Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matej Šekoranja

Razvrščevalniki procesov v operacijskih sistemih

diplomsko delo

univerzitetni študij računalništva in informatike

Mentor: prof. dr. Nikolaj Zimic

Ljubljana, 2016

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Moderni operacijski sistemi imajo jedro, ki skrbi za ustrezno razvrščanje procesov, ki so v čakalni vrsti. Razvrščanje je še posebej kritično v sistemih, kjer želimo hitro odzivnost sistema ter v sistemih v realnem času.
V diplomi iz dosegljivih virov pridobite podatke o delovanju razvrščevalnikov v operacijskih sistemih Linux, FreeBSD ter Windows. Poudarite predvsem posebnosti določenega operacijskega sistema.
V drugem delu izdelajte program, s katerim boste lahko testirali latenco operacijskega sistema Linux ter s tem pokazali na posebnosti tega operacijskega sistema.

Izjava o avtorstvu zaključnega dela

Spodaj podpisani Matej Šekoranja, vpisna številka 63000282, avtor zaključnega dela z naslovom:

Razvrščevalniki procesov v operacijskih sistemih (angl. *Process Schedulers in Operating Systems*)

IZJAVLJAM

1. da sem pisno zaključno delo študija izdelal samostojno pod mentorstvom prof. dr. Nikolaja Zimica;
2. da je tiskana oblika pisnega zaključnega dela študija istovetna elektronski obliki pisnega zaključnega dela študija;
3. da sem pridobil/-a vsa potrebna dovoljenja za uporabo podatkov in avtorskih del v pisnem zaključnem delu študija in jih v pisnem zaključnem delu študija jasno označil/-a;
4. da sem pri pripravi pisnega zaključnega dela študija ravnal/-a v skladu z etičnimi načeli in, kjer je to potrebno, za raziskavo pridobil/-a soglasje etične komisije;
5. soglašam, da se elektronska oblika pisnega zaključnega dela študija uporabi za preverjanje podobnosti vsebine z drugimi deli s programsko opremo za preverjanje podobnosti vsebine, ki je povezana s študijskim informacijskim sistemom članice;
6. da na UL neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve avtorskega dela v elektronski obliki, pravico reproduciranja ter pravico dajanja pisnega zaključnega dela študija na voljo javnosti na svetovnem spletu preko Repozitorija UL;
7. dovoljujem objavo svojih osebnih podatkov, ki so navedeni v pisnem zaključnem delu študija in tej izjavi, skupaj z objavo pisnega zaključnega dela študija.

V Ljubljani, dne 13. junija 2016 Podpis študenta/-ke:

Zahvaljujem se mentorju prof. dr. Nikolaju Zimicu za vodenje pri opravljanju diplomske naloge. Zahvala gre tudi ženi, družini in staršem za vso vzpodbudo in potrpljenje.

Daji, Liamu, Juretu in Živi.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
OS	operating system	operacijski sistem
CPE	central processing unit	centralno procesna enota
SMP	symmetric multiprocessing	simetrično multiprocesiranje
AMP	asymmetric multiprocessing	asimetrično multiprocesiranje
SMT	simultaneous multithreading	simultana večnitnost
UMA	uniform memory access	enak dostop do pomnilnika
NUMA	non-uniform memory access	neenak dostop do pomnilnika
IPI	inter-processor interrupt	medprocesorska prekinitev
V/I enota	input/output unit	vhodno/izhodna enota

Povzetek

Naslov: Razvrščevalniki procesov v operacijskih sistemih

Razvrščanje procesov je eno izmed ključnih opravil vsakega operacijskega sistema. Pravilno delovanje razvrščevalnika prestavlja ključen faktor odzivnosti sistema, predvsem pri procesih, ki zahtevajo delovanje v realnem času. Med slednje procese spada že predvajanje multimedijskih vsebin, ki dandanes prestavlja eno pogostejših opravil splošnonamenskega operacijskega sistema. V diplomski nalogi najprej predstavim teoretične osnove delovanja razvrščevalnikov: njihove naloge, načine razvrščanja in osnovne razvrščevalne algoritme. Obravnaval sem razvrščanje tako v enoprocesorskih kot tudi na večprocesorskih sistemih. Nadalje sem s podrobnim pregledom izvorne kode opisal delovanje razvrščevalnikov operacijskih sistemov *FreeBSD*, *Linux* in *Windows NT*[1]. Diplomo sem zaključil z meritvami latenc razvrščevalnikov na različnih jedrih operacijskega sistema Linux pri neobremenjenem in obremenjenem sistemu.

Ključne besede: razvrščevalnik, razvrščanje, operacijski sistem, proces, latenca, izvorna koda, implementacija.

[1]Dostopa do izvorne kode razvrščevalnika Windows NT nisem imel, raziskal sem razpoložljivo literaturo.

Abstract

Title: Operating System Process Schedulers

Process scheduling is one of the key tasks of every operating system. Proper implementation of a scheduler reflects itself in a system responsiveness, especially when processes require execution in real-time. Multimedia playback is one of these processes, also being one of the most common operating system tasks nowadays. In the beginning of this thesis, I present theoretical basics of scheduling: its goals, different scheduling types and basics algorithms. I cover scheduling in single-processor and multi-processor systems. The work continues with a detailed inspection of the source code and an explanation of internals of the following operating systems: *FreeBSD*, *Linux*, and *Windows NT*. In the end, I conduct measurement of scheduler latencies of different Linux kernels under un-loaded and loaded system conditions.

Keywords: scheduler, scheduling, operating system, process, latency, source code, implementation.

Poglavje 1

Uvod

Vsak moderni operacijski sistem podpira multiprogramiranje, kar pomeni, da je lahko v delovnem pomnilniku hkrati več procesov, ki si delijo – tekmujejo – za centralno procesno enoto (CPE) in ostale vire. Glavna naloga razvrščevalnika je razvrstiti izvajanje teh procesov, tako da čimbolje zagotovi:

Izkoriščenost (angl. *utilization*)
: Poskrbljeno mora biti, da so CPE in ostali viri čimbolje izkoriščeni.

Poštenost (angl. *fairness*)
: Poskrbljeno mora biti, da vsako je vsak proces deležen do 'pravičnega' deleža CPE.

Čimkrajši odzivni čas (angl. *low response time*)
: *Odzivni čas* je čas, ki preteče od trenutka, ko je podana zahteva po izvršitvi procesa, do trenutka, ko se sistem odzove nanj.
Ta metrika je ključnega pomena pri sistemih v realnem času – v kolikor je odzivni čas predlog, lahko to povzroči, da se kritičen proces ne izvrši v predvidenem času.

Čimkrajši obračalni čas (angl. *low turnaround time*)
: *Obračalni čas* je čas, ki preteče od trenutka, ko je podana zahteva po izvršitvi procesa, do trenutka, ko ga proces opravljen zapusti.
Ta metrika je pomembna pri paketni obdelavi (angl. *batch processing*).

Visoko prepustnost (angl. *high throughput*)
Prepustnost je definirana kot število opravljenih procesov na časovno enoto.

V sistemih za delo v realnem času, mora razvrščevalnik poskrbeti še za:

Pravočasnost (angl. *to meet the deadline*)
Kritični procesi morajo biti izvršeni v predvidenem času.

Predvidljivost (angl. *predictability*)
Zmogljivost in obnašanje sistema mora biti predvidljivo (izračunljivo, določljivo).

1.1 Razvrščanje v večprocesorskih sistemih

Večprocesorski sistemi so sestavljeni iz več CPE. Glede na sestavo večprocesorske sisteme delimo na dve glavni skupini:

Simetrično multiprocesiranje (SMP) (angl. *symmetric multiprocessing*)
Sistem je sestavljen iz več identičnih CPE. Sistem je lahko realiziran kot množica čipov (ena CPE na čip), ali kot več jeder (CPE) v enem čipu, ali pa kot kombinacija obeh možnosti (več čipov z več jedri).

Asimetrično multiprocesiranje (AMP) (angl. *asymmetric multiprocessing*)
Sistem vsebuje vsaj eno CPE, ki se razlikuje od ostalih. Taki sistemi so redki.

Moderne CPE podpirajo tudi simultano večnitnost (SMT) (angl. *simultaneous multithreading*), npr. Intelova tehnologija imenovana *Hyper-threading*. CPE je sestavljena iz več enot (kontrolna enota, aritmetično-logična enota, enota za računanje s števili z plavajočo vejico, registri, itd.) in med izvajanjem enega procesa so nekatere enote lahko neizkoriščene, npr. medtem ko

CPE čaka na podatke iz pomnilnika, bi lahko ostale enote izvajale drugo nit (proces). Ideja SMT je prav slednja - istočasno izvajanje več niti na račun deljenja različnih enot CPE. Za vsako nit mora CPE imeti ločeno t.i. arhitekturno stanje (angl. *architectural state*), ki je definirano kot množica splošnih in kontrolnih registrov. CPE vsebuje logiko, ki sama poskrbi za istočasno izvajanje niti - operacijski sistem vidi fizično CPE s podporo n nitim kot n logičnih CPE.

Glede na porazdelitev pomnilnika pa je večprocesorskih sistemov delitev sledeča:

Enak dostop do pomnilnika (UMA) (angl. *uniform memory access*)
Vse CPE enakovredno dostopajo do centraliziranega pomnilnika.

Neenak dostop do pomnilnika (NUMA) (angl. *non-uniform memory access*)
Sistem vsebuje več porazdeljenih pomnilnikov. Dostop do pomnilnika med CPE ni enakovreden (npr. dostop lokalnega pomnilnika je bistveno hitrejši kot do oddaljenega pomnilnika). Vse CPE vidijo vse pomnilnike kot celoto (delijo si isti naslovni prostor).

V tej diplomski nalogi se bom osredotočil le na UMA SMP sisteme (s podporo SMT) – centralizirane večprocesorske sisteme.

V SMP sistemih je naloga razvrščevalnika, da poskrbi za enakomerno izkoriščenosti vseh CPE v sistemu (angl. *CPU load-balancing*) s težnjo, da se en proces izvaja le na enem CPE (angl. *CPU affinity*) ter tako na račun boljše izkoriščenosti predpomnilnika izboljša zmogljivost sistema.

1.2 Značilnost izvajanja procesov

Glede na porabo virov lahko procese razdelimo v dve skupini:

- Procese, ki za svoje izvajanje potrebujejo predvsem CPE (angl. *CPU-bound processes*), npr. računsko intenzivni procesi.

- Procese, ki tekom svojega izvajanja predvsem vršijo operacije na vhodno/izhodnih (V/I) enotah, t.i. interaktivne procese (angl. *I/O-bound processes*).

Značilnost interaktivnih procesov je njihov impulziven 'karakter' – večino časa so v mirovanju, vendar pa to mirovanje prekinjajo kratkotrajni nemoteni teki ali 'izbruhi' (angl. *bursts*) porabe CPE. Vsak dober (splošnonamenski) razvrščevalnik bo izkoristil to lastnost procesov in glede na zgodovino izvajanja procesov identificiral interaktivne procese ter te izvajal pogosteje. Odzivnost sistema se zelo izboljša in tako s stališča uporabnikov sistem ostane še vedno 'uporaben' tudi pri večjih obremenitvah. Seveda se ne sme spregledati dejstva, da lahko procesi tudi spreminjajo svojo naravo izvajanja; dokaj pogost primer je kreiranje CPE intenzivnega procesa preko ukaznega interpreterja – novokreirani proces v večini operacijskih sistemov podeduje lastnosti procesa - očeta, ki pa je v tem primeru ukazni interpreter, torej izrazito interaktiven proces. Naloga razvrščevalnika je, da v čimkrajšem času zazna značajske spremembe procesov.

1.3 Razvrščanje brez ali s prekinjanjem

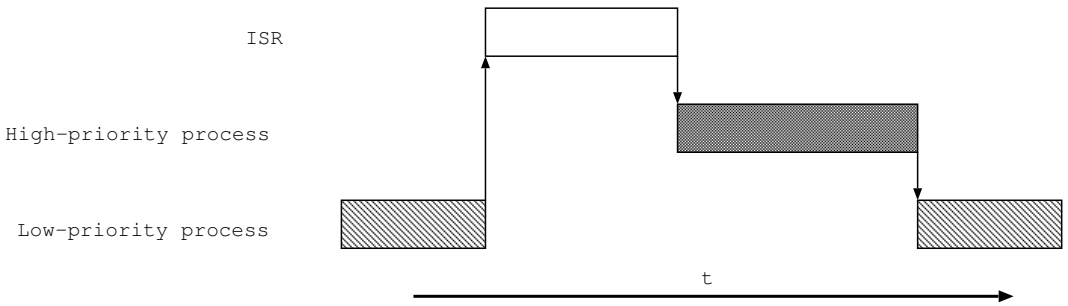
Glede na prekinjanje izvajanja tekočega procesa ločimo dva pristopa razvrščanja:

Razvrščanje brez prekinjanja (angl. *non-preemptive scheduling*)

Pri razvrščanju brez prekinjanja se izvajanje procesa ne prekine vse dokler se ta ne konča oziroma čaka na izvršitev V/I operacije ali signal. Izvajanje procesa lahko prekinejo strojne prekinitve, vendar se izvajanje procesa nadaljuje takoj po vrnitvi iz prekinitveno-servisnega programa (PSP)[1]. Proces lahko tudi prostovoljno preda CPE drugemu procesu (angl. *to yield the CPU*), npr. v primeru čakanja na izvršitev druge operacije.

Razvrščanje s prekinjanjem (angl. *preemptive scheduling*)

Pri razvrščanju s prekinjanjem se izvajanje procesa lahko prekine, čeprav bi nedokončan proces lahko nadaljeval z izvajanjem. Izvajanje procesa se prekine zaradi zahteve po izvajanju procesa z višjo prioriteto (glej sliko 1.1) ali ko procesu poteče dodeljena časovna rezina (angl. *time slice*).



Slika 1.1: Ponazoritev prekinitve izvajanja procesa z nižjo prioriteto.

Razvrščanje brez prekinjanja se praktično ne uporablja. Prvi razlog je dejstvo, da tako razvrščanje dopušča, da si en proces popolnoma prisvoji CPE, ali še huje, se ujame v mrtvi objem ter tako onemogoči izvajanje drugih

[1] angl. *interrupt service routine (ISR)*

procesov. Drugi razlog je slaba odzivnost sistema, saj se ta lahko odzove le, ko izvajajoči proces preda CPE.
Predvsem zaradi teh dveh razlogov se uporablja razvrščanje s prekinjanjem. Zelo pomembno je, da je možno procese, ki tečejo v uporabniškem načinu (angl. *user mode*), kadarkoli prekiniti, medtem ko za procese v zaščitenem načinu (angl. *protected mode*) obstajajo trenutki, ko se to ne sme, npr. med izvajanjem operacij razvrščevalnika, ki morajo biti izvedene atomarno.

Implementacija jedra, ki se da prekiniti (in ne čakati, da izvajanje sistemskega klica konča), (angl. *preemptive kernel*) zaradi dodatnih sinhronizacijskih problemov ni enostavna, zato obstajajo jedra, ki tega ne omogočajo, kar vodi do velikih odzivnih časov. Zaradi vse večjih zahtev, ki jih postavljajo multimedijske aplikacije, so taka jedra, ki se jih da prekiniti, zelo zaželena v modernih operacijskih sistemih in nuja v operacijskih sistemih za delo v realnem času.

Razvrščanje s prekinjanem lahko pri sinhronizaciji povzroči situacijo, kjer proces z višjo prioriteto čaka na sprostitev zaklepa prekinjenega procesa (-ov) z nižjo prioriteto, t. i. inverzija prioritet (angl. *priority inversion*). To lahko vodi do mrtvega objema[2] (ali do vsaj do dolgih odzivnih časov) – problem se reši s pomočjo *dedovanja prioritet* - procesi, ki imajo v lasti zaklep, podedujejo najvišjo prioriteto procesa, ki čaka na odklep.

[2]Najbolj znani primer tega problema je problem Nasinega raziskovalnega vozila 'Pathfinder' med misijo na Marsu leta 1997 – http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html

Poglavje 2

Razvrščevalni algoritmi

2.1 Osnovni razvrščevalni algoritmi

Namen tega poglavja je predstaviti osnovne razvrščevalne algoritme glede na njihove prednosti in slabosti.

Algoritem *First-Come, First-Served (FCFS)*

+ preprostost
+ pravičnost, brez stradanja[1] (angl. *starvation*)
– razvrščanje brez prekinjanja[2]
– kratka opravila čakajo na daljše

Algoritem *Shortest Job First (SJF)*

+ visoka prepustnost
+ majhen povprečni obračalni čas

[1] Pojav, kjer procesi čakajo neskončno dolgo oziroma se nikoli ne zvrstijo na CPE, imenujemo stradanje.

[2] V sistemih v realnem času je to zaželena lastnost.

– potreben je podatek o trajanju izvajanja procesa (napovedovanje)

– razvrščanje brez prekinjanja

– nepravičen do dolgih opravil (visoki obračalni časi dolgih procesov – možno stradanje)

Algoritem *Shortest Remaining Time First (SRTF)* (različica SJF z zmožnostjo prekinjanja)

\+ maksimalna prepustnost

\+ minimalen povprečni obračalni čas

\+ razvrščanje s prekinjanjem

– potreben je podatek o trajanju izvajanja procesa (napovedovanje)

– nepravičen do dolgih opravil (visoki obračalni časi dolgih procesov – možno stradanje)

Prioritetno razvrščanje

• razvrščanje z ali brez prekinjanja

\+ v primeru razvrščanja s prekinjanjem kratek odzivni čas (obratno sorazmeren s prioriteto)

\+ 'pravičnost' glede na prioriteto

\- možno stradanje procesov z nižjo prioriteto
Rešitev – staranje (angl. aging):

– zmanjšuj prioriteto procesa kot funkcijo časa izrabe CPE

– povečuj prioriteto procesa kot funkcijo časa čakanja na CPE

Algoritem *Round Robin (RR)*

+ najpravičnejši do vseh, brez stradanja
+ povprečen obračalni čas (vendar v primeru enako dolgih procesov zelo slab)
+ v povprečju dober odzivni čas (spremenljiv vendar omejen)
+ razvrščanje s prekinjanjem
– malenkost slabša zmogljivost zaradi preklapljanja med procesi

Algoritem *Multi-Level Feedback Queue (MLFQ)*

Vsak od zgoraj naštetih algoritmov ima svoje slabosti in prednosti. Prav tako tudi en sam algoritem ni primeren za vsa opravila v sistemu. Zato se uporablja kombinacija algoritmov z uporabo več vrst. Vsaka od teh vrst ima svojo prioriteto in svoj razvrščevalni algoritem. Obstajati mora tudi mehanizem za razvrščanje med vrstami samimi, lahko pa tudi za premeščanje procesov med vrstami. Takim algoritmom pravimo *multi-level queue scheduling algoritms*.

Eden izmed teh algoritmov, ki tudi omogoča premoščanje procesov med vrstami, je tudi *multi-level feedback queue (MLFQ)* algoritem, ki se uporablja v večini UNIX operacijskih sistemov. Za razvrščanje med vrstami se uporablja razvrščanje s prioritetami, medtem ko se za premeščanje procesov med vrstami uporabita naslednji pravili:

- Če proces porabi preveč časa CPE, ga premakni v vrsto z nižjo prioriteto. Tako se doseže, da interaktivni procesi ostanejo v vrstah z višjo prioriteto.
- Če proces čaka preveč časa, ga premakni v vrsto z višjo prioriteto.
  To pravilo preprečuje stradanje procesov in promovira interaktivne procese.

2.2 Razvrščanje v realnem času

Razvrščanje v sistemih za delo v realnem času (angl. *real-time operating systems (RTOS)*) se zelo razlikuje od razvrščanja v splošnonamenskih sistemih. V RTOS je vse podrejeno cilju po čimvečji zanesljivosti, predvidljivosti, čimkrajšim odzivnim časom in predvsem izvršitvi opravil v predvidenem času. Poudariti je potrebno, da algoritmi za razvrščanje v trdem realnem času potrebujejo matematičen dokaz o pravilnem delovanju in ne dajejo 'le' statistično dobre rezultate, kar pa je dovolj za razvrščanje v mehkem realnem času.

Za razvrščanje v RT se uporablja prioritetno razvrščanje – vedno se izvaja proces z najvišjo prioriteto. Ločimo jih v dve skupini:

Algoritmi s statičnim prirejanjem prioritet
: Prioriteta je določena vnaprej, a priori, in se ne spreminja med izvajanjem procesa.

Algoritmi z dinamičnim prirejanjem prioritet
: Prioriteta se določi in spreminja med izvajanjem procesa.

V RTOS se procese loči na *periodične* in *aperiodične*. Ta delitev je predvsem pomembna zaradi naključnega prihoda aperiodičnih procesov – delovanje RTOS mora biti predvidljivo, torej prihod zahtev deterministično, kar pa za aperiodične procese ne velja. Problem aperiodičnih procesov se rešuje s prevedbo na periodične procese (na CPE se rezervira čas za navidezen periodičen proces z dovolj visoko prioriteto).

Algoritmi, ki se uporabljajo v RTOS sistemih, vse bolj prodirajo v splošnonamenske sisteme, saj so zahteve multimedijskih aplikacij vse bližje zahtevam delovanja v mehkem realnem času (angl. *soft real-time*), zato se bom v nadaljevanju malce bolj posvetil dvema glavnima predstavnikoma algoritmov v RTOS.

Algoritem *Rate Monotonic (RM)* Algoritem *rate monotonic (RM)* je optimalen algoritem s statičnim prirejanjem prioritet. Prioritete so določene

glede na frekvenco procesov – procesi z višjo frekvenco imajo večjo prioriteto. Algoritem je optimalen v smislu, da ne obstaja drug algoritem s statičnim prirejanjem prioritet, ki bi znal za določene časovne omejitve najti razpored procesov, ki jih RM algoritem ne bi znal.

Vsa nadaljnja analiza algoritma temelji na naslednjih predpostavkah:

- Sistem z eno CPE.
- Čas preklopa je idealen, torej 0.
- Vsi procesi so periodični.
- Rok izvršitve je določen kar s periodo procesa.
- Procesi so med seboj neodvisni – med njimi ni sinhronizacije.
- Najdaljši čas procesiranja procesa je konstanten.

Glavna slabost RM algoritma je, da obremenitev (zasedenost) CPE pri kateri algoritem zagotavlja izvršitev opravil v roku ni 100%.
Definirajmo obremenitev CPE U pri n procesih kot:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \ , \tag{2.1}$$

kjer je C_i najdaljši čas procesiranja in T_i perioda procesa i.

Za RM algoritem velja naslednje:

Izrek 2.1 *Za množico n procesov urejenih s statičnimi prioritetami, je najvišja obremenitev CPE U_{max}, ki še zagotavlja izvršitev vseh opravil v roku [1]:*

$$U_{max} = n \cdot (2^{\frac{1}{n}} – 1). \tag{2.2}$$

Torej, v kolikor U (2.1) $\leq U_{max}$ (2.2) potem RM algoritem zagotavlja izvršitev vseh opravil v določenem roku.

Za dva procesa je mejna obremenitev $U_{max} \approx 0.86$, v primeru velikih n:

$$U_{max} = \lim_{n \to \infty} n \cdot (2^{\frac{1}{n}} - 1) = \ln 2 \approx 0.693. \tag{2.3}$$

Kljub slabši izkoriščenosti CPE, je RM algoritem eden najpogosteje uporabljenih algoritmov v komercialnih RTOS, saj je izjemo preprost, hiter in v primeru preobremenitve CPE zagotavlja, da bodo neizvršeni procesi ravno procesi z najnižjo prioriteto.

Earliest Deadline First (EDF)

Earliest Deadline First (EDF) algoritem je algoritem z dinamičnim dodeljevanjem prioritet – proces z najbližjim rokom izvršitve ima vedno najvišjo prioriteto (to velja v vsakem trenutku). Ker se prioritete izračunavajo dinamično, je EDF algoritem zahtevnejši od RM algoritma, vendar ima to lastnost, da je obremenitev CPE pri kateri algoritem zagotavlja izvršitev opravil v roku $U_{max} = 1$:

Izrek 2.2 *Za množico n procesov urejenih glede na rok izvršitve, algoritem zagotavlja izvršitev opravil v roku le in samo če [2]:*

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1. \tag{2.4}$$

Slabost EDF algoritma je ta, da v primeru preobremenitve CPE ni predvidljiv – ne moramo določiti, kateri procesi bodo prekoračili rok izvršitve (torej ne le procesi nižjo prioriteto).

Poglavje 3

Opis razvrščevalnikov

3.1 FreeBSD

FreeBSD je odprtokodni operacijski sistem, ki temelji na osnovi 4.4BSD-Lite. Je tipa UNIX in je bil razvit na kalifornijski univerzi v Berkeleyju. FreeBSD je predvsem namenjen kot operacijski sistem mrežnih strežnikov.
FreeBSD (verzija 5.1) ponuja dva različna razvrščevalnika.

3.1.1 Izboljšani razvrščevalnik 4.3BSD

FreeBSD razvrščevalnik temelji na 4.3BSD razvrščevalniku, ki interaktivnim procesom zagotavlja dobre odzivne čase tudi pri obremenjenemu sistemu. Razvrščevalnik je prioritetni razvrščevalnik s prekinjanjem – uporablja se round robin mehanizem. Privzeta časovna rezina je 100ms. Razvrščevalniku je bil dodan koncept razvrščevalnih razredov (angl. *scheduling classes*) in osnovna podpora SMP.
Že v začetku razvoja FreeBSD sta se edinemu razvrščevalnemu razredu *time sharing* dodala še dva razreda, *real-time* in *idle*. Procesi prioritete[1] *idle* se izvajajo samo, če ni v pripravljenosti nobenega procesa prioritete

[1] Razred je definiran kot fiksen interval prioritetnega prostora.

real-time ali *time sharing*, medtem ko se procesi prioritete *real-time* izvajajo vse dokler prostovoljno ne predajo CPE oziroma jih ne prekine proces z višjo prioriteto. S pojavitvijo podpore SMP se je dodal že razred *interrupt*, ki je ekvivalenten razredu *real-time*, le da ima višjo prioriteto[2]. *Time sharing* razred je edini razred, kateremu se prioritete določajo glede na porabo CPE časa – procesom z večjo porabo CPE časa se zmanjšuje prioriteta in s tem favorizira interaktivne procese. Izračun prioritet temelji na porabi CPE časa in *nice* vrednosti procesa:

$$priority = PUSER + \frac{estcpu}{8} + nice, \tag{3.1}$$

kjer je $PUSER$ najvišja prioriteta *time sharing* razreda in je definirana kot konstanta z vrednostjo 160 ter *nice* spremenljivka, ki jo določi uporabnik, na intervalu $[-20, 20]$ s privzeto vrednostjo 0.
Poraba CPE časa je določena s številom prožitev ure (angl. *ticks*) v času izvajanja procesa, t. i. *estcpu*. Ta vrednost se enkrat na sekundo zmanjša (angl. *is decayed*) glede na trenutno obremenjenost sistema – pri večji obremenjenosti počasneje. V kolikor se to ne bi počelo, bi vse prioritete procesov konverigrale proti višjim vrednostim (nižjim prioritetam), hitrost konvergence pa bi bila odvisna od porabe CPE časa. Vrednost *estcpu* se preračuna tudi, ko se proces vrne v stanje pripravljenosti. Za zmanjševanje *estcpu* se uporablja naslednje pravilo:

$$\textit{V času } 5 \cdot loadavg \textit{ sekund zmanjšaj estcpu za 90\%},$$

kjer je *loadavg* drseče povprečje procesov[3], ki se izvajajo oziroma so v čakalni vrsti za izvajanje. To ustreza naslednjemu matematičnemu izrazu:

$$decay^{5*loadavg} \approx 0.1,$$

kjer je $(1 - decay)$ faktor za koliko se zmanjša vrednost *estcpu* v eni sekundi. Ker mora biti razvrščevalnik hiter, se za izračun zgornjega izraza

[2]Višja prioriteta pomeni nižjo številčno vrednost.
[3]Povprečje *loadavg* se računa vsakih 5 sekund.

uporablja približek

$$decay = \frac{2 \cdot loadavg}{2 \cdot loadavg + 1}. \tag{3.2}$$

Za dokaz korektnosti izraza je potrebno dokazati:

$$factor^{5 \cdot loadavg} \approx 0.1 \Rightarrow factor = \frac{2 \cdot loadavg}{2 \cdot loadavg + 1} \tag{3.3a}$$

$$\frac{2 \cdot loadavg}{2 \cdot loadavg + 1}^{power} \approx 0.1 \Rightarrow power = 5 \cdot loadavg. \tag{3.3b}$$

Zaradi lepše izpeljave vpeljimo novo spremenljivko $b = 2 \cdot loadavg$ in zato je $decay = \frac{b}{b+1}$. Pri dokazu bomo uporabili tudi naslednja približka.
Za $x \approx 0$ velja

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \approx 1 + x,$$

torej je

$$e^{-\frac{1}{b}} \approx 1 - \frac{1}{b} = \frac{b-1}{b}. \tag{3.4}$$

Ter zopet za $x \approx 0$ je

$$\ln(x+1) = x - \frac{x^2}{2} + \frac{x^3}{3} - \ldots \approx x$$

in zato velja tudi

$$\ln \frac{b}{b+1} = \ln(1 - \frac{1}{b+1}) \approx -\frac{1}{b+1}. \tag{3.5}$$

Uporabili bomo tudi izračun $\ln 0.1 \approx -2.30$.
Sedaj lahko dokažimo (3.3a):

$$\begin{aligned} factor^{5 \cdot loadavg} &\approx 0.1 \\ \ln factor &\approx \frac{-2.30}{5 \cdot loadavg} \\ factor &\approx e^{-\frac{1}{\frac{5}{2.30} \cdot loadavg}} \\ &\approx e^{-\frac{1}{2 \cdot loadavg}} = e^{-\frac{1}{b}}. \end{aligned}$$

Uporabimo zvezo (3.4) in za velike b izpeljemo:

$$e^{-\frac{1}{b}} \approx \frac{b-1}{b} \approx \frac{b}{b+1} = \frac{2 \cdot loadavg}{2 \cdot loadavg + 1}.$$

Dokažimo še (3.3b):

$$\frac{b}{b+1}^{power} \approx 0.1$$
$$power \cdot \ln \frac{b}{b+1} \approx -2.30$$

Uporabimo zvezo (3.5) in poenostavimo

$$power \approx 2.30 \cdot (b+1) = 4.60 \cdot loadavg + 2.30 \approx 5 \cdot loadavg.$$

Izraz (3.2) je torej korekten.
Prav računanje *estcpu* je razlog počasnosti razvrščanja pri velikem številu procesov – algoritem je reda $O(n)$.
Za svoje delovanje razvrščevalnik potrebuje tri časovnike (angl. *timers*). Prvi časovnik je sistemska ura za statistične obdelave (angl. *statistics clock*) znana kot `statclock` (glej `FreeBSD/src/sys/kern/kern_clock.c:434`). Frekvenca ure se lahko dobi s pomočjo naslednjega ukaza:

```
# sysctl -d kern.clockrate; sysctl kern.clockrate
kern.clockrate: Rate and period of various kernel clocks
kern.clockrate: { hz = 100, tick = 10000, profhz = 1024, stathz = 128 }
```

`kern.clockrate` je tipa `CTLTYPE_STRUCT|CTLFLAG_RD`, torej je struktura, ki je samo bralne narave. Natančnejši pregled kode za platformo `i386` razkrije, da je frekvenca fiksna:

```
FreeBSD/src/sys/isa/rtc.h(72): #define RTC_NOPROFRATE  128
FreeBSD/src/sys/i386/isa/clock.c(934): stathz = RTC_NOPROFRATE;
```

Frekvenca 128Hz ni zanemarljiva, zato morala biti računanje prioritet kar se da hitro. Relevantna implementacija procedure `FreeBSD/src/sys/kern/sched_4bsd.c(448): sched_clock` je sledeča:

```
    kg->kg_estcpu = ESTCPULIM(kg->kg_estcpu + 1);
    if ((kg->kg_estcpu % INVERSE_ESTCPU_WEIGHT) == 0) {
        resetpriority(kg);
```

```
        if (td->td_priority >= PUSER)
            td->td_priority = kg->kg_user_pri;
    }
```

Koda je precej enostavna: povečaj `kg->kg_estcpu` za eno ter omeji z zgornjo mejo, ki jo dopušča makro `ESTCPULIM`. V kolikor se je `kg->kg_estcpu` povečal za število, ki je ekvivalentno povečavi prioritete za enoto (`INVERSE_ESTCPU_WEIGHT = 8`), kliči metodo `resetpriority`. Po vrnitvi iz metode vrednostno omeji prioriteto z vrednostjo `PUSER` (večja vrednost pomeni nižjo prioriteto).
Drugi časovnik se proži s frekvenco 1Hz in služi zmanjševanju vrednosti porabe CPE, torej *estcpu*. Relevantna implementacija procedure `FreeBSD/src/sys/kern/sched_4bsd.c(247): schedcpu` je sledeča (poenostavljena in preurejena zaradi nazornosti):

```
    FOREACH_CPU_IN_SYSTEM(p) {
        FOREACH_KSEGRP_IN_CPU(p, kg) {
            kg->kg_estcpu = decay_cpu(loadfac, kg->kg_estcpu);
            resetpriority(kg);
            FOREACH_THREAD_IN_GROUP(kg, td) {
                if (td->td_priority >= PUSER) {
                    sched_prio(td, kg->kg_user_pri);
                }
            }
        }
    }
```

Koda je zelo podobna zgornji kodi, le da se tukaj `kg->kg_estcpu` eksponentno zmanjšuje z manjšo frekvenco. Algoritem je teoretično reda $O(n^3)$, vendar ga lahko s praktičnega stališča (en `KSEGRP` na CPE, en proces v `KSEGRP`; kjer `KSEGRP` prestavlja množico procesov, na katere naj razvrščevalnik gleda kot celoto) gledamo nanj kot algoritem reda $O(n)$. Zopet je uporabljena procedura `resetpriority`.

Procedura `resetpriority` je ena pogosto klicanih procedur – poleg zgoraj omenjenih klicev se kliče v primeru spremembe vrednosti *nice* ter ob vsakem prehodu procesa v stanje pripravljenosti (angl. *wakeup*) – preračunavanje prioritete procesov, ki spijo (blokirajo) dlje kot eno sekundo, se ne vrši več, vendar se to 'nadoknadi', ko se proces vrne v stanje pripravljenosti (glej `FreeBSD/src/sys/kern/sched_4bsd.c(360): updatepri(struct ksegrp *kg)`). Zato ima procedura `schedcpu` se dodatno nalogo – šteti število sekund, ko proces spi. Naloga procedure `resetpriority` je preračunati prioriteto po izrazu (3.1).
Tretji časovnik je round-robin časovnik. Privzeta frekvenca proženja je 10Hz, vendar je ta nastavljiva – obstaja sistemska spremenljivka `kern.quantum`, tipa `CTLTYPE_INT|CTLFLAG_RW`, torej bralno-pisalna celoštevilčna spremenljivka.

```
# sysctl -d kern.quantum; sysctl kern.quantum
kern.quantum: Roundrobin scheduling quantum in microseconds
kern.quantum = 100000
```

Klicana procedura `FreeBSD/src/sys/kern/sched_4bsd.c(143): roundrobin` je siloma enostavna, saj je za enoprocesorske sisteme z izjemo ponastavitve časovnika prazna. Preklop okolja se izvrši v okviru prekinitve, ki jo povzroči časovnik – (angl. *process preemptation*).
FreeBSD ima še eno posebnost glede *nice* vrednosti – velja namreč pravilo: v kolikor je *nice* vrednost procesa za 20 višja od 'najmanj prijaznega' procesa, potem se izvajanje tega procesa ne dovoli.

Podpora večprocesorskim sistemom

Podpora SMP je zgolj osnovna: razvrščevalnik sicer bo izvajal procese na vseh CPE, vendar je to tudi vse kar zmore – je brez kakršnekoli podpore afinitete CPE kot tudi razlikovanja med različnimi tipi procesorskih sistemov (na SMT gleda kot na SMP).

Vse CPE izvajajo kodo razvrščevalnika in si delijo njegove podatkovne strukture, ki so temu primerno sinhronizirane. Praktično edina vidna koda razvrščevalnika, ki kaže na podporo SMP, je zgoraj omenjena procedura `FreeBSD/src/sys/kern/sched_4bsd.c(143): roundrobin`:

```
static void
roundrobin(void *arg)
{

#ifdef SMP
    mtx_lock_spin(&sched_lock);
    forward_roundrobin();
    mtx_unlock_spin(&sched_lock);
#endif

    callout_reset(&roundrobin_callout, sched_quantum,
                  roundrobin, NULL);
}
```

Round-robin časovnik s pomočjo medprocesorske prekinitve (angl. *inter-processor interrupt preemptation (IPI)*), ki se kliče v proceduri $forward\_roundrobin$, vsem ostalim procesorjem sporoči, da je čas za potencialni preklop.

3.1.2 Razvrščevalnik ULE

Razvrščevalnik ULE[4] je najnovejši razvrščevalnik za FreeBSD. Nastal je predvsem z namenom, da izboljša izkoriščenost sistema na SMP sistemih in z uporabo algoritmov časovne kompleksnosti $O(1)$ izboljša tudi delovanje pri velikem številu procesov. Veliko truda je bilo tudi vloženega v algoritem za hitro identifikacijo interaktivnih procesov in minimizacijo odzivnih časov teh procesov. Nastal je kot rezultat študije modernejših razvrščevalnikov, predvsem novega Linux $O(1)$ razvrščevalnika (glej 3.2.2). Vsaki CPE se dodeli podatkovna struktura imenovana `kseq` *(kernel scheduler entity queue)*. Struktura vsebuje tri prioritetne vrste: prvi dve se uporabljata za *interrupt*, *real-time* in *time sharing* razvrščevalne razrede, medtem ko je tretja vrsta namenjena *idle* razredu. ULE je dogodkovno voden razvrščevalnik, torej ne potrebuje časovnika za preračunavanje prioritet procesov, da bi preprečil njihovo stradanje. Poštenost je dosežena s prvima dvema vrstama – *aktivno* in *neaktivno* (angl. *current and next*). Razvrščevalnik za izvajanje izbere proces z najvišjo prioriteto iz aktivne vrste. Proces je med izvajanjem odstranjen iz vrste in ne pripada nobeni vrsti. Ko se njegovo izvajanje prekine, se proces glede na njegove lastnosti doda nazaj v aktivno ali neaktivno vrsto. Nato razvrščevalnik ponovno izbere proces z najvišjo prioriteto iz aktivne vrste. Ko se aktivna vrsta izprazni, si vrsti zamenjata vlogi – aktivna postane neaktivna ter neaktivna postane aktivna. Ta postopek zagotavlja, da bo vsak proces dobil priložnost izvajanja na CPE ne glede na njegovo prioriteto – na ta način nimamo stradanja med procesi.

Prioriteta, dolžina časovne rezine in stopnja interaktivnosti se izračunajo, ko procesu poteče celotna časovna rezina. Če se izkaže, da je proces interaktiven, se ga zopet dodeli aktivni vrsti, drugače pa v neaktivni vrsti. Z dodeljevanjem interaktivnih procesov aktivni vrsti in dodeljevanjem višjih

[4]Zakaj ime 'ULE'? Odgovor avtorja: 'You are the first person that has contacted me regarding ULE who has understood the meaning. Yes, it's just schedULE. I had many different names for it, each of which was inadequate for one reason or another.'

prioritet dosežemo nizke odzivne čase interaktivnih procesov.
Procesi razredov *interrupt* in *real-time* se vedno dodelijo aktivni vrsti, kot tudi procesom, katerih prioriteta je enaka ali višja prioritetam teh dveh razredov.
Procesi razreda *idle* imajo svojo vrsto. Procese iz te vrste se izvršuje le, če ni drugih pripravljenih procesov. *Idle* procesi so vedno dodeljeni tej vrsti, razen, če njihova prioriteta ne doseže prioritete razreda *time sharing*.
Določanje interaktivnosti procesa je ena izmed pomembnejših nalog ULE razvrščevalnika. Interaktivnost se določi s pomočjo razmerja časov 'prostovoljnega' spanja (blokiranja) in izvajanja. Za interaktivne procese je značilno, da so časi prostovoljnega spanja večji od časa izvajanja. Čas prostovoljnega spanja se uporablja zaradi natančnejšega predvidevanja značilnosti procesa, saj lahko z upoštevanjem neprostovoljnega spanja označili CPE zahtevne procese, ki so pri veliki obremenjenosti sistema čakali na CPE, označili kot interaktivne.
Čas prostovoljnega spanja je določen s številom urinih prožitev (angl. *ticks*) med klicema `sleep` in `wakeup` oziroma časa pretečenega do izpolnitve pogoja (angl. *condition variable*). Čas izvajanja je tudi določen s številom urinih prožitev med izvajanjem procesa.
Stopnja interaktivnosti (angl. *interactivity score*) se določi s pomočjo naslednjega izraza:

$$m = \frac{maximum\_score}{2},$$

$$score = \begin{cases} \frac{m}{\frac{sleep}{run}}, & sleep > run \\ m + \frac{m}{\frac{run}{sleep}}, & \text{otherwise} \end{cases}$$

kjer spremenljivka *maximum_score* predstavlja najvišjo številčno vrednost stopnje interaktivnosti, spremenljivka *sleep* predstavlja čas prostovoljnega spanja in spremenljivka *run* čas izvajanja procesa. Vrednosti spremenljivk ne naraščata v nedogled – ko vsota spremenljivk doseže določeno zgornjo mejo, se proporcionalno zmanjšata za določen faktor. Tako se ohrani del zgodovine izvajanja procesa, vendar je pri tem potrebno paziti, da ni faktor

prevelik, saj mora biti razvrščevalnik sposoben čimprej zaznati značajske spremembe procesov.
Glede na izračunano stopnjo interaktivnosti razvrščevalnik določi, ali je proces interaktiven in posledično, ali naj ga priredi aktivni vrsti – zato je določitev mejne vrednosti (angl. *threshold*) stopnje interaktivnosti med interaktivnimi in ne-interativnimi procesi ključnega pomena.
Procesom razreda *time sharing* se izračuna tudi prioriteta, ki izključno določa samo vrstni red izvajanja procesov. Za izračun prioritet se uporablja naslednji izraz:

$$priority = PUSER + \frac{score \cdot (PUSER\_MAX - PUSER)}{PUSER\_MAX} + nice. \quad (3.6)$$

kjer je $PUSER\_MAX$ najnižja prioriteta *time sharing* razreda in je definirana kot konstanta z vrednostjo 223.
Prioritete drugih razredov so določene statično.
ULE razvrščevalnik uporablja dinamične dolžine časovnih rezin. Minimalna dolžina je $\text{slice}_{\text{min}} = \frac{1}{100}\text{Hz} = 10ms$, maksimalna pa
$\text{slice}_{\text{max}} = \frac{1}{7}\text{Hz} \approx 140ms$. Kriterija za določitev dolžine sta interaktivnost procesa in *nice* vrednost. V kolikor je proces interaktiven, se mu dodeli minimalna časovna rezina. To razvrščevalniku omogoča, da hitreje odkritje dejstvo, da proces ni več interaktiven in tako ne pusti procesu, da izkoristi svojo prednost na račun zgodovine izvajanja. Izračun dolžine časovne rezine je naslednji (v ms):

$$slice = \begin{cases} 0, & (nice - nice_{lnp}) > 20 \\ \text{slice}_{\text{min}}, & \text{interactive process} \\ \text{slice}_{\text{max}}, & \text{least nice process} \\ \text{slice}_{\text{max}} - \frac{2 \cdot (nice - nice_{lnp})}{40} * (\text{slice}_{\text{max}} - \text{slice}_{\text{min}}), & \text{otherwise} \end{cases}$$

kjer $nice_{lnp}$ predstavlja *nice* vrednost procesa z najnižjo *nice* vrednostjo (angl. *least nice process*).
Določanje časovne rezine procesom razreda *interrupt* in *real-time* je nesmiselno, saj se ti izvajajo dokler prostovoljno ne predajo CPE oziroma dokler jih ne prekine proces z višjo prioriteto. Enako velja za procese

prioritete *idle*, ki se izvajajo vse dokler ni v sistemu procesov, ki čakajo na izvajanje.

3.1.3 Posodobitve razvrščevalnika ULE

Razvrščevalnik je tekom let prejel dve večji posodobitvi. V prvi (verzija 2) se je koncept aktivne in neaktivne vrste zamenjal s krožnim medpomnilnikom (angl. *circular buffer*), sam princip delovanja razvrščevalnika pa je ostal enak. V drugi (verzija 3) posodobitvi pa se je izboljšala podpora SMP (in SMT) - samo jedro operacijskega sistema FreeBSD je bilo z verzijo 7 prepisano tako, da je v celoti podprlo SMP, in ULE je postal privzeti razvrščevalnik.

Podpora večprocesorskim sistemom

Glavni cilj ULE razvrščevalnika na SMP sistemih je preprečiti nepotrebno migriranje izvajanja procesov iz ene CPE na druge in tako povečati afiniteto izvajanja procesa na eni CPE. Kljub temu cilju pa je potrebno poskrbeti za čimvečjo izkoriščenost vseh CPE v sistemu.
Vsaki CPE se dodeli podatkovna struktura *kseq*, ki vsebuje vse podatke, ki jih razvrščevalnik potrebuje za izvajanje procesov na eni CPE. Struktura vsebuje tudi seznam procesov (torej aktivno in neaktivno vrsto), ki so dodeljeni eni CPE - torej vsaka CPE ima svoj seznam procesov. Tako se doseže afiniteta izvajanja procesov na eni CPE.
Za enakomerno porazdelitev izkoriščenosti CPE (angl. *CPU load-balancing*) razvrščevalnik uporablja 2 mehanizma:

Mehanizem *pull*

Ko CPE zmanjka procesov za izvajanje (ali so vsi procesi končali z izvajanjem, ali pa vsi spijo), se poišče najbolj obremenjeno CPE ter iz nje prenese (migrira) proces z najvišjo prioriteto na prosto CPE.

Mehanizem *push*

Mehanizem *push* je glede porazdelitve izkoriščenosti bolj agresiven in se izvaja periodično. Dvakrat na sekundo se kliče procedure `sched_balance()`, ki iz najbolj obremenjene CPE prenese nekaj procesov na najmanj obremenjeno CPE. Prenašajo se procesi z najvišjo prioriteto, vse dokler se število procesov na obeh CPE ne izenači. V eni periodi se izravnava izvaja le med dvema CPE zato v eni periodi ne dosežemo enakomerne obremenjenosti vseh CPE. Da se to doseže, je potrebno več period.

3.2 Linux

Linux je danes najbolj razširjen odprtokodni operacijski sistem. Njegov prvotni avtor je Linus Tovards, ki je 1. avgusta 1991, 'izdal' prvo verzijo operacijskega sistema Linux 0.01. Ideje je avtor črpal predvsem iz operacijskega sistema Minix ter tudi iz sistemov tipa BSD in Sys V. Do danes je Linux doživel veliko sprememb, pri njegovem razvoju pa sodelujejo razvijalci celega sveta.

3.2.1 Razvrščevalnik O(n)

Linux jedro 2.4 ima preprost prioritetni razvrščevalnik, ki podpira razvrščanje s prekinjanjem. Podpira tri različne modele razvrščanja (POSIX[5] standard):

SCHED_FIFO *(POSIX.1b FIFO RT process)*
: FIFO razvrščanje v mehkem realnem času – proces se izvaja vse dokler ne blokira oziroma ga ne prekine proces z višjo prioriteto.

SCHED_RR *(POSIX.1b RR RT process)*
: Round robin razvrščanje procesov enake prioritete s prioriteto izvajanja v realnem času – ekvivalenten SCHED_FIFO le, da je proces omejen z časovno rezino, ki je odvisna od *nice* prioritete procesa. Ko procesu poteče časovna rezina, se proces razvrsti na konec vrste.

SCHED_OTHER
: Tradicionalni *time-shared* UNIX proces.

Razvrščevalnik za delovanje potrebuje naslednja polja `task_struct` strukture:

`policy`
: POSIX model razvrščanja za dotični proces.

[5]Portable Operating System Interface (POSIX) prestavlja družino standardov, ki omogočajo kompatibilnost med sistemi.

`need_resched`

Zastavica, ki zahteva izvršitev razvrščevalnika. V jedru 2.4 se zahteva preveri in po potrebi izvrši tik pred vrnitvijo v uporabniški način izvajanja, t. j. tik pred zaključkom ISR (in tudi sistemskega klica) – `linux/arch/i386/kernel/entry.S:204`

```
ENTRY(ret_from_sys_call)
    cli             # need_resched and signals atomic test
    cmpl $0,need_resched(%ebx)
    jne reschedule
```

`counter`

Število preostalih urinih prožitev do poteka časovne rezine procesa. Odštevanje se vrši v proceduri `linux/kernel/timer.c(579): update_process_times()` – ko vrednost pade na 0, se postavi zastavica `need_resched` in tako zahteva klic razvrščevalnika.
Ob kreaciji novega procesa, se časovna rezina razdeli enakovredno med očetom in sinom:

```
p->counter = (current->counter + 1) >> 1;
current->counter >>= 1;
```

To je potrebno, da se prepreči zloraba kreiranja sinov in tako pridobi 'poljubna' količina CPE časa – v primeru enakovredne razdelitve se preostala količina časovne rezine ohranja.
Na to polje lahko gledamo tudi kot na dinamično prioriteto procesa (glej (3.2.1)).
Polje je relevantno, če je `policy` enak `SCHED_RR` ali `SCHED_OTHER`.

`rt_priority`

Prioriteta procesov v realnem času.
Polje je relevantno, če je `policy` enak `SCHED_FIFO` ali `SCHED_RR`.

`priority`

Statična prioriteta procesa, ki je kar *nice* vrednost. Nastavi se jo lahko z uporabo POSIX.1b procedure `sched_setparam(2)`.
Polje je relevantno, če je `policy` enak `SCHED_OTHER`.

Izračun dinamičnih prioritet, imenovane *goodness*, vseh procesov se vrši v funkciji `linux/arch/i386/kernel/sched.c(144): goodness()` in je sledeč:

$$goodness = \begin{cases} \texttt{-1}, & \texttt{p->policy \& SCHED\_YIELD} \\ \texttt{0}, & \texttt{!p->counter} \\ \texttt{1000 + p->rt\_priority}, & \texttt{p->policy \& (SCHED\_FIFO|SCHED\_RR)} \\ \texttt{p->counter + (20 - p->nice)}, & \text{otherwise} \end{cases} \tag{3.7}$$

V primeru `p->policy == SCHED_OTHER` se *goodness* doda dodatna točka, ko je spominski prostor (angl. *address space*) ocenjevanega procesa enaka spominskemu prostoru zadnje izvajanega procesa. Na ta način se daje prednost procesom za katere predvidevamo večjo verjetnost predpomnilniških zadetkov (angl. *cache hit*) zaradi uporabe istega spominskega prostora.
Osrednja koda razvrščevalnika `linux/arch/i386/kernel/sched.c(549): schedule(void)` je zelo preprosta, vendar reda $O(n)$. Jedro procedure je naslednje:

```
next = idle_task(this_cpu);
c = -1000;
list_for_each(tmp, &runqueue_head) {
    p = list_entry(tmp, struct task_struct, run_list);
    if (can_schedule(p, this_cpu)) {
        int weight = goodness(p, this_cpu, prev->active_mm);
        if (weight > c)
            c = weight, next = p;
    }
}
```

Koda pregleda vrsto procesov v pripravljenosti in izbere proces `next` z najvišjo dinamično prioriteto *goodness* `c`. V primeru, da je po pregledu vrste `c = 0`, to pomeni, da je vsem procesom potekla časovna rezina. V tem primeru se naredi izračun časovnih rezin vsem procesom (ne samo procesom v vrsti):

```
for_each_task(p)
    p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
```

Koda ustreza izrazu:

$$\texttt{p->counter} = \frac{\texttt{p->counter}}{2} + (\frac{20 - \texttt{p->nice}}{4} + 1)$$

Izraz dodeli daljše časovne rezine in s tem posledično višjo dinamično prioriteto (vrednost *goodness*) procesom, ki niso porabili celotne časovne rezine – v večini primerov interaktivnim procesom.
Maksimalno dinamično prioriteto izrazito interaktivnega procesa lahko določimo s pomočjo geometrijske vrste:

$$\texttt{counter} = \texttt{priority} \Rightarrow \texttt{counter} \rightsquigarrow \texttt{counter} \cdot \sum_{n=0}^{\infty} (\frac{1}{2})^n = 2 \cdot \texttt{counter}$$

Pri privzeti frekvenci sistemskega časovnika 100Hz je minimalna časovna rezina $10ms$ in maksimalna $110ms$, vendar se lahko ta poveča za faktor 2.
Po izračunu se ponovi izbira procesa `next` z najvišjo dinamično prioriteto. Izjema so procesi `SCHED_RR`, ki se obdelajo pred izbiro novega procesa `next`, saj potrebujejo posebno obravnavo:

```
if (prev->policy == SCHED_RR)
    if (!prev->counter) {
        prev->counter = NICE_TO_TICKS(prev->nice);
        move_last_runqueue(prev);
    }
```

V primeru izteka časovne rezine je potrebno ponovno določiti vrednosti časovne rezine ter proces premakniti na konec vrste.

Podpora večprocesorskim sistemom

Razvrščevalnik podpira večprocesorske sisteme z osnovnim algoritmom, ki podpira oziroma nagrajuje afiniteto CPE kot tudi SMT.
Vse CPE izvajajo kodo razvrščevalnika in si delijo njegove podatkovne strukture, ki so temu primerno sinhronizirane.
Nagrajevanje afinitete je realizirano že v izračunu dinamične prioritete *goodness*. V primeru `p->policy == SCHED_OTHER` se *goodness* prišteje vrednost $PROC\_CHANGE\_PENALTY$, če je ocenjevan proces na zadnje tekel na CPE za katerega se dela izračun. Vrednost $PROC\_CHANGE\_PENALTY$ je odvisna od platforme, v veliki večini pa je ta 15 ali 20.
Glavni del podpore SMP je procedura *reschedule_idle*. Ta se kliče, ko se proces zbudi, in določi CPE na katerem se bo dotični proces izvajal. Algoritem določanja je naslednji:

- če je CPE na katerem je proces nazadnje tekel prosta, jo uporabi;
- če je sorodna CPE (podpora SMT) na katerem je proces nazadnje tekla prosta, jo uporabi;
- drugače uporabi CPE, ki je prosta najdlje časa;
- v primeru, da prostega CPE ni, uporabi CPE na katerem teče proces z manjšo in najnižjo dinamično prioriteto relativno na dinamično prioriteto procesa za katerega iščemo CPE, če bi le ta prekinil izvajanje trenutnega procesa na tej CPE;
- v kolikor se ne zadosti nobenemu naštetemu pogoju, se proces ne obudi in pusti na čakanju (v spanju)

Algoritem je reda $O(n)$, kjer je n število logičnih CPE v sistemu.

3.2.2 Razvrščevalnik O(1)

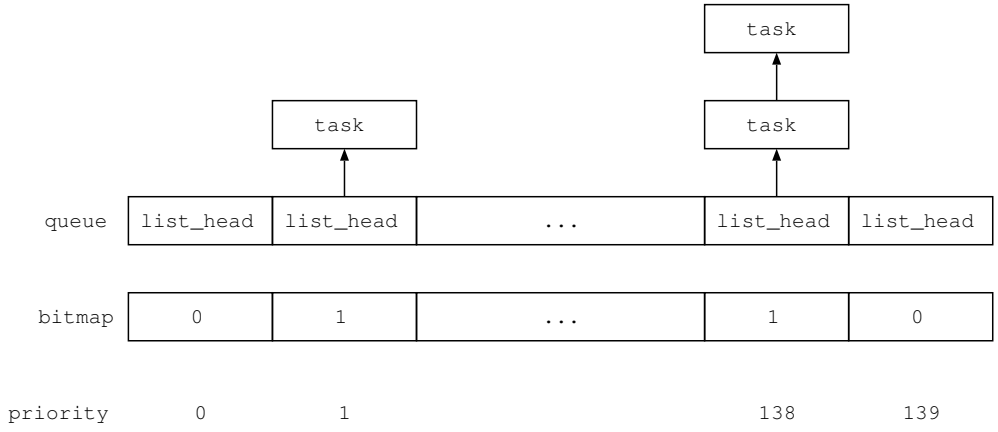
Linux je razvojni verziji 2.5.2 dobil popolnoma nov razvrščevalnik reda $O(1)$. Glavni cilj je bil znebiti se starega $O(n)$ algoritma, ki je bil zaradi preračunavanja dinamičnih prioritet v metodi `schedule()` skrajno neučinkovit. Vse postane še bolj zaskrbljujoče, če se zavedamo, da tak algoritem popolnoma povozi prvonivojski predpomnilik (angl. *L1 cache*) ter tako zmanjša hitrost izvajanja procesov, kar se pozna v manjši prepustnosti. Izjemno veliko truda je bilo vloženega tudi podpori SMP in standardu NUMA. Z verzijo 2.6 je izvajanje jedro možno tudi prekiniti, kar zelo izboljša odzivnosti sistema (glej konec poglavja).
Vsaki CPE se dodeli podatkovna struktura imenovana `runqueue`. Struktura vsebuje podstrukturi (`prio_array`), ki predstavljata dve prioritetni vrsti – *aktivno* in *pretečeno* (angl. *active and expired*). V aktivni vrsti so procesi, ki imajo na voljo še nekaj časovne rezine CPE, v pretečeni pa procesi, ki so porabili celotno časovno rezino. Obe vrsti sta urejeni glede na prioriteto procesov. Ko procesu poteče časovna rezina, se prestavi v vrsto pretečenih procesov in ko se aktivna izprazni, se vrsti zamenjata – aktivna postane pretečena in pretečena postane aktivna. Zanimiva je implementacija obeh vrst, saj nista pravi vrsti, temveč polji.

```
/* for MAX_PRIO == 140 => BITMAP_SIZE = 5 */
#define BITMAP_SIZE ((((MAX_PRIO+1+7)/8)+sizeof(long)-1)/sizeof(long))

struct prio_array {
    int nr_active;
    unsigned long bitmap[BITMAP_SIZE];
    struct list_head queue[MAX_PRIO];
};
```

Strukturo sestavlja atribut `nr_active`, ki hrani število aktivnih procesov v strukturi, in dve polji – `queue` je polje seznamov vseh možnih prioritet procesov in `bitmap` je pomožno polje za iskanje procesa z najvišjo prioriteto

(operacija iskanja zahtevnosti $O(n)$ ni potrebna). Polje `bitmap` je zaslužno za izjemno hitro iskanje procesa z najvišjo prioriteto; to se v povprečju doseže le s štirimi enostavnimi inštrukcijami, predvsem na račun x86 `BSFL` inštrukcije. To polje predstavlja bitno sliko (angl. *bitmap*) obstoja procesov z določenimi prioritetami – postavljen bit na n-tem mestu pomeni obstoj procesa (-ov) s prioriteto n, torej prisotnost procesov v seznamu `queue[n]` (glej sliko 3.1). Operaciji dodajanja novega procesa in iskanje procesa z



Slika 3.1: Ponazoritev podatkovne strukture `prio_array`.

najvišjo prioriteto sta tako zaradi učinkovite podatkovne strukture zelo hitri operaciji – zahtevnosti le $O(1)$.
Razvrščevalnik uporablja prioritetni prostor `[0..MAX_PRIO-1]` oziroma `[0..139]`, kjer je ta razdeljen na:

`[0..MAX_RT_PRIO-1]` oziroma `[0..99]`
: Prioritetni prostor namenjen procesom za delo v realnem času. Prioritete so statične in nastavljive z uporabo POSIX.1b procedure `sched_setparam(2)`.

`[MAX_RT_PRIO..MAX_PRIO-1]` oziroma `[100..139]`
: Prioritetni prostor namenjen normalnim procesom. Efektivna prioriteta (funkcija `effective_prio`) je določena kot vsota statične

prioritete in bonusa:

$$priority = static\_prio + bonus = (nice + 20) + bonus,$$

kjer je *bonus* vrednost iz intervala `[-5..5]`, ki je odvisen od interaktivne značilnosti procesa.

Razvrščevalnik pozna tri modele razvrščanja po POSIX.1b standardu (glej 3.2.1). Razvrščanje procesov `SCHED_FIFO` in `SCHED_RR` je enako kot verziji 2.4 (saj je način predpisan po standardu) – procesi `SCHED_FIFO` so najvišje prioritete in tečejo dokler ne blokirajo oziroma jih ne prekine proces z višjo prioriteto. Za procese `SCHED_RR` velja enako z izjemo, da se `SCHED_RR` procesi enake prioritete razvrščajo po round robin principu, vsak proces pa dobi vnaprej omejeno časovno rezino. Zato se bom v nadaljevanju osredotočil le na procese tipa `SCHED_NORMAL` (v verziji 2.4 `SCHED_OTHER`). Časovne rezine se določajo dinamično glede na statično prioriteto (vrednost *nice*) procesa. Minimalna časovna rezina je $\text{slice}_{\text{min}} = 10ms$, maksimalna pa $\text{slice}_{\text{max}} = 200ms$. Izračun je naslednji (poenostavljen izraz):

$$time\_slice = \text{slice}_{\text{min}} + \frac{19 - nice}{39} \cdot (\text{slice}_{\text{max}} - \text{slice}_{\text{min}})$$

Procesi z višjo prioriteto dobijo daljše časovne rezine, kar je ravno nasprotno večini razvrščevalnikom – ti dodelijo krajše časovne rezine procesom z višjo prioriteto zaradi domneve, da gre za interaktivne procese. Izračunavanje dinamičnih prioritet in dolžin časovnih rezin se vrši v proceduri `linux/kernel/sched.c(1342): scheduler_tick`, ki se kliče ob vsaki prožitvi sistemskega časovnika. Frekvenca sistemskega časovnika se je v jedru 2.6 povečala iz 100Hz na 1000Hz, kar pomeni za faktor 10 večje režijske kvote vendar manjše odzivne čase sistema[6]. Implementacija je sledeča (prikazani samo najbolj relevantni deli kode):

```
void scheduler_tick(int user_ticks, int sys_ticks) {
```

[6]Primer je realizacija sistemskega klica `nanosleep(perdiod)`, ki v jedru 2.4 v idealnih razmerah spi za $10ms + \lceil \frac{period}{10ms} \rceil \cdot 10ms$, medtem ko v jedru 2.6 za $2ms + \lfloor \frac{period}{1ms} \rfloor \cdot 1ms$.

```
...
if (!--p->time_slice) {
    dequeue_task(p, rq->active);
    set_tsk_need_resched(p);
    p->prio = effective_prio(p);
    p->time_slice = task_timeslice(p);
```

Ko procesu poteče celotna časovna rezine:

1. ga odstrani iz aktivne vrste,
2. zahtevaj razvrščanje (klic razvrščevalnika tik prek vrnitvijo v uporabniški način)
3. izračunaj efektivno dinamično prioriteto in
4. novo dolžino časovne rezine.

```
    if (!rq->expired_timestamp)
        rq->expired_timestamp = jiffies;
    if (!TASK_INTERACTIVE(p) || EXPIRED_STARVING(rq)) {
        enqueue_task(p, rq->expired);
    } else
        enqueue_task(p, rq->active);
```

V tem delu kode se proces, ki je porabil celotno časovno rezino, premakne v pretečeno vrsto. Izjema so interkativni procesi.
V prvih dveh vrsticah si razvrščevalnik zapomni čas, ko je potekla časovna rezina prvemu procesu v vrsti. Ta podatek je pomemben pri testiranju pogoja `EXPIRED_STARVING(rq)`, ki vrne `true`, v kolikor je od `rq->expired_timestamp` pretekel določen čas – zgornja meja imenovana `STARVATION_TIME`, ki je odvisna od števila obremenjenosti sistema (števila procesov v aktivni vrsti). Pogoj je potreben, da prepreči relativno dolgotrajno čakanje na preklop med aktivno in pretečeno vrsto (ta se izvrši, ko se aktivna vrsta sprazni) in s tem stradanje procesov v pretečeni vrsti.

```
    } else {
        if (TASK_INTERACTIVE(p) && !((task_timeslice(p) -
            p->time_slice) % TIMESLICE_GRANULARITY(p)) &&
            (p->time_slice >= TIMESLICE_GRANULARITY(p)) &&
            (p->array == rq->active)) {

            dequeue_task(p, rq->active);
            set_tsk_need_resched(p);
            p->prio = effective_prio(p);
            enqueue_task(p, rq->active);
        }
    }
}
```

Zgornji del kode je posebnost razvrščevalnika – gre za drobljenje časovnih rezin interaktivnim procesom z namenom preprečiti dolgotrajno zasedenost CPE (v primeru, ko interaktivni proces začne spreminjati svoj značaj). Časovna rezina se dobesedno zdrobi na več delov, kar ne pomeni, da se procesu časovna rezina skrajša, le izvajanje se prepusti drugim procesom z enako prioriteto. Makro drobitve `TIMESLICE_GRANULARITY` je določen z naslednjim predpisom (poenostavljeno):

$$timeslice\_granularity = \begin{cases} \text{slice}_{\min}, & bonus = 5 \\ \text{slice}_{\min} \cdot 2^{(5-bonus)-1}, & \text{otherwise} \end{cases}$$

Glavna procedure razvrščevalnika je sledeča:

```
asmlinkage void schedule(void) {
    ...
    preempt_disable();
    ...
```

Jedro 2.6 omogoča tudi prekinjanje sebe med izvajanjem (glej 1.3), vendar nekateri deli ne smejo biti prekinjeni – razvrščevalnik je eden izmed teh kritičnih delov.

```
pick_next_task:
    if (unlikely(!rq->nr_running)) {
        next = rq->idle;
        rq->expired_timestamp = 0;
        goto switch_tasks;
    }
```

V kolikor ni pripravljenih procesov, izberi korenski `idle` proces.

```
    array = rq->active;
    if (unlikely(!array->nr_active)) {
        rq->active = rq->expired;
        rq->expired = array;
        array = rq->active;
        rq->expired_timestamp = 0;
    }
```

Preveri, ali je aktivna vrsta je prazna, če je, izvedi zamenjavo vrst in resetiraj `rq->expired_timestamp`.

```
    idx = sched_find_first_bit(array->bitmap);
    queue = array->queue + idx;
    next = list_entry(queue->next, task_t, run_list);
```

Poišči proces s najvišjo prioriteto v aktivni vrsti – vse so hitre $O(1)$ operacije.

```
switch_tasks:
    ...
        prev = context_switch(rq, prev, next);
    ...
    preempt_enable_no_resched();
    if (test_thread_flag(TIF_NEED_RESCHED))
        goto need_resched;
}
```

In še zaključna faza razvrščanja, t. i. preklop. Če smo na začetku onemogočili prekinjanje razvrščevalnika, ga je na koncu potrebno tudi nazaj omogočiti. To se naredi s klicem `preempt_enable_no_resched()`, ki omogoči prekinjanje brez klica razvrščevalnika – procedura (v resnici marko) `preempt_enable()` še dodatno kliče razvrščevalnik z namenom, da preveri, ali je med onemogočenim prekinjanjem v stanje pripravljenosti prišel kakšen proces z višjo prioriteto. Tak princip je skupen (nujen) tudi RTOS sistemom. Princip je siloma preprost, saj omenjene procedure le operirajo na spremenljivki `preempt_count` in povečujejo oziroma zmanjšujejo števec (podpora ugnezdenim zahtevam). V trenutku ko števec pade na 0, se takoj kliče razvrščevalnik. Potrebna je le še koda ob koncu vsake prekinitve, ki preveri stanje števca in v primeri neničelne vrednosti onemogoči klic razvrščevalnika (`linux/arch/i386/kernel/entry.S(217)`):

```
ENTRY(resume_kernel)
    cmpl $0,TI_PRE_COUNT(%ebp)  # non-zero preempt_count ?
    jnz restore_all
need_resched:
    movl TI_FLAGS(%ebp), %ecx   # need_resched set ?
    testb $_TIF_NEED_RESCHED, %cl
    jz restore_all
```

Razvrščevalnik določa procesom stopnjo interaktivnosti glede na čas, ki so ga porabili za izvajanje oziroma, ko so bili neaktivni (za razliko od ULE je tukaj vključena latenca razvrščanja) – spremenljivka je določena enostavno kot `sleep_avg = sleep_time - run_time`, nakar se izračuna že omenjeni `bonus` (poenostavljeno)

$$bonus = \frac{sleep\_avg}{100ms} \cdot bonus_{range} - \frac{bonus_{range}}{2},$$

kjer je $bonus_{range} = 10$.
S pomočjo nadaljnjih podrobnih izračunov se doseže še:

- procesi z minimalno prioriteto `nice = +19` nikoli ne prekinejo procesa s privzeto prioriteto `nice = 0` in

- procesi s privzeto prioriteto `nice = 0` nikoli ne prekinejo procesa z maksimalno prioriteto `nice = -20`.

Podpora večprocesorskim sistemom

Kot je bilo že omenjeno, se vsaki CPE se dodeli podatkovna struktura imenovana `runqueue`, ki vsebuje dve prioritetni vrsti – *aktivno* in *pretečeno* (angl. *active and expired*). V primeru, ko je jedro prevedeno s SMP podporo, struktura dodatno dobi polja, ki vsebujejo podatek `cpu_load` o obremenjenosti CPE in polja za podporo jedrni niti (ena na CPE), ki se uporablja za prenos (migracijo) procesov iz ene CPE na drugo. Polje obremenjenosti CPE `cpu_load` je odvisno od števila procesov v obeh vrstah kot oteženo drseče povprečje zadnjih dolžine 3 in se posodobi ob vsakokratnem klicu `scheduler_tick()`.
Za enakomerno porazdelitev izkoriščenosti CPE (angl. *CPU load-balancing*) razvrščevalnik uporabljajo 3 mehanizmi:

Mehanizem *pull* - CPE zmanjka procesov

Ko CPE zmanjka procesov za izvajanje (ali so vsi procesi končali z izvajanjem, ali pa vsi spijo), se poišče najbolj obremenjeno CPE ter iz nje prenese (migrira) procese z najnižjo prioriteto na prosto CPE. Najprej se pregleda *pretečena* vrsta in nato šele *aktivna* vrsta. Iz najbolj obremenjene CPE se prenese toliko procesov, dokler se obremenjenost CPE ne izenači povprečni obremenjenosti vseh CPE.

Mehanizem *pull* - periodična izravnava porazdelitve obremenjenosti

Izravnava porazdelitve obremenjenosti med CPE se izvaja periodično. Za vsako CPE se med izvajanjem metode `scheduler_tick()` kliče `trigger_load_balance()`, ki sproži programsko prekinitev. Ob izvršitvi te prekinitve se preveri, ali je preteklo dovolj časa od zanje izravnave, in v kolikor je temu pogoju zadoščeno, se kliče `load_balance()`. Slednja metoda poišče najbolj obremenjeno CPE in

prenese nekaj procesov na CPE, ki izvaja dotično izravnavo. Prenašajo se procesi z najnižjo prioriteto. Najprej se pregleda *pretečena* vrsta in nato šele *aktivna* vrsta. Iz najbolj obremenjene CPE se prenese toliko procesov, dokler se obremenjenost CPE ne izenači povprečni obremenjenosti vseh CPE.

Mehanizem *push*

V kolikor *pull* mehanizem za periodično izravnava porazdelitve obremenjenosti po večih poskusih ne uspe opraviti izravnave (neuspešen prenos procesa) se preko migracijske jedrne niti kliče `active_load_balance()`. Ta metoda poišče najbolje obremenjeno CPE iz nje prenese en proces na CPE, ki ni uspela izvesti izravnave preko *pull* mehanizma. Pri tem se ne upošteva, kakšna bo nastala porazdelitev obremenitve med CPE, edini pogoj je, da razbremenjeni CPE ostane v vrstah vsaj en proces.

3.3 Windows

Operacijski sistem Microsoft Windows je daleč najbolj razširjen uporabniški operacijski sistem. Posebnost sistema je, da je jedro sistema tesno povezano z grafičnim vmesnikom.
V nadaljevanju bom opisal razvrščevalnik operacijskega sistema Windows 2000 (NT 5.0), ki temelji na Windows NT 4.0.
Ker Microsoft Windows sistemi niso odprtokodni sistemi, nisem imel vpogleda v izvorno kodo in zato nadaljnji opis temelji le na obstoječi dokumentaciji.

3.3.1 Razvrščevalnik Windows 2000

Razvrščevalnik Windows 2000 je prioritetni razvrščevalnik, ki podpira razvrščanje s prekinjanjem – izvaja se vedno proces, ki ima največjo prioriteto, procesi z enakimi prioritetami pa se razvrščajo po principu round robin.
Razvrščevalnik uporablja prioritetni prostor `[0..31]`, kjer višje vrednosti pomenijo višjo prioriteto. Prioritetni prostor je razdeljen na:

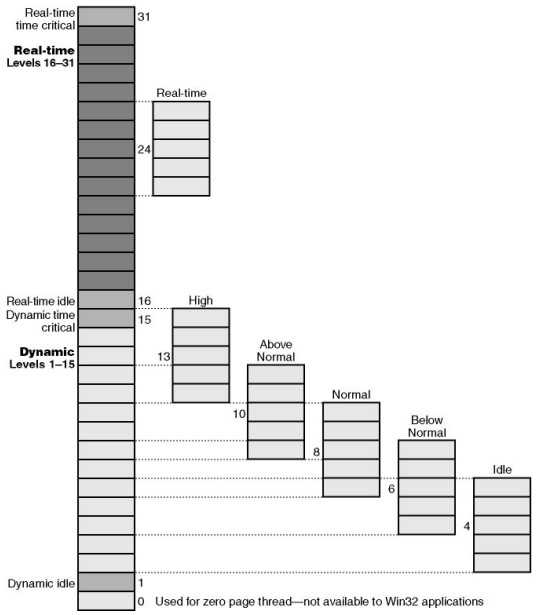
`[0]`
: Prioriteta rezervirana samo za *idle* proces, torej ničeln proces, ki se izvaja, ko ni na voljo nobenega procesa v stanju pripravljenosti.

`[1..15]`
: Prioritete namenjene normalnim procesom. Prioritete so dinamične, zato se ta interval tudi imenuje *dinamični prioritetni prostor*.

`[16..31]`
: Prioritetni interval rezerviran za procese za delo v realnem času. Prioritete so statične.

Prioritetne se določajo v dveh korakih – najprej se mora določiti prioritetni razred (razvrščevalnik razred), nato pa znotraj tega razreda še relativno prioriteto.

Obstajajo naslednji prioritetni razredi s pripadajočimi privzetimi prioritetami: *realtime* (24), *high* (13), *normal* (8) in *idle* (4).
Nadalje se znotraj razredov določa še relativna prioriteta, ki se prišteje privzeti razredni prioriteti: *highest* (+2), *above normal* (+1), *normal* (0), *below normal* (-1) in *lowest* (-2).



Slika 3.2: Ponazoritev prioritetnega prostora v Windows 2000 [3].

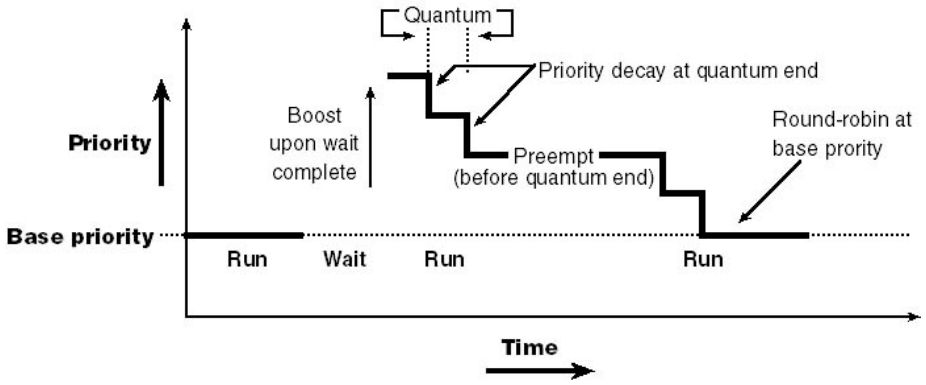
Vsakemu procesu se tudi določi dolžina časovne rezine. Ta ni podana v času vendar v kvantih (angl. *quantum*), kjer je dolžina kvantuma določena s frekvenco prožitve sistemske ure. Na enoprocesorskem sistemu Windows 2000 Professional ta znaša $10ms$.
Vsakemu procesu se na začetku priredi časovno rezino 6 kvantov (36 na Server različici sistema), vendar se ob vsaki prožiti sistemske ure ta zmanjša za 3 kvante – efektivna dolžina časovne rezine sta torej 2 prožitvi sistemske ure (12 na Server različici sistema). Razlog zakaj se ne uporablja kar začetni kvant 2 in zmanjševanje za 1, je zato, da se lahko procesom zmanjša le del periode sistemske ure, natančneje za $\frac{1}{3}$. Vsem procesom prioritete manjše od 14, se ob klicu čakalne funkcije (primera sta `WaitForSingleObject` ali `WaitForMultipleObjects`). To se počne iz

razloga, da se prepreči procesom, ki bi prešli v stanje čakanja pred prožitvijo sistemske ure, neobračunavanje delno porabljene časovne rezine (problem diskretnih sistemov).
V primeru preklopa oken v grafičnem vmesniku razvrščevalnik podaljša časovne rezine – procesu (če smo natančni vsem nitim procesa) katerega okno preide v ospredje (angl. *foreground*), se lahko podaljša časovna rezina vse do 4 kvante (nastavljivo v Control Panel → System applet → Performance tab) ter tako izboljša odzivnost aplikacij v ospredju.
Procesom prioritetnega razreda *realtime* se dolžina časovne rezine resetira ob vsaki prekinitvi, medtem ko se procesom z nižjo prioriteto ponovno nastavi le, ko ti porabijo celotno časovno rezino.
Procesom v dinamičnem prioritetnem prostoru se prioritete procesov dinamično spreminjajo z naslednjima omejitvama:

- maksimalna prioriteta je omejena s 15 – stopnja višje je že *realtime* prioritetni prostor,
- minimalna prioriteta je omejena z nastavljeno (programsko) prioriteto, t. i. bazna prioriteta

Ko proces porabi celotno časovno rezino, se njegova trenutna prioriteta zmanjša (angl. *to decay*) za 1. Kot protiutež zmanjševanju, deluje proces večanja prioritet (angl. *to boost*) in se vrši po naslednjem postopku:

- ob vsakem zaključku V/I operacije povečaj bazno (*ne trenutno!*) prioriteto za določeno število kvantov
  (primeri: disk, CD-ROM, video - 1; mreža - 2; tipkovnica, miška - 6; zvok - 8),
- po koncu čakanja na dogodek (angl. *event*) ali semafor povečaj trenutno (*ne bazno!*) prioriteto za 1,
- aplikacijam v ospredju po zaključku čakanja operacije v jedru povečaj prioriteto za število kvantov vrednosti `PsPrioritySeparation`,



Slika 3.3: Ponazoritev dinamike prioritet v Windows 2000 [3].

- ob obuditvi GUI niti povečaj njeno prioriteto za 2.

Razvrščevalnik Windows 2000 ima tudi mehanizem preprečevanja stradanja procesov. Enkrat na sekundo pregleda procese v vrsti in v primeru, ko proces 'strada' že 300 urinih period sistemske ure (3s), nastavi njegovo prioriteto na 14 (s Service Pack 2 na 15) in podvoji časovno rezino procesa. Proces tako prejme t. i. *anti-starvation boost*. Da pa bi omejili porabo CPE, se enkrat na sekundo pregleda maksimalno 16 procesov in prav tako pospeši maksimalno 10 procesov. Ko takim procesom poteče podaljšana časovna rezina, se njihova prioriteta in dolžina časovne rezine vrneta v prvotno stanje.

Podpora večprocesorskim sistemom

Windows 2000 podpira SMP sisteme. Interne podatkovne strukture podpirajo do 32 CPE. Razvrščevalnik podpira tako afiniteto CPE kot porazdelitev obremenitve CPE.

Poglavje 4

Meritve latence razvrščevalnika

Opravil sem meritev latence[1] razvrščevalnika. Latenca razvrščevalnika je čas zakasnitve med prekinitvijo (ali izpolnitve semaforja oziroma pogojne spremenljivke) in začetka izvajanja samega procesa, ki čakajo nanjo. Meritev sem opravil na operacijskem sistemu Linux z jedrom s podporo le eni CPE. To sem opravil z namenom, da ponazorim prednosti implementacije jedra, ki ga razvrščevalnik lahko prekine (angl. *preemptive kernel*).

4.1 Testni sistem

Za testni sistem sem uporabil delovno postajo z naslednjimi komponentami:

- matična plošča Supermicro X9SRA
- CPE Intel Xeon E5-2620 (15M, 2.00GHz, 6C/12T)
- pomnilnik 4x 8GB Samsung M393B1K70CH0-CH9 DDR3 1333MHz ECC
- (brez diskov, OS je bil naložen preko CD bralnika, testni programi pa preko USB ključka)

[1] webster.com: Čas med stimulacijo in odzivom.

Uporabljen operacijski sistem je Slackware Linux 11.0 z minimalno konfiguracijo (na sistemu ne teče nobena dodatna storitev). Sistem ni priključen na omrežje, da se izogne nedeterministični obremenitvi s strani omrežja. Uporabljeni sta bili dve jedri Linux OS, 2.4.33.3 in 2.6.17.13.

4.2 Obremenitev testnega sistema

Za obremenitev testnega sistema sem uporabil odprtokodni program Harvarske univerze imenovan *stress*[2]. Program je konfigurabilne narave in lahko obremeni različne dele sistema:

- za obremenitev CPE v zanki računa kvadratni koren naključnih števil (popolna obremenitev CPE, natančneje dela CPE za računanje s števili s plavajočo vejico),
- za obremenitev navideznega pomnilnika v zanki alocira 256MB pomnilnika, piše vanj in ga dealocira,
- za obremenitev vhodno/izhodnega dela v zanki kliče sistemski klic *sync()*, ki zahteva zapis vseh podatkov iz predpomnilnikov vhodnih/izhodnih naprav na dotične vhodne/izhodne naprave.

Za vsako breme se ustvari lastna nit.
Breme sem ustvaril z naslednjim ukazom:

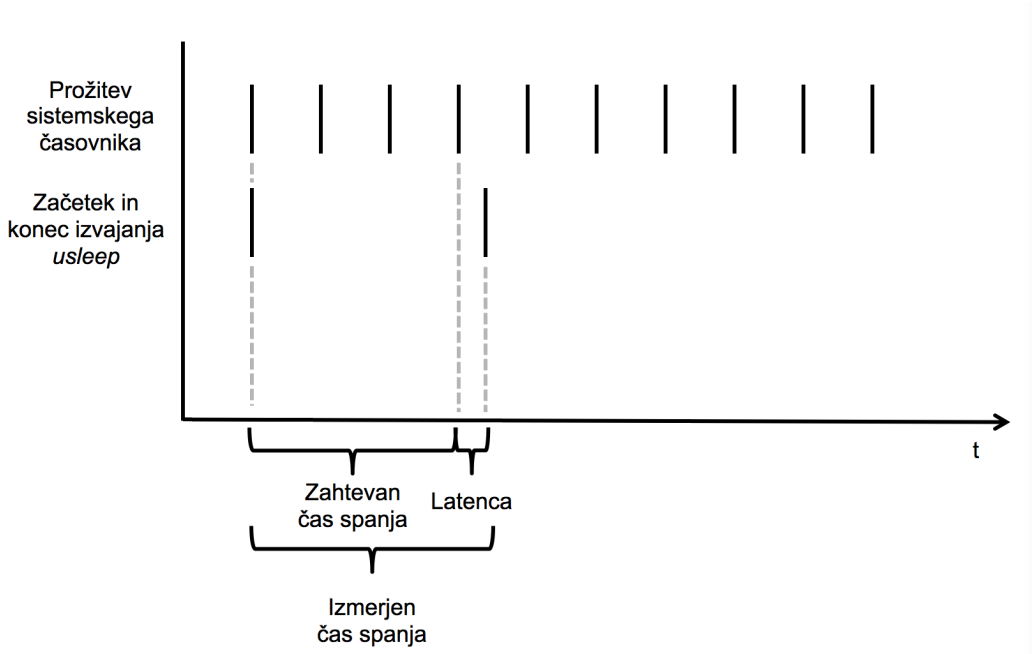
```
stress --cpu 2 --vm 1 --io 1
```

Ukaz ustvari 2 niti, ki popolnoma obremenita CPE, eno nit za obremenitev dela navideznega pomnilnika in eno nit, ki obremeni vhodno/izhodni del sistema.

[2] `http://people.seas.harvard.edu/~apw/stress/`

4.3 Postopek meritve

S pomočjo napisanega programčka (glej A.1) sem meril latenco razvrščevalnika z uporabo sistemskega ukaza *usleep*, ki prekine delovanje procesa vsaj za podano število mikrosekund. Ker je razvrščevalnik vezan na sistemske časovnike, bo razvrščevalnik imel možnost izvajanja drugega procesa tudi ob prekinitvi časovnika. Ukaz *usleep* zahteva uporabo časovnika in v primeru, da ima proces, ki izvaja *usleep* ukaz, najvišjo prioriteto, se bo izvajanje tega procesa nadaljevalo takoj ob prekinitvi časovnika. Prav to dejstvo izkorišča program za merjenje latence. V kolikor je zahtevan čas spanja mnogokratnik periode časovnika in se ukaz *usleep* zahteva ob začetku periode časovnika, predstavlja razlika med dejanskim spanjem in ukazanim časom spanja latenco razvrščevalnika (glej 4.1).



Slika 4.1: Grafični prikaz meritve.

Perioda časovnikov jedra OS Linux 2.4.x je 10ms, jedra OS Linux 2.6.x pa 4ms (v kolikor ni omogočena uporaba visoko-resolucijskih časovnikov). Izbran čas spanja je 20ms, ki predstavlja najmanjši skupni mnogokratnik

10ms in 4ms. Zahtevo po sinhronizaciji med začetkom izvajanja ukaza *usleep* in začetkom periode časovnika, pa se deloma reši s tem, da se ukaz poda takoj ob koncu prejšnje periode časovnika, torej ob koncu izvajanja prejšnjega ukaza *usleep*. S tem bo začetek izvajanja naslednjega ukaza *usleep* zaostajalo za čas latence razvrščevalnika prejšnje meritve. Tako bo posledično bo naslednja meritev krajša za latenco prejšnje meritve. Ob upoštevanju tega dejstva in ob predpostavki, da nas zanimajo le maksimalni odkloni latenc, ki so drugega velikostnega reda napram povprečni latenci, je opisan algoritem za meritev latence razvrščevalnika primeren.
Ključni koraki programčka so:

Nastavi prioriteto procesa na `SCHED_FIFO`.

Tako se zagotovi, da bo proces, ki meri latenco, res prvi proces, ki bo ob prekinitvi časovnika dobil CPE.

'Zakleni' meritveni program v fizični delovni pomnilnik.

V kolikor se to ne bi naredilo, bi lahko operacijski sistem (pri veliki porabi delovnega pomnilnika) začasno prestavil na disk[3], kar bi ob ponovni aktivaciji meritvenega procesa zahtevalo ponoven prenos programa v fizični delovni pomnilnik. To se bi seveda odražalo kot dodatna zakasnitev pri merjenju latence.

Uporaba realne ure procesorja.

Za meritev časa sem uporabil realne ure procesorja – branje registra (števca), ki se poveča ob vsaki prožitvi ure procesorja, t. j. s frekvenco delovanja procesorja.
Uporaba sistemskega klica `gettimeofday()` ne bi bila pravilna rešitev, saj se ta posodobi le ob prožitvah sistemske ure.
Čas branja realne ure procesorja je zanemarljivo majhen (približno $10ns$).

Meritev N(=1000) vzorcev.

[3]Linux uporablja navidezni pomnilnik z ostranjevanjem.

Opravi se N meritev, takoj druga za drugo v diskretnem času. Najprej se prebere register realne ure, nato kliče sistemski klic `usleep(20000)`, kjer proces odda CPE za $20us$, ter nato znova prebere register realne ure. Izračuna se razlika in se jo pretvori v us ter shrani v polje rezultatov meritev.

Izpis rezultatov meritev.

Rezultate meritev se izpiše na standardni izhod, ki se jih potem uporabi za analizo.

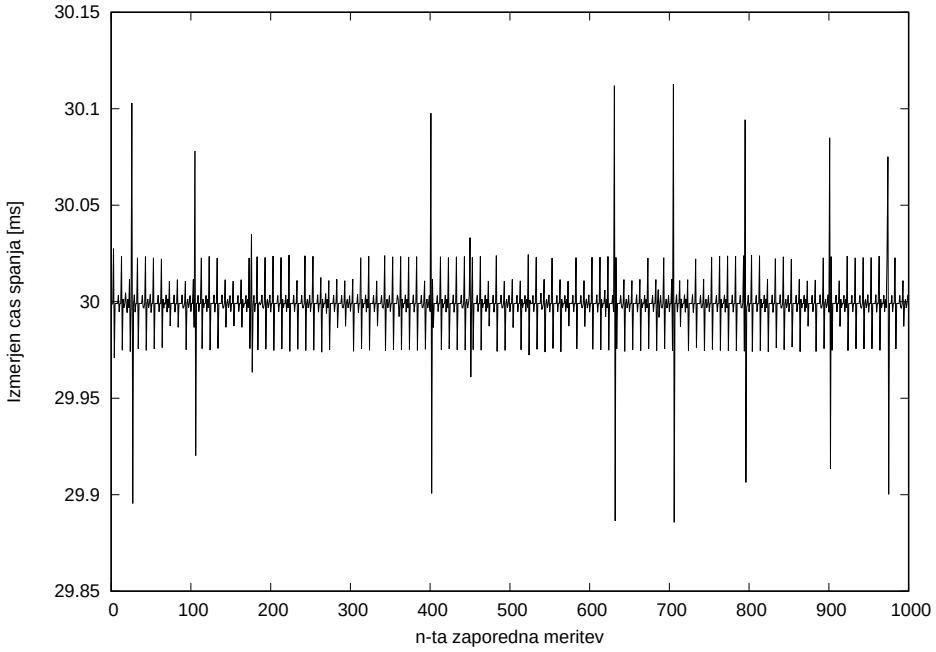
4.4 Meritve

Meritve sem opravil na dveh različnih jedrih OS Linux, *2.4.33.3* in *2.6.17.13*.

4.4.1 Meritve pri neobremenjenem sistemu

Naslednja grafa prikazujeta izmerjene čase spanja (y os) glede na zahtevanih $20ms$ tekom 1000 zaporednih meritev (x os) opravljenih druga takoj za drugo v diskretnem času pri neobremenjenem sistemu.
Meritve dejanskega časa spanja glede na zahtevanih $20ms$ na jedru OS Linux *2.4.33.3* (glej 4.2) dajo naslednji rezultat:

$$\mathrm{t}_{\text{čas spanja}} = 30.00ms \pm 0.02ms,$$
$$\mathrm{t}_{\min} = 29.89ms,$$
$$\mathrm{t}_{\max} = 30.11ms$$



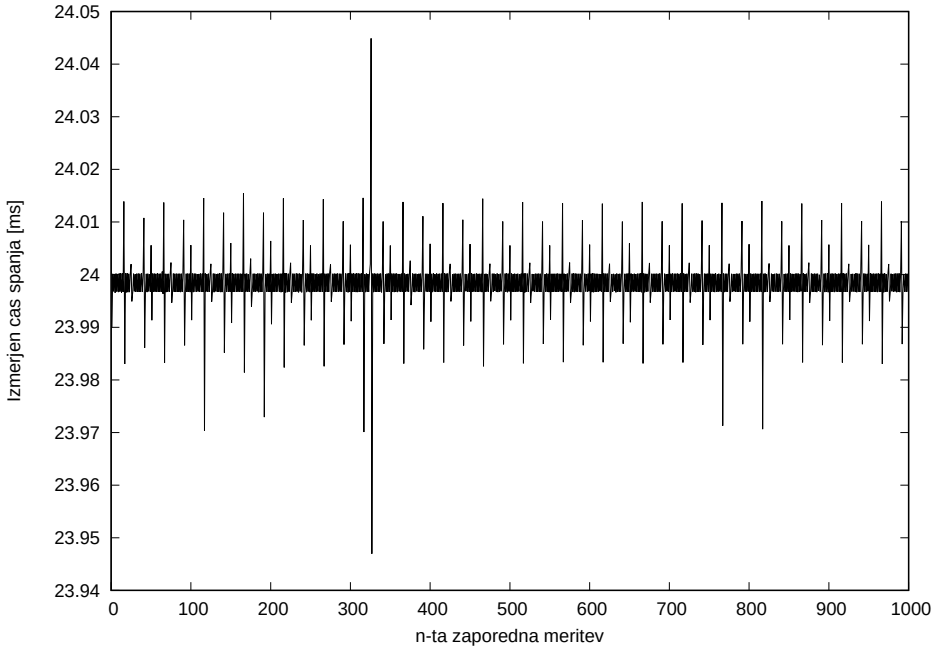
Slika 4.2: Neobremenjen sistem na jedru OS Linux *2.4.33.3*.
Pozor: x os ne seka y osi pri vrednosti 0!

Postavi se vprašanje, zakaj je povprečje $30ms$ in ne zahtevanih $20ms$?
Razlog je implementacija procedure `usleep`. Proces v resnici spi za

$10ms + \lceil \frac{period}{10ms} \rceil \cdot 10ms$. Diskretnost intervala $10ms$ je ravno perioda proženja sistemske ure. Zakaj potem pribitek $10ms$? Iz razloga, da ne bi prišlo do tega bi proces čakal manj kot je zahteval – to se vidi kot zakasnitve, ki so pod povprečjem $30ms$.

Rezultati meritev na novejšem jedru OS Linux *2.6.17.13* (glej 4.3) so naslednji:

$$t_{\text{čas spanja}} = 24.00ms \pm 0.01ms,$$
$$t_{\min} = 23.95ms,$$
$$t_{\max} = 24.04ms$$



Slika 4.3: Neobremenjen sistem na jedru OS Linux *2.6.17.13*.
Pozor: x os ne seka y osi pri vrednosti 0!

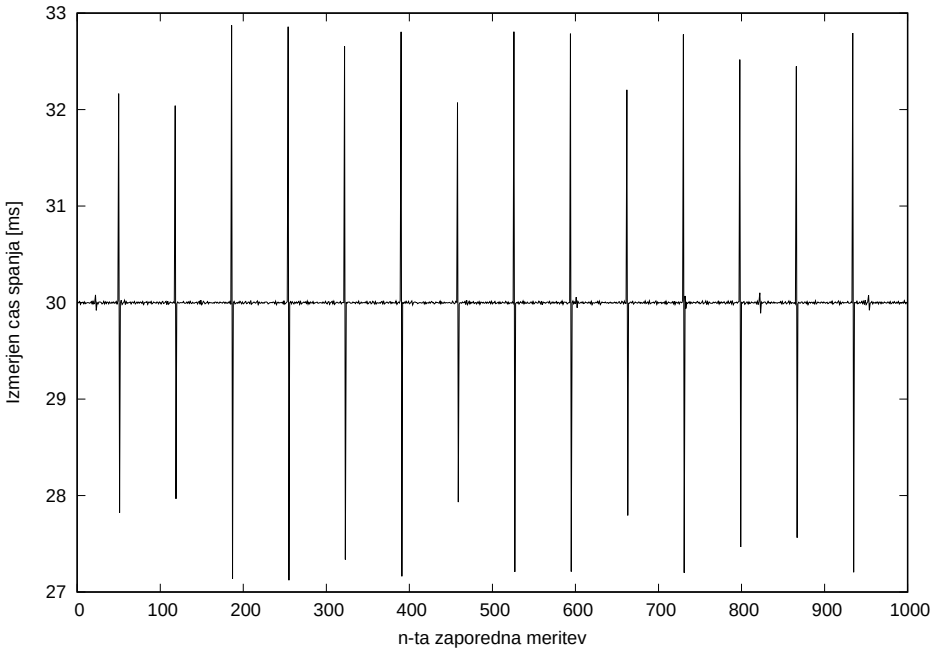
Jedro OS Linux *2.6.17.13* ima za faktor 2.5 večjo frekvenco proženja ure časovnika, zato je dejanski čakalni čas $4ms + \lfloor \frac{period}{4ms} \rceil \cdot 4ms$. Standardna deviacija latence je pol manjša napram Linux 2.4 jedru OS.

4.4.2 Meritve pri obremenjenem sistemu

Naslednji grafi prikazujejo izmerjene čase spanja (y os) glede na zahtevanih $20ms$ tekom 1000 zaporednih meritev (x os) opravljenih druga takoj za drugo v diskretnem času pri obremenjenem sistemu.

Meritve dejanskega časa spanja napram ukazanim $20ms$ na jedru OS Linux *2.4.33.3* (glej 4.4) dajo naslednji rezultat:

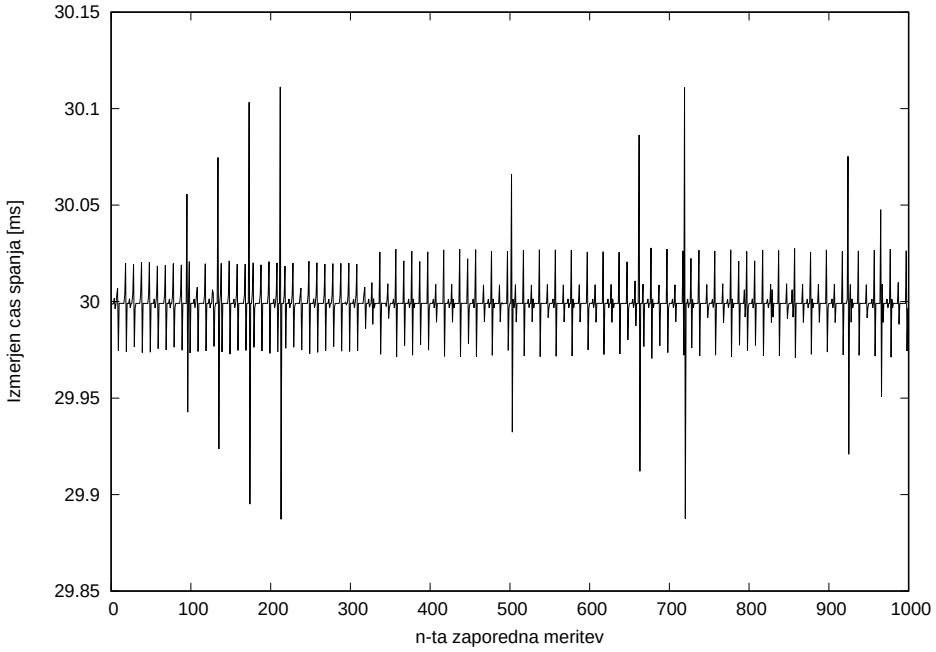
$$\begin{aligned} t_{\text{čas spanja}} &= 30.00ms \pm 0.43ms, \\ t_{\min} &= 27.12ms, \\ t_{\max} &= 32.87ms \end{aligned}$$



Slika 4.4: Obremenjen sistem na jedru OS Linux *2.4.33.3*.
Pozor: x os ne seka y osi pri vrednosti 0!

V povprečju so latence relativno majhne, vendar se pojavlja periodičen vzorec z večjimi latencami. Te so posledica periodičnih operacij na sistemu navideznega pomnilnika, saj nekaterih operacij tega dela sistema ni možno prekiniti. V kolikor gre za daljšo operacijo, se to odraža na večjih latencah. Če sem bremenu odstranil izvajanje operacij navideznega pomnilnika, so bile latence naslednje:

$$\mathrm{t}_{\text{čas spanja}} = 30.00ms \pm 0.02ms,$$
$$\mathrm{t}_{\min} = 29.88ms,$$
$$\mathrm{t}_{\max} = 30.11ms$$



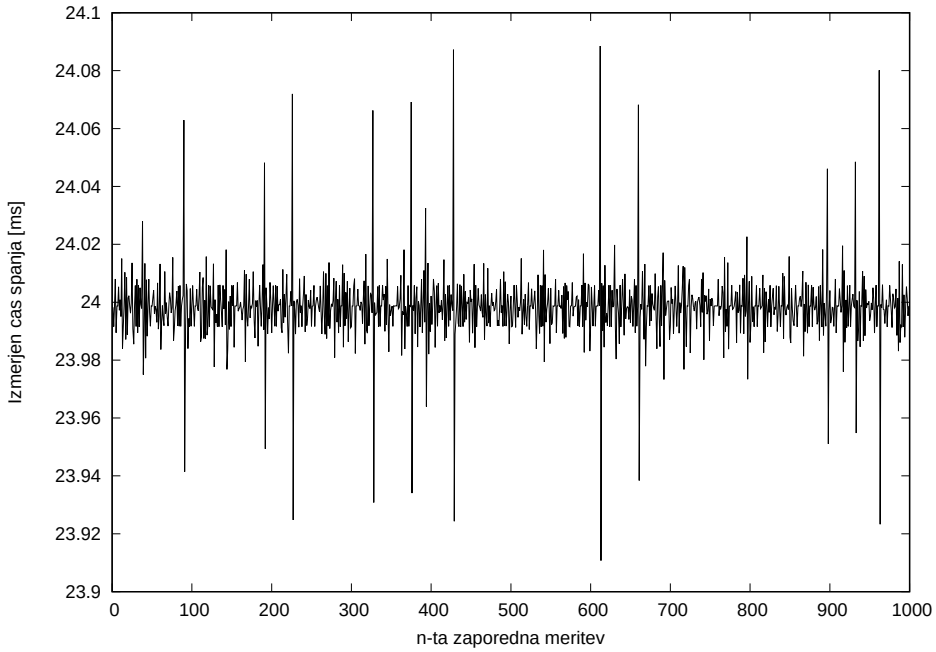
Slika 4.5: Obremenjen sistem na jedru OS Linux *2.4.33.3* brez bremena operacij navideznega pomnilnika.
Pozor: x os ne seka y osi pri vrednosti 0!

Graf 4.5 pokaže, da v kolikor ni operacij, ki se jih ne da prekiniti, to ne

vpliva na latenco - rezultati so podobni kot pri neobremenjenem sistemu 4.2.

Rezultati meritev na novejšem jedru OS Linux *2.6.17.13* (glej 4.6) so naslednji:

$$\mathrm{t_{\check{c}as\ spanja}} = 24.00ms \pm 0.01ms,$$
$$\mathrm{t_{min}} = 23.91ms,$$
$$\mathrm{t_{max}} = 24.09ms$$



Slika 4.6: Obremenjen sistem na jedru OS Linux *2.6.17.13*.
Pozor: x os ne seka y osi pri vrednosti 0!

Latence kljub obremenitvi ostanejo spoštljivo majhne, jedro OS Linux *2.6* jasno kaže napredek s stališča odzivnosti napram jedru OS Linux *2.4*.

Poglavje 5

Zaključek

Razvoj razvrščevalnikov v splošnonamenskih operacijskih sistemih kaže izjemen napredek, predvsem kar se tiče odzivnosti in podpore SMP. Veliko pozornosti je bilo vloženega v identifikacijo in optimizacijo odzivnosti interaktivnih procesov, saj se le ta direktno kaže uporabniku kot odzivnost celotnega sistema. S pojavom in splošno razširitvijo večjedrnih procesorjev je postala podpora SMP sistemom razvrščevalnika dandanes praktično nujna.

V sistemih z več CPE postane vprašanje porabe električne energije vedno bolj pereče – v primeru nezasedenosti več CPE bi bila poraba energije za ničelno delo relativno velika. Zato je nekatere dele modernih CPE možno (deloma) izklopiti in tako varčevati z energijo, vendar pa sta za ponoven zagon CPE potreben dodaten čas (večja latenca) in energija. Prepogosto in kratkotrajno varčevanje CPE z energijo bi se lahko odražalo v velikih neodzivnosti sistema in celo večji porabi energije. Potrebno je optimirati oziroma predvidevati, kdaj se splača preklopiti CPE v varčevalen način, in časovno celo prilagoditi izvajanje procesov. Prav to pa je smernica v kateri se že opaža nadaljnji razvoj razvrščevalnikov.

Dodatek A

Izvorna koda

A.1 latency.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sched.h>

/**
 * Preberi RDTSC register.
 * OPOZORILO: RDTSC register je vezan na Intel Pentium procesorje.
 */
unsigned long long int rdtsc() {
    unsigned long long int tsc;
    __asm__ __volatile__("rdtsc" : "=A" (tsc));
    return tsc;
}

/**
 * Nastavi realtime prioriteto procesa.
 */
```

```
int set_realtime_priority(int policy, int max) {
    struct sched_param schp;

    memset(&schp, 0, sizeof(schp));
    if (max)
    schp.sched_priority = sched_get_priority_max(policy);
    else
    schp.sched_priority = sched_get_priority_min(policy);

    if (sched_setscheduler(0, policy, &schp) != 0)
      {
    perror("sched_setscheduler");
    return -1;
      }

    return 0;
}

/**
 * Doloci frekvenco CPE (frekvenca je potrebna pri uporabi RDTSC).
 */
double determine_cpu_hz(void) {
  FILE *f;
  char *res;
  char s1[100];
  double cpu_hz = 0;

  /* preberi CPE frekevenco iz /proc/cpuinfo */
  f = fopen("/proc/cpuinfo","r");
  if (!f)
  {
    perror("open");
    exit(1);
```

```
  }

  /* izlusci frekvenco */
  for(;;)
  {
    res = fgets(s1, 100, f);
    if (!res) break;
    if (!memcmp(s1, "cpu MHz", 7))
      {
    cpu_hz = atof(&s1[10]) * 1000000.0;
    break;
      }
  }

  fclose(f);

  if (!cpu_hz)
    {
      perror("determine_cpu_hz");
      exit(1);
    }

  return cpu_hz;
}



#define SAMPLES 1000
#define U_SLEEP_TIME 20000

int main(int argc, char**argv) {
  int i;
  double start;
  double stop;
```

```
  double deltas[SAMPLES];
  double cpu_mhz = determine_cpu_hz() / 1000000.0;

  /* nastavi maksimalno realtime prioriteto */
  set_realtime_priority(SCHED_FIFO, 1);

  /* zahtevaj, da program ostane v delovnem pomnilniku */
  mlockall(MCL_CURRENT|MCL_FUTURE);

  /* vzorci */
  for (i=0; i < SAMPLES; i++)
  {

    start = rdtsc();

    /* spi */
    usleep(U_SLEEP_TIME);

    stop = rdtsc();

    /* izracunaj razliko v ms */
    deltas[i] = (stop - start) / cpu_mhz;
  }

  for (i=0; i < SAMPLES; i++)
    {
      // izpis v ms
      fprintf(stdout, "%d %f\n", i, deltas[i]/1000.0);
    }

  return 0;
}
```

Literatura

[1] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real time environment. *JACM, 20(1): 46-61*, 1973.

[2] Jia Xu and David Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Trans. Softw. Eng.*, 16(3):360–369, March 1990.

[3] David A. Solomon and Mark E. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, 2000.

[4] Scott A. Banachowski. Using the best-effort scheduling model to support soft real-time processing. *not published*, 2002.

[5] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchical cpu scheduler for multimedia operating systems. *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, 1996.

[6] Witold Jaworski Arnd C. Heursch, Alexander Koenen and Helmut Rzehak. Improving conditions for executing soft real-time tasks timely in linux. *Fifth Real-Time Linux Workshop*, 2003.

[7] Clifford W. Mercer. An introduction to real-time operating systems - scheduling theory. *not published*, 1992.

[8] Jeff Roberson. Ule: A modern scheduler for freebsd. *BSDCon '03*, 2003.

[9] Andreas Schlapbach. Linux process scheduling. *not published*, 2000.

[10] Hynek Schlawack. Der linux-o(1)-scheduler. *not published*, 2003.

[11] Till Straumann. Open source real time operating systems overview. *8th International Conference on Accelerator and Large Experimental Physics Control Systems*, 2001.

[12] Clark Williams. Linux scheduler latency. *white paper*, 2002.

[13] The FreeBSD Documentation Project. *FreeBSD Architecture Handbook*. The FreeBSD Documentation Project, 2002.

[14] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Longman, Inc, 1996.

[15] et al Tigran Aivazian. *Linux Kernel 2.4 Internals*. The Linux Documentation Project, 2002.

[16] Macro Cesati Daniel P. Bovet. *Understanding THe Linux Kernel*. O'Reilly, 2002.

[17] Herman Bruyninckx. Real-time and embedded guide, 2002.

[18] Matt Veber. Real-time operating systems - practical perspective, 1998.