SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Registration number: FEI-5384-72983

CONCURRENCY IN LSM MEDUSA

MASTER'S THESIS

2018 **Bc. Roderik Ploszek**

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Registration number: FEI-5384-72983

CONCURRENCY IN LSM MEDUSA

MASTER'S THESIS

Study Programme:	Applied Informatics
Field Number:	2511
Study Field:	9.2.9 Applied Informatics
Training Workplace:	Institute of Computer Science and Mathematics
Supervisor:	Mgr. Ing. Matúš Jókay, PhD.

Bratislava 2018 **Bc. Roderik Ploszek**

Slovak University of Technology in Bratislava
Institute of Computer Science and Mathematics

Faculty of Electrical Engineering and Information Technology
Academic year: 2017/2018
Reg. No.: FEI-5384-72983

STU FEI

MASTER THESIS TOPIC

Student: **Bc. Roderik Ploszek**
Student's ID: 72983
Study programme: Applied Informatics
Study field: 9.2.9. Applied Informatics
Thesis supervisor: Mgr. Ing. Matúš Jókay, PhD.
Workplace: Ústav informatiky a matematiky

Topic: **Concurrency in LSM Medusa**

Language of thesis: English

Specification of Assignment:

The goal of this thesis is to allow concurrent communication of the LSM Medusa system with an Authorization server. For this reason, the architecture of Medusa in the Linux kernel has to be revised.

In the current version, LSM Medusa doesn't support concurrent execution of multiple system calls. These system calls are used for communication between Medusa system in the kernel controlled by the operating system and authorization server, that decides if the monitored operation is allowed or dismissed. This is because earlier versions of Medusa were developed for kernels without the support of concurrent code execution. Circumstances and possibilities of hardware (and therefore the possibilities of operating system kernels, that create an abstraction for this hardware) have substantially changed. To increase effectiveness of Medusa system, it's necessary to extend LSM Medusa with as much of concurrent execution of code as possible.

Tasks:
1) Analyze current state of LSM Medusa concerning the concurrent execution of code
2) Analyze options of extending/reworking/replacing those parts of the system where concurrent execution of code is possible
3) Propose a change of LSM Medusa that adds support of concurrency
4) Implement your proposal
5) Evaluate the contribution of your work

Selected bibliography:

1. Káčer, J. – Jókay, M. *Medúza DS9.* Diplomová práca. Bratislava : FEI STU, 2014. 35 s.

Assignment procedure from: 18. 09. 2017

Date of thesis submission: 11. 05. 2018

Bc. Roderik Ploszek
Student

prof. RNDr. Otokar Grošek, PhD.
Head of department

prof. Dr. Ing. Miloš Oravec
Study programme supervisor

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Bc. Roderik Ploszek
Diplomová práca:	Konkurencia v LSM Medusa
Vedúci záverečnej práce:	Mgr. Ing. Matúš Jókay, PhD.
Miesto a rok predloženia práce:	Bratislava 2018

Táto práca sa zaoberá implementáciou podpory konkurencie pre bezpečnostný modul Medusa v operačnom systéme Linux. Pôvodná verzia modulu podporovala iba sériovú autorizáciu a nevyužívala vlastnosti nových viacjadrových systémov. To znamená, že keď prebieha autorizácia jedného systémového volania, zvyšné procesy v systéme nemôžu vykonávať systémové volania, až kým nebude rozhodnuté o prvej žiadosti. V novej verzii Medusy bola implementovaná fronta, do ktorej môžu procesy konkurentne vkladať žiadosti o autorizáciu. Tieto žiadosti sú postupne odosielané autorizačnému serveru, ktorý vykonáva rozhodnutie. V práci popisujeme spôsob implementácie fronty, spôsob alokácie pamäte pre prvky fronty a riešenie problémov, ktoré vznikli počas vývoja modulu. Taktiež popisujeme použité synchronizačné nástroje na ochranu zdieľanej pamäte. Na konci práce sa venujeme meraniu výkonu po pridaní podpory konkurencie. Popisujeme použité metódy, analyzujeme výsledky meraní a navrhujeme ďalšie zlepšenia riešenia. V prílohe práce popisujeme spôsob ladenia operačného systému Linux spolu s ďalšími technickými popismi bezpečnostného modulu Medusa.

Kľúčové slová: Medusa, Constable, mYstable, Linux, LSM, bezpečnosť, modul, jadro

ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Author:	Bc. Roderik Ploszek
Master's thesis:	Concurrency in LSM Medusa
Supervisor:	Mgr. Ing. Matúš Jókay, PhD.
Place and year of submission:	Bratislava 2018

This thesis presents an implementation of concurrency support for the Medusa security module on the Linux operating system. The original version of the module supported serial authorization only and didn't use the facilities of new multicore systems. This means that when an authorization of a system call is in progress, no other processes could execute system calls until the authorization was finished. A queue for decision requests was implemented in the new version of Medusa. Processes requesting an authorization inserts a decision request to the queue. These requests are sequentially sent to the authorization server that carries out the authorization. In the thesis we explain the implementation of the queue, method of memory allocation for elements of the queue and solutions of problems that occurred during the module's development. We also explain the synchronization tools used to guard shared memory access. Lastly, we measure the performance changes after adding the support of concurrency. We explain the methods used, analyze the measured results and propose further improvements. In the appendix, we describe process of debugging the Linux kernel with other technical descriptions of the Medusa LSM.

Keywords: Medusa, Constable, mYstable, Linux, LSM, security, module, kernel

Acknowledgments

I would like to thank my thesis supervisor, Matúš Jókay for his valuable suggestions, knowledge, guidance and also for creating a good atmosphere on our consultations.

Contents

List of Figures and Tables

List of Algorithms

List of listings

Introduction

This thesis focuses on the implementation of concurrency support for the Medusa LSM. Medusa is a security module for the Linux operating system that enhances possibilities of security enforcement in Linux. Functionality of Medusa is divided into two parts. The first one is a kernel module that uses LSM framework to integrate into the Linux kernel. It has layered architecture consisting of five layers (L0–L4). The second one is an authorization server that authorizes operations in the system. Authorization server can handle authorization of more than one host using a network interface. Currently, two implementations of authorization servers exists. *Constable*, written in C is the original authorization server. For its configuration, it uses a language similar to C that is later compiled to bytecode and run by a virtual machine. Because of that, *Constable* is quite complex and also fast. Second implementation, *mYstable* is written in Python. It trades simplicity of configuration for slowness of execution. It is designed to be used during development for rapid prototyping of new features.

Module in the kernel and the authorization server communicate using a character device. This is implemented in the fourth layer of the Medusa kernel module. This layer also contains function that handles decisions sent to the authorization server. Original version of Medusa supported only serial decision of events. After an event decision is requested, no other system calls can be executed until the decision request is answered by the authorization server. This thesis removes this deficiency and allows multiple processes to request a decision at the same time.

In the first section, we present the Medusa LSM along with its infrastructure. We explain the LSM framework, Medusa in the kernel and finally, two authorization servers. In the second section, we describe the communication layer. Section starts with a summarization of the original state of Medusa before changes. Then it explains the functionality of a queue to temporarily store decision requests that are sequentially sent to the authorization server. It also explains method used to allocate free memory to store the queue elements. Lastly, it explains problems encountered during development with solutions to these problems. Last section focuses on performance analysis of the new concurrent solution. Testing was performed with different number of processes concurrently running, executing system calls, thus creating decision requests for the authorization server. Lastly, we analyze gathered testing results and propose an improvement that could help when the computer is running many processes.

1 Introduction to LSM Medusa

In this section we will explain the LSM Medusa system in detail. We will describe its architecture and operating principles.

1.1 LSM

Medusa is using LSM framework to integrate into the Linux kernel. Linux Security Modules is a framework that provides an interface for loadable kernel modules that can enforce arbitrary security models.

LSM has been included in the Linux kernel since version 2.6. It made the process of implementing a security module easier for its developers. Thanks to a generic interface, security modules don't have to patch kernel code or reimplement kernel functions and users can choose any framework (or more than one if they support module stacking) they want to use.

LSM modifies the kernel in various ways to provide the most generic and efficient interface for security modules with diverse requirements [1]:

- LSM provides opaque security fields in numerous kernel data structures for security modules to use. These are of `void*` type, meaning they can point to any data structure defined by a security module. Data structures containing the security field include `task_struct`, `inode`, `net_device` and others.

- LSM allows to mediate access to internal kernel structures. It allows this by placing hooks in specific parts of the kernel — usually before such an access is granted.

  Security modules can assign their own callbacks to hooks provided by the LSM framework by registering them. Hook registration usually happens during initialization of a security module, but it can happen even after the security module is initialized. Hooks can be also unregistered, though this is not very common.

  LSM provides only restrictive access control, meaning that an operation that would be allowed by a standard DAC control can be denied by a security module. However, opposite is not true. An operation that would be denied by the DAC control can't be forcibly allowed, since that event never gets to a security module.

  When an event is denied, an error is returned to the caller. There's no way to return a positive result when an operation has been denied. This is one of LSM's disadvantages.

In listing 1 we show an LSM hook in the `mkdir` system call implementation. Function `may_create()` checks for DAC permissions. LSM hook `security_inode_mkdir()` goes through a list of registered callbacks for this hook and calls each of them with arguments passed to the system call. Security module can then access its security structure bundled with the system call objects and together with other information (depending on the implemented security model) determine, if the system call is allowed or denied.

```
int vfs_mkdir(struct inode *dir, struct dentry *dentry, umode_t mode)
{
        int error = may_create(dir, dentry);
        unsigned max_links = dir->i_sb->s_max_links;

        if (error)
                return error;

        if (!dir->i_op->mkdir)
                return -EPERM;

        mode &= (S_IRWXUGO|S_ISVTX);
        error = security_inode_mkdir(dir, dentry, mode);
        if (error)
                return error;

        if (max_links && dir->i_nlink >= max_links)
                return -EMLINK;

        error = dir->i_op->mkdir(dir, dentry, mode);
        if (!error)
                fsnotify_mkdir(dir, dentry);
        return error;
}
```

Listing 1: Implementation of `mkdir` system call in the Linux kernel

Since the introduction of LSM to the kernel, many security modules have been included with the kernel. These are *AppArmor*, *SELinux*, *Smack*, *Tomoyo* and *Yama*.

Some security modules have not been transferred to use LSM and still continue to be developed as kernel patches. Authors of these projects criticize LSM for various reasons. These include *RSBAC* or *grsecurity*.

1.2 Medusa

Medusa is a security module for the Linux operating system that provides an additional layer of security to the standard security facilities of the Linux kernel.

1.2.1 History

LSM Medusa was created on Faculty of Electrical Engineering and Information Technology STU. Version Medusa DS9 was actively developed during years 2000-2004 with the last supported kernel version 2.4.26 [2].

Development of the project then stalled for ten years. In 2014, the development was resumed by Ján Káčer who modified Medusa to support new versions of Linux kernels along with 64-bit support [3]. The new version was named Medusa Voyager. A year later, a team project worked on transferring authorization server Constable to support 64-bit architecture [4].

Work on Medusa continued with other bachelor theses and a team project. These included a testing environment [5], support for new system calls [6], monitoring process activity using Constable [7], support for *dentry* hooks together with an addition of a new layer [8].

Development of Medusa continues with this thesis and two others, one focused on adding support for mediating IPC communication and other one focused on developing a new authorization server usable for development purposes—*mYstable*.

1.2.2 Architecture

Medusa is distinct from other security modules in its architecture. Only a small part of Medusa code is in the kernel. The actual authorization logic is located in a user process, called authorization server. This has several advantages:

- Authorization server can be implemented in any language, suited to the needs of the end user. Programmer doesn't need to learn the Linux kernel programming to make changes in Medusa authorization.
- One authorization server can serve many network nodes and adjust accordingly to what's happening on the network.
- Kernel footprint is small, since Medusa uses small bit array (size can be chosen by the user) as its security structure.
- Authorization server can use services provided by the operating system, such as memory allocation and file operations.

- Medusa decision can be as fast as an AND operation.

Medusa uses simple abstraction on kernel entities. Based on the role they serve in an operation these are *objects* and *subjects*. *Subjects* are entities that execute an operation (access). These include processes and threads. *Objects* are entities on which the operation (access) is executed. These include files, processes or IPC.

As can be seen, some entities can be both a *subject* and an *object*. For example, process sending a signal is a *subject* and process receiving a signal is an *object*.

Objects are then assigned to virtual subsystems called *virtual spaces*. One object can be assigned to multiple *virtual spaces*. Subjects are also assigned to *virtual spaces*, but each operation of a subject can have different set of *virtual spaces*. There are three possible operations for subjects:

Read reading information from an object

Write writing information into an object

See the ability to check the existence of an object

When an operation is authorized, Medusa checks for virtual spaces corresponding to the *object* and the *subject's* operation. If there is no intersection, the operation is denied right away without consulting the authorization server. If virtual spaces of *object* and *subject* intersect, Medusa checks if the authorization server monitors the current type of access. If so, a decision request is sent to the authorization server.

Virtual spaces are implemented as a bit map. Therefore, intersection of *virtual spaces* can be checked using binary AND operation. User can choose number of virtual spaces at the time of compilation. Default value is 32 virtual spaces (one integer).

This abstraction makes it possible to implement any security model in the authorization server suited to the user. In comparison, other security modules enforce security models chosen by their developers.

To make authorization server as portable as possible, Medusa uses special abstraction of kernel objects for communication. Hence they are called *k-objects*. These objects are transferred between the kernel and the authorization server.

K-class *K-objects* are defined on the second layer by *k-classes*. *K-classes* contain:

- Name of the *k-object*
- Definition of attributes contained in *k-objects*

- Operation methods for
  - fetch
  - update

K-objects are defined in the kernel by two structures:

***K-object* structure** This structure defines data objects transferred between the kernel and the authorization server. It has to contain `MEDUSA_OBJECT_HEADER` as its first field. Definitions of data fields specific to the kernel object follow. For example, these can be process identification numbers for processes or device number and inode number for files. Then object or subject variables are defined based on the type of the *k-object*. If the *k-object* can act both as a subject and an object, both are defined. Structure of object and subject variables is the same for every *k-object*.

Object variables contain a bitmap of virtual spaces this object belongs to, actions of the object monitored by the authorization server, magic number for keeping data in the *k-object* valid and a special 64-bit number `cinfo`, used by the authorization server to save additional data. Usually it is used to identify a *k-object* in the authorization server.

Subject variables contain three bitmaps of virtual spaces, one for each operation, monitored actions and `cinfo` value for the authorization server.

Attribute definition This structure defines all attributes of a *k-object*. It includes attribute type (mutable/non-mutable), size, name and position in the structure. This is important for remote authorization servers that don't have access to the C structure definition.

To create abstractions for the authorization server, the architecture of Medusa is layered. Explanations of layers follow.

Zeroth Layer This layer has been added recently [8] to solve problems with objects (namely *inodes* and *task security contexts*) created before Medusa was initialized.

It registers four hooks and saves mentioned objects on linked lists for later initialization when objects from second layer become available.

First Layer This layer registers Medusa functions to LSM hooks. When it's initialized, it also unregisters temporary hooks registered by the zeroth layer and properly initializes objects collected by the zeroth layer.

Second Layer This layer defines *k-objects*, *event types* and *access types*. It's also responsible for implementing conversion functions of kernel objects to *k-objects* and *fetch*

and *update* functions for *k-objects*.

Third Layer Third layer is responsible for registering the *k-classes* with the authorization server. It also provides definitions of data types used for representation of *k-objects*, interfaces between second and fourth layer and architecture independent functions for locking, printing and bit operations.

Fourth Layer This layer controls communication between the kernel and the authorization server. This thesis concerns mostly this layer. More information can be found in section 2.

1.3 Authorization server

Authorization server is a userspace process that makes decisions about monitored operations. It communicates with the kernel using a character device `/dev/medusa` or, in the case of monitoring a network of computers, using a network interface. Kernel can execute these operation on the authorization server:

Decision request Authorization server can selectively monitor operations. These operations are sent to the authorization server for authorization. This is usually done when the *virtual spaces* model is not sufficient for the authorization of the operation.

Kernel sends information about subject and object that the authorization server needs for the decision. These information are stored in *k-objects*.

Authorization server answers one of the following:

Allow Absolute permission to execute the operation. Overrides the standard UNIX permissions. This can't be done in the current version of Medusa, as it uses the LSM hooks.

Deny Absolute prohibition of the operation.

Skip Operation is forbidden, but kernel returns succesful return value (as if the operation was executed). Again, this can't be done in the current version of Medusa for the same reason stated earlier.

OK Allows the execution of the operation (if the UNIX permissions allowed it).

Object or event registration Kernel registers *k-objects* and *event types*, so authorization server can read and work with them.

Authorization server supports two operation in the kernel:

Fetch request Authorization server makes this request when it demands a *k-object* from the kernel.

Update request Authorization server makes this request when it wants to update an existing object in the kernel with new information.

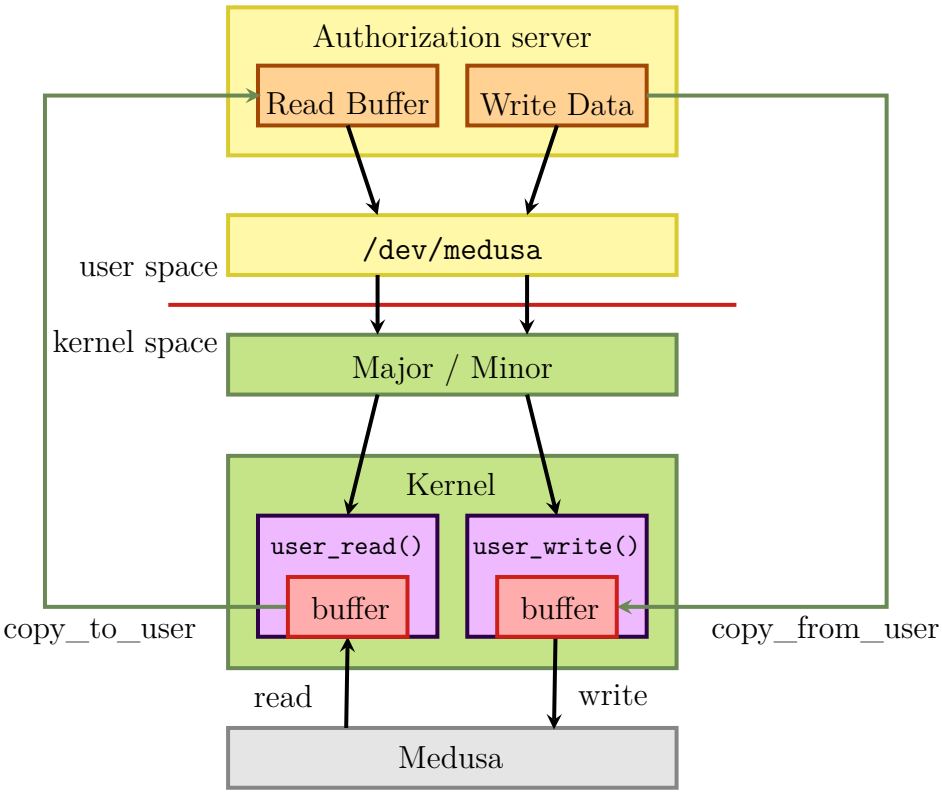


Figure 1: Communication between the kernel and the authorization server, adapted from [9]

1.3.1 Constable

Constable is the original authorization server written in C. [10] The main part of *Constable* is the *unified namespace* that stores system and authorization server objects in a tree hierarchy. Root node of this tree contains branches for other types of objects. For example, branch *fs* contains files as they are located in the linux root directory and branch *proc* contains processes in the system stored according to their hierarchy. There is also a branch for *RBAC* module and new branches can be created for storage of any data (that doesn't necessarily have to represent system objects) in a tree structure.

The purpose of the *unified namespace* is to identify objects, not to store an actual state of the system. This means that object that doesn't exist on the monitored system can be added to the tree.

Constable has a modular architecture. There's a *main module* that controls the communication interface, parses the main configuration file and provides an infrastructure for other modules. Then there are modules that provide different security models: *Basic security module*, that provides hierarchical order for files and processes as explained above, *Domain module*, that saves processes to domains with no hierarchy and *RBAC module* that implements role-based access control with its own configuration file.

Infrastructure *Main module* (or infrastructure) of *Constable* provides basic operational tasks for other modules. This includes:

- Creation of new virtual spaces, allocating one bit for each. A virtual space is identified by its name.
- Management of the *unified namespace* tree
- Abstractions of access types. *Constable* uses seven different access types. First three are the same as used in Medusa (read, write and see). Other four used internally by *Constable* are:

  Create Create a new non-system object

  Erase Delete a non-system object

  Enter Acquire a state. For example, moving a node to a different place in the tree.

  Control Modify properties of an object

- Management of communication objects.
- Management of subjects and their privileges.
- Distribution of events.
- Parsing the configuration file.

Configuration file Basic configuration defines:

trees These are branches created below the root node. They are usually specified with an event that automatically inserts them into the tree. For example, event *getfile* can insert new objects to the tree based on its property *filename*.

spaces These are definitions of virtual spaces. Virtual spaces can contain just one node or a whole branch of the tree.

access types These are privileges for operations between virtual spaces.

event handling Constable allows definitions of functions that handle events with more precision. These functions are written in a language similar to C.

Event handling When *Constable* is starting up, each module will register callback functions for events based on their configuration. When an event occurs, all registered functions are called and modules make their decision. Final decision is chosen as the most restrictive of all returned decisions.

1.3.2 mYstable

New authorization server, *mYstable* is written in Python. It was designed to be used during development as Python allows rapid application development without need to manage the memory at the cost of execution speed.

Configuration of *mYstable* is done via a Python module. This means that configuration can use all features of Python programming language along with external libraries to add additional functionality.

mYstable doesn't support management of virtual spaces and domains in a tree structure as *Constable* does. Instead, handlers can be registered using the `Register` decorator from `framework` module to authorize decisions requests. These functions have access to subject, object and type of the event along with all attributes of these objects. Since Python supports object oriented programming paradigm, *k-objects* and *events types* are represented by Python objects.

mYstable allows communication with any number of hosts. Communication is supported via a character device or a network interface. Hosts are configured in a *json* file parsed at authorization server startup. Configuration of one host consists of:

- host name
- path to configuration module
- communication type (character device or network interface)
- path to the device or network address

mYstable supports communication with little or big endian architectures and implements its own `Bitmap` class for bitmap operations built on top of `bitarray` external module.

Each host defined in the configuration file gets its own thread for operations. In a host thread, two helper threads are created:

Write thread reads data from a queue and sends it to the kernel.

Request thread authorizes decision requests

In the current version, authorization thread can process just one decision request at a time, but in the future it will create additional threads that can authorize concurrently.

See figure 2 for a schematic of threads in *mYstable*. Planned thread functionality is shown in dashed lines.

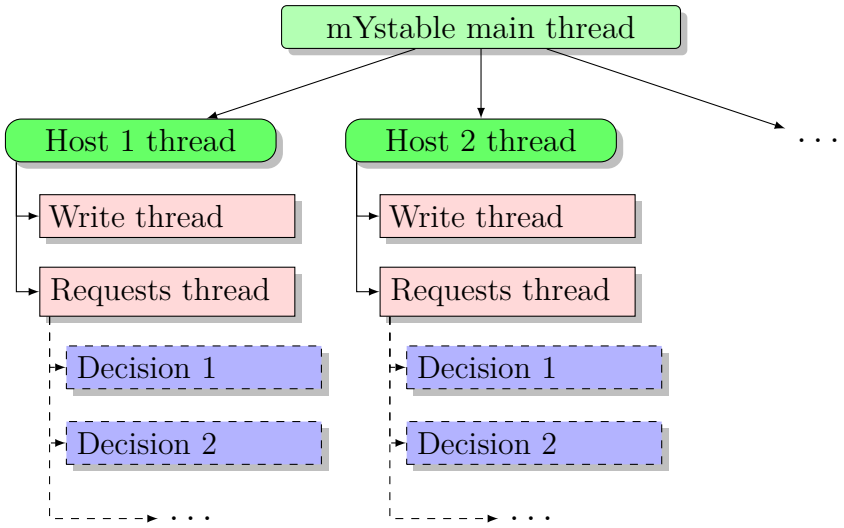


Figure 2: Thread organization in *mYstable*

2 Communication layer

Communication layer, or *L4* provides an interface between the authorization server and the kernel. All decision requests and registration messages from the kernel, decision answers, update and fetch requests from the authorization server go through this layer.

2.1 Original state

In this section we present the original state of the communication layer before our changes.

2.1.1 Basic description of communication

There are three possible basic states for the communication layer:

Opening state During this state the authorization server is starting up. This is done by calling `open` on the character device file descriptor. This function is defined in the kernel as `user_open()`.

It works as follows: first, it checks if the kernel is not already connected to an authorization server. If it is connected, it will return an error value `-EPERM`. If not, kernel will prepare a hello message for the authorization server and saves it into the *teleport*. The teleport is explained in section 4. Based on the hello message, the authorization server can determine the endianness of the authorized system and adjust its messages accordingly. Then it proceeds to fill out two linked lists: `evtypes_to_register` and `kclasses_to_register` with event types and *k-object* definitions accordingly. Thanks to these definitions, the authorization server can read *k-objects* and *event types* sent from the kernel.

After `open` sucessfully finishes, authorization server can call `read` on the communication file descriptor which calls the `user_read` function in the kernel. Authorization server receives the hello message and continues with the registration of *k-objects* and *event types* (both in the code also referred to as *announces*).

Working state Registration of *k-objects* or *event types* has higher priority than decision requests. This means that before the first decision request is done, kernel has to register all objects in the aforementioned linked lists. As mentioned earlier, most of *k-objects* and *event types* are registered after the authorization server startup, but registration or unregistration can happen anytime during the operation of the authorization server. Medusa supports object unregistration, but function for handling it hasn't been implemented yet.

In this state, the authorization server can call two functions on the communication file descriptor: `read` and `write`.

read Operation `read` is implemented in the `user_read()` function. It is used to send data from the kernel to the authorization server. Based on the state of signal variables in the kernel, this function can send different data. If any *fetch* or *update* operations are waiting, these are executed first. Registrations of *k-objects* and *event types* are done when all previous operations are finished. If there are no fetch or update requests or registration announces, decision requests are processed.

For each case, the procedure is similar. The *teleport* is filled with data about the event and afterwards it is sent to the authorization server.

One special property of the `read` function is that the authorization server can read one structure field by multiple `read` calls. This property was added, so a network client that doesn't understand the communication protocol can be used [11].

This works thanks to the *teleport* that remembers how much data has been sent and how much data is needed to be sent.

write Operation `write` is implemented in the `user_write()` function. It is used to send data from the authorization server to the kernel. This can happen in three cases:

- Authorization server has decided about a decision request and has an answer for the waiting process.
- Authorization server wants to *fetch* data about a *k-object* from the kernel.
- Authorization server wants to *update* a *k-object* in the kernel.

Closing state Closing state begins when the authorizations server closes the communication file descriptor or if a fatal error occurs. This operation calls function `user_release()` in the kernel.

If the authorization server is not running, it returns zero. Otherwise, it starts by unregistering *k-objects* and *event types*. Registered objects are stored in two linked lists, `evtypes_registered` and `kclasses_registered`. After all objects are unregistered, system may shutdown or reboot depending on the configuration set during compilation. If this configuration was not set, function sets the atomic variable `constable_present` to zero. If there was an unanswered decision request, it returns an error value. Finally, global variables are reset to default values.

Graphical representation of the full communication protocol in the form of UML sequence diagram can be found in figure 3. Please note that operations *update* and *fetch* can happen even outside of working state.

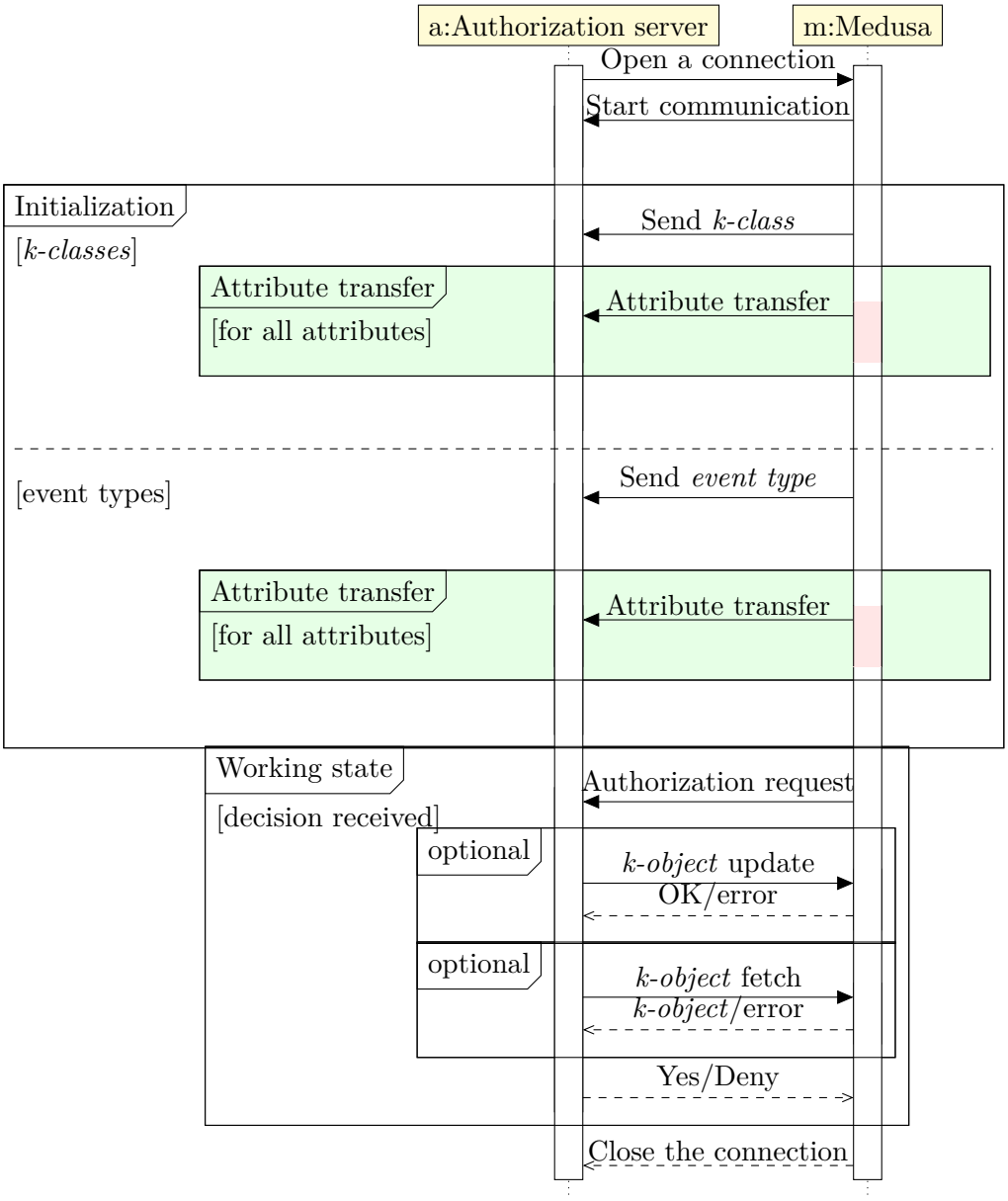


Figure 3: Communication protocol scheme, adapted from [3]

2.1.2 l4_decide

A process issues a decision request in the `l4_decide` function. If the issuing process is authorization server, kernel will automatically allow it. Otherwise, a deadlock would occur. Development version of Medusa will also automatically allow *gdb* process debugging the authorization server.

Information about event, object and subject are then stored in global variables and atomic variable `question_ready` is set as a signal to the `read()` function that a decision request is ready to be sent to the authorization server.

Then this thread waits on the `userspace_answer` waitqueue until authorization server doesn't wake it up which means that answer is ready. Process will find it in the `user_answer` global variable.

The whole process of filling out data structures, sending data to the authorization server and receiving decision answer executes under the `constable_mutex` lock. That means that during that time, no other process can make a decision request. Basically, other processes can't execute system calls that have to be processed by authorization server.

When Medusa was created, this wasn't a problem, since symmetric multiprocessing suppert was added to the Linux kernel in version 2.6 [12] and sequential processing was the only way to go.

But nowadays, with better operating system and hardware capabilities, this limitation causes degradation of performance. Better results could be reached with concurrent processing of decision requests. Adding support of concurrency is the main focus of this thesis.

2.1.3 Polling function

Authorization server uses `select()` to monitor the communicaton file descriptor to determine if it is ready to perform a read or a write operation.

To do that, it first has to register two sets of file descriptors (`fd_set` structures) for read and write operations. In this case, the authorization server always registers the read descriptor. Write descriptor is registered only when the authorization server has data ready for the kernel.

Based on the filled sets, a mask is created with monitored operations. For read, the value of the mask is `POLLIN | POLLRDNORM` and for write it's `POLLOUT | POLLWRNORM`.

After the `select()` function is called, the kernel will execute each device's internal *poll* function. In Medusa that's `user_poll()`. In this function, `wait_poll()` is called

to put the calling process on a waitqueue in case there's no data available. If there is an ongoing write operation, `user_poll()` will return *write mask*, which means that the authorization server can continue writing to the file descriptor. If there is an ongoing read operation (*global teleport* is not halted), `user_poll()` returns *read mask*, which means that the authorization server can continue reading data from the file descriptor. If a signal variable is indicating request for fetch or update operation, or the kernel wants to register a new object or issue a decision request, then `user_poll()` returns *read mask*. If none of these conditions apply, `user_poll()` returns *write mask* by default.

Based on the returned value, `select()` computes bitwise AND of requested and returned masks. If the result is zero, the authorization server is put to sleep. It can be woken up by calling `wake_up()` on the waitqueue that was specified earlier in the `wait_poll()` function. This is done when adding a *k-class* or *event type* or issuing a decision request.

When woken up, `user_poll()` is called once again to determine which action took place, and process is repeated until the computed mask is not zero. In that case, `select()` returns number of *ready* descriptors and sets them accordingly. Authorization server then executes *read* and *write* operations based on set values.

2.1.4 General communication

Communication between the kernel and the authorization server is as follows: When userspace process wants to make a system call, an LSM hook will be called before the actual system call is executed. Medusa has registered a callback function for this hook on the first layer. This function will call another function on the second layer that will prepare data structures for the decision operation. Then it calls function `med_decide` on the third layer that will check if an authorization server is available. If it is, it will call a decision function. In the current implementation of Medusa that is the `l4_decide` function. After initial checks, the `constable_mutex` is locked. This operation stops any process from issuing a decision request before the first request is completed and answered.

Kernel will prepare data structure that contain information about event and its object and subject. This information is then sent to the authorization server. Authorization server will decide about the operation based on the received data and send back decision answer code. Kernel will receive the answer and return it back to the third layer until it gets to the system call code where it will continue (in the case of allowed operation) or will be stopped (in the case of denied operation).

2.2 Changes

In this section we describe changes that were made to the LSM Medusa to support concurrency.

2.2.1 Proposed solution

The main goal when adding support of concurrency was to get rid of the `constable_mutex` lock, so multiple processes can request a decision simultaneously . This operation however brings new problem — there is only one teleport structure for sending data to the authorization server. If more than one process would be allowed to change it simultaneously, data would be inconsistent and authorization server would probably return error, or worse, stop working.

We proposed to create a queue that would store all the data structures (decision requests) as they come from various running processes until they are sent to the authorization server for a decision.

But decision requests aren't the only events where kernel needs to send data to the authorization server. Therefore this queue mechanism had to be implemented also for *k-class* and *event type* registrations, *fetch* and *update* requests and initial *hello* message when the authorization server is starting up. Details about queue implementation can be found in section 2.2.2.

2.2.2 Queue

Implementation of the queue is based on the basic doubly linked list defined in the `list.h` file as a standard part of the Linux kernel.

We define the head of the list as `tele_queue`. One item of the queue is represented by the `tele_item` structure (see table 1). Concurrent access to the queue is guarded by the `queue_sem` semaphore.

Now that we have defined the queue along with objects that will be stored in it, we have to think about where are we going to store these objects — specifically the list items and teleports contained within them. Since decision requests can come in any number (it depends on number of processes on the system making system calls) we can't use statically allocated memory since it may not be big enough for many requests simultaneosly and kernel would have to wait until some decision request would be finished and memory available again for a new decision. Only one option is possible — dynamic allocation during runtime.

Standard way to dynamically allocate memory of a specific size in the Linux kernel is by calling the `kmalloc()` function. But this function is unsuitable for our purpose,

Table 1: Definition of the `tele_item` structure

type	name	meaning
`teleport_insn_t*`	`tele`	pointer to a teleport instruction array
`struct list_head`	`list`	head of the list
`size_t`	`size`	size of the teleport instruction array
`void (*post)(struct kmem_cache*, void*)`	`post`	pointer to a function that deallocates the teleport instruction array

since there'll be many allocations needed — as many as number of system calls running and dynamic allocation is not cheap performance-wise. This means we have to use other allocation option that Linux kernel provides — the *slab layer* allocation.

Slab layer is based on *free lists*. These are linked lists of available objects that have been already allocated and are ready to be used. If a program needs a new structure, it takes one from the list. When it's done with it, it returns it to the list. These operations are much faster than allocating new objects every time program needs them.

Linux kernel provides a unified interface for working with *free lists*, so it can have more control over them — if not much memory is left, it can deallocate unused objects from these lists and provide free pages to a process that needs more memory. These structures are called *memory caches*.

Memory cache can only provide objects of a constant size specified when a new cache is created:

```
struct kmem_cache *kmem_cache_create(const char *name,
                                     size_t size,
                                     size_t align,
                                     slab_flags_t flags,
                                     void (*ctor)(void *))
```

Here's an explanation of parameters of this function [13]:

name string with the name of the cache

size size of the objects the cache should provide

align offset of the first object within a slab

flags flags specifying options for the cache

ctor pointer to a function called when new pages are added to the cache

When a memory cache is created, new objects can be allocated from it using the `kmem_cache_alloc()` function. When finished with the object, it can be returned back to the slab cache with `kmem_cache_free()` function.

In LSM Medusa we need to dynamically allocate objects with varying sizes. These are:

1. items of the queue (`tele_item`)
2. teleports (different sizes of `teleport_insn_t` structure arrays)
3. storage *k-objects* from *L2* (different sizes) (see section 2.2.4)

Since these objects have varying sizes we have two options:

1. Create one memory cache with objects of size of the biggest object needed
2. Create more memory caches with objects of varying sizes

First option is wasteful, because much of the allocated space won't be used, since currently the biggest *k-object* in LSM Medusa (*fuck k-object*) has 4152 bytes. This leaves us with the second option. We have to decide how many caches with what sizes we need. For that we've analyzed all sizes that need to be dynamically allocated in the current version of LSM Medusa. For queue and biggest teleport data structures, we need 40 and 144 bytes respectively. For *k-classes* we need 96, 312, 512, 544 and 4152 bytes.

Trivial solution would be to hardcode theses sizes to their respective memory caches and use them. But if a new *k-class* would be added or its definition changed, then these values would have to be changed as well — and that is not a good programming practice. Instead, we proposed a more general solution — memory caches would be created with sizes of powers of two based on the sizes of objects kernel would need. More about this allocation system can be found in section 2.2.3.

2.2.3 Memory cache allocation system

Memory caches are stored in a global array `med_cache_array`. This array is dynamically allocated for a given size using the `med_cache_create()` function. It takes one argument — length of the array to be allocated. Internally, it uses `kmalloc()` for allocation. It is usually called when the authorization server is starting up (in the `user_open()` function).

When `med_cache_array` is allocated, it is initialized to `NULL` values, thus no caches are initially created. New caches can be created using the `med_cache_register(size)` function. This function is called in two places — in the `user_open()` function, where sizes of the queue item and teleport are registered and in `l4_add_kclass()`, where sizes of *k-classes* are registered. Register function of new memory caches checks size of the memory cache array — if there's a request for a memory cache with an index that's not currently available, `realloc_med_cache_array()` function is called which uses `krealloc()` to realloc the array to a new size.

Memory caches are ordered in the array based on their object size growing exponentially. That means that at index `[0]`, there may be a cache of objects of 1 byte size if it's been registered before via `med_cache_register()`. At index `[1]`, there may be a cache of objects of 2 byte size and so on. Exponential growth of sizes was chosen so that the buffers are aligned in the memory.

When requesting new memory from the cache we use a technique known as *fat pointers* to store the index of memory cache that was used to request the object to make returning of the object easier. When requesting a new object, we store the index of the memory cache in the first byte of the returned memory and return pointer to the memory shifted by one byte.

New memory can be requested using two functions: either `med_cache_alloc_size()` or `med_cache_alloc_index()`. Former function takes a size in bytes as an argument while the latter takes an index in the memory cache array. In case of choosing the index, offset of the *fat pointer* has to be taken into account.

Returning the memory is done via `med_cache_free(void*)`. It takes one argument — pointer to the allocated memory. This function reads a byte before the memory area the argument points to — the index of the memory cache in memory cache array where this memory comes from. This makes working with memory caches easier, as there's no need to externally store number of cache the memory pointer came from. When the index is known, memory is freed (along with the index) using `kmem_cache_free()`. Fat pointers are illustrated in figure 4.

When authorization server is disconnected, all memory caches are destroyed and

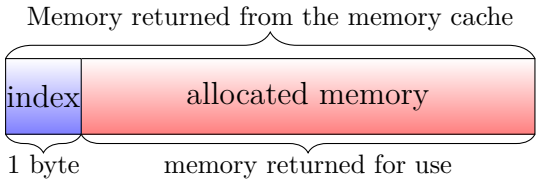


Figure 4: Fat pointer

memory cache array is freed with the `free_med_cache_array()`.

Since space can be wasted on objects just slightly larger than an exponent of two, it is adviced to design new *k-objects* with size slightly below an exponent of two (with the fat pointer size accounted) to better utilize usage of memory.

2.2.4 Storage objects

Before the concurrent changes, Medusa used static buffers to temporarily store *k-objects* fetched by the authorization server, one for each *k-class*. Information about fetched *k-object* would be passed to a *fetch* function on the second layer for a given type of *k-object*. This function would convert a system object to a *k-object*, copy it to a static buffer and return the address of this buffer back to the fourth layer. Afterwards, the fetched *k-object* would be sent to the authorization server. This approach was good for serialized fetches, but not for concurrent ones. Simple solution to this problem would be to serialize the fetch operation, but that would affect the performance and since we aim to maximize the concurrency we proposed a better solution.

Since we already have implemented a dynamic allocation system, we used it to allocate new buffers for *k-objects* in the `user_write()` function when the authorization server is requeting a *fetch* operation. Thanks to this change, we could simplify fetch functions on the second layer and remove static buffers for temporary storage of *k-objects*. Now multiple fetches can be executed concurrently, since each uses its own buffer for *k-object* storage. Allocated memory is returned to the kernel in the `post_write()` function set in the `post` field of the queue entry structure.

2.2.5 Event counters

Atomic counters were added to LSM Medusa to count number of operations waiting to be processed in the teleport queue. List of them along with their meaning follows:

questions Number of decision requests waiting in the teleport queue.

questions_waiting Number of decision requests sent to the authorization server with decision pending.

fetch_requests Number of fetch requests waiting in the teleport queue.

update_requests Number of update requests waiting in the teleport queue.

announce_ready Number of *k-objects* or *event types* waiting in the teleport queue to be registered.

These counters are incremented when a new teleport is added to the queue. Counts are used in the polling function to let the authorization server know if there's any data ready for reading.

Incrementing is an easy operation — we know which counter to increment based on the operation we're doing. Decrementing is more special, since we know that a teleport has been sent to the authorization server, but we have to read its content to know which counter to decrement. For this, we're using algorithm 1.

As can be seen from the algorithm, it depends on a specific data in the teleport instruction array characteristic for a given operation. If format of the data in the teleport were to change, counter decrementing would work incorrectly. Therefore function `decrement_counters()` that implements this algorithm should be thoroughly tested in the case of protocol change.

2.2.6 Meaning of global variables

Global variables are considered a bad practice in software engineering since they can be changed from any place in the code and that makes the debugging process harder. But in some cases, like in multithreaded programming, they are useful, since they can be used to interchange data between running threads. Locks that guard data against concurrent access are also defined as global variables.

Comprehensive list of all global variables in the `chardev.c` file can be found in appendix 3. Along with the name of the variable we state its purpose and functions[1] that use this variable.

Lock variables Lock variables are used to guard concurrent access to shared data resources. Full list can be found in table C.1.

Signal variables These variables are used to signal an event or a currently valid state. They are listed in the table C.2

Other variables In this section we list other important global variables that couldn't be assigned to any of the previous categories. They are listed in the table C.4.

[1] Allthese variables are used also in `user_open` and `user_release` functions, where they are initialized to their default values.

Algorithm 1 Decrementing event counters

```
if operation code in the second field is HALT then
  return
end if
if operation code of the third field is CUTNPASTE then
  decrement questions
  decrement questions_waiting
  return
end if
if operation code of the third field is PUTPtr then
  if type of the argument of the second field is MEDUSA_COMM_FETCH_ANSWER then
    decrement fetch_requests
    return
  end if
  if type of the argument of the second field is MEDUSA_COMM_UPDATE_ANSWER then
    decrement update_requests
    return
  end if
end if
if operation code of the third field is PUTKCLASS or PUTEVTYPE then
  decrement announce_ready
  return
end if
```

2.2.7 Decide

All decision requests from processes that require a decision from authorization server are prepared for sending in the `l4_decide` function.

The most important change is the removal of the `constable_mutex` lock. A new array of teleport instructions is created as a local array on the stack (it doesn't have to be created dynamically, since it is not needed after a decision request is decided).

Teleport instructions are loaded according to D.3. More information about instruction array creation can be found in section D.6.

After the teleport instruction array is filled and queue element is ready, we lock the lightswitch (see 2.2.11) and check if authorization server is still running. If not, we free the queue element data structure, unlock the lightswitch and return `MED_ERR`. Otherwise, the decision request ID is increased to be ready for the next decision. Queue element is added to the queue under the `queue_lock` semaphore. Question counter is increased along with the `queue_items` semaphore. Lightswitch is unlocked and process is put to sleep to be woken up by the `user_write()` function.

When the process is woken up, it compares its decision request ID with ID that has been answered. If it's not equal, it goes back to sleep. Otherwise, the presence of the authorization server is checked once again. If it's running, process will take the answer by setting it's local variable `retval`, release the `take_answer` semaphore and return the answer.

2.2.8 Read

Read is used to serve data from the kernel to the authorization server. In the new version of LSM Medusa, the `user_read()` function has been almost completely rewritten, except for the initial checks.

It works as follows. First, the lightswitch is locked. Initial checks consist of:

- check if the authorization server is running
- check if the authorization server is not already making a *read* call
- check if file position corresponds to the read offset
- check if buffer address is valid

After initial checks, the `user_read_lock` is locked. Then we check if any bytes are left to read from the previous teleport instruction array. If not, we get a new teleport from the queue using the `teleport_pop()` function.

Then we call the `teleport_cycle()` function (see D.5) and check the return value. In case of an error, we clean up global values, release locks and return number of succesfully transferred bytes.

Otherwise, we subtract number of read bytes from local and global counters, add them to the total sum of bytes read and check if all bytes have been read. If so, current teleport instruction array is deallocated using `teleport_put` function (it also does various cleanups of related global variables). If authorization server doesn't want any more data (based on the `count` counter), we can unlock the lightswitch and return number of transferred bytes.

Otherwise, we check if there's any teleport in the queue by calling `teleport_pop(1)`. By using argument `1` we make sure that this function call won't block waiting for new teleport if the queue is empty. In case of an empty queue we do the returning procedure.

Whole process can be found simplified in a form of a flow chart in figure 5.

Helper functions As mentioned in the previous section, `user_read()` uses two helper inline functions — `teleport_pop()` and `teleport_put()`. In this section we'll explore them in more detail.

teleport_pop This function is used to take one teleport from the queue. It can be used in two variations based on the used argument. When it's zero, this function will block until either a new teleport is added to the queue or the authorization server is detached. This is made possible by acquiring the `queue_items` *semaphore* with a timeout of 5 seconds. Before we acquire `queue_items`, the lightswitch is released, therefore allowing the authorization server to be *closed*.

When a timeout is encountered, the lightswitch is locked and presence of the authorization server is checked. If it's no longer running, `-EPIPE` is returned. Otherwise the lighswitch is unlocked and `queue_items` is acquired for another 5 seconds or until a new teleport is added to the queue.

If the argument to this function is non-zero, then this function is non-blocking and returns `1` if the queue is empty.

The rest of this function takes a teleport from the queue, sets global variable `processed_teleport` to point to the acquired teleport instruction array, sets counter `left_in_teleport` (based on the size that was determined during the creation of the teleport) and deletes the teleport from the queue.

Then, `teleport_reset` is called, which resets teleport data structures and prepares them for reading the instructions inside the teleport. Last thing done is decrementing counters, as described in 2.2.5.

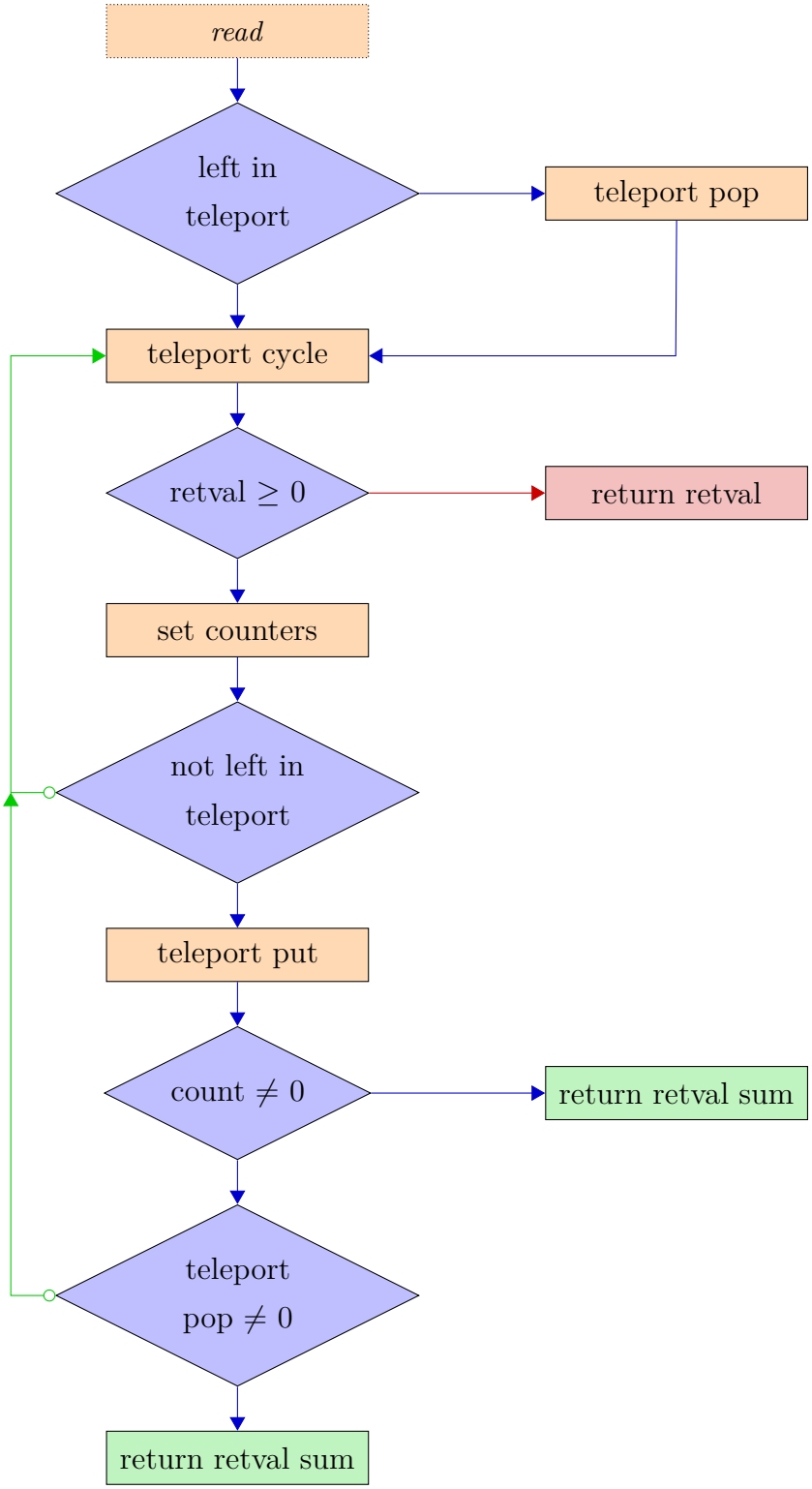


Figure 5: Simplified algorithm of the new `user_read()` function

teleport_put This function does cleaning operations after a teleport instruction array has been fully executed.

First, it calls `post` function that frees teleport instruction array data structure. Then it frees the queue element data structure and sets global variables pointing to these data structures (now invalid pointers) to *NULL*.

2.2.9 Write

Function `user_write()` also went through a major overhaul.

One of the first things that became evident with adding support for concurrency was that it couldn't work with non-atomic *writes*. The original authorization server, Constable, called *write* multiple times, sending parts of a complete message. The original `user_write()` was written to support this behaviour — this function internally used macro `GET_UPTO` to read bytes at a desired position. Global variable `recv_phase` was used to count bytes already copied to a local buffer. Thanks to this variable, the actual position in the received message was known.

Now, with many threads concurrently accessing just one variable for a buffer position wouldn't be enough. We would need as many variables as number of active *writes*. But this change still wouldn't be enough — the calling thread would have to be also somehow identified and that would require a change of the Medusa protocol. We wanted to avoid such big changes when a simpler solution exists — change implementation of authorization server to send messages atomically, one *write* meaning one message. This change would also remove global variables and simplify the function, since the complex macro `GET_UPTO` would also be removed.

Here is a description of the new version: First, the lightswitch is locked so a *close* operation can't occur during `user_write()`. Then, checks similar to the check in `user_read()` are done.

After checks, values from the buffer are read according to the Medusa protocol. First value with of 64 bits signifies type of the received message. This value can be one out of three types: authorization answer, fetch request or update request (defined as numeric constants). Any other value is considered a protocol error. Before this value is checked, `take_answer` semaphore is locked in case it is an authorization answer.

Authorization answer If the message is an authorization answer, that means that authorization server has decided about some event and it has sent a decision answer to the kernel. Two important values are read from the buffer — the answer and decision ID, so the answer can be assigned to a decision request. Decision ID is saved in a global

variable `recv_req_id`. The `userspace_answer` waitqueue is woken up. Every process on the waitqueue will compare its internal value with the global ID. The one with the identical value will continue as described in 2.2.8.

Fetch or update request In the case of a fetch or an update request, the `take_answer` semaphore is released. Next value in the protocol is pointer to *k-class*. Once the class is found, a buffer is allocated for a *k-object* of size retrieved from the *k-class*. After succesful allocation, *k-object* from the authorization server can be copied to the created buffer.

Update or fetch operation is carried out on the received *k-object*. Medusa then creates a new teleport instruction array where it will store the result of the operation. In the case of an update operation it is just a numeric value — result of the update operation. In the case of a fetch operation it is a full *k-object*[2]. Teleport instruction array is added to the queue in a standard way using the doubly linked list interface. Function releases the lightswitch and returns number of written bytes.

2.2.10 Open

Open starts with the registration of memory caches needed for correct function of Medusa. Memory cache allocation system is described in greater detail in 2.2.3. All global data structures are then resetted. This includes all semaphores and the global teleport state. Rest of the function remains unchanged.

2.2.11 Lightswitch

During first testing, we found out that *close* operation leaves Medusa in inconsistent state. Starting authorization server for a second time (after a manual shutdown) lead to various problems ranging from communication protocol errors to system freezes. Since Medusa uses dynamic memory allocation this also meant a presence of memory leaks.

We resolved this problem with a global lightswitch. Lightswitch is a synchronization pattern used to resolve classical synchronization problem of concurrent readers and writers [14]. The problem is that any number of readers may be in the critical section while writers must have exclusive access to it. In Medusa, readers are calls to *read* and (while it may sound strange) *write* operations. Writer is just one — the *close* operation. Thanks to the lighswitch, *close* can't occur during *read* or *write* and we can better track the state of Medusa and clean up inconsistent state.

Lightswitch implementation Lightswitch is implemented using a numeric counter and a semaphore that guards access to this counter. A lighswitch semaphore is also needed — it will grant concurrent access to readers and exclusive access to writers.

[2]Where this *k-object* is stored is described in section 2.2.4

When a lightswitch is locked by a writer, it will increment the counter and check it's value. If it's one, it is the first one in the critical section and it can lock the lightswitch semaphore (analogy of turning the light on). All these operations are made under a locked semaphore.

Unlocking is similar. If unlocking reader is the last one, it will release the lightswitch semaphore (turning the light off).

Priority When implementing a lightswitch, it's very important to consider number of readers and writers in the system. If many readers are present, ligtswitch may be always locked and writers may *starve*. In our case, that would mean that authorization server couldn't be shut down if many operations are running.

Solving this problem is simple with a turnstille. Turnstille is represented by a semaphore acquired and released at the start of lightswitch locking. A writer can then acquire this sempahore, and wait on the lightswitch semaphore until all readers leave their critical sections. When it acquires the ligtswitch semaphore, it can release the priority semaphore.

2.2.12 Close

Lightswitch doesn't permit entry to this function until all read and write calls are finished[3]. Unregistration of *k-objects* and *event types* is left unchanged from the previous version.

Afterwards, all threads waiting on the `userspace_answer` waitqueue are woken up with an error answer value. Then, all atomic state variables are set to their default state. The queue is emptied, all teleport instruction arrays are deallocated. Lastly, cache array is deallocated and lightswitch is released. If any *read* or *write* operations were waiting on the lightswitch, they will return an error.

2.3 Synchronization model

The new concurrent design of Medusa has been modelled using Petri nets to detect deadlocks and check correctness of the model. It is shown in figure 6.

Using Petri modelling software *Pipe*, we checked the model for deadlocks. Tests didn't detect any deadlocks.

2.4 Shutdown deadlock

After implementing the memory cache system, we enabled the symmetric multiprocessing in the kernel configuration to test if Medusa would be performing correctly with all the

[3]But it is a lightswitch with a priority, so once the *close* is started, no other threads can execute *read* or *write*

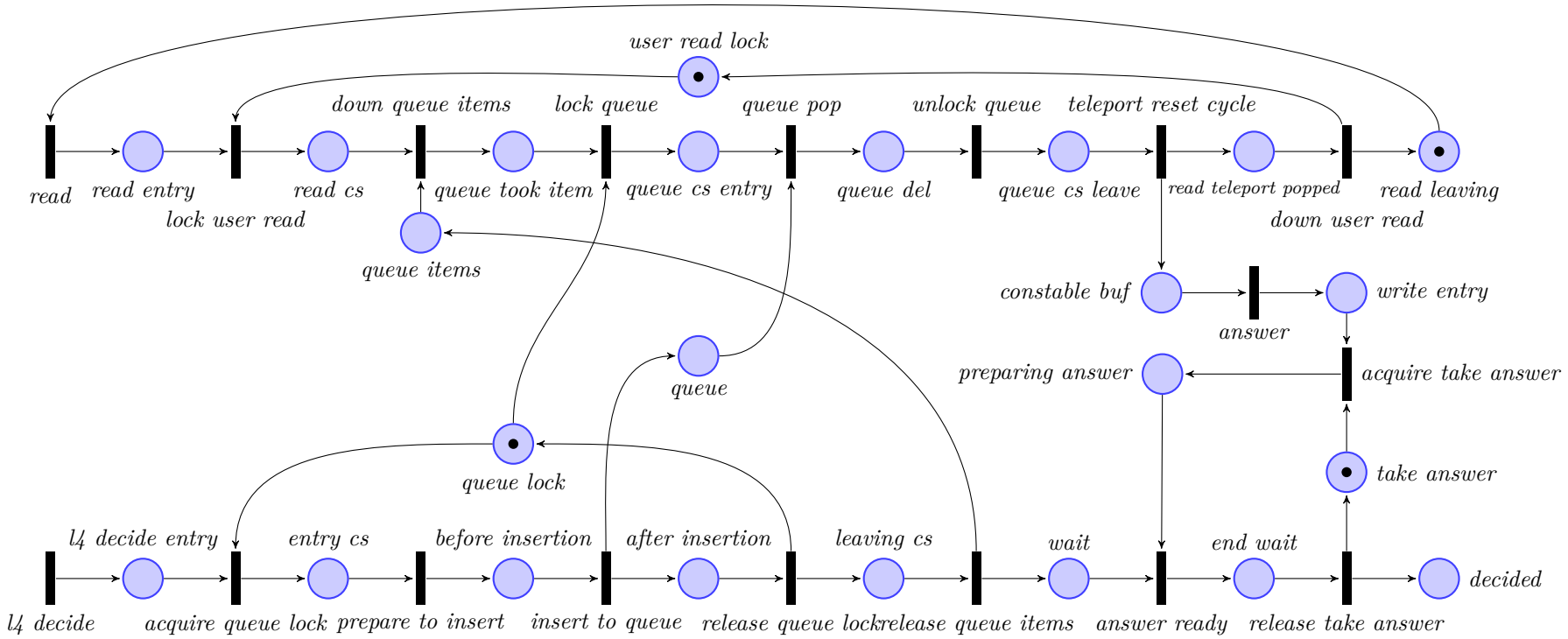


Figure 6: Petri net representation of concurrent synchronization model

new concurrent features. Initial tests with *mYstable* were positive. No deadlocks were present and authorization server was responding correctly. However, when we tried to shutdown the system, a problem occured. System would freeeze during the shutdown procedure.

We repeated the shutdown without turning the authorization server on. That was succesful. With that information, we concluded that the problem originates from Medusa.

We repeated shutdowns (after running the authorization server and turning it off) a few more times and we noticed that the freeze would occur in different parts of shutdown procedure — sometimes the desktop was still visible, sometimes the screen was black and other times the kernel message buffer was shown. One lucky time, the lockdep feature of the kernel stayed active and shown the locked lock and the kernel call trace (fig. 7).

```
[  224.032520] R13: ffffc9000037fa20 R14: 0000000000000000 R15: ffffffff81a615c0
[  224.032523] FS:  0000000000000000(0000) GS:ffff88007fd00000(0000) knlGS:0000000000000000
[  224.032525] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[  224.032529] CR2: 00007f22c5a8f160 CR3: 0000000001a0a000 CR4: 00000000000406e0
[  224.032531] Call Trace:
[  224.032537]  queued_read_lock+0x19/0x1b
[  224.032548]  _raw_read_lock_irq+0xa/0xc
[  224.032555]  process_unmonitor+0x14/0xc0
[  224.032558]  med_decide+0xb8/0xe1
[  224.032562]  process_kobj_validate_task+0xf3/0x106
[  224.032566]  medusa_sendsig+0x1bc/0x1f1
[  224.032573]  ? __free_one_page+0x91/0x1b1
[  224.032577]  ? __mod_zone_page_state+0x3c/0x44
[  224.032580]  ? __free_one_page+0x91/0x1b1
[  224.032583]  ? free_pcppages_bulk+0xc6/0x148
[  224.032587]  ? free_unref_page_commit.isra.100+0x85/0x8d
[  224.032590]  ? medusa_l1_inode_free_security+0x26/0x30
[  224.032594]  ? rcu_is_watching+0x9/0xe
[  224.032598]  medusa_l1_task_kill+0x11/0x30
[  224.032601]  security_task_kill+0x38/0x4e
[  224.032605]  check_kill_permission+0x74/0xf6
[  224.032607]  group_send_sig_info+0x16/0x37
[  224.032610]  do_exit+0x60f/0x8e7
[  224.032613]  do_group_exit+0x98/0x98
[  224.032616]  SyS_exit_group+0xf/0xf
[  224.032619]  entry_SYSCALL_64_fastpath+0x19/0x72
[  224.032621] RIP: 0033:0x7f22c9f30a38
[  224.032624] RSP: 002b:00007ffcf84b1508 EFLAGS: 00000246 ORIG_RAX: 00000000000000e7
[  224.032626] RAX: ffffffffffffffda RBX: 00005610b00322f0 RCX: 00007f22c9f30a38
[  224.032628] RDX: 0000000000000000 RSI: 000000000000003c RDI: 0000000000000000
[  224.032630] RBP: 00007ffcf84b1500 R08: 00000000000000e7 R09: fffffffffffffff50
[  224.032632] R10: 00007f22c5a8f150 R11: 0000000000000246 R12: 00007f22c5a8ed90
[  224.032635] R13: 00007ffcf84b1478 R14: 00007f22ca65a040 R15: 0000000000000000
[  224.032637] Code: 38 48 89 fb f0 81 2f 00 02 00 00 ba 01 00 00 00 48 8d 7f 04 f0 0f b1 53 04 85 c
0 74 07 89 c6 e8 6b ec ff ff f0 81 03 00 02 00 00 <8b> 03 84 c0 74 04 f3 90 eb f6 c6 43 04 00 5b 5d
c3 55 31 c0 ba
-
```

Figure 7: Lockdep calltrace in the kernel display message showing a deadlock occured during shutdown

Deadlock occured in the `process_unmonitor()` function that's used to reset virtual spaces for a given *k-nobject* It's called when the authorization server is not running.

This function locks the `tasklist_lock` for reading. It is used to access the task list

in the linux kernel that contains all running processes. It is a readers-writers lock. This type of lock allows concurrent read operations but only one write operation simultaneously. Therefore this lock had to be locked for write in the same thread before we entered the `unmonitor_process()` function and that caused the observed deadlock.

2.4.1 Debugging deadlocks

Debugging deadlocks in Linux kernel can be quite hard. However, there are techniques that may provide good results.

If you run Linux with spinlock and rw-lock debugging (`CONFIG_DEBUG_SPINLOCK`) compiled in the kernel, it is possible to determine owner of the lock and number of the processor that locked the lock. These are stored in the `owner` and `owner_cpu` structure fields of `raw_spinlock_t` structure, respectively. First one, `owner` contains a task structure of the process that currently holds the lock for writing (in the case of *rwlock*).

We used `read_trylock()` to detect when the lock is contended and compare processor number with currently executing processor. Our original plan was to `read_lock()` the lock only when processor numbers mismatch (meaning that we should wait until the write operation on the other processor finishes). This plan was however incorrect, as it is shown in the next section.

2.4.2 Solution with an RCU lock

Closer look at the captured call trace reveals two functions — `group_send_sig_info()` that calls `check_kill_permission()`, which calls LSM security hook. Important thing is, that `group_send_sig_info()` calls `check_kill_permission()` under an RCU read lock.

RCU is a synchronization mechanism that achieves higher performance than traditional *rwlocks*, since it allows readers and writers access shared data structures concurrently by keeping several versions of them as they are modified by writer threads.

Documentation comment of the `rcu_read_lock()` states that it is illegal to block inside a RCU read-side critical section. That means we can't use `read_lock_irq()` (or any type of lock that would block the thread) in the `process_unmonitor()` function.

Let's analyze operations guarded by the `tasklist_lock` in the `process_unmonitor()` function. First, `pid_task()` function is used to get the `task_struct` of the process by the *PID*. This function internally uses `rcu_dereference_check` for finding the `task_struct` in the task list. This means that we don't have to lock this call, since it's already in a RCU read-side critical section of the `group_send_sig_info()` function. But this hook can be called from various contexts. It doesn't necessarily have to be called from `group_send_sig_info()`, therefore it may not be running inside an RCU read-

side critical section. Solution is simple — since read-side critical sections can be recursive, we replace `read_lock_irq()` with `rcu_read_lock()` and `read_unlock_irq()` with `rcu_read_unlock()`.

There are other places in LSM Medusa that use read-locking in a similar way, guarding the `pid_task()` function. We replace them with *RCU* lock accordingly.

2.5 Unsolvable problems

During development, we encountered another peculiar problem. When authorization server was started, it would freeze until the kernel would detach it. However, next runs of authorization server would be succesful with no problems.

First suspicion fell on object initialization, since the problem occured only on the first run of the authorization server. All variables in `chardev.c` were checked, but no problem was found.

With source of the problem unknown, we decided to try Medusa on a fresh installation of Debian to rule out environmental causes. Test on a newest Debian version was successful, the problem was not occuring and authorization server worked on all runs.

Next step was to identify which distribution component was causing the deadlock in the older version of Debian. We usually start *mYstable* using the *lxterminal* terminal emulator. As a test, we tried to run the authorization server in *XTerm*, another terminal emulator. In this terminal, *mYstable* ran without any problems.

To better understand the problem, we ran *strace* on both versions of *lxterminal*. Linux tool *strace* is used for tracing system calls that userspace program makes along with system call arguments, return value and other useful information. With *strace* results, we found out that the deadlock is occuring after `open` system call that creates a new file in the `/tmp` directory. Newer version of *lxterminal* wasn't making that system call and so that explained why it didn't have the deadlock.

The explanation for deadlock in the older version of *lxterminal* follows. *Lxterminal* called `open` to create a new file. This was captured by *Medusa* and sent to the authorization server as a decision request. But authorization server was not yet initialized and printing out information to the terminal. The terminal couldn't print out data from the authorization server because it was waiting for a decision from the authorization server that was unitialized — a deadlock. We confirmed this explanation by running the authorization server with output redirected to a file. No deadlock was observed. This problem can't be fixed in Medusa, so we recommend running the authorization server *mYstable* on a terminal that doesn't evocate this behavior or disabling the output of *mYstable*. In

authorization server *Constable* this problem was not observed.

3 Performance measurement

In this section, we will explain how the concurrent changes reflected in the performance of Medusa.

3.1 Methodology

To test the speed of the concurrent Medusa implementation, we have to call system calls that will be authorized by the authorization server. For least biased results, it's best to use a system call that doesn't access any I/O. We chose to use the *kill* system call for that reason.

Testing consists of running a special application, that creates P pairs (parent–child) of processes. Creation of these processes is timed using the `times()` function. After all processes have been created, parent process out of each pair receives a signal. This triggers a ping–pong reaction. Parent from a pair sends a signal to its child. When the child receives the signal, it sends it back to the parent. This is repeated S times.

When all processes are finished, a final time measurement is taken and total time length is computed and printed to the screen.

Medusa was compiled with disabled kernel message output in two versions — one with the support of concurrency and one without. The only difference between them was the `chardev.c` file. Kernel version was 4.15.0-rc8.

Testing application was executed with two configurations: 1. $N = 10$, $S = 100$ 2. $N = 100$, $S = 100$ for both versions of Medusa. We also performed a control run without the authorization server to show performance without Medusa. All tests were run three times and an average of time lengths was computed.

Testing included both authorization servers. *Constable* was tested just with the serial Medusa, since it hasn't been updated to work with the concurrent version of Medusa. *mYstable* was tested with both versions of Medusa. Both authorization servers had disabled the screen output. Testing was done on runlevel 2 in text mode with two terminals opened — one for the authorization server and second one for the testing application. Testing was performed on a VirtualBox virtual machine with two CPU cores and 2048 MB of memory. Distribution was Debian Buster.

3.2 Results

Testing with values $N = 10$, $S = 100$ is shown in figure 8. As can be seen in the figure, *Constable* is considerably faster than *mYstable*. Testing shows that concurrent Medusa was 16 % faster than serial Medusa with *mYstable*. Averaged values for total time length

had small standard deviantion (less than 1 %). Length for process creation was too small to be detected by `times()` in case of control and *Constable* measurements.

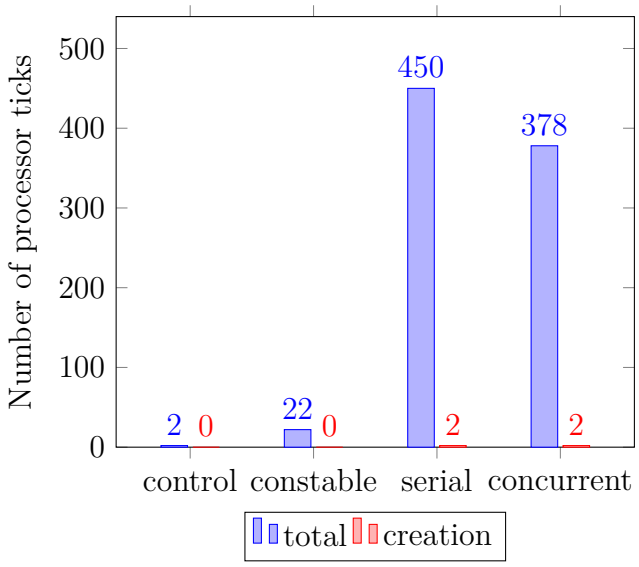


Figure 8: Results of test for 10 processess and 100 signals

Testing with values $N = 100$, $S = 100$ is shown in figure 9. *Constable* shows results similar to the previous test parameters. Concurrent testing of *mYstable*, however, shows almost 33 % increase of time than the serial testing. This worse performance can be explained by the way the kernel delivers an authorization answer to a waiting process. As described in section 2.2.7, when a new decision answer is received by the kernel, all processes waiting for an answer are woken up and they have to compare their ID with the received one. With small number of processes, this isn't a serious drawback. But when the number of processes increases, the constant scheduling of all processes causes a severe reduction of performance. This is one disadvantage that needs to be addressed in the future version of Medusa. One solution could be to use a hash map to quickly wake up the right process with no uneccessary scheduling.

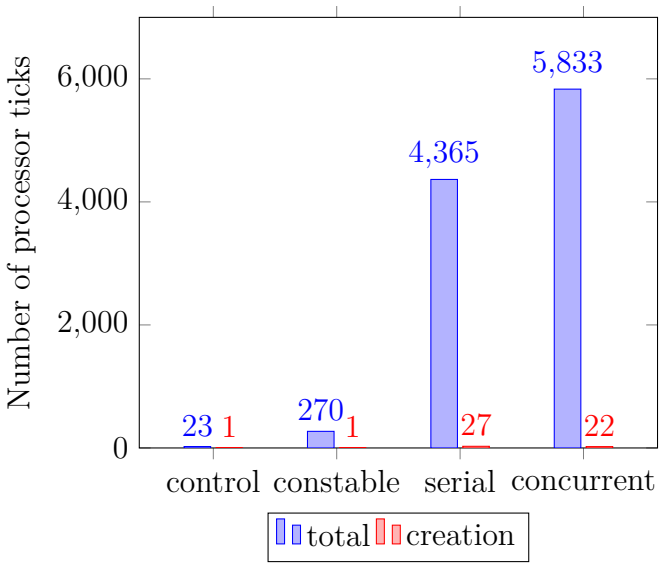


Figure 9: Results of test for 100 processess and 100 signals

Conclusion

This thesis described the implementation of concurrency in the Medusa LSM. In the first chapter, the LSM framework was explained. This framework allows simple implementation of security modules in the Linux operating system. First chapter also presented how Medusa works in the kernel and its authorization servers. Currently, two implementations of authorization servers exists — *Constable*, written in C and *mYstable*, written in Python used for development purposes. In the second chapter, we presented the communication layer. This is the part of Medusa that was originally serialized and needed to be implemented concurrently. We implemented a queue, that holds the authorization requests. This allows processes to issue decision requests concurrently and they are sequentially sent to the authorization server. We explained the allocation mechanism used to add elements to the queue. Lastly, we described issues encountered during the development and how we handled them. In the third chapter, we presented the performance analysis of the new concurrent system. With small number of processes, Medusa performs superiorly to the old serial version. However, as the number of processes increases, the scheduling problems of the workqueue become apparent and performance is worse than the serial version. We advised a solution with hash maps to be applied in the next version of Medusa.

Resumé

Táto práca sa zaoberá pridaním podpory konkurencie do Medusy. Medusa je bezpečnostný modul pre operačný systém Linux, ktorý rozširuje štandardné možnosti zabezpečenia. Medusa bola pôvodne vyvinutá na FEI STU v rokoch 1997-2000. V posledných rokoch bol jej vývoj obnovený.

Medusa je integrovaná do operačného systému Linux pomocou LSM. LSM je framework, ktorý umožňuje implementáciu bezpečnostných modulov. Vytvára im priestor na uloženie dát v štruktúrach systémových objektov a poskytuje registráciu obslužných funkcií („hookov“), ktoré sú zavolané, keď užívateľské procesy pristupujú k systémovým objektom. LSM umožňuje operácie zamietnuť v prípade, že operácia bola predtým povolená štandardnými UNIX právami. Nevýhodou LSM je, že bezpečnostným modulom neumožňuje operáciu povoliť, aj keď bola štandardnými UNIX právami zamietnutá. Bezpečnostné moduly, ktoré chcú také operácie vykonávať, musia vykonať väčší zásah do jadra. Nevýhodou takého riešenia je väčšia zložitosť pri jeho udržiavaní.

Medusa sa skladá z dvoch častí:

1. Modul v jadre, ktorý registruje obslužné funkcie.

2. Autorizačný server v užívateľskom priestore, ktorý vykonáva rozhodovanie a prideľuje systémové entity do *virtuálnych svetov*.

Medusa využíva abstrakciu *virtuálnych svetov*, do ktorých sú rozdelené systémové entity. Ak sa subjekt (vykonávateľ, napr. proces) a objekt (napr. súbor) operácie nachádzajú v tom istom *virtuálnom svete*, operácia je povolená, ak o ňu nemá záujem autorizačný server. Autorizačný server môže nastaviť typy operácií, o ktoré má záujem. Také operácie sú potom preposlané autorizačnému serveru na rozhodnutie, ak daná operácia prešla kontrolou virtuálnych svetov.

Jeden autorizačný server umožňuje riadenie bezpečnosti na viacerých zariadeniach prepojených sieťou. Takúto možnosť iné bezpečnostné moduly neposkytujú. Ďalšou výhodou je možnosť implementácie akéhokoľvek bezpečnostného modelu.

Jadro systému a autorizačný server komunikujú buď prostredníctvom znakového zariadenia, alebo pomocou sieťového spojenia. Keďže Medusa bola vyvíjaná na prelome tisícročí a jadro Linuxu nepodporovalo *symetrický multiprocesing*, aj komunikácia s autorizačným serverom bola navrhnutá sériovo. V súčasnosti sú už viacjadrové procesory štandardom a operačný systém Linux podporuje *symetrický multiprocesing* od verzie 2.6 [12]. Podpora konkurencie je preto ďalším krokom vo vývoji Medusy.

Medusa má vrstvovú architektúru pozostávajúcu z piatich vrstiev (0-4). Posledná, štvrtá vrstva slúži na komunikáciu s autorizačným serverom. Táto práca sa venuje najmä tejto vrstve.

Definícia znakového zariadenia sa nachádza v súbore `chardev.c`. V súbore sa nachádzajú funkcie, ktoré definujú operácie so znakovým zariadením — *open*, *close*, *read* a *write*. Ďalej obsahuje dôležitú funkciu `l4_decide()`, ktorá obsluhuje rozhodovanie o udalostiach.

Ďalšou súčasťou štvrtej vrstvy je *teleport*. Teleport je rodina dátových štruktúr, ktoré umožňujú jadru odosielať štruktúrované údaje s rôznymi typmi a veľkosťami. Pred odoslaním dát naplní jadro pole s inštrukciami, ktoré potom teleport prečíta a odošle autorizačnému serveru v správnom tvare.

V pôvodnej verzii Medusy bolo možné spracovanie len jednej udalosti súčasne. Ďalšie udalosti počas rozhodovania nemohli prebiehať. V takejto situácii systém nemohol efektívne využívať viacjadrové procesory. Navrhované riešenie spočívalo v odstránení zámku `constable_mutex`. Avšak také jednoduché riešenie nepostačuje — Medusa obsahuje len jedno statické pole pre inštrukcie teleportu. Preto bolo potrebné vytvoriť mechanizmus, ktorý umožní dočasné uloženie polí inštrukcií pre teleport, ktoré budú postupne odosielané autorizačnému serveru na rozhodnutie. Použitým mechanizmom bola fronta implementovaná pomocou štandardného zreťazeného zoznamu dostupného v jadre Linuxu. Jedna položka tejto fronty obsahuje: smerník na pole s inštrukciami pre teleport, *hlavu* zoznamu, veľkosť dát v poli a smerník na funkciu, ktorá slúži na uvoľnenie pamäte poľa s inštrukciami.

Fronta sa použije vždy, keď je potrebné odoslať dáta autorizačnému serveru. Toto môže nastať pri nasledovných správach: žiadosť o rozhodnutie, registrácia *k-tried* a typov udalostí[4], odpoveď na žiadosť o objekt (*fetch*), výsledok žiadosti o aktualizáciu objektu (*update*).

Ďalej bolo potrebné vyriešiť, kde sa budú položky fronty ukladať. Statická alokácia nebola pre tento prípad výhodná, keďže sa nedá dopredu povedať, koľko polí inštrukcií bude potrebných. Pri príliš malej hodnote budú musieť operácie čakať, kým sa uvoľnia miesta pre polia s inštrukciami a pri príliš veľkej hodnote bude zbytočne využívaná pamäť. Preto sme sa rozhodli použiť dynamickú alokáciu pamäte. Pamäť v jadre sa dá alokovať pomocou funkcie `kmalloc()`. Takýto spôsob alokácie je ale pomalý a keďže sa alokácia

[4] *K-triedy* a *typy udalostí* sú abstrakcie systémových objektov, ktoré používa Medusa pri komunikácii s autorizačným serverom. Definície štruktúr týchto abstrakcií sa posielajú pri zapnutí autorizačného servera. Toto umožní komnunikáciu aj so vzdialeným autorizačným serverom, ktorý nemá prístup k pôvodným štruktúram systémových objektov.

používa pri každej udalosti rozhodovanej autorizačným serverom, rozhodli sme sa použiť alokáciu pamäte pomocou *pamäťovej cache*.

Pamäťové cache sa nachádzajú na *slab* vrstve pamäťového alokátora a sú založené na voľných zoznamoch. Tieto zoznamy obsahujú vopred alokované pamäťové oblasti jednej veľkosti, ktoré môže systém okamžite použiť bez nutnosti čakať na alokáciu. Keďže sú v Meduse potrebné rôzne veľkosti inštrukcií teleportov a *k-tried*, pri inicializácii autorizačného servera a registrácii *k-tried* sú vytvorené *pamäťové cache* s veľkosťami, ktoré budú potrebné.

Po implementácii fronty a alokácii jej položiek bolo možné pristúpiť k odstráneniu zámku `constable_mutex` a ďalším potrebným úpravám funkcií v jadre. Ďalej popíšeme najdôležitejšie úpravy.

Hlavná rozhodovacia funkcia, ktorá vytvára informácie o udalostiach pre autorizačný server je `l4_decide()`. Táto funkcia vytvorí nové pole inštrukcií teleportu s informáciami o subjekte, objekte a typu prístupu operácie. Pole je potom pridané ako položka do fronty, ktorú môže autorizačný server prečítať pomocou operácie *read*. Po tom, ako je položka pridaná do fronty, je proces umiestnený do *fronty čakania*, na ktorej bude čakať, až kým nepríde odpoveď od autorizačného servera. Autorizačný server ju odošle do jadra pomocou funkcie *write*. Keď je odpoveď prijatá, všetky procesy na *fronte čakania* sú zobudené a každý si skontroluje svoje identifikačné číslo. Proces s rovnakým identifikačným číslom si prevezme odpoveď. Funkcia ju potom vráti nižším vrstvám, až prejde k systémovému volaniu, ktoré udalosť vyvolalo.

V priebehu vývoja sme sa často stretávali s problémami nekonzistentného stavu, ktorý vznikol z dôvodu predčasného ukončenia autorizačného servera. Preto sme pridali do funkcie `user_release()` *lightswitch*, ktorý povolí ukončenie autorizačného servera len vtedy, keď boli ukončené všetky *read* a *write* operácie.

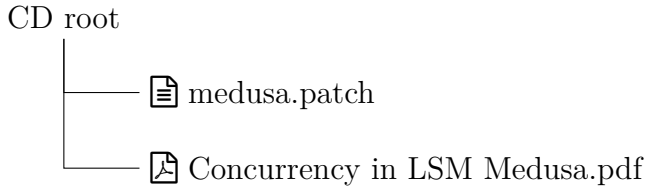
Po dokončení implementácie podpory konkurencie spolu s opravami nájdených chýb sme vykonali testovanie zlepšenia výkonu pomocou skúšobného programu, ktorý posielal signály medzi veľkým počtom procesov. Pre 10 procesov bolo zlepšenie výkonu o 16 %, ale pre 100 procesov nastalo naopak zhoršenie výkonu o 33 %. Toto zistenie prisudzujeme spôsobu odovzdania odpovede od autorizačného servera. Pri odpovedi musia byť všetky čakajúce procesy naplánované na spustenie, aby skontrolovali podmienku identifikačného čísla prijatej odpovede. Toto spôsobuje veľké spomalenie pri väčšom počte procesov. Do budúcna preto navrhujeme použiť iný spôsob predania odpovede, napríklad pomocou *hashmapy* so zobudením konkrétneho, jedného procesu.

Bibliography

1. WRIGHT, C., COWAN, C., MORRIS, J., SMALLEY, S. and KROAH-HARTMAN, G. *Linux Security Modules: General Security Support for the Linux Kernel*. 2002. Available also from: `https://www.usenix.org/legacy/event/sec02/full_papers/wright/wright_html/index.html`.
2. PIKULA, Milan and ZELEM, Marek. *Medusa DS9 Security System* [online]. 2004 [visited on 2018-04-18]. Available from: `http://medusa.terminus.sk/`.
3. KÁČER, Ján. *Medúza DS9*. Bratislava: FEI STU, 2014.
4. KYSEĽ, M., MAJZEL, M., KRAJČÍR, M., PROCHÁZKA, M. and KÚKEL, M. *Medusa Voyager*. Bratislava: FEI STU, 2015.
5. PLOSZEK, Roderik. *Medusa Testing Environment*. Bratislava: FEI STU, 2016.
6. MIHÁLIK, Viliam. *Implementácia ďalších systémových volaní do Medusy*. Bratislava: FEI STU, 2016.
7. NAGY, Zdenko Ladislav. *Sledovanie aktivity procesu pomocou Constabla*. Bratislava: FEI STU, 2016.
8. MIHÁLIK, V., PLOSZEK, R., SMOLÁR, M., SMOLEŇ, M. and SÝS, P. *Medusa*. Bratislava: FEI STU, 2017.
9. BOOTLIN. *Linux Kernel and Driver Development Training* [online]. 2018 [visited on 2018-04-07]. Available from: `http://bootlin.com/doc/training/linux-kernel`.
10. ZELEM, Marek. *Integrácia rôznych bezpečnostných politík do OS LINUX*. Bratislava: FEI STU, 2001.
11. PIKULA, Milan. *Distribuovaný systém na zvýšenie bezpečnosti heterogénnej počítačovej siete*. Bratislava: FEI STU, 2002.
12. LINDSLEY, Rick. *Kernel Korner - What's New in the 2.6 Scheduler?* 2004. Available also from: `http://www.linuxjournal.com/article/7178`.
13. LOVE, Robert. *Linux Kernel Development*. Addison-Wesley Professional, 2010.
14. DOWNEY, Allen B. *The Little Book of Semaphores*. 2016.

Appendix

A Electronic medium structure



File `medusa.patch` contains Medusa source patch that can be applied on top of the Linux source code. For development, we used version `v4.15-rc8`. Newest version of Medusa can be found of git: `https://github.com/Medusa-Team/linux-medusa`

B Debugging Guide

Finding errors in programs can be difficult, especially when programming a security module for operating system kernel. This appendix shows how to debug Linux kernel on a Windows operating system using another virtual machine with a Linux on it. Virtual machines will be connected by a serial port provided by VirtualBox using a named pipe on a Windows host system.

Debugging on a Linux system is easier — the host system is used as a debugger and there's no need for a second virtual debugging machine.

B.1 Prerequisites

- Physical computer with Windows operating system. This tutorial uses Windows 10, but any newer version of Windows (7, 8, 8.1) should suffice. We will refer to this computer as the *host system*.
- Virtual machine with a Linux operating system. We will refer to this computer as the *debugged system*. This is the system with *Medusa*.
- Virtual machine with a Linux operating system. We will refer to this machine as the *debugging system*. It doesn't have to be the same distribution as the *debugged system*, but it has to contain the source files of the *debugged system* kernel with *Medusa*.

  The `build.sh` script bundled with Medusa already contains an `rsync` command to copy source code from the *debugged system* to the *debugging system*. For it to work, a destination on the *debugged system* has to be specified in the `.dest` file in the following format: *`user_name`*`@`*`IP_address`*`:`*`path`*.

  Two virtual machines have to be on the same network. This can be achieved using the *NAT Network*, *Bridged Adapter*, *Internal Network* or *Host-only Adapter* setting in the virtual machine settings.

This appendix uses *VirtualBox* as a virtualization tool. However, information in this appendix applies to any other virtualization software.

B.2 Guide

1. Start VirtualBox and go to the settings of the *debugged system*. Go to the section *Serial ports*. Enable the first port, leave it as COM1. As a port mode, choose *Host pipe*. Check the *Connect to existing pipe/socket* checkbox. In the Path/Address field

write `\\.\pipe\debug` (NOTE: `debug` can be any other string, but make sure to use the same string in the next step.)

2. Do the exact same setup for the *debugging system*, but leave the checkbox unchecked.
3. Start up the *debugging system*. VirtualBox will create a new named pipe for the serial port.
4. On the *debugging system* open the terminal and execute following commands:

   ```
   sudo su
   stty -F /dev/ttyS0 raw -echo
   socat -d -d /dev/ttyS0 pty &
   ```

   Take note of a virtual terminal returned by the last command.

   EXPLANATION: Second command sets the serial port to a *raw* mode. That means it won't be buffered and that's very *important* for debugging to work. Third command creates a virtual terminal that allows duplex communication for gdb on a serial port.
5. Open a new terminal tab and change directory to where kernel source files are located. Run command `sudo gdb vmlinux`. *Debugging system* is now ready.
6. Start up the *debugged system*. From the *grub* menu choose *Advanced options for Debian GNU/Linux*. You'll be presented with a list of available kernels on the system. Choose the correct one (usually the first one) and press `e`. Find line that begins with `linux /boot/vmlinuz-4...` and add this command to the end of that line: `kgdboc=ttyS0,115200 kgdbwait` Press `Ctrl+X` or `F10` to boot. Kernel will now wait for the debugger to connect.
7. On the *debugger system* in the gdb console type: `target remote /dev/pty/X` Where `X` is number of the virtual terminal returned by the `socat` command in step `??`. Debugger will now connect to the machine and you can type `c` and press `Enter` (or add some breakpoints before that) to continue booting the system.

B.3 Debugging through `dmesg` over serial connection

Another way of debugging is to print messages and variable states from various important parts of the module code. This can be faster way of debugging, since programmer can spot the problem just by looking at the output without slow stepping through the code.

But in the most critical states of the operating system—at boot, before deadlock or shutdown—the `dmesg` command is usually unavailable. And even if it was already running before a deadlock, the screen usually freezes and the most important messages won't be shown on screen.

That's why we decided to send `dmesg` output through the serial connection into the *debugging machine* to capture all debugging messages that *Medusa* outputs.

B.3.1 Setting up and connecting

1. Create a second serial connection as described in step 1 of section B.2. Don't forget to name the new pipe differently than the first named pipe.
2. Start virtual machines in correct order (first the *debugging system* and then *debugged system*).
3. On the *debugging system* execute `screen /dev/ttyS1` in a terminal as root user. This creates a `screen` session to the *debugged system* over the serial port.
4. On the *debugged system* execute command `getty -L 115200 ttyS1 vt102` as root.
5. Now you should see a login terminal screen on the *debugging system*. Sign in as root and execute `dmesg -w`. You can now see kernel messages from the *debugged system* on the *debugging system*.

This can be used for deadlock situations, but this procedure is insufficient for boot and shutdown messages. For a better approach, see sectio B.3.2.

B.3.2 Grub settings

To create a connection and redirect kernel message buffer at boot, configuration of *grub* has to be modified.

1. In the *debugged system*, open up file `/etc/defaults/grub` as root.
2. Find the following configuration options and set them accordingly:
   ```
   GRUB_CMDLINE_LINUX="console=ttyS1,115200n8 ignore_loglevel"
   GRUB_TERMINAL=console
   GRUB_SERIAL_COMMAND="serial --speed=115200 --unit=0 --word=8
                        --parity=no --stop=1"
   ```
   Save the file, run command `# update-grub2` and turn off the virtual machine.
3. Start up the *debugging system* and execute the same command as in step 4 in section B.3.1. Then turn on the *debugged system*. Messages will be redirected to the *debugging system*.

C Global variables

Table C.1: Lock variables

Name	Meaning	Used in function

`take_answer`	Guards the answer from the authorization server until it's collected by the process that requested a decision.	`l4_decide()`
		`user_write()`
		`user_open()`

`user_read_lock`	Guards the main teleport used to communicate with the authorization server.	`user_read()` `user_open()`

`queue_items`	Semaphore that counts number of teleports in the queue. Can be used to make `user_read()` blocking.	`l4_add_kclass()`
		`l4_add_evtype()`
		`l4_decide()`
		`teleport_pop()`
		`user_write()`
		`user_open()`

`queue_lock`	Guards access to the teleport queue.	`l4_add_kclass()`
		`l4_add_evtype()`
		`l4_decide()`
		`teleport_pop()`
		`user_write()`
		`user_open()`
		`user_release()`

`ls_switch`	Used in the lightswitch (see 2.2.11).	`l4_decide()`
		`teleport_pop()`
		`user_read()`
		`user_write()`
		`user_release()`

`lock_sem`	Used as internal semaphore for guarding the counter in the lightswitch.	`ls_lock()` `ls_unlock()`

continued on the next page

Table C.1: Lock variables (cont.)

Name	Meaning	Used in function
`prior_sem`	Used for giving priority to lightswitch when closing the authorization server.	`ls_lock()` `user_release()`
`constable_openclose`	Used to guard exclusive access to open and close operations.	`user_open()` `user_release()`

Table C.2: Signal variables in `chardev.c`

Name	**Meaning**
`constable_present`	Value is one if the authorization server is running, otherwise zero.
`currently_receiving`	Set to one when authorization server is receiving something (`user_write()` is running).

Table C.4: Other important global variables in `chardev.c`

Name	**Meaning**
`constable`	Pointer to the `task_struct` structure of the authorization server. Used to skip decision requests (authorization server can't decide its own system calls, it would deadlock itself).
`gdb`	Pointer to the `task_struct` structure of debugger, if it is attached to the authorization server. It's use is similar to the `constable` pointer.
`decision_request_id`	Atomic integer variable that is incremented when a new decision request arrives. It's used to assign a unique identifier to each decision request.
`kclasses_registered`	A circular list of registered *k-classes*. Used to deregister them during *close*.
`evtypes_registered`	A circular list of registered *event types*. Used to unregister them during *close*.
`user_answer`	Holds the answer from authorization server for user process. Set by the `user_write()` function and used by the `l4_decide()` function.
`userspace_answer`	Waitqueue for processes waiting for a decision.
`recv_req_id`	Integer variable that determines which decision request has been asnwered by the authorization server. Set by the `user_write()` function and used by the `l4_decide()` function.

D Teleport

Teleport is a family of data structures used to send data from the kernel to the authorization server. Here is a list of these data structures along with an explanation of their meaning:

teleport_t Main structure that controls communication from the kernel to the authorization server. It consists of:

- a pointer to the currect instruction (instruction pointer)
- the current state (see D.4)
- a pointer to the user buffer
- number of remaining bytes to send
- pointer to data structure to send (attribute, *k-class* or *event type*, represented by a union)

There is one global `teleport` (of type `teleport_t`) in the kernel. Concurrent access to it is guarded by the `user_read_lock` semaphore.

teleport_insn_t Structure defining one teleport instruction. It consists of:

- Operation code (integer)
- Data arguments (type depends on operation code, see D.1 and D.2 for more). In the programming code, this is represented by a *union*.

It is always used in an array as larger instruction block. The last instruction has to be of type `HALT`, so the teleport knows when to stop reading.

tele_item Structure representing one element of the queue. It consists of:

- pointer to an array of teleport instructions
- `list_head` structure
- size of the teleport instructions in bytes
- a pointer to the deallocation function

D.1 Operation codes

We present full list of possible operation codes along with their meaning:

NOP does nothing

PUT16 puts 16-bit constant

PUT32 puts 32-bit constant

PUTPtr puts pointer value

CUTNPASTE puts a memory region

PUTATTRS put attributes

PUTKCLASS puts kclass (without attributes) and assigns it a number

PUTEVTYPE puts evtype (without attributes) and assigns it a number

HALT end of the routine, the last instruction of an instruction block

D.2 Arguments

Here is complete list of possible arguments depending on the *operation code*:

NOP array of two pointers

PUT16 16-bit unsigned integer

PUT32 32-bit unsigned integer

PUTPtr 64-bit unsigned integer

CUTNPASTE memory region, defined by:

- pointer to byte array
- number of bytes to be transferred

PUTATTRS pointer to an attribute list

PUTKCLASS pointer to a *k-class* definition

PUTEVTYPE pointer to an *event type* definition

D.3 Possible teleport messages

Here is a complete list of all possible messages currently sent between Medusa and authorization module:

K-class registration

Table D.1: Data stored in teleport for *k-class* registration

Operation code	Contents
PUTPtr	0
PUT32	KCLASS_DEF
PUTKCLASS	kclass pointer
PUTATTRS	attribute list

Event type registration

Table D.2: Data stored in teleport for *kevent* registration

Type	Contents
PUTPtr	0
PUT32	EVTYPE_DEF
PUTKCLASS	evtype pointer
PUTATTRS	attribute list

Decision request

Table D.3: Data stored in teleport for decision request

Type	Contents
PUTPtr	pointer to evtype
PUT32	request_id
CUTNPASTE	event
CUTNPASTE	object
CUTNPASTE	subject (optional)

Fetch answer

Table D.4: Data stored in teleport for fetch answer

Type	Contents
PUTPtr	0
PUT32	fetch answer or error
PUTPtr	kclass id
PUTPtr	answer seq
CUTNPASTE	answer kobject

Update answer

Table D.5: Data stored in teleport for update answer

Type	Contents
PUTPtr	0
PUT32	update answer or error
PUTPtr	kclass id
PUTPtr	answer seq
PUT32	answer result

D.4 Teleport states

Teleport saves its state in between read calls. Its current state is saved in the `cycle` variable. List of all possible states follows:

FETCH Fetch the instruction and decode it.

EXECUTE Execute the instruction.

HALT Teleport is not in use.

D.5 Teleport interface

Interface of the `teleport` consists of two functions:

D.5.1 teleport_reset()

This function is called after teleport structure has been filled with new data ready to be sent to the authorization server. It sets instruction pointer of the teleport to the beginning of the teleport structure and current state of the teleport to FETCH.

D.5.2 teleport_cycle()

This is the function that executes the instructions and sends data to the authorization server (user space). Based on the current teleport state it does two things:

1. If current state of teleport is FETCH, it will set its state to EXECUTE, read current instruction and set argument pointers to correct locations based on the instruction. It will also increment instruction pointer to the next instruction and set remaining value to the correct value based on the instruction.

2. If current state of teleport is EXECUTE, teleport will call `place_to_user()` function that will send data to the authorization server.

Returns number of bytes that have been read by the authorization server or negative value in case of an error.

D.6 Teleport instruction array

Teleport instruction array is usually dynamically allocated with one exception — teleport instruction array used in `l4_decide()` is allocated on the stack.

Individual instructions are filled with data according to the Medusa communication protocol. Possible messages are listed in section D.3. As instructions are filled in, a total size of teleport arguments is counted. Last instruction operation code has to be `HALT`.

Pointer to a teleport instruction array is saved to a dynamically allocated queue item structure. Computed size of all teleport arguments is written in the `size` structure field.