SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1105

Razvoj metodologije finog mjerenja performansi na operacijskom sustavu Linux

Matej Filković

Zagreb, srpanj 2015.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 9. ožujka 2015.

Predmet: **Diplomski rad**

DIPLOMSKI ZADATAK br. 1105

Pristupnik: **Matej Filković (0165045504)**
Studij: Računarstvo
Profil: Programsko inženjerstvo i informacijski sustavi

Zadatak: **Razvoj metodologije finog mjerenja performansi na operacijskom sustavu Linux**

Opis zadatka:

Jedna od primjena operacijskog sustava Linux je u specijaliziranim uređajima čija zadaća je usmjeravanje i obrada velikih količina mrežnog prometa. Zbog specifičnosti takvih uređaja u odnosu na računala opće namjene, razvijene su i posebne distribucije za njih kao što je primjerice Yocto Linux. Od takvih sustava očekuju se visoke performanse koje se postižu optimiranjem dijelova koda za koje se ustanovi da unose neprihvatljivo kašnjenje. Međutim, kako bi se mogle raditi optimizacije nužno je prvo znati gdje su problemi a to se postiže odgovarajućim mjerenjima.
U sklopu diplomskog rada potrebno je proučiti Yocto Linux te ga instalirati unutar virtualizirane okoline. Razmotriti jezgru koja se koristi i ispitati mogućnost korištenja službene jezgre Linux operacijskog sustava. Razviti metodologiju finog mjerenja performansi mrežnog stoga i međuprocesne komunikacije u jezgri operacijskog sustava te proširiti mjerenje i na aplikacije. Korištenjem razvijene metodologije obaviti mjerenje vremena obrade paketa u jezgri operacijskog sustava i vrijeme koje se troši na međuprocesnu komunikaciju. Radu priložiti izvorni kod razvijenih i korištenih programa. Citirati korištenu literaturu i navesti dobivenu pomoć.

Zadatak uručen pristupniku: 13. ožujka 2015.
Rok za predaju rada: 30. lipnja 2015.

Mentor:

Doc. dr. sc. Stjepan Groš

Djelovođa:

Doc. dr. sc. Igor Mekterović

Predsjednik odbora za diplomski rad profila:

Prof. dr. sc. Krešimir Fertalj

Sadržaj

1. Uvod

Jedna od primjena operacijskog sustava Linux je u specijaliziranim uređajima. Od takvih se uređaja očekuju visoke performanse koje se postižu optimizacijom dijelova koda za koje se ustanovi da unose neprihvatljivo kašnjenje. Kako bi se mogle raditi optimizacije nužno je provesti odgovarajuća mjerenja. Mjerenje performansi je izazovan zadatak i zahtjeva temeljito poznavanje računalnog sklopovlja, operacijskog sustava i aplikacija. Većina današnjih procesora sadrži jedinicu za mjerenje performansi kojom je omogućeno praćenje širokog raspona događaja vezanih uz procesor. Korištenjem jedinice za mjerenje performansi moguće je za dijelove izvornog koda brojati cikluse signala takta i tako utvrditi unose li neprihvatljivo kašnjenje. Osim ciklusa, jedinicom za mjerenje performasni moguće je promatrati i neke druge parametre kako bi se ustanovili razlozi kašnjenja.

U sklopu diplomskog rada potrebno je proučiti Yocto Project. Cilj ovoga rada je razviti metodologiju finog mjerenja performansi jezgre operacijskog sustava i korisničkih programa.

Rad je podijeljen na sedam poglavlja. U drugom poglavlju dan je pregled procesora, hijerarhije memorije i jedinice za mjerenje performansi. Treće poglavlje daje pregled međuprocesne i mrežne komunikacije. Četvrtim poglavljem dan je pregled Yocto Projecta i njegovih mogućnosti. U petom poglavlju dan je pregled sučelja i alata za mjerenje performansi na operacijskom sustavu Linux. Šestim poglavljem opisana je razvijena metodologija finog mjerenja performansi te su prikazani rezultati mjerenja. Rad završava zaključkom u sedmom poglavlju.

2. Performanse procesora

2.1. Rad procesora

Procesor je najvažniji dio računala koji obavlja različite operacije nad podacima i upravlja radom istoga. Naziva se još i CPU (engl. *central processing unit*). U svom najosnovnijem obliku, rad procesora se odvija u tri osnovna koraka koji se stalno ponavljaju [15]:

- Dohvat naredbe (*engl. fetch*): Čitanje naredbe iz memorije.
- Dekodiranje naredbe (*engl. decode*): Prepoznavanje pojedine kombinacije bitova naredbi.
- Izvođenje naredbe (*engl. execute*): Obavljanje aktivnosti koje su zadane u naredbi.

Kod većine današnjih procesora postoje i mnoga proširenja ovih osnovnih faza koja su specifična za određenu arhitekturu. Signal takta (*engl. clock signal*) je digitalni signal koji upravlja svom logikom procesora. Izvođenje pojedine instrukcije procesora može trajati jedan ili više perioda signala takta koji se još nazivaju ciklusi (engl. cycles). Procesori obično imaju fiksnu frekvenciju signala takta, te tako procesor s frekvencijom taka od 3 GHz, izvodi 3 milijarde ciklusa po sekundi. Neki procesori imaju mogućnost mijenjanja frekvencije signala takta kako bi poboljšali performanse ili ih pak smanjili radi uštede energije.

Broj ciklusa po instrukciji (*engl. cycles per instruction - CPI*) je vrlo važna mjera za razumijevanje performansi procesora. Promatranjem ciklusa po instrukciji može se vrlo brzo primijetiti da procesor gubi vrijeme čekajući na dohvat podataka iz memorije. Moguće je promatrati i broj instrukcija po ciklusu (*engl. instructions per cycle - IPC*).

Prijelaz na više nezavisnih procesora po jednom mikroprocesoru doveo je do toga da se za procesor još koristi i naziv jezgra [13]. Kroz ovaj rad termini procesor, CPU i jezgra se koriste naizmjenično, dok se termin mikroprocesor

koristi za fizičku komponentu koja sadrži jedan ili vise procesora.

2.2. Hijerarhija memorije

Signal takta i broj procesora po mikroprocesoru, predstavlja veliki faktor u ukupnim performansama sustava. Performanse memorije su jednako ključne [18]. Faktor memorije koje naviše utječe na performanse procesora je vrijeme pristupa. Procesor može pristupi memoriji kako bi učitao podatak ili kako bi pohranio podatak u memoriju. Vrijeme pristupa memoriji je vrijeme proteklo od postavljanja zahtjeva procesora za obavljanjem operacije čitanje ili pisanja do obavljanja traženog zahtjeva.

Memoriju računala moguće je kategorizirati na primaranu (engl. *primary*) i sekundarnu memoriju (engl. *secondary*) [19]. Primarna memorija se koristi kod izvođenja programa i pohrane privremenih podataka. Karakterizira ju vrlo kratko vrijeme pristupa i nepostojanost (engl. *volatile*). Memorija je nepostojana ukoliko se njezin sadržaj gubi prilikom nestanka napajanja. Daljnju kategorizaciju primarne memorije moguće je načiniti podjelom na registre, priručnu (engl. *cache*) i glavnu memoriju (engl. *main memory*). Nasuprot primarnoj, sekundarnu memoriju karakterizira poveće vrijeme pristupa i postojanost.

2.2.1. Priručna memorija

Današnji procesori obično sadrže nekoliko razina priručne memorije čija je glavna svrha smanjenje ogromne razlike između brzine procesora i radne memorije. Procesori su obično upravljani signalom takta od nekoliko GHz-a i vrijeme pristupa glavnoj memoriji je obično u rangu od nekoliko stotina ciklusa. Procesor može biti značajno zadržan dok izvodi instrukciju koja zahtjeva dohvat ili spremanje podataka u glavnu memoriju.

Hijerarhija priručne memorije prisutna u današnjim računalima obično sadrži sljedeće razine priručne memorije:

- Priručna memorija prve razine (engl. *level 1 (L1) cache*)
- Međuspremnik preslika adresa (engl. *translation lookaside buffer - TLB*)
- Priručna memorija druge razine (engl. *level 2 (L2) cache*)
- Priručna memorija treće razine (engl. *level 3 (L3) cache*)

Kod višejezgrenih procesora obično svaki procesor ima vlastiti međuspremnik preslika adresa i priručne memorije prve i druge razine. Priručna memorija treće razine je obično dijeljena između više procesora.

Kako bi se naznačila uloga priručne memorije dana je tablica 2.1 kojom je prikazano vrijeme pristupa i kapacitet pojedinih razina memorije u današnjim poslužiteljskim računalima. Prema vremenima pristupa prikazanim u tablici 2.1 procesoru koji je upravljan sa signalom takta od 2 GHz potrebno je između 20 i 40 ciklusa kako bi pristupio priručnoj memoriji treće razine.

Tablica 2.1: Razine hijerarhije memorije u današnjim poslužiteljskim računalima [13]

	Veličina	Vrijeme pristupa
Registri	1000 bajta	300 ps
L1 cache	64 KB	1 ns
L2 cache	256 KB	3-10 ns
L3 cache	2-4 MB	10-20 ns
Glavna memorija	4-16 GB	50-100 ns
Magnetski disk	4-16 TB	5-10 ms

Prilikom pristupa adresi glavne memorije koja nije prisutna u priručnoj memoriji procesor dohvaća blok memorijskih adresa koje će biti pohranjene u priručnoj memoriji. Veličina ovog bloka naziva se veličina linije priručne memorije (engl. *cache line size*). Razumijevanje veličine linije priručne memorije vrlo je važno za razne optimizacije prilikom prevođenja izvornog koda.

2.3. Virtualni adresni prostor

Virtualni adresni je raspoloživi opseg adresa koji procesor može adresirati. Stvarna fizička veličina glavne memorije koja je izvedena u računalnu manja je ili jednaka virtualnom adresnom prostoru. Jedinica za upravljanje memorijom (engl. *memory managment unit*) odgovorna je za preslikavanje virtualnih adresa u fizičke. Jedinica za upravljanje memorijom koristi međuspremnik preslika adresa za ubrzanje postupaka preslikavanja. Ako traženo preslikavanje nije pronađeno u međuspremniku (*TLB miss*) potrebno ga je pronaći u glavnoj memoriji.

2.4. Jedinica za mjerenje performansi

Današnji procesori sadrže jedinicu za mjerenje performansi (engl. *Performance monitoring unit - PMU*) koja uključuje manji broj registara. Registri jedinice

za mjerenje performansi mogu biti programirani kako bi bilježili određene događaje prisutne u procesoru. Registri koji bilježe događaje se još nazivaju brojači (engl. *counters*). U kontekstu mjerenja performansi događaj predstavlja sve ono što se može bilježiti. Tako npr. izvođenje instrukcije predstavlja događaj vezan uz procesor. U korisničkim programima događaj npr. može biti poziv određene funkcije. Jedinice za mjerenje performansi su vrlo specifične za svaki procesor, te njihove mogućnosti i raspon događaja koje mogu pratiti može biti vrlo različit.

Jedinica za mjerenje performansi obično može biti korištena na dva načina:

- Brojanje (engl. *counting*): Svi događaji koji su se dogodili tijekom izvođenja su jednostavno prebrojani. Ako se promatra određena aplikacija ili dio aplikacije moguće je dobiti vrlo precizne informacije o njezinim performansama tijekom izvođenja. Brojanje se također može iskoristiti prilikom usporedbe dvije verzije programa. Neki od primjera događaja koje brojači jedinice za mjerenje performansi mogu bilježiti su: izvođenje instrukcije, pristup memoriji određene razine.
- Uzorkovanje (engl. *sampling*): Kad god brojač događaja poprimi vrijednost veću od definirane, jedinica generira prekid. Prilikom obrade prekida određenim alatom moguće je spremiti stanja registara. Iz stanja registara moguće je odrediti koji je dio aplikacije uzrokovao prekid.

Sadržaj registara koje obuhvaća jedinica za mjerenje performansi obično se ne može čitati iz korisničkog prostora. Linux jezgra iz toga razloga ima generičko sučelje za različite arhitektura procesora. Sučeljem je omogućeno transparentno programiranje jedinica za mjerenje performansi. Čitanje vrijednosti brojača je omogućeno korištenjem sustavskih poziva. Također, sučelje omogućava odvajanje događaja različitih procesa. Održavanjem brojača za pojedini proces unosi se dodatno opterećenje u rad sustava.

2.5. ARM Cortex-A7 MPCore

Za obavljanje dijela mjerenja korišten je Raspberry Pi 2 uređaj s Broadcom BCM2836 mikroprocesorom. Broadcom BCM2836 mikroprocesor je jedna od implementacija Cortex-A7 MPCore višejezgrenog procesora. Cortex-A7 MPCore je 32-bitni višejezgreni procesor koji implementira ARMv7-A arhitekturu i može imati između jednog do četiri procesora. Broadcom BCM2836 mikroprocesor sadrži četiri jezgre. Svaka jezgra sadrži jedinicu za mjerenje performansi. Jedinica

za mjerenje performansi sadrži brojač ciklusa i četiri brojača koji mogu bilježiti skup događaja dostupnih u procesoru. U nastavku je dan pregled najvažnijih registara za rad s jedinicom za mjerenje performansi.

PMCR registar

Performance Monitors Control Register - PMCR registar pruža detalje implementacije jedinice za mjerenje performansi (npr. broj implementiranih brojača). Registar omogućava konfiguriranje i kontroliranje brojača.

Važniji bitovi PMCR registra su:

- N, bitovi[15:11]: Vrijednost ovoga polja je broj implementiranih brojača.
- C, bit[2]: Pisanje jedinice na ovaj bit resetira brojač ciklusa.
- P, bit[1]: Pisanje jedinice na ovaj bit resetira brojače koji mogu bilježiti skup događaja dostupnih u procesoru.
- E, bit[0]: Pisanje jedinice na ovaj bit uključuje sve brojače.

Pristup PMCR registru omogućen je naredbama prikazanim u ispisu 2.1.

Ispis 2.1: Naredbe za rad s PMCR registrom

```
MRC p15, 0, <Rt>, c9, c12, 0 ; Procitaj PMCR u Rt
MCR p15, 0, <Rt>, c9, c12, 0 ; Zapisi Rt u PMCR
```

PMSELR registar

Performance Monitors Event Counter Selection Register - PMSELR odabire zadani brojač i obično se koristi u kombinaciji s registrom PMXEVCNTR kako bi se odredila vrijednost brojača. Pristup PMSELR registru omogućen je naredbama prikazanim u ispisu 2.2.

Ispis 2.2: Naredbe za rad s PMSELR registrom

```
MRC p15, 0, <Rt>, c9, c12, 5 ; Procitaj PMSELR u Rt
MCR p15, 0, <Rt>, c9, c12, 5 ; Zapisi Rt u PMSELR
```

PMXEVCNTR registar

Performance Monitors Event Count Register - PMXEVCNTR čita ili zapisuje vrijednost odabranog brojača. Brojač mora biti odabran korištenjem navedenog PMSELR registra. Pristup PMSELR registru omogućen je naredbama prikazanim u ispisu 2.3.

Ispis 2.3: Naredbe za rad s PMXEVCNTR registrom

```
MRC p15, 0, <Rt>, c9, c13, 2 : Procitaj PMXEVCNTR u Rt
MCR p15, 0, <Rt>, c9, c13, 2 : Zapisi Rt u PMXEVCNTR
```

PMXEVTYPER registar

Performance Monitors Event Type Select Register - PMXEVTYPER definira događaj koji uzrokuje uvećanje brojača. Brojač mora biti odabran korištenjem PMSELR registra. Pristup PMXEVTYPER registru omogućen je naredbama prikazanim u ispisu 2.4.

Ispis 2.4: Naredbe za rad s PMXEVTYPER registrom

```
MRC p15, 0, <Rt>, c9, c13, 1 : Procitaj PMXEVTYPER u Rt
MCR p15, 0, <Rt>, c9, c13, 1 : Zapisi Rt u PMXEVTYPER
```

PMCNTENSET registar

Performance Monitors Count Enable Set Register - PMCNTENSET uključuje brojač ciklusa i ostale implementirane brojače. Kako bi registar PMCNTENSET mogao uključiti brojače bit `E` PMCR registra mora biti postavljen na jedinicu. U suprotnom, registar PMCNTENSET nema utjecaja na brojače. Čitanjem PMCNTENSET registra moguće je saznati koji su brojači uključeni. Pristup PMCNTENSET registru omogućen je naredbama prikazanim u ispisu 2.5.

Ispis 2.5: Naredbe za rad s PMCNTENSET registrom

```
MRC p15, 0, <Rt>, c9, c12, 1 : Procitaj PMCNTENSET u Rt
MCR p15, 0, <Rt>, c9, c12, 1 : Zapisi Rt u PMCNTENSET
```

Primjer rada s PMU registrima

Dodatak D sadrži jezgrin modul koji radi s registrima jedinice za mjerenje performansi. Ispisom 2.6 dana je funkcija `inicijaliziraj_pmu()`. Prva naredba funkcije `inicijaliziraj_pmu()` pristupa PMCR registru. Pohranom vrijednosti `7` u PMCR registar na bitove `E`, `P` i `C` upisane su jedinice i obavljeno je resetiranje i uključivanje svih brojača. Druga naredba funkcije `inicijaliziraj_pmu()` pristupa PMSELR registru koji služi za odabir brojača. U ovom slučaju odabran je prvi brojač. Treća naredba funkcije `inicijaliziraj_pmu()` pristupa registru PMXEVTYPER koji služi za definiranje događaja koji će uzrokovati uvećanje

brojača. Kako je drugom naredbom odabran prvi brojač, za događaj koji uzrokuje uvećanje prvog brojača definiran je događaj čiji je broj `8`. Broj `8` označuje događaj izvođenja instrukcije. Četvrta naredba funkcije `inicijaliziraj_pmu()` pristupa registru PMCNTENSET i uključuje prvi brojač.

Zbog poziva funkcije `on_each_cpu()` brojač je postavljen na svim procesorima. Funkcija `ulazni_rukovatelj()` čita vrijednost brojača naredbama prikazanim u ispisu 2.7. Prva naredba iz ispisa 2.7 pristupa PMSELR registru i odabire prvi brojač. Druga naredba iz ispisa 2.7 pristupa PMXEVCNTR registru i pohranjuje vrijednost brojača odabranog PMSELR registrom u varijablu `prvo_ocitanje`.

Ispis 2.6: Prikaz funkcije `inicijaliziraj_pmu()`

```
void inicijaliziraj_pmu(void *d)
{
    asm volatile("mcr p15, 0, %0, c9, c12, 0" : : "r"(0x00000007));
    asm volatile("mcr p15, 0, %0, c9, c12, 5" : : "r"(0x00000000));
    asm volatile("mcr p15, 0, %0, c9, c13, 1" : : "r"(0x00000008));
    asm volatile("mcr p15, 0, %0, c9, c12, 1" : : "r"(0x00000001));
}
```

Ispis 2.7: Naredbe za čitanje vrijednosti brojača

```
asm volatile("mcr p15, 0, %0, c9, c12, 5" : : "r"(0x0));
asm volatile("mrc p15, 0, %0, c9, c13, 2" : "=r"(prvo_ocitanje));)
```

3. Međuprocesna i mrežna komunikacija

3.1. Međuprocesna komunikacija

Međuprocesna komunikacija (engl. *Interprocess communication - IPC*) označava mehanizme razmjene podataka među procesima i dretvama te mehanizme za sinkronizaciju operacija istih. U ovom poglavlju dan je pregled dijeljene memorije i semafora. Dijeljena memorija omogućava razmjenu podataka među procesima, dok semafori omogućavaju koordinaciju aktivnosti prilikom pristupa istoj. U svim priloženim programskim primjerima za komunikaciju dijeljenom memorijom i semaforima koristi se POSIX API. Osim POSIX API-ja, za komunikaciju dijeljenom memorijom i semaforima postoji i stariji System V API.

3.1.1. Dijeljena memorija

Dijeljena memorija je vrlo jednostavna metoda koja omogućuje nepovezanim procesima da pristupaju istim memorijskim lokacijama. Nakon promjene zapisa na dijeljenoj memorijskoj lokaciji, izmjene su odmah vidljive svim procesima koji sudjeluju u komunikaciji. Dijeljena memorija je najbrži oblik razmjene podataka u međuprocesnoj komunikaciji. Pristup procesa dijeljenoj memoriji je jednako brz kao pristup vlastitoj memoriji i ne zahtijeva sustavski poziv i prelazak u jezgrin način rada. Također, nema kopiranja između jezgrinog i procesnog adresnog prostora. Zbog toga što jezgra ne provodi sinkronizaciju prilikom pristupa dijeljenoj memoriji, potrebno je osigurati mehanizam sinkronizacije unutar aplikacija. Tako npr. proces ne smije čitati iz dijeljene memorije sve dok drugi proces pišu u istu. Jedan od načina da se izbjegnu problemi sinkronizacije je korištenje semafora koji su objašnjeni u sljedećem odjeljku.

Rad s dijeljenom memorijom

POSIX terminologija definira objekte dijeljene memorije, dok System V terminologija definira segmente dijeljene memorije. Kako je u radu odabran POSIX API za dijeljenu memoriju, koristi se i POSIX terminologija. U nastavku su prikazane funkcije POSIX API-ja za obavljanje operacija nad dijeljenom memorijom.

Kreiranje i otvaranje objekta dijeljene memorije

Funkcija `shm_open()` kreira i otvara novi objekt dijeljene memorije ili samo otvara postojeći. Prototip funkcije je dan u ispisu 3.1 [14]. Argument `name` identificira objekt koji će biti stvoren ili otvoren. `oflag` argument je maska bitova koji modificiraju izvođenje funkcije, npr. otvara se postojeći objekt dijeljene memorije ili se kreira novi objekt. Tablica 3.1 sadrži neke od mogućih vrijednosti argumenta `oflag`. Parametar `mode` služi za postavljanje dopuštenja za objekt prilikom kreiranja. Povratna vrijednost funkcije je opisnik objekta.

Tablica 3.1: Neke od vrijednosti argumenta `oflag`

Vrijednost	Opis
`O_CREAT`	Stvori objekt ako već ne postoji
`O_RDONLY`	Otvori objekt samo za čitanje
`O_RDWR`	Otvori objekt za čitanje i pisanje

Ispis 3.1: Prototip funckije `shm_open()`

```
#include <fcntl.h>          /* Definira O_* ogranicenja */
#include <sys/stat.h>       /* Definira mode ogranicenja */
#include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode);
```

Nakon što je objekt dijeljene memorije kreiran, njegova veličina je nula i potrebno je pozvati funkciju `ftruncate()`. Objekt dijeljene memorije s postavljenom veličinom je potrebno preslikati u adresni prostor procesa pozivom funkcije `mmap()`. Prilikom poziva funkcije `mmap()` potrebno je proslijediti opisnik objekta i specificirati `MAP_SHARED` argument. Nakon što je objekt dijeljene memorije preslikan u adresni prostor procesa, opisnik objekta je moguće zatvoriti bez utjecaja na preslikavanje.

Ako je potrebno proširiti ili smanjiti veličinu objekta dijeljene memorije, moguće je naknadno pozvati funkciju `ftruncate()`. Međutim, opisnik objekta ne smije biti zatvoren. Objekti dijeljene memorije ostaju prisutni sve dok ih se eks-

plicitno ne ukloni ili do ponovnog podizanja sustava. Ako nije potreban, objekt dijeljene memorije bi trebao biti uklonjen pozivom funkcije `shm_unlink()` čiji je prototip dan u ispisu 3.2.

Uklanjanje objekta dijeljene memorije

Funkcija `shm_unlink()` uklanja objekt dijeljene memorije čiji je naziv predan kao argument. Uklanjanje objekta dijeljene memorije ne utječe na postojeća preslikavanja jer ona ostaju važeća sve dok procesi ne završe s izvođenje ili pozovu funkciju `munmap()`. Funkcijom `shm_unlink()` sprječava se pozivanje funkcije `shm_open()` za otvaranje postojećeg objekta dijeljene memorije. Nakon što svi procesi koji su koristili objekt dijeljene memorije završe s izvođenjem ili pozovu funkciju `munmap()`, objekt dijeljene memorije se uklanja.

Ispis 3.2: Prototip funkcije `shm_unlink()`

```
#include <sys/mman.h>

int shm_unlink(const char * name );
```

Primjer rada s dijeljenom memorijom

Dodatak C sadrži dva primjera programa koji koriste dijeljenu memoriju. Program iz ispisa C.2 za parametre prima naziv objekta dijeljene memorije i poruku koju će upisati kreirani objekt. Prevođenje programa potrebno je obaviti naredbom prikazanom u 3.3. Parametar `ime_datoteke` je naziv datoteke u koju je pohranjen sadržaj programa. Primjer naredbe za izvođenje programa dan je u ispisu 3.4. Izvođenjem naredbe iz ispisa 3.4 kreirat će se objekt dijeljene memorije s nazivom *objekt* u koji će biti upisan znakovni niz *Poruka koju ce procitati drugi program!*.

Ispis 3.3: Prevođenje programa koji kreira objekt dijeljene memorije

```
gcc ime_datoteke.c -o ime_datoteke -lrt
```

Ispis 3.4: Primjer naredbe za izvođenje programa koji kreira objekt dijeljene memorije

```
./ime_datoteke objekt "Poruka koju ce procitati drugi program!"
```

Program iz ispisa C.1 kao parametar prima naziv objekta dijeljene memorije čiji će sadržaj pročitati. Prevođenje programa potrebno je obaviti izvođenjem naredbe prikazanom u 3.3. Parametar `ime_datoteke` je naziv datoteke u koje je

pohranjen sadržaj programa. Primjer naredbe za izvođenje programa dan je u ispisu 3.5.

Ispis 3.5: Primjer naredbe za izvođenje programa koji čita sadržaj objekta dijeljene memorije

```
./ime_datoteke objekt
```

3.1.2. Semafori

Za razliku od prethodno prikazane dijeljene memorije, semafori ne omogućuju prijenos podataka između procesa, već služe za sinkronizaciju operacija. Jedan od najčešćih primjera korištenja semafora je sinkronizacija pristupa dijeljenoj memoriji. Semafor je cijeli broj čija vrijednost nikada ne smije biti manja od nule. Održava ga jezgra operacijskog sustava i putem sustavskih poziva omogućava obavljanje sljedećih operacija:

- Postavljanje vrijednosti semafora na pozitivni cijeli broj
- Uvećanje vrijednosti semafora
- Umanjenje vrijednosti semafora
- Čekanje na vrijednost semafora

Zadnje dvije operacije mogu blokirati pozivajući proces. Prilikom umanjena vrijednosti semafora, jezgra će blokirati svaki pokušaj koji bi rezultirao u vrijednosti semafora manjoj od nula. Na sličan način, čekajući da vrijednosti semafora bude jednaka nuli, proces ostaje blokiran sve dok neki drugi proces ne postavi vrijednost semafora na nula.

Postoje dva tipa POSIX semafora [14, str. 1089]:

- Imenovani semafori (engl. *Named semaphores*): Semafori ovog tipa imaju ime. Pozivom funkcije `sem_open()` s istim imenom kao argumentom, dva nepovezana procesa mogu pristupiti istom semaforu.
- Neimenovani semafori (engl. *Unnamed semaphores*): Semafori ovoga tipa nemaju ime, već se nalaze na unaprijed definiranim lokacijama u memoriji. Neimenovani semafori mogu biti dijeljeni medu procesima ili grupom dretvi. Ako su dijeljeni medu procesima neimenovani semafori se moraju nalazi u dijeljenoj memoriji.

3.1.3. Rad s imenovanim semaforima

U nastavku su prikazane funkcije POSIX API-ja za obavljanje operacija nad imenovanim semaforima.

Kreiranje i otvaranje imenovanog semafora

Za kreiranje i otvaranje semafora koristi se funkcija `sem_open()` čiji je prototip dan u ispisu 3.6 [14]. Ako se funkcija `sem_open()` koristi za otvaranje postojećeg semafora, tada poziv iste zahtjeva argument `name`. Argument `oflag` mora biti postavljen na `0`. Prilikom korištenja funkcije `sem_open()` za kreiranje novog semafora, potrebno je predati argumente `name`, `mode` i `value`. Argument `oflag` mora biti postavljen na `O_CREAT`. Argumentom `mode` specificiraju se dozvole korištenja na semafor, dok je argument `value` inicijala vrijednost koja će biti pridružena semaforu.

Ispis 3.6: Prototip funkcije `sem_open()`

```
#include <fcntl.h>        /* Definira O_* ogranicenja */
#include <sys/stat.h>     /* Definira mode ogranicenja */
#include <semaphore.h>


sem_t *sem_open(const char *name , int oflag, ...
                mode_t mode , unsigned int value );
```

Čekanje na imenovani semafora

Funkcija `sem_wait()` umanjuje za jedan vrijednost semafora predanog kao parametar. Prototip funkcije je dan u ispisu 3.7 [14]. Ako je vrijednost semafora predanog kao argumenta veća od nula, funkcija `sem_wait()` odmah završava s izvođenjem. Ako je vrijednost predanog semafora jednaka nula, funkcija blokira pozivajući proces sve dok vrijednost semafora ne postane veća od nula.

Ispis 3.7: Prototip funkcije `sem_wait()`

```
#include <semaphore.h>

int sem_wait(sem_t * sem );
```

Uvećavanje vrijednosti semafora

Funkcija `sem_post` uvećava za jedan vrijednost semafora predanog kao parametar. Ako je vrijednost predanog semafora bila 0, neki proces je blokiran. Pozivom

funkcije `sem_post` blokirani proces nastavlja s izvođenjem. Ako je više procesa blokirano oni se tada raspoređuju prema *Round Robin* pravilu raspodjele [14]. Prototip funkcije `sem_post` je dan u ispisu 3.8.

Ispis 3.8: Prototip funkcije `sem_post()`

```
#include <semaphore.h>

int sem_post(sem_t * sem );
```

Zatvaranje imenovanog semafora

Nakon što proces otvori imenovani semafor, jezgra bilježi asocijaciju procesa i semafora. Ako se asocijacija želi prekinuti, potrebno je pozvati funkciju `sem_close()` koja će otpustiti sve resurse koji su bili asocirani sa semaforom za pozivajući proces. Prototip funkcije `sem_close()` je dan u ispisu 3.9 [14]. Otvoreni semafori su automatski zatvoreni prilikom završetka procesa. Zatvaranje semafora ne uklanja isti, već je potrebno pozvati `sem_unlink()` funkciju.

Ispis 3.9: Prototip funkcije `sem_open()`

```
#include <semaphore.h>

int sem_close(sem_t * sem );
```

Uklanjanje imenovanog semafora

Funkcija `sem_unlink()` uklanja semafor čije je ime predano kao parametar. Semafor se ne uklanja odmah, nego nakon što ga svi procesi zatvore. Prototip funkcije `sem_unlink()` je dan u ispisu 3.10

Ispis 3.10: Prototip funkcije `sem_unlink()`

```
#include <semaphore.h>

int sem_unlink(const char * name );
```

Primjer rada s imenovanim semaforima

Dodatak F sadrži primjer program koji koristi semafor. Program iz ispisa F.1 za parametre prima naziv semafora, vrijeme čekanja i oznaku kojom je definirano hoće li program kreirati semafor ili otvoriti postojeći. Prevođenje programa potrebno je obaviti naredbom prikazanom u ispisu 3.11. Parametar `ime_datoteke`

je naziv datoteke u koju je pohranjen sadržaj programa. Primjer naredbe za izvođenje programa dan je u ispisu 3.12. Izvođenjem naredbe iz ispisa 3.12 kreirat će se semafor s nazivom *moj_semafor*. Ako se umjesto oznake `k` preda oznaka `o` pokušat će se otvoriti postojeći semafor čije je ime predano kao parametar.

Ispis 3.11: Prevođenje programa koji radi sa semaforom

```
gcc ime_datoteke.c -o ime_datoteke -lpthread
```

Ispis 3.12: Primjer naredbe za izvođenje programa koji radi sa semaforom

```
./ime_datoteke moj_semafor 3 k
```

3.2. UDP komunikacija priključnicama

Priključnica je pristupna točka preko koje korisnički program šalje podatke u mrežu i iz koje čita primljene podatke. *User Datagram Protocol* (UDP) je beskonekcijski transportni protokol za prijenos nezavisnih paketa (datagrama). Omogućuje nepouzdanu komunikaciju uz jednostavnu nadogradnju protokola IP jer nema ugrađene mehanizme za detekciju i retransmisiju datagrama.

Kreiranje priključnice za komunikaciju UDP-om prikazano je u ispisu 3.13 [6]. Obično se poslužitelji vežu na dobro poznatu adresu i klijenti iniciraju komunikaciju slanjem datagrama na tu adresu. Za povezivanje na dobro poznatu adresu potrebno je pozvati funkciju `bind()`. Primjer korištenja `bind()` funkcije za povezivanje na adresu prikazan je u ispisu 3.14 [6]. Klijente obično ne zanima koja će im se adresa dodijeliti, već im jezgra dodijeli bilo koji slobodni.

Ispis 3.13: Kreiranje priključnice za komunikaciju UDP-om

```
int socket(int family,int type,int proto);

int sock;
sock = socket( PF_INET, // IPv4
               SOCK_DGRAM, // datagram prikljucnica
               0); // standardni datagram protokol: UDP
```

Ispis 3.14: Primjer povezivanja priključnice na dobro poznatu adresu

```
int sock;
struct sockaddr_in adresa;

mysock = socket(PF_INET, SOCK_DGRAM, 0);
memset(&adresa, 0, sizeof(adresa));
myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(8888);
myaddr.sin_addr.s_addr = INADDR_ANY;

bind(adresa, (struct sockaddr *)&adresa, sizeof(adresa));
```

Slanje podataka UDP-om obavlja se korištenjem `sendto()` funkcije prikazane u ispisu 3.15 [6]. Parametri funkcije su sljedeći:

- `sockfd`: Opisnik priključnice kojom se žele poslati podaci.
- `buff`: Adresa podataka za slanje duljine `nbyets`.
- `to`: Pokazivač na strukturu `sockaddr` s adresom odredišta.

Povratna vrijednost funkcije je broj poslanih okteta, međutim to je broj okteta prihvaćenih od operacijskog sustava za slanje u datagramu, a ne broj okteta koji su stigli do odredišta [6].

Ispis 3.15: Funkcija za slanje podataka UDP-om

```
ssize_t sendto( int sockfd,
                void *buff,
                size_t nbytes,
                int flags,
                const struct sockaddr* to,
                socklen_t addrlen);
```

Prijem podataka UDP-om obavlja se korištenjem `recvfrom()` funkcije prikazane u ispisu 3.16 [6]. Parametri funkcije su sljedeći:

- `sockfd`: UDP priključnica kojom se žele primiti podatci.
- `buff`: Adresa na koju će se pohraniti primljeni podatci.
- `nbytes`: Maksimalan broj bajtova koji se može pohraniti da danu adresu.
- `from`: Popunjava se adresom pošiljatelja prilikom primitka datagrama.
- `fromaddrlen`: Duljina predanog parametra `from`.

Povratna vrijednost funkcije je broj okteta pohranjenih na adresu `buf`. Funkcija `recvfrom` je blokirajuća, odnosno pozivajući proces ostaje blokiran sve dok ne primi datagram.

Ispis 3.16: Funkcija za prijem podataka UDP-om

```
ssize_t recvfrom( int sockfd,
                  void *buff,
                  size_t nbytes,
                  int flags,
                  struct sockaddr* from,
                  socklen_t *fromaddrlen);
```

4. Yocto Project

4.1. Poky referentni sustav

Yocto Project je projekt otvorenog koda koji pruža predloške, alate i metode za kreiranje prilagođenih Linux distribucija namijenjenih za ugrađena računala. Poky je referentni sustav Yocto Projecta i čini ga kolekcija alata i meta-podataka. Platformski je neovisan i omogućuje prevođenje za druge arhitekture. Komponente Poky referentnog sustava su [17]:

- BitBake: Sustav za prevođenje koji interpretira meta-podatke kako bi preuzeo, konfigurirao i izgradio slike Linux operacijskog sustava.
- OpenEmbedded Core: Kolekcija osnovnih meta-podataka. Meta-podaci su organizirani u recepte (engl. *recipes*) i slojeve (engl. *layers*). Recepti opisuju kako pribaviti, konfigurirati i izraditi aplikaciju ili sliku operacijskog sustava. Sloj je skup recepata koji zajedno imaju neku svrhu.
- `meta-yocto` i `meta-yocto-bsp`: Slojevi meta-podataka koji proširuju OpenEmbedded Core kolekciju.

Yocto Project ne predstavlja konačan skup alata i slojeva, već pruža osnovicu na koju se dodaju prilagođeni i specifični slojevi. Poky sustav kao skup osnovnih alata i slojeva predstavlja glavni i osnovni element Yocto Projecta.

4.2. Postavljanje Poky referentnog sustava

Proces postavljanja Poky referentnog sustava ovisi o korištenoj Linux distribuciji. Poky podržava određeni skup Linux distribucija i preporučuje se korištenje jedne od podržanih kako bi se izbjegli određeni probleme koji bi se mogli pojaviti prilikom korištenja distribucija koje se ne nalaze u skupu podržanih. Za potrebe ovoga rada korištena je distribucija Ubuntu 14.04 (LTS).

Za sve prikazane naredbe podrazumijeva se izvršavanje u standardnom naredbenom retku odabrane distribucije. Kako bi se instalirali paketi potrebni za rad s Poky sustavom, potrebno je izvršiti narednu prikazanu u ispisu 4.1. Nakon instalacije potrebnih paketa moguće je u odabranom direktoriju preuzeti izvorni kod Poky sustava. Preuzimanje se obavlja izvođenjem naredbe prikazane u ispisu 4.2.

Ispis 4.1: Naredba za instalaciju potrebnih paketa

```
sudo apt-get install gawk wget git-core diffstat unzip texinfo \
build-essential chrpath libsdl1.2-dev xterm
```

Ispis 4.2: Naredba za preuzimanje repozitorija Poky sustava

```
git clone git://git.yoctoproject.org/poky
```

Nakon što proces preuzimanja repozitorija Poky sustava završi, moguće je pronaći direktorij `poky` sa sljedećim sadržajem:

- `bitbake/`: Poddirektorij koji sadrži sve skripte koje koristi alat BitBake.
- `documentation/`: Poddirektorij koji sadrži dokumentaciju Yocto Projecta.
- `meta/`: Poddirektorij koji sadrži OpenEmbedded Core kolekciju metapodatka.
- `meta-skeleton/`: Poddirektorij koji sadrži predloške za razvoj BSP-a (*Board Support Package*) i prilagodbu Linux jezgre.
- `meta-yocto/`: Poddirektorij koji sadrži konfiguracijske meta-podatke za Poky referentnu distribuciju.
- `meta-yocto-bsp/`: Poddirektorij koji sadrži konfiguracijske datoteke za referentni BSP.
- `LICENSE`: Datoteka koja sadrži licencu pod kojom je Poky sustav distribuiran.
- `oe-init-build-env`: Skripta za postavljanje OpenEmbedded razvojnog okruženja. Skripta inicijalizira direktorij koji se koristiti prilikom prevođenja. Nakon što proces prevođenja završi, inicijalizirani direktorij sadržava izgrađene aplikacije i slike.
- `scripts/`: Poddirektorij koji sadrži skripte korištene za postavljanje okruženja i razvojnih alata.

4.3. Korisnička konfiguracija

Nakon što se izvrši skripta `oe-init-build-env` s predanim parametrom naziva direktorija, obavlja se inicijalizacija direktorija i postavljaju se varijable okoline potrebne za proces izgradnje. Unutar inicijaliziranog direktorija postoji poddirektorij `conf/` koji sadrži specifične postavke. Datoteke unutar direktorija `conf/` kojima su definirane postavke su:

- `bblayers.conf`: Definira slojeve koji se koriste prilikom izrade slike.
- `local.conf`: Pruža velik broj osnovnih varijabli kojima je omogućena prilagodba okruženja za izradu slike. Primjeri varijabli su:
  - `MACHINE`: Definira uređaj za koji se izrađuje distribucija.
  - `DL_DIR`: Definira direktorij koji se koristiti prilikom preuzimanja potrebnih resursa.
  - `TMPDIR`: Definira direktorij u koji se pohranjuju izrađeni resursi.

4.4. BSP sloj

Board Support Package (BSP) sloj je kolekcija meta-podataka koji definiraju kako podržati određeni uređaj, skup uređaja ili platformu. BSP sloj uključuje podršku za specifične mogućnosti uređaja i konfiguraciju Linux jezgre s potrebnim upravljačkim programima. Također, BSP sloj može uključiti i dodatne programske komponente.

Ako se slijedi konvencija Yocto Projecta, BSP sloj se može promatrati kao bazni direktorij u kojemu su datoteke pohranjene prema određenoj strukturi. Yocto Project koristi konvenciju imenovanja BSP sloja kao `meta-bsp_ime`, gdje `bsp_ime` predstavlja naziv uređaja ili platforme. Ispis 4.9 prikazuje uobičajenu strukturu datoteka unutar BSP sloja. Iako se strukturu prikazanu u ispisu 4.9 može smatrati standardnom, stvarna struktura pojedinih BSP slojeva može se razlikovati.

Ispis 4.3: Uobičajena struktura datoteka unutar BSP sloja

```
meta-bsp_ime/bsp_license_file
meta-bsp_ime/README
meta-bsp_ime/README.sources
meta-bsp_ime/binary/bootable_images
meta-bsp_ime/conf/layer.conf
meta-bsp_ime/conf/machine/*.conf
meta-bsp_ime/recipes-bsp/*
meta-bsp_ime/recipes-core/*
meta-bsp_ime/recipes-graphics/*
meta-bsp_ime/recipes-kernel/linux/linux-yocto_kernel_rev.bbappend
```

Važnije komponente predloženog formata BSP sloja prikazanog u ispisu 4.9 su:

- `conf/layer.conf`: Definira sadržaj sloja i strukturu datoteka. Sadrži informacije o tome kako bi alat BitBake trebao koristiti sloj.
- `conf/machine/*.conf`: Datoteke koje sadrže informacije o uređaju
- `recipes-kernel/linux/linux-yocto*.bbappend`: Datoteke koje sadrže specifične promjene nad Yocto verzijom Linux jezgre.

4.5. Izrada slike za virtualiziranu okolinu

Termin *slika operacijskog sustava* označava datoteku koja sadrži operacijski sustav, programe i njihove datoteke. Za potrebe kreiranja slike namijenjene za korištenje u virtualiziranoj okolini, unutar direktorija `poky` kreirat će se direktorij `build_virtual`. Kreiranje direktorija i postavljanje okruženja potrebno je obaviti unutar direktorija `poky` izvođenjem naredbe prikazane u ispisu 4.4.

Ispis 4.4: Naredba za postavljanje okruženja

```
source oe-init-build-env build_virtual
```

Nakon što je izvedena naredba iz ispisa 4.4, postavljeno je okruženje i za trenutni direktorij naredbenog retka postavljen je `build_virtual`. Za uključivanje razvojnih alata i alata za mjerenje performansi potrebno je u datoteci `conf/local.conf` pronaći varijablu `EXTRA_IMAGE_FEATURES` i postaviti ju kako što je prikazano ispisom 4.5. Opcije dodane u varijablu `EXTRA_IMAGE_FEATURES` su:

- `debug-tweaks`: Omogućava da slika bude prilagođena za razvoj i otklanjanje pogrešaka.

– `tools-profile`: Uključuje alate za otklanjanje pogrešaka (engl. *debugging*) i praćenje (engl. *tracing*) u sliku.

– `tools-sdk`: Uključuje alate za razvoj aplikacija u sliku.

Ispis 4.5: Postavke za uključivanje alata u sliku

```
EXTRA_IMAGE_FEATURES = "debug-tweaks tools-profile tools-sdk"
```

U datoteci `conf/local.conf` moguće je pronaći prethodno spomenutu varijablu `MACHINE` čija je zadana vrijednost `qemux86`. Vrijednošću `qemux86` za ciljani uređaj je postavljen emulator x86 arhitekture procesora. Nakon što je sadržaj ispisa 4.5 dodan u datoteku `conf/local.conf` potrebno je pokrenuti proces izrade slike izvođenjem naredbe dane ispisu 4.6. Proces izrade slike je poduži te je potrebno omogućiti internetsku vezu kako bi se mogli preuzeti potrebni resursi.

Ispis 4.6: Naredba za pokretanje procesa izrade slike

```
bitbake core-image-minimal
```

Nakon što proces izrade slike završi, sliku je moguće pokrenuti izvođenjem naredbe dane u ispisu 4.7. Izvođenje slike u virtualiziranoj okolini prikazano je na slici 4.1.

Ispis 4.7: Naredba za pokretanje slike u virtualiziranoj okolini

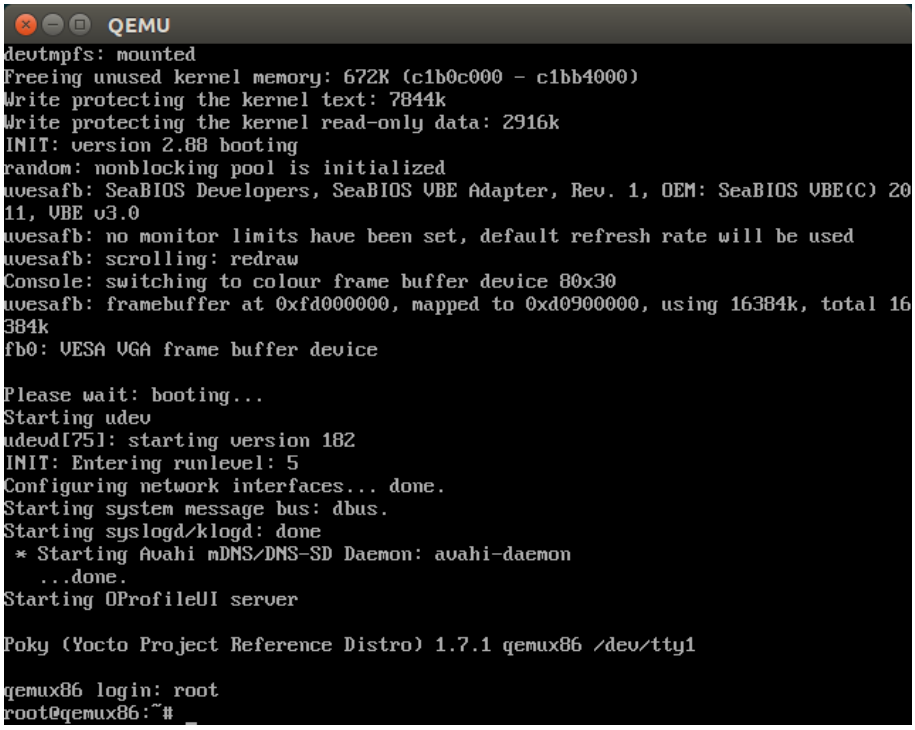
```
runqemu qemux86
```

4.6. Izrada BSP sloja sa službenom Linux jezgrom

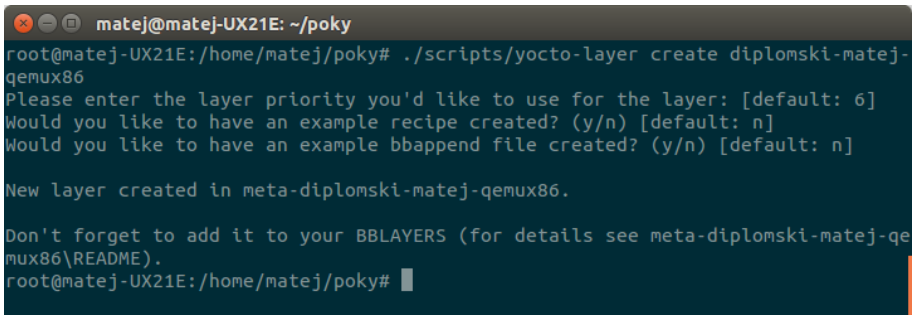
Za pokretanje procesa izrade BSP sloja potrebno je unutar direktorija Poky referentnog sustava izvesti naredbu prikazanu ispisom 4.8. Parametar `<ime_sloja` određuje naziv sloja. Nakon što je izvedena naredba iz ispisa 4.8, slijedi iterativni postupak kojim je moguće unijeti određene postavke. Primjer izrade sloja prikazan je slikom 4.2. Inicijalna struktura kreiranog sloja prikazana je u ispisu 4.9. U kreirani sloj je potrebno dodati definiciju uređaja i recepte za jezgru. Inicijalni sloj popunjen definicijom i receptom za jezgru nalazi se u prilogu, dok je njegova struktura prikazana u ispisu 4.10.

Ispis 4.8: Naredba za pokretanje izrade BSP sloja

```
./scripts/yocto-layer create <ime_sloja>
```



Slika 4.1: Izvođenje slike u virtualiziranoj okolini



Slika 4.2: Prikaz izrade sloja

Ispis 4.9: Inicijalna struktura kreiranog BSP sloja

```
meta-diplomski-matej-qemux86
meta-diplomski-matej-qemux86/README
meta-diplomski-matej-qemux86/conf
meta-diplomski-matej-qemux86/conf/layer.conf
meta-diplomski-matej-qemux86/COPYING.MIT
```

Ispis 4.10: Struktura popunjenog BSP sloja

```
./recipes-kernel
./recipes-kernel/linux
./recipes-kernel/linux/linux-sluzbena-jezgra.bb
./recipes-kernel/linux/linux-sluzbena-jezgra
./recipes-kernel/linux/linux-sluzbena-jezgra/defconfig
./conf
./conf/machine
./conf/machine/diplomski-matej-qemux86.conf
./conf/layer.conf
./README
./COPYING.MIT
```

4.6.1. Recept za Linux jezgru

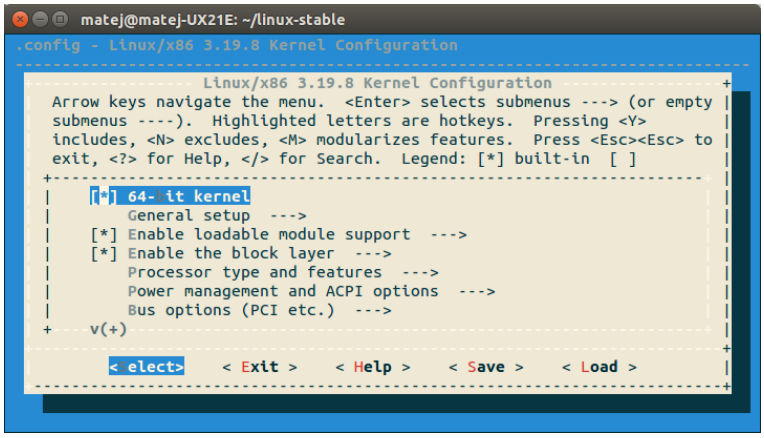
Za preuzimanje repozitorija službene Linux jezgre potrebno je u odabranom direktoriju izvesti naredbu prikazanu u ispisu 4.11.

Ispis 4.11: Naredba za preuzimanje repozitorija službene Linux jezgre

```
git clone
    git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
```

Nakon što je repozitorij službene Linux jezgre preuzet, moguće je vidjeti popis udaljenih grana (engl. *remote branches*) repozitorija izvođenjem naredbe prikazane u ispisu 4.12. Udaljene grane predstavljaju verzije Linux jezgre. Za rad s određenom verzijom Linux jezgre potrebno je kopirati udaljenu granu lokalno. Naredba za kopiranje 3.19 verzije Linux jezgre dana je u ispisu 4.13.

Prije nego što se Linux jezgra može prevesti u izvršni format, potrebno je konfigurirati jezgrin kod. Rezultat konfiguracijskog procesa je `.config` datoteka. Za izradu konfiguracijske datoteke moguće je koristiti sučelje `make menuconfig` za koje potreban repozitorij Linux jezgre, alat `make`, prevoditelj programskog jezika C i `ncurses` programska knjižica. Sučelje se pokreće unutar repozitorija Linux jezgre istoimenom naredbom. Primjer korištenja sučelja `make menuconfig` dan je na slici 4.3. Osim sučelja `make menuconfig` moguće je koristi i sučelje `make defconfig` koje kreira zadane postavke za određenu arhitekturu procesora.



Slika 4.3: Primjer korištenja sučelja `make menuconfig`

Ispis 4.12: Naredba za pregled udaljenih grana repozitorija

```
git branch -r
```

Ispis 4.13: Primjer naredbe za kopiranje udaljene grane

```
git checkout -b diplomski-linux-3.19 remotes/origin/linux-3.19.y
```

Datoteka `linux-sluzbena-jezgra.bb` unutar izgrađenog BSP definira recept za jezgru. Sadržaj datoteke `linux-sluzbena-jezgra.bb` je dan u ispisu 4.14. Četvrtom linijom ispisa 4.14 definirana je lokacija repozitorija Linux jezgre. Petom linijom uključene su konfiguracijske postavke koje su potrebne prilikom prevođenja jezgre. Priložena datoteka `defconfig` sadrži konfiguraciju jezgre izrađenu sučeljem `make defconfig`. Konfiguracija jezgre je namijenjena za i386 arhitekturu procesora. Ostale važnije varijable definirane u datoteci su:

- `LINUX_VERSION`: Varijablom se naznačuje verzija Linux jezgre i koristi se prilikom definiranja varijable `PV`.
- `PV`: Varijabla označuje verziju recepta.
- `LINUX_VERSION_EXTENSION`: Definira vrijednost `CONFIG_LOCALVERSION` varijable koja postavlja naziv jezgre vidljiv `uname` naredbom.
- `COMPATIBLE_MACHINE_diplomski-matej-qemux86`: Definira uređaj podržan receptom jezgre. U ovom slučaju vrijednost varijable je postavljena na uređaj definiran unutar istog BSP sloja.

Ispis 4.14: Sadržaj datoteke `linux-sluzbena-jezgra.bb`

```
inherit kernel
require recipes-kernel/linux/linux-yocto.inc

SRC_URI = "git:///home/matej/linux-stable;protocol=file;bareclone=1"
SRC_URI += "file://defconfig"

KBRANCH = "diplomski-linux-3.19"

LINUX_VERSION = "3.19"
LINUX_VERSION_EXTENSION = "-diplomski_verzija"

SRCREV="${AUTOREV}"

PR = "r1"
PV = "${LINUX_VERSION}+git${SRCPV}"

COMPATIBLE_MACHINE_diplomski-matej-qemux86 = "diplomski-matej-qemux86"
```

4.6.2. Konfiguracija uredaja

Datoteka konfiguracije uređaja je `/conf/machine/diplomski-matej-qemux86.conf`. Konfiguracija uređaja sadrži standardne postavke emulatora. Najvažnija linija konfiguracije kojom je definirana željena verzija Linux jezgre dana je u ispisu 4.15.

Ispis 4.15: Postavka verzije jezgre

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-sluzbena-jezgra"
```

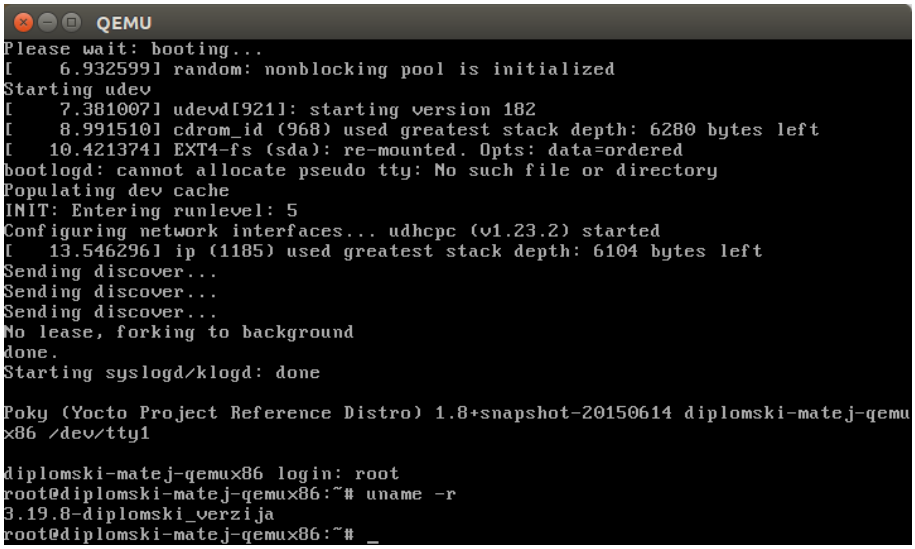
4.6.3. Izrada slike za virtualiziranu okolinu

Za potrebe kreiranja slike koja koristi izgrađeni BSP, unutar direktorija `poky` kreirat će se direktorij `build_sluzbena_jezgre`. Kreiranje direktorija i postavljanje okruženja potrebno je obaviti unutar direktorija `poky` izvođenjem naredbe prikazane u ispisu 4.16.

Ispis 4.16: Naredba za postavljanje okruženja

```
source oe-init-build-env build_sluzbena_jezgra
```

Nakon što je izvedena naredba iz ispisa 4.16, postavljeno je okruženje i za trenutni direktorij naredbenog retka postavljen je `build_sluzbena_jezgra`. Za uključivanje izrađenog sloja potrebno je urediti varijablu `BBLAYERS` unutar datoteke `conf/bblayers.conf`. U varijablu je potrebno dodati putanju do BSP sloja kao što je prikazano ispisom 4.17. U datoteci `conf/local.conf` potrebno



Slika 4.4: Izvođenje slike službene Linux jezgre

je varijablu pronaći varijablu `MACHINE` čija s vrijednošću `"qemux86"` i urediti istu kao što je prikazano u ispisu 4.18. Nakon što su uređene konfiguracijske datoteke, izradu slike sa službenom Linux jezgrom moguće je pokrenuti izvođenjem naredbe iz ispisa 4.6. Pokretanje slike službene Linux jezgre moguće je obaviti izvođenjem naredbe iz ispisa 4.19. Izvođenje službene Linux jezgre u virtualiziranoj okolini prikazano je na slici 4.4.

Ispis 4.17: Primjer postavljanja varijable `BBLAYERS`

```
BBLAYERS ?= " \
  /home/matej/poky/meta \
  /home/matej/poky/meta-yocto \
  /home/matej/poky/meta-yocto-bsp \
  /home/matej/poky/meta-diplomski-matej-qemux86 \
      "
```

Ispis 4.18: Postavka varijable `MACHINE`

```
MACHINE = "diplomski-matej-qemux86"
```

Ispis 4.19: Pokretanje slike službene Linux jezgre

```
runqemu
    tmp/deploy/images/diplomski-matej-qemux86/bzImage-diplomski-matej-qemux86.bin
tmp/deploy/images/diplomski-matej-qemux86/core-image-minimal-
diplomski-matej-qemux86.ext4
```

4.6.4. Izmjena službene Linux jezgre

Kako bi se demonstrirala lakoća izmjene repozitorija Linux jezgre, prikazana je vrlo jednostavna modifikacija jezgrinog koda. Izmjene su unesene za funkciju `filesystems_proc_show()` koja se nalazi unutar datoteke `fs/filesystems.c` repozitorija Linux jezgre. Izmijenjena funkcija prikazana je u ispisu 4.20. Funkcija `filesystems_proc_show()` dio je posebnog datotečnog sustava putem kojega jezgra pruža razne informacije. U funkciju je dodan poziv `printk()` funkcije koja bilježi znakovni niz u jezgrin dnevnik (engl. *kernel log*).

Ispis 4.20: Prikaz izmjenjene funkcije `filesystems_proc_show()`

```
static int filesystems_proc_show(struct seq_file *m, void *v)
{
    struct file_system_type * tmp;

    read_lock(&file_systems_lock);
    tmp = file_systems;
    while (tmp) {
        seq_printf(m, "%s\t%s\n",
            (tmp->fs_flags & FS_REQUIRES_DEV) ? "" : "nodev",
            tmp->name);
        tmp = tmp->next;
    }
    read_unlock(&file_systems_lock);

    printk("Matej was here!\n");
    return 0;
}
```

Provjeru obavljenih izmjena unutar repozitorija Linux jezgre moguće je obaviti izvođenjem naredbe prikazane u ispisu 4.21. Rezultat izvođenja naredbe iz ispisa 4.21 trebao bi biti sličan rezultatu danom u ispisu 4.22. Obavljene izmjene je potrebno potvrditi. Primjer naredbe za potvrdu unesenih izmjena dan je u ispisu 4.23. Ako se žele provjeriti potvrde obavljenih izmjena potrebno je izvesti naredbu `git log`. Primjer pregleda potvrda dan je u ispisu 4.24. Ako postoji slika koja je izgrađena prije nego što su promjene nad jezgrom obavljene, dovoljno je obaviti ažuriranje slike naredbama danim u ispisu 4.25. U suprotnom, potrebno je obaviti sve korake prikazane u prethodnom odjeljku. Izvođenje slike službene Linux jezgre s unesenim izmjenama prikazano je na slici 4.5.

Ispis 4.21: Naredba za provjeru unesenih izmjena

```
git diff -p HEAD
```

Ispis 4.22: Rezultat izvođenja `git diff` naredbe

```
diff --git a/fs/filesystems.c b/fs/filesystems.c
index 5797d45..9c4e8f7 100644
--- a/fs/filesystems.c
+++ b/fs/filesystems.c
@@ -233,6 +233,8 @@ static int filesystems_proc_show(struct seq_file *m, voi
    d *v)
        tmp = tmp->next;
    }
    read_unlock(&file_systems_lock);
+
+    printk("Matej was here!\n");
    return 0;
 }
```

Ispis 4.23: Primjer naredbe za potvrdu unesenih izmjena

```
git commit -a -m 'Dodana poruka koja ce biti zabiljezena prilikom poziva
    funkcije filesystems_proc_show'
```

Ispis 4.24: Primjer pregleda obavljenih potvrda

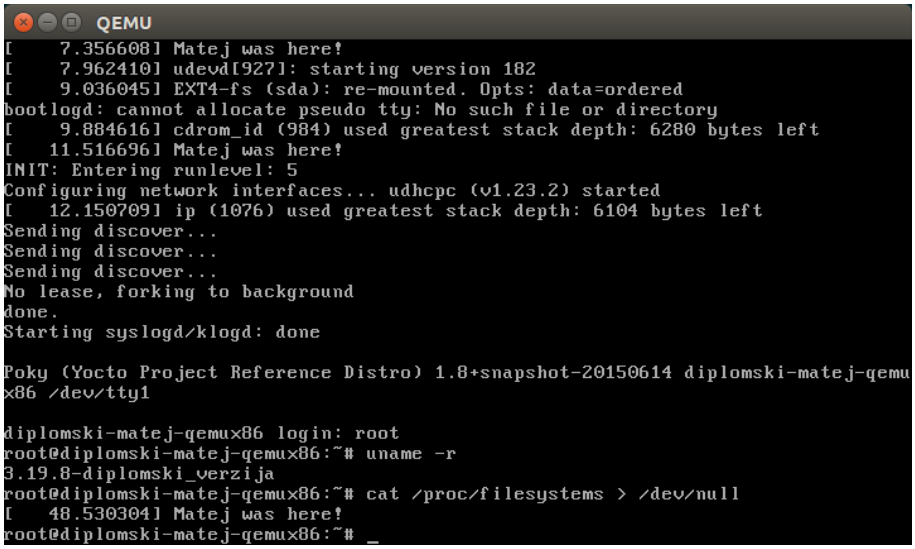
```
commit 7f4d72d0aef767c9731661720820800e232b450c
Author: Matej Filkovic <matej.filkovic@fer.hr>
Date:   Sun Jun 14 22:10:26 2015 +0200

    Dodana poruka koja ce biti zabiljezena prilikom poziva funkcije
        filesystems_proc_show

commit fcf4fe0e3e820408890ae137a684e56010c55f99
Author: Greg Kroah-Hartman <gregkh@linuxfoundation.org>
Date:   Mon May 11 05:34:10 2015 -0700

    Linux 3.19.8
```

Ispis 4.25: Naredbe za ažuriranje slike

```
bitbake -c cleanall virtual/kernel
bitbake -c deploy virtual/kernel
```

4.7. Izrada slike za Raspberry Pi 2

Izrada slike namijenjene za Raspberry Pi 2 zahtjeva odgovarajući BSP sloj. Za preuzimanje BSP sloja potrebno je unutar direktorija Poky referentnog sustava izvesti naredbu iz ispisa 4.26.



Slika 4.5: Izvođenje slike službene Linux jezgre s unesenim promjenama

Ispis 4.26: Naredba za preuzimanje BSP sloja

```
git clone git://git.yoctoproject.org/meta-raspberrypi
```

Nakon što je BSP sloj preuzet, potrebno je kreirati direktorij i postaviti okruženje za izradu. Kreiranje direktorija i postavljanje okruženja potrebno je obaviti izvođenjem naredbe iz ispisa 4.27.

Ispis 4.27: Naredba za postavljanje okruženja

```
source oe-init-build-env build_raspberry2
```

Nakon što je izvedena naredba iz ispisa 4.27, postavljeno je okruženje i za trenutni direktorij postavljen `build_raspberry2`. U datoteci `conf/local.conf` potrebno je pronaći varijablu `MACHINE` s postavljenom vrijednošću `"qemux86"` te ju urediti kao što je prikazano u ispisu 4.28. Za uključivanje preuzetog BSP sloja potrebno je urediti varijablu `BBLAYERS` unutar datoteke `conf/bblayers.conf`. U varijablu je potrebno dodati putanju do BSP sloja kao što je prikazano ispisom 4.29.

Ispis 4.28: Postavka varijable `MACHINE`

```
MACHINE = "raspberrypi2"
```

Za uključivanje razvojnih alata i alata za mjerenje performansi potrebno je u datoteci `conf/local.conf` pronaći varijablu `EXTRA_IMAGE_FEATURES` i postaviti ju kako što je prikazano ispisom 4.5. Nakon što su postavke konfiguracije pohranjene potrebno je pokrenuti izradu slike izvođenjem naredbe dane u ispisu 4.30.

Nakon što je slika izgrađena, potrebno ju je pohraniti na memorijsku karticu. Primjer naredbe za pohranu slike na memorijsku karticu dan je u ispisu 4.31.

Ispis 4.29: Primjer postavljanja varijable `BBLAYERS`

```
BBLAYERS ?= " \
    /home/matej/poky/meta \
    /home/matej/poky/meta-yocto \
    /home/matej/poky/meta-yocto-bsp \
    /home/matej/poky/meta-raspberrypi \
  "
```

Ispis 4.30: Naredba za pokretanje procesa izrade slike

```
bitbake rpi-basic-image
```

Ispis 4.31: Primjer naredbe za pohranu slike na memorijsku karticu

```
dd if=tmp/deploy/images/raspberrypi2/rpi-basic-image-raspberrypi2.rpi-sdimg
    of=/dev/sdb
```

4.7.1. Jezgrini moduli

Jezgrini moduli (engl. *kernel modules*) su dijelovi koda koji mogu biti dodani jezgri na zahtjev. Proširuju funkcionalnost jezgre bez potrebe za ponovnim podizanjem sustava. Jedan tip jezgrinih modula su upravljački programi koji omogućuju jezgri pristup sklopovlju.

Prevođenje jezgrinih modula se razlikuje od prevođenja standardnih korisničkih programa. Unutar `poky` direktorija nalazi se predložak za jezgrino modul. Točnije, direktorij predloška je `meta-skeleton/recipes-kernel/hello-mod/`. U dodatku D dan je izvorni kod jezgrinog modula namijenjenog za Raspbery Pi 2. Modul je nazvan `kprobe-modul` i za njega je kreiran istoimeni direktorij unutar `meta-raspberrypi/recepies-kernel`. U kreirani direktorij kopiran je sadržaj direktorija predloška koji je kasnije uređen. Uređeni sadržaj direktorija `kprobe-modul` prikazan je u ispisu 4.32. Izvorni kod modula pohranjen je u datoteci `files/kprobe_modul.c`. Ispisom 4.33 prikazan je recept jezgrinog modula. Datoteka `Makefile` kojom je definirano prevođenje jezgrinog modula dana je u ispisu 4.34.

Ispis 4.32: Sadržaj direktorija `kprobe-modul`

```
kprobe-modul/files
kprobe-modul/files/COPYING
kprobe-modul/files/Makefile
kprobe-modul/files/kprobe_modul.c
kprobe-modul/kprobe-modul_0.1.bb
```

Ispis 4.33: Recept za jezgrin modul

```
SUMMARY = "Modul koji postavlja kret_probe sondu i cita PMU brojace"

LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5=12f884d2ae1ff87c09e5b7ccc2c4ca7e"

inherit module

PR = "r0"
PV = "0.1"

SRC_URI = "file://Makefile \
           file://kprobe_modul.c \
           file://COPYING \
          "

S = "${WORKDIR}"
```

Ispis 4.34: Datoteka koja definira prevođenje jezgrinog modula

```
obj-m := kprobe_modul.o

SRC := $(shell pwd)

all:
	$(MAKE) -C $(KERNEL_SRC) M=$(SRC)

modules_install:
	$(MAKE) -C $(KERNEL_SRC) M=$(SRC) modules_install

clean:
	rm -f *.o *~ core .depend .*.cmd *.ko *.mod.c
	rm -f Module.markers Module.symvers modules.order
	rm -rf .tmp_versions Modules.symvers
```

Za prevođenje jezgrinog modula potrebno je prvo unutar direktorija Poky referentnog sustava izvesti naredbu iz ispisa 4.27. Zatim je potrebno izvesti naredbu iz ispisa 4.35.

Ispis 4.35: Naredba za izradu `kprobe-modul` jezgrinog modula

```
bitbake kprobe-modul
```

Putanja do izgrađenog jezgrinog modula dana je u ispisu 4.36. Modul je potrebno

postaviti u odabrani direktorij i izvesti naredbu iz ispisa 4.37. Izvođenjem naredbe 4.37 kreirat će se direktorij u kojemu je potrebno naći datoteku s ekstenzijom `.ko` i prenijeti na uređaj. Učitavanje jezgrinog modula obavlja se naredbom `insmod` kojoj je kao parametar potrebno predati datoteku s ekstenzijom `.ko`. Pregled jezgrinog dnevnika obavlja se naredbom `dmesg`.

Ispis 4.36: Putanja do izgrađenog modula

```
tmp/deploy/raspberrypi2/kernel-module-kprobe-modul-0.1-r0.raspberrypi2.rpm
```

Ispis 4.37: Naredba za raspakiranje jezgrinog modula

```
rpm2cpio kernel-module-kprobe-modul-0.1-r0.raspberrypi2.rpm | cpio -idmv
```

5. Mjerenje performansi na operacijskom sustavu Linux

5.1. Metode mjerenja performansi

Prema proučenim sučeljima i alatima za mjerenje performansi na operacijskom sustavu Linuxu, moguće je definirati sljedeće metode mjerenja performansi:

- Praćenje (engl. *tracing*): Metoda se zasniva na označavanju instrukcija procesora ili naredbi u izvornom kodu. Izvođenje označene instrukcije predstavlja događaj kojemu su pridruženi odgovarajući rukovatelji. Rukovatelji obično bilježe događaj i određene vrijednosti kao što je npr. stanje brojača jedinice za mjerene performansi u trenutku izvođenja označene instrukcije.
- Profiliranje (engl. *profiling*): Metoda se zasniva na sakupljanju uzoraka. Period sakupljanja uzoraka je obično izražen kao broj pojavljivanja događaja.

Ako se prate prva i zadnja instrukcija određene funkcije, moguće je mjeriti njezine performanse. Prije nego što se izvede prva instrukcija funkcije pročitaju se vrijednosti brojača. Brojači se zatim ponovno pročitaju nakon što se izvede zadnja instrukcija funkcije. Broj događaja unutar funkcije je razlika obavljenih čitanja. Na ovaj način praćenje se može iskoristi za mjerenje performansi funkcija jezgre operacijskog sustava.

5.2. Sučelja za mjerenje performansi

Tablica 5.1 prikazuje različita sučelja koja se mogu iskoristiti za mjerenje performansi na Linux operacijskom sustavu.

Linux jezgra održava razne statističke podatke na razini pojedinih procesa te

Tablica 5.1: Sučelja koja se mogu iskoristiti za mjerenje performansi

Namjena	Sučelje
Informacije za proces	`/proc`
Informacije na razini sustava	`/proc` i `/sys`
Informacije na razini upravljačkih programa	`/sys`
Praćenje na razini procesa	`uprobes`
Pristup PMU-u	`perf_events`
Praćenje na razini jezgre	`ftrace`, `Kprobes`, `tracepoints`

cijelog sustava. Neki od primjera su: broj promjena konteksta po sekundi i broj kreiranja procesa po sekundi. Zbog toga što je održavanje statističkih podataka uključeno u jezgri, njihovo korištenje ne predstavlja dodatno opterećenje koje bi usporilo rad sustava. Statistički podatci jezgre su dostupni putem posebnih datotečnih sustava `/proc` i `/sys`. Ova sučelja ne omogućuju fino mjerenje performansi te nisu daljnje razmatrana.

5.2.1. Sučelje perf_events

Sučelje `perf_events` omogućava bilježenje događaja. Događaji koji mogu biti bilježeni sučeljem kategorizirani su na sljedeće:

- Sklopovski događaj (engl. *hardware event*): Događaj praćen jedinicom za mjerenje performansi.
- Programski događaj (engl. *software event*): Događaj iz Linux jezgre (npr. zamjena konteksta, migracija procesa s jednog na drugi procesor).
- Događaj priručne memorije (engl. *hardware cache event*): Događaj praćen jedinicom za mjerenje performansi vezan za određenu razinu priručne memorije (npr. L1, TLB).
- Događaj prekidne točke (engl. *hardware breakpoint*): Događaj je pristup određenoj memorijskoj lokaciji.
- Događaj statičke točke praćenja (engl. *tracepoint event*): Događaj je doseg statičke točke praćenja.

Za ovaj rad važni su samo sklopovski događaji i događaji priručne memorije. Ostale kategorije događaja i njihovo praćenje sučeljem `perf_events` nije razmatrano. `perf_events` sučelje definira generalizirani skup sklopovskih događaja i događaja priručne memorije kako bi se isti mogli koristiti transparentno za različite arhitekture procesora. Korištenjem sučelja nije potrebno poznavati jedinicu

za mjerenje performansi niti njezine registre. Podsustav Linux jezgre obavlja programiranje registara jedinice za mjerenje performansi. Čitanje vrijednosti brojača omogućeno je korištenjem standardnih sustavskih poziva. Kako se dostupni događaji razlikuju među arhitekturama procesora, skup generaliziranih događaja je ograničen. Za neke procesore je moguće da svi generalizirani događaji nisu dostupni. Preporučuje se uvidom u izvorni kod sučelja vidjeti vrijednost događaja koja se programira u registar jedinice za mjerenje performansi. Ponekad naziv generaliziranog događaja može podosta odstupati od stvarnog događaja bilježenog jedinicom za mjerenje performansi.

Slično radu jedinice za mjerenje performansi, sučelje pruža dva načina rada:

- Brojanje: Svaka pojava događaja uzrokuje uvećanje vrijednosti brojača.
- Uzorkovanje: Period uzimanja uzoraka može biti definiran na dva načina:
  - broj pojava događaja
  - prosječna stopa uzoraka po sekundi

Korištenje sučelja za sakupljanje uzoraka nije razmatrano radom. Razmatra se samo brojanje kako bi se korištenjem jedinice za mjerenje performansi odredile performanse pojedine funkcije ili određenog dijela izvornog koda.

Korištenje sučelja

Korištenje `perf_events` sučelja zasniva se na `perf_event_open()` sustavskom pozivu. `perf_event_open()` sustavski poziv nema implementiranu omotnu rutinu u standardnoj programskoj knjižici. Preporučuje se napisati omotnu rutinu kako bi se olakšalo korištenje. Primjer omotne rutine za `perf_event_open()` sustavski poziv dan je u ispisu 5.1. Povratna vrijednost sustavskog poziva `perf_event_open()` je opisnik brojača.

Ispis 5.1: Prikaz omotne rutine za `perf_event_open()` sustavski poziv

```
#include <linux/perf_event.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <asm/unistd.h>

int perf_event_open(struct perf_event_attr *attr, pid_t pid, int cpu,
                    int group_fd, unsigned long flags)
{
    return syscall(__NR_perf_event_open, attr, pid, cpu, group_fd, flags);
}
```

Argument `attr` definira događaj koji se želi bilježiti sučeljem. Neki od važnijih članova strukture `struct perf_event_attr` su:

- `type`: Atributom je specificiran tip događaja. Npr. ako se želi bilježiti jedan od sklopovskih događaja, vrijednost atributa je potrebno postaviti na `PERF_TYPE_HARDWARE`.
- `config`: Atributom je specificiran događaj koji se želi bilježiti. Npr. ako je atribut `type` postavljen na `PERF_TYPE_HARDWARE`, atribut `config` može biti postavljen na `PERF_COUNT_HW_CPU_CYCLES`.
- `disabled`: Ako je atribut postavljen, brojač odabranog događaja nije uključen. Brojač je moguće uključiti pozivom `ioctl()` funkcije.
- `exclude_user`: Ako je atribut postavljen, događaji iz korisničkog prostora neće biti bilježeni.
- `exclude_kernel`: Ako je atribut postavljen, događaji iz jezgrinog prostora neće biti bilježeni.

Argumentima `pid` i `cpu` određeni su process i procesor za koje se želi bilježiti događaj definiran argumentom `attr`. Moguće su sljedeće vrijednosti argumenata:

- `pid == 0`, `cpu == -1`: Bilježi za pozivajući proces ili dretvu na bilo kojem procesoru.
- `pid == 0`, `cpu >= 0`: Bilježi za pozivajući proces samo prilikom izvođenja na definiranom procesoru.
- `pid > 0` i `cpu == -1`: Bilježi za definirani proces na bilo kojem procesoru.
- `pid > 0` i `cpu >= 0`: Bilježi za definirani proces samo prilikom izvođenja na definiranom procesoru.

Argument `group_fd` omogućuje kreiranje grupe događaja koja će se bilježiti odjednom. Grupa događaja ima voditelja. Voditelja je potrebno kreirati prvog tako da se argument `group_fd` postavi na `-1`. Ostale članove grupe potrebno je kreirati postavljanjem opisnika voditelja za argument `group_fd`.

Brojači se zaustavljaju i pokreću korištenjem funkcije `ioctl()`. Nakon što je brojač zaustavljen, on više ne bilježi događaje ali zadržava vrijednost. Čitanje brojača obavlja se `read()` sustavskim pozivom.

Dodatak A sadrži program koji koristi `perf_events` sučelje. Dio izvornog koda kojim je definiran događaj koji se želi pratiti dan je u ispisu 5.2. Događaj definiran u ispisu 5.2 je izvođenje instrukcije. Brojač izvedenih instrukcija nije

uključen i definiran je da broji samo instrukcije izvedene u korisničkom načinu rada.

Ispis 5.2: Primjer definiranja događaja `perf_events` sučeljem

```
memset(&pe, 0, sizeof(struct perf_event_attr));
pe.type = PERF_TYPE_HARDWARE;
pe.size = sizeof(struct perf_event_attr);
pe.config = PERF_COUNT_HW_INSTRUCTIONS;
pe.disabled = 1;
pe.exclude_kernel = 1;
pe.exclude_hv = 1;
```

Dio izvornog koda koji upravlja brojačem definiranog događaja dan je u ispisu 5.3. Brojači se resetiraju, uključuju i zaustavljaju pozivom `ioctl()` funkcije. Ako je definirana grupa događaja, umjesto `0` potrebno je proslijediti vrijednost `PERF_IOC_FLAG_GROUP`.

Ispis 5.3: Primjer rada s brojačima `perf_events` sučelja

```
ioctl(fd, PERF_EVENT_IOC_RESET, 0);
ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);

zbroji_matrice(matrica_a, matrica_b, matrica_rez, red);

ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);
read(fd, &count, sizeof(long long));
```

Točnost `perf_events` sučelja

Srednja vrijednost je broj kojim se predstavljaju rezultati više uzastopnih mjerenja. Računanje srednje vrijednosti provodi se po formuli 5.1, gdje je x mjerena veličina, a N broj mjerenja. Standardna devijacija označava koliko u prosjeku pojedina mjerenja odstupaju od srednje vrijednosti i računa se prema formuli 5.2.

$$\bar{x} = \frac{\sum_{i=1}^{N} x_i}{N} \tag{5.1}$$

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2} \tag{5.2}$$

Za provjeru točnosti `pref_events` sučelja i jedince za mjerenje performansi korišten je program dan u dodatku A. Test je proveden za dvije različite konfiguracije prikazane u tablici 5.2. Konfiguracija s Broadcom procesorom je Raspberry Pi 2 uređaj, za koji je izrađena slika Linux operacijskog sustava u poglavlju 4.

Uz pretpostavku da je program iz priloga A pohranjen u datoteku `test_instr`

`ukcije.c`, prevođenje programa obavlja se naredbom iz ispisa 5.4. Jedini događaj unutar procesora koji se može pobrojati su izvedene instrukcije. Program kao argument prima red kvadratne matrice i u funkciji `zbroji_matrice()` obavlja zbrajanje dvodimenzionalnih polja koja su dinamički alocirana. Korištenjem sučelja `perf_events` broje se izvedene instrukcije unutar funkcije `zbroji_matrice()`. Za generiranje instrukcija procesora iz izvršnog programa, potrebno je izvesti naredbu danu u ispisu 5.5. Instrukcije funkcije `zbroji_matrice()` za testne konfiguracije iz tablice 5.2 dane su u dodatku B.

Tablice 5.4 i 5.3 prikazuju odnos izmjerenog broja instrukcija i izračunatog broja instrukcija u funkciji `zbroji_matrice_()`. Mjerenja su ponovljena dvadeset puta. Važno je napomenuti da izračunati broj instrukcija funkcije ne uključuje instrukcije za prijenos parametara i zaustavljanje brojača. Iz priloženih mjerenja vidljivo je da za obje konfiguracije postoji stalna razlika između broja instrukcija u funkciji `zbroji_matrice()` i izmjerenog broja instrukcija. Razlika iznosi osamnaest i za veći broj instrukcija je zanemariva. Za većinu mjerenja standardna devijacija je nula i manja odstupanja se pojavlju tek kod većeg broja instrukcija. Moguće je zaključiti da su jedinice za mjerenje performansi oba procesora prilikom brojanja instrukcija vrlo točne.

Tablica 5.2: Prikaz konfiguracija korištenih prilikom testiranja

Procesor	Distribucija	Verzija Linux jezgre
Intel i5-2467M	Ubuntu 14.04 (LTS)	3.13.0-55-generic
Broadcom BCM2836	Poky referentna distribucija	3.18.11

Ispis 5.4: Naredba za prevođenje testnog programa

```
gcc test_instrukcije.c -o test_instrukcije -std=gnu99
```

Ispis 5.5: Naredba za generiranje instrukcija procesora

```
objdump -d test_instrukcije
```

5.2.2. Statičke točke praćenja

Statičke točke praćenja (engl. *static tracepoints*) su funkcije koje bilježe informacije na specifičnim mjestima u Linux jezgri. Kako bi funkcije bilježile informacije

Tablica 5.3: Mjerenja broja instrukcija za konfiguraciju s Broadcom procesorom

Red kvadratne matrice	Broj instrukcija unutar funkcije	Izmjereni broj instrukcija	
		$\bar{x}$	σ
1	66	84	0
2	185	203	0
5	962	980	0
10	3657	3675	0
20	14297	14315	0
50	88217	88235	0
100	351417	351435	0
200	1402817	1402835.4	0.489
500	8757017	8757037	0.730
1000	35014017	35014041.733	1.842
2000	140028017	140028061.333	4.526
5000	875070017	875070624.866	9.728

potrebno ih je omogućiti određenim alatom. Programerima je omogućeno definiranje vlastitih statičkih točaka u izvornom kodu Linux jezgre.

Jedna od statičkih točaka praćenja je `sched_switch`. U ispisu 5.6 dan je dio funkcije `context_switch()` koja se nalazi u datoteci `kernel/scehd/core.c` službenog repozitorija Linux jezgre. Funkcija `context_switch()` poziva funkciju `prepare_task_switch()` čiji je dio dan u ispisu 5.7. Funkcija `prepare_task_switch()` zatim poziva funkciju `trace_sched_switch()` koja bilježi informacije ako je točka `sched_switch` aktivirana. Sadržaji ispisa 5.6 i 5.7 kopirani su iz grane službene Linux jezgre kreirane u odjeljku 4.6.

Ispisom 5.8 dan je pregled informacija koje `sched_switch` točka bilježi. Rezultat je dobiven korištenjem alata LTTng opisanog u narednom dijelu. Iz ispisa je vidljiv naziv i identifikator procesa čije je izvođenje zaustavljeno. Također, vidljiv je naziv i identifikator procesa kojemu je procesor dan na korištenje.

Tablica 5.4: Mjerenja broja instrukcija za konfiguraciju s Intel procesorom

Red kvadratne matrice	Broj instrukcija unutar funkcije	Izmjereni broj instrukcija	
		$\bar{x}$	σ
1	60	78	0
2	183	201	0
5	1008	1026	0
10	3903	3921	0
20	15393	15411	0
50	95463	95481	0
100	380913	380931	0
200	1521813	1521831	0
500	9504513	9504532.8	0.909
1000	38009013	38009034.466	1.3097
2000	152018013	152018038.8	1.796
5000	950045013	950045513.062	7.275

Ispis 5.6: Prikaz dijela funkcije `context_switch()`

```
/*
/*
 * context_switch - switch to the new MM and the new thread's register state.
 */
static inline struct rq *
context_switch(struct rq *rq, struct task_struct *prev,
        struct task_struct *next)
{
  struct mm_struct *mm, *oldmm;

  prepare_task_switch(rq, prev, next);

  mm = next->mm;
  oldmm = prev->active_mm;
```

Ispis 5.7: Prikaz dijela funkcije `prepare_task_switch()`

```
static inline void
prepare_task_switch(struct rq *rq, struct task_struct *prev,
       struct task_struct *next)
{
  trace_sched_switch(prev, next);
  sched_info_switch(rq, prev, next);
  perf_event_task_sched_out(prev, next);
```

Ispis 5.8: Rezultat praćenja statičke točke `sched_switch` alatom LTTng

```
[13:50:38.943646736] (+0.000104554) matej-UX21E sched_switch: { cpu_id = 2 },
    { prev_comm = "test_trosak", prev_tid = 5711, prev_prio = 20, prev_state
    = 1, next_comm = "kworker/2:1", next_tid = 17 8, next_prio = 20 }
```

5.2.3. Sučelje Kprobes

Kprobes (*Kernel Dynamic Probes*) je sučelje koje omogućuje dinamičko definiranje točaka praćenja u Linux jezgri. Zasniva se na konceptu jezgrinih sondi (engl. *kernel probe*). Jezgrina sonda je skup rukovatelja (engl. *handlers*) postavljenih na adresu određene instrukcije koja se nalazi u jezgrinom adresnom prostoru. Trenutno postoje tri vrste jezgrinih sondi:

- `kprobe`: Mogu biti postavljene na bilo koju instrukciju jezgre.
- `jprobe`: Postavljaju se na početnu instrukciju jezgrine funkcije i omogućavaju jednostavan pristup argumentima funkcije.
- `kretprobe`: Postavljaju se na zadnju instrukciju funkcije.

Podatkovne strukture i funkcije koje omogućavaju pristup Kprobes sučelju definirane su u datoteci zaglavlja `<linux/kprobes.h>`. Zbog toga što se sonde postavljaju na instrukcije u jezgrinom adresnom prostoru, njihovo postavljanje potrebno je obaviti korištenjem jezgrinih modula.

Nakon registracije `krpobe` sonde, Kprobes sučelje kopira instrukciju na koju je sonda postavljena i zamjenjuje prvih nekoliko bajtova instrukcijom prekida (engl. *breakpoint instruction*) (npr. `int3` na i386 i x86_64 arhitekturi procesora) [5]. Kada izvođenje dosegne instrukciju prekida, registri procesora su spremljeni i kontrola izvođenja je prepuštena Kprobes sučelju. Sučelje tada poziva pred rukovatelja (engl. *pre-handler*) povezanog s `kprobe` sondom te mu kao parametre predaje adresu sonde i pohranjene registre. Nakon što pred rukovatelj obavi definiranu aktivnost, Kprobes sučelje izvodi kopiju instrukcije u koračnom načinu (engl. *single-stepped*). Kada je kopija instrukcije izvedena, poziva se naknadni rukovatelj (engl. *post-handler*) ako je definiran. Izvođenje tada nastavlja s instrukcijom koja se nalazi nakon one na koju je postavljena sonda. Moguće je definirati i rukovatelja za obradu pogrešaka (engl. *fault-handler*) koji će se pozvati ako se dogodi pogreška u jednom od navedenih rukovatelja.

`jprobe` sonda implementirana je korištenjem `kprobe` sonde koja je postavljena na početnu instrukciju funkcije. Rukovatelj `jprobe` sonde mora imati iste

argumente i povratnu vrijednost kao funkcija čijim se argumentima želi pristupiti i uvijek mora završiti pozivom funkcije `jprobe_return()`.

Prilikom registracije `kretprbe` sonde postavlja se `kprobe` sonda na ulazu u funkciju. Kada se pozove funkcija na koju je postavljena sonda, Kprobes sučelje sprema povratnu adresu i zamjenjuje ju adresom proizvoljne instrukcije (obično `nop` instrukcija) na koju je postavljena `kprobe` sonda. Korištenjem `kretprobe` sonde moguće je definirati rukovatelja koji će se pozvati prilikom ulaska u jezgrinu funkciju, te rukovatelja koji će se pozvati prilikom izlaska iz jezgrine funkcije.

Primjer korištenja Kprobes sučelja

Dodatak D sadrži jezgrin modul koji koristi `kretprobe` sondu. Dio izvornog koda kojim je definirana `kretprobe` sonda i njezini rukovatelji dan je u ispisu 5.9. Postavljanjem atributa `.maxactive` na jedan, definirano je da samo jedna sonda može biti aktivna. Linijom `kret_sonda.kp.symbol_name = "udp_v4_get_port";` rezolucija adrese funkcije prepuštena je jezgri. Definirana sonda registrana je pozivom funkcije `register_kretprobe()`. Nakon što je modul učitan, ulaskom u funkciju `udp_v4_get_port()` pozvat će se ulazni rukovatelji, dok će se povratni rukovatelj pozvati prilikom izlaska iz funkcije.

Ispis 5.9: Prikaz postavljanja rukovatelja `kretprobe` sonde

```
static struct kretprobe kret_sonda = {
    .handler = povratni_rukovatelj,
    .entry_handler = ulazni_rukovatelj,
    .maxactive = 1,
};
```

5.3. Alati za mjerenje performansi

U nastavku je dan pregled alata potrebnih za razvijenu metodologiju.

5.3.1. Alat ftrace

Alat `ftrace` je napravljen kako bi omogućio programerima i dizajnerima sustava odgonetnuti što se događa unutar Linux jezgre. Može biti korišten prilikom otklanjanja pogrešaka i analize latencije u Linux jezgri. Obično se smatra samo pratiteljem funkcija pozvanih prilikom izvođenja u jezgrinom načinu rada, međutim sadrži i dodatne funkcionalnosti kao što je npr. aktiviranje statičkih točaka praćenja.

`ftrace` koristi `debugfs` posebni datotečni sustav kako bi omogućio pristup kontrolnim datotekama i datotekama za prikaz rezultata. Datoteke alata nalaze se unutar direktorija `/sys/kernel/debug/tracing` od kojih su sljedeće važne za ovaj rad:

- `current_tracer`: Koristi se za odabir načina praćenja. Za ovaj rad jedini važan način praćenja je `function_graph` koji prati funkcije na njihovom ulazu i povratu. Funkcijski pozivi prikazani su na način sličan izvornom kodu programskog jezika C. Primjer `function_graph` načina praćenja dan je u ispisu 5.10.
- `tracing_on`: Koristi se za isključivanje i uključivanje praćenja.
- `trace`: Sadrži rezultat praćenja u čitljivom formatu.
- `set_ftrace_pid`: Omogućuje praćenje samo jednog procesa.
- `max_graph_depth`: Koristi se s `function_graph` načinom praćenja. Datotekom je moguće definirati dubinu do koje će se pozvane funkcije prikazivati. Zadanim postavkama dubina nije ograničena.

Ispis 5.10: Primjer praćenja `function_graph` načinom

```
1)               |  SyS_sendto() {
1)               |    sockfd_lookup_light() {
1)   1.122 us    |      fget_light();
1)   2.524 us    |    }
1)   0.435 us    |    move_addr_to_kernel.part.14();
1)               |    sock_sendmsg() {
1)               |      security_socket_sendmsg() {
1)               |        apparmor_socket_sendmsg() {
1)   0.526 us    |          aa_revalidate_sk();
1)   1.998 us    |        }
1)   3.756 us    |      }
```

Razlog zbog kojega alat `ftrace` uvršten u ovaj rad je ponajviše `function_graph` način praćenja kojime je moguće vrlo brzo odrediti koje se jezgrine funkcije pozivaju za sustavski poziv. Također, postavljanjem dubine poziva na odgovarajuću vrijednost moguće je učiniti rezultat preglednijim. Iako su aproksimacije, iz prikazanih latencija moguće je relativno brzo odrediti koje funkcije najviše utječu na performanse. Za odabrane funkcije potrebno je provesti fina mjerenja korištenjem jedinice za mjerenje performansi.

Rad s alatom

Ako `debugfs` datotečni sustav nije postavljen po zadanim postavkama, potrebno ga je postaviti izvođenjem naredbe prikazane u ispisu 5.11. Nakon što je postavljen `debugfs` moguće je pronaći direktorij `/sys/kernel/debug/tracing` koji sadrži sve datoteke alata `ftrace`.

Ispis 5.11: Prikaz naredbe za postavljanje `debugfs` datotečnog sustava

```
mount -t debugfs nodev /sys/kernel/debug
```

Rad s alatom `ftrace` zasniva se na izvođenju naredbi ispisa 5.12 unutar direktorija `/sys/kernel/debug/tracing`. Parametar `dubina` određuje razinu do koje će se pratiti pozvane funkcije. Parametar `pid` je identifikator procesa za koji se prate pozvane jezgrine funkcije.

Ispis 5.12: Korištenje alata `ftrace` za praćenje jezgrinih funkcija

```
echo dubina > max_grap_depth
echo pid > set_ftrace_pid
echo function_graph > current_tracer
echo 1 > tracing_on
```

Rezultati praćenja pregledavaju se izvođenjem naredbe iz ispisa 5.13. Primjer rezultata dan je u ispisu 5.10.

Ispis 5.13: Naredba za pregled rezultata praćenja jezgrinih funkcija

```
cat trace
```

5.3.2. Alat LTTng

LTTng *Linux Trace Toolkit: next generation* je alat kojim je moguće provesti praćenje u jezgrinom i korisničkom prostoru. Omogućava pristup statičkim točkama praćenja i jedinici za mjerenje performansi. Pristup jedinici za mjerenje performansi obavlja korištenjem `perf_events` sučelja. Prilikom praćenja u jezgrinom prostoru, brojači jedinice za mjerenje performansi čitaju se na razini procesora. Omogućava i dinamičko praćenje jezgrenih funkcija korištenjem `kprobe` i `kretprobe` sondi.

Koncepti alata LTTng su:

- Sjednica praćenja (engl. *tracing session*)
- Domena praćenja (engl. *domain*)
- Kanal (engl. *channel*)

– Događaj (engl. *event*)

Sjednica praćenja je spremnik stanja. Sva praćenja korištenjem alata LTTng moraju biti obavljena unutar sjednice. Informacije pojedinih sjednica praćenja su u potpunosti izolirane od drugih. Neki od važnijih atributa i objekata koje sjednica sadrži su:

– Ime

– Stanje praćenja (pokrenuto ili zaustavljeno)

– Jednu ili više domena

– Za svaku domenu listu kanala

– Za svaki kanal:

  • Ime kanala

  • Stanja kanala (uključen ili isključen)

  • Listu događaja

  • Listu kontekstnih informacija (npr. identifikator procesa, vrijednosti brojača jedinice za mjerenje performansi)

Domena određuje kategoriju praćenja. LTTng omogućuje više domena praćenja od kojih su dvije: Linux jezgra i korisnički prostor. Kanal je skup događaja sa specificiranim parametrima i informacijama vezanim uz kontekst. Kanal uvijek ima jedinstveno ime po domeni unutar sjednice praćanje. Kao događaj može se definirati poziv funkcije, izvođenje instrukcije ili statička točka praćenja. Pojava događaja popraćena je definiranim kontekstnim informacijama. Promatrani događaj je uvijek registriran na barem jedan kanal.

Rad s alatom

Za ovaj rad alat LTTng se koristi isključivo za mjerenje performansi jezgrinih funkcija i praćenje točke `sched_switch`. Praćenje se odvija u domeni jezgre i koristi se samo jedan kanal. Rad s alatom se svodi na ispis 5.14. Naredbom `lttng create` kreirana je sjednica čiji je naziv `diplomski_sjednica`. Naredbom `lttng enable-event` dodan je događaj čiji je naziv `SyS_sendto`. Događaj je u domeni jezgre i definiran je kao ulazak i izlazak iz funkcije `SyS_sendto()`. Preporučuje se događaj uvijek nazvati imenom funkcije koja se prati. Kod praćenja drugih funkcija potrebno je zamijeniti naziv događaja i naziv funkcije. Naredbom `lttng add-context` dodane su kontekstne informacije koje će se prikupiti

prilikom ulaska i izlaska iz funkcije `SyS_sendto()`. Dodane kontekstne informacije su identifikator procesa za koji se izvodi jezgrina funkcija i broj izvedenih instrukcija. Za dodavanje dodatnih kontekstnih informacija potrebno je samo u istoj liniji predati opciju `-t` i naziv kontekstne informacije. Pregled dostupnih kontekstnih informacija moguće je obaviti izvođenjem naredbe prikazane u ispisu 5.19. U domeni jezgre korištenje sučelja `perf_events` omogućeno je samo na razini procesora. Kontekstne informacije vezane uz sučelje `pref_events` označene su s `perf:cpu` prefiksom. Neke od kontekstnih informacija vezane uz sučelje `perf_events` dane su u ispisu 5.16. Naredba `lttng start` pokreće praćenje. Kada se praćenje želi zaustaviti potrebno je izvesti naredbu iz ispisa 5.15. Rezultati praćenja pregledavaju se izvođenjem naredbe iz ispisa 5.17. Primjer rezultata praćenja prikupljenih sjednicom definiranom u ispisu 5.14 dan je u ispisu 5.20. Iz ispisa je vidljivo da su kontekstne informacije prikupljene prilikom ulaska i izlaska iz funkcije `SyS_sendto()`. Broj izvedenih instrukcija je razlika vrijednosti očitane prilikom izlaska iz funkcije i vrijednosti očitane prilikom ulaska u funkciju.

Zbog toga što izvođenje jezgrinih funkcija za neki proces može biti zaustavljeno, potrebno je pratiti i točku `sched_switch` tako da se u ispis 5.14 doda naredba iz ispisa 5.21. Naredbu je potrebno dodati prije `lttng start` naredbe. Primjer naredbe kojom je moguć pregled izvođenja zadane funkcije i zamjene konteksta za promatrani proces dan je u ispisu 5.22. Za neku drugu promatranu funkciju potrebno je naziv događaja `SyS_sendto` zamijeniti definiranim. Također, potrebno je postaviti i vrijednost identifikatora procesa. Primjer neispravnog mjerenja performansi jezgrine funkcije dan je u ispisu 5.23. Iz ispisa je vidljivo da je izvođenje jezgrine funkcije `SyS_sendto()` zaustavljeno za proces čija je vrijednost identifikatora `1441`. Izvođenje jezgrine funkcije za proces je zatim nastavljeno i zabilježen je izlazak iz funkcije `SyS_sendto()`. Ovakva mjerenja potrebno je odbaciti jer se brojači jedinice za mjerenje performanse čitaju na razini procesora. Primjer ispravnog mjerenja kod kojega funkcija nije zaustavljena tijekom izvođenja dan je u ispisu 5.24. Nakon što rezultati praćenja nisu više potrebni, moguće ih je ukloniti izvođenjem naredbe iz ispisa 5.25.

Ispis 5.14: Naredbe za rad s alatom LTTng

```
1 lttng create diplomski_sjednica
2 lttng enable-event SyS_sendto -k --function SyS_sendto
3 lttng add-context -k -t pid -t perf:instructions
4 lttng start
```

Ispis 5.15: Naredba za zaustavljanje praćenja

```
lttng stop
```

Ispis 5.16: Kontekstne informacije vezane uz `perf_events` sučelje

```
perf:cpu:cpu-cycles, perf:cpu:cycles,
perf:cpu:stalled-cycles-frontend,
perf:cpu:idle-cycles-frontend,
perf:cpu:stalled-cycles-backend,
perf:cpu:idle-cycles-backend,
perf:cpu:instructions, perf:cpu:cache-references,
perf:cpu:cache-misses,
perf:cpu:branch-instructions, perf:cpu:branches,
```

Ispis 5.17: Naredba za pregled rezultata praćenja

```
lttng view
```

Ispis 5.18: Naredba za praćenje točke `sched_switch`

```
lttng enable-event sched_switch -k
```

Ispis 5.19: Naredba za pregled dostupnih kontekstnih informacija

```
lttng add-context --help
```

Ispis 5.20: Primjer rezultata praćenja

```
[09:10:16.709620680] (+0.000364309) matej-UX21E SyS_sendto_entry: { cpu_id =
    2 }, { pid = 2934, perf_instructions = 5209761285 }, { ip =
    0xFFFFFFFF8160EFB0, parent_ip = 0xFFFFFFFF81733D5D }
[09:10:16.709633895] (+0.000013215) matej-UX21E SyS_sendto_return: { cpu_id =
    2 }, { pid = 2934, perf_instructions = 5209765439 }, { ip =
    0xFFFFFFFF8160EFB0, parent_ip = 0xFFFFFFFF81733D5D }
```

Ispis 5.21: Naredba za praćenje točke `sched_switch`

```
lttng enable-event sched_switch -k
```

Ispis 5.22: Naredba za pregled praćene funkcije i zamjene konteksta

```
lttng view | grep -E "SyS_sendto|tid = 1441"
```

Ispis 5.23: Primjer neispravnog mjerenja performansi jezgrine funkcije

```
[04:02:24.570807135] (+0.001119583) raspberrypi2 SyS_sendto_entry: { cpu_id =
    1 }, { pid = 1441, perf_cpu_instructions = 3747564400, perf_cpu_cycles =
    6286813580, perf_cpu_cache_references = 2027260178, perf_cpu_cache_misses
    = 463629 }, { ip = 0x804479A4, parent_ip = 0x8000ED40 }
[04:02:24.571131250] (+0.000324115) raspberrypi2 sched_switch: { cpu_id = 1
    }, { pid = 1441, perf_cpu_instructions = 3747658912, perf_cpu_cycles =
    6287008078, perf_cpu_cache_references = 2027307321, perf_cpu_cache_misses
    = 463875 }, { prev_comm = "server_udp", prev_tid = 1441, prev_prio = 20,
    prev_state = 1024, next_comm = "rcu_preempt", next_tid = 7, next_prio =
    20 }
[04:02:24.571203281] (+0.000072031) raspberrypi2 sched_switch: { cpu_id = 1
    }, { pid = 7, perf_cpu_instructions = 3747681230, perf_cpu_cycles =
    6287050452, perf_cpu_cache_references = 2027319583, perf_cpu_cache_misses
    = 463930 }, { prev_comm = "rcu_preempt", prev_tid = 7, prev_prio = 20,
    prev_state = 1, next_comm = "server_udp", next_tid = 1441, next_prio = 20
    }
[04:02:24.571446771] (+0.000243490) raspberrypi2 SyS_sendto_return: { cpu_id
    = 1 }, { pid = 1441, perf_cpu_instructions = 3747749397, perf_cpu_cycles
    = 6287221447, perf_cpu_cache_references = 2027355556,
    perf_cpu_cache_misses = 464213 }, { ip = 0x804479A4, parent_ip =
    0x8000ED40 }
```

Ispis 5.24: Primjer ispravnog mjerenja performansi jezgrine funkcije

```
[04:02:30.679495727] (+0.001103906) raspberrypi2 SyS_sendto_entry: { cpu_id =
    1 }, { pid = 1441, perf_cpu_instructions = 3945573565, perf_cpu_cycles =
    6623873533, perf_cpu_cache_references = 2134258852, perf_cpu_cache_misses
    = 487736 }, { ip = 0x804479A4, parent_ip = 0x8000ED40 }
[04:02:30.679852394] (+0.000356667) raspberrypi2 SyS_sendto_return: { cpu_id
    = 1 }, { pid = 1441, perf_cpu_instructions = 3945682250, perf_cpu_cycles
    = 6624111737, perf_cpu_cache_references = 2134314073,
    perf_cpu_cache_misses = 488085 }, { ip = 0x804479A4, parent_ip =
    0x8000ED40 }
```

Ispis 5.25: Naredba za uklanjanje rezultata praćenja

```
lttng destroy
```

Točnost alata LTTng

Za testiranje točnosti alata LTTng korišten je jezgrin modul iz dodatka D. Točnost je testirana za broj instrukcija funkcije `udp_v4_get_port()`. Tablica 5.5 prikazuje rezultate za dvadeset ponovljenih mjerenja.

Iz prikazanih rezultata vidljivo je da alat LTTng unosi određeno odstupanje i rezultati su manje stabilni nego oni od jezgrinog modula. Međutim, za priloženi modul bilo je potrebno proučiti registre jedinice za mjerenje performansi. Osim toga, priloženim modulom nije moguće utvrditi zaustavljanje izvođenja jezgrine

Tablica 5.5: Mjerenja broja instrukcija za funkciju `udp_v4_get_port()`

	Broj instrukcija	
	$\bar{x}$	σ
LTTng	1674.75	16.896
Jezgrin modul	894	0

funkcije. Ako je jezgra konfigurirana tako da se njezine funkcije ne mogu prekinuti tijekom izvođenja, priloženi modul predstavlja jednostavnu zamjenu za alat.

6. Metodologija finog mjerenja performansi

6.1. Opis metodologije

Fino mjerenje performansi moguće je obaviti tako da se za dijelove izvornog koda broje odabrani događaji prisutni u procesoru. Potpora brojanju događaja prisutnih u procesoru ostvarena je jedinicom za mjerenje performansi. Dijelovi izvornog koda s kojima treba započeti mjerenja su funkcije. Ako se ustanovi da funkcija unosi neprihvatljivo kašnjenje potrebno je mjerenja ponoviti nad njezinim manjim dijelovima. Nakon što se ustanove dijelovi koda koji unose najviše kašnjenja potrebno je pronaći razlog i ispitati optimizaciju.

Promatranjem proteklih ciklusa vrlo brzo je moguće utvrditi koji dijelovi koda najviše utječu na izvođenje. Ako se uz cikluse broje i izvedene instrukcije moguće je promatrati i mjeru CPI. Za dijelove izvornog koda s većom vrijednošću mjere CPI postoji mogućnost da gube vrijeme čekajući na dohvat podataka iz memorije. Brojanjem događaja povezanih uz priručnu memoriju moguće je utvrditi kojim razinama memorije se najviše pristupa. Ako se ustanovi da dijelovi izvornog koda ne iskorištavaju prednosti priručne memorije potrebno je ispitati mogućnost optimizacije.

6.2. Metodologija na operacijskom sustavu Linux

Sučelje `perf_events` se može koristi za mjerenje performansi korisničkih programa. Primjer je program dan u dodatku A. Za mjerenje performansi jezgrinih funkcija potrebno je koristiti alat LTTng na način prikazan u odjeljku 5.3.2. Iako unosi određena odstupanja, odgovarajuća zamjena nije pronađena. Jezgrin modul

iz dodatka D je primjer moguće zamjene, ali njegova funkcionalnost je minimalna.

Za određivanje pozvanih jezgrinih funkcija potrebno je koristiti alat `ftrace`. Primjer rada s alatom dan u ispisu 5.12 prati sve pozvane jezgrine funkcije do definirane dubine. Kako bi se omogućilo praćenja jezgrinih funkcija koje su rezultat poziva određene funkcije iz korisničkog programa, priložene su pomoćne funkcije u dodatku E. Funkciju `ocisti_zapise()` potrebno je pozvati prije početka praćenja kako bi se uklonili stari zapisi. Preostale dvije funkcije moguće je koristi kao što je prikazno u ispisu 6.1. Funkcija `zapocni_pracenje_jezgrinih_funckija()` za argument prima dubinu do koje se pozvane jezgrine funkcije žele pratiti. Pregled rezultata obavlja se naredbom iz ispisa 5.13. Primjer rezultata dobivenih korištenjem pomoćnih funkcija dan je u ispisu 6.2. Pozvana jezgrina funkcija nalazi se iznad `SyS_open()` funkcije koja se poziva prilikom zaustavlja praćenja s funkcijom `zaustavi_pracnje_jezgrinih_funckija()`. Nakon što su pronađene pozvane jezgrine funkcije, moguće je provesti mjerenja.

Ispis 6.1: Primjer korištenja pomoćnih funkcija za alat `ftrace`

```
zapocni_pracenje_jezgrinih_funckija(1);
sendto(s, message, strlen(message) , 0 , (struct sockaddr *) &si_other, slen);
zaustavi_pracnje_jezgrinih_funckija();
```

Ispis 6.2: Primjer rezultata dobivenih korištenjem pomoćnih funkcija

```
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
  1)   0.937 us   |  mutex_unlock();
  1)   0.521 us   |  __fsnotify_parent();
  1)   2.135 us   |  fsnotify();
  1)   0.782 us   |  __sb_end_write();
  1)   5.885 us   |  SyS_close();
  1) + 12.917 us  |  do_work_pending();
  1) ! 186.354 us |  SyS_sendto();
  1) + 93.021 us  |  SyS_open();
  1)              |  SyS_write() {}
```

6.3. Primjena razvijene metodologije

U nastavku su prikazani rezultati mjerenja performansi za odabrane funkcije međuprocesne komunikacije. Prikazano je mjerenje performansi jezgrinih funkcija za slanje i primanje podataka protokolom UDP.

6.3.1. Mjerenja performansi za međuprocesnu komunikaciju

Tablica 6.1 sadrži izmjerene instrukcije i cikluse za odabrane funkcije međuprocesne komunikacije. Mjerenja su obavljena za testnu konfiguraciju s Broadcom procesorom. Za mjerenje broja instrukcija i ciklusa korišteno je `perf_events` sučelje. Brojani su događaji iz jezgrinog i korisničkog prostora. Za svaku funkciju obavljeno je dvadeset mjerenja.

Tablica 6.1: Mjerenja za odabrane funkcije međuprocesne komunikacija

	Broj ciklusa		Broj instrukcija	
Funkcija	$\bar{x}$	σ	$\bar{x}$	σ
`sem_wait()`	3837.1875	808.952	1779.939	474.832
`sem_post()`	3682	618.000	1725.321	173.143
`sem_close()`	71269.727	3058.398	14178.515	701.454
`sem_unlink()`	51216.864	4016.184	9026.432	195.6746

Uz signal takta procesora od 900 MHz prosječno vrijeme izvođenja odabranih funkcija međuprocesne komunikacije dano je u tablici 6.2.

Tablica 6.2: Prosječno vrijeme izvođenja odabranih funkcija međuprocesne komunikacije

Funkcija	Prosječno vrijeme izvođenja (μs)
`sem_wait()`	4.263
`sem_post()`	4.091
`sem_close()`	79.188
`sem_unlink()`	56.907

6.3.2. Mjerenja za jezgrine funkcije prilikom slanja podataka UDP-om

Mjerenja su provedena za testnu konfiguraciju s Broadcom procesorom. Promatrano je vrijeme obrade u jezgri prilikom slanja 1024 bajta funkcijom `sendto()`. Jezgrine funkcije praćene su do dubine 5 i rezultat praćenja je dan u ispisu 6.3

Ispis 6.3: Prikaz jezgrinih funkcija pozvanih s `sendto()`

```
0)               |  SyS_sendto() {
0)               |    sockfd_lookup_light() {
0)               |      __fdget() {
0)   0.625 us    |        __fget_light();
0)   5.990 us    |      }
0) + 11.563 us   |    }
0)               |    move_addr_to_kernel() {
0)   0.729 us    |      __copy_from_user();
0)   6.355 us    |    }
0)               |    sock_sendmsg() {
0)               |      inet_sendmsg() {
0)   0.573 us    |        __rcu_read_lock();
0)   0.573 us    |        __rcu_read_unlock();
0)               |        udp_sendmsg() {
0)   0.521 us    |          __rcu_read_lock();
0)   0.572 us    |          __rcu_read_unlock();
0) + 20.677 us   |          ip_route_output_flow();
0) + 44.896 us   |          ip_make_skb();
0) ! 123.958 us  |          udp_send_skb();
0)   0.572 us    |          dst_release();
0) ! 230.625 us  |        }
0) ! 247.968 us  |      }
0) ! 254.010 us  |    }
0) ! 292.604 us  |  }
```

Tablica 6.3 sadrži izmjerene instrukcije i cikluse za funkcije iz ispisa 6.3 koje značajnije utječu na izvođenje. Mjerenja su obavljena korištenjem alata LTTng. Za svaku funkciju obavljeno je dvadeset mjerenja.

Tablica 6.3: Mjerenja za jezgrine funkcije prilikom slanja 1024 bajta UDP-om

	Broj ciklusa		Broj instrukcija	
Funkcija	$\bar{x}$	σ	$\bar{x}$	σ
`SyS_sendto()`	235958.72	20141.82	105235.1	1750.987
`sock_sendmsg()`	189310.92	17049.63	81071	2010.508
`inet_sendmsg()`	162169.37	2140.570	70016.571	1743.912
`udp_sendmsg()`	141113.214	9993.62	60283.562	1503.929
`ip_route_output_flow()`	30913.25	2218.524	15115.222	22.468
`ip_make_skb()`	43671.8	3742.751	17164.6	33.157
`udp_send_skb()`	75103.764	776.975	31384.529	1497.389

Uz signal takta procesora od 900 MHz prosječno vrijeme izvođenja funkcija iz tablice 6.3 dano je u tablici 6.4.

Tablica 6.4: Prosječno vrijeme izvođenja za jezgrine funkcije prilikom slanja 1024 bajta UDP-om

Funkcija	Prosječno vrijeme izvođenja (μs)
`SyS_sendto()`	262.176
`sock_sendmsg()`	210.345
`inet_sendmsg()`	180.188
`udp_sendmsg()`	156.794
`ip_route_output_flow()`	34.348
`ip_make_skb()`	48.524
`udp_send_skb()`	83.448

6.3.3. Mjerenja za jezgrine funkcije prilikom primanja podataka UDP-om

Mjerenja su provedena za testnu konfiguraciju s Broadcom procesorom. Promatrano je vrijeme obrade u jezgri prilikom primanja 1024 bajta funkcijom `recvfrom()`. Jezgrine funkcije su praćene do dubine 5 i rezultat praćenja je dan u ispisu 6.4

Ispis 6.4: Prikaz jezgrinih funkcija pozvanih s `recvfrom()`

```
 0)               |  SyS_recvfrom() {
 0)               |    sockfd_lookup_light() {
 0)               |      __fdget() {
 0)   0.833 us    |        __fget_light();
 0)   6.354 us    |      }
 0) + 12.136 us   |    }
 0)               |    sock_recvmsg() {
 0)               |      inet_recvmsg() {
 0)   0.573 us    |        __rcu_read_lock();
 0)   0.573 us    |        __rcu_read_unlock();
 0)               |        udp_recvmsg() {
 0)   4.219 us    |          __skb_recv_datagram();
 0) + 11.666 us   |          skb_copy_datagram_iovec();
 0) + 21.302 us   |          skb_free_datagram_locked();
 0) + 57.084 us   |        }
 0) + 75.000 us   |      }
 0) + 81.198 us   |    }
 0)               |    move_addr_to_user() {
 0)   1.614 us    |      __copy_to_user();
 0)   8.177 us    |    }
 0) ! 118.177 us  |  }
```

Tablica 6.5 sadrži izmjerene instrukcije i cikluse za funkcije iz ispisa 6.4 koje značajnije utječu na izvođenje. Mjerenja su obavljena korištenjem alata LTTng. Za svaku funkciju obavljeno je dvadeset mjerenja.

Uz signal takta procesora od 900 MHz prosječno vrijeme izvođenja funkcija iz tablice 6.5 dano je u tablici 6.6.

Tablica 6.5: Mjerenja za jezgrine funkcije prilikom primanja 1024 bajta UDP-om

	Broj ciklusa		Broj instrukcija	
Funkcija	$\bar{x}$	σ	$\bar{x}$	σ
`SyS_recvfrom()`	141723.090	3054.59	71458.181	204.308
`sock_recvmsg()`	93587.375	3796.085	47650.625	776.294
`inet_recvmsg()`	74676.388	3004.460	37597.047	704.035
`udp_recvmsg()`	63820.6	2856.067	32998.1	131.625
`skb_copy_datagram_iovec()`	27539.187	1258.563	14143.25	17.275
`skb_free_datagram_locked()`	30963.636	1110.154	15355.818	27.461

Tablica 6.6: Prosječno vrijeme izvođenja jezgrinih funkcije prilikom primanja 1024 bajta UDP-om

Funkcija	Prosječno vrijeme izvođenja (μs)
`SyS_recvfrom()`	157.470
`sock_recvmsg()`	103.98
`inet_recvmsg()`	82.973
`udp_recvmsg()`	70.917
`skb_copy_datagram_iovec()`	30.599
`skb_free_datagram_locked()`	34.404

7. Zaključak

U ovom radu predložena je metodologija mjerenja performansi koja koristi jedinicu za mjerenje performansi. Prikazano je `pref_events` sučelje Linux jezgre kojim je omogućen pristup jedinici za mjerenje performansi. Problem uočen prilikom korištenja `perf_events` sučelja je nepostojanje odgovarajuće dokumentacije. Prebrojavanjem instrukcija utvrđeno je da su jedinice za mjerenje performansi dva različita procesora prilikom brojanja instrukcija vrlo točne. Alatom LTTng omogućeno je mjerenje performansi jezgrinih funkcija. Kao moguća alternativa alatu LTTng dan je jezgrin modul koji definira jezgrine sonde i čita brojače jedinice za mjerenje performansi. Usporedbom rezultata dobivenih jezgrinim modulom i alatom LTTng, utvrđeno je da su vrijednosti dobivene jezgrinim modulom manje i da alat LTTng unosi određeno odstupanje. Razvoj jezgrinog modula zahtijevao je detaljno proučavanje registara specifične jedinice za mjerenje performansi i nije primjenjiv za druge arhitekture procesora. Kroz ovaj rad jedinica za mjerenje performansi korištena je samo za brojanje instrukcija i ciklusa. Prilikom provedbe mjerenja mogu se brojati i ostali događaji podržani jedinicom za mjerenje performansi.

Literatura

[1] ftrace - function tracer. https://www.kernel.org/doc/Documentation/trace/ftrace.txt. Pristupljeno: 27.06.2015.

[2] Kernel lab. https://www.yoctoproject.org/training/kernel-lab, . Pristupljeno: 27.06.2015.

[3] The linux kernel module programming guide. http://linux.die.net/lkmpg/x40.html, . Pristupljeno: 27.06.2015.

[4] An introduction to kprobes. https://lwn.net/Articles/132196/, . Pristupljeno:.

[5] Kernel probes (kprobes). https://www.kernel.org/doc/Documentation/kprobes.txt, . Pristupljeno: 27.06.2015.

[6] Mrežno programiranje, fakultet elektortehnike i računarstva zagreb. http://www.fer.unizg.hr/predmet/mrepro. Pristupljeno: 27.06.2015.

[7] Perf event open. http://web.eece.maine.edu/~vweaver/projects/perf_events/perf_event_open.html. Pristupljeno: 27.06.2015.

[8] On the value of static tracepoints. https://lwn.net/Articles/330402/, . Pristupljeno: 27.06.2015.

[9] Using the linux kernel tracepoints. https://www.kernel.org/doc/Documentation/trace/tracepoints.txt, . Pristupljeno: 27.06.2015.

[10] Yocto project board support package. http://www.yoctoproject.org/docs/1.8/bsp-guide/bsp-guide.html, . Pristupljeno: 27.06.2015.

[11] Yocto project mega-manual. http://www.yoctoproject.org/docs/1.8/mega-manual/mega-manual.html, . Pristupljeno: 27.06.2015.

[12] Marco Cesat Daniel P. Bovet. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, Inc, 2006.

[13] David A. Patterson John L. Hennessy. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, 2012.

[14] Michael Kerrisk. *The Linux programming interface*. No Starch Press, Inc., 2010.

[15] Mario Kovač. *Arhitektura računala*. Fakultet elektrotehnike i računarstva, 2015.

[16] Robert Love. *Linux Kernel Development, Third Edition*. Pearson Education, Inc., 2010.

[17] Daiane Angolini Otavio Salvador. *Embedded Linux Development with Yocto Project*. Packt Publishing Ltd., 2014.

[18] Joshua Ruggiero. Measuring cache and memory latency and cpu to memory bandwidth. *Intel White Paper*, stranice 1–14, 2008.

[19] Andrew S. Tanenbaum. *Structured computer organization, Sixth edition*. Pearson Education, Inc., 2013.

Razvoj metodologije finog mjerenja performansi na operacijskom sustavu Linux

Sažetak

Ovaj rad izlaže mogućnosti Yocto Projecta i metodologiju koja koristi jedinicu za mjerenje performansi. Prikazana metodologija pokriva mjerenje performansi u jezgri operacijskog sustava i korisničkim programima.

Ključne riječi: Yocto Project, perf_events, Kprobes, LTTng, PMU, ftrace, Raspberry Pi 2

Developing methodology for fine grained performance measurements in Linux operating system

Abstract

This thesis presents the possibilities of the Yocto Project and the methodology which uses the performance monitoring unit. Presented methodology covers performance measuring in kernel and user programs.

Keywords: Yocto Project, perf_events, Kprobes, LTTng, PMU, ftrace, Raspberry Pi 2

Dodatak A
Program za testiranje broja instrukcija

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/syscall.h>
#include <linux/perf_event.h>
#include <asm/unistd.h>


int perf_event_open(struct perf_event_attr *attr, pid_t pid, int cpu,
                    int group_fd, unsigned long flags)
{
    return syscall(__NR_perf_event_open, attr, pid, cpu, group_fd, flags);
}


void zbroji_matrice(int **matrica_a, int **matrica_b, int **matrica_rez, int
    red)
{
    for(int i = 0; i < red; i++)
    {
        for(int j = 0; j < red; j++)
        {
            matrica_rez[i][j] = matrica_a[i][j] + matrica_b[i][j];
        }
    }
}

int main(int argc, char **argv)
{
    int **matrica_a;
    int **matrica_b;
    int **matrica_rez;

    struct perf_event_attr pe;
```

```
long long count;
int fd;

if(argc != 2)
{
    printf("Potrebno je predati red kvadratne matrice!!\n");
    return -1;
}

int red = atoi(argv[1]);

matrica_a = calloc(red, sizeof(int *));
matrica_b = calloc(red, sizeof(int *));
matrica_rez = calloc(red, sizeof(int *));

for(int i = 0; i < red; i++)
{
    matrica_a[i] = calloc(red, sizeof(int));
    matrica_b[i] = calloc(red, sizeof(int));
    matrica_rez[i] = calloc(red, sizeof(int));
}

memset(&pe, 0, sizeof(struct perf_event_attr));
pe.type = PERF_TYPE_HARDWARE;
pe.size = sizeof(struct perf_event_attr);
pe.config = PERF_COUNT_HW_INSTRUCTIONS;
pe.disabled = 1;
pe.exclude_kernel = 1;
pe.exclude_hv = 1;

fd = perf_event_open(&pe, 0, -1, -1, 0);
if (fd == -1)
{
    fprintf(stderr, "Pogreska prilikom otvaranja voditelja %llx\n",
        pe.config);
    exit(EXIT_FAILURE);
}

ioctl(fd, PERF_EVENT_IOC_RESET, 0);
ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);

zbroji_matrice(matrica_a, matrica_b, matrica_rez, red);

ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);
read(fd, &count, sizeof(long long));

printf("Iskoristeno %lld instrukcija\n", count);
close(fd);

for(int i = 0; i < red; i++)
{
    free(matrica_a[i]);
    free(matrica_b[i]);
    free(matrica_rez[i]);
}
free(matrica_a);
```

```
    free(matrica_b);
    free(matrica_rez);

    return 0;
}
```

Dodatak B
Instrukcije funkcije
`zbroji_matrice()`

Ispis B.1: Prikaz instrukcija za konfiguraciju s Broadcom procesorom

```
000105e0 <zbroji_matrice>:
   105e0: e52db004  push {fp}  ; (str fp, [sp, #-4]!)
   105e4: e28db000  add fp, sp, #0
   105e8: e24dd00c  sub sp, sp, #12
   105ec: e3a03000  mov r3, #0
   105f0: e50b3008  str r3, [fp, #-8]
   105f4: ea00001e  b 10674 <zbroji_matrice+0x94>
   105f8: e3a03000  mov r3, #0
   105fc: e50b300c  str r3, [fp, #-12]
   10600: ea000015  b 1065c <zbroji_matrice+0x7c>
   10604: e30039c8  movw r3, #2504 ; 0x9c8
   10608: e3403002  movt r3, #2
   1060c: e51b1008  ldr r1, [fp, #-8]
   10610: e51b200c  ldr r2, [fp, #-12]
   10614: e0812002  add r2, r1, r2
   10618: e7932102  ldr r2, [r3, r2, lsl #2]
   1061c: e30039d0  movw r3, #2512 ; 0x9d0
   10620: e3403002  movt r3, #2
   10624: e51b0008  ldr r0, [fp, #-8]
   10628: e51b100c  ldr r1, [fp, #-12]
   1062c: e0801001  add r1, r0, r1
   10630: e7933101  ldr r3, [r3, r1, lsl #2]
   10634: e0821003  add r1, r2, r3
   10638: e30039cc  movw r3, #2508 ; 0x9cc
   1063c: e3403002  movt r3, #2
   10640: e51b0008  ldr r0, [fp, #-8]
   10644: e51b200c  ldr r2, [fp, #-12]
   10648: e0802002  add r2, r0, r2
   1064c: e7831102  str r1, [r3, r2, lsl #2]
   10650: e51b300c  ldr r3, [fp, #-12]
   10654: e2833001  add r3, r3, #1
   10658: e50b300c  str r3, [fp, #-12]
   1065c: e51b300c  ldr r3, [fp, #-12]
   10660: e3530000  cmp r3, #0
   10664: daffffe6  ble 10604 <zbroji_matrice+0x24>
```

```
  10668: e51b3008  ldr r3, [fp, #-8]
  1066c: e2833001  add r3, r3, #1
  10670: e50b3008  str r3, [fp, #-8]
  10674: e51b3008  ldr r3, [fp, #-8]
  10678: e3530000  cmp r3, #0
  1067c: daffffdd  ble 105f8 <zbroji_matrice+0x18>
  10680: e24bd000  sub sp, fp, #0
  10684: e49db004  pop {fp}  ; (ldr fp, [sp], #4)
  10688: e12fff1e  bx lr
```

Ispis B.2: Prikaz instrukcija za konfiguraciju s Intel procesorom

```
000000000004008d1 <zbroji_matrice>:
  4008d1: 55                 push  %rbp
  4008d2: 48 89 e5           mov   %rsp,%rbp
  4008d5: 48 89 7d e8        mov   %rdi,-0x18(%rbp)
  4008d9: 48 89 75 e0        mov   %rsi,-0x20(%rbp)
  4008dd: 48 89 55 d8        mov   %rdx,-0x28(%rbp)
  4008e1: 89 4d d4           mov   %ecx,-0x2c(%rbp)
  4008e4: c7 45 f8 00 00 00 00  movl $0x0,-0x8(%rbp)
  4008eb: e9 93 00 00 00     jmpq  400983 <zbroji_matrice+0xb2>
  4008f0: c7 45 fc 00 00 00 00  movl $0x0,-0x4(%rbp)
  4008f7: eb 7a              jmp   400973 <zbroji_matrice+0xa2>
  4008f9: 8b 45 f8           mov   -0x8(%rbp),%eax
  4008fc: 48 98              cltq
  4008fe: 48 8d 14 c5 00 00 00  lea 0x0(,%rax,8),%rdx
  400905: 00
  400906: 48 8b 45 d8        mov   -0x28(%rbp),%rax
  40090a: 48 01 d0           add   %rdx,%rax
  40090d: 48 8b 00           mov   (%rax),%rax
  400910: 8b 55 fc           mov   -0x4(%rbp),%edx
  400913: 48 63 d2           movslq %edx,%rdx
  400916: 48 c1 e2 02        shl   $0x2,%rdx
  40091a: 48 01 d0           add   %rdx,%rax
  40091d: 8b 55 f8           mov   -0x8(%rbp),%edx
  400920: 48 63 d2           movslq %edx,%rdx
  400923: 48 8d 0c d5 00 00 00  lea 0x0(,%rdx,8),%rcx
  40092a: 00
  40092b: 48 8b 55 e8        mov   -0x18(%rbp),%rdx
  40092f: 48 01 ca           add   %rcx,%rdx
  400932: 48 8b 12           mov   (%rdx),%rdx
  400935: 8b 4d fc           mov   -0x4(%rbp),%ecx
  400938: 48 63 c9           movslq %ecx,%rcx
  40093b: 48 c1 e1 02        shl   $0x2,%rcx
  40093f: 48 01 ca           add   %rcx,%rdx
  400942: 8b 0a              mov   (%rdx),%ecx
  400944: 8b 55 f8           mov   -0x8(%rbp),%edx
  400947: 48 63 d2           movslq %edx,%rdx
  40094a: 48 8d 34 d5 00 00 00  lea 0x0(,%rdx,8),%rsi
  400951: 00
  400952: 48 8b 55 e0        mov   -0x20(%rbp),%rdx
  400956: 48 01 f2           add   %rsi,%rdx
  400959: 48 8b 12           mov   (%rdx),%rdx
  40095c: 8b 75 fc           mov   -0x4(%rbp),%esi
  40095f: 48 63 f6           movslq %esi,%rsi
```

```
400962: 48 c1 e6 02        shl   $0x2,%rsi
400966: 48 01 f2           add   %rsi,%rdx
400969: 8b 12              mov   (%rdx),%edx
40096b: 01 ca              add   %ecx,%edx
40096d: 89 10              mov   %edx,(%rax)
40096f: 83 45 fc 01        addl  $0x1,-0x4(%rbp)
400973: 8b 45 fc           mov   -0x4(%rbp),%eax
400976: 3b 45 d4           cmp   -0x2c(%rbp),%eax
400979: 0f 8c 7a ff ff ff  jl    4008f9 <zbroji_matrice+0x28>
40097f: 83 45 f8 01        addl  $0x1,-0x8(%rbp)
400983: 8b 45 f8           mov   -0x8(%rbp),%eax
400986: 3b 45 d4           cmp   -0x2c(%rbp),%eax
400989: 0f 8c 61 ff ff ff  jl    4008f0 <zbroji_matrice+0x1f>
40098f: 5d                 pop   %rbp
400990: c3                 retq
```

Dodatak C
Primjeri korištenja POSIX API-ja za dijeljenu memoriju

Ispis C.1: Program koji čita iz objekta dijeljene memorije

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int fd;
    void *adresa;
    struct stat sb;

    if(argc != 2)
    {
        printf("Potrebno je unijeti naziv objekta dijeljene memorije!\n");
        return -1;
    }

    fd = shm_open(argv[1], O_RDONLY, 0);
    if(fd == -1)
    {
        printf("Pogreska prilikom poziva shm_open()!\n");
        return -1;
    }

    /* Pronalazak duljine objekta dijeljene
       memorije */
    if(fstat(fd, &sb) == -1)
    {
        printf("Pogreska prilikom poziva funkcije fstat()!\n");
        return -1;
    }
```

```
    adresa = mmap(NULL, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
    if(adresa == MAP_FAILED)
    {
        printf("Pogreska prilikom poziva mmap()!\n");
        return -1;
    }

    write(STDOUT_FILENO, adresa, sb.st_size);
    printf("\n");

    if(shm_unlink(argv[1]) == -1)
    {
        printf("Pogreska prilikom poziva shm_unlink()!\n");
    }

    return 0;
}
```

Ispis C.2: Primjer programa koji kreira objekt djeljnje memorije

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int fd;
    void *adresa;
    int duljina;

    if(argc != 3)
    {
        printf("Potrebno je unijeti naziv objekta dijeljene memorije i
            poruku!!\n");
        return -1;
    }

    fd = shm_open(argv[1], O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
    if(fd == -1)
    {
        printf("Pogreska prilikom poziva shm_open()!\n");
        return -1;
    }

    duljina = strlen(argv[2]);
    if(ftruncate(fd, duljina) == -1)
    {
        printf("Pogreska prilikom poziva ftruncate()!\n");
        return -1;
     }
```

```
    adresa = mmap(NULL, duljina, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if(adresa == MAP_FAILED)
    {
        printf("Pogreska prilikom poziva mmap()!\n");
        return -1;
    }

    memcpy(adresa, argv[2], duljina);

    return 0;
}
```

Dodatak D
Jezgrin modul

Ispis D.1: Jezgrin modul koji koristi Kprobes sučelje

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/sched.h>
#include <linux/limits.h>

static int prvo_ocitanje;
static int drugo_ociatnje;

void inicijaliziraj_pmu(void *d)
{
    asm volatile("mcr p15, 0, %0, c9, c12, 0" : : "r"(0x00000007));
    asm volatile("mcr p15, 0, %0, c9, c12, 5" : : "r"(0x00000000));
    asm volatile("mcr p15, 0, %0, c9, c13, 1" : : "r"(0x00000008));
    asm volatile("mcr p15, 0, %0, c9, c12, 1" : : "r"(0x00000001));
}

static int ulazni_rukovatelj(struct kretprobe_instance *ri,
                             struct pt_regs *regs)
{
    asm volatile("mcr p15, 0, %0, c9, c12, 5" : : "r"(0x0));
    asm volatile("mrc p15, 0, %0, c9, c13, 2" : "=r"(prvo_ocitanje));

    return 0;
}

static int povratni_rukovatelj(struct kretprobe_instance *ri,
                               struct pt_regs *regs)
{
    asm volatile("mcr p15, 0, %0, c9, c12, 5" : : "r"(0x0));
    asm volatile("mrc p15, 0, %0, c9, c13, 2" : "=r"(drugo_ociatnje));

    printk("kprobe_modul: Izlaz iz udp_v4_get_port, Prvo ocitanje: %d, Drugo
        ocitanje %d .\n", prvo_ocitanje, drugo_ociatnje);

    return 0;
}
```

```
static struct kretprobe kret_sonda = {
    .handler = povratni_rukovatelj,
    .entry_handler = ulazni_rukovatelj,
    .maxactive = 1,
};

int init_module(void)
{
    int ret;

    kret_sonda.kp.symbol_name = "udp_v4_get_port";

    on_each_cpu(inicijaliziraj_pmu, NULL, 1);

    ret = register_kretprobe(&kret_sonda);
    if(ret < 0)
    {
        printk("Registracija sonde neuspjesna, povratna vrijednost %d\n", ret);
        return -1;
    }
    printk("Sonda postavljena na %s: %p\n", kret_sonda.kp.symbol_name,
        kret_sonda.kp.addr);
    return 0;
}

void cleanup_module(void)
{
    unregister_kretprobe(&kret_sonda);
    printk("Sonda na %p uklonjena\n", kret_sonda.kp.addr);
}

MODULE_LICENSE("GPL");
```

Dodatak E
ftrace pomoćne funkcije

Ispis E.1: Pomoćne funkcije za rad s alatom ftrace

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
void ocisti_zapise()
{
    int fd;

    fd = open("/sys/kernel/debug/tracing/current_tracer", O_WRONLY);
    if(fd < 0)
    {
        printf("Pogreska prilikom otvaranja datoteke current_tracer za
            ciscenje zapis!\n");
        exit(EXIT_FAILURE);
    }
    write(fd, "nop", 3);
    close(fd);
}

void zapocni_pracenje_jezgrinih_funckija(int dubina)
{
    int fd;
    int s;
    char linija[64];

    fd = open("/sys/kernel/debug/tracing/set_ftrace_pid", O_WRONLY);
    if(fd < 0)
    {
        printf("Pogreska prilikom otvaranja datoteke set_ftrace_pid!\n");
        exit(EXIT_FAILURE);
    }
    s = sprintf(linija, "%d\n", getpid());
    write(fd, linija, s);
    close(fd);

    fd = open("/sys/kernel/debug/tracing/max_graph_depth", O_WRONLY);
    if(fd < 0)
    {
```

```
        printf("Pogreska prilikom otvaranja datoteke max_graph_depth!\n");
        exit(EXIT_FAILURE);
    }
    s = sprintf(linija, "%d\n", dubina);
    write(fd, linija, s);
    close(fd);

    fd = open("/sys/kernel/debug/tracing/current_tracer", O_WRONLY);
    if(fd < 0)
    {
        printf("Pogreska prilikom otvaranja datoteke current_tracer\n");
        exit(EXIT_FAILURE);
    }
    write(fd, "function_graph", 14);
    close(fd);

    fd = open("/sys/kernel/debug/tracing/tracing_on", O_WRONLY);
    if(fd < 0)
    {
        printf("Pogreska prilikom otvaranja datoeke tracing_on!\n");
        exit(EXIT_FAILURE);
    }
    write(fd, "1", 1);
    close(fd);
}

void zaustavi_pracnje_jezgrinih_funckija()
{
    int fd;

    fd = open("/sys/kernel/debug/tracing/tracing_on", O_WRONLY);
    if(fd < 0)
    {
        exit(EXIT_FAILURE);
        printf("Pogreska prilikom otvaranja datoeke tracing_on!\n");
    }
    write(fd, "0", 1);
    close(fd);
}
```

Dodatak F
Primjer korištenja POSIX API-ja za rad sa semaforima

Ispis F.1: Program koji radi sa semaforom

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <signal.h>
#include <unistd.h>
#include <time.h>


void rukovatelj(int sig)
{
    exit(EXIT_SUCCESS);
}

int main(int argc, char *argv[])
{
    sem_t *sem;
    int spavanje;
    int vrijednost_semafora;
    int kreiraj;
    char *koristenje = "naziv_semafora trajanje_spavanja k/o";
    clock_t t;

    if(argc != 4)
    {
        printf("Argumenti: %s\n", koristenje);
        return -1;
    }

    if(argv[3][0] == 'k')
    {
        kreiraj = 1;
    }
```

```
    else if(argv[3][0] == 'o')
    {
        kreiraj = 0;
    }
    else
    {
        printf("Argumenti: %s\n", koristenje);
        return -1;
    }

    spavanje = atoi(argv[2]);
    signal(SIGINT, rukovatelj);

    if(kreiraj)
    {
        printf("Kreiram semafor!!\n");
        sem = sem_open(argv[1], O_CREAT, S_IRUSR | S_IWUSR, 1);
    }
    else
    {
        sem = sem_open(argv[1], 0);
    }

    if(sem == SEM_FAILED)
    {
        printf("Pogreska prilikom poziva sem_open()!\n");
        return -1;
    }

    while(1)
    {
        t = clock();
        printf("[%lu] Cekam na semafor!\n", t);
        sem_wait(sem);

        t = clock();
        printf("[%lu] Dobio semafor!\n", t);
        sleep(spavanje);

        t = clock();
        printf("[%lu] Otpustam semafor!\n", t);
        if(sem_post(sem) == -1)
        {
            printf("Pogreska prilikom poziva sem_post()!\n");
        }
        sleep(2);
    }

    if(kreiraj && sem_unlink(argv[1]) == -1)
    {
        printf("Pogreska prilikom poziva sem_unlink()!\n");
        return -1;
    }

    return 0;
}
```

Dodatak G
Primjeri za UDP komunikaciju priključnicama

Ispis G.1: Primjer UDP klijenta

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <signal.h>
#include <fcntl.h>

int sock;

void rukovatelj(int sig)
{
    close(sock);
    exit(EXIT_SUCCESS);
}

int main(int argc, char *argv[])
{
    struct sockaddr_in posluzitelj;
    int slen;
    int broj_bajtova;
    char *ip_adresa_server;
    char *port;
    char *buf;

    if(argc != 4)
    {
        printf("Argumenti ne valjaju!!\n");
        exit(EXIT_FAILURE);
    }

    ip_adresa_server = argv[1];
    port = argv[2];
    broj_bajtova = atoi(argv[3]);
    buf = calloc(broj_bajtova, sizeof(char));
```

```
    slen = sizeof(posluzitelj);
    signal(SIGINT, rukovatelj);

    printf("Slat cu %d bajtova!!\n", broj_bajtova);

    if ((sock = socket(PF_INET, SOCK_DGRAM, 0)) == -1)
    {
        printf("Pogreska prilikom kreiranja prikljucnice!\n");
        exit(EXIT_FAILURE);
    }

    memset((char *) &posluzitelj, 0, sizeof(posluzitelj));
    posluzitelj.sin_family = AF_INET;
    posluzitelj.sin_port = htons(atoi(port));

    if (inet_aton(ip_adresa_server, &posluzitelj.sin_addr) == 0)
    {
        printf("Pogreska prilikom poziva inet_aton()\n");
        exit(EXIT_FAILURE);
    }

    int primljeno;
    int poslano;
    while(1)
    {
        poslano = sendto(sock, buf, broj_bajtova , 0 , (struct sockaddr *)
            &posluzitelj, slen);
        printf("Poslao %d bajtova!!\n", poslano);

        primljeno = recvfrom(sock, buf, broj_bajtova, 0, (struct sockaddr *)
            &posluzitelj, &slen);

        printf("Primio %d bajtova!!\n", primljeno);
        sleep(4);
    }

    return 0;
}
```

Ispis G.2: Primjer UDP poslužitelja

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <signal.h>

int sock;

void rukovatelj(int sig)
{
    close(sock);
    exit(EXIT_SUCCESS);
}
```

```
int main(int argc, char *argv[])
{
    int poslano_bajtova;
    int primljeno_bajtova;
    char *spremnik;
    int velicina_spremnika;
    struct sockaddr_in adresa, klijent;
    int slen;

    if(argc != 3)
    {
        printf("Pogresni argumenti!\n");
        exit(EXIT_FAILURE);
    }

    velicina_spremnika = atoi(argv[2]);
    spremnik = calloc(velicina_spremnika, sizeof(char));
    slen = sizeof(klijent);
    signal(SIGINT, rukovatelj);

    if((sock = socket(PF_INET, SOCK_DGRAM, 0)) == -1)
    {
        printf("Pogreska prilikom kreiranja prikljucnice!\n");
        exit(EXIT_FAILURE);
    }

    memset((char *) &adresa, 0, sizeof(adresa));
    adresa.sin_family = AF_INET;
    adresa.sin_port = htons(atoi(argv[1]));
    adresa.sin_addr.s_addr = htonl(INADDR_ANY);

    if(bind(sock, (struct sockaddr*) &adresa, sizeof(adresa) ) == -1)
    {
        printf("Pogreska prilikom povezivanja prikljucnice na adresu!\n");
        exit(EXIT_FAILURE);
    }

    while(1)
    {
        primljeno_bajtova = recvfrom(sock, spremnik, velicina_spremnika, 0,
            (struct sockaddr *) &klijent, &slen);
        printf("Primio %d bajtova od %s:%d\n", primljeno_bajtova,
            inet_ntoa(adresa.sin_addr), ntohs(adresa.sin_port));

        poslano_bajtova = sendto(sock, spremnik, primljeno_bajtova, 0, (struct
            sockaddr*) &klijent, slen);
        printf("Poslao natrag %d bajtova!\n", poslano_bajtova);
    }

    return 0;
}
```