Napredno praćenje rada Linux jezgre

Balder, Filip

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Filip Balder

NAPREDNO PRAĆENJE RADA LINUX JEZGRE

ZAVRŠNI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Filip Balder

Matični broj: 0016141516

Studij: Informacijski sustavi

NAPREDNO PRAĆENJE RADA LINUX JEZGRE

ZAVRŠNI RAD

Mentor :

Izv. prof. dr. sc. Ivan Magdalenić

Varaždin, rujan 2022.

Filip Balder

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Ovaj rad se bavi prijenosom i prikazom podataka relevantnih uz rad jezgre operativnog sustava Linux. Glavna ideja je stvoriti sustav koji pouzdano pohranjuje podatke te omogućuje dohvat istih. U radu će biti prikazano kako izdvojiti podatke iz već postojećih datoteka unutar Linux operacijskog sustava te poslati ih pomoću modula i bash skripte. Poslužitelj će služiti kao posrednik između spomenutog modula i krajnje aplikacije, za čiju je izradu odabran *framework Xamarin*.

Ključne riječi: linux; poslužitelj; xamarin; REST API; kernel modul; bash skripta; proc datoteke;

Sadržaj

1. Uvod

Ovaj završni rad temeljen je na izradi sustava koji olakšava praćenje rada instance Linux operativnog sustava. Motivacija za ovaj rad se nalazi u velikom osobnom interesu za Linux operacijski sustav te željom za daljnjim razvojem u tom području. Osim Linuxa, cilj je proširiti vlastita znanja u području razvoja softvera, stoga se u ovom radu koristi više različitih tehnologija.

Glavna ideja je omogućiti korisniku udaljeno praćenje rada procesa u sustavu (i samog sustava) te njihovo korištenje resursa pomoću modificirane linux jezgre. Veći dio posla se odvija u unutarnjem Linuxovom krugu pristupa (tzv. Ring 0) ili u kernel krugu kako bi modul koji šalje podatke na poslužitelj imao sva potrebna prava. Vidjet ćemo kako povezati *userspace* i *kernelspace*, odnosno kako pozvati skriptu iz korisničkog memorijskog prostora u jezgri sustava. Također, dotaknut ćemo se i virtualnog datotečnog sustava *proc*. Ovaj rad će prezentirati i korisničku aplikaciju koja na prikladan način prikazuje prikupljene podatke. Dvije navedene strane povezivat će poslužitelj kojeg ćemo izraditi pomoću programskog jezika Javascript.

Sustav koji će biti izrađen u ovom radu predstavlja skalabilni prototip koji se lako može proširiti s više potrebnih informacija, stoga naglasak nije na količini podataka već na kvaliteti prijenosa istih.

2. Metode i tehnike rada

Kao što je već navedeno u Uvodu, za izradu ovog rada korišteno je nekoliko različitih tehnologija opisanih u nastavku.

2.1. Raspberry Pi

Kao posrednik između Linux operacijskog sustava i korisničke aplikacije korišteno je malo prijenosno računalo Raspberry Pi. Računalo je kreirano u svrhu podupiranja razvoja STEM područja, no kasnije je zaživjelo i u osobnoj upotrebi s obzirom na impresivne performanse [1]. Konkretno, u ovom radu korišten je model Raspberry Pi 3B.

2.1.1. Raspberry Pi OS

Operacijski sustav koji Raspberry Pi koristi je Raspbery Pi OS, jedna od instanci mnogo poznatijeg operacijskog sustava *Debian*. Ovaj operacijski sustav slovi i kao službeni operacijski sustav Raspberry Pi računala [2].

2.1.2. Node.js

Node.js je *JavaScript Runtime* baziran na asinkronim događajima primjeren izradi skalabilnih web aplikacija [3]. Pomoću ovog *JavaScript runtimea* možemo pokretati *backend* API servise. Radi lakše izrade RESTful servisa korišten je *Express.js framework*.

2.1.2.1. Express.js

Express.js framework sadrži niz ugrađenih funkcija i metoda koje uvelike olakšavaju izradu RESTful API servisa pomoću kojih će se zapisivati podaci u bazu podataka [4]. Također, u sklopu *Express.js*-a koristi se paket *pg-promise* za spajanje na PostgreSQL bazu podataka, kao i mnogi drugi koji će biti prikazani kasnije.

2.1.3. PostgreSQL baza podataka

PostgreSQL je jedna od najpopularnijih i najstabilnijih *open-source* baza podataka. Nakon više od 30 godina razvoja, PostgreSQL podržava razne tipove podataka te procedure za očuvanje integriteta baza podataka [5]. Također, ova baza podataka je kompatibilna s velikom većinom današnjih operacijskih sustava. PostgreSQL je za potrebe ovog završnog rada je postavljen na Linux operativnom sustavu te predstavlja pozadinski temelj ovog završnog rada.

2.2. C programski jezik

C programski jezik je jedan od prvih viših programskih jezika nastao 1972. godine iz laboratorija dobro poznatog Dennisa Ritchiea. C programski jezik je sadržan u više od 90% Linuxove jezgre stoga se moduli za nadogradnju jezgre pišu upravo u ovom programskom jeziku.

2.2.1. Makefile

Jezgreni moduli se ne kompajliraju na isti način kao i obični C programi. Najčešće se koristi tzv. *Makefile* koji sadrži niz uputa kako kompajlirati izvorni kod. *Makefile* će biti detaljnije prikazan u poglavlju Razrade rada. Uz Makefile koristit ćemo i druge konfiguracijske datoteke, npr. *Kbuild*.

2.2.2. Kernel kompajler

Kernel kompajler se najčešće nalazi skriven u Linuxovom datotečnom sustavu, npr. u `vmlinuz` direktoriju. S obzirom da se radi o C kompajleru prilagođenom za kompajliranje jezgre i na parametre koje svaka Linux instanca mora imati postavljene, ne možemo ga koristiti standardnim putem. Iz tog razloga ćemo koristiti ranije spomenuti *Makefile*.

2.3. C# programski jezik

"C# je moderan, objektno-orijentiran, i siguran za tip programski jezik." [6] Nastao kao odgovor na Java programski jezik od strane Microsofta, danas predstavlja jednu od najpopularnijih tehnologija za *enterprise* aplikacije. C# se najviše oslanja na vlastiti *framework* .NET kojemu je trenutno aktualna verzija .NET 6, a za potrebe ovog rada koristit će se mobilna tehnologija Xamarin.

2.3.1. Xamarin

Xamarin je *framework* koji omogućava izradu mobilnih aplikacija pomoću C# programskog jezika u stilu *form* aplikacija. S obzirom da je potrebno prikazati podatke koje modul pošalje, prikaz podataka bit će ostvaren u vidu mobilne aplikacije.

2.4. Oracle Virtualbox

Oracle Virtualbox je aplikacija koja omogućava simuliranje hardvera računala te time pokretanje dodatnog operacijskog sustava unutar postojećeg operacijskog sustava. Razlog korištenja ovog softvera je osiguravanje sigurne testne okoline jer se bavimo samom jezgrom operacijskog sustava. Na taj način štedimo vrijeme u slučaju greške, ali i čuvamo hardver od potencijalnog oštećenja.

2.4.1. Ubuntu 22.04 LTS

Linux distribucija koja se koristi za potrebe ovog rada je Ubuntu 22.04 LTS. Ubuntu je jedan od najpoznatijih varijanti (eng. *flavour*) Linuxa. Osim toga, Ubuntu ima jezgru baziranu na Debian distribuciji, stoga modul bi trebao biti kompatibilan s velikim brojem Linux distribucija [7]. Detaljnije o samoj distribuciji bit će pojašnjeno u poglavlju Razrada teme.

2.5. Uređivači koda

Za izradu ovog završnog rada koristi se nekoliko uređivača koda/teksta.

2.5.1. Visual studio 22

Kada je izrada aplikacije u C#-u u pitanju, prirodan izbor je Microsoftova razvojna okolina Visual Studio, u ovom slučaju verzija 2022. Već dugi niz godina ovaj alat predstavlja temelj za rad s Microsoftovim tehnologijama.

2.5.2. Visual studio Code

Još jedan uređivač koda kreiran od strane Microsofta s većom podrškom za tehnologije koje nisu Microsoftove. U ovom radu se uglavnom koristi za uređivanje *Node.js* koda na poslužitelju pomoću SSH protokola.

2.5.3. Nano

Glavni alat korišten za pisanje jezgrenog modula. Kao i mnogi drugi bazirani na Linuxovom terminalu, najvećim dijelom se oslanja na unos preko tipkovnice.

2.6. draw.io

draw.io je grafički *lightweight* alat uglavnom korišten za izradu dijagrama, stoga dijagrami za potrebe ovog rada su izrađeni upravo u ovom alatu.

3. Razrada teme

Tema ovog rada sastoji se od nekoliko dijelova od kojih je naglasak na praćenju rada. Potrebno je osmisliti sustav koji se što jednostavnije integrira u Linux okruženje kako bi ga i običan korisnik mogao koristiti. Sustav će biti implementiran kroz nekoliko komponenti koje će biti objašnjene u nastavku ovog rada.

3.1. Izrada RESTful servisa

S obzirom da je cilj sustava da prikuplja podatke i periodički ih prikazuje, potrebno je osmisliti *backend* komponentu koja će obavljati obradu i pohranu podataka. Za potrebe ovog rada bit će korišten *Javascript runtime Node.js* s razvojnim okvirom (*frameworkom*) *Express.js*.

3.1.1. RESTful servisi

RESTful servisi su servisi koji nam omogućuju jednostavan i pouzdan prijenos podataka s jednog računala na drugo. Ponekad se REST API smatra ugovorom između klijenta i poslužitelja [8]. REST predstavlja Reprezentacijski prijenos stanja (*eng. Representational state transfer*) te arhitektura (identifikacija i manipulacija resursa, samoopisujuće poruke te hipermedijski prijenos podataka[8]) na kojoj REST počiva čini ga odličnim izborom za ovaj rad.

3.1.2. Izrada servera

Platforma na kojoj će server biti izrađen je Linux, konkretno Raspberry Pi OS (opisan u Poglavlju 2.) pokrenut na malom prijenosnom računalu Raspberry Pi. Tehnologija koja će pokretati ovaj poslužitelj je sveprisutni Javascript. Prije no što napišemo prvu liniju koda potrebno je instalirati pakete koji omogućuju izvođenje Javascripta u serverskom okruženju, s obzirom da se radi o jeziku namijenjenom programiranju na klijentskoj strani.

```
sudo apt-get install npm nginx nodejs
```

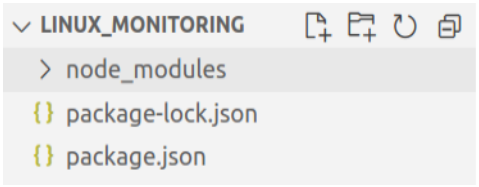
Za početak su nam potrebna dva paketa, *npm* (*Node Package Manager*) za dohvaćanje dodatnih paketa koje ćemo koristiti za izradu servera te *nginx*, alat koji će nam pomoći kasnije da proslijedimo promet s lokalne mreže na *localhost* uređaja. Kako bismo kreirali novi Node.js projekt, dovoljno je unutar mape u kojoj želimo kreirati projekt pokrenuti naredbu

```
npm init -y
```

Sada je trenutna mapa označena kao Node.js projekt što se može vidjeti po novokreiranom *package.json* dokumentu. S obzirom da radimo s Express.js frameworkom, isti je potrebno instalirati pomoću npm alata.

```
npm install express
```

Struktura projekta sada izgleda ovako.



Slika 1: Struktura projekta nakon inicijalnog postavljanja

Nakon inicijalnog postavljanja možemo kreirati i prvu Javascript datoteku. Naziv je obično proizvoljan, stoga ćemo datoteku nazvati jednostavno *server.js*. Ova datoteka predstavlja "početnu stranicu", odnosno prvu stranicu koju će poslužitelj pročitati. Za početak uključimo *express* biblioteku te biblioteku za rad s bazom podataka. PostgreSQL će poslužiti kao baza podataka za ovaj rad. Da bismo koristili PostgreSQL na našem poslužitelju, potrebno je instalirati *pg-promise* biblioteku pomoću koje se možemo spojiti na lokalnu bazu podataka.

server.js > ...

```
1  const express = require('express');
2
3  const pgp = require('pg-promise');
4  global.db = pgp('postgres://hostname:username@address:port/dbname');
```

Slika 2: Uključivanje prvih biblioteka u projekt

Možemo primijetiti da se na Slici 2. nalazi i jedna zanimljiva varijabla `global`. Ta varijabla će nam poslužiti da "recikliramo" konekciju na bazu podataka, s obzirom da ćemo kasnije imati nekoliko datoteka koje trebaju pristup bazi podataka.

Kako bi server bio u skladu s najnovijim praksama u svijetu mreža, postavit ćemo HTTPS (eng. *HyperText Transfer Protocol Secure*) protokol. Server razvijamo lokalno, stoga ga možemo samo-certificirati (eng. *self-certificate*). Poznavajući pravila certificiranja, potrebno je generirati sigurnosne ključeve pomoću nekog od kriptografskih algoritama. Na Linux operacijskom sustavu u tu svrhu možemo iskoristiti alat *openssl*[9].

```
filip@filip-VirtualBox:~/Documents/linux_monitoring$ openssl genrsa -out host.key
Generating RSA private key, 2048 bit long modulus (2 primes)
......+++++
.........................................+++++
e is 65537 (0x010001)
```

Slika 3: Generiranje lokalnog ključa pomoću alata *openssl*

Nakon generiranja privatnog RSA ključa (koji je pohranjen u obliku datoteke u korijenskoj mapi projekta) možemo generirati i javni certifikat koji ćemo koristiti za ovjeru poslužitelja.

```
open req -new -key host.key -out host.cert
```

Alat će nas zatražiti unos nekih ključnih podataka, koje će klijent moći pročitati.

```
Country Name (2 letter code) [AU]:HR
State or Province Name (full name) [Some-State]:Varazdinska
Locality Name (eg, city) []:Varazdin
Organization Name (eg, company) [Internet Widgits Pty Ltd]:FOI
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:Zavrsni
Email Address []:fbalder@foi.hr
```

Slika 4: Generiranje lokalnog ključa pomoću alata *openssl*

Zadnji korak je potpisati certifikat s privatnim ključem.

```
openssl x509 -req -days 9999 -in host.cert -signkey host.key -out
host.cert
```

Sada potpisani certifikat možemo pridružiti poslužitelju. Iskoristit ćemo ugrađenu biblioteku *fs* za čitanje datoteka.

```
const {readFileSync} = require('fs');
const serverCertificate = readFileSync('host.cert', 'utf-8');
const serverPrivateKey = readFileSync('host.key', 'utf-8');
```

Slika 5: Učitavanje certifikata i ključa

Jedino što je preostalo je pokrenuti poslužitelj.

```
const app = express();
const https = require('https');
const httpsServer = https.createServer( {key: serverPrivateKey, cert: serverCertificate}, app);
httpsServer.listen(PORT, () => console.log('Hello from server on port ' + PORT));
```

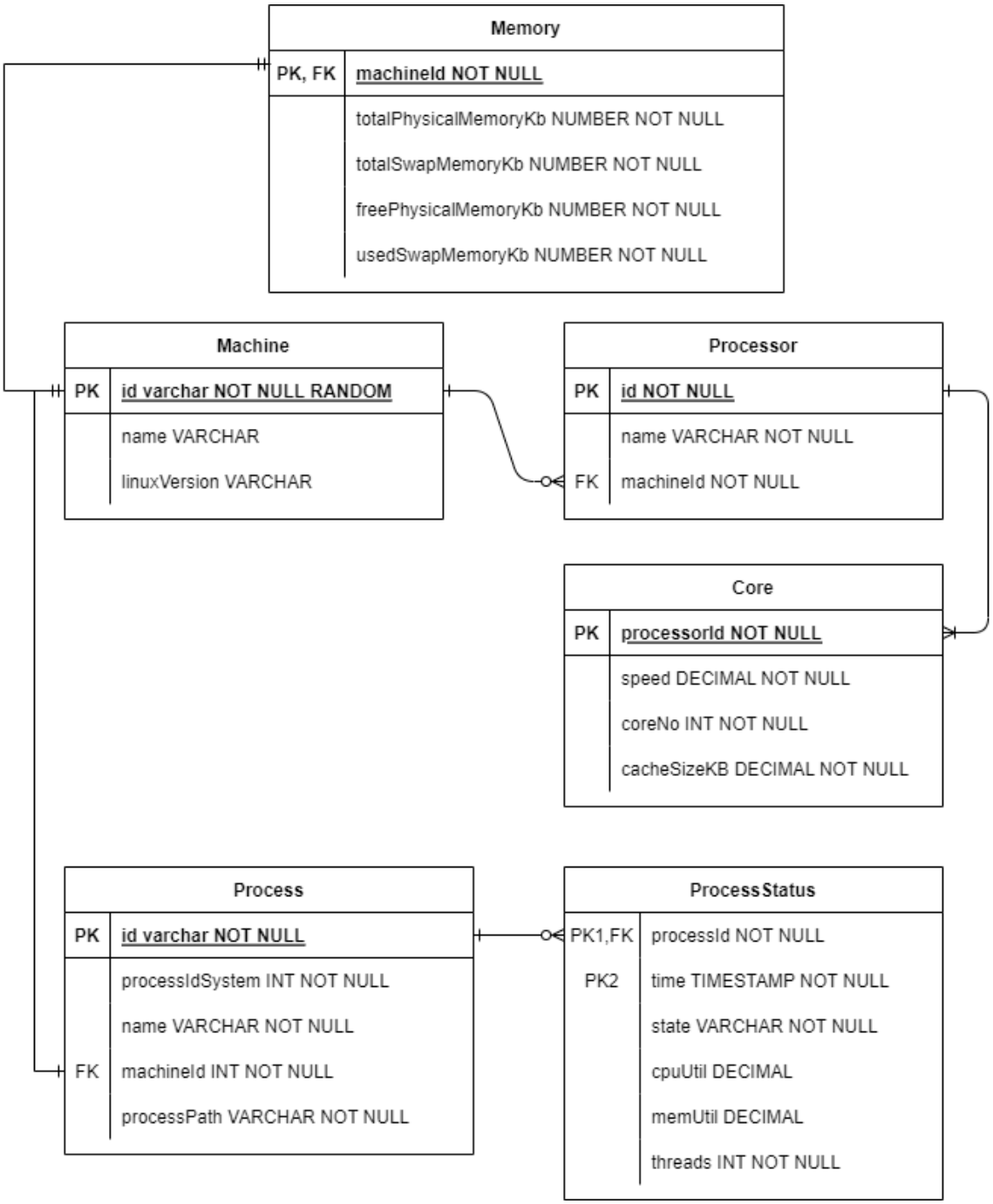
Slika 6: Instanciranje i pokretanje poslužitelja

Nekoliko stvari je dobro zapaziti s prethodne slike: da bismo pokrenuli poslužitelja potrebno je instancirati *express* klasu, a na slici je ta instanca prikazana kao *app*. Zatim nakon uključivanja *https* biblioteke, pozivamo metodu za kreiranje HTTPS veze pridružujući joj kreirani ključ i certifikat. Naposljetku, potrebno je naznačiti na kojem portu želimo primati konekcije i zahtjeve. Za potrebe ovog rada neka to bude port 5000. Sada imamo funkcionalni server na kojeg se možemo spojiti bilo gdje iz lokalne mreže. U nastavku ćemo dodati rute po kojima će razni zahtjevi prolaziti kako bi dohvatili ili pohranili podatke u bazu podataka.

3.1.3. Model baze podataka

S obzirom da bi rute koje planiramo izraditi trebale pratiti bazu podataka, poželjno je pogledati model baze podataka kojeg planiramo koristiti.

Na Slici 7. je prikazan ERA model na temelju kojeg će biti izrađena baza podataka. Naravno, ovaj model se može dodatno nadograditi u svrhu prikupljanja dodatnog sadržaja, no radi



Slika 7: Model baze podataka (ERA model)

```
create table Machine(
id varchar(100) primary key not null,
name varchar(250) not null,
linuxVersion varchar(100) not null);
```

Slika 8: Primjer SQL naredbe za izradu tablice Machine

jednostavnosti zadržat ćemo se na trenutnom, jednostavnijem modelu. Bazu je jednostavno izraditi s nekoliko SQL naredbi u *psql* komandnom alatu.

Nakon izrade i povezivanja tablica, baza podataka sadrži sljedeće relacije.

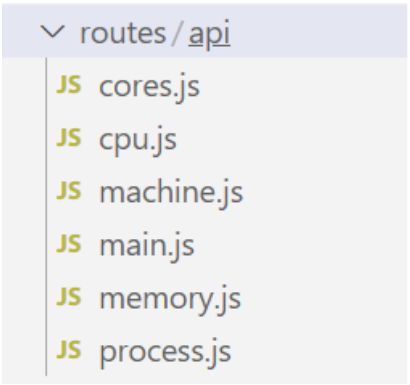
```
                List of relations
 Schema |     Name      | Type  | Owner
--------+---------------+-------+-------
 public | core          | table | pi
 public | machine       | table | pi
 public | memory        | table | pi
 public | process       | table | pi
 public | processor     | table | pi
 public | processstatus | table | pi
(6 rows)
```

Slika 9: Popis relacija u PostgreSQL bazi podataka

Sada kad je baza podataka spremna, možemo izraditi i prve rute na poslužitelju.

3.1.4. Izrada ruta

Dobra praksa prilikom kreiranja ruta je izdvojiti *endpointove* od ostatka koda na poslužitelju. Stoga, u tu svrhu su datoteke organizirane u nekoliko mapa.



Slika 10: Prikaz inicijalne strukture ruta u projektu

Da bi novokreirane datoteke bile upotrebljive početnoj skripti *server.js*, potrebno ih je registrirati kako bi naznačili da se radi ipak o upotrebljivim skriptama. Stoga dodajemo sljedeće linije koda u *main.js*.

```
const express = require('express');
const router = express.Router();
module.exports = router;
```

Slika 11: Omogućavanje importa routing modula

Ovime smo dodali u globalno polje `module` našu skriptu te je sad možemo uključiti kao rutu u *server.js*.

```
app.use('/api/main', require('./routes/api/main'));
```

Slika 12: Dodavanje rute u početnu skriptu

Metoda za prihvat ruta *use()* prima jedan parametar koji opisuje *URL* preko kojeg možemo pristupiti skripti koja je naznačena s *require()* u drugom parametru. Isti postupak ponovimo za sve ostale skripte pomoću kojih ćemo opisivati rute.

```
const mainPathRequest = '/api/main';
app.use(mainPathRequest, require('./routes/api/main'));
app.use(mainPathRequest, require('./routes/api/machine'));
app.use(mainPathRequest, require('./routes/api/cores'));
app.use(mainPathRequest, require('./routes/api/process'));
app.use(mainPathRequest, require('./routes/api/cpu'));
app.use(mainPathRequest, require('./routes/api/memory'));
```

Slika 13: Prikaz svih dodanih ruta u početnoj skripti

Nakon registracije ruta spremni smo za prihvat korisničkih zahtjeva te obradu istih. Koristit ćemo osnovne CRUD operacije (CREATE, READ, UPDATE i DELETE) u vidu HTTP metoda kao što su GET, POST, PUT i DELETE. Za početak omogućimo registraciju računala (Machine). S obzirom da ćemo koristiti POST metodu, podatke ćemo slati u tijelu zahtjeva (eng. *request body*), stoga moramo uključiti *parser* kako bismo taj *body* mogli pročitati. Uz *parser* za *body* uključit ćemo i podršku za *JSON* format (eng. *JavaScript Object Notation*) jer ćemo podatke upravo u ovom formatu slati na poslužitelj.

```
app.use(express.json());
app.use(express.urlencoded({extended: false}));
```

Slika 14: Dodavanje *parsera* u početnu skriptu

Na kraju, napišimo POST *endpoint* za registraciju novog računala u bazi podataka.

```
const uuid = require('uuid');

router.post('/machineRegister', (req, res) => {
    const newId = uuid.v4();
    const newName = req.body.name;
    const newVersion = req.body.linuxVersion;
    db.none('INSERT INTO machine (id, name, linuxVersion) \
            VALUES ($1, $2, $3)', [newId, newName, newVersion])
    .then(() => res.send(newId))
    .catch((err) => res.send(err));
})
```

Slika 15: POST *endpoint* za registraciju računala

Kreirali smo *endpoint* na adresi `localhost/api/main/machineRegister`. Prema priloženoj Slici 15. možemo vidjeti da prije svega generiramo novi ID koje će računalo imati u bazi podataka (za što nam služi biblioteka *uuid*) te dohvaćamo podatke iz tijela o nazivu i verziji linuxa. Nakon toga slijedi jednostavan SQL upit na bazu podataka koji ovisno o uspjehu vraća korisniku ili novi ID ili poruku pogreške.

Na sličan način napisat ćemo i *endpoint* za dohvaćanje već postojećeg računala.

```
router.get('/machine/:id', (req, res) => {
    db.one('SELECT * FROM machine WHERE id = $1', req.params.id)
        .then((data) => res.send(data))
        .catch((err) => res.send(err));
})
```

Slika 16: GET *endpoint* za dohvaćanje računala

Identičnim postupkom možemo implemetirati i PUT i DELETE *endpointove*.

Testirajmo kreirane *endpointove*. Kako bismo pokrenuli server potrebno je urediti *package.json* datoteku te dodati sljedeći zapis.

```
"scripts": {
  "dev": "node server"
},
```

Slika 17: Konfiguracija za pokretanje poslužitelja

Zatim u konzoli server pokrećemo naredbom

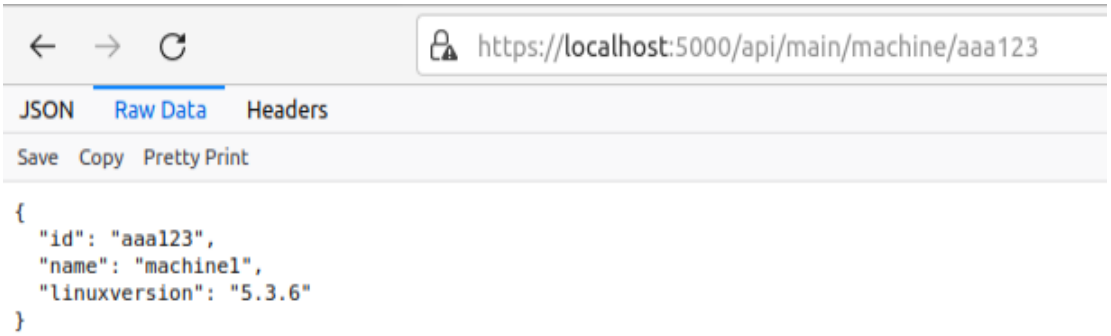
```
npm run dev
```

koji će pročitati *package.json* zapis i pokrenuti `node server` naredbu.

Kako bismo lakše testirali radi li sve ispravno, u bazu možemo ručno unijeti jedan zapis pomoću SQL naredbe

```
INSERT INTO Machine VALUES ('aaa123', 'machine1', '5.3.6')
```

Zatim *endpoint* testiramo tako što upišemo URL u web preglednik.



Slika 18: JSON zapis preuzet s poslužitelja

Kao što možemo vidjeti, API funkcionira, ali samo na jednom, lokalnom računalu. Kako bismo ovaj server mogli iskoristiti za komunikaciju s drugim računalima potrebno je ostvariti tzv. *port forwarding*. Za tu svrhu poslužit će nam alat *nginx* kojeg smo spomenuli na početku ovog poglavlja.

Kao i većina drugih alata za *port forwarding*, potrebno je urediti konfiguraciju prije pokretanja samog alata. *Nginx* konfiguracija koja je nama potrebna se nalazi u mapi `/etc/nginx/sites-available` te sadrži jednu datoteku: `default`. U toj datoteci, u polju `server` potrebno je dodati sljedeći zapis

```
location /api/main {
    proxy_pass https://localhost:5000/api/main;
}
```

Slika 19: *Nginx* konfiguracija za *port forwarding*

Ovime naznačujemo da želimo promet koji se pošalje na lokalnu IP adresu našeg uređaja s URL-om */api/main* preusmjeri upravo na naš lokalni poslužitelj. Sada je dovoljno samo pokrenuti *nginx* alat naredbom

```
sudo /etc/init.d/nginx start
```

Zadnja stvar potrebna da pristupimo udaljenom poslužitelju je IP adresa koju je vrlo jednostavno saznati naredbom u terminalu

```
hostname -i
```

3.1.5. Dovršetak poslužitelja

Za uspješan rad i praćenje velike količine podataka potrebno je dopuniti server ostalim rutama. Na kraju vrijedi spomenuti neke od važnijih ruta koje će se koristiti u sljedećem poglavlju za slanje podataka.

```
DELETE /api/main/deleteAllMachines
   POST /api/main/cpuRegister
   POST /api/main/coreRegister
  POST /api/main/processRegister
  POST /api/main/memoryRegister
   PUT /api/main/memoryUpdate
  POST /api/main/processUpdate
```

Ove rute predstavljaju jedan mali dio onoga što se zapravo može ostvariti s REST API i Node.js. Za ovaj rad bit će dovoljne kako bi se prikazala osnovna ideja iza ovakvog načina prijenosa i obrade podataka.

3.2. Izrada kernel modula

Nakon je poslužitelj izrađen, možemo se početi baviti dohvaćanjem pravih podataka iz Linux operacijskog sustava. Prije nego počnemo raditi na implementaciji, postavlja se pitanje, odakle uopće preuzeti podatke o sustavu. Poznavajući Linuxov datotečni sustav (eng. *filesystem*), dobro mjesto za početak je */proc* mapa. Na sljedećoj Slici možemo vidjeti primjer */proc* mape:

```
filip@filip-VirtualBox:~$ ls /proc
1     1357  16    1835  28    37   638          bus             modules
10    136   1605  1860  284   38   640          cgroups         mounts
100   1365  1630  1861  29    389  647          cmdline         mtd
101   1366  1636  1863  290   39   651          consoles        mtrr
102   1372  1637  188   291   391  654          cpuinfo         net
103   1373  1666  189   2924  4    677          crypto          pagetypeinfo
105   1374  17    1891  3     40   693          devices         partitions
106   1391  1706  19    30    41   694          diskstats       pressure
1066  1393  1716  190   3054  42   7            dma             schedstat
108   1395  1720  191   3057  43   709          driver          scsi
109   1399  1732  195   3062  44   752          dynamic_debug   self
11    14    1742  1953  308   45   772          execdomains     slabinfo
110   1405  1757  2     3080  459  782          fb              softirqs
111   1407  1759  20    309   460  795          filesystems     stat
112   141   1773  2032  31    464  796          fs              swaps
113   1415  1774  2036  310   5    797          interrupts      sys
114   1433  1777  2079  311   51   8            iomem           sysrq-trigger
1149  1443  1778  2081  312   56   816          ioports         sysvipc
115   1444  1780  21    3123  564  822          irq             thread-self
116   1492  1781  2128  3124  565  826          kallsyms        timer_list
117   15    1782  2140  313   567  9            kcore           tty
118   1507  1783  216   314   569  901          keys            uptime
119   1516  1784  217   315   571  94           key-users       version
12    1529  1785  22    316   575  95           kmsg            version_signature
120   1543  1786  2259  317   577  96           kpagecgroup     vmallocinfo
121   1546  18    2263  318   58   97           kpagecount      vmstat
122   1547  1804  23    32    6    98           kpageflags      zoneinfo
124   1552  1815  24    33    605  99           loadavg
1283  1555  1816  25    34    620  acpi         locks
13    1562  1818  257   35    622  asound       mdstat
132   1576  1822  26    36    623  bootconfig   meminfo
135   1591  1827  27    366   627  buddyinfo    misc
```

Slika 20: Primjer `/proc` mape

Kao što možemo vidjeti, unutar `/proc` direktorija se nalazi mnogo podmapa, kao i nekih datoteka sa zanimljivim nazivima, poput datoteke *status* ili *cpuinfo*. Upravo iz tih datoteka možemo pročitati razne informacije koje ćemo poslati na poslužitelj. Spomenute podmape će nam koristiti kasnije. Za početak pokušajmo izraditi program koji će poslati POST zahtjev na poslužitelj.

3.2.1. Kernel modul

Kernel modul je program koji omogućava manipulaciju jezgrom Linux operacijskog sustava. Kao što mu samo ime kaže, modul se izvršava u samoj jezgri, najbliže hardveru nego bilo koji drugi softver u sustavu. Time nam omogućuje pristup funkcijama i operacijama kojima u korisničkom memorijskom prostoru (*eng. userspace*) ne možemo pristupiti. Da bismo izradili bilo kakav modul prvo moramo instalirati neke od razvojnih alata, kao što je *make* ili C kompajler.

```
sudo apt-get install make gcc
```

Također, valja provjeriti imamo li kernel biblioteke instalirane u sustavu, s obzirom da prilikom razvoja jezgrenog modula moramo koristiti poseban set biblioteka.

```
sudo apt-get install linux-headers-$(uname -r)
```

Nakon postavljanja razvojne okoline, potreban nam je još samo uređivač koda, za potrebe ovog rada poslužit će *nano*. Za početak je dovoljno kreirati običan C dokument.

```
nano modul.c
```

Iako je isti programski jezik u pitanju, način programiranja u tzv. korisničkom prostoru (eng. *userspace*) i u tzv. jezgrenom prostoru (eng. *kernelspace*) se drastično razlikuje. Programiranje u jezgrenom prostoru je mnogo strože definirano što ima smisla s obzirom da se radi o jako delikatnim programima koji mogu dovesti do trajnog oštećenja računala, softverski ili čak hardverski. Jezgreni moduli se još i nazivaju *Loadable Kernel Mode* (u nastavku: LKM)[10]. Tim nazivom se sugerira da se radi o programima koji se mogu učitati u već kompajliranu jezgru. Poznavajući osnovne mehanizme operacijskih sustava može se zaključiti da je učitavanje modula vrlo pedantan postupak. Stoga *kernel* moduli imaju strogo predefiniranu strukturu koju apsolutno svaki LKM mora poštovati. Upravo je naglasak na instalaciji i deinstalaciji modula što možemo vidjeti na primjeru jedne takve strukture.

```c
#include<linux/module.h>

MODULE_DESCRIPTION("simple module");
MODULE_AUTHOR("Filip");
MODULE_LICENSE("GPL");

static int init_method(void){
        printk(KERN_INFO "simple module init\n");
        return 0;
}

static void exit_method(void){
        printk(KERN_INFO "simple module exit\n");
}

module_init(init_method);
module_exit(exit_method);
```

Slika 21: Primjer jednostavnog modula

Odmah možemo primijetiti dvije metode: `init_method` i `exit_method`. Kao što im samo ime sugerira, jedna metoda služi za inicijalizaciju modula dok druga služi za deinstalaciju modula. Te metode trenutno nemaju bitnu ulogu u sustavu, osim što ispisuju potvrdu da su se pokrenule. Bitno je primijetiti da inicijalizacijska metoda vraća i `integer`, koji zapravo predstavlja signal da je sve uredno prošlo, što je i standard prema UNIX pravilima [11]. Osim te dvije metode, LKM zahtjeva i da se specificira licenca (`MODULE_LICENSE`), inače se modul neće moći učitati u sustav. Postoje i niz drugih makro varijabli koje se mogu postaviti, poput `MODULE_DESCRIPTION` i `MODULE_AUTHOR`, kako bi se krajnjem korisniku ponudilo što više

informacija. Na kraju, metode za inicijalizaciju i deinicijalizaciju se registriraju pomoću metoda `module_init` te `module_exit` iz biblioteke `linux/module.h`.

Sada je potrebno kompajlirati program kako bismo dobili objekt kojeg možemo ubaciti u jezgru. Kompajliranje kernel modula je vrlo specifično s obzirom da ne možemo koristiti standardan način kompajliranja. Naime, potrebno je kompajlirati modul sa istim parametrima kao što je i sam kernel kompajliran. Za tu potrebu možemo napisati vlastiti `Makefile` koji će zapravo pozvati kernelov `Makefile` nad našim modulom. Potrebno je još napomenuti da kernelov `Makefile` zahtjeva dodatna uputstva u vidu `Kbuild` datoteke te ćemo također kreirati i tu datoteku.

```
EXTRA_CFLAGS = -Wall -g

obj-m += modul.o
```

Slika 22: Kbuild konfiguracija

Osim što možemo specificirati dodatne kompajlerske opcije poput `-Wall` koji će nam omogućiti da vidimo sva korisna upozorenja, unutar `Kbuilda` zapravo označavamo koju datoteku želimo kompajlirati. S obzirom da nam je cilj kreirati modul, koristimo direktivu `obj-m` u kojoj stavljamo naziv objekta koji nastaje nakon što kompajler u kernelu odradi svoj posao. Osim `obj-m` direktive, moguće je postaviti i `obj-y` za neke programe koje kernel koristi te `lib-y` i `extra-y` [12]. Sad kad je `Kbuild` napisan, možemo napisati i `Makefile`.

```
KDIR = /lib/modules/`uname -r`/build

kbuild:
	make -C $(KDIR) M=`pwd`

clean:
	make -C $(KDIR) M=`pwd` clean
```

Slika 23: Makefile konfiguracija

Unutar `Makefilea` najprije specificiramo koji ćemo kernel koristiti za kompajliranje. U varijabli `KDIR` se nalazi putanja do mape gdje se nalaze kernelove konfiguracije i njegov pripadajući `Makefile`. Komanda `uname -r` dohvaća trenutnu verziju kernela učitanog u sustav te na taj način uvijek imamo ažurnu putanju. Osim putanje, registriraju se i dva cilja (eng. *target*): `kbuild` koji specificira da ćemo koristiti `Kbuild` konfiguraciju te `clean` za jednostavno čišćenje generiranih konfiguracijskih datoteka. Naredba `make` poziva se nad varijablom `KDIR` te specificira u kojoj mapi se nalazi modul pomoću parametra `M`.

Nakon što smo pripremiti "upute" za izradu modula, možemo isti i izraditi.

```
sudo make
```

Za pokretanje kompajlera dovoljno je upisati jednostavnu naredbu `make`, pritom treba pripaziti da se naredba `make` izvodi kao administrator s obzirom da interaktiramo s kernel datotekama. Rezultat `make` naredbe izgleda ovako:

```
make -C /lib/modules/`uname -r`/build M=`pwd`
make[1]: Entering directory '/usr/src/linux-headers-5.15.0-47-generic'
  CC [M]  /home/filip/linux_monitoring/modul.o
  MODPOST /home/filip/linux_monitoring/Module.symvers
  CC [M]  /home/filip/linux_monitoring/modul.mod.o
  LD [M]  /home/filip/linux_monitoring/modul.ko
  BTF [M] /home/filip/linux_monitoring/modul.ko
Skipping BTF generation for /home/filip/linux_monitoring/modul.ko due to unavailability
 of vmlinux
make[1]: Leaving directory '/usr/src/linux-headers-5.15.0-47-generic'
```

Slika 24: Rezultat pokretanje naredbe `make`

Ukoliko `make` naredba uspješno napusti direktorij u kojem vrši operacije, možemo reći da je kompajliranje uspjelo. Također, uspješnost možemo potvrditi i jednostavnim pregledom mape u kojoj se nalazi `Makefile`.

```
filip@filip-VirtualBox:~/linux_monitoring$ ls
Kbuild    modul.c        Module.symvers  modul.mod    modul.mod.o
Makefile  modules.order  modul.ko        modul.mod.c  modul.o
```

Slika 25: Sadržaj mape nakon pokretanja naredbe `make`

Sad imamo nekoliko datoteka s različitim ekstenzijama. Te datoteke sadrže važne informacije koje će jezgra iskoristiti prilikom učitavanja modula. Učitavanje modula vrši se naredbom `insmod`.

```
sudo insmod modul.ko
```

Datoteka koju ćemo poslati u kernel mora imati ekstenziju .ko. Ako nije bilo pogrešaka prilikom učitavanja, učitavanje ne vraća nikakvu povratnu informaciju u terminalu. Razlog tomu je rad s *kernelspaceom*, što je odvojeno memorijsko područje od onog što ga koristi *userspace*. Stoga, kao takav, kernel ima i odvojene spremnike (eng. *buffer*) u kojima se prikazuju poruke iz modula. Kako bismo pristupili *kernel bufferu* dovoljno je unijeti komandu

```
sudo dmesg
```

Kao rezultat dobijemo niz poruka iz jezgre s odgovarajućom vremenskom oznakom.

```
[ 1303.255401] kauditd_printk_skb: 3 callbacks suppressed
[ 1303.255404] audit: type=1326 audit(1662396526.244:69): auid=1000 uid=1000
 gid=1000 ses=4 subj=? pid=1967 comm="pool-org.gnome." exe="/snap/snap-store
/582/usr/bin/snap-store" sig=0 arch=c000003e syscall=93 compat=0 ip=0x7f54b1
66039b code=0x50000
[ 3277.912112] modul: loading out-of-tree module taints kernel.
[ 3277.912173] modul: module verification failed: signature and/or required
key missing - tainting kernel
[ 3277.912830] simple module init
```

Slika 26: Ispis kernel *buffera*

Na Slici 26. je prikazan dio sadržaja *buffera*. Možemo vidjeti poruku iz našeg modula ('*simple module init*') što znači da se uspješno uključio u jezgru sustava. Također, valja primijetiti i poruku iznad koju je kernel generirao automatski. Naime, moduli, kao i server u prethodnom poglavlju, zahtijevaju određenu vrstu potvrde ili certifikata da su legitimni. Neki sustavi, poput Ubuntu, kojeg koristimo za izradu ovog modula, su konfigurirani tako da propuste i takve module, uz odgovorarajuću poruku kao na slici. S obzirom da će modul raditi na našem sustavu i bez verifikacije, u ovom radu nećemo potpisivati modul.

Još jedan način za provjeriti je li neki modul u sustavu je da iskoristimo naredbu za ispis svih aktivnih modula.

```
sudo lsmod
```

U kombinaciji s `grep` alatom možemo lako izdvojiti modul koji nas zanima.

```
filip@filip-VirtualBox:~/linux_monitoring$ sudo lsmod | grep modul
modul                  16384  0
```

Slika 27: Ispis modula pomoću naredbe `lsmod`

Kao što možemo vidjeti, modul je u sustavu. Ovdje možemo isčitati i veličinu (16384 bajtova) te broj drugih modula koji eventualno koriste modul (0).

Poput unosa modula, na jednostavan način možemo ga i izvaditi iz sustava.

```
sudo rmmod modul
```

Sada se u kernel *bufferu* pojavljuje poruka koju smo postavili u *exit* metodi.

```
[ 5771.650573] simple module exit
```

Slika 28: Poruka iz kernel *buffera* nakon deinstalacije modula

Osim naredbi `insmod` i `rmmod` postoji i napredniji način učitavanja modula pomoću naredbe `modprobe`. Međutim, s obzirom da se radi o naprednijem načinu, potrebno je napraviti više koraka nego kod `insmod` da bismo učitali modul. Glavna razlika je da `modprobe` za razliku

od `insmod` pazi na *dependency* module (jesu li svi dostupni, učitani i sl.). S obzirom da za ovaj rad nisu potrebni takvi moduli, `insmod` naredba će biti i više nego dostatna.

Prošli smo kroz osnovne koncepte izrade i učitavanja modula, u nastavku proširimo program s novim metodama i funkcionalnostima.

3.2.2. Dodavanje funkcija za slanje podataka

U modul možemo dodati vlastite funkcije kao i u običan C program. Glavna ideja iza funkcija u ovom radu jest da svaka od njih se bavi jednim specifičnim tipom podatka. Ako se prisjetimo ERA modela iz prethodnog poglavlja, poslužitelju bismo trebali poslati neke osnovne podatke o računalu, memoriji, procesoru te na kraju procesima. Neke od tih informacija je dovoljno poslati samo jednom, međutim podatke o procesima i memoriji bi bilo poželjno periodički ažurirati. Kao primjer kreirajmo dvije funkcije.

```
static int send_data(void *arg){
        return 0;
}

static int send_data_loop(void *arg){
        return 0;
}
```

Slika 29: Primjer funkcija u modulu

Obje funkcije vraćaju broj kao statusni kod te kao argument primaju neograničen broj istih. Također, nazivi funkcija sugeriraju da će se jedna od njih izvoditi u *loopu* ili periodički pozivati. Jednostavni poziv funkcije se izvršava kao i u svakom C programu: `send_data(NULL)`, međutim za periodičko ponavljanje ćemo morati kreirati novu dretvu. Ne smijemo zaboraviti da se ovi pozivi moraju napraviti u inicijalizacijskoj metodi modula, što znači da ne smijemo zadržati dretvu koja obavlja inicijalizaciju. Ako bismo je zadržali, modul bi ostao stalno u inicijalizacijskom modu (recimo da beskonačno puta šaljemo podatke) te ne bismo ga mogli izvaditi iz sustava standardnim putem. Dakle, za kreiranje nove dretve u jezgri poslužit će nam biblioteka `linux/kthread.h`.

Da bismo pokrenuli novu dretvu, potrebno je prvo kreirati varijablu u koju ćemo pohraniti podatke o novoj dretvi. Stoga u globalni prostor pišemo

```
struct task_struct *send_data_task;
```

`task_struct` je posebna struktura koja sadrži razne informacije o dretvi, poput ID-a ili naziva dretve kojeg ćemo i sami definirati za nekoliko trenutaka. Biblioteka `kthread.h` sadrži razne funkcije za manipulaciju dretvama, no metoda koja nama treba je `kthread_run` koja vraća upravo `task_struct` tip podataka ukoliko je dretva uspješno kreirana.

```
int err;

send_data_task = kthread_run(send_data_loop, NULL, "send_data_thread");
if(IS_ERR(send_data_task)){
        printk(KERN_ERR "Failed to create send_data_thread!\n");
        err = PTR_ERR(send_data_task);
        return err;
}
```

Slika 30: Pokretanje nove dretve u kernelu

Dakle, u ranije kreiranu strukturu pokušavamo spremiti novu dretvu pozivajući `kthread_run`. Tri su osnovna argumenta: naziv metode koju će novokreirana dretva izvršavati, parametri (koje će `void *arg` primiti, u našem slučaju je trenutno `NULL`) te proizvoljni naziv dretve (ovo nije ID, ID se generira od strane jezgre). U nastavku je prikazano i upravljanje pogreškom pomoću *error* zastavica, ukoliko se dretva iz bilo kojeg razloga ne kreira ispravno.

Preostalo je još urediti funkciju. S obzirom da stvarno želimo slati podatke beskonačno mnogo puta, možemo se poslužiti beskonačnom petljom, s jednim specifičnim uvjetom.

```
while(!kthread_should_stop()){
        msleep_interruptible(1000);
}
```

Slika 31: Beskonačna petlja u novoj dretvi

`Kthread` nam omogućuje vrlo specifičan način ostvarenja beskonačne petlje, odnosno petlja se vrti dok god ne dođe signal za gašenje dretve. Dakle, `kthread_should_stop()` zapravo predstavlja makro u kojem se nalazi informacija je li negdje pozvana metoda `kthread_stop(thread)` nad trenutnom dretvom. Upravo `kthread_stop(thread)` ćemo pozvati u deinstalacijskoj metodi modula kako ne bismo ostavili za sobom nepočišćen memorijski prostor. Unutar `while()` petlje se nalazi i jednostavan *sleep* mehanizam koji uspavljuje dretvu na 1000 milisekundi. Ovo također nije običan `sleep`, ovo je *interruptible* `sleep` što znači da drugi procesi mogu preuzeti resurse od ove dretve dok spava. To je vrlo bitan čimbenik za performans modula i općenito cijelog sustava s obzirom da će u suprotnom, jezgra prioritizirati upravo naredbu `sleep` (koja samo uspavljuje dretvu) nad nekim procesom koji je možda neophodan za normalan rad sustava.

Nakon postavljanja funkcija i njihovog pozivanja, vrijeme da pokušamo poslati i prve podatke na poslužitelj. Uzevši u obzir da je potrebno poslati HTTP zahtjeve, iskorišten je pomoćni alat radi kompleksnosti samog zahtjeva, ali i radi sigurnosti. Eventualna alternativa bi bila napisati vlastiti zahtjev pomoću kernel *socketa*. Alat koji ćemo koristiti zove se `curl`, koji je dobro poznat svima koji se bave HTTP zahtjevima. Kao što je opisano u Poglavlju 2., `curl` nam omogućuje jednostavno slanje bilo kojeg od zahtjeva s odgovarajućim teretom (eng. *payload*) što nam je iznimno bitno s obzirom da format kojeg moramo poslati je JSON. Više o samom alatu bit će opisano u sljedećem potpoglavlju.

S obzirom da je `curl` komanda koja se pokreće u *shellu*, logičan izbor je koristiti skripni jezik za upravljanje slanjem podataka. Upravo to ćemo i napraviti. Prije nego pogledamo kako napisati skriptu, pripremimo okruženje za pozivanje te skripte iz kernela.

U kernelu postoji metoda koja nam omogućava čak i pozivanje korisničkih programa iz jezgre, s tim da u tom slučaju moramo zauzeti malo korisničkog memorijskog prostora od strane trenutnog, administratorskog korisnika. Pošto je *shell* skripta ipak dio korisničkog prostora, ne možemo jednostavno smjestiti naredbe iz skripte u *kernelspace*. Spomenuta metoda nalazi se u biblioteci `linux/umh.h` pod nazivom `call_usermodehelper`. Razložit ćemo tu metodu na dva dijela kako bi bilo jasnije koje sve mogućnosti nudi.

Prije pozivanja skripte potrebno je pripremiti argumente i varijablu u kojoj će biti pohranjene informacije o procesu kojeg želimo aktivirati.

```
struct subprocess_info *scriptInfo;
int callStatus;

char *argv[] = { "/usr/bin/bash",
                 "script.sh",
                 "script_args",
                NULL };
char *envp[] = { "HOME=/",
                 "TERM=linux",
                 "PATH=/sbin:/usr/sbin:/bin:/usr/bin",
                 NULL };
```

Slika 32: Argumenti za metodu `call_usermodehelper()`

Metoda `call_usermodehelper_setup()` koju ćemo nešto kasnije iskoristiti, vraća podatke o procesu u obliku `subprocess_info` strukture. Stoga, moramo instancirati tu strukturu. Osim strukture, pripremamo i varijablu u kojoj ćemo pohraniti status izvršenja procesa. A podatke o procesu kojeg želimo pozvati, specificiramo u dva dvodimenzionalna *char* polja. U `argv` navodimo koji program želimo pokrenuti, usput navodeći i argumente koje će taj program primiti. U našem slučaju, želimo pokrenuti *bash shell* te mu proslijediti skriptu `script.sh` koja pak prima svoje vlastite argumente u nastavku. Važno je napomenuti da oba polja moraju završiti s `NULL` jer na taj način *shell* zna gdje naredba završava. Drugo polje služi za postavljanje *enviroment* varijabli. Sad kad imamo spremne osnovne parametre, možemo kreirati `subprocess_info` strukturu.

```
scriptInfo = call_usermodehelper_setup(argv[0], argv, envp, GFP_KERNEL, NULL, NULL, NULL);
```

Slika 33: `call_usermodehelper_setup()` poziv

Metoda `call_usermodehelper_setup()` zahtjeva sedam parametara. Prvi parametar služi za postavljanje putanje do izvršne datoteke (eng. *executable*). Putanja se nalazi na nultoj poziciji u `argv` polju, stoga ju možemo iskoristiti. Drugim argumentom specificiramo ostale parametre koje želimo poslati izvršnoj datoteci navedenoj u putanji, dok trećim postavljamo *enviroment* varijable. S obzirom da alociramo dio memorije za našu strukturu, moramo odrediti na koji način želimo izvršiti alokaciju, stoga koristimo zastavicu `GFP_KERNEL` koja za-

pravo predstavlja normalnu dinamičku alokaciju kernel memorijskog prostora. Alternativa ovoj zastavici je `GFP_ATOMIC` koja zabranjuje prekidanje izvođenja ove alokacije. S obzirom da ne radimo nikakav kritičan posao, obična alokacija će biti i više no dovoljna. Preostala tri argumenta služe za dodavanje metoda koje bi se izvodile prilkom kreiranja i prilikom uništavanja `subprocess_info` strukture te za specificiranje dodatnih podataka koje želimo smjestiti u spomenutu strukturu. Takve metode nam trenutno nisu potrebne, tako da ćemo ih postaviti na `NULL`.

Da bismo pozvali proces, iskoristit ćemo `call_usermodehelper_exec()` metodu.

```
callStatus = call_usermodehelper_exec(scriptInfo, UMH_WAIT_EXEC);
```

Slika 34: `call_usermodehelper_exec()` poziv

Ova metoda prima samo dva parametra, jedan je opisnik procesa, odnosno struktura koju smo maloprije kreirali, a drugi je makro zastavica koja naznačava čekanje na izvršenje procesa. U našem slučaju, postavit ćemo zastavicu čekanja na `UMH_WAIT_EXEC` što označava čekanje do uspješnog poziva programa. Osim ovog moda čekanja, možemo upotrijebiti `UMH_WAIT_PROC` koji čeka dok proces ne završi s izvođenjem ili pak `UMH_NO_WAIT` koji uopće ne čeka proces. Također, statusni kod spremamo u varijablu `callStatus` koju smo pripremili nešto ranije. Taj statusni kod možemo provjeriti, te ako nije 0, vratiti pogrešku. Mala je šansa da će se to dogoditi s obzirom da pozivamo *bash shell* koji je ugrađen u sustav.

Struktura `subprocess_info` se sad dealocira te se poziva metoda čišćenja ukoliko smo je specificirali ranije u `call_usermodehelper_setup()`.

Sad kad smo vidjeli kako pozvati skriptu iz modula, možemo i napisati skriptu koju ćemo pozivati na ovaj način.

3.2.3. Bash skripta

Svaki Linux operacijski sustav ima neku vrstu ljuske koja omogućuje skriptiranje (*eng. scripting shell)* ugrađenog u sebi. U našem slučaju, koristit ćemo *bash* skriptu. Kao što je navedeno u prethodnom potpoglavlju, ova skripta bi trebala poslati podatke na poslužitelj pomoću `curl` alata. Svaka *bash* skripta počinje s `#!/bin/bash` linijom, ili tzv. *shebang* linijom prema nekim literaturama [13], a zatim se u datoteku mogu upisivati komande kao se nalazimo u terminalu.

Prije nego je upotrijebimo u skripti, pogledajmo kako radi naredba `curl`. Pokušajmo registrirati neko računalo na poslužitelj i iste te podatke dobiti nazad u terminalu. Za početak prvo pišemo `curl` te kao parametar dodajemo zastavicu `-X` koja nam omogućuje da postavimo tip HTTP zahtjeva. U našem slučaju to će biti `POST`. Zatim još jedna zastavica `-H` za postavljanje zaglavlja zahtjeva. Shodno tome, u nastavku pišemo `"Content-type: application/json"`. Zatim pomoću zastavice `-d` postavljamo podatke (eng. *data*) ili tijelo zahtjeva: `""name": "machine1", "linuxVersion":"5.3.6""`. Važno je staviti dvostruke navodnike s obzirom da tako nalaže JSON standard, stoga će u skripti biti potrebno uvesti

i tzv. *escape* znakove kako bi skripta dobro protumačila *payload*. Na kraju dodajemo IP adresu poslužitelja, odnosno *endpointa* na koji želimo poslati JSON.

```
filip@filip-VirtualBox:~/linux_monitoring$ \
> curl -X POST -H "Content-type: application/json" \
> -d {"\"name\":\"machine1\",\"linuxVersion\":\"5.3.6\""} \
> --insecure https://192.168.137.161:5000/api/main/machineRegister
747fc2cf-ad42-4f5e-9303-0d69edb52a18
```

Slika 35: Rezultat naredbe `curl`

U konačnu naredbu dodana je i zastavica `insecure` s obzirom da radimo sa samostalno potpisanim serverom. Primijetimo da je zahtjev vratio ID novokreiranog računala. Taj ID ćemo morati pohraniti negdje u sustavu pošto će nam biti potreban u daljnjem radu. Za tu svrhu iskoristit ćemo virtualni datotečni sustav `/proc`. U sljedećem potpoglavlju pogledat ćemo kako izraditi jednostavan `/proc` zapis pomoću kernel modula.

Pokušajmo sad dohvatiti isto to računalo iz baze koristeći HTTP zahtjev GET.

```
filip@filip-VirtualBox:~/linux_monitoring$ \
> curl -X GET -H "Content-type: application/json" \
> --insecure https://192.168.137.161:5000/api/main/machine/747fc2cf-ad42-4f5e-9303-0d6
9edb52a18
{"id":"747fc2cf-ad42-4f5e-9303-0d69edb52a18","name":"machine1","linuxversion":"5.3.6"}
```

Slika 36: Rezultat dohvaćanja računala pomoću naredbe `curl`

I doista, dobili smo zapis kojeg smo samostalno unijeli pomoću `curla`. Sada možemo prenijeti prikazano u skriptu i iskoristiti prave podatke o računalu.

```
#!/bin/bash

ip_address=$1
linuxVersion=$(cat /proc/version | cut -d ' ' -f3)
name=$(hostname)
newMachineId=$(curl -s -X POST -H "Content-type: application/json" \
               -d "{\"name\":\"$name\", \"linuxVersion\":\"$linuxVersion\"}" \
               --insecure https://$ip_address/api/main/machineRegister)
```

Slika 37: Dohvaćanje podataka o računalu i slanje

Kako bi skripta bila prilagodljiva koristit ćemo varijable. Varijabla se kreira jednostavno navodeći naziv te pridruživanjem vrijednosti znakom jednakosti. Kako bismo spremili vrijednost iz neke komande koristimo znak dolara($). U tom slučaju rezultat naredbe se neće ispisati u konzolu kao inače, već će se pohraniti u varijablu. Također, u skriptu možemo unositi i vlastite parametre kao što je prikazano u liniji `ip_address=$1`. Ovaj parametar poslužit će jednostavnoj izmjeni IP adrese na kojoj se nalazi poslužitelj. Na kraju, pozivamo `curl` naredbu na identičan način kao u prethodnim primjerima te spremamo ID koji poslužitelj vraća u varijablu `newMachineId`.

Skriptu ćemo proširiti s još jednim argumentom, a to je naziv akcije koju želimo izvršiti. Naime, cilj nam je da sve potrebne radnje obavljamo putem jedne skripte, stoga ćemo dodati argument `command`. Kasnije možemo dodati npr. *switch-case* grananje za određivanje akcije koju želimo pokrenuti.

```
command=$2
```

Kako bismo mogli sačuvati ID-jeve i informacije iz skripte iskoristit ćemo `/proc` direktorij. U nastavku ćemo vidjeti kako izraditi jednostavnu datoteku u virtualnom datotečnom sustavu.

3.2.4. /proc datotečni sustav

Kao što je već napomenuto, da bismo kreirali `/proc` datoteku, moramo kreirati modul koji će upravljati tom datotekom. Drugim riječima, `/proc` datoteka i njezini podaci će se nalaziti u *kernelspaceu*, ali mi kao programeri se moramo pobrinuti da se podaci koje korisnik unese prenesu iz *userspacea* u naš *kernelspace*. Također, i u suprotnom smjeru, kada korisnik zatraži čitanje datoteke, moramo se pobrinuti da se podaci prebace iz *kernelspacea* u *userspace* [14].

Metoda za kreiranje `/proc` datoteke je `proc_create()` iz biblioteke `linux/proc_fs.h` te prima nekoliko ključnih parametara: naziv, prava pristupa, `/proc` naddirektorij te opis operacija. Naziv i prava pristupa je lako definirati, također i naddirektorij kojeg u ovom slučaju možemo postaviti na `NULL` kako bi datoteka bila smještena u korijenski direktorij. Međutim, zadnji argument zahtjeva inicijalizaciju strukture `proc_ops` u kojoj će biti opisane podržane operacije nad našom datotekom. Također, trebamo i varijablu u kojoj ćemo pohraniti informacije o datoteci, stoga inicijalizirat ćemo i `proc_dir_entry` strukturu.

```
#include<linux/proc_fs.h>

static struct proc_dir_entry *proc;

static struct proc_ops proc_operations_info = {};
```

Slika 38: Strukture potrebne za izradu `/proc` datoteke

Sada možemo u inicijalizacijskoj metodi modula pozvati metodu za kreiranje `/proc` datoteke.

```
static int init_proc(void){
        proc = proc_create("simple_proc", 0666, NULL, &proc_operations_info);
        if(proc == NULL){
                printk(KERN_ERR "Could not initialize /proc/simple_proc\n");
                return -ENOMEM;
        }
        return 0;
}
```

Slika 39: Inicijalizacija `/proc` datoteke

Također, odmah dodajmo i brisanje `/proc` datoteke prilikom deinstalacije modula.

```
static void exit_proc(void){
        proc_remove(proc);
}
```

Slika 40: Brisanje `/proc` datoteke

Pošto sad imamo više modula u istoj mapi, moramo shodno ažurirati `Kbuild` datoteku kako bi `make` obuhvatio sve prisutne module.

```
EXTRA_CFLAGS = -Wall -g

obj-m += modul.o
obj-m += simple_proc.o
```

Slika 41: `Kbuild` konfiguracija - više modula

Postupak učitavanje je identičan kao u prethodnom primjeru.

```
sudo make
  sudo insmod simple_proc.ko
```

```
filip@filip-VirtualBox:~/linux_monitoring$ ls /proc | grep simple_proc
simple_proc
```

Slika 42: Prikaz novokreiranog `/proc` unosa

Ostalo je još napisati kako će se datoteka ponašati kad pokušamo nešto upisati u nju ili pročitati iz nje. Trenutno bismo dobili pogrešku s obzirom da nismo nikakvu akciju registrirali u modulu.

Prije nego implementiramo čitanje i pisanje, moramo osigurati memorijski prostor gdje ćemo pohraniti poruke iz *userspacea*, stoga kreirajmo jedno polje *charova* u globalnom prostoru modula.

```
char buf[1048576];
```

Ovo polje će predstavljati spremnik naše datoteke veličine točno 1 MB. Pogledajmo sada jednostavno implementaciju pisanja u `/proc` datoteku.

```
static ssize_t proc_write(struct file *file, const char *ubuf, size_t count, loff_t *ppos){
        int buf_len;
        char *temp_buf;

        temp_buf = kmalloc(strlen(ubuf), GFP_USER);

        copy_from_user(temp_buf, ubuf, count);

        strcat(buf, temp_buf);

        buf_len = strlen(buf);

        *ppos = buf_len;

        kfree(temp_buf);

        return buf_len;
}
```

Slika 43: Implementacija pisanja u `/proc` datoteku

Kao što se može vidjeti na Slici 43. potrebno je izvršiti niz operacija kako bi sve funkcioniralo. U kratkim crtama, potrebno je kreirati privremeni spremnik (`temp_buf`), koji se dinamički alocira kako bismo koristili minimalno potrebne memorije. Taj spremnik ćemo iskoristiti za pohranu novog sadržaja iz korisničkog spremnika (`ubuf`) te ćemo ga nadodati u naš glavni spremnik `buf` pomoću metode `strcat`. Na kraju, ažuriramo pokazivač na poziciju u datoteci te vraćamo duljinu spremnika kao rezultat opreracije.

Na identičan način, možemo implementirati i čitanje iz datoteke.

```
static ssize_t proc_read(struct file *file, char *ubuf, size_t count, loff_t *ppos){
        int buf_len;

        if(*ppos > 0){
                *ppos = 0;
                return 0;
        }

        buf_len = strlen(buf);

        copy_to_user(ubuf, buf, buf_len);

        *ppos = buf_len;
        return buf_len;
}
```

Slika 44: Implementacija čitanja iz `/proc` datoteku

U ovom slučaju, tijek događanja je obrnut. Sada prenosimo iz našeg glavnog spremnika (`buf`) u korisnički (`ubuf`) te vraćamo duljinu prenesenog sadržaja. Mnogi alati za čitanje (poput `cat`) će pokušavati pročitati datoteku sve dok ne dobiju statusni kod 0. Iz tog razloga moramo implementirati provjeru jesmo li pročitali sav sadržaj iz datoteke. Radi jednostavnosti u ovom primjeru je napravljena samo provjera gdje se nalazi pokazivač `*ppos` jer se pretpostavlja da se sav sadržaj datoteke može prenijeti odjednom. U našem slučaju je to istina, stoga ovo funkcionira.

Istražujući mogućnosti ovakvih virtualnih datoteka dolazi se do zaključka da se može ostvariti vrlo visok stupanj kontrole sustava. No za sad je dovoljno nadopuniti operacije mogućnostima poput čišćenja datoteke, nema potrebe ići dalje od toga s obzirom da ćemo ove

datoteke za sad koristiti kao da su tekstualne datoteke.

U novokreirane datoteke sada možemo spremiti ID-jeve koje poslužitelj vraća, stoga nadopunimo skriptu.

```
#!/bin/bash

output_file_ids="/proc/monitoring_info"

ip_address=$1
linuxVersion=$(cat /proc/version | cut -d ' ' -f3)
name=$(hostname)
newMachineId=$(curl -s -X POST -H "Content-type: application/json" \
                -d "{\"name\":\"$name\", \"linuxVersion\":\"$linuxVersion\"}" \
                --insecure https://$ip_address/api/main/machineRegister)

printf "MachineId: %s\n" $newMachineId >> "$output_file_ids"
```

Slika 45: Pohrana dobivenog ID-ja u `/proc` datoteku

S obzirom da će se skripta višestruko koristiti, dobra je praksa pohraniti putanju do datoteke u varijablu kako bi lako mogli pozvati, ali i izmijeniti putanju. Za ispis u datoteku možemo jednostavno iskoristiti *output redirection operator* u kombinaciji s naredbom `printf`.

```
root@filip-VirtualBox:/proc# cat monitoring_info
MachineId: 8c036451-0734-4471-8402-c3045d510f80
```

Slika 46: Prikaz ID-ja u `/proc` datoteci

Kao što vidimo, ID se sad uspješno zapisuje u `/proc` datoteku. U nastavku možemo dodatno doraditi skriptu.

3.2.5. Dovršetak skripte

Kako bi skripta što bolje radila, možemo dodati razna grananja kako bismo provjerili postoji li već registrirano računalo u `/proc` memoriji. Osim provjera, dodat ćemo na identičan način još jednu `/proc` datoteku koja će služiti kao zapisnik (eng. *log*). Na kraju, kao što je već ranije spomenuto, dodat ćemo `switch` grananje kako bismo mogli filtrirati koju akciju želimo izvršiti. Nakon primjene svega spomenutog, dobivamo skriptu kao na slici.

```
#!/bin/bash

ip_address=$1
command=$2

output_file_ids="/proc/monitoring_info"
output_file_log="/proc/monitoring_log"

case $command in
        "machineRegister")
                linuxVersion=$(cat /proc/version | cut -d ' ' -f3)
                name=$(hostname)

                if [ -z "$(cat ${output_file_ids} | grep MachineId)" ]
                then
                        echo "[machineRegister] Creating new machine!" >> "$output_file_log"

                        newMachineId=$(curl -s -X POST -H "Content-type: application/json" \
                                -d "{\"name\":\"$name\", \"linuxVersion\":\"$linuxVersion\"}" \
                                --insecure https://$ip_address/api/main/machineRegister)
                        printf "MachineId: %s\n" $newMachineId >> "$output_file_ids"
                else
                        echo "[machineRegister] Machine ID already present!" >> "$output_file_log"

                        newMachineId=$(cat "$output_file_ids" | grep MachineId | cut -d ':' -f2 | xargs)
                fi

                echo 'Machine id: '$newMachineId$'\n'
```

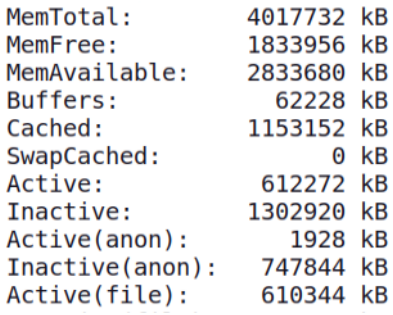
Slika 47: Prikaz skripte za registraciju računala

U nastavku skriptu je potrebno dovršiti na sličan način tako da podržava i druge funkcionalnosti, poput registracije procesora, memorije, procesa te ažuriranja istih. Pogledajmo kako bismo dohvatili neke osnovne informacije o memoriji računala pomoću *bash* skripte.

```
totalPhysicalMemoryKb=$(cat /proc/meminfo | grep MemTotal | cut -d ':' -f2 | xargs | cut -d ' ' -f1 | xargs)
totalSwapKb=$(cat /proc/meminfo | grep SwapTotal | cut -d ':' -f2 | xargs | cut -d ' ' -f1 | xargs)
freePhysicalMemoryKb=$(cat /proc/meminfo | grep MemFree | cut -d ':' -f2 | xargs | cut -d ' ' -f1 | xargs)
freeSwapMemoryKb=$(cat /proc/meminfo | grep SwapFree | cut -d ':' -f2 | xargs | cut -d ' ' -f1 | xargs)
```

Slika 48: Dohvat osnovnih podataka o memoriji računala

Naizgled kompleksni niz naredbi se može svesti na prenošenje podataka u svrhu ljepšeg oblikovanja. Prva u nizu naredba je `cat` koja služi za čitanje datoteke. Pogledajmo izgled datoteke `/proc/meminfo`.



Slika 49: Dio podataka iz `/proc/meminfo` datoteke

Naime, ova datoteka sadrži relevantne podatke o memoriji računala. Možemo primijetiti da svaki zapis ima identični format. Stoga možemo izvojiti podatak pomoću `grep` naredbe, razdvojiti ga po dvotočki te izvući u našem slučaju broj kilobajta. U naredbi se nalazi i `xargs` naredba koja se brine da nema nepotrebnih razmaka s lijeve ili s desne strane kako bi zapis bio što čišći.

Unutar `/proc` direktorija se nalaze i mnoge druge korisne informacije o sustavu. Važno je napomenuti kako svaki proces u sustavu ima svoj poddirektorij u spomenutom direktoriju. U svakom od tih poddirektorija se nalazi mnoštvo informacija koje se može iskoristiti. Možda najkorisnija datoteka za ovaj rad je `/proc/pid/status`.

```
root@filip-VirtualBox:/proc/1# cat status
Name:   systemd
Umask:  0000
State:  S (sleeping)
```

Slika 50: Dio podataka iz `/proc/1/status` datoteke

Ovdje je prikazan samo mali dio onoga što `status` sadrži, ali i to je dovoljno da saznamo kako se proces zove u sustavu (`systemd`) te u kojem je stanju (`S (sleeping)`). Još neke zanimljive datoteke u trenutnom direktoriju su `cmdline` koji nam omogućava da vidimo s kojim parametrima je proces pokrenut ili pak `environ` koji nam prikazuje tzv. *enviroment* varijable. Naposljetku, moguće je skriptu kombinirati i s postojećim alatima za skeniranje i praćenje sustava, poput naredbe `ps`.

Pogledali smo sve bitne aspekte skripte za slanje i ekstrakciju podataka, u nastavku ćemo objediniti sve prikazano u jednu funkcionalnu cjelinu.

3.2.6. Završni izgled modula

Kreiranu skriptu je sad lako uključiti u početni modul s obzirom da je potrebno samo poslati putanju do skripte kao argument u `call_usermodehelper_setup()` metodi. Ono što još želimo omogućiti korisniku je da samostalno odabere procese koje želi pratiti. Za tu svrhu dodat ćemo parametre prilikom učitavanja kernel modula.

Da bismo koristili modul parametre, potrebno je uključiti biblioteku `linux/moduleparam.h`. Dodavanje podrške za parametre je vrlo jednostavno jer spomenuta biblioteka ima ugrađene metode koje nam omogućuju dodavanje jednog parametra ili pak niza parametara. S obzirom da želimo omogućiti da korisnik unese nekoliko identifikatora procesa odjednom, koristit ćemo drugu opciju.

```
static int pids[100];
static int processIdsCount;
module_param_array(pids, int, &processIdsCount, 0660);
MODULE_PARM_DESC(pids, "User filled array with process ID (int) targeted for monitoring.");
```

Slika 51: Dodavanje parametara u modul

Kao i obično, i ovi parametri se pohranjuju u varijablu, stoga kreirajmo polje *integera* u koje ćemo pohraniti korisnički unos. Osim niza pohrane niza elemenata, `module_param_array()` nam omogućuje i pohranu broja upisanih elemenata, što u jednu ruku olakšava baratanje tim nizom s obzirom da ne moramo ručno računati broj elemenata. Uz pohranu parametara, također možemo i dodati poseban opis što taj parametar predstavlja. U ovom primjeru je pripremljeno polje od maksimalno 100 elemenata pod pretpostavkom da korisnik neće unijeti više od 100 procesnih identifikatora.

Parametri se unose tako da nakon naziva modula specificiramo naziv parametra kojeg želimo unijeti, zatim nakon znaka jednakosti navedemo željene elemente odvojene zarezom.

```
sudo insmod modul.ko pids=1,10
```

Ova naredba će učitati `modul.ko` s parametrima `1` i `10`. U konačnoj verziji modula će biti dodani još neki parametri, poput frekvencije slanja, kako bi korisničko iskustvo bilo što bolje.

Pogledajmo sad konačni izgled metode za registraciju računala.

```
static int machine_register(void *arg){
        char functionName[20] = "machine_register";
        struct subprocess_info *scriptInfo;
        int callStatus;

        printk(KERN_INFO "%s thread id: %d\n", functionName, current->pid);

        char * argv[] = { "/usr/bin/bash",
                          abs_script_path,
                          MAIN_IP,
                          "machineRegister",
                          NULL };

        char * envp[] = { "HOME=/",
                          "TERM=linux",
                          "PATH=/sbin:/usr/sbin:/bin:/usr/bin",
                          NULL };

        scriptInfo = call_usermodehelper_setup(argv[0], argv, envp, GFP_KERNEL,
                                        NULL, &machine_register_cleanup, NULL);
        if(scriptInfo == NULL){
                printk(KERN_ERR "Error while creating script (%s)\n", functionName);
                return -ENOMEM;
        }

        printk(KERN_INFO "%s %s %s %s\n", argv[0], argv[1], argv[2], argv[3]);
        callStatus = call_usermodehelper_exec(scriptInfo, UMH_WAIT_EXEC);

        if(callStatus != 0){
                printk(KERN_ERR "Error while calling script (code: %d) (%s)\n",
                        -callStatus, functionName);
                return -callStatus;
        }

        printk(KERN_INFO "Machine register called. Status: %d (%s).\n",
                          callStatus, functionName);
        return 0;
}
```

Slika 52: Završna verzija metode za registraciju računala

Uz dodavanje putanje skripte u obliku varijable `abs_script_path` dodane su i razne provjere i ispisi informacija te sada konačna funkcija izgleda kao na Slici 52. Sve ostale metode vezane uz memoriju, procese i sl. se mogu kreirati na identičan način. Osim metoda za kreiranje, kreirana je i jedna metoda za brisanje zapisa sa poslužitelja. Ideja je da se nakon deinstalacije modula obrišu podaci relevantni za taj modul. Podatke je moguće i zadržati zahvaljujući parametru `keep` kojeg korisnik može postaviti. U tom slučaju se ne poziva metoda za brisanje sadržaja. Također, u inicijalizacijsku metodu je dodano nekoliko neophodnih grananja provjere kako bi modul radio ispravno u slučaju da korisnik ne unese ni jedan proces ID. U tom slučaju će se registrirati samo računalo bez procesa.

```
if(processIdsCount != 0){
        process_register(pids);
        process_update_task = kthread_run(process_update, pids, "process_update_thread");
        if(IS_ERR(process_update_task)){
                printk(KERN_ERR "ERROR: Cannot create process_update_thread!\n");
                err = PTR_ERR(process_update_task);
                process_update_task = NULL;
                return err;
        }
}
else{
        printk(KERN_INFO "No PiDs provided. Process registration and update skipped!\n");
}

memory_update_task = kthread_run(memory_update, NULL, "memory_update_thread");
if(IS_ERR(memory_update_task)){
        printk(KERN_ERR "ERROR: Cannot create memory_update_thread!\n");
        err = PTR_ERR(memory_update_task);
        memory_update_task = NULL;
        return err;
}
```

Slika 53: Završna verzija inicijalizacijske metode

Na kraju, pogledajmo i deinstalacijsku metodu u kojoj se gase dretve ažuriranja podataka te ovisno o parametru `keep` brišu podaci.

```
void exit_routine(void){
        if(processIdsCount != 0)
                kthread_stop(process_update_task);
        kthread_stop(memory_update_task);
        if(keep != 1)
                remove_machine(NULL);
        printk("Main monitoring module unloaded!\n");
}
```

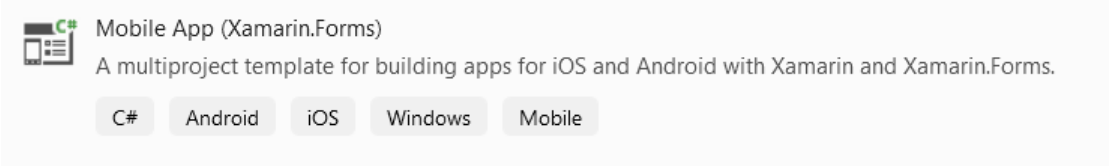
Slika 54: Završna verzija deinstalacijske metode

Ovime smo završili pregled i izradu kernel modula koji će poslužiti za slanje podataka o jezgri sustava. Preostalo je još izraditi prikaz podataka na klijentskoj strani ovog sustava.

3.3. Izrada korisničke aplikacije

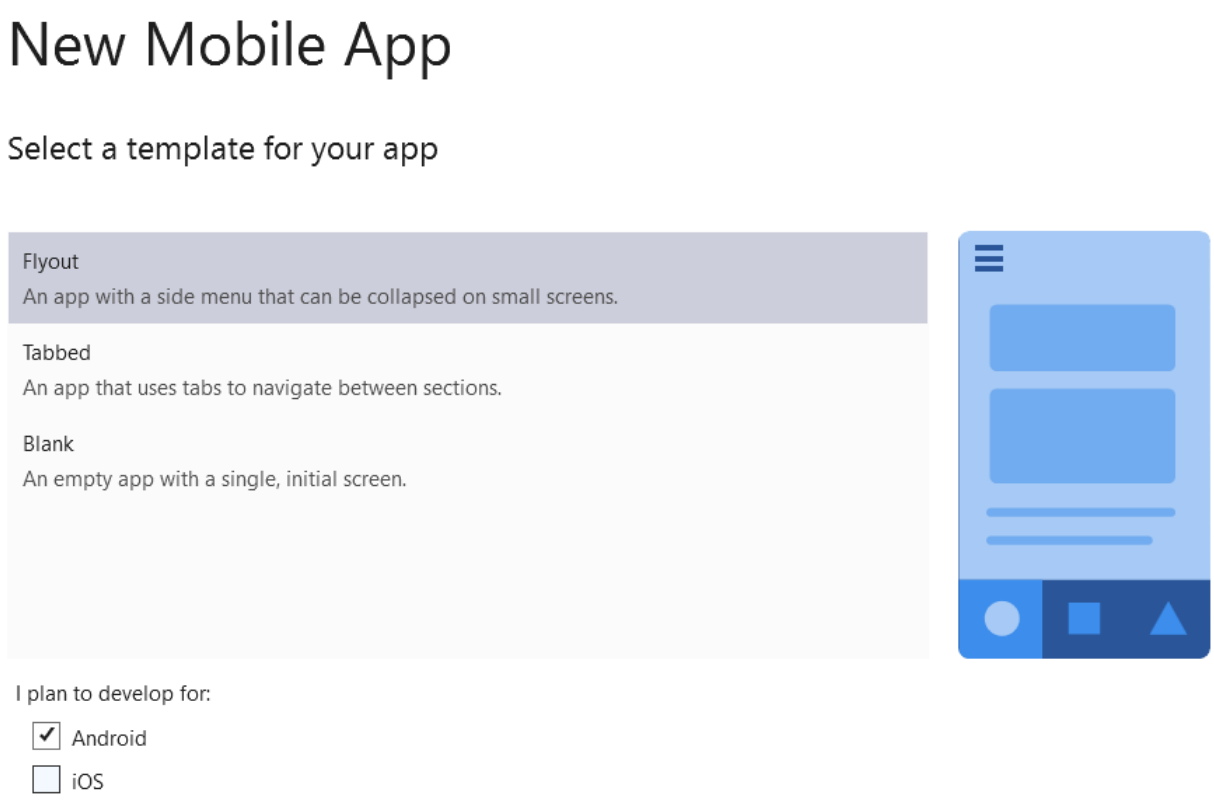
Naposljetku, korisnička aplikacija je konačni prikaz rezultata rada. Za izradu aplikacije je odabrana C# tehnologija, Xamarin. Xamarin je *framework* za izradu mobilnih aplikacija što je idealno razvojno okruženje za praćenje rada sustava s udaljenog mjesta. Zahvaljujući REST API-ju kojeg koristimo, lako možemo dohvatiti podatke iz baze koje smo slali pomoću modula. Kao razvojno okruženje koristit ćemo Visual Studio 2022 s obzirom da je to službeni alat za razvoj C# aplikacija. Nakon instalacije Xamarin *frameworka* možemo kreirati projekt.

3.3.1. Postavljanje novog projekta



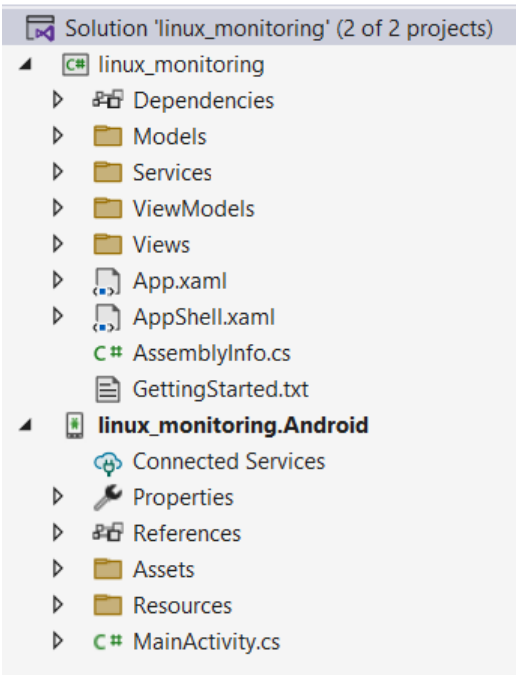
Slika 55: Kreiranje Xamarin.Forms projekta

U ovom radu poslužit će nam forme s obzirom da ne trebamo nikakvo napredno sučelje, već samo jednostavan i pristupačan prikaz podataka. Radi jednostavnosti iskoristit ćemo već kreiran predložak za izbornik (*flyout* izbornik) te ćemo se fokusirati na razvoj za Android platformu.



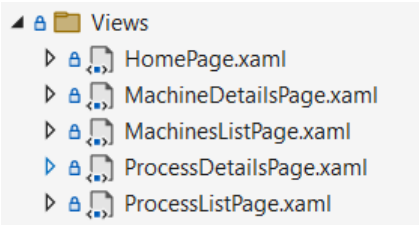
Slika 56: Dodatne opcije Xamarin.Forms projekta

Projekt kojeg kreiramo ima već predefiniranu strukturu vidljivu na sljedećoj slici.



Slika 57: Početna struktura Xamarin.Forms projekta

Primjećujemo kako su kreirana dva projekta, jedan općeniti i jedan specifični za Android uređaje. Naime, Xamarin *framework* se bazira na dijeljenju C# koda između različitih sustava, stoga veći dio aplikacije kodiramo u općem projektu, a zatim prilagodimo određene stavke prema potrebi u projektu specifičnom za određeni sustav [15]. Osim *shared code* komponente, Xamarin nudi nekoliko načina razvoja aplikacije, npr. možemo koristiti isključivo C# kod za razvoj aplikacije. U ovom radu primijenit ćemo tzv. MVVM uzorak (eng. *Model-View-ViewModel*) što u prijevodu znači da ćemo korisiti kombinaciju C#-a i XAML-a (eng. *extensible application markup language*). Prije nego započnemo s kreiranjem *Viewa* počistimo sve datoteke koje se nalaze u `Models`, `Services`, `ViewModels` te `Views` mapama. Potrebno je pročistiti `App` i `AppShell` datoteke od referenci na obrisane datoteke. Nakon toga možemo kreirati svoje datoteke.



Slika 58: Struktura *Views* direktorija

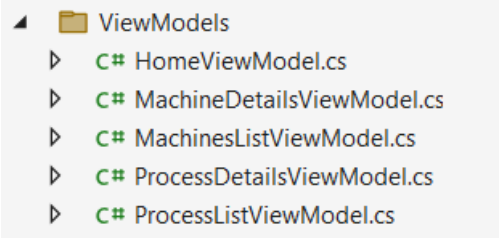
View kreiramo desnim klikom na mapu `Views`, zatim `Add->New Item->Content Page (XAML)`. S obzirom da je cilj prikazati sve podatke koje pošaljemo na poslužitelj, potrebna nam je stranica koja će prikazivati popis registriranih računala te infomacije ili detalje o odabranom računalu. Slična ideja se krije i iza procesa. Uz to, dodali smo i jednu početnu (*home*) stranicu. Kao što je rečeno ranije, u projektu se već nalazi predložak izbornika u dato-

teci `AppShell.xaml`, stoga možemo jednostavno registrirati nove stranice u izbornik.

```
<FlyoutItem Title="Home">
    <ShellContent Route="HomePage"
                  ContentTemplate="{DataTemplate local:HomePage}" />
</FlyoutItem>
<FlyoutItem Title="Machines">
    <ShellContent Route="MachinesPage"
                  ContentTemplate="{DataTemplate local:MachinesListPage}" />
</FlyoutItem>
<FlyoutItem Title="Processes">
    <ShellContent Route="ProcessesPage"
                  ContentTemplate="{DataTemplate local:ProcessListPage}" />
</FlyoutItem>
```

Slika 59: Registracija stranica u izbornik

U nastavku možemo kreirati pripadajuće `ViewModel` datoteke za svaku od stranica. Ove datoteke će nam služiti za upravljanje logikom iza `Viewa`.



Slika 60: Struktura `ViewModels` direktorija

Kako bismo mogli upotrijebiti varijable i podatke koje obradimo s `ViewModel` datotekom moramo ju registrirati u odgovarajućoj `View` datoteci. XAML, kao i programski jezici zahtjeva direktivu kako bi znao koju datoteku treba uključiti u svoj radni tijek. Stoga u opis `ContentPage` stavljamo dvije direktive.

```
xmlns:viewmodels="clr-namespace:linux_monitoring.ViewModels"
        x:DataType="viewmodels:MachineListViewModel"
```

Ove dvije linije će pronaći odgovarajući `ViewModel` i pridružiti ga kontekstu ovog XAML dokumenta. Sad možemo i eksplicitno naznačiti da želimo koristiti taj `ViewModel` za dohvaćanje podataka.

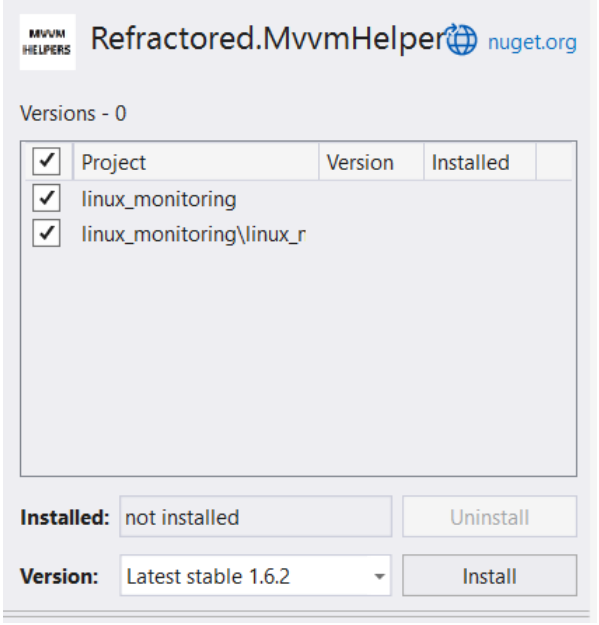
```
<ContentPage.BindingContext>
    <viewmodels:MachinesListViewModel/>
</ContentPage.BindingContext>
```

Slika 61: Dodavanje `ViewModel` datoteke u kontekst stranice

Da bismo prikazali podatke na stranici, možemo u već postojeći `StackLayout` dodati element `ListView` koji sukladno njegovom imenu, služi za prikaz liste. No trenutno ne možemo prikazati nikakve podatke jer iste nismo ni dohvatili, stoga moramo urediti pripadajuću `ViewModel` datoteku.

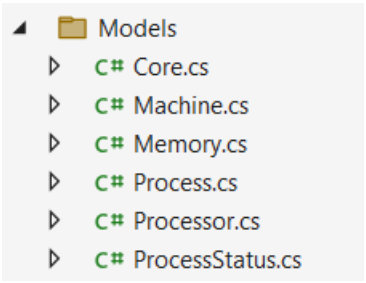
3.3.2. Unutarnja logika aplikacije

Kako bismo lakše implementirali MVVM predložak, instalirat ćemo biblioteku `MvvmHelpers`.



Slika 62: Instalacija pomoćne biblioteke

Ova biblioteka sadrži mnoga predefinirana svojstva i metode koje olakšavaju razvoj Xamarin aplikacija. Prvi korak koji moramo napraviti prije nego dohvatimo podatke je kreirati klase koji će poslužiti kao modeli podataka te svojstvo koje će pohraniti podatke. Stoga u mapi `Models` kreirajmo presliku ERA modela iz prvog potpoglavlja.



Slika 63: Struktura `Models` direktorija

Unutar pojedine datoteke moramo definirati i svojstva. Poželjno je svojstva nazvati identično kao u bazi podataka pošto želimo mapiranje podataka odraditi automatski.

```
0 references
public class Memory
{
    0 references
    public string MachineId { get; set; }
    0 references
    public int TotalPhysicalMemoryKb { get; set; }
    0 references
    public int TotalSwapMemoryKb { get; set; }
    0 references
    public int FreePhysicalMemoryKb { get; set; }
    0 references
    public int UsedSwapMemoryKb { get; set; }
}
```

Slika 64: Primjer `Memory` modela

Sad kad imamo modele, možemo kreirati i listu u koju ćemo pohraniti podatke sa servera. Upravo iz biblioteke `MvvmHelpers` ćemo iskoristiti tip podataka `ObservableRangeCollection` za pohranu.

```
private ObservableRangeCollection<Machine> machines;
0 references
public ObservableRangeCollection<Machine> Machines
{
    get => machines;
    set => SetProperty(ref machines, value);
}
```

Slika 65: Lista za spremanje podataka o računalu

Da bismo dohvatili podatke potrebna nam je C# klasa `HttpClient` iz biblioteke `System.Net.Http`. Međutim, da bismo kreirali klijenta moramo uzeti u obzir da i dalje radimo s lokalnim poslužiteljem, koji je samostalno potpisan. Zbog sigurnosnih standarda, `HttpClient` neće htjeti preuzeti podatke s takvog poslužitelja, stoga moramo pregaziti (eng. *override*) mehanizam koji provjerava certifikat.

```
private HttpClient client = new HttpClient(
    new HttpClientHandler
    {
        ServerCertificateCustomValidationCallback = CustomValidation
    });


private static bool CustomValidation
    (HttpRequestMessage message,
    X509Certificate cert,
    X509Chain ch, SslPolicyErrors err) => true;
```

Slika 66: Ručna validacija certifikata

Prikazani kod će jednostavno zanemariti certifikat, tako što će metoda za provjeru certifikata uvijek vratiti `true`, odnosno ispravan rezultat. Pošto sad imamo klijenta, možemo mu dodijeliti početnu adresu u konstruktoru `ViewModel` datoteke.

```
0 references
public MachinesListViewModel()
{
    client.BaseAddress = new Uri("https://192.168.137.161:5000/api/main/");
    Machines = new ObservableRangeCollection<Machine>();
}
```

Slika 67: Dodjela početne adrese `HttpClientu`

Na kraju, možemo kreirati metodu koja će poslužiti za dohvaćanje liste računala.

```
0 references
public async Task GetMachines()
{
    var machinesJson = await client.GetStringAsync("machine");
    var machinesList = JsonConvert.DeserializeObject
                            <IEnumerable<Machine>>(machinesJson);
    Machines = new ObservableRangeCollection<Machine>(machinesList);
}
```

Slika 68: Metoda za dohvat računala s poslužitelja

S obzirom da znamo da će se podaci pojaviti u obliku stringa, možemo iskoristiti metodu `GetStringAsync` koja će odraditi ekstrakciju stringa umjesto nas. Nakon toga, možemo iskoristiti `Newtonsoftovu` metodu `DeserializeObject` da pretvorimo, odnosno mapiramo podatke iz JSON stringa u naše modele. Na kraju instanciramo novi `ObservableRangeCollection` na temelju dobivenog niza podataka.

Prije nego se prebacimo na XAML dokument, moramo još kreirati način kako pozvati metodu iz grafičkog sučelja. Za tu potrebu možemo iskoristiti iz biblioteke `MvvmHelpers` strukturu `AsyncCommand` koju možemo pozvati iz XAML-a. U globalni prostor klase dodajemo

```
public AsyncCommand GetMachinesCommand { get; set; }
```

Te u konstruktor:

```
GetMachinesCommand = new AsyncCommand(GetMachines);
```

Sada možemo nastaviti na XAML datoteku. Za početak povežimo `ListView` i listu računala.

```
<ListView
    ItemsSource="{Binding Machines}">
```

Slika 69: Povezivanje `ListViewa` i varijable `Machines`

Ovo svojstvo će pridružiti varijablu `Machines` listi te prikazati podatke koje ona sadrži. Također, ukoliko ažuriramo varijablu, ažurirat će se i prikaz na listi. Međutim, ovaj pristup će prikazati sve podatke koje model sadrži, stoga moramo urediti `ItemTemplate` kojeg lista koristi.

Da bismo mogli dohvatiti svojstva koje model sadrži, moramo registrirati modele na identičan način kao i *viewmodele*.

```
xmlns:model="clr-namespace:linux_monitoring.Models"
```

Sad možemo dodati sljedeći komad koda.

```
<ListView.ItemTemplate>
    <DataTemplate x:DataType="model:Machine">
        <ViewCell>
            <StackLayout>
                <Label VerticalOptions="Center"
                       FontSize="20"
                       Text="{Binding Name, StringFormat='Name: {0}'}"
                       TextColor="Black"/>
                <Label VerticalOptions="Center"
                       FontSize="20"
                       Text="{Binding LinuxVersion, StringFormat='Version: {0}'}"
                       TextColor="Black"/>
            </StackLayout>
        </ViewCell>
    </DataTemplate>
</ListView.ItemTemplate>
```

Slika 70: Uređivanje `ItemTemplate` svojstva

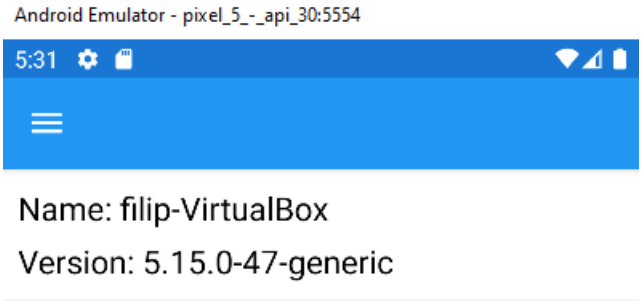
Ono na što najviše treba obratiti pažnju je svojstvo `DataTemplate` u kojem specificiramo da radimo s modelom `Machine`. Stoga nešto niže u svojstvu `Label` možemo postaviti kao `Text` proizvoljni natpis, koristeći samo jedan podatak iz modela.

Preostalo je još samo pozvati metodu i dohvatiti prave podatke s poslužitelja. Pošto želimo da se podaci pojave prilikom otvaranja stranice, iskoristit ćemo još jednu biblioteku koja će nam omogućiti da određeni *event* stranice pretvorimo u komandu. U biblioteci `Xamarin.CommunityToolkit` postoji konverter `EventToCommandBehaviour` pomoću kojeg ćemo pretvoriti događaj '*appearing*' u asinkronu komandu koju smo nešto ranije pripremili u `ViewModel` datoteci.

```
<ContentPage.Behaviors>
    <xct:EventToCommandBehavior
        EventName="Appearing"
        Command="{Binding GetMachinesCommand}"/>
</ContentPage.Behaviors>
```

Slika 71: Pozivanje `GetMachinesCommand` prilikom pojavljivanja stranice

Dakle, prilikom pojavljivanja stranice aktivirat će se komanda koja poziva metodu za dohvaćanje podataka.



Slika 72: Prikaz liste u Android emulatoru

I doista, ukoliko pokrenemo aplikaciju, pojavit će se informacija da je jedno računalo registrirano u sustav. Pogledajmo sad kako bismo dohvatili informacije o procesoru i memoriju o odabranom računalu. Poslužit ćemo se svojstvom `SelectedItem` unutar `ListViewa`.

```
SelectedItem="{Binding SelectedMachine, Mode=TwoWay}"
```

Ovime naznačujemo da ćemo koristiti `ViewModel` svojstvo `SelectedMachine` za pohranu odabranog računala. Uz varijablu, stavljamo i `Mode=TwoWay` što označava da želimo imati ažurne podatke i u `ViewModel` datoteci u `View` datoteci.

```
private Machine selectedMachine;
0 references
public Machine SelectedMachine
{
    get => selectedMachine;
    set => SetProperty(ref selectedMachine, value);
}
```

Slika 73: Dodavanje svojstva odabranog računala

Osim što želimo odabrati računalo, želimo otvoriti i novu stranicu koje sadrži informacije za odabrano računalo. Ponovo ćemo se poslužiti `Xamarin.Community Toolkitom`. Ovog puta želimo uzeti konverter koji će dohvatiti odabrano računalo kao argument te ga poslati u komandu. Taj argument ćemo zatim poslati na drugu stranicu te tako prikazati informacije o odabranom računalu. Uključimo konverter u XAML datoteku.

```
<ContentPage.Resources>
    <ResourceDictionary>
        <xct:ItemSelectedEventArgsConverter x:Key="ItemSelectedEventArgsConverter"/>
    </ResourceDictionary>
</ContentPage.Resources>
```

Slika 74: Dodavanje konvertera za odabran element

Slijedno prikazanom, dodajmo pozivanje komande u `ListView`.

```
<ListView.Behaviors>
    <xct:EventToCommandBehavior
    EventName="ItemSelected"
    Command="{Binding SelectedCommand}"
    EventArgsConverter="{StaticResource ItemSelectedEventArgsConverter}"/>
</ListView.Behaviors>
```

Slika 75: Dodavanje komande u `ListView`

Unutar `ViewModela` kreirajmo komandu za odabrano računalo.

```
public AsyncCommand<Machine> SelectedCommand { get; private set; }
```

U konstruktor stavljamo inicijalizaciju kao i do sad.

```
SelectedCommand = new AsyncCommand<Machine>(Selected);
```

Primijetimo da ovog puta `AsyncCommand` ima pridružen i tip u šiljastim zagradama. Taj tip predstavlja vrstu argumenta koju želimo poslati po komandi. Hvatamo računalo, stoga je tip argumenta model računala `Machine`. Naposljetku, kreirajmo i metodu u kojoj ćemo pozvati ovu stranicu.

```
1 reference
public async Task Selected (Machine selectedMachine)
{
    var route =
        $"{nameof(MachineDetailsPage)}?SelectedMachineID={selectedMachine.Id}";
    await Shell.Current.GoToAsync(route);
}
```

Slika 76: Pozivanje nove stranice s ID-jem odabranog računala

Ono što možemo prvo primijetiti jest da metoda prima jedan argument, što je i bio kranji cilj jer sada možemo dohvatiti ID računala i poslati ga na novu stranicu. Za slanje ID-a možemo iskoristiti tzv. *query string* što je poznat način slanja parametara u web programiranju. Međutim, ovaj kod neće raditi jer nismo registrirali rutu po kojoj bi se trebala pozvati nova stranica. Stoga u `AppShell.cs` dodajmo sljedeću liniju.

```
Routing.RegisterRoute(nameof(MachineDetailsPage),
            typeof(MachineDetailsPage));
```

Osim `AppShell.cs` moramo urediti i odredišnu datoteku `MachineDetailsPage`. Dodat ćemo atribut `QueryProperty`.

```
[QueryProperty(nameof(SelectedMachineID), nameof(SelectedMachineID))]
0 references
public class MachineDetailsViewModel : BaseViewModel
{
    private string selectedMachineID;
    1 reference
    public string SelectedMachineID
    {
        get => selectedMachineID;
        set => SetProperty(ref selectedMachineID, value);
    }
}
```

Slika 77: Dodavanje *query* atributa nad odredišnom klasom

Ovim atributom naznačujemo kako se zove parametar kojeg šaljemo, te u koju varijablu spremiti parametar. Svojstvo za pohranu parametra je nazvano isto kao i parametar, `SelectedMachineID`. U nastavku je potrebno ponoviti već prikazane korake: napraviti klijenta, dohvatiti podatke i prikazati ih na ekranu.

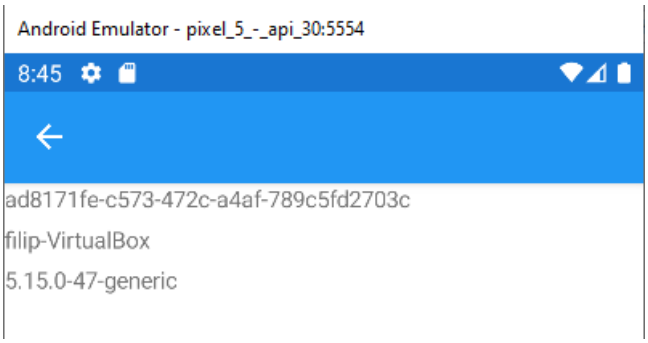
```
1 reference
public AsyncCommand GetMachineInfoCommand { get; set; }
public HttpClient client = new HttpClient(new HttpClientHandler
{
    ServerCertificateCustomValidationCallback = CustomValidation
});
1 reference
private static bool CustomValidation
    (HttpRequestMessage requestMessage,
    X509Certificate certificate,
    X509Chain chain, SslPolicyErrors errors) => true;
0 references
public MachineDetailsViewModel()
{
    GetMachineInfoCommand = new AsyncCommand(GetMachineInfo);
    client.BaseAddress = new Uri("https://192.168.137.161:5000/api/main/");
}
1 reference
public async Task GetMachineInfo()
{
    var machineJson =
        await client.GetStringAsync($"machine/{SelectedMachineID}");
    CurrentMachine =
        JsonConvert.DeserializeObject<Machine>(machineJson);
}
```

Slika 78: Dohvaćanje informacija o odabranom računalu

```
<ContentPage.BindingContext>
    <viewmodels:MachineDetailsViewModel/>
</ContentPage.BindingContext>
<ContentPage.Behaviors>
    <xct:EventToCommandBehavior
        EventName="Appearing"
        Command="{Binding GetMachineInfoCommand}"/>
</ContentPage.Behaviors>
<ContentPage.Content>
    <StackLayout>
        <Label Text="{Binding CurrentMachine.Id}"/>
        <Label Text="{Binding CurrentMachine.Name}"/>
        <Label Text="{Binding CurrentMachine.LinuxVersion}"/>
    </StackLayout>
</ContentPage.Content>
```

Slika 79: Prikaz dodatnih informacija o odabranom računalu

Na kraju, prikaz dohvaćenih podataka preko ID-a izgleda ovako.



Slika 80: Jednostavan prikaz dohvaćenih podataka o računalu

Aplikacija se može doraditi dodavanjem novih podataka, uredivanjem elemenata, ubacivanjem animacije i sl. Ovdje smo vidjeli samo osnove onoga što Xamarin kao *framework* nudi te ovime završavamo s konceptima korištenim za izradu korisničke aplikacije. U posljednjem poglavlju pogledat ćemo konačni izgled aplikacije i rad sustava.

4. Pregled rada sustava

Na kraju, preostaje vidjeti kako sustav radi u cjelini. Nakon što smo prošli kroz sve komponente sustava i osnovne mehanizme korištene u njihovoj izradi pogledajmo krajnji rezultat. Svaka komponenta je dovršena sukladno početnim zahtjevima, poslužitelju su dodani novi *endpointovi*, a modul šalje više podataka. Korisnička aplikacija je estetski uređenija i smislenija, što ćemo i vidjeti u nastavku.

Trenutno se u direktorij uz glavni modul nalaze sljedeće datoteke:

- `Kbuild`
- `Makefile`
- `proc_info_monitoring.c`
- `proc_log_monitoring.c`
- `send_request.sh`
- `params -> popis mogućih parametara u tekstalnom obliku`

Nakon pokretanja poslužitelja, postavljanja korektne IP adrese i kompajliranja, učitajmo sve module. Sustav trenutno zahtjeva tri modula za ispravan rad. Uz već spomenutu `/proc` datoteku dodana je još jedna takva koja će služiti kao *log*. U glavni modul učitat ćemo *process* ID programa *Firefox* kako bismo dobili podatke.

```
sudo insmod proc_info_monitoring.ko
sudo insmod proc_log_monitoring.ko
sudo insmod main_monitoring.ko pids=2745 keep=1 freq=2000
```

Osim *process* ID-ja, poslali smo i parametar `keep` jer želimo zadržati podatke i nakon deinstalacije te parametar `freq=2000` kako bi se podaci slali svake 2 sekunde. Nakon učitavanja modula, podaci su se počeli slati, što se može i vidjeti i u kernel *bufferu*.

```
[  697.316009] Proc monitoring entry created (/proc/monitoring_info)
[  700.715254] Proc monitoring entries created (/proc/monitoring_log)
[  722.202594] working dir: /home/filip/linux_monitoring
[  722.202604] abs script path: /home/filip/linux_monitoring/send_request.sh
[  722.202605] Initializing monitoring module...
[  722.202606] machine_register thread id: 4771
[  722.202607] /usr/bin/bash /home/filip/linux_monitoring/send_request.sh 192.168.137.161:5000 machineRegister
[  722.202979] machine_register_cleanup
[  722.202980] Machine register called. Status: 0 (machine_register).
[  722.212852] READ HANDLER MONITORING INFO! Count: 0 ppos: 0

[  722.213312] WRITE HANDLER MONITORING LOG! Count: 40
[  722.476330] WRITE HANDLER MONITORING INFO! Count: 48
[  722.482276] READ HANDLER MONITORING INFO! Count: 48 ppos: 0
```

Slika 81: Kernel *buffer* poruke prilikom inicijalizacije

Kao što možemo vidjeti, iz modula dobivamo podosta korisnih informacija pomoću kojih možemo pratiti što se događa u sustavu. Naravno, najčešće nije potrebno ovoliko poruka slati u zapisnik, no kako bismo lakše shvatili što se događa opisan je skoro svaki korak koji modul napravi. Tako možemo vidjeti koji je trenutno radni direktorij, na koji način se skripta poziva, poruke iz `/proc` direktorija i sl.

Također, možemo vidjeti i da se proces registrirao na poslužitelja.

```
[  723.231966] Process ID: 2745
[  723.231967] /usr/bin/bash /home/filip/linux_monitoring/send_request.sh 192.168.137.161:5000 processRegister 2745
[  723.232348] process_register_cleanup
[  723.232350] Process register called. Status: 0 (process_register).
[  723.232649] process_update thread id: 4886
[  723.232652] Process ID: 2745
[  723.232654] /usr/bin/bash /home/filip/linux_monitoring/send_request.sh 192.168.137.161:5000 processUpdate 2745
[  723.232672] WRITE HANDLER MONITORING INFO! Count: 109
```

Slika 82: Poruke o procesu iz kernel *buffera*

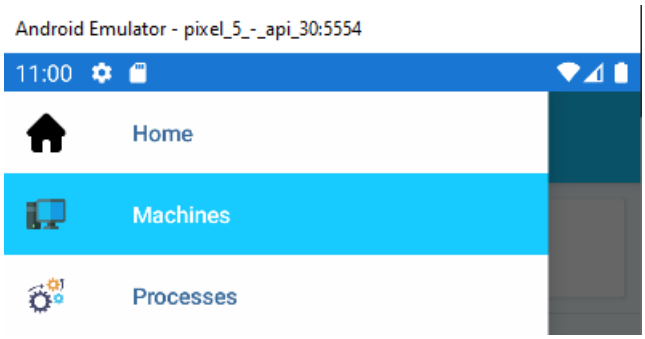
Točno možemo vidjeti o kojem ID-ju se radi, koji je status pozivanja skripte i sl. Pogledajmo i poruke s poslužitelja kako bi vidjeli da se uistinu stanje o procesu šalje svake dvije sekunde kako smo i naveli ranije.

```
[08/Sep/2022:20:24:20 +0000] "POST /api/main/processUpdate
[08/Sep/2022:20:24:22 +0000] "POST /api/main/processUpdate
[08/Sep/2022:20:24:24 +0000] "POST /api/main/processUpdate
[08/Sep/2022:20:24:26 +0000] "POST /api/main/processUpdate
```

Slika 83: Prikaz zapisnika sa poslužitelja

Iz zapisnika poslužitelja možemo vidjeti učestalost poziva `processUpdate` u uglatim zagradama. Sad kad smo ustanovili da se podaci ispravno šalju, pogledajmo i grafički prikaz unutar mobilne aplikacije.

Za početak, u izborniku možemo odabrati opciju *Machines*.

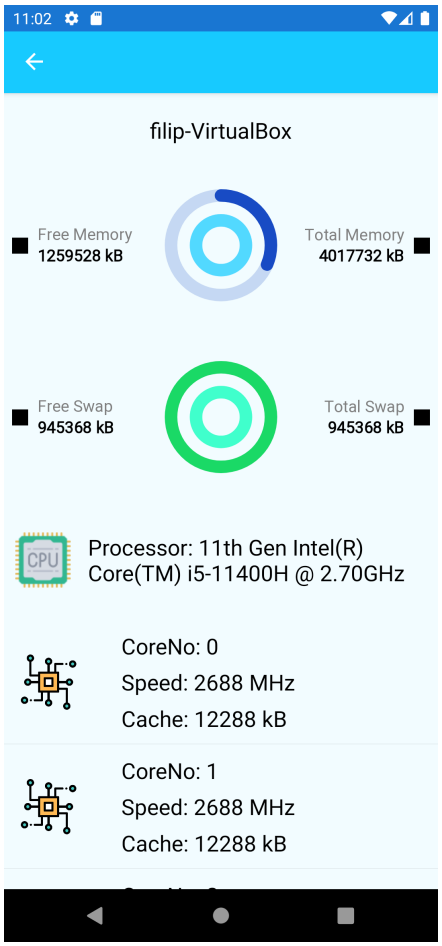


Slika 84: Izbornik unutar aplikacije

Nakon odabira otvara se lista trenutno registriranih računala.

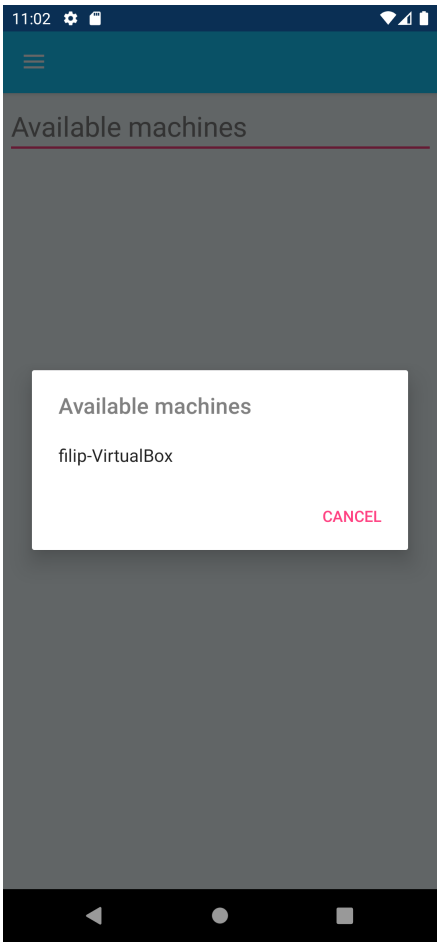
Slika 85: Lista registriranih računala

Pritiskom na željeno računalo, otvara se nova stranica koja prikazuje podatke o računalu. U svrhu izrade stranice iskorišteni su grafovi iz biblioteke `Microcharts`.



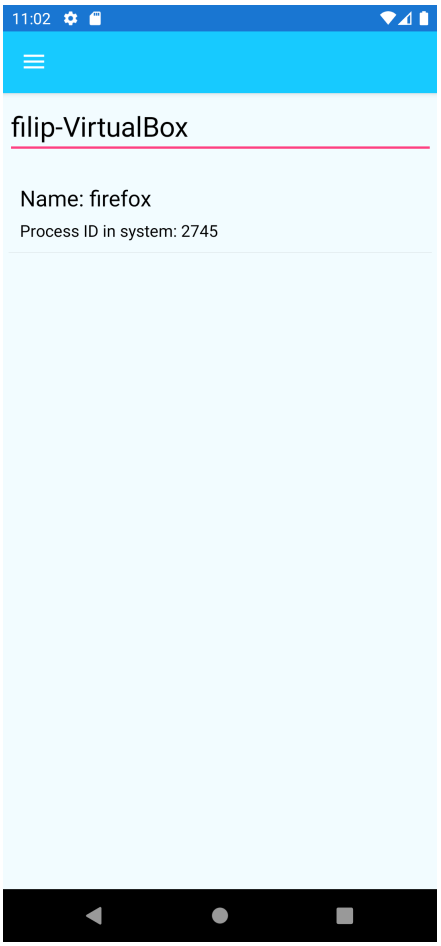
Slika 86: Detalji o računalu

Nakon pregleda računala, možemo pregledati i koje procese to računalo ima registrirano. Natrag u izborniku izaberemo *Processes* i dobijemo stranicu gdje pomoću *pickera* biramo koje računalo želimo provjeriti.



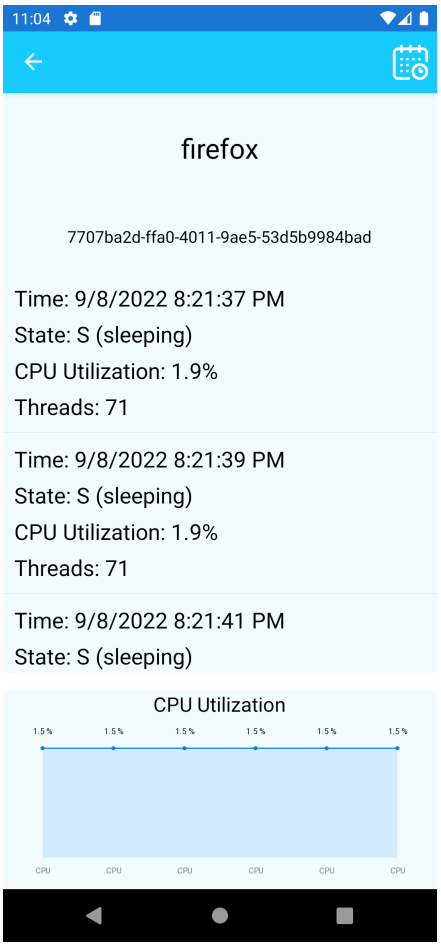
Slika 87: *Picker* računala

Odabirom na računalo, otvaraju se procesi i pojavljuje se onaj isti *Firefox* proces kojeg smo prije nekoliko trenutaka poslali iz modula.



Slika 88: Prikaz liste procesa u računalu

Pritiskom na proces, otvara se nova stranica koja prikazuje dodatne detaljne informacije o procesu.



Slika 89: Prikaz zapisnika sa poslužitelja

Stranica trenutno prikazuje naziv procesa, ID u bazi te neke osnovne informacije koje smo poslali pomoću modula. Na dnu je grafičkim prikazom prikazana uporaba procesora za ovaj specifični proces.

Ovime završavamo pregled ovog sustava. Kao što je već spomenuto u uvodu, ovaj rad predstavlja prototip kako bi jedan sustav za slanje podataka mogao raditi. Naravno, sada je lako dodati još podataka, proširiti bazu i uskladiti poslužitelj, ovisno o potrebama korisnika. Sustav je testiran isključivo u lokalnom okruženju s obzirom na pogreške koje bi se potencijalno pojavile tijekom prijenosa na druge okoline. Stoga, za ozbiljniju primjenu bilo bi potrebno još poraditi na optimizaciji, sigurnosti i općenito integraciji sustava.

5. Zaključak

Kao rezultat ovog rada postignut je jednostavan, ali dovoljno stabilan sustav prijenosa podataka od Linux operativnog sustava do korisničke aplikacije. U sustavu se nalaze ukupno tri komponente: linux kernel modul s pripadajućom skriptom, poslužitelj i korisnička aplikacija. Za potrebe poslužitelja iskorištena je tehnologija Node.js koja nam omogućuje izvođenje Javascript programskog jezika na poslužiteljskoj strani. Poslužitelj je realiziran na malom prijenosnom računalu Raspberry Pi, što ga čini idealnim za kućnu upotrebu. Nastavno na to, modul koji pokreće sustav je napisan u C programskom jeziku te koristi i jednu *bash* skriptu koja se brine o slanju HTTP zathjeva. Na kraju, korisnička aplikacija je izrađena u Xamarin-u, *frameworku* za izradu mobilnih aplikacija u C# programskom jeziku. Iako ovaj sustav trenutno ne prikazuje veliku količinu podataka, građen je s naglaskom na skalabilnost kako bi sustav ostao stabilan i nakon dodavanja veće količine podataka.

Zbog svoje kompleksnosti sustav je testiran lokalno. Ustvrđeno je da bi sustav poslužio svojoj svrsi u slučaju da želimo pratiti specifični proces iz Linux operacijskog sustava, no ipak za širu upotrebu trebalo bi uložiti još neko vrijeme u razvoj i testiranje.

Popis literature

[1] Raspberry Pi Foundation, *About us*, https://www.raspberrypi.org/about/, Pristupano: 2022-09-07.

[2] Raspberry Pi, *Raspberry Pi OS*, https://www.raspberrypi.com/software/, Pristupano: 2022-09-07.

[3] OpenJS Foundation, *About Node.js [Na internetu]*, https://nodejs.org/en/about/, Pristupano: 2022-07-26.

[4] Express, *Express.js [Na internetu]*, expressjs.com, Pristupano: 2022-07-26.

[5] The PostgreSQL Global Development Group, *About PostgreSQL [Na internetu]*, https://www.postgresql.org/about/, Pristupano: 2022-07-26.

[6] Microsoft, *A tour of the C# language [Na internetu]*, https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/, Pristupano: 2022-07-26.

[7] Canonical Ltd., *Debian is the rock on which Ubuntu is built [Na internetu]*, https://ubuntu.com/community/debian, Pristupano: 2022-07-26.

[8] Lokesh Gupta, *What is REST [Na internetu]*, https://restfulapi.net/, Pristupano: 2022-08-28.

[9] Node.JS, *How to create an https server?* https://nodejs.org/en/knowledge/HTTP/servers/how-to-create-a-HTTPS-server/, Pristupano: 2022-09-01.

[10] The kernel development community, *Kernel modules*, https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_modules.html, Pristupano: 2022-09-02.

[11] ——, *Kernel API*, https://linux-kernel-labs.github.io/refs/heads/master/labs/kernel_api.html, Pristupano: 2022-09-02.

[12] ——, *Linux Kernel Makefiles*, https://www.kernel.org/doc/html/latest/kbuild/makefiles.html, Pristupano: 2022-09-02.

[13] A. Prakash i A. Alkabary, *Learn Bash Quickly: A Friendly Guide with Exercises to Easily Get Started with Bash Scripting*. Independently Published, 2020., ISBN: 9798686086746. adresa: https://books.google.hr/books?id=aH3zzQEACAAJ.

[14] Linuxtopia, *Read and Write a /proc File*, https://www.linuxtopia.org/online_books/Linux_Kernel_Module_Programming_Guide/x773.html, Pristupano: 2022-09-03.

[15] Microsoft, *Cross-platform with Xamarin*, https://dotnet.microsoft.com/en-us/apps/xamarin/cross-platform, Pristupano: 2022-09-07.

[16] *The Linux Kernel*, https://linux-kernel-labs.github.io/refs/heads/master/index.html, Pristupano: 2022-05-10.

[17] S. Patil, *What is a Makefile and how does it work? [Na internetu]*, https://opensource.com/article/18/8/what-how-makefile/, Pristupano: 2022-07-26.

[18] Microsoft, *Xamarin [Na internetu]*, https://dotnet.microsoft.com/en-us/apps/xamarin, Pristupano: 2022-07-26.

[19] Oracle, *VirtualBox [Na internetu]*, https://www.virtualbox.org/wiki/VirtualBox, Pristupano: 2022-07-26.

[20] *Linux Basics for Hackers: Getting Started with Networking, Scripting, and Security in Kali*. No Starch Press, 2018., ISBN: 9781593278557. adresa: https://books.google.hr/books?id=P1v6DwAAQBAJ.

[21] Linux Journal, Kedar Sovani, *Kernel Korner - Sleeping in the Kernel [Na internetu]*, https://www.linuxjournal.com/article/8144, Pristupano: 2022-08-20.

[22] LIRAN B.H, *LINUX KERNEL DEVELOPMENT – KERNEL MODULE PARAMETERS [Na internetu]*, https://devarea.com/linux-kernel-development-kernel-module-parameters/, Pristupano: 2022-08-30.

[23] ——, *LINUX KERNEL DEVELOPMENT – CREATING A PROC FILE AND INTERFACING WITH USER SPACE [Na internetu]*, https://devarea.com/linux-kernel-development-creating-a-proc-file-and-interfacing-with-user-space/, Pristupano: 2022-08-30.

Popis slika