't enter [will exit] the sleeping loop, it will decrement gdev->sleepers and possibly (if it was the last sleeper on the barrier) it resets gdev->wakeup. The critical section is left.

AWAKE_BARRIER

In the light of the implementation of SLEEP_ON_BARRIER ioctl command, AWAKE_BARRIER is straightforward. As soon as the ioctl switch case is entered, the thread spins on the barrier_lock.

[1] The use of sleeping_wq was adopted to make the implementation more CPU-friendly than a simple busy-waiting loop. When the next writer invokes *wake_up_interruptible*, the sleepers' PID is restored to TASK_RUNNING.

Now, the only criticality is in the **return value**. Indeed, depending on the actual scenario, a different execution path is taken:

- If there's no other thread sleeping on the group's barrier, gdev->wakeup is not set, and the return value is **2**.
- If gdev->wakeup was already set by a former awaker (and the *last sleeper* was not scheduled yet), the return value is **1**.
- Otherwise, gdev->wakeup is set and gdev->sleeping_wq is woken up (*wake_up_interruptible*). Return value is **0**.

SET_SEND_DELAY

This silly ioctl command simply copies from userspace the new value of delay (in milliseconds) and, if no error occurs, atomically stores it into gdev->delay (*atomic_set*).

REVOKE_DELAYED_MESSAGES

Keeping track of delayed_msg_t(s) within each gdev->delayed_list is necessary thing, just in case a REVOKE_DELAYED_MESSAGES ioctl command (as well as a flush syscall) is invoked.

The implementation of this ioctl switch case is straightforward. It atomically (*spin_lock_bh*) iterates over the whole list of delayed messages, and for each delayed_msg:

1. Tries to delete the associated timer (*del_timer*).
2. If the delete operation fails, it means that the timer callback is being concurrently executed, and the delayed_msg->node was not yet unlinked from gdev->delayed_list. Nothing is done here.
3. If del_timer returns 1, then the timer was successfully deleted and delayed_msg->is_revoked flag is set[2]. In this case:
    a. delayed_msg is atomically *unlinked* the list of delayed messages;
    b. gdev->size is atomically *decreased*;
    c. delayed_msg is atomically assigned to the *garbage collector*;

Once the iteration over gdev->delayed_list is over, the garbage collector for delayed messages is enqueued to the workqueue.

Cost Analysis

Revoking delayed messages has linear cost w.r.t. the number of delayed messages in the group's list.

Flush: group_flush

Similarly to REVOKE_DELAYED_MESSAGES ioctl, also group_flush iterates over gdev->delayed_list, but this time the c.s. (*spin_lock_bh* on gdev->delayed_list_lock) is temporarily left within each loop:

1. Each delayed message is flagged with delayed_msg->is_flushed and unlinked from the delayed list. The critical section is **temporarily left**.
2. *del_timer_sync* is invoked to **synchronously** delete the associated timer:
    a. If it is deleted, the message is atomically published to the group-shared queue.
    b. Otherwise, *del_timer_sync* will only return when the callback execution has terminated executing, on whatever CPU.[3]

[2] Within **delayed_work_fn**, if delayed_msg->is_revoked flag is on, not only the delayed_msg is returned to the delayed_msg_cache, but also the embedded final **message** and its **payload** are properly deallocated.
[3] The delayed_msg->is_flushed tag is used to coordinate with the *timer_callback* routine: if the flag is on, a concurrently running callback function will not deallocate nor unlink the delayed_msg.

3. The delayed_msg is atomically assigned to the garbage collector.

Once the iteration over gdev->delayed_list is over, the garbage collector for delayed messages is enqueued to the workqueue.

Cost Analysis
Same of REVOKE_DELAYED_MESSAGES.

Sysfs: max_message_size, max_storage_size

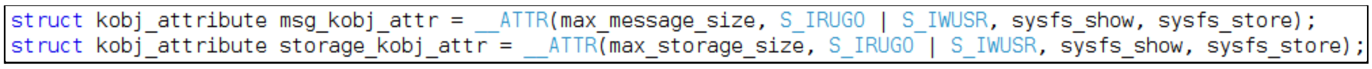
The LKM exposes via the /sys file system, for each group_dev_t, the following reconfigurable parameters:

- ***max_message_size*** : the maximum size (in bytes) for the single messages to be posted on the group's shared-queue.
- ***max_storage_size*** : the maximum size (in bytes) globally allowed for keeping messages in the device file (both in the published_list and in the delayed_list).

Every folder in sysfs corresponds to a *kobject* (or to a *kset*) data structure within the kernel, and every file to a kobj_attribute for that *kobject*. *max_message_size* and *max_storage_size* are exposed as kobj_attributes w.r.t. their group's kobject, which is automatically generated by *create_device* in */sys/devices/virtual*.

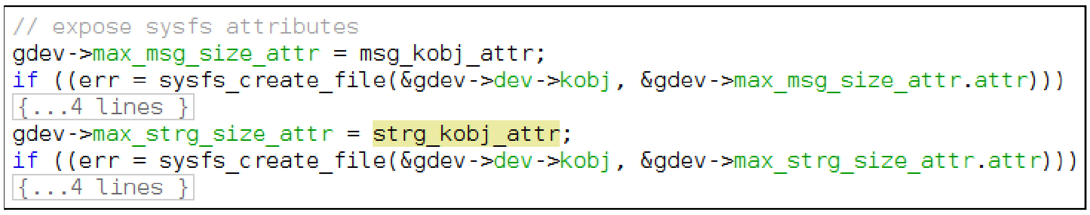
Since a device class (class name = "*groups*") is created (in *initfn*), each group's kobject can be referenced via both (I) */sys/class/groups* and (II) */sys/devices/virtual/groups.* Indeed, the device class is represented as a kset in sysfs, containing its members' kobjects (*__class_register* eventually calls *kset_init* and *device_create* eventually calls *device_add_class_symlinks*).

Below is the gdev_init snippet to initialize[4] new *kobj_attributes*:



```
struct kobj_attribute msg_kobj_attr = __ATTR(max_message_size, S_IRUGO | S_IWUSR, sysfs_show, sysfs_store);
struct kobj_attribute storage_kobj_attr = __ATTR(max_storage_size, S_IRUGO | S_IWUSR, sysfs_show, sysfs_store);
```

Below is the gdev_init snippet to create the sysfs files under the group's kobject directory:



```
// expose sysfs attributes
gdev->max_msg_size_attr = msg_kobj_attr;
if ((err = sysfs_create_file(&gdev->dev->kobj, &gdev->max_msg_size_attr.attr)))
{...4 lines }
gdev->max_strg_size_attr = strg_kobj_attr;
if ((err = sysfs_create_file(&gdev->dev->kobj, &gdev->max_strg_size_attr.attr)))
{...4 lines }
```

Finally, **sysfs_show** and **sysfs_store** were implemented to gap the bridge between the sysfs files interactions and the corresponding effect on the group_dev_t's parameters:

- A switch case is performed on the *kobj_attribute* name to recognize whether the interaction targets the message or the storage attribute.
- A reference to the targeted *group_dev_t* is retrieved via *container_of* on the *kobj_attribute*.
- Depending on the case (show/store), the target attribute (*gdev->max_msg_size* or *gdev->max_strg_size*) is atomically (gdev->size_lock) read or written.

[4] Generally, it's not considered a good practice setting sysfs attribute files as writable by any user. Even the __ATTR macro, eventually, boils down to the macro-assisted check VERIFY_OCTAL_PERMISSIONS that blocks compilation if OTHER_WRITABLE flag is on. Thus, the module only lets *root* reconfigure those parameters.

Udev rules: 83-groups.rules

In src/udev the *83-groups.rules* file specifies the **udev rules** to be loaded on the udev daemon in order to correctly create symlinks for groups' device files in */dev/synch*.

This simple udev rule does it all:

```
KERNEL=="group[0-9]*", SYMLINK+="synch/group%n", MODE="0666"
```

By default, the kernel function *device_create*, deals with the creation of device nodes in /dev.

Hierarchically organizing device nodes within /dev subfolders is not a concern of the kernel, but of the userspace udev daemon. This rule intercepts any device node creation (by the kernel) matching the tokenizable device name "*group[0-9]**" and (I) adds a symbolic link for that device at /dev/synch/group[0-9]*, (II) modifies its permissions to readable/writable for all users in the system.

This rule is installed by copying this .rules file in /etc/udev/rules.d and reloading the udev daemon with the "udevadm control -R" command (superuser privileges are required for these two steps). These steps are performed by the module installation script: /out/scripts/install.sh.

Also, at module finalization:

- While *device_destroy* is invoked on the groups' device nodes, the udev daemon automatically removes their associated symlink from /dev/synch.
- After destroying the last device node, the /dev/synch directory is automatically deleted.

User Space Library

If you rely on some userspace library to wrap access to kernel-level facilities, this is the place to describe its interface to applications and its internal organization.

A userspace library was released together with the kernel-level subsystem: src/include/library/lgroups.h . That header file defines the API offered to userspace applications to wrap access to the kernel level facilities offered by the subsystem.

struct lgroup_t

Thanks to this library, userspace applications become completely agnostic both to the **group_t** data structure and to the ioctl commands exported by the LKM within scr/include/kernel/groups.h .
Indeed, the userspace library *lgroups* defines a new, opaque data structure to operate groups: **lgroup_t**.

The library will internally define the fields for the lgroup_t data structure:

- struct group_t **__group** : the actual group, as exported by the kernel-level subsystem.
- int **__fd** : the file descriptor of the group's device file.

lgroup_init, lgroup_destroy

In order to use groups, userspace applications are expected to declare a lgroup_t pointer and initialize it through the library function **lgroup_init(..)**. Internally, this routine will:

1. Allocate heap space for an actual struct lgroup_t.
2. Initialize both __group and __fd.
3. Return a pointer of the data structure to the caller.

Symmetrically, once the group is no longer required, **lgroup_destroy** shall be invoked, passing the initialized lgroup_t pointer as parameter. Internally, this routine will:

1. Check if the lgroup pointer points to NULL. In that case, the function returns immediately.
2. Check if the __fd field was changed. If that is the case, the file descriptor is closed[5].
3. Finally, free the heap memory reserved for the struct by lgroup_init.

[5] The main purpose of the library is to introduce a layer of abstraction between userspace applications and the kernel-level subsystem. From now on, when mentioning **open/close/read/write/ioctl** operations, I'll be referring to the syscall wrappers defined in the standard library: **unistd.h**. They are, indeed, the function calls bridging the gap between userspace and kernel syscalls, eventually boiling down to the LKM file operations.

API

The lgroups API can be subdivided into logically separated families, which are discussed below.

Every API function has an **int** return type: each of them returns an exit code that informs the caller (in details) about the effects of the call on the system, both in case of failure or of success.

Control Operations

install_group (struct lgroup_t *lgroup, char *group_id)

This function installs a new group, with id group_id, into the subsystem, and configures the referenced lgroup to manage the newly installed group. Return values allow to discriminate the cases in which:

- The installation succeeded.
- The installation succeeded, although the group was already present within the subsystem.
- The installation fails : the actual return value allows to discriminate the specific error scenario. In case the return value is not enough, errno can provide more details.

At the implementation level, this routine does the following:

1. Checks if the main group (/dev/synch/group0) was already opened previously. If not, it is opened.
2. Checks if the referenced lgroup already contains a non-zero file descriptor. In that case, that file descriptor is closed before proceeding.
3. The __group field is initialized, and the INSTALL_GROUP ioctl is invoked.
4. The pathname to the group's device file at /dev/synch is crafted with the returned __group.devname; the device file is opened, and the file descriptor is stored into __fd.

get_max_message[storage]_size (struct lgroup_t *lgroup, unsigned long *size)

This function sets *size* to the current value of the max_message[storage]_size group's parameter. Internally, this function boils down to a read to the sysfs parameter attribute of the targeted device kobject. The read, in turn, eventually boils down to the kernel-level **sysfs_show** function.

set_max_message[storage]_size (struct lgroup_t *lgroup, unsigned long size)

This function sets the max_message[storage]_size group's parameter to *size*.
Internally, this function boils down to a write to the sysfs parameter attribute of the targeted device kobject. The write, in turn, eventually boils down to the kernel-level **sysfs_store** function.

Messaging Operations

publish_message (struct lgroup_t *lgroup, char *msg)

Posts a message to the group-shared queue. Returns the amount of bytes written, or an error code. This function is a wrapper for the standard library write operation.

deliver_message (struct lgroup_t *lgroup, char *buf, unsigned long size)

Delivers a message from the group-shared queue. Returns the amount of bytes read, or an error code. This function is a wrapper for the standard library read operation.

set_send_delay (struct lgroup_t *lgroup, unsigned long delay)

Sets the group's delay to the value passed as parameter (expressed in milliseconds).
This function is a wrapper for the standard library ioctl, with command SET_SEND_DELAY.

revoke_delayed_messages (struct lgroup_t *lgroup)

Revokes all delayed messages for which the delay timer has not expired yet.
This function is a wrapper for the standard library ioctl, command REVOKE_DELAYED_MESSAGES.

Barrier Operations

sleep_on_barrier (struct lgroup_t *lgroup)

Goes to sleep on the group's barrier: the caller is descheduled, and will not be rescheduled until the next *awake_barrier* invocation, by some other thread in the group.
This function is a wrapper for the standard library ioctl, with command SLEEP_ON_BARRIER.

awake_barrier (struct lgroup_t *lgroup)

Wakes up all the threads that are currently sleeping on the group's barrier.
This function is a wrapper for the standard library ioctl, with command AWAKE_BARRIER.

Testcase

Describe here how you have tested your implementation.

A battery of automatic unit tests was written to test out the different aspects of the LKM. They are located at src/test/unit on the source base.

Tests were written with the aid of an open-source *C unit testing facility*: Acutest. It consists of a single C header file, defining the **testing macros** (*TEST_CHECK, TEST_ASSERT*, *TEST_LIST...*) and the **main** function (I.E.: the entry point) for the *automatic unit testing process*.

unit_tests.c

It contains the **TEST_LIST**, I.E.: the active list of unit tests to be executed at runtime. The acutest.h entry point will iterate over TEST_LIST entries, executing each test at runtime. Each unit test is defined within an homonym source file in src/test/unit.

All tests discussed below were run on an Ubuntu 18.04.5 VirtualBox guest system (Linux OS version 5.4.0-70-generic) with 4 CPU cores and 4GB of dedicated RAM.

test_install_group.c

This test checks that the module correctly serves INSTALL_GROUP ioctl requests:

- A group with random group id is installed twice.
- A TEST_ASSERT macro on the return value is called both times, making sure that the first time the ioctl return value is 1 (fresh new group, installed), whereas the second it is 0.
- Also, this test makes sure that the corresponding device file is created under /dev/$devname and /dev/synch/$devname (to also verify the correctness of the udev rules).

test_rw_fifo.c

This multithreaded test checks whether message deliveries happen in FIFO order w.r.t. writes.

I defined two main routines that will be passed as arguments to *pthread_create*: **reader** and **writer**. This test can be configured to run with arbitrary amounts of children processes, although the standard value is 4: 2 readers, 2 writers.

The program is structured in a **producer-consumer** fashion:

- Writers are in charge of: (I) crafting new messages, (II) writing them into the group with id *“fifo”* and (III) writing them into a local-shared buffer of messages.
  - Writes to the LKM group and to the shared resource are serialized within the context of a *writing critical section*, because the program has to make sure that the write order on the group is preserved also locally.
  - Every time a write is performed, a sem_post on the resource semaphore is invoked, to signal the availability of a new message in the resource buffer.
- Readers are in charge of: (I) reading a message from the “fifo” group and (II) from the local buffer, to make sure that the two messages match.
  - Reads from the group and from the shared resource are serialized within the context of a *reading critical section*, not to generate order swaps at read time either.

test_sysfs.c

This test verifies the correctness of sysfs attributes exposition. Hence, the first check it performs is verifying that the running thread has **root privileges**, required to call set_max_message[storage]_size. If that is the case:

- A new group is installed, with id = “sysfs”.
- The initial values of **max_message_size** and **max_storage_size** are read through the dedicated library functions. These are asserted against the initial values set by the LKM (respectively 100 and 10’000).
- They are changed with new values: 200 and 2’100 respectively (two random values).
- A 201 B write is performed, asserting that **message size violations** are detected.
- A loop of legal writes are performed, of size *max_message_size*, to saturate the group’s capacity.
- Another legal *max_message_size* write is performed, asserting that **storage size violations** are detected.
- Finally, original size parameters are restored.

test_barrier.c

This multithreaded test checks the correctness of the group’s barrier functionality.

I defined two main routines, which are passed as arguments to *pthread_create*: **sleeper**, **awaker**.
The father process creates 3 sleeper processes and 1 awaker.

This program is centered on the **semaphore signalling** between the sleepers and the awaker:

1. Each sleeper, as soon as it is spawned, posts the *awake_semaphore* and goes to sleep on the group’s barrier (*group->id = barrier*).
2. As soon as the awaker receives 3 posts on *awake_semaphore*, he invokes AWAKE_BARRIER, waking the group’s barrier up(1).
   After the ioctl command, the writer thread asserts that the return value is 0.
3. As soon as each sleeper is woken up, again, it posts the *awake_semaphore* and waits on a second semaphore: *sleep_semaphore*.
4. As soon as the awaker receives 3 posts on *awake_semaphore*, he again invokes AWAKE_BARRIER, asserting that the return value is **2**: no thread sleeping on the barrier.
5. Finally, the writer posts 3 times on *sleep_semaphore*, and the test restarts from step 1, for **n** times (by default, 10 times).

(1)This step is not really deterministic: it could happen that the last sleeper thread is descheduled soon after the semaphore posting and right before invoking SLEEP_ON_BARRIER ioctl. Also, it might have invoked the ioctl syscall, but, being the Linux kernel preemptible, the thread itself might have been descheduled before entering the *barrier critical section*. If the awaker enters the barrier critical section before the last sleeper thread, the test deadlocks although the LKM barrier implementation is correct. To decrease the chances of this to happen, the writer thread is put to sleep right after receiving the three posts, increasing the probabilities that the last sleeper thread reaches the *barrier critical section* on time. If test_barrier fails on that writer’s assertion, try enlarging the sleep time before blaming the LKM.

test_delay.c

This test verifies the correctness of the delayed operating mode:

- A new group is installed, with id “delay”.
- An arbitrary delay is set (standard value is 700ms).

- A delayed write is performed on the group, with a randomly crafted message. The same thread is used both for write and for read, so that no flush operation is invoked in the meanwhile.
- 3 reads are performed on the new group:
    1. An immediate read, soon after the delayed write. This should return an empty read.
    2. An early read, after sleeping *DELAY - EPSILON* ms, asserting that nothing is read.
    3. A delayed read, after *2 * EPSILON* ms, asserting that the original message is read.

test_revoke.c

This test checks the correctness of the REVOKE_DELAYED_MESSAGES ioctl syscall:

1. It installs a new group with id “revoke”.
2. It sets an arbitrary delay value, 700ms by default.
3. 10 random (30 bytes sized) delayed writes are performed on the group.
4. An early read is performed immediately after the last write, asserting that the amount of bytes read is 0.
5. An early read is performed after sleeping *DELAY - EPSILON* ms, asserting that nothing is read.
6. *revoke_delayed_messages* is invoked on the “revoke” group.
7. A delayed read is performed, after sleeping *2 * EPSILON* ms, asserting that nothing is read.

test_flush.c

This test checks the correctness of the group_flush file operation:

1. It installs a new group with id “flush”.
2. It sets an arbitrary delay value, 700ms by default.
3. 10 random (30 bytes sized) delayed writes are performed on the group.
4. An early read is performed immediately after the last write, asserting that the amount of bytes read is 0.
5. The group is flushed.
6. 10 early reads (soon after the flush) are performed and asserted for equality (also w.r.t. ordering) with the writes at step 3.

test_stress.c

This is a multithreaded stress test for the LKM:

- It operates on multiple groups at once: stress0 up to stress3.
- It spawns 8 threads: 7 stressers and 1 awaker.
- The test has variable duration, by default set to 10s.

A **stresser** thread is a random generator of commands among the following:

1. SWITCH_GROUP: closes the current working group and starts operating on a new one.
2. WRITE: writes a message of gdev->max_message_size on the stresser’s current group.
    - If an EMSGSIZE error is triggered, a *get_max_message_size* is called to update the local copy of max_msg_size for the current group (which could have changed in the meanwhile).
3. READ: max_msg_size bytes are read from the current group.
4. SET_DELAY, UNSET_DELAY: delay is set for the current group (by default 100ms), or restored back to 0.
5. SLEEP
6. REVOKE

7. SET_MSG_SIZE and SET_STRG_SIZE: these commands can only be randomly generated if the testing thread is running with superuser privileges.

Every command is asserted to complete as expected (verifying that the return value is correct) and logged on the output file logs/test_stress.log.

Of course, this test is not capable of asserting the correctness of every command of every stresser with respect to every requirement, but, the longer the test duration, the higher the possibility of checking whether specific execution paths are capable of breaking the LKM implementation. If that is the case, events can be retraced back from the log file.

test_max_install.c

This verifies that the LKM returns the EDQUOT error code as soon as the amount of pre-registered minors is reached (IE.: the group's resource is exhausted).

The **next_group(..)** routine (src/include/test/utils.h) returns the next group number that is expected to be created by the LKM (pattern matching on the /dev folder for group%d).
As soon as the next group is out of the pre-registered range, a new group installation is asserted to fail, and EDQUOT to be found on the errno variable.

test.c

In addition to automatic unit tests, a general purpose **userspace test application** was implemented : src/test/test.c .

This application allows users to interact with the subsystem via command line.
It remaps commands (encoded as command line arguments) to the lgroups library functions shown in the previous chapter, allowing to test out every functionality offered by the kernel subsystem.

Moreover, specifying the command lines [*writer, reader, awaker, sleeper*], the testing application will enter a never ending loop which continuously keeps [writing, reading, awakening, going to sleep] on the selected group. Actually, the *sleeper* application also allows specifying the number of sleeper threads to be spawned for the test.
These never ending applications were used, together with automatic unit tests, to stress out the module at runtime under the more disparate circumstances.

Finally, an helper screen can be shown specifying **-h** as command line argument.

Benchmark

If you have carried out performance tests, this is the place to describe them and give the results.

rw_tps.c

This program was designed to measure the maximum amount of concurrent read/write transactions per second supported by the LKM. It is structured in a way which is similar to test_rw_fifo.c :

- A command line argument determines the **amount of threads** for this program, half of which will be writers, the other half readers.
- A second command line argument determines the **message size**, I.E.: the length of the messages to be posted (with immediate operating mode) and delivered to/from the group-shared queue.

Both the writer and the reader thread implementations consist of a while loop conditioned on a termination variable controlled by the father. Results and statistics about the benchmark are, finally, written by the father thread on a log file : rw_tps.data .

The experiment is conducted on a dedicated group with id “benchmark”:

- The writer threads keep continuously writing the same message (of size **message size**). Each thread keeps locally track of the amount of correct writes performed, as well as those that failed by ENOSPC violations. The latters do not contribute to the computation of the LKM throughput. Therefore benchmark.sh, before invoking rw_tps.out, **adjusts max_message_size** and **max_storage_size** in order to (I) equal the benchmark *message size* argument and (II) reduce as much as possible the likelihood of getting *ENOSPC* write errors. For this reason, benchmark.sh requires superuser privileges.

- Reader threads continuously keep reading new messages from the group, locally keeping track of the amount of (I) correct reads and (II) empty reads (times the group shared queue was empty); this last scenario is something which depends on the actual scheduling of the threads. Again, empty reads will not be considered in the final results. In order to reduce their number as much as possible, since they depend on the actual scheduling activity on the testing machine, reader threads call **sched_yield** as soon as an empty read is performed.[6]

In the **benchmark.sh script**, rw_tps is run 10 times for each combination of load/message size. Results show the average of those runs and, in general, confirm expectations:

- **Given a load value**, the amount of maximum r/w transactions is inversely proportional to the message size. Indeed, the group’s I/O activity will predictably take longer to move data to/from user/kernel space.
- **Given a message size**, the higher amounts of r/w transactions are centered on the amount of CPU cores of the testing machine (IE.: 4). **Less threads** correspond to amounts of userspace requests below potential, whereas **more threads** (EG.: 8) correspond to an exceeding scheduling machinery w.r.t. optimal. In both cases, throughput decreases.

[6] In this way, reader threads yield CPU time to other running threads. Since this benchmark is supposed to run in isolation (with no other stressing userspace threads running), CPU time is supposed to be yielded to writers.

Here are the results of the experiments, conducted on two different versions of the module:

Figure 1

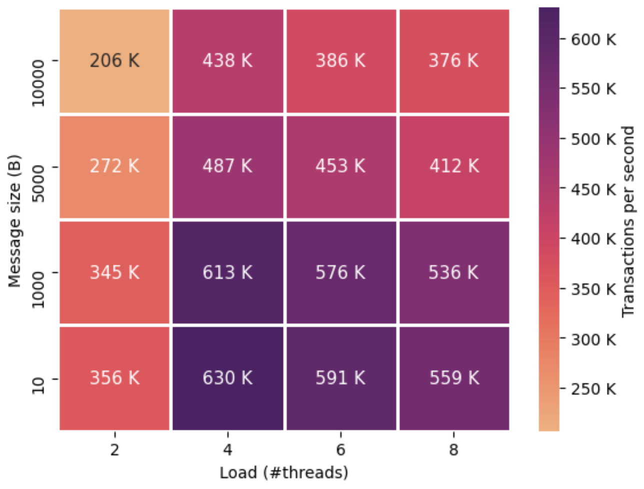
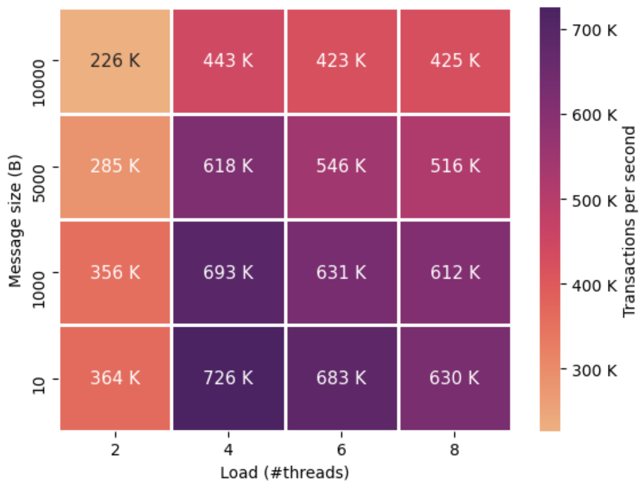


Figure 2



- **Figure 1** refers to the subsystem's implementation with garbage collection facilities, but without lookaside caches (data structures are simply allocated with kmalloc, from the slab allocator).
- **Figure 2** refers to the final version of the subsystem, including both garbage collection facilities and lookaside caches.

Results shown in the heatmaps only capture the amount of **non-empty reads** performed during the experiments. This means that the **true throughput** of the subsystem is **at least double** considering (I) the write operations insisting on those reads and (II) additional, unread writes captured in the log file.