PREDVIDLJIVO DINAMIČNI PRINCIP RAZVRŠČANJA OPRAVIL V REALNEM ČASU

INFORMATICA 3/92

Ključne besede: strogi namenski sistem, jedro OS za delo v realnem času, princip razvrščanja, časovne in logične prioritete opravil, idealna razvrstitev opravil, zaščita skupnih virov

Barbara Koroušić, Peter Kolbezen
Laboratorij za računalniške arhitekture
Inštitut Jožef Stefan, Ljubljana

Vloga operacijskih sistemov za delo v realnem času (OS) postaja pri namenskih računalniških sistemih vse pomembnejša. Programer se lahko posveti razvoju programske opreme na abstraktnem nivoju in prepusti odgovornost za upravljanje z opravili aplikacije sistemskim procesom jedra OS. V članku opisujemo takšen proces razvrščanja opravil, ki se istočasno potegujejo za vire računalnika. Podan je opis in analiza dinamičnega principa razvrščanja, ki dodeljuje prednost tistim opravilom, ki so za aplikacijo pomembnejša. Hkrati upošteva tudi časovno omejenost opravil in zagotavlja njihovo pravočasno izvršitev. Princip smo poimenovali *predvidljivo dinamični*, ker predvideva zavrnitev opravil takoj ob prihodu zahtev za njihovo izvajanje. O zavrnitvi govorimo, ko nabor pripravljenih opravil na procesorsko izvajanje ne omogoča pravočasne izvršitve opravila. Zavrnjeno opravilo lahko preusmerimo v drugo procesorsko vozlišče, če je namenski računalnik porazdeljeni ali večprocesorski sistem. Pomembna je čimprejšnja zavrnitev, tako da ima opravilo še dovolj časa za izvršitev vseh potrebnih operacij.

PREDICTABLE DYNAMIC TASK SCHEDULING IN HARD REAL-TIME - The essential feature of hard real-time systems is that the embedded computer must deliver its results correctly and on-time - otherwise malfunction may result. To assure software constructions of such systems a support of a real-time operating system is needed. We present a task scheduling and mutual exclusion protocol as one of real-time operating system structuring mechanisms. The protocol is applicable to aperiodic tasks having specific priority and timing characteristics.

1. Uvod

Računalniški sistem, ki je prilagojen okolju, v katerem izvaja nadzor, imenujemo *namenski* ali *vloženi* (*embedded*) sistem. Njegova osnovna naloga je spremljanje zunanjih dogodkov in odločanje o nadaljnem dogajanju v okolju. Ker je spreminjanje stanja v okolju dinamično in pogosto tudi nepredvidljivo, mora namenski računalnik delovati pod pogoji realnega časa:

1. Odločitve morajo biti **logično** in **časovno** pravilne.
2. Sistem mora delovati **varno** tudi ob pojavitvi morebitnih napak.

Namenske računalnike delimo glede na stopnjo upoštevanja zahtev realnega časa na *stroge* (*hard*) in *mehke* (*soft*) sisteme [1].

Potniško letalo A-320 [2] je primer sistema, ki je pod nadzorom *strogega* namenskega računalnika. Po obsegu okolja, nad katerim ima računalnik nadzor, so verjetno največji namenski sistemi v elektrarnah, kemijsko predelovalni industriji ali v upravljanju prometa. Manjši namenski sistemi so mobilni roboti, ki jih uporabljajo za opravljanje težaških, monotonih ali nevarnih opravil [3].

Razvoj programske opreme namenskih sistemov je bil do nedavnega prepuščen izkušenim programerjem, ki so imeli bogato znanje o strojni opremi sistema. Dandanes pa poznamo že vrsto

- operacijskih sistemov (npr. VRTX32, FlexOS [4]),
- visokonivojskih programskih jezikov (npr. Ada [5], Modula-2 [6], Real-Time Euclid [7]),
- CASE (*Computer Aided Software Engineering*) orodij (npr. VRTXdesigner [8]),

ki so prilagojeni zahtevam realnega časa. Uporaba tovrstnih orodij poenostavi oblikovanje namenskih

sistemov in hkrati zagotavlja večjo učinkovitost razvite kode.

Uporaba CASE orodij vodi oblikovanje programske opreme iz definicije problema. Podatkovne in nadzorne strukture visokonivojskih jezikov za programiranje pod pogoji realnega časa omogočajo razbitje problema na abstraktnem nivoju. Sistemske funkcije jedra OS (*real-time kernel*) pa prevzamejo odgovornost za upravljanje z opravili, ki predstavljajo osnovne abstraktne enote programske aplikacije.

Eden najpomembnejših sistemskih procesov jedra je *razvrščanje* sočasnih opravil aplikacije računalniškega sistema. Dogodki v okolju namreč lahko zahtevajo sočasen odziv sistema, ne glede na vrstni red prihoda zahtev za izvajanje opravil. Naloga razvrščevalnika je spremljanje dogodkov v okolju in odločanje o vrstnem redu izvajanja prebujenih opravil, saj večina namenskih računalnikov ne omogoča istočasnega izvajanja vseh zahtevanih opravil.

Princip razvrščanja (*scheduling policy*) je pogojen z aplikacijo. V preprostih aplikacijah, kjer je celotno dogajanje v sistemu predvidljivo, so zelo učinkoviti *statični* principi, ki določajo vrstni red izvajanja opravil z upoštevanjem statičnih (nespremenljivih) lastnosti opravil. Večina namenskih aplikacij pa mora upoštevati dinamično spreminjanje lastnosti opravil. Takoimenovani *dinamični* principi razvrščanja določajo zaporedje izvajanja prebujenih opravil med samim izvajanjem aplikacije.

Razvrščevalnik lahko teži k *idealnem* (*feasible*) ali *optimalnem* razvrščanju opravil. V prvem primeru je razvrstitev pripravljenih opravil čakajočih na procesor določena tako, da nobeno opravilo ne more preseči vnaprej določene časovne omejitve (*deadline*). Pri optimalni razvrstitvi pa so dovoljene minimalne zakasnitve.

V nadaljevanju bomo predstavili *predvidljivo dinamični* princip razvrščanja, ki določa prioritete prebujenih opravil z upoštevanjem njihove **časovne omejenosti** ter **stopnje pomembnosti** za celotno aplikacijo. Časovna omejenost in pomembnost opravil sta namreč kriterija, ki se lahko medsebojno izključujeta. Opravilo, ki je časovno kritično, ima lahko nizko stopnjo pomembnosti za aplikacijo in obratno. Princip teži k *idealnem* razvrščanju opravil.

2. Opis PD principa razvrščanja opravil

PD princip razvrščanja opravil temelji na sledečih predpostavkah:

1. Stanje namenskega sistema se lahko spreminja tudi asinhrono.
2. Odziv na dogodke je časovno omejen.
3. Opravila aplikacije so različno pomembna za delovanje sistema.
4. Dovoljeno je prekinjanje izvajanja opravil.
5. Dostop do skupnih virov je zaščiten.
6. Izvajanje namenske aplikacije mora biti robustno.

1. Dogodki, ki sprožijo izvajanje opravil v računalniškem sistemu, se pojavljajo sinhrono ali asinhrono. Trenutek, v katerem sproži asinhroni dogodek izvajanje opravila, ni znan pred zagonom aplikacije. Asinhroni dogodki lahko sprožijo izvajanje periodičnih ali aperiodičnih opravil.

2. Dogodki v sistemu zahtevajo pravočasen odziv. Vsako opravilo je časovno omejeno. Če računalniški sistem ne more zagotoviti odziva v predvidenem (odzivnem) času, je opravilo zavrnjeno.

3. Prednost pri izvajanju imajo opravila, ki so bolj pomembna za aplikacijo. Če procesor ni zmožen pravočasno izvesti vseh prebujenih opravil, razvrščevalnik zavrne najprej opravila z najmanjšo stopnjo pomembnosti.

4. Ker so dogodki večinoma asinhroni in opravila časovno omejena, mora princip podpirati prekinjanje opravil. Časovne lastnosti opravil se dinamično spreminjajo in tako se tudi prednost opravil pri dostopu do procesorja spreminja s časom izvajanja aplikacije. Prekinjanje izvajanja opravil je dovoljeno, če je prioriteta prebujenega opravila višja od prioritete opravila, ki se trenutno izvaja.

5. Opravila lahko zahtevajo dostop do skupnih virov. Metoda medsebojnega izključevanja (*mutual exclusion*) z uporabo *monitorja*, zagotavlja zaščito skupnih virov [9]. Ker pa smo želeli zadostiti tudi zahtevi po možnosti prekinjanja opravil, smo princip monitorja razširili in ga priredili za uporabo v

strogih namenskih sistemih. Podrobnejši opis sledi v poglavju 2.1.

6. Robustnost izvajanja namenskih aplikacij smo želeli zagotoviti posredno s principom razvrščanja. Opravilo, ki poruši *idealno* razvrstitev prebujenih opravil, je zavrnjeno že ob prihodu dogodka, ki je sprožil izvajanje opravila. Če je namenski sistem večprocesorski, se lahko opravilo preusmeri v drugo procesorsko vozlišče. Zato je zelo pomembno, da je zavrnitev objavljena, ko ima opravilo še dovolj časa, da opravi vse operacije. Upoštevali smo tudi morebitne napake pri izračunu operacij opravil. Če ima opravilo še dovolj časa, lahko ob pojavitvi napake ponovi niz operacij z alternativnimi procedurami (razširjeni mehanizem *okrevanja po napaki (recovery block programming)* [1]).

Rezultati matematične in simulacijske analize, ki sta jih objavila Craig in Woodside [10], kažejo na učinkovitost dinamičnega *principa prednosti časovno kritičnih opravil* (*Earliest Deadline As Soon As Possible*) [1]. Princip dodeljuje prednost opravilom, ki so časovno bolj kritična. Ker princip ne upošteva pomembnosti opravil za aplikacijo, smo vgradili dodaten mehanizem „izločanja" manj pomembnih opravil v primeru nasičenja v vrsti pripravljenih opravil (*ready queue*). To je programska struktura, ki hrani zahteve za izvajanje prebujenih opravil v vrstnem redu, kot ga določa razvrščevalnik.

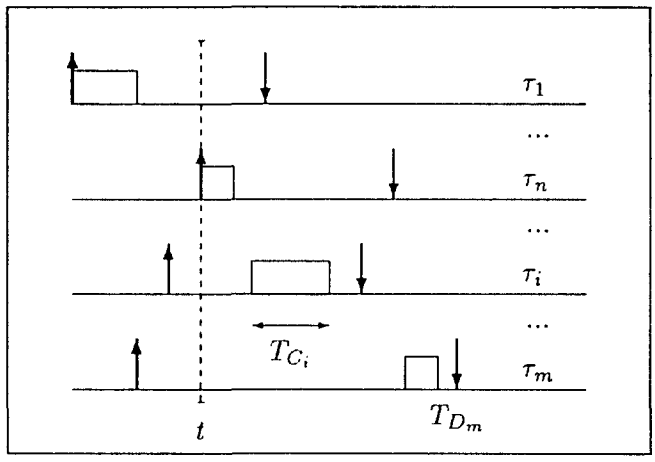
Ob prihodu nove zahteve za izvajanje opravila τ_n razvrščevalnik preveri pogoj:

$$(T_{D_k} - \sum_{i=1}^{k} T_{C_i}) \geq t,\ n \leq k \leq m, \qquad (1)$$

ki zagotavlja *idealno* razvrstitev nabora m prebujenih aperiodičnih opravil. T_{D_k} predstavlja čas, do katerega se mora izvršiti opravilo τ_k ter T_{C_i} predvideni čas izvajanja opravila τ_i. Indeksi opravil so določeni z njihovo časovno kritičnostjo, $i < j$ pri pogoju $T_{D_i} \leq T_{D_j}$. t je čas prihoda zahteve za izvajanje novega opravila τ_n (**slika 1**).

Razvrščevalnik preveri pogoj (1) le za vrednosti indeksa $k \geq n$, saj je pri nižjih vrednostih indeksa zanesljivo izpolnjen.

Če je pogoj izpolnjen, poteka razvrščanje opravil po *principu prednosti časovno kritičnih opravil.*



Slika 1: Časovne lastnosti in urejenost opravil

Prednost pri dostopu do procesorja ima opravilo, ki je časovno najbolj omejeno. V nasprotnem primeru pa se vključi mehanizem za "izločanje„ manj pomembnih opravil.

Recimo, da pogoj ni izpolnjen pri vrednosti indeksa $k = j$, $n \leq j \leq m$. Potem razvrščevalnik izloči iz vrste opravilo, katerega stopnja pomembnosti je najmanjša v podmnožici prebujenih opravil $\{\tau_1, \tau_2, ...\tau_j\}$ ter ponovno preveri pogoj pri vrednosti indeksa $k \geq j$.

Biyabani in Stankovic [11] sta dokazala, da je princip izločanja najmanj pomembnih opravil najbolj učinkovit. Druga možnost je izločanje opravil, katerih časovna prioriteta je najmanjša in hkrati stopnja pomembnosti ne preseže stopnje pomembnosti opravila τ_n. V primerjavi s prvo možnostjo, ki smo jo označili kot najbolj učinkovito, je slednja časovno manj potratna. Postopek izločanja opravil se ponavlja, dokler pogoju (1) ni zadoščeno.

Ker so predvideni časi izvajanja opravil določeni za „najslabši" (najdaljši) primer, razvrščevalnik dejansko ne izloči opravil iz vrste prebujenih opravil, temveč jih prestavi na konec vrste in obvesti sistem o predvideni zakasnitvi odziva na dogodke, ki so sprožili zahteve za izvajanje teh opravil. Sistem lahko zahteva dejansko izločitev teh opravil iz vrste pripravljenih opravil. Če je dejanski čas izvajanja opravila krajši od predvidenega, razvrščevalnik ponovno preveri pogoj (1) po zaključenem izvajanju opravila.

Če je pogoju (1) zadoščeno, je zagotovljeno *idealno* razvrščanje opravil, četudi je procesor 100 %

zaseden:

$$U = \frac{\sum_{i=1}^{m} T_{C_i}}{T_{D_m}} \leq 1 - \frac{t}{T_{D_m}}, \qquad (2)$$

saj je časovna meja T_{D_m} zmeraj manjša od časa t in velja:

$$\lim_{m \to \infty} 1 - \frac{t}{T_{D_m}} = 1. \qquad (3)$$

2.1 Mehanizem za zaščito skupnih virov

Pri opisu zahtev smo omenili, da monitor, kot mehanizem za zaščito skupnih virov, ne zagotavlja pravočasnega izvajanja namenskih aplikacij v realnem času, ker ne dopušča prekinjanja izvajanja opravil v kritičnih področjih [12]. Izogniti se moramo tudi težavam, ki nastopijo ob blokiranju (časovno in logično) pomembnih opravil, zaradi zasedenosti kritičnih področij z manj pomembnimi opravili:

- verižno blokiranje (*chained blocking*) izvajanja opravil,
- problem mrtve zanke (*deadlock*).

Opravilo je blokirano, kadar čaka na nadaljevanje izvajanja operacij zaradi zasedenosti zahtevanega vira z manj pomembnim opravilom. Težave nastopijo, ko opravilo zahteva vstop v več kritičnih področij, ki so zasedene z manj pomembnimi opravili (verižno blokiranje). Kot primer podajmo situacijo, ko opravilo τ_C vstopi v kritično področje. Opravilo τ_B, ki je bolj pomembno, prekine izvajanje opravila τ_C ter vstopi v drugo kritično področje. Opravilo τ_A, ki je še bolj pomembno, prekine izvajanje opravila τ_B. Če zahteva opravilo τ_A vstop v obe kritični področji, se njegovo izvajanje blokira, ker sta področji zasedeni z opravili τ_C in τ_B.

Problemu mrtve zanke pri potegovanju za skupni vir se lahko izognemo z uporabo semaforjev [9]. Mrtva zanka pa se lahko pojavi tudi po vstopu opravil v kritična področja, ko opravila navzkrižno zahtevajo dostop do skupnih virov, ki so že zasedeni.

Princip monitorja temelji na uporabi semaforjev za zaščito kritičnih področij in signalov za sporazumevanje z opravili, ki zahtevajo uporabo skupnih virov [9]. Princip razširimo z dodatnim mehanizmom, ki omogoča varno prekinjanje izvajanja opravil v kritičnih področjih [12]. Poleg osnovnih lastnosti opravil, ki jih mora določiti programer pred zagonom aplikacije, zahteva mehanizem seznam semaforjev, katere mora opravilo upoštevati pred vstopom v kritična področja. Vsakemu kritičnemu področju določimo prioriteto, ki se med izvajanjem aplikacije dinamično spreminja. Prioriteta kritičnega področja je enaka najvišji prioriteti prebujenega opravila, ki lahko zahteva vstop v to kritično področje.

Mehanizem za varno prekinjanje izvajanja opravil v kritičnih področjih razdelimo v dva dela. Prvi del omogoča dinamično določanje prioritet opravil v kritičnih področjih. Opravilo, ki ima trenutno dostop do skupnega vira, lahko blokira izvajanje višje prioritetnih opravil. V takšnem primeru prevzame opravilo v kritičnem področju najvišjo prioriteto blokiranih opravil. Ko izstopi iz kritičnega področja, dobi ponovno "originalno„ prioriteto. Drugi del upravlja s kritičnimi področji. Opravilo, ki zahteva ključ za vstop v kritično področje, mora ustrezati pogoju, da je njegova prioriteta višja od vseh prioritet trenutno zasedenih kritičnih področij. Sicer razvrščevalnik blokira izvajanje opravila in čaka na vstop v zasedeno področje.

Opišimo dva možna načina blokiranja opravil, ki zahtevajo dostop do skupnih virov:

1. Opravilo τ zahteva ključ za vstop v kritično področje S, vendar ga ne dobi, ker je njegova prioriteta nižja ali enaka najvišji prioriteti zasedenih področij, S_H. Opravilo τ je blokirano, dokler se področje S_H ne sprosti, opravilo v S_H pa prevzame prioriteto opravila τ, če je njegova prioriteta nižja.
2. Opravilo τ je blokirano, ker zahteva vstop v področje S in ima prioriteto nižjo ali enako najvišji prioriteti zasedenih področij, S_H. Če je opravilo v področju S_H že prevzelo višjo prioriteto od predhodnje blokiranega opravila, potem ostane njegova prioriteta nespremenjena.

Dokaz, da mehanizem onemogoča verižno blokiranje izvajanja opravil, temelji na sledečih lemah [12].

LEMA 1: *Če opravilo τ blokira izvajanje višje prioritetnega opravila τ_h, potem τ zaseda vsaj eno kritično področje S s prioriteto višjo ali enako prioriteti τ_h v času prihoda opravila τ_h (t).*

Dokaz: Predpostavimo, da je najvišja prioriteta kritičnih področij, ki jih zaseda τ v času t, prioriteta področja S_H. Če je ta prioriteta nižja od prioritete opravila τ_h, potem lahko opravilo τ prevzame le prioriteto, nižjo od prioritete opravila τ_h. Drugače rečeno, dokler je opravilo τ_h aktivno (prebujeno), opravilo τ ne bo razvrščeno tako, da bi blokiralo τ_h. Torej obstaja vsaj eno področje, ki ga zaseda τ, s prioriteto večjo ali enako prioriteti τ_h, v času t ♠.

LEMA 2: *V vsakem trenutku t obstaja največ eno kritično področje s prioriteto višjo ali enako P, ki ga zaseda opravilo z „originalno" prioriteto nižjo od P.*

Dokaz: Predpostavimo, da lema ne velja. V času t sta področji S_i in S_j zasedeni in imata obe prioriteto višjo ali enako P. Dodatno predpostavimo, da je najprej opravilo τ_p zasedlo področje S_i v času t_p in nato opravilo τ_q področje S_j v času t_q. Prioriteti opravil τ_p in τ_q sta nižji od P. Upoštevajoč princip našega mehanizma mora biti prioriteta opravila τ_q višja od prioritete kateregakoli zasedenega področja v času t_q. Ker pa zahtevamo, da je prioriteta opravila τ_q zmeraj nižja od P in prioriteta področja S_i višja ali enaka P, pridemo v nasprotje ♠.

S pomočjo lem smo dokazali, da lahko opravilo τ blokirajo le opravila z nižjimi prioritetami, ki zasedajo področja, katerih prioritete so višje ali enake prioriteti opravila τ (**LEMA 1**). V času prihoda opravila τ je zasedeno največ eno kritično področje s prioriteto višjo od prioritete τ, ki ga zaseda opravilo z nižjo prioriteto (**LEMA 2**). Tako smo dokazali, da lahko opravilo τ blokira največ eno opravilo z nižjo prioriteto in pogoj za verižno blokiranje nikoli ni izpolnjen.

Pri izboru mehanizma za varno prekinjanje izvajanja opravil v kritičnih področjih, smo se želeli izogniti tudi mrtvim zankam. Pogoj za mrtvo zanko je navzkrižno čakanje opravil, ki zasedajo kritična področja, medtem ko čakajo na semaforje področij, ki jih zasedajo ostala opravila, sodelujoča v zanki.

Predpostavimo, da obstaja mrtva zanka in ima opravilo τ najvišjo prioriteto med opravili v zanki. τ zaseda področje medtem, ko čaka na semafor področja, ki je zasedeno z drugim (nižje prioritetnim) opravilom iz zanke. Dokažemo pa lahko, da opravilo, ki zaseda področje, ne more biti blokirano z nobenim nižje prioritetnim opravilom [12] (**LEMA 3**). Tako pogoj za pojavitev mrtve zanke ne more biti izpolnjen.

LEMA 3: *Opravilo je lahko blokirano le pred vstopom v prvo kritično področje.*

Dokaz: Ko opravilo τ vstopi v prvo kritično področje v času t, ima najvišjo prioriteto med prebujenimi opravili. Ostala zasedena področja, ki jih zasedajo nižje prioritetna opravila v času t, označimo z množico S. Prioriteta opravila τ je zagotovo višja od prioritet področij iz S. Predpostavimo, da bo izvajanje opravila τ blokirano v času t_1, $t_1 > t$, ko se bo začelo izvajati opravilo z nižjo prioriteto τ_l. Opravilo τ_l bo začasno prevzelo prioriteto od opravila τ. Upoštevajoč princip mehanizma lahko opravilo τ_l zaklene področje S_k, če je prioriteta opravila τ nižja ali enaka prioriteti S_k. Ker nobeno opravilo ni bilo razvrščeno v času med t in t_1, je množica zasedenih področij v času t_1 enaka S in $S_k \in S$. Tako pridemo v nasprotje, saj nobeno področje iz S nima prioritete višje ali enake od prioritete opravila τ ♠.

Razvrščevalnik mora preveriti pred vstopom opravila v kritično področje, ali je njegova prioriteta višja od najvišje prioritete zasedenih področij. Ker pa smo v zgornji lemi dokazali, da je opravilo lahko blokirano le pred vstopom v prvo kritično področje, moramo preveriti pogoj le pred vstopom v prvo področje. Nadaljne zahteve za dostop do skupnih virov izvaja razvrščevalnik brez preverjanja pogoja, ali je prioriteta opravila višja od najvišje prioritete zasedenih področij.

Predstavljeni mehanizem za zaščito skupnih virov opravil vpliva na potek izvajanja opravil. Blokiranje izvajanja opravil zaradi zasedenosti kritičnih področij vpliva na odzivni čas, kar moramo upoštevati pri preverjanju pogoja o idealni razvrstitvi prebujenih opravil. V pogoj (1) vpeljemo dodatni parameter B_k:

$$(T_{D_k} - (\sum_{i=1}^{k} T_{C_i} + B_k)) \geq t,\ 1 \leq k \leq m, \qquad (4)$$

ki predstavlja najdaljši čas blokiranja izvajanja opravila τ_k. Za vsako opravilo τ_k določimo množico Z_k, ki vsebuje seznam kritičnih področij, do katerih imajo dostop nižje prioritetna opravila in je njihova prioriteta višja ali enaka prioriteti opravila τ_k. Drugače povedano, Z_k vsebuje vsa področja, ki jih lahko zasedejo nižje prioritetna opravila v trenutku zahteve opravila τ_k ali opravila z višjo prioriteto za vstop v ta področja. Seznami Z_k, $k = 1, ..., m$ se spreminjajo s časom. Ker lahko opravila blokirajo opravila z nižjo prioriteto, moramo pogoj preveriti pri vseh vrednostih indeksa k. Pogoj zagotavlja *idealno* razvrščanje nabora prebujenih opravil.

PRIMER: Oglejmo si primer *idealnega* razvrščanja treh aperiodičnih opravil z lastnostmi:

opravilo	T_A	T_C	T_D	*kritičnost*	*semaforji*
τ_1	3	2	6	2	$[S_1, S_2]$
τ_2	1	2.5	7	1	$[S_1, S_3]$
τ_3	0	3	8	3	$[S_2, S_3]$

Tabela 1: Lastnosti treh aperiodičnih opravil

kjer parameter T_A predstavlja čas prihoda zahteve za izvajanje opravila, T_C preostali izvajalni čas opravila in T_D trenutek, do katerega se mora opravilo zagotovo izvršiti, sicer je zavrnjeno.

Razvrščevalnik ureja prebujena opravila v vrsti pripravljenih opravil RQ po PD principu.

V času $t = 0$, ob prihodu zahteve za izvajanje opravila τ_3, je vsebina vrste $RQ = [\tau_3]$. Opravilo se takoj prične izvajati, saj je procesor prost.

Ob prihodu nove zahteve za izvajanje opravila τ_2, v času $t = 1$, vsebuje vrsta $RQ = [\tau_2, \tau_3]$, tako da ima opravilo τ_2 prioriteto $p(\tau_2) = 1$ in $p(\tau_3) = 2$. Opravilo $p(\tau_2)$ ima višjo prioriteto od $p(\tau_3)$, ker je časovno bolj omejeno. Prioritete kritičnih področij, v katera imata opravili vstop med izvajanjem, so sledeče: $c(S_1) = c(S_3) = p(\tau_2) = 1$ in $c(S_2) = p(\tau_3) = 2$. Pogoj (4) preverimo najprej za prvo opravilo v vrsti RQ (τ_2):

$$T_{D_1} - (T_{C_1} + B_1) = 7 - (2.5 + 0.5) > 1,$$

pri čemer sta T_{D_1} in T_{C_1} časovna parametra opravila τ_2. Najdaljši možni čas blokiranja opravila τ_2 $B_1 = 0.5$. To je čas zasedenosti področja S_3, do katerega ima dostop tudi nižje prioritetno opravilo τ_3. Ker je pogoju zadoščeno, preverimo pogoj pri vrednosti indeksa $k = 2$:

$$T_{D_2} - (\sum_{i=1}^{2} T_{C_i} + B_2) = 8 - (2.5 + 2 + 0) > 1.$$

Vrednost indeksa 2 označuje lastnosti opravila τ_3 in 1 lastnosti τ_2. Opravilo τ_3 ima nižjo prioriteto, zato njegovega izvajanja ne more blokirati τ_2 ($B_1 = 0$).

V času $t = 3$ se pojavi nova zahteva za izvajanje opravila τ_1. $RQ = [\tau_1, \tau_2, \tau_3]$, $p(\tau_1) = 1$, $p(\tau_2) = 2$ in $p(\tau_3) = 3$, $c(S_1) = c(S_2) = p(\tau_1) = 1$ in $c(S_3) = p(\tau_2) = 2$. Pogoju (4) ni zadoščeno pri vrednosti indeksa $k = 3$:

$$T_{D_1} - (T_{C_1} + B_1) = 6 - (2 + 0.5) > 3,$$

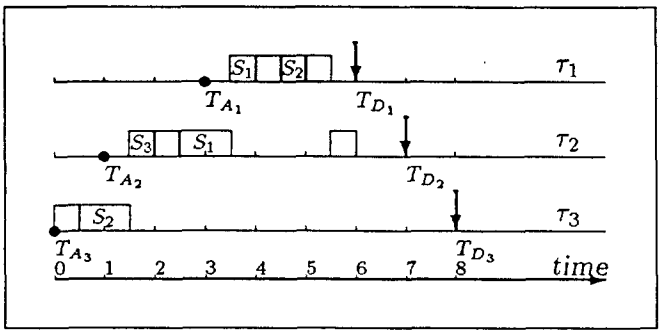
$$T_{D_2} - (\sum_{i=1}^{2} T_{C_i} + B_2) = 7 - (2 + 1 + 0.5) > 3,$$

$$T_{D_3} - (\sum_{i=1}^{3} T_{C_i} + B_3) = 8 - (2 + 1 + 2.5 + 0) < 3,$$

zato je potrebno izločanje manj kritičnih opravil iz nabora prebujenih opravil $\{\tau_1, \tau_2, \tau_3\}$. Vrednost indeksa $i = 1$ označuje parametre opravila τ_1, $i = 2$ opravila τ_2 in $i = 3$ opravila τ_3. $B_1 = 0.5$, ker lahko izvajanje opravila τ_1 blokira τ_2 v S_1 za 0.5 časovne enote. $B_2 = 0.5$, ker lahko izvajanje opravila τ_2 blokira τ_3 v S_3 za 0.5 časovne enote.

Slika 2 natančno prikazuje potek izvajanja opravil.

V času $t = 1$ prekine τ_2 izvajanje opravila τ_3. Ker se nahaja τ_3 v kritičnem področju S_2 in τ_2 ne zahteva dostop do skupnega področja, se izvajanje τ_3 v S_2 nadaljuje. Ko zahteva opravilo τ_2 v času $t = 1.5$ ključ za vstop v S_3, ga dobi, ker ni zasedeno nobeno področje. Opravilo τ_2 se ob prihodu zahteve za izvajanje τ_1 nahaja v področju S_1. Ker je $c(S_1) = p(\tau_1)$ in $p(\tau_1) > p(\tau_2)$, blokira opravilo τ_2 izvajanje τ_1 in prevzame do izstopa iz S_1 prioriteto 1.



Slika 2: Primer PD razvrščanja

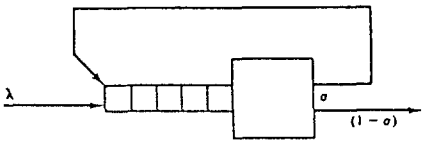
3. Primerjava PD principa z dinamičnimi principi razvrščanja

Osnovo PD principa razvrščanja predstavljata *princip prednosti časovno kritičnih opravil* (ED) in *princip kritičnih opravil* (*Priority Scheduling* - PS). Slednji dodeli prednost pri dostopu do procesorja opravilu, ki ima tisti trenutek najvišjo stopnjo pomembnosti v vrsti pripravljenih opravil. Stopnjo pomembnosti določi programer subjektivno in se le ta med izvajanjem aplikacije ne spreminja [1].

Pri snovanju PD principa razvrščanja smo težili k odpravi pomankljivosti ED in PS pristopov. Seveda pa smo želeli ohraniti tudi vse dobre lastnosti. V nadaljevanju bomo podali primerjalno analizo, ki je sestavljena iz dveh delov, matematičnega in simulacijskega. Matematični model je zahtevnejši za izpeljavo, vendar podaja zanesljive rezultate, ki temeljijo na primerjavi lastnosti posameznih principov pod enakimi pogoji. S pomočjo simulacije pa lahko preverimo rezultate matematične analize brez upoštevanja predpostavk, katere običajno uvedemo zaradi lažje izpeljave zahtevnega matematičnega modela.

3.1 Matematični model PD principa

Namenski računalniški sistem lahko predstavimo kot matematični model iz teorije vrst (slika 3). Okolje pošilja asinhrone zahteve za izvajanje opravil, ki se kopičijo v vrsti pripravljenih opravil. Zahteve so časovno neodvisne in časi med prihodi opravil so eksponentno porazdeljeni (*Markov process*). Prav tako so določeni tudi strežni in čakalni časi opravil z eksponentno porazdelitvijo. Hitrost prihoda zahtev je podana s parametrom λ, srednja vrednost dolžine strežnih časov z μ ter čakalnih časov z γ. Opravilo zapusti sistem, ko opravi vse zahtevane operacije. Verjetnost, da je opravilo prekinjeno in se mora vrniti v čakalno vrsto pripravljenih opravil je določena s parametrom $(1-\sigma)$.



Slika 3: Sistem z vrsto

Izhodni parametri našega matematičnega modela so naslednji:

1. Izkoriščenost procesorja.
2. Verjetnost zavrnitve opravil.
3. Lastnosti zavrnjenih opravil:
   - Preostali strežni čas.
   - Preostali čakalni čas.

1. Izkoriščenost procesorja ($U = \rho$) izračunamo s pomočjo dveh podatkov, povprečnega strežnega časa $E(T_C)$ ter povprečnega časa W, ki ga opravilo preživi v sistemu [1]:

$$\rho = \frac{E(T_C)}{W}. \tag{5}$$

Če upoštevamo, da so strežni in čakalni časi porazdeljeni eksponencialno:

$$\begin{aligned} g(x) &= \mu e^{-\mu x}, \\ l(x) &= \gamma e^{-\gamma x}, \end{aligned} \tag{6}$$

potem sledi izpeljava:

$$\begin{aligned} C(t) &= \int_0^t x g(x) \int_0^{t-x} l(y) dy dx, \\ H(t) &= \int_0^t g(x) \int_0^{t-x} l(y) dy dx, \end{aligned} \tag{7}$$

$$W'(t) = \frac{\lambda}{2(1-\lambda C(t))^2} \cdot \left(\int_0^t x^2 g(x) \int_0^{t-x} l(y)dydx + \int_0^t x^2 g(x)[1-H(x)]dx\right), \quad (8)$$

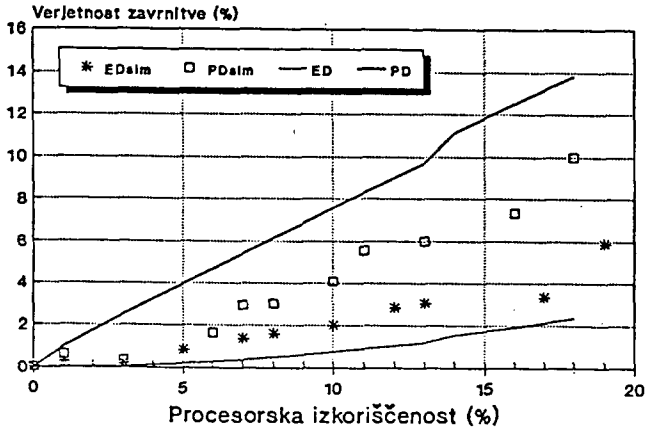
$$W''(t) = \int_0^{t_1} \frac{g(x)}{1-\lambda C(t_1)} \int_0^{t_1-x} l(y)dydx,$$
$$t_1 = W'(t), \quad (9)$$

$$W(t) = W'(t) + W''(t). \quad (10)$$

$W(t)$ je povprečni odzivni čas opravila, katerega predvideni odzivni čas je enak t. Povprečni čas opravila s poljubnim predvidenim odzivnim časom izračunamo s približkom vrednosti t s srednjo vrednostjo $E(t)$:

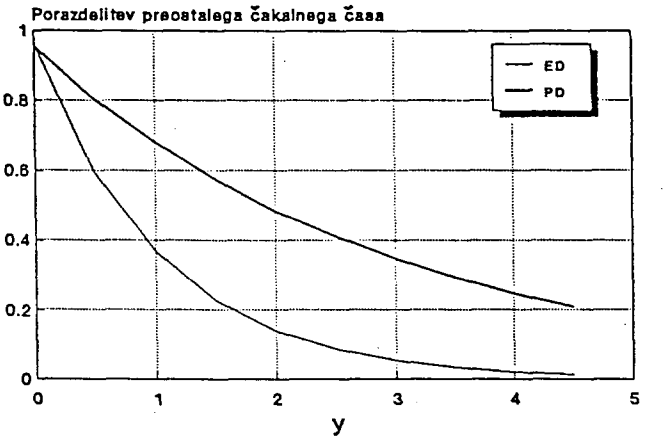
$$E(t) = \int_0^{\infty} t\, h(t)dt. \quad (11)$$

2. - 3. Podrobnejši opis analize verjetnosti zavrnitve opravil ter njihovih lastnosti smo podali v prispevku [13] ter v nalogi [1]. Zato podajmo na tem mestu le grafično primerjavo rezultatov analize dinamičnih principov razvrščanja (**graf 1, 2, 3**).

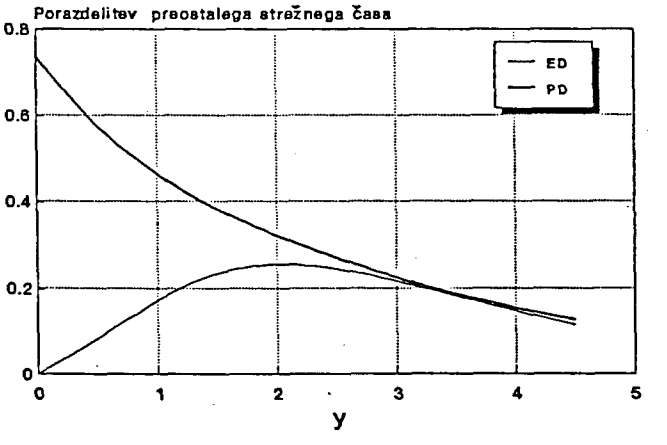


Graf 1: Verjetnost zavrnitve opravil

Pri analizi matematičnega modela smo privzeli zelo močno predpostavko, da se lahko v določenem



Graf 2: Porazdelitev preostalih čakalnih časov zavrnjenih opravil



Graf 3: Porazdelitev preostalih strežnih časov zavrnjenih opravil

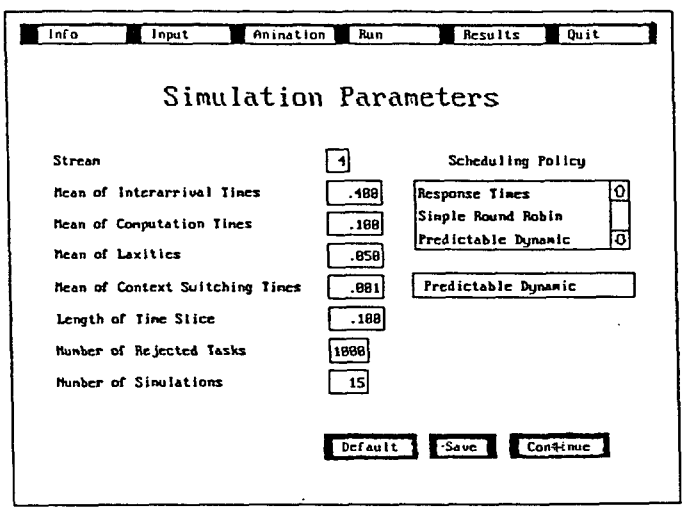
trenutku potegujeta za procesor največ dve opravili. Ker pa je procesorska izkoriščenost v *strogih* namenskih računalnikih nizka (največ 10 % [14]) in upoštevamo Markov proces prihoda zahtev za izvajanje opravil, rezultati analize ne odstopajo močno od dejanskih. To smo tudi potrdili s simulacijo, ki temelji na bolj realnih pogojih.

3.2 Simulacijski model PD principa

Program, ki omogoča simulacijo principov razvrščanja opravil, smo razvili v programskem jezi-

ku SimScript II.5 za PC kompatibilni računalnik. Uporabnik lahko spreminja sledeče parametre simulacij:

- razvrščevalni princip,
- dolžino simulacij in
- časovne parametre opravil (slika 4).



Slika 4: Vhodni simulacijski parametri

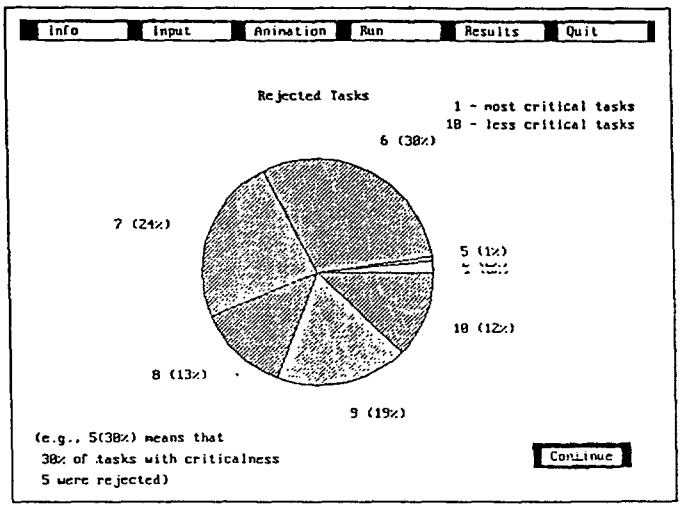
ter spremlja rezultate:

- verjetnost zavrnitve opravil pri določeni procesorski izkoriščenosti (**graf 1**),
- lastnosti zavrnjenih opravil (**graf 4**),
- uteženo verjetnost pravočasne izvršitve opravil (**graf 5**).

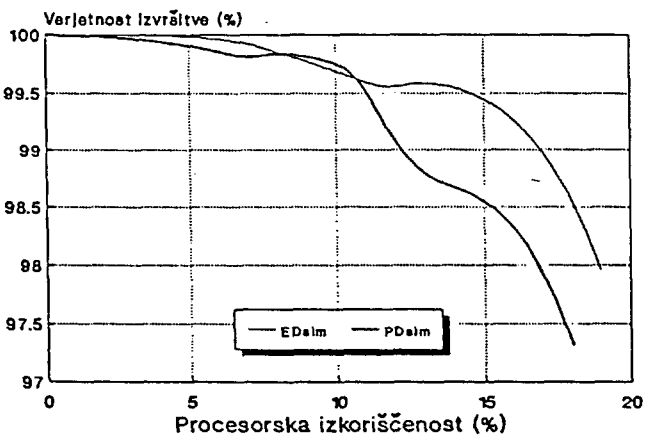
Mero utežene verjetnosti izvršitve opravil smo izračunali s pomočjo funkcije:

$$WGR = 100 \cdot \frac{\sum_i (c_i \cdot | \{\tau^i_{exe}\} |)}{\sum_i (c_i \cdot | \{\tau^i_{all}\} |)},$$
$$c_i = e^{i-1}. \qquad (12)$$

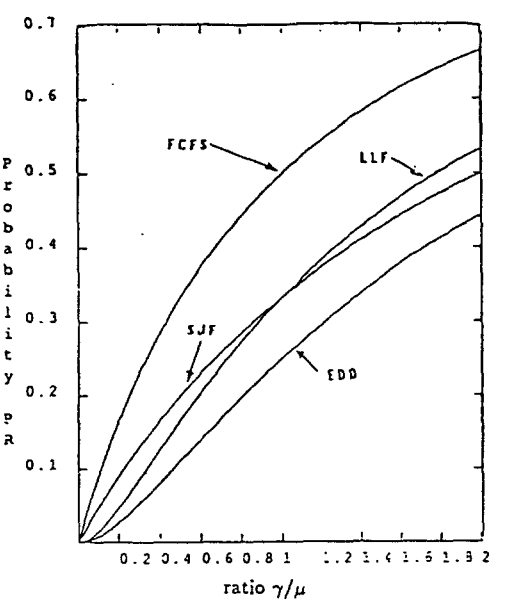
Množica $\{\tau^i_{exe}\}$ vsebuje opravila s stopnjo pomembnosti i, ki so se pravočasno izvršila in $\{\tau^i_{all}\}$ vsa prebujena opravila z i-to stopnjo pomembnosti za aplikacijo. Nižja vrednost indeksa i pomeni višjo stopnjo pomembnosti opravila.



Graf 4: Lastnosti zavrnjenih opravil



Graf 5: Utežena verjetnost izvršitve opravil



Graf 6: Verjetnost zavrnitve opravil

4. Zaključek

Predvidljivo dinamični princip razvrščanja opravil, na katerem temelji jedro OS za delo v *strogem* realnem času, upošteva zahteve realnega časa. To pomeni, da obravnava zahteve za izvajanje aperiodičnih opravil, ki so časovno in logično različno zahtevna za programsko aplikacijo.

V primerjavi z ostalimi dinamičnimi principi razvrščanja opravil (**graf 6** [10]) lahko povzamemo sledeče zaključke:

1. Princip PD zagotavlja pri določeni procesorski izkoriščenosti večjo verjetnost zavrnitve manj pomembnih opravil.

2. Verjetnost zavrnitve opravil z visoko stopnjo pomembnosti za aplikacijo je manjša.

3. Ker princip temelji na predvidljivosti, so lastnosti zavrnjenih opravil ugodnejše. Razvrščevalnik lahko preusmeri opravilo v drugo procesorsko vozlišče poraždeljenega ali večprocesorskega namenskega sistema, če je preostali čakalni čas še dovolj velik.

Področje večprocesorskih in porazdeljenih namenskih sistemov odpira nove težave, s katerimi se je potrebno soočiti. Naše nadaljne delo bo usmerjeno v raziskavo topologij namenskih sistemov s porazdeljenimi vhodno-izhodnimi napravami, ki omogočajo hitro in zanesljivo zaznavanje zunanjih dogodkov ter komunikacijo med opravili.

Literatura

[1] B. Koroušić. *Jedro operacijskega sistema za delo v strogem realnem času (magistrska naloga).* Univerza v Ljubljani, Fakulteta za elektrotehniko in računalništvo, april 1992.

[2] M. Thaler. Nova tehnologija v pilotski kabini. *KRILA*, page 28, maj 1989.

[3] L.S. McTamaney. Real-time intelligent control. *IEEE Expert*, 2(4):55–68, 1987.

[4] Operating systems in real time. *.EXE Magazine*, 4(7):72–77, 1989.

[5] W.A. Halang. Evaluation of ada from the viewpoint of control engineering. *IEEE*, pages 8–13, 1986.

[6] A. Brodnik in M. Špegel in T. Lasbaher. Programiranje z modulo-2. *Informatica, Ljubljana*, (2):78–82, 1987.

[7] E. Kligerman in A.D. Stoyenko. Real-time euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, 12(9):941–949, september 1986.

[8] Real-time software. *Micro Technology*, december 1991.

[9] J.E. Cooling. *Software Design for Real-Time Systems.* Chapman and Hall, 1990. ISBN 0-412-341-808.

[10] D.W. Craig in C.M. Woodside. The rejection rate for tasks with random arrivals, deadlines, and preemptive scheduling. *IEEE Transactions on Software Engineering*, 16(10), oktober 1990.

[11] J.A. Stankovic. Misconceptions about real-time computing. *COMPUTER*, 21(10):10–19, oktober 1988.

[12] M.I. Chen in K.J. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Real-Time Systems - The international journal of time-critical computing systems*, 2(4):325–345, november 1990.

[13] B. Koroušić in J.E. Cooling in P. Kolbezen. Predictable hard real-time scheduling. *In Proceedings of the* 4^{th} *EUROMICRO Workshop on Real-Time*, junij 1992.

[14] C.J. Paul in A. Acharya in B. Black in J.K. Strosnider. Reducing problem-solving variance to improve predictability. *COMMUNICATIONS OF THE ACM*, pages 81–93, avgust 1991.