

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5384-72886

**IMPLEMENTÁCIA KONTROLY IPC DO SYSTÉMU
MEDUSA
DIPLOMOVÁ PRÁCA**

2018

Viliam Mihálik

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5384-72886

IMPLEMENTÁCIA KONTROLY IPC DO SYSTÉMU
MEDUSA
DIPLOMOVÁ PRÁCA

Študijný program: Aplikovaná informatika
Číslo študijného odboru: 2511
Názov študijného odboru: 9.2.9 Aplikovaná informatika
Školiace pracovisko: Ústav informatiky a matematiky
Vedúci záverečnej práce: Mgr. Ing. Matúš Jókay, PhD.

Bratislava 2018

Viliam Mihálik

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Viliam Mihálik
Diplomová práca:	Implementácia kon- troly IPC do systému Medusa
Vedúci záverečnej práce:	Mgr. Ing. Matúš Jókay, PhD.
Miesto a rok predloženia práce:	Bratislava 2018

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean et est a dui semper facilisis. Pellentesque placerat elit a nunc. Nullam tortor odio, rutrum quis, egestas ut, posuere sed, felis. Vestibulum placerat feugiat nisl. Suspendisse lacinia, odio non feugiat vestibulum, sem erat blandit metus, ac nonummy magna odio pharetra felis. Vivamus vehicula velit non metus faucibus auctor. Nam sed augue. Donec orci. Cras eget diam et dolor dapibus sollicitudin. In lacinia, tellus vitae laoreet ultrices, lectus ligula dictum dui, eget condimentum velit dui vitae ante. Nulla nonummy augue nec pede. Pellentesque ut nulla. Donec at libero. Pellentesque at nisl ac nisi fermentum viverra. Praesent odio. Phasellus tincidunt diam ut ipsum. Donec eget est. A skúška mäččėňov a dĺžnov.

Kľúčové slová: Medusa, IPC, Linux, LSM

ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Author:	Viliam Mihálik
Master's thesis:	Implement IPC control on Medusa system
Supervisor:	Mgr. Ing. Matúš Jókay, PhD.
Place and year of submission:	Bratislava 2018

On the other hand, we denounce with righteous indignation and dislike men who are so beguiled and demoralized by the charms of pleasure of the moment, so blinded by desire, that they cannot foresee the pain and trouble that are bound to ensue; and equal blame belongs to those who fail in their duty through weakness of will, which is the same as saying through shrinking from toil and pain. These cases are perfectly simple and easy to distinguish. In a free hour, when our power of choice is untrammelled and when nothing prevents our being able to do what we like best, every pleasure is to be welcomed and every pain avoided. But in certain circumstances and owing to the claims of duty or the obligations of business it will frequently occur that pleasures have to be repudiated and annoyances accepted. The wise man therefore always holds in these matters to this principle of selection: he rejects pleasures to secure other greater pleasures, or else he endures pains to avoid worse pains.

Keywords: Medusa, IPC, Linux, LSM

Pod'akovanie

I would like to express a gratitude to my thesis supervisor.

Obsah

Úvod	1
1 IPC	2
1.1 Signály	2
1.2 Rúry	3
1.2.1 Anonymné rúry	3
1.2.2 Pomenované rúry	4
1.3 Fronty správ	5
1.3.1 Štruktúry fronty správ	5
1.3.2 Štruktúra kern_ipc_perm	7
1.3.3 Požívanie fronty správ	7
1.4 Semaforey	9
1.4.1 Štruktúry semaforov	10
1.4.2 Použitie semaforov	11
1.5 Zdieľaná pamäť	13
1.5.1 Štruktúry zdieľanej pamäte	13
1.5.2 Požitie zdieľanej pamäte	14
1.6 Sokety	15
1.6.1 Použitie soketov	15
2 Medusa	17
2.1 LSM framework	17
2.2 Autorizačný server	18
2.3 Princíp fungovania	18
2.3.1 k-objekt	18
2.3.2 Typy prístupov	19
2.4 Architektúra	20
3 Implementácia	22
3.1 Bezpečnostná štruktúra	22
3.2 Vytvorenie <i>k-objektu</i>	23
3.2.1 Konverzné funkcie	24
3.2.2 Operácie <i>fetch</i> a <i>update</i>	27
3.3 Typy prístupov	29
3.4 Konfigurácia autorizačného servera	31

Záver	32
Zoznam použitej literatúry	33
Prílohy	I
A Štruktúra elektronického nosiča	II
B Algoritmus	III
C Výpis subline	XIII

Zoznam obrázkov a tabuliek

Obrázok 1	Tok dát medzi procesmi.	4
Obrázok 2	Schéma komunikačného protokolu[20]	21
Obrázok 3	Princíp fungovania <i>container_of</i> makra [22]	25

Zoznam skratiek

FIFO	First In, First Out
IPC	Inter-Process Communication
LSM	Linux Security Module
PID	Process Identifier
POSIX	Portable Operating System Interface
RAM	Random Access Memory
RCU	Read-Copy-Update
SELinux	Security-Enhanced Linux - bezpečnostné riešenie, ktoré poskytuje bezpečnostnú politiku riadenia prístupu
VFS	Virtual File System

Zoznam algoritmov

Zoznam výpisov

1	Použite anonymných rúr	4
2	Príklad vlastnej štruktúry správy	5
3	Štruktúra <code>msg_msg</code>	6
4	Štruktúra <code>msg_msgseg</code>	6
5	Štruktúra <code>msg_queue</code>	6
6	Vytvorenie soketu	15
7	<i>Bindovanie</i> soketu	15
8	Bezpečnostná štruktúra Medusy pre IPC	19
9	Bezpečnostná štruktúra Medusy pre IPC	22
10	Príklad použitia pomocnej funkcie na alokovanie bezpečnostnej štruktúry	23
11	LSM <i>hooky</i> pre operácie <code>get</code> a <code>ctl</code>	23
12	Všeobecný <i>k-objekt</i>	23
13	Signatúra obslužnej funkcie pre LSM <i>hook</i> <code>ipc_permission</code>	24
14	Konverzná funkcia zo štruktúry jadra na štruktúru Medusy pre SEM	25
15	Konverzná funkcia zo štruktúry Medusy na štruktúru jadra pre SEM	26
16	Konverzná funkcia zo štruktúry Medusy na štruktúru jadra	27
17	Operácie <code>fetch</code> pre <i>k-objekt</i> semaforu	27
18	Operácie <code>update</code> pre <i>k-objekt</i> semaforu	28
19	Volanie funkcie <i>typu prístupu</i> z vrstvy L1	30
B.1	Štruktúra <code>kern_ipc_perm</code>	III
B.2	Štruktúra <code>msqid64_ds</code>	III
B.3	Štruktúra <code>msqid_ds</code>	III
B.4	Štruktúra <code>msginfo</code>	IV
B.5	Štruktúra <code>semid_ds</code>	IV
B.6	Štruktúra <code>semid64_ds</code>	IV
B.7	Štruktúra <code>semun</code>	V
B.8	Štruktúra <code>shmid_kernel</code>	V
B.9	Ukážka použitia soketu	V
B.10	Ukážka použitia soketu na strane klienta	VI
B.11	K-objekt procesu	VII
B.12	LSM <i>hooky</i> pre IPC	VIII
B.13	Alokovanie bezpečnostnej štruktúry	VIII
B.14	Uvoľnenie pamäte bezpečnostnej štruktúry	IX

B.15 Operácie fetch	IX
B.16 Operácie update	IX
B.17 <i>Typ prístupu</i> ipc_associate	X
B.18 Konfigurácia autorizačného servera	XI
C.1 Ukážka sublime-project	XIII

Úvod

Tu bude krásny úvod s diakritikou atd.

A možno aj viac riadkový úvod.

1 IPC

Inter-Process Communication (IPC) predstavuje súbor mechanizmov určených na komunikáciu a správu dát medzi viacerými procesmi. Operačný systém Linux obsahuje niekoľko takýchto mechanizmov, medzi najhlavnejšie patria:

- Signály
- Rúry
- Fronty správ
- Semaforey
- Zdieľaná pamäť
- Sokety

Jadro operačného systému Linux obsahuje dve rôzne implementácie pre semaforey, fronty správ a zdieľanú pamäť. Tieto implementácie sa nazývajú *System V* a *POSIX*. *System V* je staršia implementácia, ktorá je odvodená od komerčného Unix systému *System V*. Implementácia *POSIX* je štandard, ktorý taktiež implementuje tieto mechanizmy avšak funkcie a systémové volania sa líšia. Každá implementácia má svoje výhody a nevýhody, avšak implementácia *POSIX* bola vyvinutá neskôr ako LSM a teda nemá vytvorené LSM *hooky*, ktoré by mali dopad na túto diplomovú prácu. Preto v nasledujúcich odsekoch budeme popisovať *System V* semaforey(ďalej len semaforey), *System V* fronty správ(ďalej len fronty správ) a *System V* zdieľanú pamäť(ďalej len zdieľaná pamäť).

1.1 Signály

Ide o jeden z najstarších IPC mechanizmov používaný v Unix systémoch. Signál je asynchrónne upozornenie zaslané procesu alebo konkrétnemu vláknu v rámci toho istého procesu, za účelom upozornenia na udalosť, ktorá sa vyskytla. V momente keď sa signál odošle, operačný systém preruší vykonávanie procesu, ktorý má byť signalizovaný a v tomto procese sa vykoná obsluženie signálu.[1]

Je potrebné si uvedomiť že **signály nie sú** to isté ako **prerušená**. Rozdiel medzi signálom a prerušením je, že prerušenie je vyvolané procesorom a signál je vyvolaný z jadra systému. Signál je možné vyvolať systémovým volaním **kill**. Toto systémové volanie má dva parametre[2]:

- *pid* - identifikátor procesu, ktorý má byť signalizovaný

- *sig* - typ signálu

Podporované typy je možné zistiť pomocou príkazu *kill -l* alebo v súbore */include/linux/signal.h*.

V prípade že je definovaná obslužná funkcia, táto funkcia sa vykoná, v opačnom prípade je použitá štandardná obsluha signálu. Obslužnú funkciu je možné definovať pomocou funkcie **signal**, avšak správanie tejto funkcie môže byť rozdielne vzhľadom na platformu. Preto sa odporúča používať funkciu **sigaction**, ktorá bola definovaná v štandarde **POSIX.1**. Toto systémové volanie má parametre[3]:

- *signum* - typ signálu, ktorý chceme obslúžiť
- *act* - definuje akciu, ktorá sa má vykonať pri obsluhu signálu
- *oldact* - definuje starú obsluhu signálu

Signály je možné použiť pre komunikáciu ako aj synchronizáciu avšak ide o veľmi slabý nástroj pre tieto potreby.¹

1.2 Rúry

Rúry predstavujú jednosmerný tok dát medzi procesmi: všetky dáta zapísané procesom do rúry sú jadrom presmerované do iného procesu, ktorý z nej môže čítať. Poznáme 2 druhy rúr [4]:

- Anonymné rúry - žiadny objekt v súborovom strome
- Pomenované rúry - objekt v súborovom strome

1.2.1 Anonymné rúry

Anonymné rúry je možné vytvoriť pomocou systémového volania **pipe**, alebo tiež pomocou znaku *|* vo väčšine *Unix* príkazových riadkoch. Systémové volanie **pipe** obsahuje jeden parameter, ktorým je pole *pipefd* o veľkosti 2. Toto pole obsahuje po návrate z funkcie súborové deskriptory. Tieto dva súborové deskriptory predstavujú konce rúry, *pipefd[0]* je čítací koniec rúry a *pipefd[1]* je zapisovací koniec rúry. Tieto súborové deskriptory je následne možné použiť na zapisovanie a čítanie pomocou systémových volaní *write*² a *read*³. Tieto operácie sú blokujúce v dvoch prípadoch:

- *write* - rúra je plná

¹http://man7.org/conf/lca2013/IPC_Overview-LCA-2013-printable.pdf

²<http://man7.org/linux/man-pages/man2/write.2.html>

³<http://man7.org/linux/man-pages/man2/read.2.html>

- read - rúra je prázdna

Niektoré *Unix* systémy ako napríklad *System V Release 4*, implementuje **full-duplex** rúry, teda rúry, pri ktorých oba konce rúry(súborové deskriptory) je možné použiť ako na zapisovanie tak aj na čítanie. Avšak štandard **POSIX** definuje iba **half-duplex** rúry, pričom každý proces musí zatvoriť jeden deskriptor pred použitím druhého.[4]

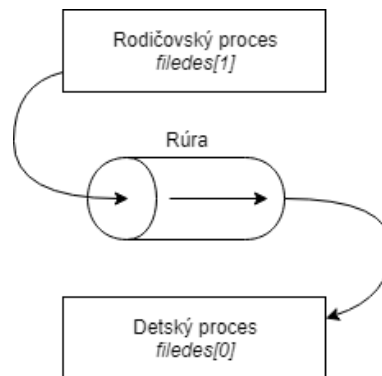
Takto vytvorená rúra umožňuje komunikáciu medzi rodičovským procesom a jeho potomkami. Avšak je potrebné zabezpečiť aby procesy, ktoré medzi sebou chcú komunikovať zdieľali rovnaké súborové deskriptory. Toto je možné jednoducho dosiahnuť tak, že sa rúra vytvorí pred vytvorením detského procesu(*fork*). Zjednodušený pseudokód môžeme vidieť tu 1 spolu s ilustračným obrázkom 1.[5]

```
int filedes[2];

pipe(filedes);

child_pid = fork();
if (child_pid == 0) {
    close(filedes[1]);
} else {
    close(filedes[0]);
}
```

Listing 1: Použité anonymných rúr



Obr. 1: Tok dát medzi procesmi.

Nevýhodou tohto systémového volania je absencia v súborovom strome a teda nie je možné využiť tento IPC mechanizmus na komunikáciu medzi ľubovoľnými dvoma alebo viacerými procesmi.[5]

1.2.2 Pomenované rúry

Pomenované rúry taktiež nazývané aj **FIFO** na rozdiel od anonymných rúr majú meno v súborovom systéme. Jednou z možností ako vytvoriť tento typ rúry je pomocou

funkcie **mkfifo**, ktorá je definovaná v štandarde POSIX. Táto funkcia v svojej implementácii volá systémové volanie *mknod* s príznakom, ktorý definuje že ide o pomenovanú rúru. Funkcia **mkfifo** má 2 parametre:

- *pathname* - názov rúry
- *mode* - práva súboru

Návratová hodnota funkcie je 0 v prípade úspechu a v prípade chyby je návratová hodnota -1. Takto vytvorenú rúru je možné rovnako ako anonymné rúry obsluhovať pomocou systémových volaní *read* a *write*.^[6]

1.3 Fronty správ

Fronty správ si najlepšie môžeme predstaviť ako zretazený zoznam v adresnom priestore jadra. Správy môžu byť odosielané do fronty v poradí a následne načítané z fronty pomocou niekoľko rôznych spôsobov. Každá fronta správ je jednoznačne identifikovaná pomocou IPC identifikátora. Pre lepšie pochopenie tohto konceptu si popíšeme 3 hlavné štruktúry, ktoré zabezpečujú fungovanie fronty správ v jadre Linuxu.^[7]

1.3.1 Štruktúry fronty správ

Štruktúra `msgbuf` (definovaná v `linux/msg.h`) predstavuje predlohu ako by mala vyzeráť správa, ktorú budeme posilať. Táto predloha obsahuje dve položky:

- long **mtype** - umožňuje určiť o aký typ správy ide, napríklad *chybová správa*, *normálna správa* a podobne, možností je nekonečno
- char **mtext**[1] - samotné dáta správy, túto položku je možné ľubovoľne rozšíriť, ako napríklad v ukážke kódu 2, avšak s limitom, ktorý je maximálna dĺžka správy **MSGMAX**=8192⁴

```
struct message {
    long type;
    struct my_special_struct data;
} msg;
```

Listing 2: Príklad vlastnej štruktúry správy

Každá správa je rozdelená na stránky, ktoré sú dynamicky alokované v pamäti. Veľkosť tejto stránky je závislá od architektúry a zisťuje sa nasledovne `sysconf(_SC_PAGESIZE)`.

⁴Táto veľkosť je definovaná v `linux/msg.h` a môže sa líšiť od verzie jadra. Túto hodnotu je taktiež možné zistiť pomocou príkazu `ipcs -l`.

Štruktúra ***msg_msg***, ktorú môžeme vidieť tu 3, predstavuje hlavičku každej správy pričom sa inštancie tejto štruktúry nachádzajú v zretazenom zozname, ktorý je definovaný položkou *m_list*. V prípade že dĺžka správy je menšia ako

$$\text{PAGE_SIZE} - \text{sizeof}(\text{struct msg_msg})$$

celý obsah správy sa nachádza v pamäťovej oblasti za štruktúrou *msg_msg*. V opačnom prípade sa prvá časť správy nachádza na rovnakom mieste a ostatné stránky sa nachádzajú v pamäťovej oblasti, na ktorú ukazuje ukazovateľ *next*. Tento ukazovateľ odkazuje na štruktúru *msg_msgseg* pozri 5, ktorá obsahuje ukazovateľ *next* na ďalšiu stránku, pričom v pamäťovej oblasti za touto štruktúrou sa nachádzajú dáta aktuálnej stránky.

```
struct msg_msg {
    struct list_head m_list;
    long m_type;
    size_t m_ts;
    struct msg_msgseg *next;
    void *security;
    /* obsah prvej stránky */
};
```

Listing 3: Štruktúra *msg_msg*

```
struct msg_msgseg {
    struct msg_msgseg *next;
    /* obsah stránky */
};
```

Listing 4: Štruktúra *msg_msgseg*

Poslednou štruktúrou, ktorá sa nachádza najvyššie v hierarchii týchto štruktúr je ***msq_queue*** a jej deklaráciu môžete vidieť na ukážke kódu 5. Najdôležitejšou položkou tejto štruktúry je položka *q_messages*, ktorá predstavuje prvý element zretazeného zoznamu všetkých správ vo fronte. Ďalšie zretazené zoznamy, ktoré sa tu nachádzajú sú *q_receivers* a *q_senders*, ktoré obsahujú zretazené zoznamy procesov, ktoré posielajú správy a procesov, ktoré prijímajú správy. Zaujímavou položkou z pohľadu bezpečnosti je položka *q_perm*, ktorá obsahuje inštanciu štruktúry ***kern_ipc_perm***. Túto štruktúru si popíšeme v kapitole 1.3.2.[4]

```
struct msg_queue {
    struct kern_ipc_perm q_perm;
    /* meta dáta */
    struct list_head q_messages;
    struct list_head q_receivers;
    struct list_head q_senders;
} __randomize_layout;
```

Listing 5: Štruktúra *msq_queue*

1.3.2 Štruktúra `kern_ipc_perm`

Štruktúra `kern_ipc_perm` sa nenachádza len pri fronte správ ale taktiež aj pri semaforoch a zdieľanej pamäti. Táto štruktúra predstavuje sadu meta dát o konkrétnom IPC objekte a jej položky spolu s typmi môžeme vidieť na ukážke kódu B.1. Vyznám týchto položiek je nasledovný:

- `lock` - uzamykací mechanizmus pre ochranu IPC objektu
- `deleted` - príznak, či bol zdroj uvoľnený
- `id`
- `key` - jednoznačný identifikátor, v rámci konkrétneho typu objektu, to znamená, že jedna inštancia semafora, zdieľanej pamäte alebo fronty správ môže mať rovnaký identifikátor
- `uid` - ID používateľa, ktorý vlastní IPC objekt
- `gid` - ID skupiny, ktorá vlastní IPC objekt
- `cuid` - ID používateľa, ktorý vytvoril IPC objekt
- `cgid` - ID skupiny, ktorá vytvorila IPC objekt
- `mode` - bitová maska oprávnení
- `seq` - sekvenčné číslo, používané sa na vytvorenie nového identifikátora pre IPC objekt
- `security` - ukazovateľ na štruktúru, ktorú vytvára zvolené bezpečnostné riešenie v jadre Linuxu
- `khtnode`
- `rcu` - RCU synchronizačný mechanizmus
- `refcount` - počítadlo použitia IPC objektu

1.3.3 Požívanie fronty správ

Na vytvorenie fronty správ sa používa systémové volanie `msgget`. Toto systémové volanie má dva parametre, ktorými sú `key`(identifikátor objektu) a `msgflg` príznaky IPC objektu. Nová fronta je vytvorená v prípadoch, že:

- `key` sa rovná `IPC_PRIVATE`
- `key` nemá ešte priradený žiadny IPC objekt a `IPC_CREATE` príznak je definovaný v parametre `msglflg`

Ak je definovaný príznak `IPC_EXCL` spolu s `IPC_CREATE` a identifikátor už existuje, volanie funkcie zlyhá s chybovou správou `EEXIST`, ktorá definuje, že IPC objekt už existuje. Naopak v prípade, že `IPC_EXCL` nie je definované tak návratová hodnota je identifikátor už existujúceho IPC objektu.[8]

Na posielanie a prijímanie správ z fronty sa používajú systémové volania `msgsnd` a `msgrcv`. Funkcia `msgsnd` má nasledovné parametre[9]:

- `msgid` identifikátor fronty získaný z funkcie `msgget`
- `msgp` ukazovateľ na štruktúru, ktorú si používateľ definuje sám a mal by vychádzať zo šablóny ktorú sme si popísali v kapitole 2
- `msgsz` veľkosť dátovej štruktúry, ktorú chceme prenášať
- `msgflg` príznaky, ktoré definujú čo sa má diať v prípade, že je fronta plná

Funkcia `msgrcv`, odstráni správu z frontu a premiestni do pamäte, na ktorý ukazuje parameter funkcie `msgp`. Ďalšie parametre sú[9]:

- `msgsz` maximálnu veľkosť dát v bytoch pre položku `mtext`, štruktúry, na ktorú ukazuje ukazovateľ `msgp`
- `msgflg` príznaky, ktoré definujú čo sa má diať v prípade, že je fronta prázdna
- `msgtyp` číslo, ktoré definuje ktorý typ správy bude ako prvý vybraný z fronty (nemôže byť definovaný príznak `MSG_COPY`), v prípade že je definovaný príznak `MSG_EXCEPT`, tak sa z fronty vyberá prvá správa z typom odlišným od `msgtyp`

Posledným systémovým volaním tohto mechanizmu je `msgctl`, ktoré vykonáva kontrolné operácie nad objektom. Funkcia má 3 parametre[10]:

- `msgid` identifikátor objektu
- `cmd` typ operácie

- `buf` ide o štruktúru, ktorá sa v jadre ako aj v manuálových stránkach nachádza v 32 bitovej verzii (viď B.3), avšak v jadre sa označuje za zastaralú pričom ju nahrádza 64 bitová verzia (viď B.2), táto štruktúra slúži na prenos meta dát z jadra systému do užívateľského priestoru

Typy operácie nad frontami správ sú nasledovné[10]:

- `IPC_STAT` a `MSG_STAT` kopíruje informácie z jadra do štruktúry na ktorú ukazuje ukazovateľ `buf`
- `IPC_SET` nastavuje položky jadra na základe ukazovateľa `buf`, konkrétne `msg_perm.uid`, `msg_perm.gid`, `msg_qbytes` a posledných 9 bitov `msg_perm.mode`
- `IPC_RMID` odstraňuje frontu správ
- `IPC_INFO` a `MSG_INFO` získava limity a systémové nastavenia pre fronty správ, dáta sa nachádzajú na adrese ukazovateľa `buf`, avšak dáta sú v štruktúre typu `msginfo` (viď B.4) a preto je potrebné pre-typovanie⁵

1.4 Semaforey

Semafor ako IPC mechanizmus nepredstavuje nástroj na prenášanie dát ale slúži ako synchronizačný mechanizmus na ochranu zdieľaných zdrojov pri viac procesovom alebo viac vláknovom vykonávaní programu. Všeobecný semafor si môžeme predstaviť ako počítadlo, ktoré je možné atomicky upravovať. Semafor zvyčajne implementuje dve základné funkcie, ktoré slúžia na zvýšenie(`signal`) a zníženie(`wait`) tohto počítadla. Napríklad ak proces 1 chce vstúpiť do chránenej oblasti (pristúpiť k zdieľaným zdrojom) zníži počítadlo semaforu. Proces 2, ktorý taktiež bude chcieť pristúpiť k týmto zdrojom zníži semafor čo má za následok blokovanie procesu 2, ktorý musí počkať na zvýšenie počítadla. Proces 1 v prípade že bude opúšťať kritickú oblasť zvýši počítadlo semaforu čo zabezpečí odblokovanie procesu 1.

Semafor *System V* semaforey na rozdiel od POSIX semaforov nepredstavujú len jedno počítadlo ale skupiny počítadiel. Každé jedno počítadlo môže chrániť nejakú kritickú oblasť a teda jeden *System V* semafor môže chrániť viacero kritických oblastí. Veľkou výhodou je taktiež schopnosť navrátiť operácie vykonané na semafore v prípade, že proces, ktorý bol v kritickej oblasti neočakávane skončí a zabezpečiť tak aby čakajúci proces mohol vstúpiť do kritickej oblasti.

⁵`IPC_INFO` a `MSG_INFO` ako aj `IPC_STAT` a `MSG_STAT` vracajú mierne odlišné dáta a sú platformovo závislé pre viac info pozri <http://man7.org/linux/man-pages/man2/msgctl.2.html>

V nasledujúcich odsekoch si popíšeme interné štruktúry v jadre Linuxu, ktoré zabezpečujú fungovanie semaforov a taktiež aj použitie tohto IPC mechanizmu.[4]

1.4.1 Štruktúry semaforov

Základná štruktúra, ktorá v sebe nesie hodnotu jedného počítadla má nasledovné položky:

- `semval` hodnota počítadla
- `semval` PID procesu, ktorý posledný modifikoval semafor
- `lock` uzamykací mechanizmus pre ochranu počítadla
- `pending_alter` a `pending_const` operácie, ktoré čakajú na vykonanie
- `sem_otime` čas posledného volania funkcie `sem_op` nad počítadlom

Nadradenou štruktúrou, ktorá predstavuje skupiny počítadiel je `sem_array`, ktorá má nasledovné položky:

- `sem_perm` štruktúra typu `kern_ipc_perm`, ktorú sme si popísali v kapitole 1.3.2
- `sem_ctime` čas posledného volania funkcie `sem_ctl` nad semaforom
- `pending_alter` a `pending_const` operácie, ktoré čakajú na vykonanie
- `list_id` spätné operácie na celú skupinu semaforov, ide o vlastnosť, ktorú sme si popísali v úvode do kapitoly
- `sem_nsems` počet semaforov
- `complex_count` počet komplexných operácii ktoré čakajú na vykonanie
- `use_global_lock` globálny uzamykací mechanizmus nad celou skupinou semaforov
- `sems` pole jednotlivých semaforov/počítadiel

Operácie, ktoré sa nad týmito semaformi vykonávajú sú uložené v štruktúre `sem_queue`. Ide o zretazený zoznam a každá jedna inštancia tejto štruktúry predstavuje operácie jedného procesu, ktorý je blokovaný(spí) na semafore. Táto štruktúra vyzerá nasledovne:

- `list` ďalšie položky zretazeného zoznamu
- `sleeper` štruktúra typu `task_struct`, teda proces, ktorý spí

- **undo** spätné operácie
- **sops** pole štruktúr typu **sembuf**, ktoré definuje nevykonané operácie
- **blocking** pole štruktúr typu **sembuf**, ktoré definuje operácie ktoré sú blokovacie
- **nsops** počet operácií
- **alter** príznak, ktorý označuje či operácia modifikuje pole semaforov
- **dupsop** TODO príznak, ktorý označuje či operácia modifikuje pole semaforov

Štruktúry na prácu s operáciami, ktoré majú byť v prípade ukončenia programu obnovené do pôvodného stavu sú štruktúry **sem_undo** a **sem_undo_list**. Každý proces má jednu prislúchajúcu štruktúru **sem_undo** a v prípade že je proces ukončený tak sa tieto operácie vykonajú. Štruktúra **sem_undo_list** zabezpečuje zdieľaný prístup k **sem_undo** štruktúram v prípade že viacero procesov zdieľa jeden list čo je možné zabezpečiť pomocou príznaku **CLONE_SYSVSEM** pri vytváraní nového procesu.[11]

1.4.2 Použitie semaforov

System V semafor sa vytvára pomocou systémového volania **semget**, ktoré je analogické k funkcii **msgget**, ktorú sme si popísali v kapitole 1.3.3. Jediným rozdielom je parameter **nsems**, ktorý definuje koľko jednotlivých semaforov chceme vytvoriť.[12] Takto vytvorený semafor je možné používať a vykonávať nad ním operácie pomocou systémového volania **semop**, ktoré má nasledovné parametre:

- **semid** identifikátor semaforu
- **sops** operácie nad semaforom
- **nsops** počet operácií v poli

Jednotlivé operácie sú definované pomocou štruktúry **sembuf**, ktorá má nasledujúce položky:

- **sem_num** číslo semaforu nad ktorým chcem operáciu vykonať
- **sem_op** typ operácie
- **sem_flg** príznaky operácie

sem_flg môže nadobúdať dve hodnoty, ktoré sú **SEM_UNDO** a **IPC_NOWAIT**. **SEM_UNDO** indikuje že chceme aby daná operácia bola obnoviteľná. Príznak **IPC_NOWAIT** priamo súvisí s parametrom **sem_op**, ktorý môže nadobudnúť tieto stavy[13]:

- `sem_op > 0` hodnota `sem_op` sa pričíta k hodnota počítadla(vyžaduje sa právo na zápis)
- `sem_op == 0` a `semval == 0` tak operácia okamžite prebehne, inak ak je definovaný príznak `IPC_NOWAIT` operácie skončí s chybou ak však tento príznak nieje definovaný proces čaká pokiaľ bude semafor 0(vyžaduje práva na čítanie semaforu)
- `sem_op < 0` a zároveň `semval >= sem_op` tak je operácia vykonaná okamžite, inak analogicky k predošlému prípadu operácia skončí buď s chybou alebo proces čaká pokiaľ bude zvýšená hodnota počítadla(vyžaduje sa právo na zápis)

Posledným systémovým volaním je `semctl`, ktoré rovnako ako systémové volanie `msgctl` z kapitoly 1.3.3 získava alebo zapisuje informácie o IPC objekte avšak s rozdielom že využíva odlišnú štruktúru na ukladanie dát. `semctl` má nasledovné argument[14]:

- `semid` identifikátor objektu
- `semnum` index semaforu v skupine semaforov
- `cmd` typ operácie
- `arg` posledný argument je voliteľný podľa typu operácie, ide o union, ktorý je definovaný na ukážke kódu B.7

Typy operácii, ktoré sme si definovali pri `msgctl` taktiež existujú aj pri semafóroch a zachovávajú rovnakú funkcionálnosť ale výsledok týchto operácií sa ukladá do položky `arg.buf`, ktorá je typu `semid_ds`(viď B.5) pre 32 bitové systémy alebo `semid64_ds`(viď B.6) pre 64 bitové systémy. Toto systémové volanie umožňuje aj semaforovo špecifické operácie, ktoré sú nasledovné[14]:

- `GETALL/SETALL` vráti/nastaví hodnotu každého počítadla v skupine semaforov a uloží ho do položky `arg.array`, parameter `semnum` je ignorovaný
- `GETNCNT/GETZCNT` vráti počet procesov, ktoré čakajú na zvýšenie(`GETNCNT`) alebo zníženie(`GETZCNT`) konkrétneho počítadla, ktoré je definované argumentom `semnum`
- `GETPID` vráti PID procesu, ktorý posledný vykonal operáciu nad počítadlom, ktoré je definované argumentom `semnum`
- `GETVAL` vráti hodnotu jedného konkrétneho počítadla definovaného pomocou `semnum`
- `SETVAL` nastaví hodnotu, ktorá je v položke `arg.val`, jedného konkrétneho počítadla definovaného pomocou `semnum`

1.5 Zdieľaná pamäť

Zdieľaná pamäť predstavuje užitočný mechanizmus, ktorý umožňuje dvom alebo viacerým procesom pristupovať k spoločným dátovým štruktúram, ktoré sú uložené v oblasti zdieľanej pamäte IPC.[4] Proces, ktorý chce takúto zdieľanú pamäť používať potrebuje namapovať túto pamäť na adresný priestor procesu. Následne túto pamäť môže používať akoby lokálnu pamäť, čo nevyžaduje prepínanie do módu jadra a preto tento IPC mechanizmus patrí medzi najrýchlejšie.[11]

1.5.1 Štruktúry zdieľanej pamäte

Hlavnou štruktúrou, ktorá má informácie o objektoch zdieľanej pamäte je `shmid_kernel`, ktorej položky sú nasledovné:

- `shm_perm` štruktúra typu `kern_ipc_perm`, ktorú sme si popísali v kapitole 1.3.2
- `shm_file` pointer na štruktúru `file`, ktorá predstavuje zdieľaný pamäťový priestor
- `shm_nattch` počet procesov, ktoré sú *pripojené* k zdieľanej pamäti
- `shm_segsz` veľkosť pamäťového segmentu
- `shm_atim/shm_dtim/shm_ctim` sú posledné časy prístupu/odpojenia/zmeny
- `shm_cprid` PID procesu, ktorý vytvoril objekt
- `shm_lprid` PID procesu, ktorý posledný pristupoval ku objektu
- `mlock_user` ukazovateľ na štruktúru `user_struct`, ktorá definuje používateľa, ktorý zamkol zdieľanú pamäť v RAM⁶
- `shm_creator` ukazovateľ na štruktúru `task_struct`, ktorý definuje proces, ktorý vytvoril zdieľanú pamäť(NULL v prípade že bol proces ukončený)
- `shm_clist` zoznam štruktúr `shmid_kernel`, ktoré majú rovnaký proces, ktorý ich vytvoril

Najdôležitejšou položkou je `shm_file`, ktorá predstavuje samotnú zdieľanú pamäť a keďže ide o súbor, môžeme vidieť blízke prepojenie s Linux VFS. Avšak nejde o normálny súbor a nie je možné ho nájsť v strome súborového systému, pretože sa využíva špeciálny *shm* súborový systém.[4] Preto ak proces chce zapisovať alebo čítať z tohoto pamäťového segmentu je potrebné aby sa pripojil. Ako na to sa dozvieme v kapitole 1.5.2.

⁶pre viac info pozri <http://man7.org/linux/man-pages/man2/mlock.2.html>

1.5.2 Požitie zdieľanej pamäte

Zdieľanú pamäť podobne ako ostatne *System V* mechanizmy je možné vytvoriť pomocou systémového volania `shmget`. Toto systémové volanie sa líši v argumente `size`, ktorý určuje akú veľkú pamäť chceme alokovať, pričom táto pamäť je zaokrúhlená nahor k najbližšiemu násobku `PAGE_SIZE`. Jedným z rozdielov je taktiež možnosť definovať príznaky `SHM_HUGETLB`, `SHM_HUGE_2MB` a `SHM_HUGE_1GB`, ktoré signalizujú alokáciu s použitím *huge pages*⁷. Posledný z príznakov je `SHM_NORESERVE`, ktorý definuje že sa nemá rezervovať *swap* pamäť.

Nad takto vytvorenou zdieľanou pamäťou je možné robiť dve operácie `shmat` a `shmdt`. `shmat` pripojí zdieľanú pamäť do adresného priestoru procesu, argumenty sú nasledovné:

- `shmid` identifikátor zdieľanej pamäte
- `shmaddr` adresa na ktorú sa zdieľaná pamäť pripojí, môže nadobudnúť nasledovné hodnoty:
 - `NULL` systém sám vyberie najvhodnejšiu nepoužívanú adresu
 - rôzna od `NULL` a zároveň je definovaný príznak `SMH_RND` tak pamäť je pripojená a zarovnaná dole na najbližší násobok `SHMLBA`⁸ ak však príznak nie je definovaný adresa musí byť zarovnaná na násobok `PAGE_SIZE` a následne môže byť pamäť pripojená
- `shmflg` môže nadobudnúť nasledovné hodnoty
 - `SHM_EXEC` povoľuje spúšťanie obsahu, ktorý sa nachádza v zdieľanej pamäti
 - `SHM_RDONLY` pripojí proces len s prístupom na čítanie, ak príznak nie je definovaný proces sa bude pripájať s prístupom na čítanie a zápis avšak **proces musí mať práva na čítanie a zápis**
 - `SHM_REMAP` príznak povoľuje prepísanie existujúceho mapovania, `shmaddr` nesmie byť `NULL`

Po úspešnom pripojení funkcia `shmat` vracia adresu pripojenej pamäte v opačnom prípade `(void *) -1`.

⁷Pre viac info pozri <https://elixir.bootlin.com/linux/latest/source/Documentation/vm/hugetlbpage.txt>

⁸Táto hodnota je zväčša násobok `PAGE_SIZE` a na väčšine Linuxových architektúrach je rovnaká ako `PAGE_SIZE`

Na odpojenie od zdieľanej pamäte sa používa funkcia `shmdt`, ktorá má jeden argument `shmaddr`, ktorý definuje adresu na ktorej je zdieľaná pamäť pripojená. Táto funkcia vracia 0 v prípade úspechu, inak vracia -1.

Posledným systémovým volaním je `shmctl`, ktoré pracuje analogicky k systémovému volaniu `msgctl` o ktorom sa môžete viac dočítať v závere kapitoly 1.3.3. `shmctl` poskytuje na rozdiel od `msgctl` 2 príznaky, ktoré sú `SHM_LOCK` a `SHM_UNLOCK`. Tieto príznaky povolia alebo zabráňujú *swapovaniu* zdieľanej pamäte. `SHM_UNLOCK` definuje, že napríklad v situáciu veľkého vyťaženia pamäte môže *swap* pamäte.[15]

Na prácu so zdieľanou pamäťou teda zapisovanie a čítanie z pamäte používame rovnaké nástroje ako na prácu s bežným súborom a teda na zapisovanie môžeme použiť napríklad `fprintf` a na čítanie `putchar`.

1.6 Sokety

Soket predstavuje obojsmernú komunikačnú rúru, ktorá môže byť použitá na široké množstvo oblastí. Jedna z najpoužívanějších oblastí pri soketoch je komunikácia cez Internet. Avšak sokety taktiež umožňujú aj lokálnu komunikáciu medzi dvoma procesmi.[16] Nakoľko sokety zaberajú široký záber v nasledujúcich odsekoch sa pokúsime tento koncept opísať na príklade lokálnych soketov a preto, niektoré informácie nemusia byť kompletne a pre odlišné použitie sa treba inštruovať podľa[17].

1.6.1 Použitie soketov

Soket sa vytvára pomocou funkcie `socket` a vytvorenie lokálneho soketu vyzerá nasledovne:

```
unsigned int s;  
s = socket(AF_UNIX, SOCK_STREAM, 0);
```

Listing 6: Vytvorenie soketu

Prvý parameter `AF_UNIX`(`AF_LOCAL`) definuje že ide o lokálny soket, príznak `SOCK_STREAM` určuje sekvenčný, spoľahlivý a obojsmerný typ komunikácie. Posledný parameter definuje protokol pre konkrétny typ komunikácie a väčšinou existuje len jeden takýto protokol. V prípade že volanie tejto funkcie je úspešné tak návratová hodnota je deskriptor súboru v opačnom prípade -1.

Takto vytvorený deskriptor je potrebné namapovať na nejakú cestu v súborovom systéme aby ju mohlo používať viacero procesov. Na tento účel nám slúži systémové volanie `bind`, ktoré môžeme použiť nasledovne:

```
struct sockaddr_un local;  
unsigned int s;
```

```

local.sun_family = AF_UNIX;
strcpy(local.sun_path, "/home/mysocket");
unlink(local.sun_path);
bind(s, (struct sockaddr *)&local, sizeof(local));

```

Listing 7: *Bindovanie* soketu

Prvým parametrom tejto funkcie je deskriptor soketu, druhý parameter je ukazovateľ na štruktúru, typu `sockaddr`, typ tejto štruktúry sa líši od použitého typu komunikácie, v ukážke kódu 7 používame štruktúru `sockaddr_un`. Tretím parametrom je veľkosť štruktúry. Táto funkcia zabezpečí prepojenie medzi adresou soketu v súborovom systéme a súborovým deskriptorom.

Takto vytvorené sokety je možné použiť dvomi spôsobmi a to buď ako server alebo ako klient. Jednoduchý server s použitím soketov môžeme vidieť na ukážke kódu B.9. Na tejto ukážke kódu môžeme vidieť vytvorenie soketu a následne použitie funkcie `listen`, ktorej prvý parameter je deskriptor soketu a druhý parameter definuje maximálny počet požiadavok, ktoré môžu prísť pokiaľ nebude volaná funkcia `accept`.

Funkcia `accept` je blokovacia funkcia, ktorá blokuje vykonávanie programu⁹ pokiaľ sa nevytvorí spojenie. Argumenty tejto funkcie sú `sockfd`, `addr` a `addrlen`. `sockfd` je súborový deskriptor na ktorom akceptuje pripojenie, `addr` predstavujú štruktúru do ktorej bude uložená adresa klienta, ktorý sa pripojil a `addrlen` predstavuje veľkosť tejto štruktúry. Návratová hodnota funkcie je nový súborový deskriptor, ktorý bude slúžiť na komunikáciu s pripojeným klientom.

Takto vytvorený súborový deskriptor je možné použiť na zapisovanie alebo čítanie. Na čítanie používame funkciu `recv`. Táto funkcia je taktiež blokovacia a čaká pokiaľ nepríjme správu. Prvý argument funkcie je suborový deskriptor `sockfd`, druhým argumentom je `buf`, úložisko do ktorého bude uložená správa s dĺžkou maximálne `len`, čo je tretí parameter. Posledným parametrom tejto funkcie sú príznaky, ktoré pre jednoduchosť preskočíme. Funkcia `send` má rovnakú signatúru len s rozdielom že dáta ktoré sa nachádzajú v úložisku `buf` sú odoslané do soketu.

Na druhej strane klient, ktorý chce odosielať a prijímať dáta cez soket potrebuje vytvoriť soket a následne zavolať funkciu `connect`. Táto funkcia má rovnakú signatúru ako funkcia `bind` a zabezpečuje pripojenie súborového deskriptora k soketu, ktorý je definovaný cestou `sun_path` v štruktúre `sockaddr`, ktorá je druhým argumentom funkcie. Takto pripojený soket je možné používať pomocou vyššie spomínaných funkcií `send` a `recv`. Vytvorenie a používanie takéhoto klienta môžeme vidieť na ukážke kódu B.10.[16]

⁹Ak nie je definovaný príznak `SOCK_NONBLOCK`

2 Medusa

V 90. rokoch 20. storočia sa na Fakulte Elektrotechniky a Informatiky zrodil bezpečnostný systém Medusa. Tento bezpečnostný systém od svojho počiatku prešiel niekoľkými výraznými zmenami a v súčasnej podobe predstavuje bezpečnostnú politiku, ktorá funguje nad rámec základnej bezpečnosti v Linux (oprávnenia a skupiny). Tento bezpečnostný systém na svoje plnohodnotné fungovanie vyžaduje autorizačný systém, ktorý rozhoduje o povolení alebo zamietnutí nejakej akcie. Fungovanie tohto bezpečnostného riešenia v jadre systému Linux umožňuje LSM framework, ktorý využívajú aj ostatné bezpečnostné riešenia ako napríklad SELinux. Princíp fungovania ako aj jednotlivé pojmy si vysvetlíme v nasledovných kapitolách.

2.1 LSM framework

Linux Security Module (LSM) framework poskytuje mechanizmus, ktorý umožňuje pripojiť nové bezpečnostné rozšírenie do jadra. Avšak v skutočnosti nejde o modul ako by sa mohlo z názvu zdať, nakoľko rozšírenia, ktoré chceme zakomponovať do jadra je potrebné vybrať počas kompilácie jadra za pomoci konfigurácie a možnosti *CONFIG_DEFAULT_SECURITY*. Takto zakomponované bezpečnostné riešenie je následne možné vybrať počas načítavania jadra pomocou argumentu *security=...* v príkazovom riadku jadra.[18]

Bezpečnostné riešenia, ktoré chcú implementovať nejakú bezpečnostnú politiku sú závislé od funkcií, ktoré LSM framework poskytuje. Tieto funkcie sú definované v hlavíčkovom súbore *./include/linux/lsm_hooks.h* a predstavujú vstupný bod kde môže bezpečnostné riešenie definovať ľubovoľnú bezpečnostnú politiku. Tieto funkcie sú volané pri rôznych udalostiach, ktoré sa v jadre Linuxu vyskytnú. V prípade že bezpečnostné riešenie chce implementovať obsluhu takejto funkcie je potrebné použiť makro *LSM_HOOK_INIT*, ktorého prvý parameter je názov funkcie LSM frameworku a druhý parameter je obslužná funkcia v nami definovanom bezpečnostnom riešení.[19] Tieto obslužné funkcie majú návratovú hodnotu typu *int*, ktorá môže nadobudnúť nasledovné hodnoty:

- 0 udalosť, ktorá nastala, bezpečnostný mechanizmus povoľuje
- !0 napríklad *-EPERM* udalosť, ktorá nastala, bezpečnostný mechanizmus zakazuje

Bolo by veľmi obtiažne vytvárať bezpečnostnú politiku, keby si vybrané bezpečnostné riešenie nemohlo ukladať vlastné/interné dáta k prislúchajúcim objektom, ktoré bezpeč-

nostný systém ovplyvňuje. Preto, každá štruktúra, ktorá nejakým spôsobom vystupuje v LSM funkciách obsahuje položku `void *security`, táto položka predstavuje ukazovateľ na ľubovoľnú štruktúru, ktorú si definuje bezpečnostné riešenie. Konkrétne bezpečnostné riešenie má za úlohu alokovať pamäť pre túto štruktúru ako aj priradiť hodnotu tejto premennej.

2.2 Autorizačný server

Autorizačný server je špecifická entita pre bezpečnostné riešenie Medusa a predstavuje rozhodovací prvok. Ide o program, ktorý pracuje v užívateľskom prostredí a prostredníctvom soketu a definovaného komunikačného protokolu **prijíma, rozhoduje a vracia** odpoveď do jadra systému. Rozhodovanie autorizačného servera je vykonávané na základe konfiguračného súboru, ktorý definuje čo je povolené a čo nie. Výhodou takéhoto oddeleného rozhodovacieho prvku je možnosť použitia jedného autorizačného serveru pre viacero systémov s bezpečnostným systémom Medusa. Momentálne existujú dve implementácie autorizačného serveru:

- *Constable* - implementovaný v jazyku C, syntax konfiguračného súboru podobná jazyku C, ťažko čitateľný zdrojový kód
- *mYstable* - implementovaný v jazyku Python, konfiguračný súbor v jazyku Python

2.3 Princíp fungovania

Úlohou bezpečnostného systému Medusa v procese rozhodovania je zachytiť udalosť, ktorá sa v jadre systému vykonala a následne túto udalosť spracovať a vytvoriť štruktúry, ktoré sú definované v komunikačnom protokole. Pre lepšie pochopenie fungovania systému Medusa je potrebné si popísať dve základné entity, ktoré sú ***k-objekt*** a **typy prístupu**.

2.3.1 k-objekt

K-objekt je štruktúra systému Medusa, ktorá v sebe nesie informácie o nejakej internej štruktúre jadra ako napríklad *inode* alebo *process*. Rozlišujeme dva typy štruktúr:

- subjekt - ide o *k-objekt*, ktorý vykonáva nejakú operáciu
- objekt - ide o *k-objekt*, nad ktorým je vykonávaná nejaká operácia

Na definovanie typu sa používajú makrá *MEDUSA_SUBJECT_VARS* a *MEDUSA_OBJECT_VARS* ktoré rozširujú *k-objekt* o interné položky *Medusy*. [20] *K-objekt* môže obsahovať rôzne položky, ktoré z hľadiska konfigurácie rozhodovania považujeme za dôležité. Napríklad *k-objekt* procesu obsahuje položky, ktoré sú odvodené z internej štruktúry jadra *task_struct*

a tieto položky môžeme vidieť na definícii štruktúry B.11. Z definície môžeme vidieť, že tento *k-objekt* môže vystupovať ako objekt ale aj ako subjekt. Takto definovanej štruktúre je možné priradiť operácie, ktoré autorizačný server môže potrebovať. Poznáme 2 typy operácií:

- ***fetch*** - operácie, pri ktorej sú prenesené dáta z jadra do *k-objektu*
- ***update*** - operácie, pri ktorej je aktualizovaná interná štruktúra jadra dátami *k-objektu*

Tieto operácie umožňujú autorizačnému serveru meniť stav systému alebo sa informovať o zmene dát v jadre.

2.3.2 Typy prístupov

Typy prístupov v *Meduse* predstavujú obsluhu pre jednotlivé funkcie LSM frameworku. *Typ prístupu* definuje subjekt a objekt(*k-objektu*), ktoré vystupujú pri rozhodovaní. Na definovanie sa používa makro `MED_ACCTYPE`, ktoré môžeme vidieť na ukážke kódu B.17. Ďalej *typ prístupu* definuje extra dáta, ktoré sa nevzťahujú ani ku subjektu ani ku objektu ale ku konkrétnej udalosti, ktorá sa uskutočnila. A v neposlednom rade *typ prístupu* má za úlohu vytvorenie a konverziu dát jadra na *k-objekt* a naspäť. Konverzia dát nastáva pri nasledovných udalostiach:

- vyvolá sa LSM *hook* a Medusa potrebuje komunikovať s autorizačným serverom(štruktúra jadra -> *k-objekt*)
- nastane udalosť *update*, ktorú vyvolal autorizačný server(*k-objekt* -> štruktúra jadra)
- nastane udalosť *fetch*, ktorú vyvolal autorizačný server(štruktúra jadra -> *k-objekt*)

```
MED_ACCTYPE(typ_typu_pristupu, "nazov_typu_pristupu", typ_subjektu, "nazov_subjektu", typ_objektu,
    "nazov_objektu");
```

Listing 8: Bezpečnostná štruktúra Medusy pre IPC

Medusa obsahuje aj špeciálny *typ prístupu*, ktorý sa nazýva *udalosť*. *Udalosti* narozdiel od *typu prístupu* nereagujú na *lsm* framework, ale *udalosť* je volaná interne v rámci *Medusy* a slúži na inicializáciu *k-objektu* v autorizačnom servery.[20] *Udalosti* sú volané vždy pred *typom prístupu* aby sa zabezpečilo, že *typ prístupu* bude volaný s validným *k-objektom*.

Typy prístupov majú taktiež za úlohu kontrolovať prienik virtuálnych svetov. **Virtuálne svety** predstavujú skupiny do ktorých je možné zaradiť *k-objekty*. V prípade že sa dva rôzne *k-objekty* nachádzajú v rôznych virtuálnych svetoch je možné automaticky operáciu zamietnuť. Priradovanie *k-objektov* do virtuálnych svetov sa uskutočňuje na základe

konfigurácie. Virtuálne svety taktiež zrýchľujú rozhodovanie nakoľko v prípade rôznych virtuálnych svetov je možné bez zásahu autorizačného serveru rozhodnúť.

typ prístupu po konverziách a kontrolách virtuálnych svetov má za úlohu volať funkciu vyššej vrstvy, ktoré sa postará o komunikáciu s autorizačným serverom a rozhodnutie. Na volanie tejto funkcie sa používa makro `MED_DECIDE`, ktoré má nasledovné parametre:

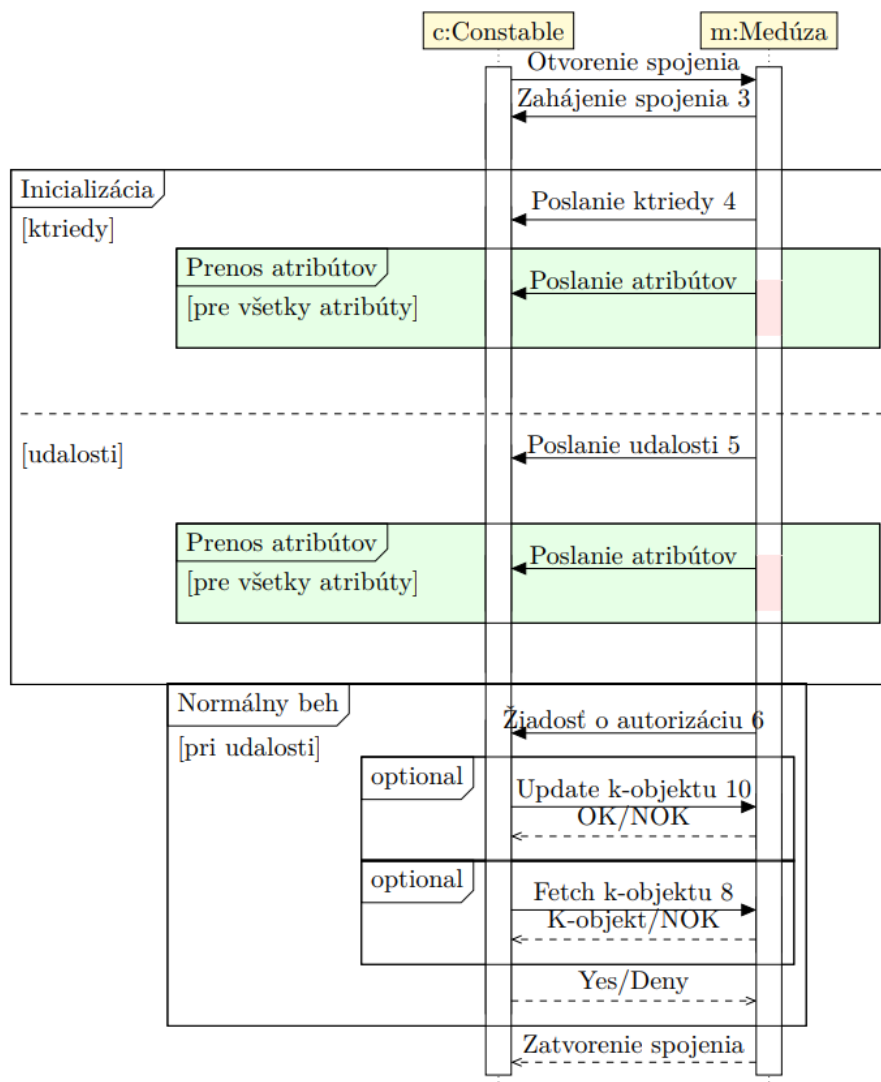
- *typ štruktúry*, ktorá definuje prístupu
- ukazovateľ na inštanciu štruktúry, ktorá definuje prístup
- ukazovateľ na inštanciu štruktúry, ktorá definuje subjekt
- ukazovateľ na inštanciu štruktúry, ktorá definuje objekt

Návratová hodnota tohto makra je rozhodnutie, ktoré vrátia vyššie vrstvy ide však väčšinou o odpoveď od autorizačného servera.

2.4 Architektúra

Systém *Medusa* sa skladá z 5 vrstiev a ich funkcie sú nasledovné:

- L0 - registrácia funkcií *lsm* frameworku pre skorú inicializáciu, táto vrstva zabezpečuje správnu inicializáciu systémových štruktúr pri štarte systému, tejto problematike sa venuje [21]
- L1 - registrácia všetkých funkcií *lsm* frameworku, teda aj tých ktoré neboli registrované vo vrstve L1
- L2 - *typy prístupov, udalosti a k-objekty*
- L3 - registrovanie a odregistrovanie entít z vrstvy L2
- L4 - komunikačná vrstva, ktorá definuje aj komunikačný protokol, ktorý je možné vidieť na obrázku 2, tento protokol je detailnejšie popísaný v diplomovej práci *ME-DÚZA DS9*[20]



Obr. 2: Schéma komunikačného protokolu[20]

3 Implementácia

V predchádzajúcich kapitolách sme si popísali entity (štruktúry, funkcie), ktoré je potrebné implementovať v prípade, že chceme rozšíriť bezpečnostné riešenie o ďalší LSM *hook*. Prvým krokom bolo zdefinovať si LSM *hook*, ktoré chceme implementovať. Vybrali sme si IPC mechanizmy semafor, zdieľanú pamäť a fronty správ, ktoré zdieľajú podobnú logiku a ich štruktúry v jadre majú taktiež podobnú kostru. K týmto IPC mechanizmom prislúchajú LSM funkcie, ktoré vidíme na ukážke kódu B.12.

3.1 Bezpečnostná štruktúra

Ďalším krokom bolo vytvorenie bezpečnostnej štruktúry, ktorá sa ukladá do položky `security_s`, ktorá sa nachádza v štruktúre `kern_ipc_perm`, ktorú sme si popísali v kapitole 1.3.2. Nami vytvorenú bezpečnostnú štruktúru môžeme vidieť na ukážke štruktúry 9 a jej definícia sa nachádza v súbore `include/linux/medusa/l1/ipc.h`. Táto štruktúra obsahuje položku `ipc_class`, ktorá definuje o aký typ IPC mechanizmu ide. Táto položka môže nadobúdať hodnoty 0, 1 alebo 2, ktoré sú definované makrami v rovnakom súbore. Taktiež sa v tejto štruktúre definujú makrá `MEDUSA_SUBJECT_VARS` a `MEDUSA_OBJECT_VARS` ako pri k-objekte, ktoré sme si popísali v kapitole 2.3.1.

```
#define MED_IPC_SEM 0
#define MED_IPC_MSG 1
#define MED_IPC_SHM 2
#define MED_IPC_UNDEFINED 3

struct medusa_l1_ipc_s {
    unsigned int ipc_class;
    MEDUSA_SUBJECT_VARS;
    MEDUSA_OBJECT_VARS;
};
```

Listing 9: Bezpečnostná štruktúra Medusy pre IPC

Takto zadanú bezpečnostnú štruktúru je potrebné alokovať načo nám slúžia LSM *hooky* `*_alloc_security` a na uvoľnenie pamäte `_free_security` tieto *hooky* môžeme vidieť v hornej časti výpisu B.12. Aj keď tieto funkcie majú rôzne parametre, je možné z týchto parametrov získať položku `kern_ipc_perm`, ktorú využívame v pomocnej funkcii `medusa_l1_ipc_alloc_security`, ktorej prvý parameter je práve táto štruktúra a druhým parametrom je typ IPC mechanizmu. Príklad použitia pomocnej funkcie môžete vidieť na ukážke kódu 10. Avšak táto pomocná funkcia nepokrýva všetky alokácie, nakoľko LSM framework obsahuje aj funkciu `msg_msg_alloc_security`, ktorá má argument typu `msg_msg`, ktorý neobsahuje štruktúru `kern_ipc_perm` ale bezpečnostná štruktúra sa

nachádza priamo v tejto štruktúre teda `msg_msg->security`. Na uvoľnenie pamäte je tiež vytvorená spoločná funkcia, ktorú môžete vidieť na ukážke kódu B.14. Táto funkcia obsluhuje všetky LSM *hooky* s výnimkou `medusa_l1_msg_msg_free_security`, ktorá má rovnaké obmedzenia, ktoré sme si spomenuli vyššie, ako funkcia na alokovanie.

```
static int medusa_l1_msg_queue_alloc_security(struct msg_queue *msq)
{
    return medusa_l1_ipc_alloc_security(&msq->q_perm, MED_IPC_MSG);
}
```

Listing 10: Príklad použitia pomocnej funkcie na alokovanie bezpečnostnej štruktúry

3.2 Vytvorenie *k-objektu*

Pri definovaní *k-objektu* sme narazili na niekoľko obmedzení, ktoré sa vynorili až v neskorších fázach implementácie a preto sa aj definícia *k-objektu* menila v priebehu vývoja. Prvý priamočiary návrh bolo vytvoriť 3 rôzne *k-objekty* pre SEM, SHM a MSG. Toto riešenie však prinášalo so sebou veľké množstvo kódu nakoľko tieto 3 mechanizmy majú dve operácie `get` a `ctl`, ktoré pri svojich volaniach vyvolávajú LSM *hooky*, ktoré majú signatúry, ktoré môžeme vidieť na ukážke kódu 11.

```
//get
int medusa_l1_[msg_queue/shm/sem]_associate(struct [msg_queue/shmid_kernel/sem_array] *obj, int semflg)
//ctl
int medusa_l1_[msg_queue/msg/shm_shm/sem_sem]ctl(struct [msg_queue/shmid_kernel/sem_array] *obj, int cmd)
```

Listing 11: LSM *hooky* pre operácie `get` a `ctl`

Keby sme pre každý mechanizmus vytvorili samostatný *k-objekt* a chceli vytvoriť *typ prístupu*, pre tieto dve spomínané operácie vyžadovalo by si to v konečnom dôsledku 6 rôznych *typov prístupu*, keďže *typ prístupu* je v kóde spätý s typom *k-objektu* ako môžeme vidieť v kapitole 2.3.2. Pre tento fakt sme sa rozhodli vytvoriť 4. *k-objekt*, ktorý predstavuje jednotný *k-objekt*, ktorý sa bude využívať pri *typoch prístupu* a teda pre vyššie spomínané operácie budeme definovať len 2 *typy prístupu*. Všeobecný *k-objekt* môžeme vidieť na ukážke kódu 12.

```
struct ipc_kobject {
    unsigned char data[max_simple(max_simple(sizeof(struct ipc_sem_kobject), sizeof(struct
        ipc_shm_kobject)), sizeof(struct ipc_msg_kobject))];
    MEDUSA_OBJECT_VARS;
};
```

Listing 12: Všeobecný *k-objekt*

Tento *k-objekt* obsahuje položku `data`. Ide o bajtové pole, ktoré má z definície veľkosť najväčšieho z konkrétnych *k-objektov*. Definíciu vyššie spomínaného *k-objektu* ako aj defi-

niacie konkrétnych *k-objektov* je možné nájsť v hlavičkovom súbore `security/medusa/12/kobject_ipc_common.h`.

Konkrétne *k-objekty* (pre SEM, SHM a MSG) obsahujú nasledovné položky:

- `ipc_class` - definuje typ mechanizmu
- `ipc_perm` - predstavuje internú štruktúru Medusy, `medusa_ipc_perm`, ktorá je definovaná v `include/linux/medusa/11/ipc.h`. Obsahuje jednoduché položky štruktúry `kern_ipc_perm`, ktorá bola popísaná v kapitole 1.3.2 a v štruktúre je definovaná pomocou makra `MEDUSA_IPC_VARS`.
- `MEDUSA_OBJECT_VARS` - makro, ktoré sme si bližšie popísali v kapitole 2.3.1

Konkrétne *k-objekty* môžu obsahovať aj špecifické údaje pre konkrétny mechanizmus, napríklad *k-objekt* `ipc_sem_kobject` obsahuje položku `sem_nsems`, v ktorej je uchovaný počet počítadiel v skupine semaforu.

3.2.1 Konverzné funkcie

Takto vytvoreným *k-objektom* je potrebné definovať konverzné funkcie medzi štruktúrou jadra, v našom prípade ide o štruktúry `msg_queue`, `sem_array` a `shmid_kernel` na štruktúry Medusy, ktoré sú `ipc_msg_kobject`, `ipc_sem_kobject` a `ipc_shm_kobject`. Avšak medzi LSM funkciami sa nachádza aj `ipc_permission`, ktorej signatúru môžeme vidieť na ukážke kódu 13.

```
static int medusa_11_ipc_permission(struct kern_ipc_perm *ipcp, short flag)
```

Listing 13: Signatúra obslužnej funkcie pre LSM *hook* `ipc_permission`

Ako môžeme vidieť táto funkcia obsahuje argument `kern_ipc_perm`, ktorý nepredstavuje konkrétnu štruktúru nejakého z mechanizmov ale ide len o podštruktúru. Na základe tohto faktu sme sa preto rozhodli pri konverzných funkciách na vstupe pracovať z touto štruktúrou a zabezpečiť tým kompatibilitu aj pre tento špecifický LSM *hook*. *K-objekty* pre SEM, SHM a MSG obsahujú vlastné konverzné funkcie v súbore *k-objektu*, teda pre semafor sa nachádzajú tieto funkcie v súbore `security/medusa/12/kobject_ipc_sem.c`. Tieto funkcie majú nasledovné názvy:

- `ipc_[typ]_kern2kobj` - konverzná funkcia zo štruktúry jadra na štruktúru Medusy
- `ipc_[typ]_kobj2kern` - konverzná funkcia zo štruktúry Medusy na štruktúru jadra

Pričom `[typ]` môže byť `sem`, `shm` alebo `msg`. Tieto funkcie sa pre každý mechanizmus líšia hlavne v položkách, ktoré sa kopírujú preto si princíp ukážeme len na semaforochoch. Na ukážke kódu 14 môžeme vidieť konkrétnu implementáciu.

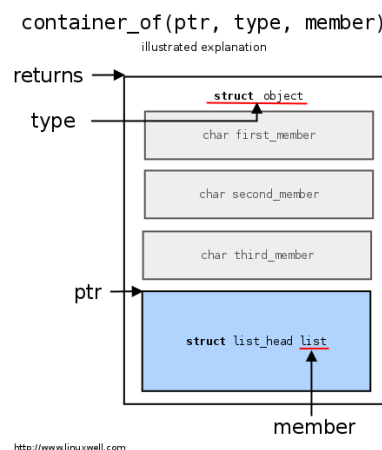
```

1 static struct ipc_sem_kobject storage;
2
3 void * ipc_sem_kern2kobj(struct kern_ipc_perm * icp)
4 {
5     struct medusa_l1_ipc_s* security_s;
6     struct sem_array * sem_array;
7
8     security_s = (struct medusa_l1_ipc_s*) icp->security;
9     sem_array = container_of(icp, struct sem_array, sem_perm);
10
11     memset(&storage, '\0', sizeof(struct ipc_sem_kobject));
12
13     if(!security_s)
14         return NULL;
15
16     storage.ipc_class = security_s->ipc_class;
17     storage.sem_nsems = sem_array->sem_nsems;
18
19     COPY_WRITE_IPC_VARS(&(storage.ipc_perm), icp);
20     COPY_READ_IPC_VARS(&(storage.ipc_perm), icp);
21     COPY_MEDUSA_OBJECT_VARS(&storage, security_s);
22     return (void *)&storage;
23 }

```

Listing 14: Konverzná funkcia zo štruktúry jadra na štruktúru Medusy pre SEM

Na začiatku si deklarujeme pomocné premenné, ktoré následne inicializujeme. Prvou pomocnou premennou je bezpečnostná štruktúra, ktorú sme si popísali v kapitole 3.1 a druhou premennou je štruktúra, ktorá je hlavnou štruktúrou pre konkrétny IPC mechanizmus. Dôležitým makrom, ktoré sa tu používa je makro `container_of`. Toto makro na základe ukazovateľa `ptr` na nejakú položku `member` dokáže vrátiť adresu na kontajner, ktorý má typ `type`. Princíp fungovania môžeme vidieť aj na obrázku 3. Po-



Obr. 3: Princíp fungovania `container_of` makra [22]

mocou tohto makra dokážeme získať hlavnú štruktúru IPC mechanizmu len za pomoci

štruktúry `kern_ipc_perm`. Ďalším krokom je vynulovanie pamäte premennej `storage`. Táto premenná slúži na uloženie *k-objektu* v pamäti, pokiaľ sa nepošle autorizačnému serveru. Na ďalšom riadku sa nachádza kontrola bezpečnostnej štruktúry, ktorá by nemala byť `NULL`, ak však takýto prípad nastane tak funkcia vráti `NULL` a vyššie funkcie ktoré túto konverziu používajú musia túto možnosť ďalej ošetriť. Na riadku 16 a 17 sa do *k-objektu* prenesú údaje o type mechanizmu a špecifická vlastnosť SEM, ktorou je počet semaforov. Na riadkoch 19 a 20 sú použité pomocné makrá, ktoré sme si vytvorili len pre potreby IPC a slúžia na prekopírovanie položiek štruktúry `kern_ipc_perm`, ktoré sa v *k-objekte* nachádzajú v položke `ipc_perm`, ktorú sme si popísali vyššie v tejto kapitole. Implementácia týchto dvoch makier sa nachádza v hlavičkovom súbore `include/linux/medusa/l1/ipc.h`. Makro `COPY_READ_IPC_VARS` kopíruje položky, ktoré je možné len čítať a makro `COPY_WRITE_IPC_VARS` kopíruje položky, ktoré je možné čítať aj prepisovať v štruktúre jadra. Posledným makrom, ktoré sa pri konverzii používa je makro `COPY_MEDUSA_OBJECT_VARS`, ktoré kopíruje interné položky definované pre *objekt* z bezpečnostnej štruktúry do *k-objektu*. Návrátová hodnota tejto funkcie je `void` ukazovateľ na úložisko kde máme nový *k-objekt*.

```
medusa_answer_t ipc_sem_kobj2kern(struct medusa_kobject_s * ipck, struct kern_ipc_perm * ipcp)
{
    struct medusa_l1_ipc_s* security_s;
    struct ipc_sem_kobject * ipck_sem;

    ipck_sem = (struct ipc_sem_kobject *)ipck;
    security_s = (struct medusa_l1_ipc_s*) ipcp->security;

    COPY_WRITE_IPC_VARS(ipcp, &ipck_sem->ipc_perm);
    COPY_MEDUSA_OBJECT_VARS(security_s, ipck_sem);
    MED_MAGIC_VALIDATE(security_s);
    return MED_OK;
}
```

Listing 15: Konverzná funkcia zo štruktúry Medusy na štruktúru jadra pre SEM

Na ukážke kódu 15 môžeme vidieť funkciu s opačnou funkcionalitou a teda konverziu z *k-objektu* na štruktúru jadra. Pri takejto konverzii je potrebné si uvedomiť, ktoré položky štruktúry je možné v jadre systému upravovať. Pri IPC mechanizmoch sme sa inšpirovali operáciou *ctl*, ktorá taktiež má schopnosť meniť štruktúru jadra a práve táto operácia dokáže priamo aktualizovať nasledovné položky `uid`, `gid` a `mode`. Preto aj táto konverzná funkcia môže meniť len tieto tri položky a nemôže meniť napríklad pri semaforochoch položku `sem_nsems`, čo by mohlo spôsobiť nedefinované správanie. Na zmenu údajov v jadre používame vyššie spomínané makrá pričom na konci vykonávanie funkcie je použité makro `MED_MAGIC_VALIDATE`, ktorého úlohou je nastaviť internú položku Medusy, ktorá indikuje

že objekt je už zaregistrovaný v autorizačnom servery. Návrátovou hodnotou tejto funkcie je status `MED_OK`, ktorý indikuje úspešnú aktualizáciu.

Tieto konverzné funkcie pre SEM, SHM a MSG nie je možné priamo použiť v *type prístupu*, nakoľko ako sme písali v kapitole 3.2 vytvorili sme 4. všeobecný *k-objekt*. Kvôli tomuto faktu je vytvorená funkcia `ipc_kern2kobj`, ktorú môžeme vidieť na ukážke kódu 16. Táto funkcia na základe typu IPC mechanizmu, ktorý získa z bezpečnostnej štruktúry, rozhodne ktorú konkrétnu funkciu zavolať. Následne skopíruje pamäťovú oblasť kde je uložený *k-objekt* do položky `data` štruktúry `ipc_kobject`. Návrátová hodnota funkcie je 0 v prípade úspešnej konverzie v opačnom prípade -1.

```
int ipc_kern2kobj(struct ipc_kobject * ipck, struct kern_ipc_perm * ipcperm)
{
    struct medusa_ll_ipc_s* security_s;
    unsigned int ipc_class;

    security_s = ipc_security(ipcperm);
    ipc_class = security_s->ipc_class;
    switch(ipc_class){
        case MED_IPC_SEM: {
            struct ipc_sem_kobject *new_kobj;
            new_kobj = (struct ipc_sem_kobject *)ipc_sem_kern2kobj(ipcperm);
            memcpy(ipck->data, (unsigned char *)new_kobj, sizeof(struct ipc_sem_kobject));
            break;
        }
        ...
        default:
            printk("Unkown ipc_class\n");
            return -1;
    }
    return 0;
}
```

Listing 16: Konverzná funkcia zo štruktúry Medusy na štruktúru jadra

3.2.2 Operácie *fetch* a *update*

Pre plnohodnotnú funkcionálnosť bolo potrebné definovať operácie *fetch* a *update*, ktoré sme si vysvetlili v kapitole 2.3.1. Tieto operácie sa definujú *k-objektom* konkrétnych mechanizmov, pričom všeobecný *k-objekt* `ipc_kobject` tieto operácie nedefinuje čo treba zohľadniť pri konfigurácii autorizačného servera.

```
static struct medusa_kobject_s * ipc_sem_fetch(struct medusa_kobject_s * kobj)
{
    struct ipc_sem_kobject * ipc_kobj;
    struct medusa_kobject_s * new_kobj;
    ipc_kobj = (struct ipc_sem_kobject *)kobj;
    new_kobj = (struct medusa_kobject_s *)ipc_fetch(ipc_kobj->ipc_perm.id, ipc_kobj->ipc_class,
        ipc_sem_kern2kobj);
    return new_kobj;
}
```

```
}
```

Listing 17: Operácie `fetch` pre *k-objekt* semaforu

Na ukážke kódu 17 vidíme operáciu `fetch` *k-objektu* semaforu, ktorá v svojom tele volá funkciu `ipc_fetch`. Funkcia `ipc_fetch` je spoločná pre všetky 3 IPC *k-objekty* a jej kód môžeme vidieť na ukážke kódu B.15. V tejto spoločnej funkcii sa vykonávajú nasledovné operácie:

- získanie štruktúry `ipc_ids` podľa typu IPC mechanizmu
- získanie štruktúry `kern_ipc_perm` na základe identifikátora IPC mechanizmu
- konverzia štruktúr pomocou funkcii, ktoré sme si popísali v kapitole 3.2.1
- vrátenie nového *k-objektu*

```
static medusa_answer_t ipc_sem_update(struct medusa_kobject_s * kobj)
{
    struct ipc_sem_kobject * ipc_kobj;
    medusa_answer_t answer;
    ipc_kobj = (struct ipc_sem_kobject *)kobj;
    answer = ipc_update(ipc_kobj->ipc_perm.id, ipc_kobj->ipc_class, kobj, ipc_sem_kobj2kern);
    return answer;
}
```

Listing 18: Operácie `update` pre *k-objekt* semaforu

Operácia *update* funguje na rovnakom princípe ale odlišuje sa hlavne v uzamykacích mechanizmoch, ktoré sme tu museli použiť, nakoľko narozdiel od operácie *fetch*, táto operácia mení dáta v štruktúrach jadra a preto je potrebné pomocou rôznych zámkov zabezpečiť zachovanie integrity údajov. Funkciu `ipc_update` môžeme vidieť na ukážke kódu B.16 a jej použitie na ukážke kódu 18. Pri IPC je možné použiť niekoľko druhou zámkou avšak operácie ktoré pod týmito známkami vykonávame sú presne definované a sú nasledovné

- `rcu_read_lock/rcu_read_unlock`
 - počiatočné kontroly (povolenia, audity, ...)
 - len čítacie operácie, ktoré nevyžadujú atomicitu
- `ipc_lock_object/ipc_unlock_object`
 - čítacie operácie, ktoré vyžadujú atomicitu
 - aktualizácie údajov, ako napríklad `SET`, `RMID` príkazy a operácie špecifické pre mechanizmy

- `ids->rwsem`
 - vytváranie, odstraňovanie a iterácia cez existujúce objekty v IPC skupine
 - iterovanie cez súbory na ceste `/proc/sysvipc/`

Jadro systému Linux taktiež obsahuje aj operácie na zvýšenie(`ipc_rcu_getref`) a na zníženie(`ipc_put_getref`) počítadla referencii, ktoré zabráňuje tomu aby bol IPC objekt odstránený a zároveň používaný nejakým procesom. Na základe týchto poznatkov sme v operácii *fetch* použili len RCU uzamykací mechanizmus keďže vykonávame len čítacie operácie a pri operácií *update* používame systém RCU spoločne s funkciou `ipc_lock_object` nakoľko vykonávame aj zapisovanie do štruktúr jadra. Aby sme predišli nechcenému odstráneniu objektu zo systému, tak po získaní štruktúry taktiež zvyšujeme aj referenciu na tento objekt. Keďže pri operácii *update* nevytvárame, nemažeme ani neprechádzame objekty IPC, nebolo potrebné použiť zámok `ids->rwsem`.

3.3 Typy prístupov

Po tom ako sme si vytvorili potrebné *k-objekty*, bolo potrebné zadať *typy prístupov*. Definícia *typu prístupu* priamo nadväzuje na LSM *hook* avšak snažili sme sa LSM *hooky*, ktoré sú podobné zlúčiť pod jeden *typ prístupu*. Výsledkom sú nasledovné *typy prístupov*:

- `ipc_ctl` - združuje *hooky* pre operáciu `ctl`
- `ipc_associate` - združuje *hooky* pre operáciu `get`
- `ipc_msgrcv`
- `ipc_msgsnd`
- `ipc_permission`
- `ipc_semop`
- `ipc_shmat`

Každý z týchto *typov prístupu* má definovaný *object*, ktorým je aktuálny proces, ktorý získavame pomocou makra `current` a *subjektom* je IPC entita, ktorá je typu `ipc_kobject`. Taktiež každý *typ prístupu* obsahuje extra položku `ipc_class` aby bolo možné na strane autorizačného servera skonvertovať všeobecný *k-objekt* na konkrétny *k-objekt*. Rozdielom medzi týmito *typmi prístupov* sú extra dáta, ktoré sa pridávajú k *typu prístupu*, napríklad

pre `ipc_semop` sú to položky `sem_op`, `sem_num`, `sem_flg`, `alter`, ktoré definujú ako a nad čím sa má operácia vykonať. Pre jednotlivé *typy prístupov* je možné tieto položky nájsť v definícii štruktúry *typu prístupu* na začiatku súboru, ako môžeme vidieť na ukážke kódu B.17. Ďalej sa v tomto súbore nachádza hlavná funkcia, ktorá na začiatku validuje *k-objekt* procesu ako aj IPC *k-objekt*, nasleduje kontrola virtuálnych svetov, konverzia dátových štruktúr a na koniec rozhodovanie v podobe volania makra `MED_DECIDE`. Ostatné *typy prístupov* môžeme nájsť v priečinku `security/medusa/12/` a súboroch `acctype_ipc_*.c`.

Aby bolo možné kompletne implementovať *typy prístupov* bolo nevyhnutné vytvoriť *udalosť*, ktorá bude kontrolovať a validovať *k-objekty*. Preto sme vytvorili *udalosť* `getipc`, ktorá má za úlohu inicializovať interné položky a zaregistrovať objekt u autorizačného serveru. Takto zaregistrovaný objekt považujeme za validný a *typ prístupu* môže ďalej vykonávať rozhodovanie. *Typ prístupu* vyvoláva túto *udalosť* pomocou volania funkcie `medusa_ipc_validate`, ktorej návratová hodnota je typu `medusa_answer_t` a v prípade že nadobúda hodnotu `MED_OK` tak rozhodovanie pokračuje v opačnom prípade rozhodovanie nenastane a systému je vrátená hodnota, ktorá povoľuje systémové volanie. Ide o vlastnosť Medusy, ktorá v prípade chyby väčšinou systémové volanie povolí.

Pri implementácii *typov prístupu* sme narazili na problém pri systémovom volaní typu `ctl`, ktoré sme si popísali aj v kapitole 1.3.3. Toto systémové volanie môže byť použité s rôznymi prepínačmi a pri použití prepínača `IPC_INFO` alebo `[type]_INFO`, kde *type* predstavuje konkrétny IPC, nastáva situácia kedy do funkcie nevstupuje žiadny IPC objekt. Preto bolo nevyhnutné tento prípad pri *type prístupu* riešiť a v prípade že sa jedná o jeden zo spomínaných prepínačov tak sa pri rozhodovaní ako objekt posielal `NULL` a položka `ipc_class` v *type prístupu* nadobúda hodnotu `MED_IPC_UNDEFINED`. Toto je potrebné zohľadniť pri konfigurácii aby sa predišlo chybám spôsobením hodnotou `NULL`. Môže vystávať otázka či je nevyhnutné rozhodovať o systémovom volaní `ctl` pri takomto použití, teda keď používateľ zisťuje limity IPC mechanizmu. Avšak z pohľadu útočníka aj toto systémové volanie môže predstavovať vhodný bod na získanie cenných informácií o systéme, ktorý sa snaží skompromitovať a preto je nevyhnutné aj toto volanie kontrolovať.

Takto definované *typy prístupov* sme následne využili vo vrstve L1, kde sme v jednotlivých LSM *hookoch* zavolali príslušné funkcie. Príklad použitia *typu prístupu* môžeme vidieť na ukážke kódu 19.

```
static int medusa_l1_msg_queue_associate(struct msg_queue *msq, int msqflg)
{
    if(medusa_ipc_associate(&msq->q_perm, msqflg) == MED_NO)
        return -EPERM;
    return 0;
}
```

}

Listing 19: Volanie funkcie *typu prístupu* z vrstvy L1

3.4 Konfigurácia autorizačného servera

Po doplnení Medusy o ďalšie *typy prístupov* bolo potrebné vytvoriť aj konfiguráciu na strane autorizačného servera s ktorou by sme mohli vykonať testy a odskúšať funkčnosť. Pri našom testovaní sme použili autorizačný server *mYstable* a jeho konfiguráciu v jazyku Python, ktorú môžete vidieť na ukážke kódu B.18. Hlavným zdrojom kontroly v našom prípade bol výstup s programu **dmesg** a kontrola či sa jednotlivé udalosti vykonávajú a kontrolujú na strane autorizačného servera.

Záver

Conclusion is going to be where?

Here.

Zoznam použitej literatúry

1. RUSLING, David A. Interprocess Communication Mechanisms. 1996-1999. Dostupné tiež z: <https://www.tldp.org/LDP/tlk/ipc/ipc.html>.
2. *KILL(2) Linux Programmer's Manual*. 4.15. vyd. 2017. Dostupné tiež z: <http://man7.org/linux/man-pages/man2/kill.2.html>.
3. *SIGACTION(2) Linux Programmer's Manual*. 4.15. vyd. 2017. Dostupné tiež z: <http://man7.org/linux/man-pages/man2/sigaction.2.html>.
4. CESATI, Marco a BOVET, Daniel P. *Understanding the Linux Kernel: Third Edition*. Third. O'Reilly Media, 2005. ISBN 978-0596005658.
5. KERRISK, Michael. An introduction to Linux IPC. 2013. Dostupné tiež z: http://man7.org/conf/lca2013/IPC_Overview-LCA-2013-printable.pdf.
6. *MKFIFO(3) Linux Programmer's Manual*. 4.15. vyd. 2017. Dostupné tiež z: <http://man7.org/linux/man-pages/man3/mkfifo.3.html>.
7. GOLDT, Sven, MEER, Sven van der, BURKETT, Scott a WELSH, Matt. The Linux Programmer's Guide. 1995. Dostupné tiež z: <http://tldp.org/LDP/lpg/lpg.html>.
8. *MSGGET(2) Linux Programmer's Manual*. 4.15. vyd. 2017. Dostupné tiež z: <http://man7.org/linux/man-pages/man2/msgget.2.html>.
9. *MSGOP(2) Linux Programmer's Manual*. 4.15. vyd. 2017. Dostupné tiež z: <http://man7.org/linux/man-pages/man2/msgop.2.html>.
10. *MSGCTL(2) Linux Programmer's Manual*. 4.15. vyd. 2017. Dostupné tiež z: <http://man7.org/linux/man-pages/man2/msgctl.2.html>.
11. BHARADWAJ, Raghu. *Mastering Linux Kernel Development*. First. Packt Publishing, 2017. ISBN 978-1785883057.
12. *SEMGET(2) Linux Programmer's Manual*. 4.15. vyd. 2017. Dostupné tiež z: <http://man7.org/linux/man-pages/man2/semget.2.html>.
13. *SEMOP(2) Linux Programmer's Manual*. 4.15. vyd. 2017. Dostupné tiež z: <http://man7.org/linux/man-pages/man2/semop.2.html>.
14. *SEMCTL(2) Linux Programmer's Manual*. 4.15. vyd. 2017. Dostupné tiež z: <http://man7.org/linux/man-pages/man2/semctl.2.html>.
15. *SHMCTL(2) Linux Programmer's Manual*. 4.15. vyd. 2017. Dostupné tiež z: <http://man7.org/linux/man-pages/man2/shmctl.2.html>.

16. HALL, Brian "Beej Jorgensen". Beej's Guide to Unix IPC. 1.1.3. vyd. 2015. Dostupné tiež z: <http://beej.us/guide/bgipc/html/multi/index.html>.
17. *SOCKET(2) Linux Programmer's Manual*. 4.15. vyd. 2017. Dostupné tiež z: <http://man7.org/linux/man-pages/man2/socket.2.html>.
18. *Linux Security Module Usage*. 4.16.0. vyd. Dostupné tiež z: <https://www.kernel.org/doc/html/v4.16/admin-guide/LSM/index.html>.
19. MIHÁLIK, Viliam. *Implementácia ďalších systémových volaní do Medusy*. 2016. bachelor thesis. STU FEI. FEI-5382-72886.
20. KÁČER, Ján. *Medúza DS9*. 2014. diploma thesis. STU FEI. FEI-5384-64746.
21. MIHÁLIK, V., PLOSZEK, R., SMOLÁR, M., SMOLEŇ, M. a SÝS, P. Medusa tímový projekt. 2017.
22. PAZDERA, Radek. *The Magical container_of() Macro* [http://radek.io/2012/11/10/magical-container_of-macro/]. Online; accessed 30.4.2018.

Prílohy

A	Štruktúra elektronického nosiča	II
B	Algoritmus	III
C	Výpis sublineXIII

A Štruktúra elektronického nosiča

/CHANGELOG.md

- file describing changes made to FEIstyle

/example.tex

- main example *.tex* file for diploma thesis

/example_paper.tex

- example *.tex* file for seminar paper

/Makefile

- simply Makefile – build system

/fei.sublime-project

- is project file with build in Build System for Sublime Text 3

/img

- folder with images

/includes

- files with content

/bibliography.bib

- bibliography file

/attachmentA.tex

- this very file

B Algoritmus

```
struct kern_ipc_perm {
    spinlock_t    lock;
    bool          deleted;
    int           id;
    key_t         key;
    kuid_t        uid;
    kgid_t        gid;
    kuid_t        cuid;
    kgid_t        cgid;
    umode_t       mode;
    unsigned long seq;
    void          *security;
    struct rhash_head khnode;
    struct rcu_head rcu;
    refcount_t refcount;
} ____cacheline_aligned_in_smp __randomize_layout;
```

Listing B.1: Štruktúra kern_ipc_perm

```
struct msqid64_ds {
    struct ipc64_perm msg_perm;
    __kernel_time_t msg_stime; /* last msgsnd time */
#ifdef __BITS_PER_LONG != 64
    unsigned long __unused1;
#endif
    __kernel_time_t msg_rtime; /* last msgrcv time */
#ifdef __BITS_PER_LONG != 64
    unsigned long __unused2;
#endif
    __kernel_time_t msg_ctime; /* last change time */
#ifdef __BITS_PER_LONG != 64
    unsigned long __unused3;
#endif
    __kernel_ulong_t msg_cbytes; /* current number of bytes on queue */
    __kernel_ulong_t msg_qnum; /* number of messages in queue */
    __kernel_ulong_t msg_qbytes; /* max number of bytes on queue */
    __kernel_pid_t msg_lspid; /* pid of last msgsnd */
    __kernel_pid_t msg_lrpid; /* last receive pid */
    __kernel_ulong_t __unused4;
    __kernel_ulong_t __unused5;
};
```

Listing B.2: Štruktúra msqid64_ds

```
struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first; /* first message on queue, unused */
    struct msg *msg_last; /* last message in queue, unused */
    __kernel_time_t msg_stime; /* last msgsnd time */
};
```

```

__kernel_time_t msg_rtime; /* last msgrcv time */
__kernel_time_t msg_ctime; /* last change time */
unsigned long msg_lcbytes; /* Reuse junk fields for 32 bit */
unsigned long msg_lqbytes; /* ditto */
unsigned short msg_cbytes; /* current number of bytes on queue */
unsigned short msg_qnum; /* number of messages in queue */
unsigned short msg_qbytes; /* max number of bytes on queue */
__kernel_ipc_pid_t msg_lspid; /* pid of last msgsnd */
__kernel_ipc_pid_t msg_lrpid; /* last receive pid */
};

```

Listing B.3: Štruktúra msqid_ds

```

struct msginfo {
    int msgpool;
    int msgmap;
    int msgmax;
    int msgmnb;
    int msgmni;
    int msgssz;
    int msgtql;
    unsigned short msgseg;
};

```

Listing B.4: Štruktúra msginfo

```

struct semid_ds {
    struct ipc_perm sem_perm; /* permissions .. see ipc.h */
    __kernel_time_t sem_otime; /* last semop time */
    __kernel_time_t sem_ctime; /* create/last semctl() time */
    struct sem *sem_base; /* ptr to first semaphore in array */
    struct sem_queue *sem_pending; /* pending operations to be processed */
    struct sem_queue **sem_pending_last; /* last pending operation */
    struct sem_undo *undo; /* undo requests on this array */
    unsigned short sem_nsems; /* no. of semaphores in array */
};

```

Listing B.5: Štruktúra semid_ds

```

struct semid64_ds {
    struct ipc64_perm sem_perm; /* permissions .. see ipc.h */
    __kernel_time_t sem_otime; /* last semop time */
    __kernel_ulong_t __unused1;
    __kernel_time_t sem_ctime; /* last change time */
    __kernel_ulong_t __unused2;
    __kernel_ulong_t sem_nsems; /* no. of semaphores in array */
    __kernel_ulong_t __unused3;
    __kernel_ulong_t __unused4;
};

```

Listing B.6: Štruktúra semid64_ds

```

union semun {
    int val; /* value for SETVAL */
    struct semid_ds __user *buf; /* buffer for IPC_STAT & IPC_SET */
    unsigned short __user *array; /* array for GETALL & SETALL */
    struct seminfo __user *__buf; /* buffer for IPC_INFO */
    void __user *__pad;
};

```

Listing B.7: Štruktúra semun

```

struct shmid_kernel /* private to the kernel */
{
    struct kern_ipc_perm shm_perm;
    struct file *shm_file;
    unsigned long shm_nattch;
    unsigned long shm_segsz;
    time64_t shm_atim;
    time64_t shm_dtim;
    time64_t shm_ctim;
    pid_t shm_cprid;
    pid_t shm_lprid;
    struct user_struct *mlock_user;

    /* The task created the shm object. NULL if the task is dead. */
    struct task_struct *shm_creator;
    struct list_head shm_clist; /* list by creator */
} __randomize_layout;

```

Listing B.8: Štruktúra shmid_kernel

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SOCK_PATH "echo_socket"

int main(void)
{
    int s, s2, t, len;
    struct sockaddr_un local, remote;
    char str[100];

    if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    local.sun_family = AF_UNIX;
    strcpy(local.sun_path, SOCK_PATH);

```

```

unlink(local.sun_path);
len = strlen(local.sun_path) + sizeof(local.sun_family);
if (bind(s, (struct sockaddr *)&local, len) == -1) {
    perror("bind");
    exit(1);
}

if (listen(s, 5) == -1) {
    perror("listen");
    exit(1);
}

for(;;) {
    int done, n;
    printf("Waiting for a connection...\n");
    t = sizeof(remote);
    if ((s2 = accept(s, (struct sockaddr *)&remote, &t)) == -1) {
        perror("accept");
        exit(1);
    }

    printf("Connected.\n");

    done = 0;
    do {
        n = recv(s2, str, 100, 0);
        if (n <= 0) {
            if (n < 0) perror("recv");
            done = 1;
        }

        if (!done)
            if (send(s2, str, n, 0) < 0) {
                perror("send");
                done = 1;
            }
    } while (!done);

    close(s2);
}

return 0;
}

```

Listing B.9: Ukážka použitia soketu

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

```

```

#define SOCK_PATH "echo_socket"

int main(void)
{
    int s, t, len;
    struct sockaddr_un remote;
    char str[100];

    if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    printf("Trying to connect...\n");

    remote.sun_family = AF_UNIX;
    strcpy(remote.sun_path, SOCK_PATH);
    len = strlen(remote.sun_path) + sizeof(remote.sun_family);
    if (connect(s, (struct sockaddr *)&remote, len) == -1) {
        perror("connect");
        exit(1);
    }

    printf("Connected.\n");

    while(printf("> "), fgets(str, 100, stdin), !feof(stdin)) {
        if (send(s, str, strlen(str), 0) == -1) {
            perror("send");
            exit(1);
        }

        if ((t=recv(s, str, 100, 0)) > 0) {
            str[t] = '\0';
            printf("echo> %s", str);
        } else {
            if (t < 0) perror("recv");
            else printf("Server closed connection\n");
            exit(1);
        }
    }

    close(s);

    return 0;
}

```

Listing B.10: Ukážka použitia soketu na strane klienta

```

struct process_kobject { /* was: m_proc_inf */

    pid_t pid, parent_pid, child_pid, sibling_pid;
    struct pid* pgrp;
    kuid_t uid, euid, suid, fsuid;
    kgid_t gid, egid, sgid, fsgid;

```

```

char cmdline[128];

kuid_t luid;
kernel_cap_t ecap, icap, pcap;
MEDUSA_SUBJECT_VARS;
MEDUSA_OBJECT_VARS;
__u32 user;
#ifdef CONFIG_MEDUSA_SYSCALL
    /* FIXME: this is wrong on non-i386 architectures */

    /* bitmap of syscalls, which are reported */
    unsigned char med_syscall[NR_syscalls / (sizeof(unsigned char) * 8)];
#endif
};

```

Listing B.11: K-objekt procesu

```

//alokovanie
LSM_HOOK_INIT(msg_msg_alloc_security, medusa_l1_msg_msg_alloc_security),
LSM_HOOK_INIT(msg_msg_free_security, medusa_l1_msg_msg_free_security),
LSM_HOOK_INIT(msg_queue_alloc_security, medusa_l1_msg_queue_alloc_security),
LSM_HOOK_INIT(msg_queue_free_security, medusa_l1_msg_queue_free_security),
LSM_HOOK_INIT(shm_alloc_security, medusa_l1_shm_alloc_security),
LSM_HOOK_INIT(shm_free_security, medusa_l1_shm_free_security),
LSM_HOOK_INIT(sem_alloc_security, medusa_l1_sem_alloc_security),
LSM_HOOK_INIT(sem_free_security, medusa_l1_sem_free_security),

LSM_HOOK_INIT(ipc_permission, medusa_l1_ipc_permission),
LSM_HOOK_INIT(ipc_getsecid, medusa_l1_ipc_getsecid),
LSM_HOOK_INIT(msg_queue_associate, medusa_l1_msg_queue_associate),
LSM_HOOK_INIT(msg_queue_msgctl, medusa_l1_msg_queue_msgctl),
LSM_HOOK_INIT(msg_queue_msgsnd, medusa_l1_msg_queue_msgsnd),
LSM_HOOK_INIT(msg_queue_msgrcv, medusa_l1_msg_queue_msgrcv),
LSM_HOOK_INIT(shm_associate, medusa_l1_shm_associate),
LSM_HOOK_INIT(shm_shmctl, medusa_l1_shm_shmctl),
LSM_HOOK_INIT(shm_shmat, medusa_l1_shm_shmat),
LSM_HOOK_INIT(sem_associate, medusa_l1_sem_associate),
LSM_HOOK_INIT(sem_semctl, medusa_l1_sem_semctl),
LSM_HOOK_INIT(sem_semop, medusa_l1_sem_semop),

```

Listing B.12: LSM hooky pre IPC

```

int medusa_l1_ipc_alloc_security(struct kern_ipc_perm *ipcp, unsigned int ipc_class)
{
    struct medusa_l1_ipc_s *med;

    med = (struct medusa_l1_ipc_s*) kmalloc(sizeof(struct medusa_l1_ipc_s), GFP_KERNEL);
    if (med == NULL)
        return -ENOMEM;

    med->ipc_class = ipc_class;
    ipcp->security = med;
    return 0;
}

```

```
}
```

Listing B.13: Alokovanie bezpečnostnej štruktúry

```
void medusa_l1_ipc_free_security(struct kern_ipc_perm *ipcp)
{
    struct medusa_l1_ipc_s *med;

    if(ipcp->security != NULL) {
        med = ipcp->security;
        ipcp->security = NULL;
        kfree(med);
    }
}
```

Listing B.14: Uvoľnenie pamäte bezpečnostnej štruktúry

```
void * ipc_fetch(unsigned int id, unsigned int ipc_class, void * (*ipc_concrete_kern2kobj)(struct
kern_ipc_perm *))
{
    struct kern_ipc_perm *ipcp;
    struct ipc_ids *ids;
    void *new_kobj = NULL;

    ids = medusa_get_ipc_ids(ipc_class);
    if(!ids)
        goto out_err;

    rcu_read_lock();

    ipcp = ipc_obtain_object_check(ids, id);
    if(IS_ERR(ipcp) || !ipcp)
        goto out_unlock0;

    new_kobj = ipc_concrete_kern2kobj(ipcp);
out_unlock0:
    rcu_read_unlock();
out_err:
    return new_kobj;
}
```

Listing B.15: Operácie fetch

```
medusa_answer_t ipc_update(unsigned int id, unsigned int ipc_class, struct medusa_kobject_s * kobj, int
(*ipc_kobj2kern)(struct medusa_kobject_s *, struct kern_ipc_perm *))
{
    struct medusa_l1_ipc_s* security_s;
    struct kern_ipc_perm *ipcp;
    struct ipc_ids *ids;
    int retval = MED_ERR;

    ids = medusa_get_ipc_ids(ipc_class);
    if(!ids)
```

```

        goto out_err;

rcu_read_lock();

ipcp = ipc_obtain_object_check(ids, id);
if(IS_ERR(ipcp) || !ipcp)
    goto out_unlock0;

if (!ipc_rcu_getref(ipcp)) {
    goto out_unlock0;
}

security_s = ipc_security(ipcp);

ipc_lock_object(ipcp);

retval = ipc_kobj2kern(kobj, ipcp);

ipc_unlock_object(ipcp);
ipc_rcu_putref(ipcp, ipc_rcu_free);
out_unlock0:
    rcu_read_unlock();
out_err:
    return retval;
}

```

Listing B.16: Operácie update

```

#include <linux/medusa/l3/registry.h>
#include <linux/medusa/l1/task.h>
#include <linux/medusa/l1/ipc.h>
#include <linux/init.h>
#include <linux/mm.h>
#include "kobject_process.h"
#include "kobject_ipc_common.h"
#include "evtype_ipc.h"

struct ipc_associate_access {
    MEDUSA_ACCESS_HEADER;
    unsigned int ipc_class;
    int flag;
};

MED_ATTRS(ipc_associate_access) {
    MED_ATTR_RO (ipc_associate_access, flag, "flag", MED_SIGNED),
    MED_ATTR_RO (ipc_associate_access, ipc_class, "ipc_class", MED_UNSIGNED),
    MED_ATTR_END
};

MED_ACCTYPE(ipc_associate_access, "ipc_associate", process_kobject, "process", ipc_kobject, "object");

int __init ipc_acctype_associate_init(void) {
    MED_REGISTER_ACCTYPE(ipc_associate_access, MEDUSA_ACCTYPE_TRIGGEREDATSUBJECT);
    return 0;
}

```



```

}

medusa_answer_t medusa_ipc_associate(struct kern_ipc_perm *ipcp, int flag)
{
    medusa_answer_t retval = MED_OK;
    struct ipc_associate_access access;
    struct process_kobject process;
    struct ipc_kobject object;
    memset(&access, '\0', sizeof(struct ipc_associate_access));
    /* process_kobject parent is zeroed by process_kern2kobj function */

    if (!MED_MAGIC_VALID(&task_security(current)) && process_kobj_validate_task(current) <= 0)
        goto out_err;
    if (!MED_MAGIC_VALID(ipc_security(ipcp)) && medusa_ipc_validate(ipcp) <= 0)
        goto out_err;

    if (!VS_INTERSECT(VSS(&task_security(current)), VS(ipc_security(ipcp))) ||
        !VS_INTERSECT(VSW(&task_security(current)), VS(ipc_security(ipcp))))
    )
        return MED_NO;

    if (MEDUSA_MONITORED_ACCESS_S(ipc_associate_access, &task_security(current))) {
        access.flag = flag;
        access.ipc_class = ipc_security(ipcp)->ipc_class;

        process_kern2kobj(&process, current);
        if(ipc_kern2kobj(&object, ipcp) != 0)
            goto out_err;

        retval = MED_DECIDE(ipc_associate_access, &access, &process, &object);
        if (retval == MED_ERR)
            retval = MED_OK;
    }
out_err:
    return retval;
}
__initcall(ipc_acctype_associate_init);

```

Listing B.17: *Typ prístupu ipc_associate*

```

def get_concrete_ipc(ipc_class, ipc_object):
    if ipc_class == 0:
        sem = Ipc_Sem(ipc_object.data)
        return sem
    elif ipc_class == 1:
        msg = Ipc_Msg(ipc_object.data)
        return msg
    elif ipc_class == 2:
        shm = Ipc_Shm(ipc_object.data)
        return shm
    elif ipc_class == 3:
        return None

@register('ipc_ctl')

```

```

def ipc_ctl(event, process, ipc_object):
    concrete = get_concrete_ipc(event.ipc_class, ipc_object)
    if not concrete:
        printk("MYSTABLE IPC_CTL CMD:{}\n".format(event.cmd))
        return MED_OK
    printk("MYSTABLE IPC_CTL id: {}, gid:{}\n".format(concrete.id, concrete.gid))
    return MED_OK

@register('ipc')
def ipc(event, sender):
    return MED_OK

```

Listing B.18: Konfigurácia autorizačného servera

C Výpis sublime

```
../.. / fei .sublime-project
```

Listing C.1: Ukážka sublime-project