# Operating Systems, 5dv171, Fall 2017

## Project - Linux Kernel Module

**Teacher**
Jan Erik Moström

**Teaching assistant**
Adam Dahlgren

**Group members**
Igor Ramazanov, ens17irv@cs.umu.se
Erik Ramos, c03ers@cs.umu.se
Bastien Harendarczyk, ens17bhk@cs.umu.se

**Gitlab repository**
https://git.cs.umu.se/c03ers/5dv171-project

**Abstract**

The project described in this report implements a Linux Kernel Module that provides a key-value storage, which can be used by different programs to exchange data and store it between program executions.

2017-10-26

# Problem description

The goal with this project is to write a Linux Kernel Module (LKM) that allows different processes running on the same Linux machine, to access shared data through key-value mappings, stored in the kernel. The system must be robust enough, that multiple processes, running on multiple processors, can use it without accidentally corrupting the stored data. A user space application must also be implemented to demonstrate the functionality of the LKM.

# User's guide

The project consists of three parts - the kernel module, a user-space test program and a benchmark program to test the performance of the kernel module. All files required to build these parts can be found in the gitlab repository.

To download the project, begin by cloning the repository typing the following in a terminal:

```
git clone https://git.cs.umu.se/c03ers/5dv171-project
```

Each part is assigned to a separate directory; `module` for the kernel module, `program` for the user-space program and `benchmark` for the benchmark program. Each of these directories contain their own make-files, so building any of the parts can be done by typing `make` in the terminal, from the respective directory.

Once built, a super-user may install the module by typing `insmod shared_map.ko` in the terminal, while in the module directory. Running `rmmod shared_map` will remove the module.

The test program *test*, can be used both for putting data into the module and to retrieve data from it. Inserting data is done like this:

```
./test set key value
```

To retrieve data from the module using the test program, instead write this:

```
./test get key
```

Data entered into the module is saved as files in

```
/var/tmp/shared_map/
```

and will remain on the hard-disk even after removing the module. Re-installing the module will recover data from previous installations.

# Solution

To implement the kernel module, four problems had to be addressed; how to internally represent the key-value mappings, how to store the mappings between module installations, how to create the mappings in the kernel from user-space and how to ensure that the module could be safely used in a multi-threaded environment.

## Internal data representation

To store the data in a hashtable seemed like a natural choice, since it would be both fast and provide the same functionality required from the module. While implementing a new hashtable was always an option, the Linux kernel comes with its own datatypes. The datatype chosen to represent the mappings ended up being `rhashtable`.

The `rhashtable` provides per-bucket locking, which allows processes on different processors to modify the datatype in parallel, as long as they are working in different buckets. Additionally, RCU synchronization allows both reading and writing of data to *the same* bucket at the same time. This is done by deferring updates until no threads are trying to read the data. Care must be taken though, to make sure that reads don't happen during the updates.

## Synchronization

Three mechanisms to ensure mutual exclusion that are provided by the Linux kernel are `spinlocks`, `mutexes` and `semaphores`.

### Spinlock

A spinlock is a lock. If a thread is trying to acquire it, it has to wait in a loop while repeatedly checking if the lock is available.

When a thread tries to lock a spinlock and it does not succeed, it will continuously re-try locking it, until it finally succeeds; thus it will not allow another thread to take its place.

### Mutex

A mutex is a synchronization primitive used to avoid shared resources to be used at the same time. The mutex has two states: lock or unlock.

When a thread tries to lock a mutex and it does not succeed, because the mutex is already locked, it will go to sleep, immediately allowing another thread to run. It will continue to sleep until being woken up, which will be the case once the mutex is being unlocked by whatever thread was holding the lock before. Putting threads to sleep and waking them up again are both rather expensive operations, they'll need quite a lot of CPU instructions and thus also take some time.

### Sempahores

A Semaphore is a variable used to control access to a common resource. A semaphore is a data structure that is maintained by the operating system and contains:

- an integer that stores the value, positive or zero, of the semaphore.
- a queue that contains pointers to threads that are stuck waiting on this semaphore.

When initialized to 1, a semaphore can be used in the same way as a mutex. When a thread gets access to the the semaphore, the semaphore counter will decrease.

When a thread free the semaphore, the semaphore counter will increase. A thread, that is waiting for the semaphore, will go to sleep as in the Mutex case.

In Linux, there is a special kind of semaphore called: reader/writer semaphore. It allows multiple readers into the critical section but only one writer at any given time.

**Conclusion**
After discussing these solutions, we decided to implement spinlock to synchronize our threads. This solution allow us to minimize the amount of CPU instructions.

When we started to benchmark our system with several processes, we noticed that our system was freezing when several write operations were running.

After considering this problem, we concluded that there was deadlock. That deadlock was due to the policy of spinlock. When a thread is waiting for the spinlock to be unlock, it's blocked in loop and ask at each iteration of the loop. So any other thread can run because the previous thread is still running.

So we moved to another solution. First we tried to implement read/write semaphores, because those semaphores are designed for those operations. So we set up this solution. But during the compilation there was an error. We tried to find the cause of this error but we didn't succeed.

Therefore, we decided to use mutex. And now it works. We assume that this solution uses more CPU operations to put in sleep or to wake up and takes more time. But it's the best solution to avoid deadlock as when a thread is waiting it's put in sleep and allow other another thread to run.

## Communication

Different methods to communicate with the kernel from user-space exist. Among these are system calls and writing to special files, such as for example character or block devices. A third option is to use sockets to do the communication.

System calls requires recompilation of the kernel which runs counter to what modules were designed to do. Writing to special files would solve the problem, but seemed like a bad idea, because they usually represent an acutal, physical device. For this reason, sockets were chosen to do the communication.

The sockets (called `Netlink sockets`) work much like regular sockets used for communication over the Internet. Using them requires the definition of a comunication protocol.

**The communication protocol**
Communication is initiated by a request from the user-space program and regardless of the outcome, the kernel responds in some way.

The requests available to the user are to create a new mapping and to look up a value associated with an existing mapping.

For lookups, if a key *does not* exist, the kernel will respond with a message indicating this. If the key *does* exist, the mapped value is returned. For insertions, the kernel may return one of two messages; one indicating that a new mapping was created and another to indicate that an old mapping was updated with a new value. An additional message to indicate errors of some other kind (such as for example out-of-memory) is also avaible.

**The message structure**
Communication between user-space and the kernel is done by sending message structures over Netlink. The `Message` structure has the five fields; `type`, `key_length`, `value_length`, `key` and `value`. `type`, `key_length` and `value_length` are all integers and can easily be passed back and forth between kernel and user-space, but since both keys and values can be arbitrarily long, they must be dynamically allocated. Passing pointers from user-space to the kernel is possible, but the converse is not. Because of this, the message must be serialized before transmission. This process is done by special message building functions.

To preserve the illusion of sending the message as a structure with pointers, some (perhaps overly so) clever pointer magic is done. When a message is created by one of the builder functions, the size of the memory allocated is that of the message structure itself plus the combined size of the key and value. The key is copied into the area directly after the message structure fields and the value is copied directly after it. The key- and value pointers are then set to point into the data region where the key and value are stored respectively.

One consequence of this method is that the length of the data sent is not `sizeof(struct message)`, but rather a value that can be extracted by a function called `message_length`. Another is that the pointers received on the other end of the netlink connection will not make sense to the receiver, so the first thing the receiver does once it receives the message is to update the pointers to something that makes sense. This is done by a call to `message_cast`.

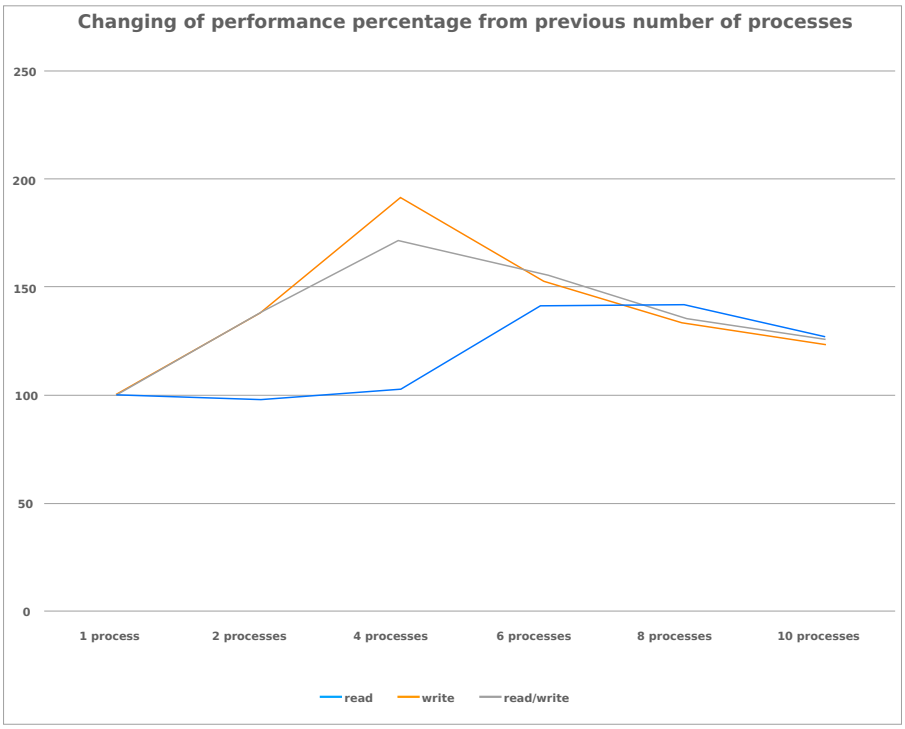
### Persistant storage

## Performance analysis

## System limitations

## Discussion

Developing for the Linux kernel is not an easy task. Much of the information available online is outdated and the kernel in many places, is poorly documented. Most of the time spent on the project, was dedicated to sifting through kernel code, trying to to



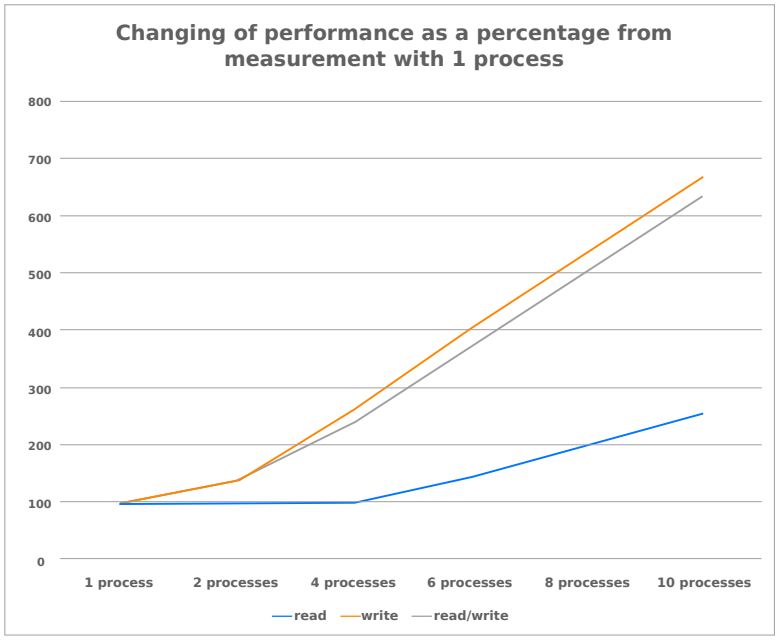




figure out what the functions do, which functions to use and in what contexts they are safe to use.

Debugging is also difficult, in part because many tools run only in user space. Another

| Results (all measurements in milliseconds): | | | |
|---|---|---|---|
| | **read** | **write** | **read/write** |
| **1 process** | 0.926 | 3.082 | 4.049 |
| **2 processes** | 0.897 | 4.223 | 5.596 |
| **4 processes** | 0.928 | 8.076 | 9.618 |
| **6 processes** | 1.316 | 12.371 | 15.067 |
| **8 processes** | 1.861 | 16.568 | 20.408 |
| **10 processes** | 2.363 | 20.563 | 25.690 |

| Same measurements caluculated as percentage from previous processes: | | | |
|---|---|---|---|
| | **read** | **write** | **read/write** |
| **1 process** | 100 000 | 100 000 | 100 000 |
| **2 processes** | 96 868 | 137 005 | 138 195 |
| **4 processes** | 103 400 | 191 249 | 171 893 |
| **6 processes** | 141 833 | 153 190 | 156 654 |
| **8 processes** | 141 458 | 133 924 | 135 444 |
| **10 processes** | 126 967 | 124 116 | 125 884 |

complicating factor, is the fact that the operating system can freeze when there is a bug in the code and that error print-outs are lost when rebooting.