AOSV 2019/2020 Final Project: Thread Synchronization and Messaging Subsystem

Specification

This year project involves the design and development of a subsystem for the Linux kernel which allows threads from (different) processes to exchange units of information (aka messages) and synchronize with each other.

The subsystem works according to the concept of *groups* to orchestrate which threads can synchronize and exchange messages. A group boils down to the creation of a device file in the `/dev/synch/` folder (see note below). Threads (from any process) can use a custom-defined `group_t` type to tell what is the group of threads they want to synchronize/exchange messages with. You can think of `group_t` as a system-wide "descriptor" of a group. Each thread can belong to more than one group (i.e., a thread can use the facilities associated with more than one group to interact with different threads).

The subsystem offers four fundamental primitives:

- *install a group*: by providing a `group_t` descriptor to the subsystem, a new device file is installed only if a corresponding device is not existing. In any case, the path to the corresponding device file is returned. This primitive can be implemented using a custom `ioctl()` call or a new system call, depending on whether you are implementing a module or a patch to the kernel.
- *send a message*: a new unit of information is delivered to the subsystem. This must be implemented by means of a `write()` system call to the device file corresponding to the thread group.
- *retrieve a message*: a previously-sent unit of information is delivered to the caller of this primitive. Messages are delivered in FIFO order, per group of threads. This must be implemented by means of a `read()` system call to the corresponding device file.
- *sleep on barrier*: a thread calling this primitive is descheduled and will not be re-scheduled until it is explicitly woken up by other threads in the system. For this primitive, either a custom `ioctl()` or a new system call should be provided (depending on whether you are implementing a module or a patch to the kernel).
- *awake barrier*: all threads sleeping on a barrier are woken up. For this primitive, either a custom `ioctl()` or a new system call should be provided (depending on whether you are implementing a module or a patch to the kernel).

With respect to messages, the following semantic must be implemented. Each message posted to the device file is an independent data unit (i.e., a mail-slot message in the WinAPI analogy) and each read operation can extract the content of a single message (if present). Messages that are already stored in the device file must be delivered to readers in FIFO order. The message receipt fully invalidates the content of the message to be delivered to the user land buffer (so the message logically disappears from the device file, and cannot be delivered again---exactly-once delivery semantic), even if the `read()` operation requests less bytes than the current size of the message to be delivered. In addition, both `write()` and `read()` operations can be controlled by relying on the `ioctl()` interface, which must support a set of commands for defining the actual

operating mode of `read()` and `write()` for a group. In more detail, `ioctl()` can be used to send these commands:

- `SET_SEND_DELAY`: this command sets the associated group to a mode that does not directly stores the messages to the device file upon `write()` operations, rather they are stored after a timeout (in milliseconds). However the `write()` operation immediately returns control to the calling thread. Clearly, timeout set to the value zero means immediate storing.
- `REVOKE_DELAYED_MESSAGES`: this command allows retracting all delayed messages for which the delay timer has not expired.

The system call `flush()` must be supported, in order to cancel the effect of the delay. If a special device file is flushed, all delayed messages are made immediately available to subsequent `read()` calls.

The driver of the device file must therefore support at least the following set of file operations:

- `open`
- `release`
- `read`
- `write`
- `unlocked_ioctl`
- `flush`

Concurrent I/O sessions on the device file must be supported.

Finally, the kernel must expose via the `/sys` file system the following set of reconfigurable parameters:

- `max_message_size`, the maximum size (bytes) currently allowed for posting messages to the device file
- `max_storage_size,` the maximum number of bytes globally allowed for keeping messages in the device file, so that if a new message post (i.e., a `write()`) is requested and such maximum size is already met, then the post must fail.

NOTE ABOUT DEVICE FILES CREATION: creating folders in `/dev` is not a concern of the kernel. This is something which the userspace `udev` daemon does. To this end, the code should enclose some `udev` ruleset which intercepts the creation of a new device file, and sets symlinks accordngly.

As an example, consider the following line, which is taken from the `50-udev.rules` file on a Gentoo system for the Video for Linux subsystem:

```
KERNEL=="video[0-9]*",  NAME="v4l/video%n", SYMLINK+="video%n", GROUP="video"
```

This line searches for a device that the kernel names `video[0-9]*` -- that is, `video0`, `video1`, and so on. (You can use wildcards, such as `[0-9]` and `*`, in much the same way as you can use wildcards when specifying filenames in a Linux shell.) Once found, the rule creates a device file with a name based on the kernel's internal name in the `v4l` subdirectory of `/dev` (`NAME="v4l/video%n"`), creates a symbolic link to this newly-created device file in the main `/dev` directory (`SYMLINK+="video%n"`), and assigns group ownership of the device file to the `video` group (`GROUP="video"`).

As you should know, `udev` rules must be installed in `/etc/udev/rules.d/`.

How to carry out the work

We are using GitHub Classroom to handle the code and Google Classroom to handle essays. This is also an opportunity to experiment with the git Version Control System.

Follow this **INVITATION LINK** to create a new repo on GitHub, in which you will be carrying out the development work.

If you want to get the best from this experience, remember that git is just a tool. Although it helps you at keeping the work organized, if you use it wrong, you will mess everything up in any case.

Refer to the Git Usage document in the repository to find out some rules which I strongly encourage to follow, when developing collaboratively. In the case of this project, if you follow a *branch-based model*, **you can mention me** (@alessandropellegrini) **on GitHub when you create a pull request**, to ask for intermediate help remotely (please, don't abuse of this possibility!).

Deadline to complete the project

As mentioned, there is **one year period** after the end of the classes to complete the project. This means that all projects should be completed by end of June, 2021. After that date, the system will freeze all repositories, and you will be given a mark on what is found in the last commit on `master`.

What to hand over

To discuss the final project and get the final mark, the students are supposed to provide to the lecturer:

- The implementation of the subsystem
- The implementation of a userspace test application
- The documentation of the code, realized according to the provided template (that's the Google Doc file which you can obtain in Google Classroom).

No printed copy of anything should be carried at the discussion. To discuss the final project, simply send an email to the instructor pointing to the repository where the code is provided ***at least one week before any office hours***. Then, come to the office hours for the discussion (or, if no office hours are there due to the COVID-19 emergency, we'll arrange a meet call to discuss the project).

What will be considered for the mark

Several different aspects will be considered to give the final mark on the project:

- adherence to the specification of the proposed implementation;
- quality of code;
- performance and correctness of the code;
- clarity and quality of the associated documentation;
- good usage of git.