Izgradnja novog rasporedivača poslova u Linuxu

Breznički-Herceg, Mihael

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1265

IZGRADNJA NOVOG RASPOREDIVAČA POSLOVA U LINUXU

Mihael Breznički-Herceg

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1265

IZGRADNJA NOVOG RASPOREDIVAČA POSLOVA U LINUXU

Mihael Breznički-Herceg

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1265

Pristupnik:	**Mihael Breznički-Herceg (0036539279)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	doc. dr. sc. Leonardo Jelenković

Zadatak: **Izgradnja novog rasporedivača poslova u Linuxu**

Opis zadatka:

Proučiti izvorni kod jezgre operacijskog sustava Linux i kako se iz njega izgrađuje jezgra. Istražiti izvorni kod raspoređivača poslova za normalne, nekritične dretve. Posebice proučiti njegovu strukturu i povezivanje s ostatkom sustava. Osmisliti i ostvariti novi raspoređivač za nekritične poslove i dodati ga uz postojeće raspoređivače. Usporediti svojstva dodanog raspoređivača s postojećim.

Rok za predaju rada: 14. lipnja 2024.

SADRŽAJ

1. UVOD

Raspoređivač poslova u operacijskom sustavu (eng. task scheduler) raspoređuje zadatke (procese i dretve) na procesore, odnosno odlučuje koje zadatke može izvoditi na procesorima u pojedinim trenucima. Svaki raspoređivač pokušava rasporediti zadatke što je moguće bolje prema svojim pravilima.

Cilj ovog rada je proučiti raspoređivač poslova u jezgri operacijskog sustava Linux, osmisliti i implementirati novi raspoređivač i usporediti ga s ostalim raspoređivačima.

Rad je podijeljen u sedam poglavlja. Drugo poglavlje opisuje strukturu i rad raspoređivača poslova koji se nalazi u jezgri Linuxa. U trećem poglavlju je naveden idejni opis rada novog raspoređivača koji se ostvaruje i dodaje u sustav. Četvrto poglavlje opisuje ostvarenje raspoređivača. Peto poglavlje prikazuje proces izgradnje jezgre u koju je ugrađen novi raspoređivač te kako se on može koristiti iz programa. U šestom se poglavlju uspoređuju svojstva novog raspoređivača s postojećima. Sedmo poglavlje iznosi zaključak.

2. RASPOREĐIVAČI ZADATAKA U JEZGRI LINUXA

2.1. Terminologija

Zadaci ili poslovi (eng. task) su osnovni elementi koje raspoređivač raspoređuje. U stvarnosti su to dretve i procesi koje je potrebno izvoditi na procesoru.

Zadaci se spremaju u red pripravnih zadataka (eng. run queue) i od tamo se odabire koji zadatak se šalje sljedeći na izvođenje. Red pripravnih zadataka je najčešće lista, ali može biti bili koja struktura za spremanje elemenata (npr. stablo, stog, ...).

Način raspoređivanja (eng. scheduling policy) određuje prema kojim pravilima se zadaci raspoređuju. Jedan raspoređivač može imati jedan ili više načina raspoređivanja.

2.2. Načini raspoređivanja zadataka u Linuxu

U ovom radu se nadograđuje jezgra Linuxa u inačici 6.6.9. U toj inačici postoji šest načina raspoređivanja korisničkih zadataka: SCHED_DEADLINE, SCHED_FIFO, SCHED_RR, SCHED_NORMAL, SCHED_BATCH i SCHED_IDLE. Svaki od tih načina raspoređivanja pripada jednom od raspoređivača i definira način kako se zadaci raspoređuju. Način SCHED_DEADLINE pripada raspoređivaču prema rokovima (interno ime DEADLINE, u nastavku DL), načini SCHED_FIFO i SCHED_RR pripadaju raspoređivaču kritičnih poslova (REALTIME, RT, eng. Real-Time, poslovi koji rade u *stvarnom vremenu*) te SCHED_NORMAL, SCHED_BATCH i SCHED_IDLE pripadaju raspoređivaču normalnih nekritičnih poslova (FAIR). Raspoređivač FAIR koristi algoritam CFS (Completely Fair Scheduler) pa se on često naziva i prema tome. Navedeni raspoređivači nisu ravnopravni: dok ima pripravnih zadataka koji se raspoređuju prema SCHED_DEADLINE ostali zadaci čekaju. Idući na redu su RT zadaci. Ako ni njih nema tek onda na red dolaze normalni poslovi. Osim navedenih, Linux ima dodatne dvije klase koje koristi samo interno, STOP i IDLE.

Za svaki od raspoređivača definirane su potrebne strukture podataka i operacije. Strukture podataka uključuju opisnik zadatka, `task_struct` i opisnik reda pripravnih `rq` te operacije raspoređivača kroz `sched_class`.

U strukturi `task_struct` nalaze se sve potrebne informacije za izvođenje i raspoređivanje zadatka. Element `prio` sadrži interni prioritet zadatka u rasponu od -1 do 139. Vrijednost -1 se koristi za raspoređivač prema rokovima DL, vrijednosti od 0 do 99 je za raspoređivač kritičnih poslova RT, a od 100 do 139 za raspoređivanje normalnih poslova FAIR. Elementi `se`, `rt` i `dl` su opisnici koji sadrže informacije za pojedini način raspoređivanja. U element `se` se sprema opisnik za FAIR način raspoređivanja, u `rt` se sprema opisnik za RT način raspoređivanja i u `dl` se sprema opisnik za DL način raspoređivanja. Element `on_rq` sadrži informaciju nalazi li se zadatak u redu pripravnih zadataka ili ne. Element `sched_class` sadrži pokazivač na opisnik raspoređivača koji se koristi za taj zadatak.

Isječak koda 2.1. Struktura `task_struct` (iz include/linux/sched.h)

```
struct task_struct {
	...
	int on_rq;
	int prio;
	...
	struct sched_entity se;
	struct sched_rt_entity rt;
	struct sched_dl_entity dl;
	const struct sched_class *sched_class;
	...
}
```

Opisnik `sched_class` je opisnik u kojem se nalaze adrese funkcija za pojedine akcije koje raspoređivač mora napraviti. Svaka od tih akcija se poziva preko opisnika `sched_class` od pojedinog raspoređivača. Na primjer, RT raspoređivač ima svoj instancu opisnika koji se zove `rt_sched_class`. U toj instanci opisnika ima spremljene adrese funkcija koje mora obavljati. Na primjer za stavljanje u red pripravnih zadataka ima funkciju `enqueue_task_rt()` i njezina adresa se sprema u element `enqueue_task` u instancu opisnika `rt_sched_class`. Svaki zadatak sadrži pokazivač na opisnik `sched_class` koja pripada raspoređivaču koji ga raspoređuje. Na primjer zadatak `t` koji se raspoređuje s RT raspoređivačem ima u sebi adresu na instancu `rt_sched_class` i kada se taj zadatak mora staviti u red pripravnih zadataka pozove se `t->sched_class->enqueue_task()`.

Isječak koda 2.2. Struktura `sched_class` (iz kernel/sched/sched.h)

```
struct sched_class {
	void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
	void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
	struct task_struct *(*pick_next_task)(struct rq *rq);
	...
}
```

Struktura `rq` je zasebna struktura koja predstavlja red pripravnih zadataka. Svaki procesor ima dodijeljen jedan red pripravnih zadataka `rq`. Zadaci koji su unutar reda pripravnih zadataka će se izvoditi na procesoru kojemu pripada taj red. Element `nr_running` sadrži broj zadataka koji su u tom redu za čekanje. Element `cfs` je pokazivač na strukturu `cfs_rq` koja je red pripravnih zadataka za FAIR način raspoređivanja. Element `rt` je pokazivač na strukturu `rt_rq` koja je red pripravnih zadataka za RT način raspoređivanja. Element `dl` je pokazivač na strukturu `dl_rq` koja je red pripravnih zadataka za DL način raspoređivanja. Element `curr` je pokazivač na zadatak koji se trenutno izvodi.

Isječak koda 2.3. Struktura `rq` (iz kernel/sched/sched.h)

```
struct rq {
	unsigned int nr_running;
	...
	struct cfs_rq cfs;
	struct rt_rq rt;
	struct dl_rq dl;
	...
	struct task_struct __rcu *curr;
	...
}
```

2.3. Raspoređivač poslova prema rokovima (DL)

Raspoređivač DL (SCHED_DEADLINE) ima najveći prioritet od svih raspoređivača u Linuxu i njegovi zadaci će se uvijek izvoditi prvi. Ideja je da se ovaj način raspoređivanja koristi za periodičke zadatke. Oni se periodički javljaju u sustav i trebaju biti gotovi sa svojim poslom do roka (relativno od početka periode). Raspoređivač razmatra te rokove i onaj s najbližim odabire za izvođenje.

Npr. neka postoji zadatak s periodom od 100 ms i rokom do 20 ms. Raspoređivač će na početku svake periode zadatka izračunati novi rok (početak periode + 20 ms) i usporediti ga rokovima drugih zadataka. Ako je rok ovog zadatka bliže, raspoređivač njega odabire za izvođenje (inače odabire drugi čiji je rok bliži).

Zadatak treba u svom kodu imati odgode do iduće periode nakon što završi s periodičkim poslom. Zadaća raspoređivača je samo da osigura ispravan redoslijed izvođenja pripravnih zadataka koji se raspoređuju prema roku. U ovaj raspoređivač (u Linuxu) je ugrađena i zaštita od predugih izvođenja nekih zadataka - zadacima koji premaše zadano vrijeme rada unutar periode automatski se pomiče rok za jednu periodu. Na taj način se osigurava da svi zadaci dobiju postavljeno vrijeme unutar svake svoje periode.

2.4. Raspoređivač kritičnih zadataka (RT)

Raspoređivač RT je drugi po prioritetu, ima veći prioritet od CFS-a, ali manji od DL-a. Njemu pripadaju poslovi s načinima SCHED_FIFO i SCHED_RR. Oba raspoređuju primarno prema prioritetu, ali se razlikuju kad u redu pripravnih ima više dretvi ista prioriteta.

Način SCHED_FIFO izvodi zadatke ista prioriteta po principu "tko prvi dođe, taj će se i prvi izvoditi" (eng. First In First Out). Zadatak koji se izvodi prestat će se izvoditi samo ako završi ili dođe drugi posao većeg prioriteta. Primjerice, ako imamo zadatke Z1 i Z2 prioriteta 50 te Z3

prioriteta 40, raspoređivač će najprije izvoditi zadatke Z1 i Z2 (prvo Z1 pa onda Z2 za SCHED_FIFO) te kad su oni gotovi onda Z3.

Način raspoređivanja SCHED_RR izvodi zadatak određeni kvant (period) i nakon toga prepušta procesor drugom zadatku istog prioriteta da se izvodi isto jedan kvant. Za isti primjer zadataka Z1, Z2 i Z3, zadaci Z1 i Z2 će dobivati po kvant vremena. Tek kad se oba maknu na red će doći Z3.

Raspoređivač RT interno koristi prioritete od 1 do 99 gdje je 1 najmanji prioritet, a 99 najveći prioritet. Taj prioritet se (prema van) preslikava u `prio` koji koriste svi raspoređivači zajedno (u strukturi `task_struct`) tako da 0 postaje najveći (prije je bio 99), a 98 postaje najmanji (prije je bio 1). Svaki prioritet ima svoj red pripravnih zadataka koji je zapravo dvostruko povezana lista. Na primjer svi zadaci s prioritetom 10 se spremaju u listu s prioritetom 10, svi zadaci s prioritetom 11 se spremaju u listu s prioritetom 11 itd.

2.5. Raspoređivač normalnih zadataka (FAIR)

Raspoređivač FAIR (načini SCHED_NORMAL, SCHED_BATCH i SCHED_IDLE) ima manji prioritet od DL i RT raspoređivača. Svaki zadatak koji se pokrene na Linux operacijskom sustavu, ako nije drugačije zadano, bit će raspoređivan prema SCHED_NORMAL (koristit će algortiam CFS).

CFS za svaki zadatak bilježi virtualno vrijeme izvođenja (eng. virtual runtime, u izvornom kodu `vruntime`) i uvijek izvodi zadatak s najmanjim virtualnim vremenom izvođenja. Zadaci su spremljeni u crveno-crno stablo (eng. red-black tree) u kojem su poredani od zadatka s najmanjim virtualnim vremenom izvođenja na lijevoj strani do onog s najvećim vremenom izvođenja na desnoj strani. Pod virtualno vrijeme izvođenja se sprema ukupno vrijeme koje se određeni zadatak izvodio od kada je stavljen u red pripravnih zadataka. Kada se zadatak izvodi, nakon nekog vremena, vrijeme koje se izvodio se doda u njegovo virtualno vrijeme izvođenja. Potpom se provjerava postoji li u stablu zadatak s manjim virtualnim vremenom izvođenja. Ako ne postoji onda se nastavlja izvođenje s istim zadatakom, a ako postoji onda se trenutni stavlja u stablo i onaj s najmanjim virtualnim vremenom izvođenja se izvodi.

Raspoređivač FAIR u postupku raspoređivanja interno koristi dodatni parametar „prema volji prepuštanja procesora drugim zadacima“ koja se zove `nice` tj. dobrota. Što je dobrota veća to je zadatak više voljan prepustiti izvođenje drugom zadatku. Vrijednost dobrote je između -20 i 19, gdje s -20 nije voljan uopće pustiti drugi zadatak da se izvodi, a s 19 je skoro uvijek voljan. Vrijednost dobrote se prevodi u `prio` tako da zadatak s dobrotom -20 ima `prio` 100, a zadatak s dobroto 19 ima `prio` 139. Kvantitativno, razlika u jednoj dobroti bi trebala predstavljati 10-15% procesorskog vremena.

SCHED_NORMAL je način izvođenja za obične zadatke koji nemaju eksplicitno zadan niti jedan drugi način izvođenja.

SCHED_BATCH je način izvođenja za zadatke koji nisu interaktivni i nemaju puno čekanja, nego dosta vremena provode u izvođenju. Takvi zadaci će uvijek imati manji prioritet od onih sa SCHED_NORMAL načinom izvođenja.

SCHED_IDLE je način izvođenja za zadatke koji se obavljaju kada nema drugih zadataka za obavljati. Procesor uvijek treba obavljati neki zadatak, čak i kada nema zadatka koji rade koristan posao pa mu se onda daje zadatak koji ne radi ništa korisno. Zadaci s ovim načinom izvođenja su najmanjeg prioriteta od svih i bilo koji zadatak s drugim načinom izvođenja ga zaustavlja.

3. KONCEPT NOVOG RASPOREĐIVAČA

Ideja novog raspoređivača je da koristi pravednu podjelu procesorskog vremena i da bude vrlo jednostavan za ostvarenje.

Novi raspoređivač prema prioritetu se postavlja između FAIR i RT raspoređivača. Zadaci novog raspoređivača imaju globalni prioritet 1 (odnosno `prio` 98), a RT raspoređivač ima prioritete od 2 do 99. Uz globalni prioritet 1 raspoređivač koristi dodatni prioritet od 0 do 9 za svaki pojedini zadatak, taj prioritet vrijedi samo u tom novom raspoređivaču.

Redovi pripravnih zadataka su slični kao i kod RT raspoređivača, svaki prioritet ima svoj red pripravnih zadataka koji je ostvaren dvostruko povezanom listom.

Rad raspoređivača je sličan kao i SCHED_RR načina izvođenja – uvijek se odabire naprioritetniji zadatak i daje mu se jedan kvant vremena. Ali nakon što zadatak dobije kvant procesorskog vremena, smanjuje mu se prioritet za jedan i takav stavlja u red pripravnih. Ako se dogodi da ga neki zadatak većeg prioriteta prekine prije isteka njegovog kvanta, onda mu se prioritet povećava za jedan prije stavljanja u red pripravnih. Ako zadatak dobrovoljno odustane od izvođenja prije isteka njegovog kvanta, onda mu se prioritet smanjuje za jedan.

4. OSTVARENJE NOVOG RASPOREĐIVAČA

Novi raspoređivač implementiran je u jezgri Linuxa inačice 6.6.9.

4.1. Definiranje politike NEW_SCHED

Prvo je potrebno dodati opciju za novi raspoređivač da korisnik prilikom izgradnje jezgre može odabrati želi li koristiti novi raspoređivač ili ne. Potrebno je u datoteku arch/x86/Kconfig dodati tekst:

```
menu "NEW scheduler"
config SCHED_NEW_POLICY
	bool "NEW scheduling policy"
	default y
endmenu
```

Nakon dodavanja opcije za izgradnju jezgre potrebno je nadodati novi način raspoređivanja. Način raspoređivanja se dodaje među ostale načine raspoređivanja u datoteku include/uapi/linux/sched.h. Prilikom dodavanja koristi se prije definirana opcija CONFIG_SCHED_NEW_POLICY tako da se prilikom izgradnje jezgre to uključi samo kada je ta opcija odabrana u .config datoteci.

Isječak koda 4.1. Definiranje politike SCHED_NEW (u include/uapi/linux/sched.h)

```
#ifdef CONFIG_SCHED_NEW_POLICY
#define SCHED_NEW 7
#endif
```

4.2. Dodavanje potrebnih struktura podataka

Sljedeće je potrebno napraviti strukture koje su potrebne za raspoređivač: struktura reda za pripravne dretve `new_rq` (koja se dodaje u `struct rq`) i struktura za zadatak raspoređivača `new_sched_task` (koja se dodaje u `task_struct`).

Struktura `new_rq` dodaje se u datoteku kernel/sched/sched.h. Ona sadrži dva elementa, `sched_queue` koji je polje listi dužine 10 i predstavlja pojedine liste za različite prioritete. Drugi element je `nr_running` i on sadrži broj zadataka koji se nalaze u redu za čekanje.

Isječak koda 4.2. Struktura `new_rq` (u kernel/sched/sched.h)

```
struct new_rq {
	struct list_head sched_queue[10];
	unsigned int nr_running;
};
```

Struktura `new_sched_task` dodaje se u datoteku include/linux/sched.h. Ona u sebi sadrži četiri elementa. Element `node` sadrži pomoćnu strukturu za postavljanje opisnika zadatka u listu pripravnih. Element `priority` je prioritet za NEW_SCHED s kojim se određuje u koji red pripravnih se stavlja zadatak. Element `time_slice` je kvant vremena u kojem se zadatak izvodi. Element `on_rq` je element u kojemu se zapisuje je li zadatak u redu pripravnih zadataka ili nije.

Isječak koda 4.3. Struktura `new_sched_task` (u include/linux/sched.h)

```
struct new_sched_task {
	struct list_head node;
	int priority;
	u64 time_slice;
	int on_rq;
};
```

Nakon ostvarenja potrebnih struktura, potrebno je te strukture dodati u strukture `task_struct` i `rq`. U strukturu `task_struct` u datoteci include/linux/sched.h nadodaje se novi element pod imenom `nst` koji sadrži `new_sched_task`. U strukturu `rq` u datoteci kernel/sched/sched.h nadodaje se novi element pod imenom `new_rq` koji sadrži strukturu `new_rq`.

Isječak koda 4.4. Izmjena strukture `task_struct` (u include/linux/sched.h)

```
struct task_struct {
	...
	#ifdef CONFIG_SCHED_NEW_POLICY
	struct new_sched_task nst;
	#endif
	...
}
```

Isječak koda 4.5. Izmjena strukture `rq` (u kernel/sched/sched.h)

```
struct rq {
	...
	struct new_rq new_rq;
	...
}
```

4.3. Funkcije raspoređivača

Nakon što je nadodano sve potrebno može se započeti s ostvarenjem raspoređivača. U jezgri Linuxa svaki raspoređivač napisan je u zasebnoj datoteci pa je tako i novi raspoređivač zapisan u zasebnoj datoteci kernel/sched/newsched.c. U toj datoteci se nalaze sve funkcije potrebne za

rad raspoređivača. Adrese tih funkcija se spremaju u varijablu `new_sched_class` koja je tipa `sched_class`.

Funkcija `init_new_rq` prima pokazivač na red pripravnih zadataka raspoređivača (struktura `new_rq`) i inicijalizira ga. Ona ide kroz polje `sched_queue` u `new_rq` i inicijalizira svaku listu i postavlja `nr_running` na 0, kao što se vidi iz koda.

Isječak koda 4.6. Funkcija `init_new_rq` (u kernel/sched/newsched.c)

```
void init_new_rq(struct new_rq* new_rq) {
	for(int i=0; i<10; i++) {
		INIT_LIST_HEAD(&new_rq->sched_queue[i]);
	}
	new_rq->nr_running = 0;
	printk(KERN_INFO "init_new_rq\n");
}
```

Funkcija `enqueue_task_new` prima pokazivač na red pripravnih zadataka, pokazivač na zadatak i zastavice. Posao funkcije je postaviti zadatak u red pripravnih zadataka.

Isječak koda 4.7. Funkcija `enqueue_task_new` (u kernel/sched/newsched.c)

```
static void enqueue_task_new(struct rq *rq, struct task_struct *p, int flags) {
	printk(KERN_INFO "enter enqueue_task_new\n");
	if(! p) {
		printk(KERN_INFO "exit enqueue_task_new, p is NULL\n");
		return;
	}
	if(! rq) {
		printk(KERN_INFO "exit enqueue_task_new, rq is NULL\n");
		return;
	}
	if(p->nst.on_rq) {
		printk(KERN_INFO "exit enqueue_task_new, task on rq\n");
		return;
	}
	//set time_slice (default RR time_slice is used)
	p->nst.time_slice = RR_TIMESLICE;
	//add task to the end of the list
	list_add_tail(&p->nst.node, &rq->new_rq.sched_queue[p->nst.priority]);
	//mark that the task is on runqueue
	p->nst.on_rq = 1;
	p->on_rq = 1;
```

```
	//increment the number of tasks on runqueue
	(rq->new_rq.nr_running)++;
	printk(KERN_INFO "exit enqueue_task_new, task %d enqueued\n", p->pid);
	print_rq(&rq->new_rq.sched_queue[p->nst.priority]);
}
```

Funkcija `dequeue_task_new` prima pokazivač na red pripravnih zadataka, pokazivač na zadatak i zastavice. Posao funkcije je maknuti zadatak iz reda pripravnih zadataka.

Isječak koda 4.8. Funkcija `dequeue_task_new` (u kernel/sched/newsched.c)

```
static void dequeue_task_new(struct rq *rq, struct task_struct *p, int flags) {
	printk(KERN_INFO "enter dequeue_task_new\n");
	if(! p) {
		printk(KERN_INFO "exit dequeue_task_new, p is NULL\n");
		return;
	}
	if(! rq) {
		printk(KERN_INFO "exit dequeue_task_new, rq is NULL\n");
		return;
	}
	//remove task from the list
	list_del_init(&p->nst.node);
	//set that task isn't on runqueue
	p->nst.on_rq = 0;
	p->on_rq = 0;
	//decrement number of tasks on runqueue
	(rq->new_rq.nr_running)--;
	printk(KERN_INFO "exit dequeue_task_new, task %d dequeued\n", p->pid);
	print_rq(&rq->new_rq.sched_queue[p->nst.priority]);
}
```

Funkcija `pick_next_task_new` prima pokazivač na red pripravnih zadataka. Posao funkcije je odabrati sljedeći zadatak koji će se izvoditi.

Isječak koda 4.9. Funkcija `pick_next_task_new` (u kernel/sched/newsched.c)

```
static struct task_struct* pick_next_task_new(struct rq *rq) {
	printk(KERN_INFO "enter pick_next_task_new\n");
	if(! rq) {
		printk(KERN_INFO "exit pick_next_task_new, rq is NULL\n");
		return NULL;
	}
```

```
	//find the biggest priority list that isn't empty
	int pos = find_not_empty(rq);
	//if it returns -1 all lists are empty
	if(pos < 0) {
		printk(KERN_INFO "exit pick_next_task_new, no task to run next\n");
		return NULL;
	}
	//get new_sched_task from list_head
	struct new_sched_task* new_task = list_entry(rq→new_rq.sched_queue[pos].
						next, struct new_sched_task, node);
	//get task_struct from new_sched_task
	struct task_struct* task = container_of(new_task, struct task_struct,
						nst);
	printk(KERN_INFO "exit pick_next_task_new\n");
	return task;
}
```

Funkcija `check_preempt_curr_new` provjerava treba li trenutni zadatak prepustiti izvođenje drugom zadatku. Funkcija prima pokazivač na red pripravnih zadataka, pokazivač na zadatak i zastavice. Funkcija prvo provjerava ako je `prio` trenutnog zadatka veći od zadatka `p` ili ako su isti i prioritet zadatka `p` je veći od trenutnog zadatka. Ako je jedan od uvjeta istinit makni trenutni zadatak iz liste, povećaj mu prioritet i stavi ga u listu koja ima novo izračunati prioritet. Nakon toga pozovi funkciju `resched_curr` koja označi da trenutni zadatak mora prestati s izvođenjem.

Isječak koda 4.10. Funkcija `check_preempt_curr_new` (u kernel/sched/newsched.c)

```
static void check_preempt_curr_new(struct rq *rq, struct task_struct *p, int
flags) {
	printk(KERN_INFO "enter check_preempt_curr_new\n");
	if ((p->prio < rq->curr->prio) || (rq->curr->prio == p->prio && p->prio ==
			98 && p->nst.priority > rq->curr->nst.priority)) {
		//remove task from the list
		list_del_init(&rq->curr->nst.node);
		//increment priority
		if(rq->curr->nst.priority < 9){
			rq->curr->nst.priority++;
		}
		//add task to the end of the list
		list_add_tail(&rq->curr->nst.node, &rq→new_rq.sched_queue[rq→curr→
				nst.priority]);
```

```
            printk(KERN_INFO "task %d preempted\n", rq->curr->pid);
            //reschedul current runqueue
            resched_curr(rq);
      }
      printk(KERN_INFO "exit check_preempt_curr_new\n");
}
```

Funkcija `task_tick_new` se povremeno poziva i ima ulogu sata. Funkcija smanji `time_slice` zadatka i ako je veći od 0 izlazi iz funkcije. Ako je `time_slice` 0, ponovno postavi `time_slice` , makne zadatak iz liste, smanji mu prioritet i stavi ga u listu s novim prioritetom. Nakon toga pozove funkciju `resched_curr` da se drugi zadatak može izvoditi.

Isječak koda 4.11. Funkcija `task_tick_new` (u kernel/sched/newsched.c)

```
static void task_tick_new(struct rq *rq, struct task_struct *p, int queued) {
      printk(KERN_INFO "enter task_tick_new\n");
      update_curr_new(rq);
      //decrement time_slice and if it isn't 0 return
      if(--p->nst.time_slice) {
            printk(KERN_INFO "exit task_tick_new, time_slice decremented\n");
            return;
      }
      //if time_slice is 0 after decrement
      //reset time_slice (default RR time_slice is used)
      p->nst.time_slice = RR_TIMESLICE;
      //remove task from list with current priority
      list_del_init(&p->nst.node);
      //lower priority
      if(p->nst.priority > 0) {
            p→nst.priority--;
      }
      //add task to list with new priority
      list_add_tail(&p->nst.node, &rq->new_rq.sched_queue[p->nst.priority]);
      //reschedul runqueue
      resched_curr(rq);
      printk(KERN_INFO "exit task_tick_new, task rescheduled\n");
}
```

Funkcija `yield_task_new` dobrovoljno predaje izvođenje drugom zadatku. Funkcija uklanja trenutni zadatak iz liste, smanjuje prioritet i stavlja trenutni zadatak u listu s novim prioritetom.

Isječak koda 4.12. Funkcija `yield_task_new` (u kernel/sched/newsched.c)

```
static void yield_task_new(struct rq *rq) {
	printk(KERN_INFO "enter yield_task_new\n");
	//remove task from list with current priority
	list_del_init(&rq->curr->nst.node);
	//lower priority
	if(rq->curr->nst.priority > 0) {
		rq->curr->nst.priority--;
	}
	//add task to list with new priority
	list_add_tail(&rq->curr->nst.node, &rq->new_rq.sched_queue[rq->curr->
			nst.priority]);
	printk(KERN_INFO "exit yield_task_new\n");
	}
```

Funkcija `switched_to_new` poziva se kada se zadatak prebaci na SCHED_NEW. Funkcija provjerava je li se zadatak koji se prebacio trenutno izvodi, ako da nije potrebno ništa i vraća se. Ako se ne izvodi onda provjerava treba li prestati izvođenje trenutnog zadatka.

Isječak koda 4.13. Funkcija `switched_to_new` (u kernel/sched/newsched.c)

```
static void switched_to_new(struct rq *rq, struct task_struct *p) {
	printk(KERN_INFO "enter switched_to_new\n");
	//if we are running
	if(task_current(rq, p)) {
		printk(KERN_INFO "exit switched_to_new, we are running\n");
		return;
	}
	//if we are not running, we must check if current task needs to be
	preempted
	check_preempt_curr_new(rq, p, 0);
	printk(KERN_INFO "exit switched_to_new, we are not running\n");
}
```

Funkcija `prio_changed_new` se poziva kada se promjeni `prio`. Funkcija provjerava je li se zadatak mora prestati izvoditi ili ne.

Isječak koda 4.14. Funkcija `prio_changed_new` (u kernel/sched/newsched.c)

```
static void prio_changed_new(struct rq *rq, struct task_struct *p, int oldprio) {
	printk(KERN_INFO "enter prio_changed_new\n");
	//check if the task needs to be preempted
```

```
	check_preempt_curr_new(rq, p, 0);
	printk(KERN_INFO "exit prio_changed_new\n");
}
```

4.4. Omogućavanje korištenja raspoređivača

Nakon što je raspoređivač implementiran potrebno je nadodati kod u postojeće funkcije koji će omogućiti da se raspoređivač koristi.

U funkciji `__normal_prio` u datoteci kernel/sched/core.c u if uvjet za `rt_policy` potrebno je nadodati "`|| new_policy(policy)`". U slučaju da se to ne nadoda `prio` može biti bilo koja vrijednost jer `prio` nije inicijaliziran. Funkcija `new_policy` može se zamijeniti izrazom "`policy==SCHED_NEW`" jer funkcija samo provjerava vrijedi li taj uvjet.

Isječak koda 4.15. Izmijenjena funkcija `__normal_prio` (u kernel/sched/core.c)

```
static inline int __normal_prio(int policy, int rt_prio, int nice) {
...
else if (rt_policy(policy) || new_policy(policy))
	prio = MAX_RT_PRIO - 1 - rt_prio;
else
...
}
```

U funkciju `__setscheduler_prio` u datoteci kernel/sched/core.c potrebno je nadodati da se zadatku postavi adresa od `new_sched_class`.

Isječak koda 4.16. Izmijenjena funkcija `__setscheduler_prio` (u kernel/sched/core.c)

```
static void __setscheduler_prio(struct task_struct *p, int prio)
{
	...
#ifdef CONFIG_SCHED_NEW_POLICY
	} else if(prio == 98) {
		p->sched_class = &new_sched_class;
		printk(KERN_INFO "sched_class set to new_sched_class\n");
#endif
	} else if (rt_prio(prio)) {
	...
}
```

U funkciji `__pick_next_task` u datoteci kernel/sched/core.c postoji optimizacija za CFS koja stvara probleme kod poziva `pick_next_task` za novi raspoređivač. Na početku funkcije potrebno je dodati kod:

```
#ifdef CONFIG_SCHED_NEW_POLICY
```

```
	if(prev->policy == SCHED_NEW) {
		p = new_sched_class.pick_next_task(rq);
		if(p) {
			return p;
		}
	}
#endif
```

U funkciju `sched_init` u datoteci kernel/sched/core.c potrebno je nadodati poziv funkcije `init_new_rq` iz datoteke kernel/sched/newsched.c.

U funkciji `valid_policy` u datoteci kernel/sched/sched.h potrebno je dodati "`|| new_policy(policy)`".

Isječak koda 4.17. Izmijenjena funkcija `valid_policy` (u kernel/sched/sched.h)

```
static inline bool valid_policy(int policy) {
	return idle_policy(policy) || fair_policy(policy) || rt_policy(policy) ||
		dl_policy(policy) || new_policy(policy);
}
```

U datoteci include/uapi/linux/sched/types.h potrebno je izmijeniti strukture `sched_attr` i `sched_param` tako da sadrže prioritet zadatka novog raspoređivača. U obje strukture potrebno je dodati kod:

```
#ifdef CONFIG_SCHED_NEW_POLICY
	int new_sched_prio;
#endif
```

5. PREVOĐENJE JEZGRE LINUXA I KORIŠTENJE NOVOG RASPOREĐIVAČA

Izgradnja jezgre je složen postupak. Nakon promjena i dodataka prikazanih u prethodnom poglavlju, ovdje je detaljnije prikazan postupak prevođenja i pokretanja jezgre i primjer korištenja novog raspoređivača.

5.1. Izgradnja jezgre Linuxa

Prvo je potrebno generirati .config datoteku i staviti ju u korijenski direktorij izvornog koda jezgre Linuxa. Najjednostavniji način je kopirati .config datoteku od Linux distribucije koja se koristi.

Na Ubuntu Linuxu .config datoteka se kopira s naredbom:

```
$ cp /boot/config-`$(uname -r)' .config
```

Na Arch Linuxu .config datoteka se dobije naredbama:

```
$ cp /proc/config.gz ./

$ gunzip config.gz

$ mv config .config
```

Nakon dobivene .config datoteke potrebno je dodati nove konfiguracije u nju, to se radi s naredbom:

```
$ make oldconfig
```

Kod Ubuntua prilikom izgradnje jezgre se provjerava digitalni potpis, a u jezgru je dodan raspoređivač pa potpis neće vrijediti i neće željeti izgraditi jezgru. Kako bi se to zaobišlo potrebno je isključiti provjeru potpisa s naredbama:

```
$ scripts/config --disable SYSTEM_TRUSTED_KEYS

$ scripts/config --disable SYSTEM_REVOCATION_KEYS
```

Kod Archa je potrebno u .config datoteci promijeniti CONFIG_DEBUG_INFO_BTF=y u CONFIG_DEBUG_INFO_BTF=n i CONFIG_EXT4_FS=m u CONFIG_EXT4_FS=y. Kada je CONFIG_DEBUG_INFO_BTF postavljen na y onda prilikom izgradnje se javlja greška. Opcija CONFIG_EXT4_FS=m znači da se driver za ext4 datotečni sustav gradi kao zasebni modul koji se učitava prilikom pokretanja umjesto da se upakira u jezgru prilikom izgradnje (kao kad je postavljen na y). Ako se koristi ext4 datotečni sustav i ne promjeni se s m na y dolazi do paradoksa da ne može učitati driver za ext4 jer nema driver za ext4, a on mu je potreban za čitanje s diska.

Također je pametno u .config datoteci postaviti neko ime za CONFIG_LOCALVERSION da se lakše razlikuje od ostalih jezgri, ali nije obavezno.

Izgradnja jezgre se pokreće s naredbom:

```
$ make -j$(nproc)
```

Zastavica `-j` specificira koliko će se procesora koristiti za izgradnju jezgre, `$(nproc)` vraća broj procesora koliko ima računalo za najbrže moguće prevođenje.

Ako se dogodi greška tijekom izgradnje potrebno je pokrenuti naredbu:

```
$ make mrproper
```

Nakon nje potrebno je ponovno početi od generiranja .config datoteke.

Ako se izgradnja uspješno završila, potrebno je instalirati module s naredbom:

```
$ sudo make modules_install -j$(nproc)
```

Ako je za izgradnju korišten Ubuntu onda je potrebno instalirati jezgru s naredbom:

```
$ sudo make install
```

Izgradnja jezgre na Ubuntu Linuxu završava ovdje.

Ako je za prevođenje korišten Arch onda je potrebno instalirati jezgru s naredbom (gdje je localversion zamijenjen s imenom navedenim u CONFIG_LOCALVERSION):

```
$ sudo install -Dm644 `$(make -s image_name)' /boot/vmlinuz-6.6.9-localversion
```

Nakon toga potrebno je napraviti datoteku /etc/mkinitcpio.d/linux-localversion.preset i u nju je potrebno upisati:

```
ALL_config="/etc/mkinitcpio.conf"

ALL_kver="/boot/vmlinuz-6.6.9-localversion"

PRESETS=('default' 'fallback')

default_image="/boot/initramfs-6.6.9-localversion.img"

fallback_options="-S autodetect"
```

Nakon toga potrebno je generirati početni ram disk s naredbom:

```
$ sudo mkinitcpio -P
```

Nakon generiranja početnog ram diska potrebno je ažurirati bootloader, za GRUB bootloader potrebno je pokrenuti naredbu:

```
$ sudo grub-mkconfig -o /boot/grub/grub.cfg
```

5.2. Korištenje novog raspoređivača

Prije korištenja potrebno je dodati neke promjene u neke datoteke. Na Ubuntu Linuxu u datoteku /usr/include/x86_64-linux-gnu/bits/types/struct_sched_param.h potrebno je u strukturu `sched_param` nadodati:

```
int new_sched_prio;
```

Na Arch Linuxu u datoteku /usr/include/bits/types/struct_sched_param.h potrebno je u strukturu `sched_param` nadodati:

```
int new_sched_prio;
```

Raspoređivač se koristi tako da se promjeni preko funkcije `sched_setscheduler`. Parametri funkcije su: pid zadatka (za trenutni zadatak koristi se 0), način raspoređivanja (ali broj jer nije definiran makro, broj za SCHED_NEW je 7) i struktura `sched_param` koja sadrži parametre raspoređivača.

Isječak koda 5.1. Primjer korištenja novog raspoređivača

```
struct sched_param priority;
priority.sched_priority = 1;
priority.new_sched_prio = 5;
if(sched_setscheduler(0, 7, &priority)) {
      printf("Error: %s\n", strerror(errno));
      return 1;
}
```

6. ANALIZA RADA RASPOREĐIVAČA

Raspoređivač je ispitan sljedećim programom:

```
#include<sched.h>
#include<stdio.h>
#include<errno.h>
#include<string.h>
#include<unistd.h>
#include<sys/wait.h>

int main() {
	int n=3;
	printf("Entered program\n");
	int p=1;
	for(int i=0; i < n; i++) {
		if(p) {
			p = fork();
		}
	}
	if(!p) {
		struct sched_param priority;
		priority.sched_priority = 1;
		priority.new_sched_prio = 5;
		if(sched_setscheduler(0, 7, &priority)) {
			printf("Error: %s\n", strerror(errno));
			return 1;
		}
		printf("Policy changed\n");
		printf("using sched_policy: %d\n", sched_getscheduler(0));
		int j=0;
		for(; j < 10000; j++);
		printf("%d\n", j);
	}
	for(int i=0; i<n; i++) {
		wait(NULL);
	}
	return 0;
}
```

Program stvara tri procesa djeteta i čeka da završe s radom. Djeca promjene raspoređivač na novo implementirani raspoređivač, povećavaju `j` za 1 od 0 do 10000 i ispišu `j`.

Program se prevodi i pokreće naredbama:

```
$ gcc test.c

$ ./a.out
```

Nakon izvođenja provjeravaju se datoteke zapisnika naredbom:

```
$ journalctl -b 0 | grep new
```

U ispisu naredbe trebao bi ispisati ispis poruke koje ispisuje funkcija printk u funkcijama raspoređivača (primjer ispisa na slici 6.1.).

Raspoređivač ima minimalne funkcionalnosti, ostvareno je sve potrebno da bi radio, ali nema dodatne funkcionalnosti. Ostali raspoređivači koji već postoje u jezgri Linuxa imaju dodatne funkcionalnosti poput SMP-a ili raspoređivanja grupa zadataka. U buduće bi bilo dobro nadodati neke od tih funkcionalnosti.

Također postoje neke preinake koje bi bilo dobro dodati u algoritam da bolje radi. Na primjer da se kada zadatak sam prepusti izvođenje drugom zadatku, da mu se prioritet poveća po uzoru na CFS. Ako zadatak često prepušta izvođenje drugom zadatku on će se manje izvoditi u odnosu na druge zadatke pa bi trebao imati veću prednost.

Također jedna od stvari koje bi bilo dobro promijeniti je da se zadatak može započeti s politikom SCHED_NEW. U trenutnom ostvarenju se to ne može nego treba s funkcijom sched_setscheduler promijeniti politiku u SCHED_NEW. Mijenjanje politike oduzme određeno vrijeme prilikom kojeg se zadatak mogao izvoditi.

Prednost koju novi raspoređivač ima je složenost O(1) za dodavanje i micanje zadataka iz i u red pripravnih zadataka, za razliku od CFS-a koji ima složenost O(log n). Novi raspoređivač ima listu za red pripravnih zadataka, a uzimanje prvog zadatka iz liste i stavljanje na kraj liste imaju složenost O(1). CFS raspoređivač mora smjestiti zadatak u stablo pa zato ima složenost O(log n).

```
[user@host ~]$ journalctl -b 0 | grep new
Jun 12 21:04:27 host kernel: init_new_rq
Jun 12 21:04:27 host kernel: init_new_rq
Jun 12 21:04:27 host kernel: init_new_rq
Jun 12 21:04:27 host kernel: init_new_rq
Jun 12 21:04:27 host kernel: usbcore: registered new interface driver usbfs
Jun 12 21:04:27 host kernel: usbcore: registered new interface driver hub
Jun 12 21:04:27 host kernel: usbcore: registered new device driver usb
Jun 12 21:04:27 host kernel: usbcore: registered new interface driver usbserial_generic
Jun 12 21:04:27 host kernel: xhci_hcd 0000:03:00.0: new USB bus registered, assigned bus number
Jun 12 21:04:27 host kernel: xhci_hcd 0000:03:00.0: new USB bus registered, assigned bus number
Jun 12 21:04:27 host kernel: usb 1-1: new high-speed USB device number 2 using xhci_hcd
Jun 12 21:04:27 host kernel: usbcore: registered new interface driver usbhid
Jun 12 21:04:28 host NetworkManager[368]: <info>  [1718226268.6289] manager: (lo): new Loopback
Jun 12 21:04:28 host NetworkManager[368]: <info>  [1718226268.6318] manager: (ens1): new Etherne
Jun 12 21:05:28 host kernel: sched_class set to new_sched_class
Jun 12 21:05:28 host kernel: enter enqueue_task_new
Jun 12 21:05:28 host kernel: exit enqueue_task_new, task 753 enqueued
Jun 12 21:05:28 host kernel: new_rq[0]: 753
Jun 12 21:05:28 host kernel: set_next_task_new
Jun 12 21:05:28 host kernel: enter switched_to_new
Jun 12 21:05:28 host kernel: exit switched_to_new, we are running
Jun 12 21:05:28 host kernel: enter pick_next_task_new
Jun 12 21:05:28 host kernel: exit pick_next_task_new
Jun 12 21:05:28 host kernel: sched_class set to new_sched_class
Jun 12 21:05:28 host kernel: enter enqueue_task_new
Jun 12 21:05:28 host kernel: exit enqueue_task_new, task 754 enqueued
Jun 12 21:05:28 host kernel: new_rq[0]: 754
Jun 12 21:05:28 host kernel: set_next_task_new
Jun 12 21:05:28 host kernel: enter switched_to_new
Jun 12 21:05:28 host kernel: exit switched_to_new, we are running
Jun 12 21:05:28 host kernel: sched_class set to new_sched_class
Jun 12 21:05:28 host kernel: enter enqueue_task_new
Jun 12 21:05:28 host kernel: exit enqueue_task_new, task 755 enqueued
Jun 12 21:05:28 host kernel: new_rq[0]: 755
Jun 12 21:05:28 host kernel: set_next_task_new
Jun 12 21:05:28 host kernel: enter switched_to_new
Jun 12 21:05:28 host kernel: exit switched_to_new, we are running
Jun 12 21:05:28 host kernel: enter dequeue_task_new
Jun 12 21:05:28 host kernel: exit dequeue_task_new, task 753 dequeued
Jun 12 21:05:28 host kernel: enter pick_next_task_new
Jun 12 21:05:28 host kernel: exit pick_next_task_new, no task to run next
Jun 12 21:05:28 host kernel: put_prev_task_new
Jun 12 21:05:28 host kernel: enter pick_next_task_new
Jun 12 21:05:28 host kernel: exit pick_next_task_new
Jun 12 21:05:28 host kernel: enter dequeue_task_new
Jun 12 21:05:28 host kernel: exit dequeue_task_new, task 754 dequeued
Jun 12 21:05:28 host kernel: enter pick_next_task_new
Jun 12 21:05:28 host kernel: exit pick_next_task_new, no task to run next
Jun 12 21:05:28 host kernel: put_prev_task_new
Jun 12 21:05:28 host kernel: enter dequeue_task_new
Jun 12 21:05:28 host kernel: exit dequeue_task_new, task 755 dequeued
Jun 12 21:05:28 host kernel: enter pick_next_task_new
Jun 12 21:05:28 host kernel: exit pick_next_task_new, no task to run next
Jun 12 21:05:28 host kernel: put_prev_task_new
```

Slika 6.1. Ispis datoteke zapisnika

7. ZAKLJUČAK

U okviru ovog rada proučen je raspoređivač u jezgri operacijskog sustava Linux. Potom je osmišljen i ostvaren novi jednostavan raspoređivač. Raspoređivač je osmišljen s 10 prioriteta, svaki prioritet ima svoju listu u kojoj se spremaju zadaci. Zadaci se izvode određeni kvant vremena i nakon toga im se smanjuje prioritet. Ako je zadatak prekinut tako da mu dođe drugi zadatak većeg prioriteta, onda mu se prioritet povećava. Ako zadatak sam odustane od izvođenja, na primjer spava određeno vrijeme, onda mu se smanjuje prioritet. Raspoređivač je jako jednostavan u smislu da ima implementirano samo ono što mu je potrebno za rad za razliku od ostalih raspoređivača u jezgri Linuxa koji imaju neke optimizacije i funkcionalnosti koje omogućuju bolji i efikasniji rad. Moguća poboljšanja su nadogradnja i bolja optimizacija raspoređivača. Prednost raspoređivača u odnosu na CFS je složenost, novi raspoređivač ima složenost O(1), a CFS ima složenost O(log n).

8. LITERATURA

[1] *Implementing a new real-time scheduling policy for Linux: Part 1,* 26. 07. 2010., https://www.embedded.com/implementing-a-new-real-time-scheduling-policy-for-linux-part-1/ pristupljeno: 10. 03. 2024.

[2] *Implementing a new real-time scheduling policy for Linux: Part 2,* 27. 07. 2010., https://www.embedded.com/implementing-a-new-real-time-scheduling-policy-for-linux-part-2/ pristupljeno: 10. 03. 2024.

[3] *Implementing a new real-time scheduling policy for Linux: Part 3,* 28. 07. 2010., https://www.embedded.com/implementing-a-new-real-time-scheduling-policy-for-linux-part-3/ pristupljeno: 10. 03. 2024.

[4] Pratham Patel, *A Guide to Compiling the Linux Kernel All By Yourself,* 08. 02. 2024., https://itsfoss.com/compile-linux-kernel/ pristupljeno 10. 03. 2024.

[5] Mitchell Gouzenko, *Understanding sched_class,* 24. 11.2016., https://mgouzenko.github.-io/jekyll/update/2016/11/24/understanding-sched-class.html pristupljeno: 28. 03. 2024.

[6] Jinkyu Koo, *Linux kernel scheduler*, 05. 01. 2015., https://helix979.github.io/jkoo/post/os-scheduler/ pristupljeno: 28. 03. 2024.

[7] The kernel development community, *Deadline Task Scheduling*, 07. 03. 2020., https://www.kernel.org/doc/html/latest/scheduler/sched-deadline.html pristupljeno 07. 06. 2024.

[8] The kernel development community, *CFS Scheduler*, 05. 03. 2020., https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html pristupljeno 07. 06. 2024.

9. SAŽETAK

Naslov: Izgradnja novog raspoređivača poslova u Linuxu

U okviru ovog rada proučen je raspoređivač poslova jezgre Linuxa, osmišljen i ostvaren novi raspoređivač poslova. Objašnjen je način rada raspoređivača jezgre Linuxa, sve potrebne strukture, načini raspoređivanja i njihov međusobni odnos. Također su objašnjeni raspoređivači DL, RT i FAIR i njihovi načini rada. Opisan je koncept novog raspoređivača koji se dodaje u jezgru Linuxa i objašnjena je njegova implementacija uz izvorni kod. Opisan je postupak prevođenja jezgre Linuxa i potrebni koraci da se raspoređivač može koristiti. Dan je i objašnjen primjer koda kojim se raspoređivač testirao i dane su ideje za buduće poboljšanje raspoređivača.

Ključne riječi: raspoređivač poslova, jezgra Linuxa, deadline raspoređivač, RT raspoređivač, CFS raspoređivač

10. ABSTRACT

Title: Developing a new task scheduler in Linux kernel

In this thesis Linux kernel's task scheduler is examined and then a new task scheduler is implemented. It is explained how Linux kernel's task scheduler works, including all the necessary structures, scheduling policies and relations between them. Deadline, RT and FAIR schedulers are also explained as is explained how they work. The concept of the new scheduler that is being added in Linux kernel is described and its implementation and source code are explained. Linux kernel compilation process is also described and the process of setting up everything necessary for new scheduler to work. Source code of program that tests the new scheduler is shown and ideas for future improvement are given.

Key words: task scheduler, Linux kernel, deadline scheduler, RT scheduler, FAIR scheduler

11. PRIVITAK: LINK NA GITHUB REPOZITORIJ S IZVORNIM KODOM

Izvorni kod raspoređivača i jezgre u kojoj je implementiran nalazi se na github repozitoriju:

https://github.com/mihaelbh/adding-new-sched-policy-to-linux-6.6.9