SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 550

Prilagodba Linuxova raspoređivača za mnogoprocesorske sustave

Luka Milić

Zagreb, srpanj 2013.

Fǭlâ ti, škȍlju mộj,
za ditînjstvo,
za prâče i skrâče,
ȍbruči i karȍce;
za bočộtu vȍdu kû son pî
i mânule î,
zâ zrno sȍli na ûsnici,
lîpu rȉbu na ȕdici,
stộru zîkvu,
za zdộh meštrǭlâ
i besȉdu is oltǭrâ.

Fǭlâ ti, škȍlju mộj,
za kûs zemljệ drộge
čȁ š mi dâti.
U njǭdrȉman tvojȉman
za vâjk ču počīvâti.

SADRŽAJ

1. Uvod **1**

2. Opis raspoređivanja u Linuxu **2**

2.1. Raspoređivanje operacijskih sustava na višeprocesorskim sustavima . 2

2.2. Općenite zamisli za poboljšanje 3

2.3. Opis Linuxova raspoređivača 4

2.4. Algoritmi guranja i povlačenja 8

2.5. Nedostatci Linuxova raspoređivača 11

3. Izmjene u svrhu poboljšanja algoritma raspoređivanja **13**

3.1. Zamisli za poboljšanje Linuxova raspoređivača 13

3.2. Dodavanje nove rasporedbene politike 15

3.3. Procesi i niti u Linuxu 17

3.4. Čuvanje popisa svih procesa 19

3.5. Povezani popisi u Linuxovoj jezgri 22

3.6. Zaključavanje struktura u Linuxovoj jezgri 23

3.7. Ostvaraj izmjena 23

3.8. Zamatanje promijenjena koda u makroe 28

4. Opis ispitne okoline **31**

4.1. Prva inačica ispitne okoline 31

4.2. Druga inačica ispitne okoline 32

4.3. Treća inačica ispitne okoline 32

4.4. Konačna ispitna okolina 33

4.5. Početna ispitna jezgra 35

4.6. Prva ispitna jezgra 36

4.7. Druga ispitna jezgra 36

4.8. Pokretanje sustava 37

4.9. Prevođenje sustava 38
4.10. Dodatna ispitna jezgra 38

5. Rezultati **41**
5.1. Očekivani rezultati 41
5.2. Opis ispitivanja 43
5.3. Mjerilo poboljšanja 44
5.4. Dobiveni rezultati 46
5.5. Komentar rezultata 55

6. Zaključak **56**

Literatura **57**

1. Uvod

Današnji operacijski sustavi u višeprocesorskim sustavima ostvaruju razmjerno naivne algoritme koji ne mogu u potpunosti iskoristiti sve složeniju procesorsku hijerarhiju. Posebno, ostvareni algoritmi obično ne uzimaju u obzir to da niti istoga procesa dijele isti memorijski prostor i da bi ih zato bilo poželjno staviti što bliže jednu drugoj. Jedan od operacijskih sustava koji uopće ne uzima u obzir ikakvu susjednost zadataka pri raspoređivanju upravo je Linux. Zato je Linux i odabran kao operacijski sustav na kojem će se pokušati provesti u djelo algoritam koji će uzimati u obzir povezanost.

U prvom poglavlju opisuje se sustav raspoređivanja kako je izveden na Linuxu. U drugom poglavlju opisuje se kako je taj sustav izmijenjen za uklapanje algoritma bliskosti zadataka. U trećem poglavlju opisuje se okolina na kojoj se ispitivalo moguće poboljšanje rada u izmijenjenom sustavu. U četvrtom poglavlju opisuju se konkretni rezultati dobiveni na njoj.

2. Opis raspoređivanja u Linuxu

2.1. Raspoređivanje operacijskih sustava na višeprocesorskim sustavima

U ovom se radu, dakle, pokušalo pronaći algoritam raspoređivanja koji će bolje odgovarati mnogoprocesorskim sustavima od onih koji su danas ostvareni u popularnim operacijskim sustavima. Sveza „mnogoprocesorski sustavi" prevedenica je engleske sveze „*manyprocessor systems*" [1].

Naime, u radu se nisu razmatrali jednostavni sustavi poput hipotetskoga sustava na kojem se nalazi samo jedan procesor sa samo jednom fizičkom jezgrom, ali dvije logičke. Odmah se išlo na općenit slučaj više procesora s više hipernitnih fizičkih jezgara, po mogućnosti u čvorovima. Što se tiče odabira operacijskoga sustava za rad, Linux, je, naravno, odabran zbog otvorenosti koda, ali i goleme potpore internetske zajednice.

Dakle, današnji se operacijski sustavi ne odlikuju posebnom prilagođenošću za mnogoprocesorske sustave, iako bez teškoća djeluju na takvim sustavima. Odnosno, njihovi su algoritmi raspoređivanja razmjerno naivni. Algoritmii ne uzimaju u obzir da će uskoro u računalu biti naravna stvar imati nekoliko šesnaesterojezgrenih procesora koje ti algoritmi ne će moći dovoljno iskoristiti. Tako danas već postoje osmojezgreni hipernitni procesori u skupljim računalima [2] i samo preostaje korak nabave nove matične ploče s dvama ili četirima procesorskim utorima.

Razmotrit će se tri najčešća današnja operacijska sustava za stolna, odnosno prijenosna računala. Na takvim se računalima i očekuje stalno sve veće usložnjenje procesorske hijerarhije, budući da je era jednoprocesora (engl. *uniprocessor*) odavno završila. Sustavi su koji će se usporediti na ovom polju Windowsi, Mac OS X i Linux.

Windowsi su, primjerice, usredotočeni na jednoliku raspodjelu procesa svim procesorima, iako razlikuju hipernitne sustave, višejezgrene sustave i sustave NUMA. Posebna je zamjerka algoritmima u današnjim operacijskim sustavima slabo iskori-

štavanje načela susjednosti, prostorne i vremenske, pa makar u osnovnu obliku. Taj bi osnovni oblik bio uzimanje u obzir da su niti višenitnoga procesa vrlo povezane. Windowsi ipak donekle pokušavaju staviti takve niti blizu jednu drugoj određenim mehanizmom najpogodnijega procesora [3]. Ti algoritmi inače nemaju nikakve veze s algoritmima grupiranja koji su ostvareni u radu.

Mac OS X s druge je strane potpuno prilagođen višeprocesorskim sustavima te će zadatci koje raspoređuje imati nešto slabije performanse na rijetkim današnjim sustavima s jednoprocesorom. Ipak, on uopće ne uzima u obzir nikakvu susjednost zadataka [4].

I na Linuxu je slično. Usprkos tomu što jezgra zna za procese i niti, raspoređivač se prema svemu ophodi kao prema generičkomu „zadatku" i uopće ne razmišlja o tom koje niti pripadaju kojemu procesu. Ovo je djelomično i do posebnoga Linuxova shvaćanja procesa i niti, razlikoga od školskoga shvaćanja, o kojem će biti riječi u poglavlju 3. Istina je da bi praćenje iskorištavanja resursa sustava poput priručne memorije zamrsilo algoritme raspoređivanja, ali raspoređivač, recimo, uopće nije svjestan da su dva zadatka zapravo niti istoga procesa.

2.2. Općenite zamisli za poboljšanje

Upravo sa zadnjim razmišljanjem o povezanosti na umu u ovom se radu predlaže pokušaj grupiranja takvih povezanih niti. Algoritam je ostvaren na Linuxu, što pokušava bar donekle iskoristiti načelo susjednosti. O ostvarenju će se govoriti u poglavlju 3. Ponavljam da je to nepovezano s algoritmima u Windowsima iz odjeljka 2.1.

Usprkos zamisli grupiranja s kojom se je išlo, treba spomenuti da Linux vodi opsežne statistike za svaki zadatak poput mehanizma `schedstat` [5], a da se to vidi dovoljno je pretražiti datoteku `kernel/sched/core.c` nizom „`stat`". Ova će datoteka biti objašnjena u poglavlju 3. Vođenje statistika omogućuje naprednije pokušaje raspoređivanja u nekom drugom diplomskom radu.

Kad se već spominju zamisli za druge diplomske radove, možda bi se i moglo pokušati ostvariti neku inačicu načina bliskosti ostvarenoga u Windowsima. Čini se da bi to dobro radilo uz rasporedbenu politiku `SCHED_FIFO`, o kojoj više u odjeljku 2.3. Ova je pak zamisao izvrsno pristala uz politiku `SCHED_RR`.

Valja napomenuti da je područje poboljšanja raspoređivanja u višeprocesorskim sustavima živo, te je predložen niz zamisli kako bi se to moglo izvesti. Te su zamisli, primjerice, korisničko pridjeljivanje povezanosti niti [6] ili praćenje promašaja priručne memorije [7]. O tom se više može vidjeti u seminaru „Raspoređivanje u vi-

šeprocesorskim sustavima“ [8]. Seminar je bio upravo posvećen tomu, a u njem je i predložena upravo nekakva kombinacija toga dvojega spomenutoga.

Naravno, takvo bi nešto zahtijevalo daleko previše vremena, te se ipak išlo s jednostavnom zamišlju popravljanja algoritama raspoređivanja. U spomenutom se seminaru konkretno razmatralo još pet drugih načela, a to su bila uravnotežavanje tovara, ograničavanje pridruživanja zadataka procesorima, emuliranje izvođenja na izoliranoj platformi, korisničko odlučivanje o raspoređivanju i raspodijeljeno raspoređivanje.

Usprkos živomu istraživanju ovoga područja, algoritmi se raspoređivanja najčešće nisu mijenjali godinama. Konkretno, Linuxov se raspoređivač nije mijenjao od 29. studenoga 2007.. Pozorniji pogled na početak datoteke `kernel/sched/core.c`, gdje su popisane sve promjene algoritama, pokazuju da se on vrlo teško mijenja. Preporuka je i pogledati i početak recimo datoteke `mm/memory.c` za slična razmatranja.

2.3. Opis Linuxova raspoređivača

Osnovna je jedinica kojom upravlja raspoređivač u Linuxu zadatak (engl. *task*), odnosno, jezgrin proces (engl. *kernel process*) s procesnim identifikatorom PID. Takav zadatak ili proces nalazi se u jednonitnom procesu ili je dio višenitnoga procesa. Za podrobniji opis pogledati komentare za funkciju `clone` u poglavlju 3. Linuxov raspoređivač zadatke dijeli u tri disjunktna rasporedbena razreda:

- obični zadatci (engl. *fair*),
- vremenski kritični zadatci (engl. *real time*),
- te zadatci najmanje važnosti (engl. *idle*).

Za početak će se odmah objasniti razred *idle*. Taj se razred, kao što mu naziv govori, odnosi na vrstu zadataka koji se izvode kad procesor nema što izvoditi. Takve su recimo, Windowsove niti koje pune ništicama slobodne dijelove radne memorije [9]. Razred se odnosi na rasporedbenu politiku `SCHED_IDLE` i o takvim zadatcima više ne će biti riječi.

U kodu, konkretno pri definiciji makroa rasporedbenih politika poput ovoga `SCHED_IDLE`, spominje se i nekakav razred ili politika pod nazivom `SCHED_ISO`. Vidi ispis 2.1. Za taj makro piše da je „predbilježen, ali ne i ostvaren“, što vjerojatno znači da će se u budućnosti dodati još neka politika. To ipak vjerojatno ne će biti u bliskoj.

Ispis 2.1: include/linux/sched.h

```
39 #define SCHED_BATCH   3
40 /* SCHED_ISO: reserved but not implemented yet */
41 #define SCHED_IDLE    5
```

Razred običnih zadataka odnosi se na normalne zadatke, za razliku od vremenski kritičnih zadataka. Razred se zasniva na potpuno pravednoj raspodjeli prividnoga vremena (engl. *virtual time*) svojim zadatcima. Zbog toga se i zove potpuno pravednim raspoređivačem (engl. *completely fair scheduler*) [10] ili CFS-om. Ovo je prividno vrijeme prilično složena funkcija prioriteta običnoga zadatka, stvarnoga vremena i opterećenja procesora običnim zadatcima.

Prioritet tih običnih zadataka (engl. *nice*) ide od -20 do 19, pri čem je manje bolje [11]. Dodati je da se zadatci najmanje važnosti ponekad smatraju običnim zadatcima s prioritetom 20, a to će se vidjeti i pri preslikavanju korisničkoga u jezgreni prioritet. Razred obuhvaća dvije politike, a to su `SCHED_BATCH` i `SCHED_NORMAL`. Razlika između tih dviju politika ta je da zadatci politike `SCHED_BATCH` ne mogu prekinuti zadatke politike `SCHED_NORMAL` [12].

U vremenski kritičnom sustavu priča je potpuno drukčija i dijametralno suprotna po predodređenosti (engl. *determinism*). Prioritet vremenski kritičnih zadataka ide od 1 do 99, pri čem je veće bolje. Vremenski kritični razred također obuhvaća dvije politike, a to su `SCHED_FIFO` i `SCHED_RR`. Kao što im nazivi kažu, prvi se zadatci izvode dokle god mogu, odnosno sve dok ne završe izvođenje, blokiraju se ili sami prepuste (engl. *yield*) procesor. To se zove načelom „prvi unutra, prvi van“ (engl. *first-in-first-out*). Drugima se pak po načelu kružne podjele vremena dodjeljuje vremenski odsječak na procesoru (engl. *round robin*). Taj je odsječak podrazumijevano dug 100 milisekunda.

Zapravo je jedina razlika u tima dvjema politikama to što nakon što aktivnomu (engl. *current*) zadatku `SCHED_RR` otkuca vrijeme, oduzima mu se procesor i on se stavlja na kraj reda zadataka, odnosno podreda njegove prioritetne razine. Naravno, osim ako je jedini u toj razini. To se događa u funkciji `task_tick_rt` i tako reći ne čini više od tri retka koda, na što će se pozvati u poglavlju 3. Konkretno, funkcija ne čini ništa ako je zadatak `SCHED_FIFO`, a otkucava i ako je otkucalo vrijeme, ono se poništava i čini se već spomenut postupak. Vidi ispis 2.2.

Ispis 2.2: kernel/sched/rt.c

```
2107   if (p->policy != SCHED_RR)
2108     return;
2109
```

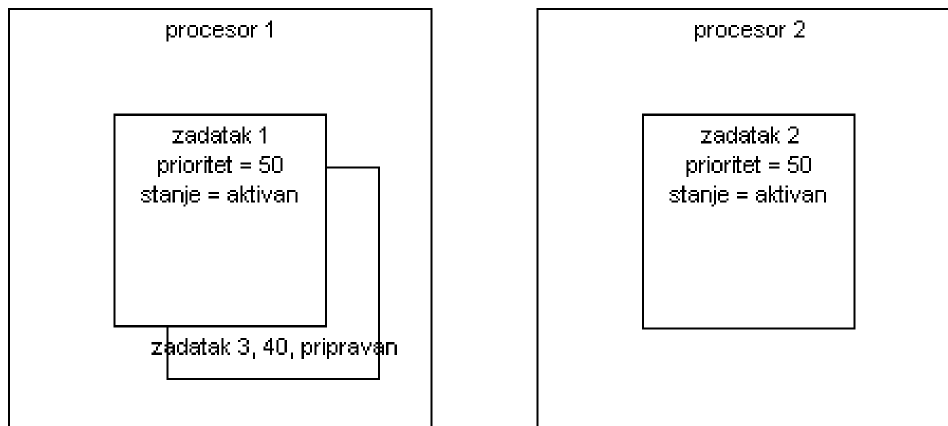
```
2110  if (--p->rt.time_slice)
2111    return;
2112
2113  p->rt.time_slice = DEF_TIMESLICE;
2114
2115  if (p->rt.run_list.prev != p->rt.run_list.next) {
2116    requeue_task_rt(rq, p, 0);
2117    set_tsk_need_resched(p);
2118  }
```

Ispravno shvaćanje skoro nepostojeće razlikosti ovih dviju politika ključno je za shvaćanje zašto se ne će posebno osvrtati na rasporedbenu politiku `SCHED_FIFO`. Ipak, bit će spomenuta u surječju ispitivanja u poglavlju 3.

Vremenski se kritični sustav, kako će se govoriti umjesto nespretne složenice RT-sustav, zasniva na potpuno razlikim pretpostavkama od sustava običnih zadataka. Te se pretpostavke stručno zovu strogim vremenski kritičnim prioritetnim raspoređivanjem širom sustava (engl. *system-wide strict real-time priority scheduling*), na koji će se pozivati kao na SVPRŠS.

Strogo prioritetno raspoređivanje u sustavu znači da u takvu sustavu *N* najprioritetnijih zadataka obvezatno drži *N* procesora, pri čem je *N* ukupan broj procesora, svaki po jedan. Pod procesorom se zapravo misli na najsitniju zrnatost hijerarhije, odnosno na logičku jezgru. Takvo je prioritetno raspoređivanje potpuno normalno u takvim sustavima za rad u stvarnom vremenu gdje treba osigurati predodređenost.

Treba napomenuti da u sustavu s, primjerice, dvama procesorima i dvama zadatcima prioriteta 50 te jednim zadatkom prioriteta 40, svima pripravnima, ovo znači da će na jedan procesor ići jedan zadatak prioriteta 50, a na drugi procesor drugi zadatak prioriteta 50. Vidi sliku 2.1. Iako bi možda u logici `SCHED_RR` imalo smisla staviti oba na isti procesor, za zadatke `SCHED_FIFO` ovo nema smisla jer se jedan nikad ne će izvoditi. Tako da se to ne čini na takav način. Ovo je slaba strana ignoriranju razlikosti dvaju politika.



Slika 2.1: Raspodjela zadataka u opisanom primjeru

Važno je napomenuti da korisnik može potpuno srušiti SVPRŠS postavljanjem čudna procesorskoga afiniteta. Procesorski je afinitet (engl. *processor affinity*) skup procesora na kojima se zadatak smije izvoditi. Govori se o funkciji `sched_setaffinity` [13].

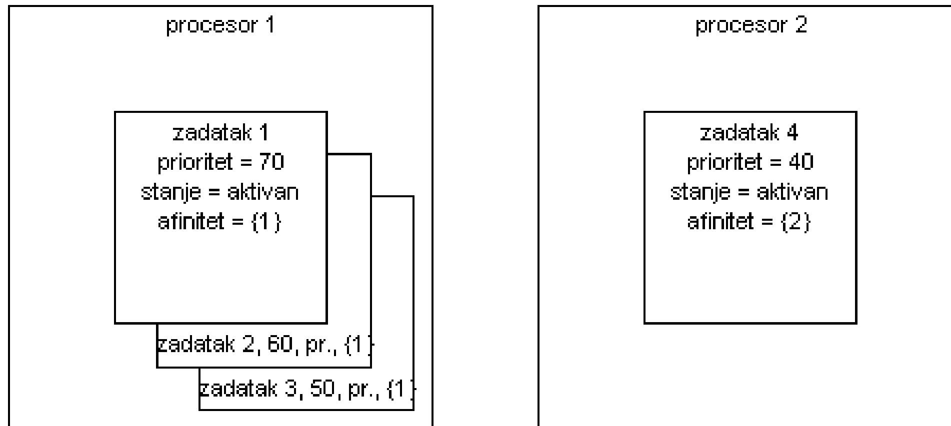
Prije objašnjavanja kako se to može dogoditi napomenut će se da se u cijelom radu, pa tako i u prethodnom primjeru, podrazumijevati da veći broj odgovara većemu vremenski kritičnomu prioritetu. Tako i jest s korisnikove strane gledišta, dok jezgra interno veći vremenski kritični prioritet ostvaruje kao manji, pa o tom treba voditi računa pri gledanju koda raspoređivača i koda ostvarenih izmjena.

Dakle, sustavski pozivi koji postavljaju vremenski kritični prioritet zadatka, poput `sched_setscheduler` [14] ili `sched_setschedparam` [15], gledaju stvari na takav način. Jezgra onda preslikava prioritet običnih zadataka u od 100 do 139, te vremenski kritični u od 98 do 0, s tim da je zadatak *idle* predstavljen stupnjem 140, što može zbunjivati programera, i često zbunjuje.

Ipak, valjda je takvim preslikavanjem jednostavnije ispitivati je li jedan općenit zadatak važniji od drugoga, što uvijek vrijedi za vremenski kritični zadatak, obični zadatak i zadatak *idle*, neovisno o politici. Za jezgreni prioritet vidjeti funkciju `convert_prio` u datoteci `kernel/sched/cpupri.c`, gdje se on preslikava u `cpupri`-prioritet, o kojem u poglavlju 3.

Još je jedna stvar koja će se cijelo vrijeme podrazumijevati da su svi procesori aktivni (engl. *online*). Dakle, sustav se može srušiti, primjerice, postavivši procesorski afinitet tako da tri zadatka *FIFO*, to jest općenito vremenski kritična zadatka, prioriteta 50, 60 i 70 idu na procesor 1, a jedan drugi takav prioriteta 40 na procesor 2. Vidi sliku

2.3. Misli se, naravno, na rušenje načela sustava, a ne na nemogućnost rada sustava, iako se i o tom može govoriti ako se pretpostavlja da sustav radi prema SVPRŠS-u.



Slika 2.2: Onemogućavanje SVPRŠS-a

Na ovakvu će se tako sustavu cijelo vrijeme izvoditi zadatak prioriteta 70 na procesoru 1, a zadatak prioriteta 40 na procesoru 2. Prema SVPRŠS-u umjesto zadatka prioriteta 40 morao bi se izvoditi zadatak prioriteta 50. Rušenje sustava procesorskim afinitetom ipak je otežano razlikošću skupova zadataka koje algoritmi mogu gurati i povlačiti [16], ali se o tom ne će govoriti u ovom radu.

U sustavu u kojem nema zlorabljenja afiniteta jamči se SVPRŠS, ali samo unutar korijenske domene. Korijenska je domena (engl. *root domain*) skup procesora unutar kojega se želi osigurati SVPRŠS. Naime, SVPRŠS u baš cijelom sustavu, ako je sustav stvarno mnogoprocesorski, uzrokuje uska grla provjerom redova svih tih procesora, a to uvođenjem takvih domena više nije slučaj. SVPRŠS je zapanjujuće jednostavno ostvaren algoritmima guranja i povlačenja, o kojima u odjeljku 2.4.

2.4. Algoritmi guranja i povlačenja

Ostvarenje se SVPRŠS-a, dakle, provodi kroz vremenski kritični sustav uravnotežavanja tovara (engl. *load balancing*). To je sustav koji se unutar rasporedbenoga razreda brine o pravilnu iskorištavanju resursa procesorskoga sustava. Odnosno, on sprječava pretovarivanje jednih dijelova procesorskoga sustava dok drugi ništa ne rade.

Sustav običnih zadataka sveudilj brine o tim stvarima i svojim sustavom uravnotežavanja tovara prožima i osnovnu logiku raspoređivača, za što je dovoljno pretražiti

njezinu datoteku nizom „`trigger_load_balance`“. Za to vrijeme, vremenski se kritični sustav osniva na jednostavnu mehanizmu guranja i povlačenja, odnosno algoritmu *push-pull*. Uzevši u obzir stroga ograničenja koja nameće sustav koji se treba izvoditi u stvarnom vremenu, pri čem se misli na SVPRŠS, jednostavnost ovoga mehanizma možda i ne čudi toliko.

Isto su tako zadatci u vremenski kritičnom raspoređivaču jednostavno raspodijeljeni u prioritetne procesorske redove s po 99 razina (engl. *priority queues*). U raspoređivaču običnih zadataka raspodijeljeni su u crveno-crna stabla (engl. *red-black trees*), što zahtijeva i dosta potporne strukture poput `include/linux/rbtree.h`. Radi potpunosti izlaganja spomenut će se da su crveno-crna stabla zapravo najbolja dosad otkrivena stabla za binarno pretraživanje (engl. *binary search trees*). U nastavku se razmatraju samo vremenski kritični zadatci, to jest kad se spomene „zadatak“, to se odnosi na vremenski kritični zadatak, osim ako nije rečeno drukčije.

Preopterećenim se redom zadataka smatra red u kojem postoji najmanje jedan zadatak koji se može po procesorskom afinitetu preseliti na drugi procesor i u kojem postoje najmanje dva pripravna zadatka. Taj jedan preseljiv (engl. *migratable*) zadatak može biti među pripravnima (engl. *running*), ali i ne mora.

Osnovna je zamisao mehanizma guranja-povlačenja da se kad se procesorski red preoptereti pokuša naći neki drugi red na kojem bi se drugi najprioritetniji zadatak s ovoga procesora odmah mogao početi izvoditi. To se zove guranjem zadataka **s procesora**. Sveza „drugi najprioritetniji“, primjerice, u sustavu s dvama zadatcima prioriteta 40 označuje onaj zadatak od tih dvaju koji nije aktivan. Komplementarno guranju postoji i povlačenje zadataka **na procesor**, a osniva se na tom da kad procesorski red shvati da bi možda mogao primiti takav zadatak pokuša ga i povući.

Sustav guranja-povlačenja kao uravnotežavanje tovara vremenski kritičnoga razreda, osim dotičnih dvaju slučajeva, treba podijeliti i u još dva slučaja:

– slučaj dolaska novoga zadatka u sustav,

– te buđenje zadatka koji je bio potpuno odraspoređen (engl. *descheduled*).

Odraspoređen je zadatak onaj koji se više ne nalazi ni na jednom procesoru. Pred kraj će odjeljka biti jasno zašto se ovako dijele slučajevi. Također, ova će dva uskoro postati šest slučajeva.

Spomenut će se da i zadatak može biti i nepotpuno odraspoređen. To je jedan od razloga zašto se buđenje zadatka i potpuna probuđenost zadatka razmatraju odvojeno u kodu raspoređivača, pa se tako obrađuju stanja `task_waking` i `task_woken`.

Prije ulaska u spomenute slučajeve još će se objasniti i to da se prekidanje izvođe-

nja nekoga zadatka kako bi se onaj koji izvodi prekidanje mogao početi izvoditi obično zove istiskivanjem (engl. *preemption*). Istiskiv je raspoređivač (engl. *preemptible scheduler*) onaj koji to dopušta, a istiskiva je jezgra (engl. *preemptible kernel*) jezgra u kojoj se i jezgrene funkcije (engl. *kernel functions*) mogu prekinuti, odnosno istisnuti (engl. *preempt*). Sad će se krenuti na slučajeve.

Guranje zadatka **s procesora** pokreće se u trima slučajevima, a povlačenje isto u trima. Za pronalazak toga u kodu treba potražiti nizove „`push_rt_task`" i „`pull_rt_task`" u datoteci `kernel/sched/rt.c`, što će biti jasnije pri opisu tih datoteka u poglavlju 3. Guranje se pokreće:

1. poslije svake zamjene aktivnoga zadatka;
2. poslije prebacivanja zadatka u vremenski kritični razred, što znači da se pojavio nov vremenski kritični zadatak i tu je guranje ograničeno na jedan zadatak;
3. te poslije buđenja zadatka na procesoru na kojem je trenutačno aktivan drugi vremenski kritični zadatak ako probuđeni može na drugi procesor, a aktivni ili ne može ili je prioritetniji od probuđenoga ili jednako prioritetan njemu.

Ako nije isprva jasno, ovaj posljednji slučaj implicira guranje probuđenoga zadatka, a ne aktivnoga. Algoritam se guranja mora podijeliti u tri podalgoritma, *guranje*, *nalaženje* i *traženje*. Pred kraj će odjeljka biti jasno zašto se to tako čini.

Podalgoritam je *guranja* sljedeći. Ako je red na danom procesoru preopterećen, uzmi njegov najprioritetniji zadatak koji se ne izvodi i *nađi* procesor na koji ćeš ga preseliti.

Podalgoritam je *nalaženja* procesora sljedeći. Do tri puta *traži* procesor na koji ćeš preseliti zadatak i ako je on nađen, a nije onaj na kojem zadatak jest, dok je najprioritetniji zadatak na novom procesoru manje prioritetan od njega, vrati nov procesor.

Podalgoritam je *traženja* procesora sljedeći. Uzmi najmanju razinu prioriteta u sustavu i uzmi kao skup mogućih procesora sve procesore koji izvode zadatke s tom najmanjom razinom prioriteta. To podrazumijeva i razinu „procesor je slobodan". Između tih procesora najprije pokušaj vratiti procesor na kojem je zadatak zadnji put bio. Ako taj nije među mogućima, pokušaj se penjati po rasporedbenim domenama od toga procesora prema korijenu stabla. Na trenutačnoj razini među procesorima pokušaj vratiti procesor na kojem se izvodi sam algoritam, a ako ni taj nije među mogućima, onda bilo koji od procesora. Ako su i domene pretražene, pokušaj vratiti procesor na kojem se izvodi algoritam, a onda bilo koji.

Rasporedbene su domene (engl. *scheduling domains*) logička procesorska hijerarhija i podrazumijevano su preslika fizičke hijerarhije. Povlačenje se pokreće:

1. poslije svake zamjene aktivnoga zadatka ako je novi aktivni manje zadatak prioritetan od staroga;
2. poslije prebacivanja zadatka iz vremenski kritičnoga razreda ako tad više nema takvih na procesoru;
3. te poslije smanjenja prioriteta zadatku ako je zadatak aktivan.

Algoritam je povlačenja sljedeći. Ako ima preopterećenih redova u korijenskoj domeni, idi po svim njezinim procesorima. Uzmi procesorov prvi najprioritetniji zadatak koji se ne izvodi i, ako je on prioritetniji od aktivnoga zadatka na ovom procesoru, povuci ga u ovaj red, a u svakom slučaju nastavi ići po procesorima.

Preostala su još ona dva slučaja. Dakle, ako u sustav dolazi nov zadatak, kao njegov se zadnji procesor postavlja onaj na kojem je njegov roditelj koji je pozvao stvaranje novoga zadatka, a onda se poziva algoritam *traženja* koji je opisan prije.

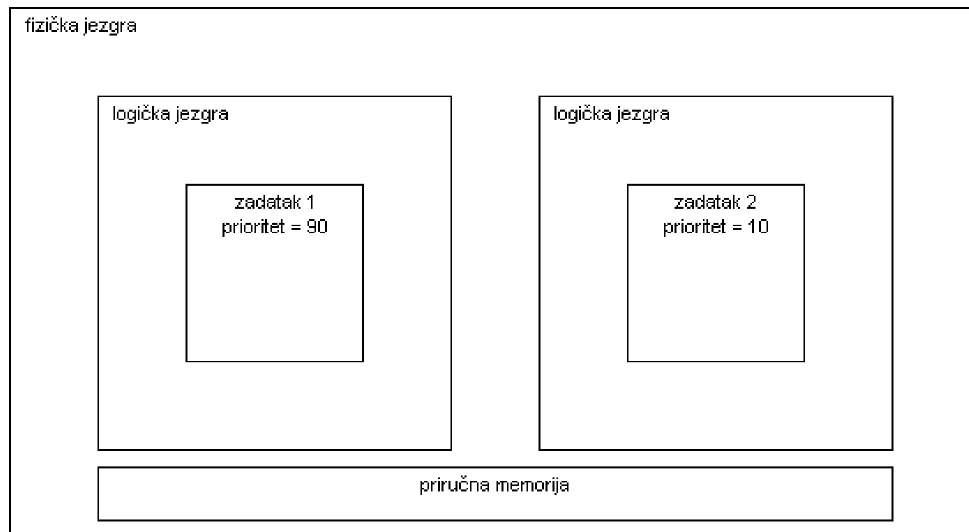
Kod buđenja zadatka koji je bio posve odraspoređen također se poziva algoritam *traženja*. Ipak, on se poziva samo ako je već na njegovu zadnjem procesoru neki vremenski kritični zadatak i probuđeni može ići na drugi, dok aktivni ili ne može ili je prioritetniji od probuđenoga ili jednako prioritetan njemu, slično kao kad se zadatak potpuno probudi. Sad je jasno zašto se je inzistiralo na razdvajanju podalgoritma traženja procesora prilikom njihova opisivanja.

2.5. Nedostatci Linuxova raspoređivača

Linuxov sustav raspoređivanja ima nedostatke, od kojih je već nekoliko pobrojeno. Prvi je nedostatak bio potpuno potpuno neiskorištavanje povezanosti niti, a drugi mogućnost onemogućavanja raspoređivanja prema prioritetu zlorabljenjem afiniteta zadataka prema procesorima.

Još je jedan nedostatak to što algoritam u svojoj jednostavnosti uopće ne uzima u obzir slučajeve ometanja zadataka višega prioriteta zadatcima nižega prioriteta. Ovakav se problem obično naziva problemom inverzije prioriteta (engl. *priority inversion*). Recimo, na hipernitnu procesoru, a pogotovo kad logičke jezgre dijele priručnu memoriju, na jednoj se jezgri recimo izvodi zadatak prioriteta 90, a na drugoj zadatak puno manjega prioriteta, recimo 10. Vidi sliku 2.3. Taj zadatak, primjerice stalnim

pražnjenjem priručne memorije, može ometati ovaj prvi zadatak. To jest, algoritam ne uzima u obzir dinamička svojstva sustava.



Slika 2.3: Inverzija prioriteta

Napomenuti je da je Linux prvenstveno namijenjen običnim poslovima te navedeni nedostatci samo upozoravaju na moguće teškoće njegove uporabe u okružjima za koje on nije optimiziran. Kao što je već rečeno, predloženo grupiranje također rješava prvi spomenuti problem. K tomu se potrudilo naći rješenje koje će jako malo zadirati u sustav, a ujedno ga i organizirati tako da se jednostavno može uključiti u stvaranje sustava i isključiti ga iz njega. Također i rješenje koje, naravno, ne će dalje pogoršati svojstva sustava.

3. Izmjene u svrhu poboljšanja algoritma raspoređivanja

3.1. Zamisli za poboljšanje Linuxova raspoređivača

Budući da je sustav običnih zadataka vrlo zamršen, pri ostvaraju se grupiranja usredotočilo na vremenski kritične zadatke. To ne znači da su se pokušale poboljšati performanse vremenski kritičnih sustava, nego da se zbog toga što je tako bilo puno jednostavnije algoritam uklopio u taj rasporedbeni razred, a onda se taj rasporedbeni razred ispitivao kao općenit algoritam raspoređivanja. Rješenje grupiranja niti u raspoređivaču pronađeno je u nekolicini izmjena algoritama guranja i povlačenja. Točnije, radi se o tri izmjene algoritma guranja i jedne izmjene algoritma povlačenja. Cilj izmjena bio je pokušati grupirati niti istoga procesa, a to je, kao što će se pokazati u poglavlju 5, bilo vrlo uspješno.

Izvorno se namjeravalo uzeti u obzir i povezanosti zadataka općenito poput dijeljene memorije (engl. *shared memory*) između razlikih procesa. Ipak, ubrzo se ustanovilo da Linuxov raspoređivač uopće ne uzima u obzir višenitnost i sve gleda kao generičke zadatke te se usredotočilo na grupiranje niti. Inače, tako i jezgra sve gleda kao „jezgrene procese“, osim kad treba izići iz svih niti istoga procesa kad glavna iziđe, a onda kad treba dostaviti neki signal procesu i slično. Zadatci koji su niti istoga procesa u daljnjem će se tekstu nazivati sunitima, iako na engleskom ne postoji nazivak koji znači isto.

Rješenje problema povezanosti također nije imalo više od nekoliko stotina redaka zbrojivši i kod i konfiguraciju. Za usporedbu, kod cjelokupnoga raspoređivača ima oko dvadeset tisuća redaka, što se može pobrojiti naredbom `wc -l 'find linsched/kernel/sched'`.

Rješenje je ostvarno četirima zamislima, odnosno izmjenama, za algoritme guranja i povlačenja, koje će sad biti izložene. One su popisane redom kojim su ostvarene, iako takav redoslijed možda nije najsretniji. Na redoslijed se može gledati kao redoslijed

od jednostavnije prema složenijoj. Logički gledano, grupiraju se zamisli 1 i 4 te 2 i 3.

1. U algoritmu *traženja* procesora, između pokušaja vraćanja procesora na kojem je zadatak zadnji put bio i pokušaja vraćanja najbližega procesoru na kojem je zadatak zadnji put bio, ubaci i pokušaj vraćanja nekoga procesora na kojem se nalazi neka zadatkova sunit.

2. U algoritmu *traženja* procesora, kad se može vratiti procesor na kojem je zadatak zadnji put bio, prije nego što se vrati ubaci pokušaj vraćanja nekoga procesora na kojem se nalazi zadatkova sunit, ali samo ako se na ovom ne nalazi neka druga sunit.

3. U algoritmu povlačenja, nakon provjere je li zadatak na drugom procesoru većega prioriteta nego trenutačni na ovom procesoru, ubaci provjeru i je li jednako prioritetan i ako jest i ako je ne ovom procesoru neka njegova sunit, a na njegovu nije, također ga preseli ovamo.

4. U algoritmu *traženja* procesora, između pretraživanja domena i vraćanja bilo kojega procesora, ubaci pokušaj pretraživanja domene za sve procesore na kojem se nalaze suniti.

Pokazat će se da te četiri izmjene ne narušavaju svojstva vremenski kritičnoga sustava. Naime, zamisli 2 i 3 zapravo zadatak koji se ne može izvoditi na svojem procesoru sele na neki drugi procesor na kojem se još uvijek ne može izvoditi, ali se tom promjenom postiže bolje grupiranje zadataka. Ako su na novom procesoru samo zadatci kružnih politika, zadatak će se početi i izvoditi, ali se komplementarno može razmišljati i što ako su na izvorišnom samo zadatci kružnih politika.

Glede narušavanja, isto tako, zamislima 1 i 4 samo se zadatak koji će se upravo početi izvoditi na jednom procesoru u skupu najneprioritetnijih stavi na malo pametnije odabran jedan od njih tako da se postigne bolje grupiranje suniti. Za ograničenja je sustava svejedno koji se od mogućih procesora odabere.

Zamisli također na naravan način proizlaze iz koda. Naime, algoritam guranja glasi:

- pokušaj ostati na istom procesoru,
- te pokušaj ostati u blizini.

Zamislima 1 i 4 taj je algoritam komplementarno nadopunjen tako da glasi:

- pokušaj ostati na istom procesoru,

– pokušaj se priključiti skupini,

– pokušaj ostati u blizini,

– te se pokušaj priključiti blizini skupine.

Zamislima 2 i 3 pak po provjeravanju je li nov zadatak prioritetniji od staroga dodano je provjeravanje je li jednako prioritetan. Upravo zato što se ove zamisli ovako lijepo uklapaju u kod olakšano je njihovo možebitno uključivanje jednoga dana, o čem u poglavlju 5.

Zamisli su, dakle, ostvarene u kodu Linuxove jezgre, a onda je taj kod preveden, pokrenut i ispitan u stvarnu okružju, o čem će biti riječi u poglavlju 4 i 5.

3.2. Dodavanje nove rasporedbene politike

Prije konkretna opisa izmjena koda treba reći da se ne mijenja zapravo raspoređivanje vremenski kritičnih zadataka `SCHED_FIFO` i `SCHED_RR`. Umjesto toga dodana je nova rasporedbena politika zasnovana na vremenski kritičnoj politici `SCHED_RR`, a nazvana je, logično, `SCHED_RR2`.

Budući da dvije politike `SCHED_FIFO` i `SCHED_RR` ionako dijele 99.99% koda, odnosno konkretno 2038 redaka od 2052 retka, a `SCHED_RR` i `SCHED_RR2` dijelit će samo malo manje, nije se stvarao cio nov rasporedbeni razred u novoj datoteci. Umjesto toga samo se u ovaj razred uvrstila još jedna politika. Njezino pak dodavanje mijenja malo toga u jezgri, a sad će se iznijeti što točno.

Najprije se u datoteci `include/linux/sched.h` ustoličila nova rasporedbena politika. Ova je datoteka sučelje raspoređivača prema vanjskim programima, a cio je direktorij `include` sučelje jezgre prema vanjskim programima. Ubacili su se redci 42-44, čim se definirao nov makro koji obilježava novu politiku. Vidjeti ispis 3.1. Radi se o globalnom prostoru imena. Za makroe `CONFIG_DA i CONFIG_DA2` vidjeti odjeljak 3.8.

Ispis 3.1: include/linux/sched.h

```
42 #ifdef CONFIG_DA
43 #define SCHED_RR2 6
44 #endif
```

Onda se u datoteci `kernel/sched/sched.h` dodala oznaka da se radi o vremenski kritičnoj politici. Ova je pak datoteka sučelje raspoređivača prema ostatku jezgre, a cio je direktorij ostvaraj sustava raspoređivanja. Ubacili su se redci 52-54

i 56, čim se uvjet ispitivanja radi li se o SCHED_FIFO-u ili o SCHED_RR-u proširio ispitivanjem radi li se o makrou SCHED_RR2. Vidi ispis 3.2. Radi se o funkciji rt_policy, koja određuje je li neka politika vremenski kritična ili nije.

Ispis 3.2: kernel/sched/sched.h

```
52 #ifdef CONFIG_DA
53   if (policy == SCHED_FIFO || policy == SCHED_RR ||

53       policy == SCHED_RR2)
54 #else
55   if (policy == SCHED_FIFO || policy == SCHED_RR)
56 #endif
```

U datoteku kernel/sched/core.c dodalo se omogućivanje izbora nove politike. Ova je datoteka ostvaraj zajedničke logike raspoređivanja. Ubacili su se redci 4051-4053, čim se učinilo zadavanje novoga makroa valjanim. Vidi ispis 3.3. Radi se o funkciji __sched_setscheduler, koja je omot sustavskoga poziva.

Ispis 3.3: kernel/sched/core.c

```
4050     if (policy != SCHED_FIFO && policy != SCHED_RR &&
4051 #ifdef CONFIG_DA
4052         policy != SCHED_RR2 &&
4053 #endif
4054         policy != SCHED_NORMAL && policy != SCHED_BATCH &&
```

U istoj su se datoteci u funkciji sched_get_priority_max dodali redci 4712-4714, čim se je prvim dijelom izjednačio prioritetni raspon s politikom SCHED_RR. Vidi ispis 3.4. Ova funkcija dohvaća najveći prioritet za zadanu politiku.

Ispis 3.4: kernel/sched/core.c

```
4711   case SCHED_RR:
4712 #ifdef CONFIG_DA
4713   case SCHED_RR2:
4714 #endif
```

Također su se u istoj datoteci u funkciji sched_get_priority_min dodali redci 4740-4742, čim se i potpuno izjednačio novi prioritetni raspon. Vidi ispis 3.5. Ova pak funkcija dohvaća najmanji prioritet za zadanu politiku.

Ispis 3.5: kernel/sched/core.c

```
4739   case SCHED_RR:
```

```
4740 #ifdef CONFIG_DA
4741   case SCHED_RR2:
4742 #endif
```

U sljedeću se datoteku, `kernel/sched/rt.c`, najprije dodala oznaka da se radi o kružnoj, a ne politici *FIFO* ili nekoj drugoj. Ubacili su se redci 2148-2150 i 2152, čim je ispitivanje radi li se o politici `SCHED_RR` prošireno ispitivanjem radi li se o novoj politici. Vidjeti ispis 3.6. Radi se o funkciji `get_rr_interval_rt`, koja dohvaća vremenski odsječak za zadanu politiku.

Ispis 3.6: kernel/sched/rt.c

```
2148 #ifdef CONFIG_DA
2149   if (task->policy == SCHED_RR || task->policy == SCHED_RR2)
2150 #else
2151   if (task->policy == SCHED_RR)
2152 #endif
```

Na kraju se je omogućilo otkucavanje sata novoj politici. Ubacili su se redci 2111-2113 i 2115, čim se proširio uvjet ispitivanja radi li se o politici `SCHED_RR` ispitivanjem radi li se o ovoj. Vidi ispis 3.7. Radi se o funkciji `task_tick_rt`, koja se poziva svakim otkucajem raspoređivača.

Ispis 3.7: kernel/sched/rt.c

```
2113 #ifdef CONFIG_DA
2114   if (p->policy != SCHED_RR && p->policy != SCHED_RR2)
2115 #else
2116   if (p->policy != SCHED_RR)
2117 #endif
```

Tako su zaokružene sve promjene unutar sustava raspoređivanja potrebne za uvođenje nove politike `SCHED_RR2`. To u dobro oblikovanu sustavu mora biti dovoljno za ispravan rad i ostalih podsustava. Ako pak neki slabo oblikovan modul umjesto uporabe javne funkcije `rt_policy` krene stvarno provjeravati o kojoj se točno politici radi, takav modul ne će ispravno raditi. Takvi se dijelovi mogu pronaći pretraživanjem koda makroima.

3.3. Procesi i niti u Linuxu

Prije opisivanja sljedećega dijela izmjena koda potrebno je shvatiti kako Linux shvaća procese i niti. To će biti ključno za razumijevanje odjeljka 3.4. Zadatci se

na Linuxu općenito stvaraju sustavskim pozivom `clone` [17]. On se općenito može shvatiti kao poseban oblik funkcije `fork` [18] koji omogućuje da, umjesto da roditelj i dijete ne dijele ništa, dijele točno određene dijelove.

Funkcija `fork`, naravno, stvara nov proces, i zajedno s funkcijom `pthread_create` za stvaranje nove niti čini školsko shvaćanje višenitnosti [19]. Prema tom shvaćanju, procesi i niti nešto su posve drukčije. U Linuxu su, s druge strane, taj sustavski poziv `fork` i taj knjižnični poziv `pthread_create` samo omotači pozivu funkcije `clone`, koji stvara nov jezgreni proces. Jezgreni je proces, ponovno, zapravo istoznačnica za zadatak.

Poziv će funkcije `fork` funkciji `clone` reći da novi zadatak s onim koji je funkciju pozvao dijeli najmanje moguće, a poziv funkcije `pthread_create` da dijeli najviše moguće u skladu sa semantikom niti. Izravnim pozivom funkcije `clone` može se pak stvoriti nešto između. Tako je moguće odabrati, primjerice, da dva jezgrena procesa dijele ili ne dijele adresni prostor makroom `CLONE_VM`, datotečne opisnike makroom `CLONE_FS`, rukovače signalima makroom `CLONE_SIGHAND` i slično.

Niti su u Linux dodane u inačici jezgre 2.4 samo kao zadovoljavanje forme propisane standardom POSIX 1.c [20], gdje je bilo propisano da postoje funkcije tipa `pthread_create` i što one stvaraju. Linux pak ne shvaća procese na isti način kao i Unix [21]. To zadovoljavanje forme pokazuje i stalno miješanje naziva *tid*, *tgid*, *pid*, *spid*, *ppid* i sličnih [22]. Forma je zadovoljena tako što je stvaranje niti zakrpano prosljeđivanjem određenih parametara sustavskomu pozivu `clone`. Podrobnije pod opisom parametra `CLONE_THREAD` u `man`-stranici funkcije `clone`. To je, naime, ključni parametar za zakrpani poziv.

Kad se govori o sustavskim pozivima (engl. *system calls*) i o knjižničnim pozivima (engl. *library calls*), misli se na odjeljke 2 i 3 naredbe `man` onako kako su definirani na Linuxu [23]. Odnosno, upitima `man 2 funkcija` i `man 3 funkcija`. Ne misli se na sustavski poziv u užem smislu, što je slanje identifikatora sustavskoga poziva funkciji `syscall` [24]. To se, naime, skoro nikad ne radi jer su skoro uvijek ostvareni istoimeni omotači za te pozive, što su sustavski pozivi u širem smislu. Ako oni nisu ostvareni, vjerojatno se radi o nekoj izvanstandardnoj funkciji poput `gettid` [25].

Sustavski su pozivi u širem smislu zapravo također nekakvi knjižnični pozivi istoga naziva kao i sustavski. Ti pozivi obično nemaju `man`-stranicu, osim ako nemaju drukčiju semantiku. Takav poziv, kao i ostali knjižnični pozivi, jezgrene funkcije poziva upravo preko funkcije `syscall`.

Na arhitekturi x86 funkcija `syscall` znači stavljanje argumenata jezgrene funkcije i njezina identifikatora na korisnički stog i izazivanje prekida `0x80` [26]. Taj

prekid na toj arhitekturi ulazi u jezgru, gdje se procesor prebacuje u jezgreni način rada s vlastitim stogom. Nakon završetka funkcije iziđe se obrnutim redoslijedom, a na korisničkom stogu ostaje povratna vrijednost funkcije. Korisnikovu se kodu to upravo čini kao poziv obične funkcije. Ostale arhitekture rade na sličan način.

Govoreći o omatanju funkcija misli se na općenit slučaj. Ovisno o ostvaraju knjižnice i arhitekturi sustava među omotačem i omotom može biti još funkcija. Knjižnica određuje preslikavanje između knjižničnoga poziva i sustavskoga poziva, a arhitektura preslikavanje između sustavskoga poziva i jezgrene funkcije. Konkretno se ovdje radilo o knjižnici `eglibc` i arhitekturi x86, ali se sad ne će popisivati točan redoslijed funkcija.

3.4. Čuvanje popisa svih procesa

Još jedna važna promjena koda koja ide s izmjenama algoritma čuvanje je popisa svih procesa u memoriji. Svaki proces u popisu onda ima popis svojih suniti. Naime, raspoređivač baš nigdje ne drži popis svih zadataka. Postoji mogućnost dobavljanja suniti za neku nit, ali je ovdje procijenjeno da će se ići s vlastitom strukturom umjesto ovisnosti o takvoj. Naime, u jezgri, karikirano govoreći, svaka podatkovna struktura ima najmanje tri objekta za uzajamno isključivanje.

Kao primjer, u podatkovnoj strukturi `task_struct`, koja predstavlja zadatak u raspoređivaču, pretraga koda nizom „`lock`“ daje ni više ni manje nego 32 rezultata. Struktura je određena u datoteci `include/linux/sched.h` i također predstavlja jezgreni proces za ostatak jezgre.

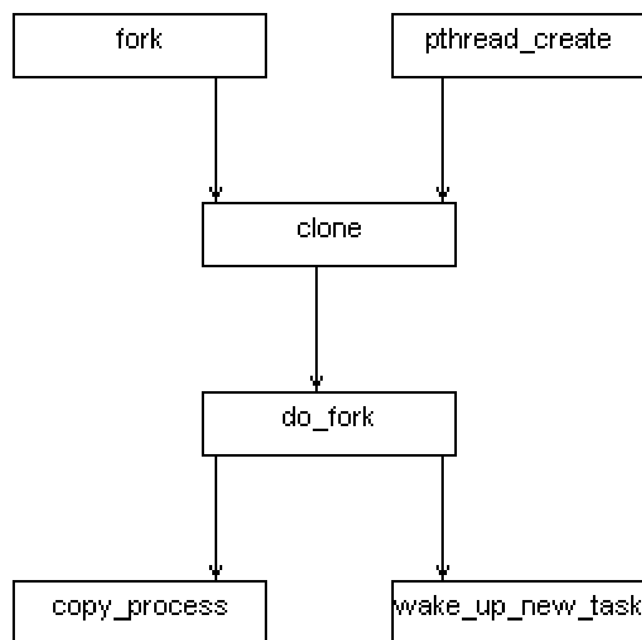
Zadatci se inače povezuju u redove (engl. *queues*). Postoje redovi pripravnih zadataka, redovi odgođenih zadataka, redovi blokiranih zadataka i slično, ali nigdje nema popisa svih. To je posebno nezgodno ako zadatci koje se razmatra nisu dijelom istoga procesa.

Spomenuta se nova struktura ažurira samo u trenutku stvaranja i u trenutku uništavanja zadatka u sustavu. To čini nekoliko desetaka instrukcija po zadatkovu cijelu životnom vijeku. Ovdje se ne misli na stvaranje i uništavanje u procesorskom redu zadataka, što zapravo i nije stvaranje i uništavanje nego povezivanje u popis i razvezivanje.

Naprotiv, misli se na globalno stvaranje i uništavanje, odnosno, kao što će se pokazati, funkcije `wake_up_new_task` i `do_exit`. Na procesoru koji izvodi nekoliko milijarda naredaba u sekundi, poput procesora na kojem se je ispitivalo, vidi poglavlje 4, tih nekoliko desetaka instrukcija stvarno nisu teškoća. Algoritam je složenosti *O(n)*,

pri čem je *n* broj procesa u sustavu.

Sustavski je poziv `clone` omotač (engl. *wrapper*) za jezgrenu funkciju `do_fork` u datoteci `kernel/fork.c`. Ona je opet čisti omotač za jezgrenu funkciju `copy_process` iz iste datoteke iako `do_fork` na kraju mora probuditi zadatak. Funkcija `do_fork` za to buđenje poziva rasporedbenu funkciju `wake_up_new_task` iz datoteke `kernel/sched/core.c`. Vidi sliku 3.1. Ta funkcija ubacuje zadatak u sustav raspoređivanja. Zato se u tu funkciju ubacio kod koji ažurira novu strukturu tako da se doda nov proces, odnosno nova nit ako se radi o novoj niti nekoga procesa, u popis. Dodani su redci 1790-1794 i 1798-1816, u što se ne će ulaziti.

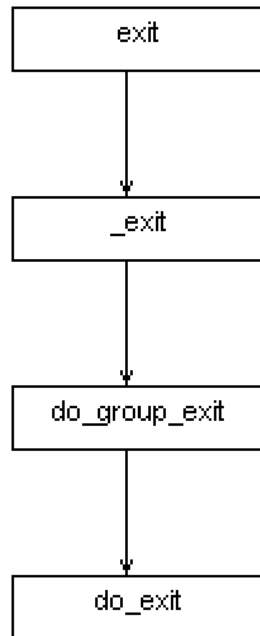


Slika 3.1: Stvaranje zadatka

Ažuriranje se moglo obaviti i u jezgrenoj funkciji `do_fork`. Ipak, budući da se datoteka `kernel/sched/core.c` ionako morala promijeniti, i to već dodavanjem politike `SCHED_RR2`, ušteđena je promjena cijele jedne datoteke. K tomu će se u poglavlju 4 pokazati da se datoteka `kernel/fork.c` mijenjala za rad s LinSchedom, pa se nije htjelo miješati dvije vrste mijenjanja datoteke.

Na završetku svojega izvođenja svaki zadatak u sustavu prolazi kroz knjižnični poziv `exit`, bilo da ga pozove ručno, bilo da se pozove kroz potporni sustav za pokretanje programa u C-u (engl. *C runtime*) [27]. Ta funkcija omata sustavski poziv `_exit`. `_exit` omata jezgrenu funkciju `do_group_exit` u datoteci `kernel/exit.c`.

Ova funkcija inače obavlja u odjeljku 3.1 spomenuto uklanjanje suniti pri izlasku glavne niti. Ona onda omata funkciju `do_exit` u istoj datoteci. Vidi sliku 3.2. Tako je u funkciju `do_exit` dodan kod koji čini suprotnu stvar on one u funkciji `wake_up_new_task`. Algoritam je složenosti *O(1)*. Dodani su redci 900-902 i 1069-1078.



Slika 3.2: Uništavanje zadatka

Konačno se u datoteci `kernel/sched/core.c` redcima 88-93 dodala deklaracija popisa procesa i njegov mehanizam zaključavanja. Vidi ispis 3.8.

Ispis 3.8: kernel/sched/core.c

```
88 #ifdef CONFIG_DA2
89 DEFINE_RAW_SPINLOCK(proc);
90
91 LIST_HEAD(procesi);
92
93 #endif
```

O popisima i zaključavanjima u odjeljcima 3.5 i 3.6. Slično se u datoteci `kernel/sched/rt.c` redcima 10-15 dodala vanjska deklaracija toga dvoga mehanizmom *extern*, jer se u toj datoteci rabe, kako je prije opisano. Slično i s datotekom `kernel/exit.c` i redcima 61-66. Sama je struktura deklarirana u datoteci

`include/linux/sched.h` redcima 74-81, gdje se vidi da zauzima 20 bajtova po jednom procesu. Vidi ispis 3.9.

Ispis 3.9: include/linux/sched.h

```
74 #ifdef CONFIG_DA2
75 struct proces {
76   pid_t tgid;
77   struct list_head niti;
78   struct list_head sljedeci;
79 };
80
81 #endif
```

Dakle, iako se možda tako čini, dodavanje nove strukture podataka ne predstavlja posebno dodatno zauzimanje memorije. Pogotovo na ispitnom sustavu, koji je ima i više nego dovoljno. Konačan argument za uporabu posebne strukture taj je da je sustav već bio posve gotov kad se ustanovilo da se traženje suniti moglo izvesti i drukčije. Ipak, da se išlo s ispitivanjem dijeljene memorije, ovaj bi popis još uvijek trebao.

3.5. Povezani popisi u Linuxovoj jezgri

Prije pokazivanja konkretnih izmjena treba također razumjeti i još dvije stvari iz jezgrenoga programiranja. One su povezane s popisivanjem podataka i njihovim zaključivanjem. Krenut će se s popisima (engl. *lists*). Popisi su u Linuxovoj jezgri ostvareni ne kao jednostruko, nego kao dvostruko povezani i ne kao linearni, nego kao kružni popisi (engl. *doubly linked circular list*) [28].

To jest, svaki čvor u popisu ima pokazivače na prethodni i sljedeći član. Prvomu je članu popisa prethodni član glava, a zadnjemu sljedeći član isto glava. U glavi su pokazivači usmjereni komplementarno tomu. Ako je popis prazan, glava je dvostruko usmjerena na samu sebe. Popis ovako povezan čini dvostruki lanac, pri čem je glava jedna od njegovih karika.

Popisi se definiraju i inicijaliziraju makro-funkcijom `LIST_HEAD`, inicijaliziraju makro-funkcijom `INIT_LIST_HEAD`, a u njih se dodaje funkcijom `list_add_entry`. Iz njih se uklanja funkcijom `list_del_entry`, a po njima se iterira funkcijom `for_each_list_entry`. Sve se ove funkcije odnose na najjednostavniji slučaj uporabe popisa. Više u `include/linux/list.h`.

3.6. Zaključavanje struktura u Linuxovoj jezgri

Drugi je važan koncept jezgrenoga programiranja u ovom slučaju zaključavanje podatkovnih struktura. Osnovni je mehanizam za to kružno zaključavanje (engl. *spinlock*), a to je sustav u kojem algoritmi pokušavaju zaključati, odnosno otključati, strukturu dok god ne uspiju [29]. Struktura se pritom zaključava nekim ispravnim algoritmom uzajamnoga isključivanja (engl. *mutual exclusion*) više niti. Ovdje se misli na algoritme poput Lamportova protokola.

Takvo je zaključavanje primjereno samo podatkovnim strukturama koje se zaključavaju na kratko vrijeme. To ovdje i jest slučaj, posebno za pretraživanje strukture. Objekt za uzajamno isključivanje (engl. *mutex*) za kružno zaključavanje definira se makro-funkcijom `DEFINE_RAW_SPINLOCK`, zaključava funkcijom `raw_spin_lock_irqsave` i otključava funkcijom `raw_spin_lock_irqrestore`. Postoji još vrsta *spinlockova*, ali su ovi najjednostavniji koji ne stvaraju teškoće u istiskivu sustavu.

Također postoji i još funkcija za zaključavanje i otključavanje, ali su ove najosnovniji slučaj u kojem se rukuje stanjem blokiranja prekida. U tom se najosnovnijem slučaju prekidi pri zaključavanju blokiraju, a pri otključavanju se odblokiraju ako su prema spremljenom stanju prije bili odblokirani. O kružnom zaključavanju više u datoteci `include/linux/spinlock.h`.

3.7. Ostvaraj izmjena

Konačno će se krenuti na izmijenjeni algoritam. Za njegov su ostvaraj promijenjene dvije datoteke, `kernel/sched/cpupri.c` i `kernel/sched/rt.c`. O potonjoj je datoteci već bilo govora, a sad će se nešto reći o prvoj, odnosno infrastrukturi `cpupri`.

`Cpupri` je skraćeno od „procesorski prioritet" (engl. *cpu priority*), a taj skup struktura čuva apstrahirano stanje procesorskih redova. To jest, on omogućava brzo nalaženje procesora koji izvode zadatke između kojih je najprioritetniji zadatak prioriteta x, a također i brzo nalaženje prioriteta najprioritetnijega zadatka na procesoru y [30]. Prioritet najprioritetnijega zadatka na procesoru smatra se tim procesorskim prioritetom. U funkciji `cpupri_find` dodani su redci 74-76 i 78. Vidi ispis 3.10. Ta se funkcija poziva kod treba pronaći moguće procesore za guranje zadatka u algoritmu *traženja* i promijenjena je da umjesto da ide po prioritetima do isključivo prioriteta zadanoga zadatka tako da ide uključivo.

Ispis 3.10: kernel/sched/cpupri.c

```
74 #ifdef CONFIG_DA2
75   for (idx = 0; idx <= task_pri; idx++) {
76 #else
77   for (idx = 0; idx < task_pri; idx++) {
78 #endif
```

Ova je promjena učinjena zato da zamisli 2 i 3 uopće mogu raditi. To nema nikakvih dalekosežnih posljedica, budući da uvjet za prestanak pretraživanja prioritetnih razina samo služi za brz izlazak ako traženje procesora više nema smisla. Budući da se ovdje promijenilo stanje kad traženje prestane imati smisla, to je promijenjeno. Iako, dakle, mijenjanje semantike funkcije ne utječe na njezina tri trenutačna poziva, moglo bi utjecati na neke buduće pozive. Zato bi se to u sljedećoj inačici ovoga sustava možda trebalo popraviti. Ostvareni je način jednostavno bio puno manje zadirljiv (engl. *invasive*) od definiranja funkcije `cpupri_find2`.

Dakle, mijenjanje datoteke `kernel/sched/cpupri.c` bilo je samo pomoć za ostatak. Kao pomoć za zamisli mogu se istaknuti i neki dijelovi datoteke `kernel/sched/rt.c`. Vidi tablicu 3.1.

Tablica 3.1: Potpora za uklapanje zamisli

Dio	Redci	Zamisao
1	1250-1252, 1254	2
2	1447-1451	1, 2, 4
3	1601-1603, 1605	2
4	1628-1630, 1632	2
5	1767-1771	4
6	1789-1791, 1793	4

Ti se redci ne će popisivati. Konkretno, prvi dio rješava mogući problem s postavljanjem zastavica opisanim nekoliko rečenica dalje, drugi i peti deklariraju potrebne varijable za zamisli, treći i četvrti provjeravaju je li ta zastavica postavljena za prilagodbu algoritma *nalaženja*, a šesti u algoritmu *guranja* uzima u obzir i zadatke jednakoga prioriteta. Konačno se dolazi do izmjena. Izmjene su također ostvarene u datoteci `kernel/sched/rt.c`. Vidi tablicu 3.2.

Tablica 3.2: Ostvarene izmjene

Zamisao	Redci	Funkcija
1	1494-1506	find_lowest_rq
2	1472-1490, 1492	find_lowest_rq
3	1843-1885	pull_rt_task
4	1537-1571	find_lowest_rq

Funkcija find_lowest_rq ostvaruje algoritam *traženja* reda, a pull_rt_task algoritam povlačenja zadatka. Inače funkcija find_lock_lowest_rq ostvaruje *nalaženje*, a push_rt_task *guranje*. Izmjene su, kao što se će se vidjeti, ostvarene samo i jedino za politiku `SCHED_RR2`, a oslanjaju se na potpornu strukturu opisanu u odjeljku 3.4.

Budući da se politike `SCHED_FIFO` i `SCHED_RR`, kao što je opisano u poglavlju 2, razlikuju u samo desetak redaka koda, moglo se ostvariti i `SCHED_FIFO2`. Postavlja se pitanje smislenosti toga, budući da je osnovna zamisao bila pokušavanje stavljanja bliskih niti što bliže jednu drugoj. To u sustavu gdje se stalno izvode isti zadatci, odnosno, stalno se izvodi N zadataka na N procesora, ne bi puno mijenjalo. Vidi i posljednje ispitivanje u poglavlju 5. Drugo bi bilo da ti zadatci stalno prepuštaju procesor ili idu spavati i slično, ali se tad postavlja pitanje koja je uopće razlika između takva sustava i sustava sa zadatcima ograničenima odsječkom koji se normalno izvode.

Opisivanje koda Linuxova vremenski kritičnoga raspoređivača daleko bi predugo trajalo i ne će se iznositi, ali će se zato podrobno opisati svaka od izmjena. Misli se na opisivanje konkretnoga koda.

Prva izmjena čini sljedeće. Vidi ispis 3.11. Ako je zadatak `SCHED_RR2`, zaključaj strukturu i prođi po njegovim sunitima. Ako je neka od njih na nekom drugom procesoru među mogućim procesorima, otključaj strukturu i vrati taj procesor. Na kraju otključaj strukturu.

Ispis 3.11: kernel/sched/rt.c

```
1494 #ifdef CONFIG_DA2
1495   if (task->policy == SCHED_RR2) {
1496     raw_spin_lock_irqsave(&proc, z);
1497     list_for_each_entry(task2, &task->tg->niti, sljedeca) {
1498       cpu2 = task_cpu(task2);
1499       if (cpu2 != cpu &&

1499           cpumask_test_cpu(cpu2, lowest_mask) != 0) {
1500         raw_spin_unlock_irqrestore(&proc, z);
```

```
1501          return cpu2;
1502        }
1503      }
1504      raw_spin_unlock_irqrestore(&proc, z);
1505    }
1506 #endif
```

Druga izmjena čini sljedeće. Vidi ispis 3.12. Ako zadatak može ostati na procesoru na kojem je bio, prije nego što vratiš nepromijenjen procesor, ako je zadatak SCHED_RR2 zaključaj strukturu i prođi po njegovim sunitima. Ako je na njegovu procesoru jedna od suniti, otključaj strukturu i vrati nepromijenjen procesor. Ako je jedna od suniti na nekom drugom procesoru koji pripada mogućima, uzmi ga kao privremen izbor i postavi zastavicu dizanjem najgornjega bita povratne vrijednosti, što će algoritmu *traženja* signalizirati da treba gurnuti zadatak iako je zadatak istoga prioriteta kao nov procesor.

Ispis 3.12: kernel/sched/rt.c

```
1472 #ifdef CONFIG_DA2
1473    {
1474      if (task->policy == SCHED_RR2) {
1475        raw_spin_lock_irqsave(&proc, z);
1476        list_for_each_entry(task2, &task->tg->niti, sljedeca) {
1477          cpu2 = task_cpu(task2);
1478          if (task2 != task && cpu2 == cpu) {
1479            raw_spin_unlock_irqrestore(&proc, z);
1480            return cpu;
1481          }
1482          if (task2 != task &&

1482              cpumask_test_cpu(cpu2, lowest_mask) != 0) {
1483            odgovor = cpu2 | INT_MIN;
1484          }
1485        }
1486        raw_spin_unlock_irqrestore(&proc, z);
1487      }
1488      return odgovor;
1489    }
1490 #else
1491      return cpu;
```

```
1492 #endif
```

Ovo je postavljanje bitova nužno jer se ne može na neki drugi način, a jednostavno, označiti da se zadatak treba preseliti iako ne mora. Na kraju otključaj strukturu i vrati trenutačni izbor.

Treća izmjena čini sljedeće. Vidi ispis 3.13. Ako je zadatak `SCHED_RR2` i jednakoga je prioriteta kao najprioritetniji na ovom procesoru, prođi po njegovim sunitima. Ako se neka od njih nalazi na njegovu procesoru, prekida se traženje i kaže se da nema povlačenja. Ako se neka nalazi na ovom procesoru, trenutačni je izbor da ima povlačenja. Na kraju se otključa struktura i ako je rečeno da se može povući, kopija koda od nekoliko redaka gore. Kopiran kod ispisuje upozorenje ako je zadatak aktivan na svojem procesoru ili ako spava izvan svoga reda. Onda provjeri na nije slučajno prioritetniji od svojega aktivnoga, što znači da će ga uskoro prekinuti. Na kraju se stavi povratna vrijednost 1 i prebaci se na ovaj procesor.

Ispis 3.13: kernel/sched/rt.c

```
1843 #ifdef CONFIG_DA2
1844     else if (p != NULL && p->policy == SCHED_RR2 &&

1844         p->prio == this_rq->rt.highest_prio.curr) {
1845       raw_spin_lock_irqsave(&proc, z);
1846       list_for_each_entry(p2, &p->tg->niti, sljedeca) {
1847         cpu2 = task_cpu(p2);
1848         if (p2 != p && cpu2 == cpu) {
1849           moze = 0;
1850           break;
1851         }
1852         if (p2 != p && cpu2 == this_cpu) {
1853           moze = 1;
1854         }
1855       }
1856       raw_spin_unlock_irqrestore(&proc, z);
1857       if (moze != 0) {/*<cp>!!!*/
```

(...)

```
1883       }/*</cp>!!!*/
1884     }
1885 #endif
```

Četvrta izmjena čini sljedeće. Vidi ispis 3.14. Ako je zadatak SCHED_RR2, zaključaj i prođi po njegovim sunitima. Za sunit idi po rasporedbenim domenama njezina procesora i za svaku domenu kopija koda od nekoliko redaka gore. Na kraju otključaj strukturu. Kopiran kod provjerava može li se domena buditi, a onda pogleda pripada li trenutačni procesor i mogućima i domeni. Ako jest, otključava se i vraća njega. Ako nije, otključava se i vraća bilo koji koji pripada i mogućima i domeni, ako ima takvih.

Ispis 3.14: kernel/sched/rt.c

```
1537 #ifdef CONFIG_DA2
1538   if (task->policy == SCHED_RR2) {
1539     raw_spin_lock_irqsave(&proc, z);
1540     list_for_each_entry(task2, &task->tg->niti, sljedeca) {
1541       if (task2 != task) {
1542         cpu2 = task_cpu(task2);
1543         for_each_domain(cpu2, sd) {/*<cp!!!>*/
```

(...)

```
1566         }/*</cp!!!>*/
1567       }
1568     }
1569     raw_spin_unlock_irqrestore(&proc, z);
1570   }
1571 #endif
```

Spomenute provjere nalazi li se već sunit na istom procesoru sprječavaju pojavu takozvanoga dobacivanja (engl. *ping-pong*) zadatcima. Naime, u slučaju da već postoji po jedna sunit na dvama procesorima treća bi se sunit mogla stalno seliti s jednoga na drugi. Stalno seljenje zadataka naziva se dobacivanjem.

3.8. Zamatanje promijenjena koda u makroe

Sve su izmjene u kodu u odnosu na početni kod zamotane (engl. *wrapped*) u odgovarajuće makroe preko mehanizama `#ifdef` i `#ifdef-#else`. To su makroi `CONFIG_DA`, koji uključuje novu politiku `SCHED_RR2` u jezgru, i `CONFIG_DA2`, koji uključuje i izmjene nove politike, pa treba još i objasniti takav način izmjena koda.

Pri prevođenju koda glavnu ulogu ima datoteka `.config` u vršnom direktoriju koda. O njoj će više riječi biti i u poglavlju 4. Datoteku ne treba miješati s datotekom

.config.old, koja je automatska pričuvna preslika te datoteke koja se stvara pri prevođenju, a nalazi se na istom mjestu.

Datoteka .config drži konfiguraciju jezgre, a zapravo se sastoji od popisa svih omogućenih konfiguracijskih makroa u sustavu. Konfiguracijski su makroi svi makroi koji počinju nizom „CONFIG_“. Ti se makroi definiraju po razlikim direktorijima izvornoga koda za trenutačni direktorij unutar datoteka koje se zovu Kconfig ili Kconfig.{ovaj_dio, taj_dio, onaj_dio}. Tako se glavni konfiguracijski makroi nalaze u datoteci init/Kconfig. Usput budi rečeno da direktorij init okuplja kod za inicijalizaciju sustava.

Primjer takva makroa koji se stalno vuče kroz sustav makro je CONFIG_SMP, koji određuje prevodi li se sustav za jednoprocesorske ili višeprocesorske sustave. Makroi se mogu definirati kao, na primjer, obvezno uključeni ako je uključen neki drugi makro, obvezno isključeni ako je isključen neki drugi makro, podrazumijevano uključeni, podrazumijevano isključeni i slično [31]. Ipak se nije išlo učiti skladnju, iako CONFIG_DA2 ne može bez CONFIG_DA-a.

Umjesto toga se je pri definiciji makroa vodilo prema makrou CONFIG_RT_GROUP_SCHED. Taj makro kontrolira skupno raspoređivanje za vremenski kritične zadatke, o čem će biti riječi u poglavlju 4. Za taj je makro procijenjeno da je po funkciji jedan od najbližih novim makroima, tako da je njegova definicija kopirana s promijenjenim nazivom. Samo da se napomene da se konfiguracijski makroi definiraju nizom „config XYZ“, što označava makro „CONFIG_XYZ“, pa **zato** svi moraju tako počinjati.

Po definiranju makroa treba ga i omogućiti. To se najjednostavnije čini dodavanjem retka CONFIG_XYZ=y unutar datoteke .config. I ovdje se vodilo makroom CONFIG_RT_GROUP_SCHED, pa su makroi CONFIG_DA i CONFIG_DA2 ispod njega, kao i u datoteci init/Kconfig.

Datoteka .config ne dolazi s izvornim kodom sustava i najčešće se kopira trenutačna konfiguracija sustava. Ta se konfiguracija nalazi u datoteci /boot/config-inačica, što je ovdje /boot/config-3.2.0.45. Ta se konfiguracija uredi, a to se može i preko tekstualnoga sučelja naredbom make config, kao i preko grafičkoga sučelja [32], pa se ostavi u vršnom direktoriju. Pri prvom prevođenju za sve će nejasne konfiguracijske parametre program make pitati korisnika što želi prije nego se krene s prevođenjem. To se izbjeglo obavljanjem toga jednom i onda kopiranjem uvijek iste datoteke.

Prilaganjem datoteka .config i init/Kconfig zaokružuje se prilaganje svih važnih datoteka za rad, a najvažnije svih izmijenjenih. Zapravo, izuzevši nekoliko

datoteka iz poglavlja 4, onda ispitivanja o kojima će biti riječi u poglavlju 5, i dvije skripte za ljusku Bash. Prva je skripta datoteka `skr.sh`, koja je služila za stvaranje pričuvne kopije datoteka u radu na vanjsko računalo. Ona može poslužiti kao neizravan opis hijerarhije priloženih datoteka. Druga je skripta datoteka `skr2.sh`, koja je služila za uspostavljanje okoline pri preformatiranju diska, što se često činilo zbog razloga opisanih u poglavlju 4. Ona može pak služiti za jednostavno ispitivanje svega napisanoga u radu.

Tako kopiranjem datoteka s CD-a i sitnim izmjenama potonje skripte svatko tko proučava ovaj rad može i sam pokrenuti kod ako želi i ispitati rezultate. Glede izmjena, također treba spomenuti da u ovoj inačici skripta nije posve autonomna i da u nju treba tijekom rada s vremena na vrijeme nešto upisati. Tako da će netko tko ju pokreće vjerojatno željeti i to promijeniti.

4. Opis ispitne okoline

4.1. Prva inačica ispitne okoline

Ispitivanje se provelo na stvarnu računalnom sustavu. O konkretnim će se rezultatima govoriti u poglavlju 5, a u ovom će se prije toga opisati ispitna okolina.

Dugo se vremena tražilo odgovarajuće okružje za ispitivanje izmijenjenoga sustava. Za prvu se ruku upogonio simulator sustava raspoređivanja zvan LinSched, skraćeno od Linux Scheduler [33]. LinSched simulira Linuxov raspoređivač u korisničkom okružju, a emulira proizvoljnu procesorsku topologiju. Pod topologijom se misli na preslikavanje logičkih jezgara u fizičke jezgre, fizičkih jezgara u cijele procesore i procesora u čvorove NUMA, uz definiranje „udaljenosti" između čvorova.

Raspoređivač se prevodi kao izoliran dio jezgre i izvodi kao običan program. Uz njega se prevede i dio jezgre, ali je to prvenstveno LinSchedova infrastruktura u direktorijima `tools/linsched` i `arch/linsched`. Premrežene su ovisnosti raspoređivača i ostatka jezgre većinom razriješene nadomjestnim funkcijama (engl. *dummy functions*) koje obično samo vraćaju ništicu ili jedinicu. Takve se funkcije onda povežu umjesto ugrađenih jezgrinih.

Primjerice, ako raspoređivač treba provjeriti može li trenutačni korisnik postaviti zadanu rasporedbenu politiku, postoji jezgrena funkcija `security_task_setscheduler`. Također postoji i nadomjestna funkcija istoga naziva u datoteci `tools/linsched/stubs/sched.c`, koja bi se trebala povezati umjesto jezgrene. To je i logično, budući da vođenje računa o korisnicima sustava nema veze sa sustavom raspoređivanja. Nadomjestna funkcija vraća ništicu, odnosno, dopušta postavljanje politike u svakom slučaju. U odjeljku će se 4.10 pokazati da konkretno baš ta funkcija nije dobro zakrpana.

LinSched inače nema čak niti službenu internetsku stranicu i najslužbenije bi podatke imao njegov e-poštanski popis Linuxove jezgre (engl. *Linux kernel mailing list*) [34], koji će se još spomenuti u odjeljku 4.10. Tako za LinSched skoro pa nema ni dokumentacije, pa je njegova uporaba bila zamršenija nego što je trebala biti.

Da bi se ustanovilo što se događa, najčešće se moralo pretraživati cio njegov izvorni kod oruđem `grep` [35], iako se je to činilo i za kod Linuxove jezgre. Naredba `grep -EI '\bfunkcija\b' `find linsched/tools/linsched`` pokazala se izvrsnom u tu svrhu. Za pretraživanje koda cijele jezgre uklonio bi se niz „`tools/linsched`“.

LinSched nije često ni održavan, ali zato svaka njegova nova inačica donosi mnogo novih mogućnosti. Usporedbom dosadašnjih inačica 2.6.23.14, 2.6.32, 2.6.35 i 3.3.0-rc7 u to se može i sam uvjeriti [36]. Izvorno se nije moglo ni emulirati proizvoljnu topologiju, nego samo onu stvarnu. Usprkos svim nabrojenim nedostatcima, čini se kao najbolji dostupan simulator Linuxova raspoređivača, pa ga, recimo, hvali i IBM [37]. Također je besplatan i otvorena koda. Glavni je njegov razvijatelj trenutačno Paul James Turner iz Googlea.

Ovakav je sustav izvorno bio pokretan unutar virtualnoga stroja s instaliranim sustavom Ubuntu 12.04.1 LTS unutar virtualizacijskoga programa VMware Player. U sklopu prošlosemestarskoga Projekta u taj je sustav dodano nešto novih mogućnosti, za što vidi projekt [38]. U Projektu se, inače, radio sustav u kojem se svako malo slučajno odaberu dva zadatka iz razlikih redova, a zatim im se zamijene redovi ako mogu.

4.2. Druga inačica ispitne okoline

Budući da LinSched ne zna apsolutno ništa o dinamičkim svojstvima memorijske hijerarhije jer emulira samo tijek algoritma, a ne i njegov višeprocesorski rad, brzo se od njega odustalo. Naime, takve performanse ne bi imale veze sa stvarnošću. Tad se pokušalo prevoditi cijele izmijenjene jezgre i instalirati ih, ali još uvijek unutar istoga virtualnoga stroja preko VMware Playera. Budući da VMware Player može emulirati najviše četiri prividna procesora, a čak ni VMware Workstation ne može više, odustalo se i od toga. O emuliranju prividnih procesora više ovdje [39].

4.3. Treća inačica ispitne okoline

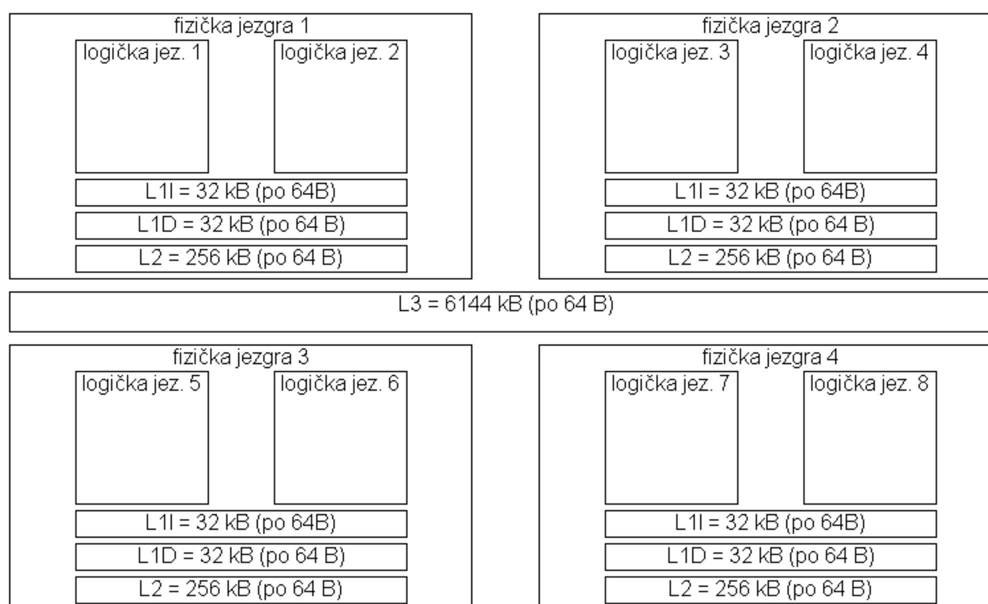
Shvativši da VirtualBox može emulirati čak i do 32 prividna procesora, pokušalo se samo zamijeniti VMware Player njim, ali instaliranjem novoga virtualnoga stroja. Međutim, budući da se na kraju sve ipak svodi na virtualizaciju, ustvrdilo se da rezultati još uvijek ne će biti mjerodavni. Tako se pokretanjem naredbe `lscpu` za ljusku Bash,

koja ispisuje procesorsku hijerarhiju [40], ustanovilo da ona nije ni emulirana precizno. Konkretno, na ovom se računalu, o čijoj će konfiguraciji riječi biti u odjeljku 4.4, emulira jedan procesor s osam fizičkih jezgara. Konačno se zamijenio VirtualBox konkretnim okruženjem, odnosno sekundarnim diskom računala na kojem je rad pisan. Još se uvijek radi o Ubuntuu 12.04.1 LTS.

4.4. Konačna ispitna okolina

Ukupno se tri puta mijenjalo cijelo okružje za rad, što je oduzelo daleko previše vremena. Posebno je dugo potrajalo istitravanje nepodnošljivosti GRUB-a, što je Linuxov učitavač (engl. *bootloader*), i Windowsa 7. Također i površno proučavanje oruđa `boot-repair`, koje će se spominjati u odjeljku 4.5. Teško je i zamisliti što bi se dogodilo da se Linux išao instalirati na isti disk kao i Windowsi. Ipak, usprkos svemu, smatra se da je konačna ispitna okolina upravo ona okolina koja može dati najpouzdanije rezultate. U svakom je slučaju za zadano ispitivanje bolja od bilo koje emulirane okoline.

Računalo na kojem se ispitivao raspoređivač ima Intelov procesor skoro najnovije arhitekture Sandy Bridge, i to model i7-2670QM. Taj procesor ima četiri hipernitne jezgre od kojih svaka fizička jezgra, to jest njezine dvije logičke koje ju dijele, ima po 32 kB priručne memorije L1 i 256 kB priručne memorije L2. Govoreći o memoriji L1, misli se na podatkovnu priručnu memoriju, dok je količina instrukcijske memorije jednaka. Ovih ukupno osam jezgara dijeli i 6144 kB priručne memorije L3. Veličine linija L1-memorija, L2-memorije i L3-memorije redom su 64 B, 64 B i opet 64 B. Vidi sliku 4.1. Sam je procesor frekvencije 2.2 GHz, a svi su ovi podatci dobiveni oruđem `PC Wizard` [41].



Slika 4.1: Ispitni procesor

Računalo ima i 8 GB radne memorije DDR3 [42], što zapravo tako reći znači da je ima beskonačno mnogo. Naime, ispitni sustav sastojat će se od nekoliko desetaka zadataka gdje po njih nekoliko dijeli komad memorije reda veličine megabajta, o čem će više biti u poglavlju 5. Ovako zaokružen sustav čini osnovu današnjega prosječna nepokretna (engl. *non-mobile*) računala. Konkretno se radi o prijenosnom računalu K93SM-YZ109, koje je pri kupnji spadalo u viši cjenovni razred, čemu danas više nije slučaj.

Iako bi se u idealnom slučaju poboljšanje Linuxova raspoređivača trebalo ispitati na nekoj naprednoj arhitekturi NUMA, do takve arhitekture nije bilo moguće doći unutar zadana vremena za pisanje rada. O tom će još biti riječi u odjeljku 5.5. Ovo nužno ne mora biti problem. Naime, ako se na današnjem prosječnom višeprocesorskom računalu ostvaruje poboljšanje performansa od deset ili dvadeset posto, to bi trebao biti dovoljan razlog za razmatranje uključivanja opisanih izmjena u glavnu jezgru (engl. *mainline kernel*) Linuxa.

Ostaje, ipak, pitanje koliko prosječan korisnik izvodi vremenski kritične zadatke, ali se slične zamisli mogu ostvariti i za sustav običnih zadataka. U ovom radu nije se tomu posvetilo, opet, zbog daleko veće zamršenosti sustava običnih zadataka. Ta zamršenost proizlazi upravo iz spomenute veće predodređenosti u vremenski kritičnih zadataka. Isto bi tako možda mogli profitirati i drugi operacijski sustavi s rasporedbenim politikama sličnim ovom kružnom raspoređivanju, pa tako i oni razmatrani u

poglavlju 2.

Tako su na ispitno računalo instalirane ukupno tri jezgre operacijskoga sustava na isti Ubuntu, usporedno. Sve su jezgre 64-bitne i ne treba buniti često spominjanje simbola `x86` umjesto `x86_64` jer su se otprije nekoliko godina direktoriji za te dvije arhitekture spojili, osim nekih sitnica u svrhu unatražne kompatibilnosti [43]. Tako je, primjerice, direktorij `arch/x86_64` prazan, izuzevši prečac `bzImage` u direktoriju `boot` koji pokazuje na istu stazu u direktoriju `arch/x86`.

I pregledom se prvih nekoliko redaka datoteke `.config`, koju se već spominjalo, vidi da je sustav konfiguriran u 64-bitnu inačicu. Može se i neuspješno pokušavati pokrenuti `gcc` sa zastavicom `-m32` ili ispisati osmicu kao `sizeof(long)`. Ako isprva nije jasno o čem govorim, prva će stvar pokušati prevesti neki kod za 32-bitni sustav i ne će uspjeti. Druga će pak pokazati da veličina tipa podatka `long` nije 4 kao na 32-bitnim inačicama Linuxa. Na kraju budi rečeno da se pojmovima `x86` i `x86_64` obuhvaćaju i Intelove i AMD-ove arhitekture, usprkos nazivu.

4.5. Početna ispitna jezgra

Početna, a poslije će se vidjeti zašto se tako zove, jezgra, ona je dobivena s Linuxom. Takva se jezgra obično naziva jezgrom *vanilla*. Konkretno se radi o jezgri 3.2.0.29, iako zbog mnogih problema koje donosi usporedno pokretanje Windowsa i Ubuntua na istom računalu oruđe `boot-repair`, koje popravlja te probleme, zapravo instalira inačicu 3.2.0.45.

Naime, kad se Ubuntu instalira uz neki drugi operacijski sustav u mnogo se slučajeva uopće ne može pokrenuti, a tako i u ovom slučaju s Windowsima 7 na prvom disku. To se onda rješava ponovnim pokretanjem instalacijskoga programa, odlaskom na *Try Ubuntu* umjesto na *Install Ubuntu* i pokretanjem ovoga oruđa [44]. Program pak neminovno instalira najnoviju inačicu jezgre. Ta se jezgra skida s Ubuntuovih poslužnika. Ona se svakih desetak dana nadograđuje, pa će se njezina inačica skoro sigurno promijeniti između dva preformatiranja diska, ali je uvijek 3.2.0.x. Ova je jezgra konkretno bila aktualna oko 10. lipnja ove godine.

Za tu se početnu jezgru nekolicinom ispitivanja ustanovilo da daje iste rezultate kao i jezgra iz odjeljka 4.6. Također su se usporedile datoteke izvornoga koda između tih dviju jezgara i nije otkrivena niti jedna važna promjena koja bi mogla utjecati na rezultate. Nađeno je jedino nekoliko ispravaka sitnih pogrješaka. Jedina je veća stvar bila preslikavanje datoteka staze `kernel/sched_*` u stazu `kernel/sched/*` te o jezgri *vanilla* više ne će biti govora.

Spomenut će se jedino da se ta jezgra ostavila u sustavu za slučaj da ne bi koja izmjena onesposobila druge dvije jezgre, a na njoj su se i prevodile druge jezgre s ciljem uklanjanja možebitnih ovisnosti.

4.6. Prva ispitna jezgra

Prva je jezgra, brojeći od ništice, jezgra dobivena pak zajedno s LinSchedom. Dobiven je cio izvorni kod jezgre, kao kad se pozove naredba `sudo apt-get source` za jezgru. Ta je jezgra inačice 3.3.0-rc7 i dovoljno je bliska jezgri `vanilla`, tako da nije bilo nikakvih problema u njezinu instaliranju. Čak je skoro i normalno radila bez ponovna prevođenja modula. Za najnovije inačice Ubuntua poput onih s jezgrom inačice 3.8 [45] ili novijom ne mora značiti da ne bi bilo problema. Usprkos nizu „rc“, jezgra je stabilna.

U ovu se prvu jezgru samo dodala nova rasporedbena politika `SCHED_RR2` da se ispitivanja mogu uredno izvoditi, što nije utjecalo na rezultate. Da se nije dodala nova politika, ne bi se mogle usporediti performanse između politike `SCHED_RR2` istovjetne `SCHED_RR`-u i politike `SCHED_RR2` s izmjenama, a ovako su i ispitni programi isti za obje jezgre.

4.7. Druga ispitna jezgra

Druga je jezgra jednaka prvoj jezgri, s tim da je u njoj i izmijenjen algoritam guranja i povlačenja te sve ostalo što je opisano u poglavlju 3. Kod je **potpuno** isti kao u prvoj jezgri, ali konfiguracija jezgre nije, jer je u prvoj definiran samo makro `CONFIG_DA`, a u drugoj i `CONFIG_DA2`. Ova će se jezgra zvati jezgrom 3.3.0-rc8, kako joj je postao nov naziv u datoteci `Makefile`, budući da dvije jezgre iste inačice ne mogu supostojati na istom sustavu. Dakle, promijenjena je i datoteka `Makefile`, redak 4, „`rc7`“ u „`rc8`“. Vidi ispis 4.1.

Ispis 4.1: Makefile

```
4 EXTRAVERSION = -rc8
```

Ta promjena nazivne podinačice jezgre onda utječe na direktorij unutar kojega se instaliraju moduli, što je `/lib/modules/3.3.0-rc{7,8}`, zbog čega se i ne mogu imati dvije iste inačice jezgre. To onda utječe na naziv poslije obrađenih datoteka `config-3.3.0-rc{7,8}` i parametra naredbe `mkinitramfs`. Rezultati koji će

se pokazati usporedba su, dakle, performansa ispitnih programa na ovim dvjema jezgrama.

4.8. Pokretanje sustava

Ne treba posebno naglašavati da je bilo potrebno prilagoditi i konfiguraciju pokretanja (engl. *boot*) operacijskih sustava. To se po lijepom sustavskom administriranju obavlja mijenjanjem GRUB-ovih konfiguracijskih poddatoteka, a onda pokretanjem Bashove naredbe `grub-update` [46]. Ovdje se ipak tako nije činilo, nego se odmah išlo u datoteku `/boot/grub/grub.cfg` i ručno sve promijenilo. Tako je kopirana konfiguracija jezgre *vanilla*, s tim da se je promijenila staza prevedene jezgre i početne slike. Vidi ispis 4.2.

Ispis 4.2: /boot/grub/grub.cfg

```
112 menuentry 'DA' --class ubuntu --class gnu-linux --class gnu
112     --class os {
(...)
120   linux /boot/DA/bzImage root=UUID=(...) ro   quiet splash
120       $vt_handoff
121   initrd  /boot/DA/initrd
122 }
123 menuentry 'DA2' --class ubuntu --class gnu-linux --class gnu
123     --class os {
(...)
131   linux /boot/DA2/bzImage root=UUID=(...) ro   quiet splash
131       $vt_handoff
132   initrd  /boot/DA2/initrd
133 }
```

Promijenjena je datoteka također priložena, ali njezin sadržaj ovisi o inačici jezgre, o slučajnu identifikatoru `UUID` koji instalacijski program pridijeli primarnoj particiji diska, a i o ostatku diskovnoga sustava.

4.9. Prevođenje sustava

Sustav se prevede tako da se, po izmjeni koda, najprije u vršnom direktoriju pozove naredba `make`, točnije `sudo make all`. Naredba će prevesti kod, stvoriti jezgru sustava i prevesti module ako treba. Više se može doznati naredbom `make help`. Prevođenje se modula najčešće ne će dogoditi, ali se u slučaju da se ono ipak dogodi onda oni moraju ponovno instalirati naredbom `sudo make modules_install`. Naravno, prvi se put uvijek moraju prevesti.

Od modula se tada treba i stvoriti „početna slika"za sustav, što se obavi naredbom `sudo mkinitramfs -o /boot/DA/initrd 3.3.0-rc7`, odnosno `sudo mkinitramfs -o /boot/DA2/initrd 3.3.0-rc8` [47]. Prije toga mora se kopirati već znana datoteka `.config` kao `/boot/config-3.3.0-rc{7,8}`. Ovdje je `mkinitramfs` zamjena za stariji `mkinitrd` [48], a, kao što se vidi, za instaliranje ovih jezgara stvoreni su i direktoriji `/boot/DA` i `/boot/DA2`. Svi su ovi koraci opisani u datoteci `skr2.sh`.

Dakle, sustav čine prevedena jezgra i početna slika, tako da još treba kopirati jezgru iz u odjeljku 4.4 spomenute staze `arch/x86/boot/bzImage` u `/boot/DA`, odnosno `/boot/DA2`. U tim se direktorijima dakle drži jezgra `bzImage`, početna slika `initrd` i konfiguracijska datoteka `config-3.3.0-rc{7,8}`. `bzImage` skraćeno je od *big zip image*, što je razmjerno nov format prevedene jezgre [49]. Konfiguracijska se datoteka po potrebi seli u direktorij `/boot` i natrag ako bude ponovno trebalo prevesti module.

4.10. Dodatna ispitna jezgra

Kao dodatno se okružje na ispitnom sustavu uzeo ipak i sam simulator LinSched, iz nekoliko razloga. Za početak, puno je lakše ispitivati ispravnosti izmijenjenih algoritama, bilo tijekom prevođenja, bilo tijekom izvođenja, na takvu sustavu, umjesto da se svaki put stvara nova inačica jezgre. To stvaranje, naime, može potrajati i satima ako se promijeni neki dio o kojem previše toga ovisi.

Tako mijenjanje datoteke `include/linux/sched.h`, kao datoteke koju skoro sve uključuje, ima jednako razoran učinak kao i mijenjanje datoteke `Makefile`. Zbog toga se je došlo u napast prebaciti makro `SCHED_RR2` u datoteku `kernel/sched/sched.h`. Prevođenje jezgre ni iz čega inače na ispitnom sustavu traje dva sata. Budući da se stvaraju dvije jezgre, to može jako dugo potrajati.

Naravno, nakon što bi se ustanovilo da algoritam ipak radi na simulatoru LinSched,

promjene bi se propagirale u ispitne jezgre i provjerile na stvarnom sustavu, dok bi se rezultati izvođenja na LinSchedu odbacili. Možda isprva nije jasno, ali je za pokretanje LinScheda svejedno na kojoj se jezgri pokreće jer se izvodi kao program u korisničkom prostoru (engl. *user space*) te nema veze s jezgrom.

Nadalje, glede razloga za LinSched, posebno se je zgodnim pri ispitivanju na LinSchedu pokazalo ispravljanje zaključavanja podatkovnih struktura. Naime, LinSchedove nadomjestne funkcije za zaključavanje jednostavno su postavljanja zastavice, s rušenjem sustava (engl. *kernel panic*) ako se pokuša dvaput zaključati ili otključati. Vidi datoteku `arch/linsched/asm/spinlock.h`. To se inače ne smije dogoditi budući da se emuliranje raspoređivača izvodi kao jednonitni program bez istiskivanja, pa će dvostruki pokušaj signalizirati zaboravljanje otključavanja strukture ondje gdje se je trebala otključati. Naravno, u stvarnom sustavu takav sve samo ne ispravan algoritam uzajamnoga isključivanja ne može raditi.

Još je jedan razlog to što LinSched također olakšava ispitivanje sustava, budući da se u njem funkcijom `linsched_create_sleep_run` vrlo jednostavno mogu stvoriti zadatci koji određeno vrijeme rade, a određeno spavaju. Čak se može izvesti i izvlačenje toga vremena iz razlikih slučajnih razdioba poput Gaussove ili Poissonove funkcijom `linsched_create_rnd_dist_sleep_run`.

U takvu se radu s LinSchedom usput otkrilo i ispravilo nekoliko pogrješaka u njegovu kodu. Tako vremenski kritični zadatci na LinSchedu u najnovijoj inačici uopće ne rade jer spominjana funkcija `security_task_setscheduler` nije dobro zakrpana, pa se poziva ugrađena jezgrena funkcija umjesto nje. To je riješeno mijenjanjem datoteke `kernel/sched/core.c` tako što je u funkciji `__sched_setscheduler` promijenjen redak 4108 tako da se funkcija ne poziva. Vidi ispis 4.3. Tu je pogrješku netko i prijavio na spomenuti e-poštanski popis, iako se je to vidjelo prekasno da se ne izgubi vrijeme.

Ispis 4.3: kernel/sched/core.c

```
4108    retval = 0;//security_task_setscheduler(p);
```

Popravljanjem te pogrješke naletjelo se na drugu pogrješku. U internim LinSchedovim provjerava dosljednosti (engl. *sanity check*) za vremenski kritične procesorske redove zagrada je bila postavljena na pogrješno mjesto. To uzrokuje beskonačnu petlju ako u sustavu postoji ijedan vremenski kritični zadatak. Mijenjanjem datoteke `tools/linsched/sanity_check.c` u funkciji `check_rt_rq`, redcima 80-81 i to se popravilo. Vidi ispis 4.4. Rabim priliku ustanoviti koliko je apsurdno da kod

ruše provjere njegove ispravnosti.

Ispis 4.4: tools/linsched/sanity_check.c

```
80 idx = find_next_bit(array->bitmap, MAX_RT_PRIO, idx+1);
81   while(idx <
```

Treća pogrješka koju je trebalo ispraviti bila je u datoteci `/tools/linsched/tests/Makefile`, redak 36. Vidi ispis 4.5. Tu su zastavice povezivača postavljene na pogrješno mjesto i matematičke se funkcije poput funckije `sqrt` zato uopće ne mogu povezati. To se riješilo pomicanjem zastavica udesno.

Ispis 4.5: /tools/linsched/tests/Makefile

```
36   @${CC} -o $@ $@.percpu ${LFLAGS} -MMD
```

Još jedna stvar koja se je promijenila ta je da LinSched uopće ne podupire stvaranje višenitnih procesa, što je i očekivano budući da se to raspoređivača u izvornoj inačici uopće ne tiče. To se riješilo ručnim postavljanjem TGID-a prilikom stvaranja nova zadatka. Konkretno, u datoteku `kernel/fork.c`, u spominjanu funkciju `do_fork`, dodani su redci 1585-1590 tako da se nekad slučajno ne promijeni TGID, što bi značilo da se stvara samo nova sunit. Vidi ispis 4.6. Ovo je pak zamotano u makro `CONFIG_DA3`. Redcima 95-98 dodana je i jedna potrebna varijabla za generiranje slučajnih brojeva Microsoftovim LCG-om.

Ispis 4.6: kernel/fork.c

```
1585 #ifdef DA3
1586     slucajni = slucajni * 214013L + 2531011L;
1587     if (((slucajni >> 16) & 0xff) < 50) {
1588       p->tgid = current->tgid;
1589     }
1590 #endif
```

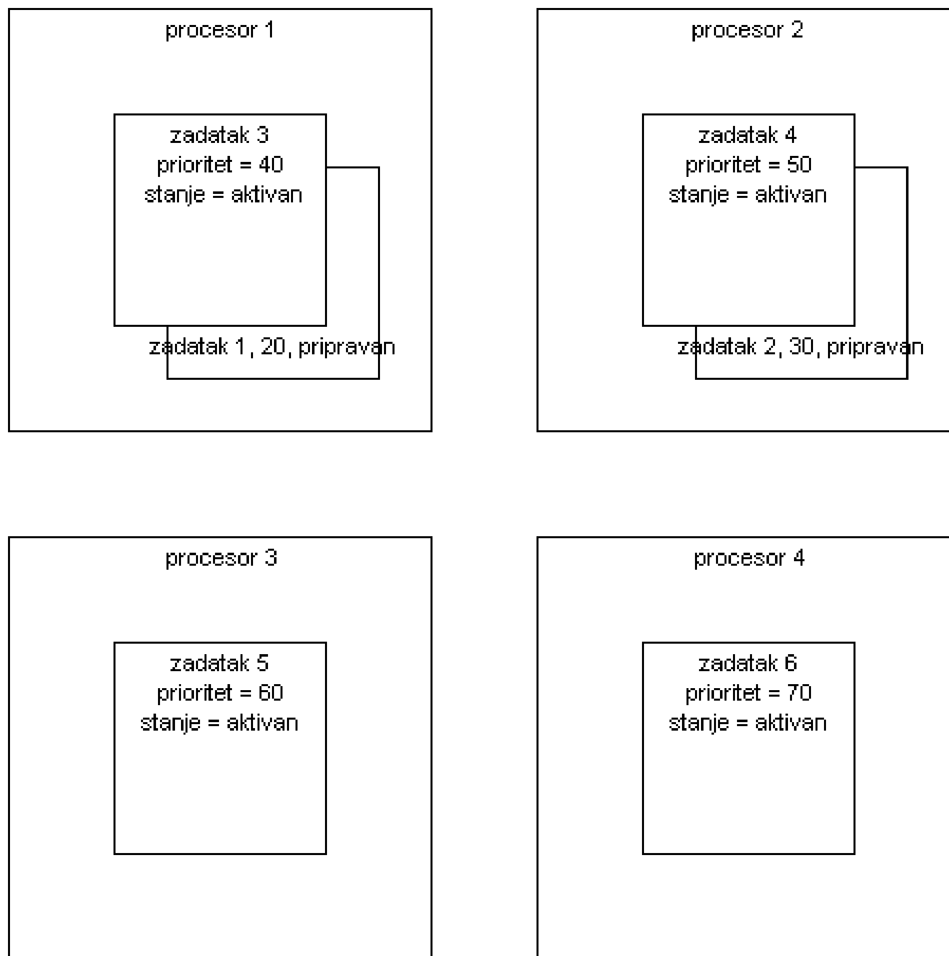
Na kraju budi rečeno da ovakvo nemaštovito nazivanje makroa proizlazi iz toga da je izvorno sve bilo zamotano u makro jednostavno nazvan `DA`. Odvajanjem dodavanja politike i izmjene politike, izvorno za lakše ispravljanje pogrješaka, nov je makro nazvan `DA2`. Konačno, ispravljanje pogrješaka u LinSchedu stavljeno je, logično, u `DA3`. Poslije su se i jezgre nazvale `DA` i `DA2`, pa se nazivi više nisu mijenjali. Jedino su iz razloga u poglavlju 4 u jezgri predmetkovani (engl. *prefixed*) nizom „`CONFIG_`“.

5. Rezultati

5.1. Očekivani rezultati

Ispitnim programom koji se rabio za ovjerovljenje poboljšanja performansa pokušalo se predstaviti skup prosječnih višenitnih programa koji dijele neke podatke. Program će biti opisan u odjeljku 5.2. Takvi bi višenitni programi u okružju s bar malo pametnijim raspoređivanjem njihovih suniti, u usporedbi sa slučajnim raspoređivanjem, trebali davati i nešto bolje rezultate.

Kao što je rečeno u poglavlju 3, vremenski je kritični sustav mnogo predodređeniji od sustava koji to nije. Tako se na njem može postići manje poboljšanje što je skup zadataka određeniji. Odnosno, ako su svi zadatci razlikoga prioriteta, ne treba očekivati čuda. Primjerice, razmotrimo sustav s četirima procesorima i s šest zadataka prioriteta redom 20, 30, 40, 50, 60 i 70. Vidi sliku 5.1. Ovdje je nebitno radi li se o zadatcima `SCHED_FIFO`, `SCHED_RR` ili čak `SCHED_RR2`. U takvu je sustavu strogo određeno koji će se zadatci morati izvoditi, a koji se ne će smjeti izvoditi.



Slika 5.1: Potpuno predodređen vremenski kritični sustav

Ovdje, naravno, po jedan procesor mora izvoditi zadatke s prioritetima 40, 50, 60 i 70, dok se preostala dva zadatka uopće ne smiju izvoditi. Takva je odredba sustava cijelo vrijeme stalna. S druge pak strane, ako su u nekom sustavu neki zadatci istoga prioriteta, otvara se prostor za optimiziranje. U primjerima koji će se ispitivati obično će biti dovoljno zadataka istoga prioriteta.

Sljedeće razmišljanje odnosi se na dijeljeni memorijski prostor suniti. Što više toga grupirani zadatci dijele u memoriji, priručna se memorija bolje iskorištava, a manje promašaja priručne memorije znači i bolje performanse. Ako pak zadatci dijele previše podataka, to se ne će moći iskoristiti na danoj memorijskoj hijerarhiji.

Tako, ako zadatci dijele polje veličine 100 megabajta i nešto slučajno čine po njem, na sustavu s najviše 6144 kB priručne memorije grupiranje takvih zadataka ne bi trebalo imati skoro nikakvu ulogu. Naime, i priručna memorija L3 trebala bi se prečesto

prazniti. Ovih nekoliko razmišljanja pomoći će pri razmatranju malena, odnosno velika, dobitka na performansama u pojedinim slučajevima u odjeljku 5.4.

5.2. Opis ispitivanja

Ispitni program koji se rabio u svojem osnovnom obliku, prema datoteci `test1.c`, stvara određen broj procesa, a svaki od tih procesa stvori određen broj novih niti. U osnovnom je programu to konkretno postavljeno na devet procesa s po tri niti zato što se na ispitnom procesoru nalazi ukupno 8 logičkih jezgrara. Sve niti spomenutih procesa zasad imaju novu rasporedbenu politiku `SCHED_RR2`.

Te suniti tih procesa dijele komad memorije određene veličine i obavljaju slučajne osnovne računske operacije na jednom potkomadu, a tih potkomada ima 16. Komad ili potkomad memorije zapravo je polje cijelih brojeva te je broj indeksa u polju manji od veličine polja `sizeof(int)` puta. Indeks je potkomada memorije dan globalno za proces i on se s vremenom slučajno mijenja. Konkretno se išlo na mijenjanje svakih u prosjeku nekoliko milijuna operacija tako da se on promijeni nekoliko desetaka puta. Veličina velikoga komada memorije u tom je osnovnom programu postavljena na veličinu jedne priručne memorije L2. To u ovom slučaju znači 256 kB, u skladu s odjeljkom 4.4.

Dakako, ovakvo je shvaćanje višenitnosti vrlo pojednostavljeno. Niti u općem slučaju uopće ne moraju dijeliti nikakve podatke. Ipak je odlučeno da je ovakvo shvaćanje prilično dobro za osnovnu procjenu dobitka na performansama pri izmijenjenim algoritmima. Zbog ovakva se shvaćanja neki rezultati, posebno oni veći od deset posto, mogu činiti veličanstvenim poboljšanjima, iako se ne radi o revolucionarnim izmjenama.

Glede oblika ispitnoga programa, osnovna zamisao višenitnosti ostaje ista u svim slučajevima, a variraju se određeni parametri. To su parametri poput prioriteta zadataka, veličine komada memorije, postotka opterećenosti sustava i slično. Sve će provedene varijacije u odjeljku 5.4 biti sustavno popisane, a njihova će ispitanost biti predstavljena rezultatima i ostvarenim dobitkom.

Prije tih konkretnih rezultata, objasnit će se i zašto se nije ispitala kombinacija običnih zadataka i vremenski kritičnih zadataka. To se nije učinilo zato jer obični zadatci ne utječu na vremenski kritične zadatke. Iako običnih zadataka u svježe pokrenutu sustavu ima stotinjak, što se može vidjeti naredbom `ps ax`, vremenski se kritični zadatci doživljavaju mnogo važnijima od njih. To je i potpomognuto činjenicom da ih može stvarati samo administrator, odnosno `root`, pa se tomu nije uopće čuditi. Ovdje

se po treći put vraća na funkciju `security_task_setscheduler`.

Dakle, Linux izvodi vremenski kritične zadatke dokle ih god ima i za to vrijeme uopće ne obraća pozornost na druge zadatke. Dobro, to zapravo nije točno jer Linux ne dopušta da se oni izvode cijelo cijelcato vrijeme, nego najviše 95 posto vremena. To se i vidi po tom da se za vrijeme ispitivanja, ako se pokuša još nešto raditi, to uspori, ali ne i blokira.

Ako se pak neki vremenski kritični zadatak pokuša izvesti više od 95 posto vremena, jezgra će ispisati upozorenje da je aktiviran takozvani *RT throttling*. Jezgrina se upozorenja mogu pregledati naredbom `dmesg` [50]. Izvorno je bilo predlagano da se procesorsko vrijeme vremenski kritičnih zadataka ograniči na deset uzastopnih sekunda [51]. *RT throttling* zapravo je privremeno onemogućivanje vremenski kritičnih zadataka. Ono je povezano sa skupnim raspoređivanjem i ne utječe na ispitni program više nego što mora.

Ovo pokazuje da nije ni predviđeno da se vremenski kritični zadatci vrte cijelo cijelcato vrijeme, pa se ne treba čuditi što se to ne dopušta. Očito je procijenjeno da dobivanje 5%, točnije 100/95, više vremena za vremenski kritične zadatke nije vrijedno potpuna blokiranja svih ostalih zadataka dok se oni ne završe. Također, ako se neki stvarnovremenski zadatak izvodi bez prestanka, nitko ga ne može zaustaviti.

Skupno raspoređivanje (engl. *group scheduling*), takozvani *control groups* u koje se grupiraju zadatci, nema veze s nitima. Tako recimo u Linuxu postoje i takozvani *process groups* za grupiranje procesa (engl. *process grouping*) [52], ali ni oni nemaju veze s njima. Dakle, kao što postoje takozvani `pgrpi` za lakše upravljanje procesima, tako unutar sustava običnih zadataka i vremenski kritičnoga sustava odvojeno postoje `cgroupi` za grupiranje zadataka pri raspoređivanju [53], ali ni jedni ni drugi nemaju veze s nitima. `cgroupi` postoje i izvan sustava raspoređivanja i služe za općenitu raspodjelu resursa.

To ne znači da se u nekom diplomskom radu ne bi moglo pokušati eksperimentirati s, recimo, automatskim grupiranjem niti u jedne ili druge. Zaključimo utvrđivanjem da se skupno raspoređivanje u ispitivanju ne rabi. Tako ni *RT throttling* nema neočekivanih učinaka, iako se često uključuje, budući da ispitni zadatci više-manje stalno rade.

5.3. Mjerilo poboljšanja

Što se tiče metrike, broji se svaka operacija koju svaka nit obavi. Pod operacijom se misli na jednu slučajnu računsku operaciju iz odjeljka 5.2. Na kraju se programa,

koji se inače uvijek izvodi deset sekunda, ispisuje koliko je ukupno posla cio proces obavio. Taj posao bude reda veličine desetaka do stotina milijuna operacija.

Ovi milijuni postaju mjerilo performansa tako da se za ispitivanje dodaju još i dvije skripte. U priloženim su datotekama domišljato nazvane `gg.sh` i `hh.sh`. Prva skripta od dviju pokreće program pet puta za redom i svaki put zbraja koliko su ukupno posla obavili svi procesi. To čini tako da jednostavno skupi i zbroji sve što je ispitni program ispisao.

Druga pak skripta pokreće onu prvu skriptu četiri puta za redom i svaki put zbraja koliko je ukupno posla obavilo peterostruko pokretanje ispitnoga programa koje je ona izvela. Ta se druga skripta pokreće četiri puta za redom i ukupno ispisuje šesnaest brojeva. Skripta se mora pokrenuti uz administratorske ovlasti jer se inače uopće ne mogu stvoriti vremenski kritični zadatci.

Od spomenutih šesnaest brojeva uzima se prvih pet najvažnijih znamenaka i one se uprosječe. Ne treba naglašavati da se u ovom cijelom složaju (engl. *complex*) pazilo na zapis cijelih brojeva u računalu te nije dolazilo do preljeva. U načelu ove skripte sve skupa znače da se ispitni program koji se trenutačno razmatra pokreće ukupno 80 puta, a onda se iz toga računa koliko se prosječno obavi u pet njegovih pokretanja.

Ovakvo bi ispitivanje trebalo biti dovoljno, i dalo je prilično pouzdane rezultate u smislu da se spomenuti prosjek nikad ne raspe više od nekoliko posto. Konkretno, u slučaju ispitnoga programa s najvećim poboljšanjima performansa, a radi se o inačici broj 2 i datoteci `test2.c`, ispitna se skripta broj dva može u bilo kojem trenutku pokrenuti na prvoj jezgri. Onda se u bilo kojem trenutku pokrene na drugoj jezgri i svaki će, ali baš svaki put, biti očito da se na drugoj jezgri program izvodi znatno bolje nego na prvoj jezgri. Naravno, u svrhu osiguravanja istih uvjeta, ispitivanja se uvijek pokreću na svježe pokrenutu sustavu.

Čini se da se algoritam pokušavanja smještanja niti na procesor na kojem se već nalazi neka njezina sunit uspješno istitrava. Pritom se hoće reći da se brzo nađe dobro rješenje. To se može i provjeriti oruđima poput naredbe `top` [54] ili naredbe `ps` [55]. Za naredbu `top` ključna je opcija `H` te polja `PID` i `P`, a za naredbu `ps` opcija `H` te polja `lwp/spid/tid` i `pr`. Naredba je `ps` bolja za to jer se naredbom `top` ne može vidjeti koja nit pripada kojemu procesu.

Uistinu, ako su u sustavu zadatci koji su svi istoga prioriteta, stvarno je očekivati da ne će doći do grupiranja suniti na više mjesta. Pod grupiranjem se misli na grupiranje niti o kojem se cijelo vrijeme govori, koje je izvedeno u ovom radu i koje se trudi sve suniti postaviti što bliže, po mogućnosti na isti procesor. Naime, recimo u sustavu s pet procesora, prva će se sunit smjestiti na, primjerice, procesor 3, a svaka će sljedeća

biti stavljena upravo na taj procesor, osim u slučaju da se još nije stvorilo dovoljno vremenski kritičnih zadataka za zauzimanje svih procesora.

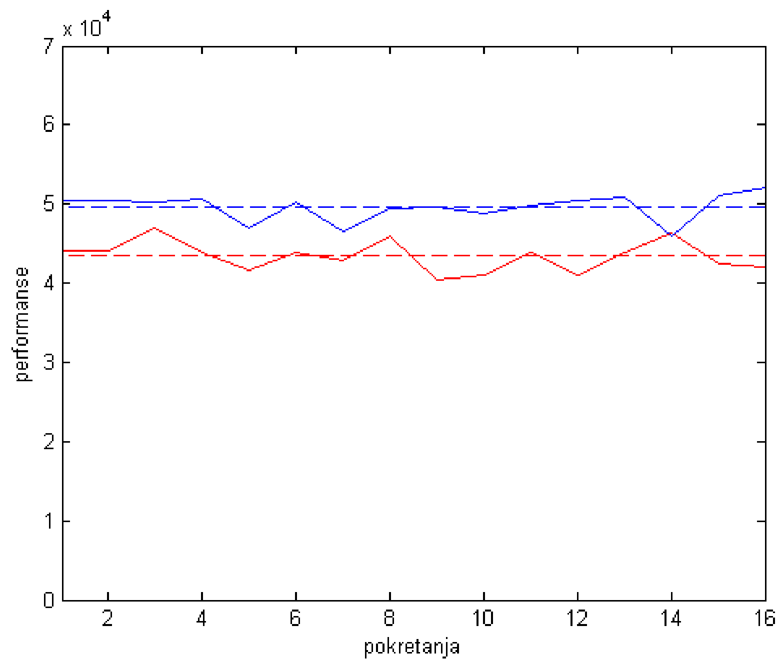
U tom bi pak slučaju, recimo, dvije suniti mogle zauzeti procesore 4 i 5, a onda bi nove suniti išle na oba ta procesora. To jest istina, ali ako se prije toga stigne pokrenuti algoritam guranja ili povlačenja, ove će se dvije suniti opet grupirati. Sve ostale niti ići će k njima. To će se dogoditi čak i ako se algoritam pokrene i kasnije, ali dovoljno rano da se niti nisu počele grupirati na obama mjestima.

Dakle, u ovakvu sustavu, osim u rijetkim slučajevima, grupiranje jako dobro radi. Treba naglasiti ogradu da se ovdje ne vodi računa o pravednosti, nego se samo pokušalo suniti staviti što bliže. Tako će se u slučaju da postoje tri procesora i jedan šesteronitni proces te dva jednonitna pokušati staviti svih šest niti prvoga procesa na isti procesor.

Glede broja pokretanja programa, smatralo se da bi veći broj pokretanja zamrsio ispitivanje. Naime, svako se pojedino ispitivanje ionako izvodi petnaestak minuta. 80 puta desetak sekunda čini upravo toliko. Ako se zbroji ponovno pokretanje sustava da bi se pokrenula druga jezgra, pokretanje ispitivanja na toj drugoj jezgri i skupljanje rezultata, to čini i do 45 minuta. Poenta je ispitivanja bila da se u otprilike jednom danu mogu izvesti sva pojedina ispitivanja, a to povećanjem broja izvođenja programa ne bi bilo moguće. Sva su ispitivanja, zajedno sa svim ostalim napisanim u sklopu ovoga Diplomskoga rada, priložena na CD-u, a zovu se `test1.c`, `test2.c`, `test3.c` i tako do `test12.c`.

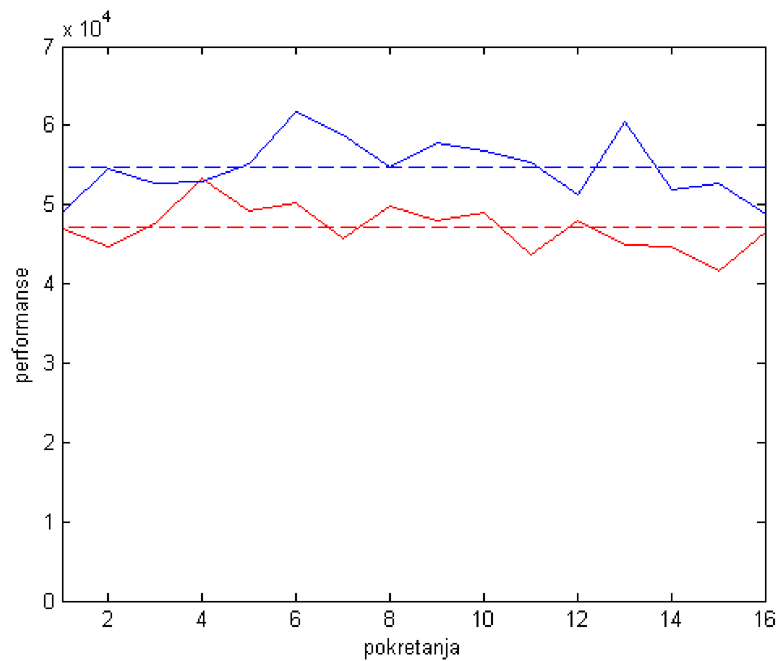
5.4. Dobiveni rezultati

Prvo se izvedeno ispitivanje odnosi na osnovni program kako je opisan u odjeljku 5.2. Brojevi koji se vide na grafu, crtanu u Matlabu [56] i prebačenu ovdje, odnose se na gore opisano mjerilo pet znamenaka. Crvenom su bojom označene performanse na prvoj jezgri, a plavom na drugoj. Tako su prikazane sve skupine od po pet pokretanja i konačan prosjek. U ovom ispitnom programu rezultati pokazuju da se radi o veliku dobitku pokretanjem programa na drugoj jezgri. Vidi se da grupiranje vrlo dobro radi. Dobitak je 14.29%.



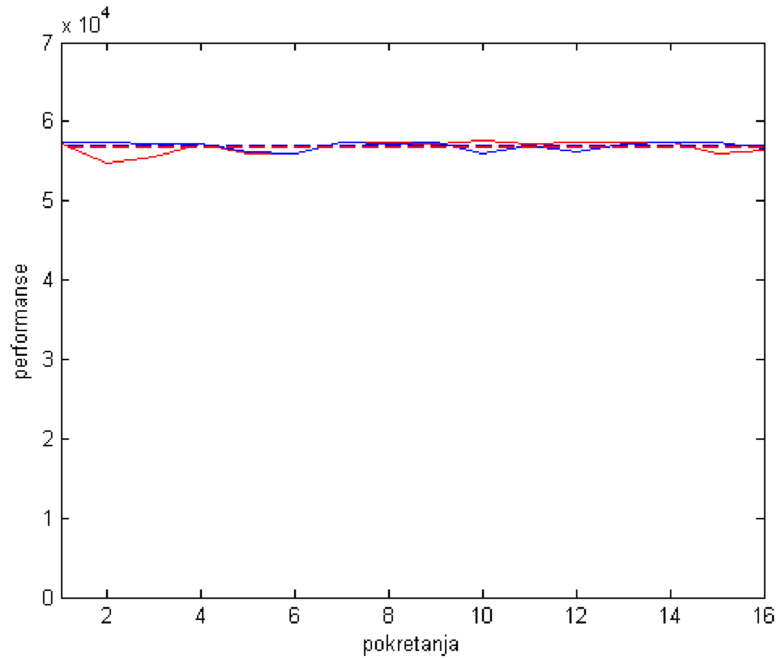
Slika 5.2: Prvo ispitivanje

U drugom je ispitivanju promijenjen broj zadataka u 11 procesa * 5 niti i sad je dobitak postao 16.02%. To je očekivano jer sad grupiranje može još više doći do izražaja.



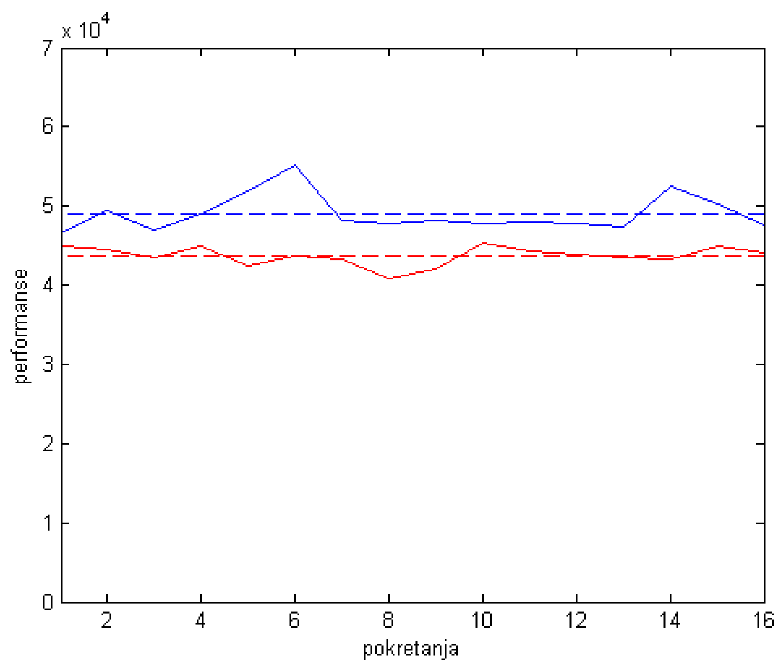
Slika 5.3: Drugo ispitivanje

U trećem je ispitivanju promijenjen broj zadataka u 7 procesa * 1 nit i sad je dobitak postao 0.37%. To je očekivano jer sad grupiranje nema nikakvu ulogu. Nema se, naime, što grupirati.



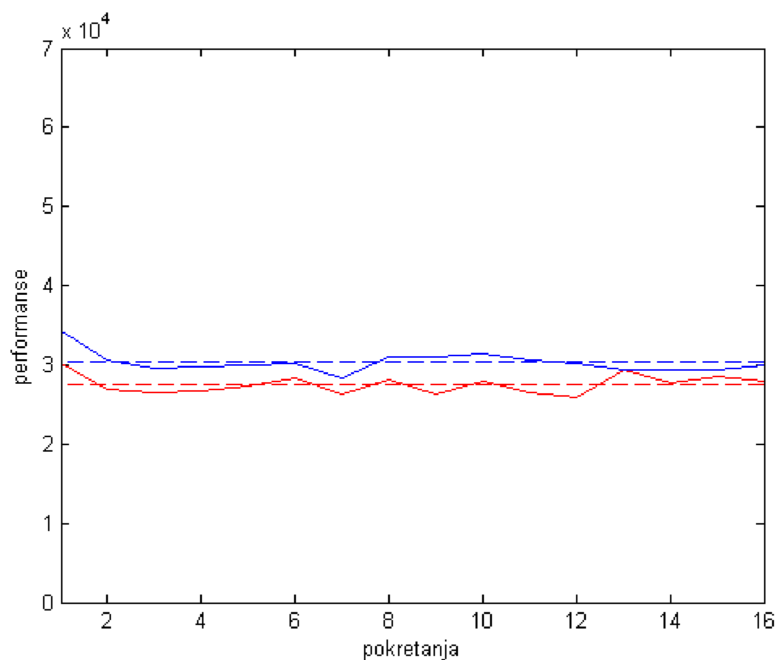
Slika 5.4: Treće ispitivanje

U četvrtom je ispitivanju promijenjena veličina potkomada na veličinu L2, odnosno, ona je ušesnesterostručena. Sad je dobitak postao 12.15%. To je očekivano jer sad ima više promašaja priručne memorije.



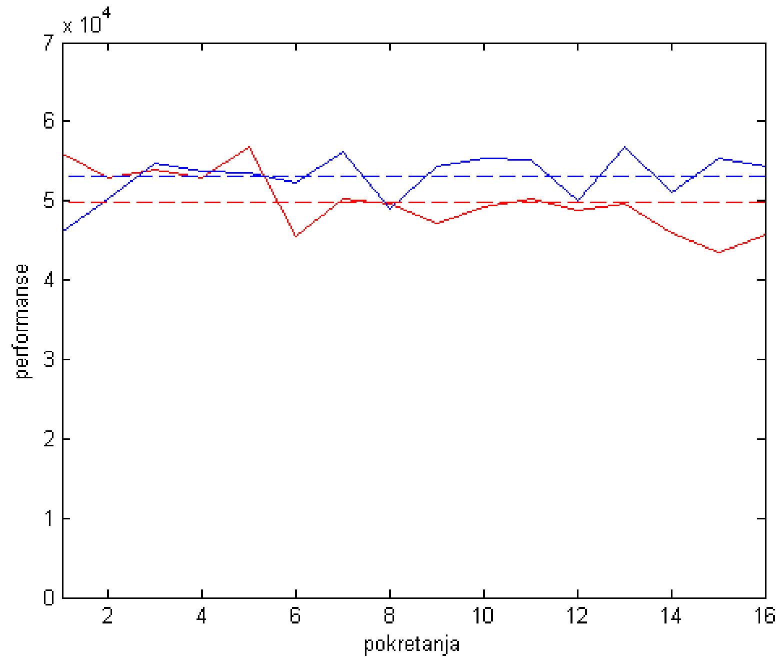
Slika 5.5: Četvrto ispitivanje

U petom je ispitivanju promijenjena veličina potkomada na veličinu L2 * 16, odnosno, onda je opet ušesnaesterostručena. Sad je dobitak postao 9.93%. To je očekivano jer sad ima još više promašaja priručne memorije. Ipak, očekivalo se da će performanse biti još manje.



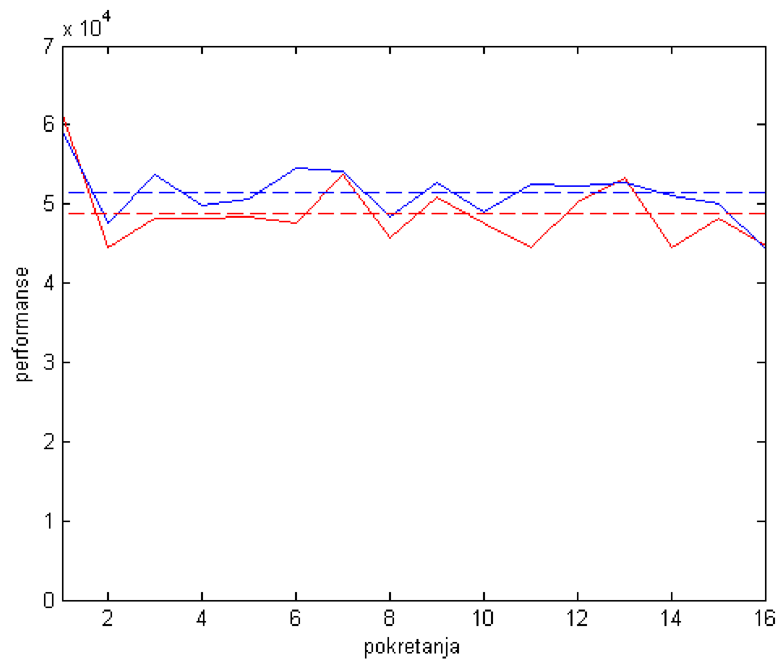
Slika 5.6: Peto ispitivanje

U šestom je ispitivanju promijenjen prioritet tako da zadatci slučajno odaberu jedan od četiriju prioriteta i sad je dobitak postao 6.26%. To je očekivano jer sad grupiranje ima manju ulogu.



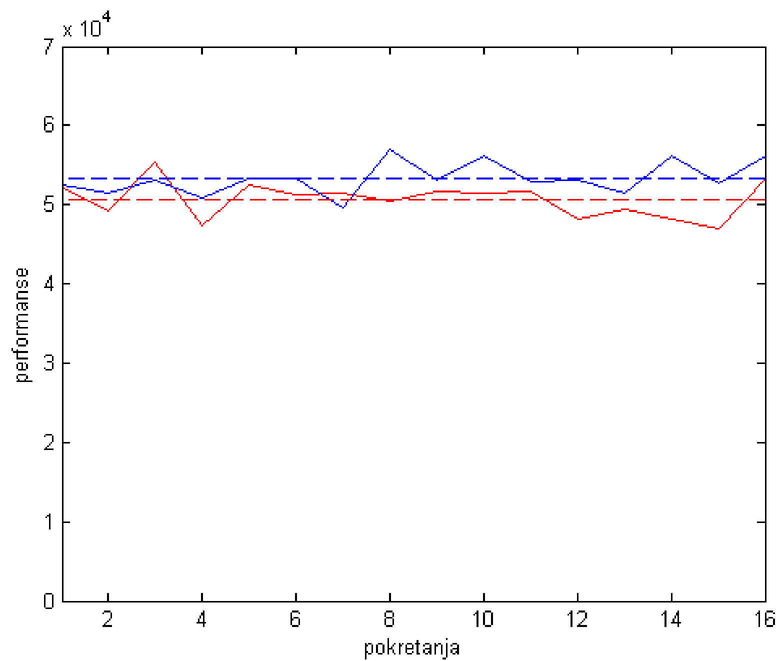
Slika 5.7: Šesto ispitivanje

U sedmom je ispitivanju promijenjen prioritet tako da zadatci slučajno odaberu jedan od četiriju prioriteta, ali trećina zadataka svaka četiri milijuna operacija odspava jednu sekundu i sad je dobitak postao 5.20%.



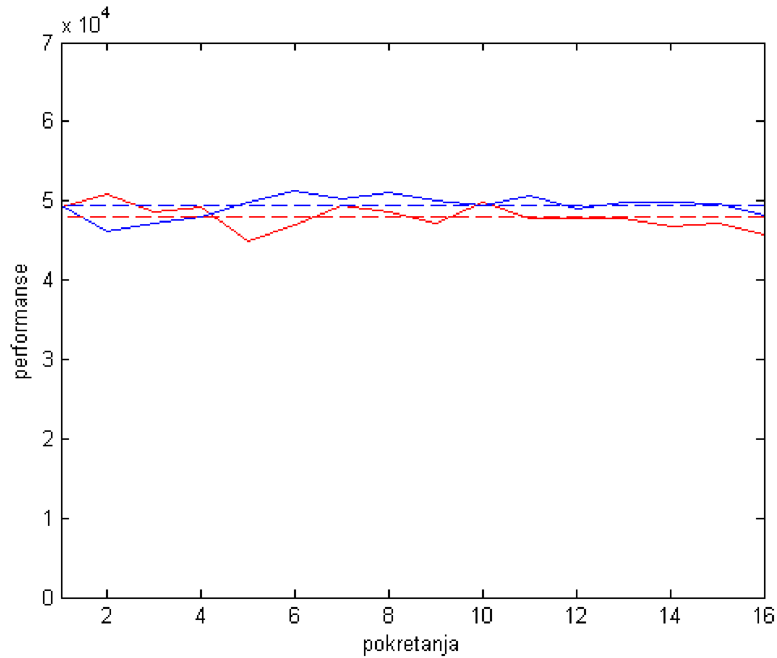
Slika 5.8: Sedmo ispitivanje

U osmom je ispitivanju promijenjen prioritet tako da zadatci opet slučajno odaberu jedan od četiriju prioriteta, ali dvije trećine zadataka svaka četiri milijuna operacija odspava jednu sekundu i sad je dobitak postao 5.24%.



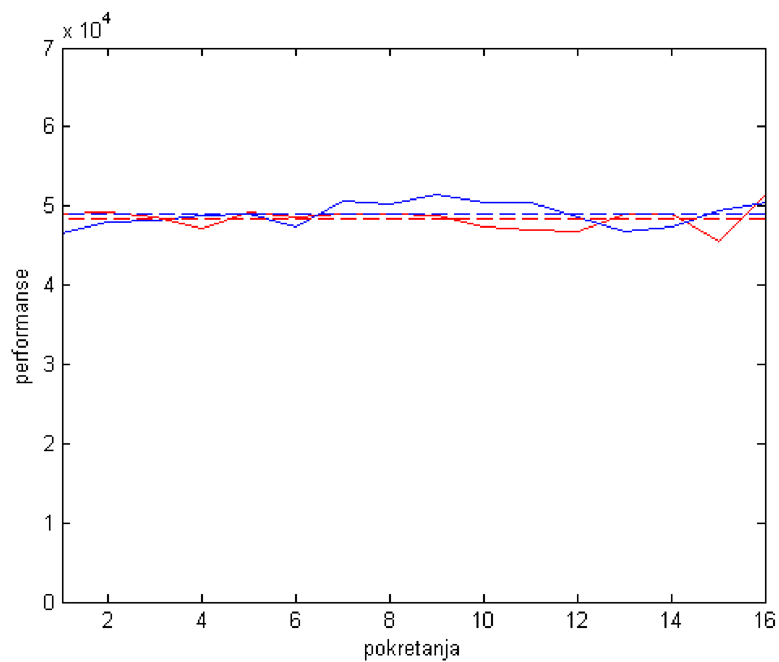
Slika 5.9: Osmo ispitivanje

U devetom je ispitivanju promijenjen prioritet tako da zadatci opet slučajno odaberu jedan od četiriju prioriteta, ali svi zadatci svaka četiri milijuna operacija odspavaju jednu sekundu i sad je dobitak postao 2.84%.



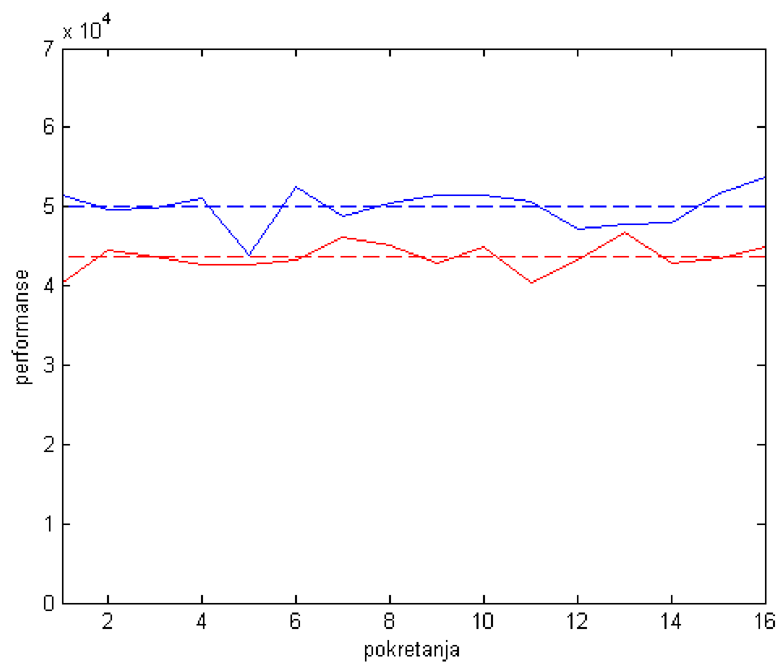
Slika 5.10: Deveto ispitivanje

U desetom je ispitivanju promijenjen prioritet tako da zadatci slučajno odaberu jedan od pedeset prioriteta, ali trećina zadataka svakih četiri milijuna operacija odspava jednu sekundu i sad je dobitak postao 1.17%. Ipak, očekivalo se da će performanse biti još manje.



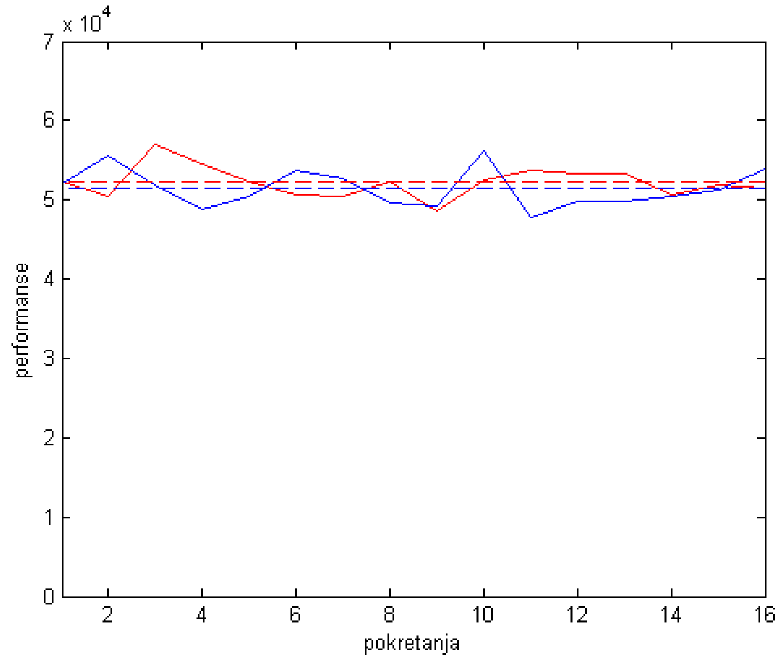
Slika 5.11: Deseto ispitivanje

U jedanaestom je ispitivanju promijenjena politika zadataka tako da trećina zadataka ima politiku `SCHED_RR` i sad je dobitak postao 14.52%. Čini se da grupiranje još uvijek ispravno radi.



Slika 5.12: Jedanaesto ispitivanje

U dvanaestom je ispitivanju promijenjena politika zadataka tako da trećina zadataka ima politiku `SCHED_FIFO` i sad je dobitak postao -1.48%. U ovom se ispitivanju vidi utjecaj slučajnosti.



Slika 5.13: Dvanaesto ispitivanje

Vidi tablicu 5.1.

Tablica 5.1: Konačni rezultati

Broj	Procesa	Niti	Potkomad	Prioriteta	Pospanih	RR	FIFO	Dobitak
1	9	3	L2 / 16	1	0	0	0	14.29%
2	11	5	L2 / 16	1	0	0	0	16.02%
3	7	1	L2 / 16	1	0	0	0	0.37%
4	9	3	L2	1	0	0	0	12.15%
5	9	3	L2 * 16	1	0	0	0	9.93%
6	9	3	L2 / 16	4	0	0	0	6.26%
7	9	3	L2 / 16	4	1 / 3	0	0	5.20%
8	9	3	L2 / 16	4	2 / 3	0	0	5.24%
9	9	3	L2 / 16	4	3 / 3	0	0	2.84%
10	9	3	L2 / 16	50	3 / 3	0	0	1.17%
11	9	3	L2 / 16	1	0	1 / 3	0	14.52%
12	9	3	L2 / 16	1	0	0	1 / 3	-1.48%

5.5. Komentar rezultata

Očit je zaključak da algoritam predložen u ovom radu poboljšava performanse u sustavu gdje je više višenitnih procesa koji dobro iskorištavaju dijeljenje memorijskih struktura između svojih niti. Podrazumijeva se i višeprocesorski sustav, naravno. Ipak, budući da su ova ispitivanja pokazala da se rezultati uz ostvarene popisane izmjene teško mogu pogoršati, razumno je pretpostaviti da bi takav kod u nekom svojem obliku mogao ući u jezgru.

U sretnije izvedenu povezivanju Linuxove programerske zajednice i akademske zajednice bilo bi moguće i provesti neopisivo opsežnija ispitivanja ovih algoritama na razlikim arhitekturama. Ta bi ispitivanja očito dala daleko pouzdanije rezultate, a onda bi se ti algoritmi, pod uvjetom da su dovoljno dobri, predložili za uvrštavanje u Linuxovu jezgru [57].

Ipak bi prije svega toga trebalo taj kod počistiti tako da se prevede na engleski i uklopi traženje suniti u postojeće mehanizme, umjesto uvođenja novoga, o čem se govorilo u poglavlju 3. Kao druga moguća poboljšanja onoga ostvarenoga u radu moglo bi se kod i opsežno komentirati, a možda i smisliti još koju zamisao pored navedenih četiri. Nekoliko je prijedloga za to dano i u tekstu.

6. Zaključak

U ovom se radu predlaže da se u mnogoprocesorskim sustavima niti istoga procesa pokušaju držati što bliže, po mogućnosti na istom procesoru. Taj je algoritam grupiranja zadataka iskušan na Linuxovu raspoređivaču preko razreda vremenski kritičnih zadataka. Potom je on ispitan na stvarnu računalnom sustavu. Rezultati su pokazali da se njim skoro uvijek postižu poboljšanja.

Rezultati su pokazali da je zamisao o držanju suniti što bliže jednu drugoj urodila plodom u prosječnom sustavu s kružnim raspoređivanjem zadataka i vremenskim odsječkom. Sljedeći bi korak bio provesti zamisao grupiranja u Linuxovu raspoređivaču običnih zadataka te provesti opsežna ispitivanja na razlikim arhitekturama. Ako se zamisao pokaže uspješnom na skupu običnih zadataka, taj se kod može uključiti u Linuxovu jezgru. Nakon toga, i ostali operacijski sustavi sa sličnim sustavima raspoređivanja mogu imati koristi od ove osnovne zamisli grupiranja povezanih zadataka.

LITERATURA

[1] Wikipedia. Multiprocessing. *Wikipedia - the free encyclopedia*, 2013. http://en.wikipedia.org/wiki/Multiprocessing.

[2] Wikipedia. Sandy bridge. *Wikipedia - the free encyclopedia*, 2013. http://en.wikipedia.org/wiki/Sandy_Bridge.

[3] Thomas Zangerl. Operating system scheduling on multicore architectures. Seminarski rad, University of Innsbruck, 2008. http://www.scribd.com/doc/4838281/Operating-System-Scheduling-on-multicore-architectures.

[4] Matija Horvat. Postupci raspoređivanja. Seminarski rad, Sveučilište u Zagrebu, 2012. ?

[5] *Documentation/scheduler/sched-stats.txt*. https://www.kernel.org/doc/Documentation/scheduler/sched-stats.txt.

[6] Todd A. Anderson Mohan Rajagoplan, Brian T. Lewis. Thread scheduling for multi-core platforms. Technical report, Programming Systems Lab, Intel Corporation, 2007. http://static.usenix.org/event/hotos07/tech/full_papers/rajagopalan/rajagopalan.pdf.

[7] Michael D. Smith Alexandra Fedorova, Margo Seltzer. Cache-fair thread scheduling for multicore processors. Technical report, Harvard University, Sun Microsystems, ? http://www.eecs.harvard.edu/ fedorova/papers/cache-fair.pdf.

[8] Luka Milić. Raspoređivanje u višeprocesorskim sustavima. Seminarski rad, Sveučilište u Zagrebu, 2012.

[9] http://www.codeproject.com/Articles/29449/Windows-Memory-Management.

[10] Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009. http://www.linuxjournal.com/magazine/completely-fair-scheduler.

[11] *man 2 getpriority*. http://linux.die.net/man/2/getpriority.

[12] http://ck.wikia.com/wiki/SchedulingPolicies.

[13] *man 2 sched_setaffinity*, . http://linux.die.net/man/2/sched_setaffinity.

[14] *man 2 sched_setscheduler*, . http://linux.die.net/man/2/sched_scheduler.

[15] *man 2 sched_setschedparam*, . http://linux.die.net/man/2/sched_schedparam.

[16] http://lkml.indiana.edu/hypermail/linux/kernel/0808.3/0503.html.

[17] *man 2 clone*. http://linux.die.net/man/2/clone.

[18] *man 2 fork*. http://linux.die.net/man/2/fork.

[19] *man 3 pthread_create*. http://linux.die.net/man/3/pthread_create.

[20] Wikipedia. Posix. *Wikipedia - the free encyclopedia*, 2013. http://en.wikipedia.org/wiki/POSIX.

[21] *man 7 pthreads*. http://linux.die.net/man/7/pthreads.

[22] *man 2 getpid*. http://linux.die.net/man/2/getpid.

[23] *man*. http://linux.die.net/man.

[24] *man 2 syscall*. http://linux.die.net/man/2/syscall.

[25] *man 2 gettid*. http://linux.die.net/man/2/gettid.

[26] http://www.youtube.com/watch?v=FkIWDAtVIUM.

[27] *man 3 exit*. http://linux.die.net/man/3/exit.

[28] http://kernelnewbies.org/FAQ/LinkedLists.

[29] Robert Love. Kernel locking techniques. *Linux Journal*, 2002. http://www.linuxjournal.com/article/5833r.

[30] Ankita Garg. Real-time linux kernel scheduler. *Linux Journal*, 2009. http://www.linuxjournal.com/magazine/real-time-linux-kernel-scheduler.

[31] *Documentation/kbuild/kconfig-language.txt*. https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt.

[32] http://wiki.gentoo.org/wiki/Kernel/Configuration.

[33] http://www.cs.unc.edu/ jmc/linsched/linsched_paper.pdf.

[34] Paul Turner. Linsched for v3.3-rc7. *Linux Weekly News*, 2012. http://lwn.net/Articles/486635/.

[35] *man 1 grep*. http://linux.die.net/man/1/grep.

[36] Ranjit Manomohan. Linsched for 2.6.35 released. *Linux Weekly News*, 2010. http://lwn.net/Articles/409680/.

[37] http://www.ibm.com/developerworks/library/l-linux-scheduler-simulator/.

[38] Luka Milić. Linsched. Seminarski rad, Sveučilište u Zagrebu, 2012.

[39] https://blogs.oracle.com/jsavit/entry/virtual_smp_in_virtualbox_3.

[40] *man 1 lscpu*. http://linux.die.net/man/1/lscpu.

[41] http://www.cpuid.com/softwares/pc-wizard.html.

[42] http://www.asus.com/Notebooks_Ultrabooks/K93SM/#specifications.

[43] http://forums.gentoo.org/viewtopic-t-897096-start-0.html.

[44] https://help.ubuntu.com/community/Boot-Repair.

[45] Wikipedia. List of ubuntu releases. *Wikipedia - the free encyclopedia*, 2013. https://en.wikipedia.org/wiki/List_of_Ubuntu_releases.

[46] http://www.dedoimedo.com/computers/grub-2.html.

[47] *man 1 mkinitramfs*. http://linux.die.net/man/1/mkinitramfs.

[48] http://permalink.gmane.org/gmane.comp.lpi.discuss/554.

[49] Wikipedia. Vmlinux. *Wikipedia - the free encyclopedia*, 2013. http://en.wikipedia.org/wiki/Vmlinux.

[50] *man 1 dmesg*. http://linux.die.net/man/1/dmesg.

[51] Jonathan Corbet. Sched_fifo and realtime throttling. *Linux Weekly News*, 2008. http://lwn.net/Articles/296419/.

[52] *man 2 getpgrp*. http://linux.die.net/man/2/getpgrp.

[53] *Documentation/cgroups/cgroups.txt*. https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[54] *man 1 top*. http://linux.die.net/man/1/top.

[55] *man 1 ps*. http://linux.die.net/man/1/ps.

[56] *help plot*. http://www.mathworks.com/help/matlab/ref/plot.html.

[57] http://www.linuxchix.org/content/courses/kernel_hacking/lesson9.

Prilagodba Linuxova raspoređivača za mnogoprocesorske sustave

Sažetak

U ovom se radu opisuje pristup raspoređivanju u višeprocesorskim sustavima grupiranjem niti istoga procesa, kao i ostvaraj zamisli u Linuxovu raspoređivaču. Najprije se opisuje raspoređivač, zatim prilagodbe, onda ispitivanje, pa rezultati.

Ključne riječi: raspoređivanje, višeprocesorski sustavi, Linux, SCHED_RR, LinSched, guranje, povlačenje, grupiranje niti

Adapting the Linux scheduler for manyprocessor systems

Abstract

In this thesis an approach to manyprocessor scheduling by grouping threads of the same process is described, and also the realization of the idea in the Linux scheduler. At the beginning the scheduler is described, then the adaptations, then the testing, at the end the results.

Keywords: scheduling, multiprocessor systems, Linux, SCHED_RR, LinSched, push, pull, grouping threads