RĪGAS TEHNISKĀ UNIVERSITĀTE

DATORZINĀTNES UN INFORMĀCIJAS TEHNOLOĢIJAS FAKULTĀTE

DATORVADĪBAS, AUTOMĀTIKAS UN DATORTEHNIKAS INSTITŪTS

Datoru tīklu un sistēmas tehnoloģijas katedra

“Reāla laika operētājsistēmu apskats”

Referāts priekšmetā Mūsdienu datoru arhitektūra

Darba autors:

Vitālijs Hodiko

091REB325, 2 gr.

Pārbaudīja:

prof. V. Zagurdskis

Rīga, 2013

Saturs

1. Ievads

Operētājsistēma (angl. Operating System - OS) ir atbildīga par programmu, kas darbojas uz ierīces un aparatūras resursu pārvaldību. Reāla laika operētājsistēma(angl. Real-Time operating system - RTOS) arī veic šos uzdevumus, bet tā bija speciāli izstrādāta, lai varētu kontrolēt procesu darbības laiku ar zināmu precizitāti. Tas var būt īpaši svarīgi, mērīšanas un automatizācijas sistēmas, kur dīkstāve ir dārga vai programmas kavēšanās var radīt draudus drošībai.

Pēdējos gados reāla laika operētājsistēmas(angl. Real-Time operating systems - RTOS) tiek plaši pielietotas uz mikroprocesoriem bāzētas iegultas sistēmas, piemēram, robotika, kameras, komunikācijā, militāros lietojumos.

Šajā darba tiek apskatītas RTOS un parastu operētājsistēmu galvenas īpašības, atšķirības, vājas un stipras puses, ka arī metodes, kas tiek pielietotas RTOS veiktspējas noteikšanai. Salīdzinot četras dažādas RTOS to veiktspējas ziņa, katrs no tiem strādāja labāk par citu veicot dažādus uzdevumus.

2. Reāllaika operētājsistēmas

Reāllaika operētājsistēma ir programmatūras kopums, ko izstrādātāji var izmantot, lai veidotu gatavas sistēmas. Lai OS varētu uzskatīt par reāllaika operētājsistēmu, katrai tajā izpildāmai operācijai jābūt zināms maksimālais darbības laiks, ko tā var garantēt. Dažas no šīm darbībām ietver OS izsaukumus un pārtraukumu apstrādi. Operētājsistēmas, kas var strikti garantēt maksimālo laiku šīm darbībām parasti dēvē par "strikti reālā laikā (angl. Hard real time)", bet OS, kas var garantēt lielāko daļu no paredzēta laika tiek saukti par "elastīgu reālā laikā (angl. Soft real time)". Protams, praksē katrs RTOS risinājums uzvedas dažādi vairākos lietojumos, tāpēc lietotājam rūpīgi jāizpēta visus ietekmējošus faktorus. Piemēram, iedomājaties kā tika uzprojektēta gaisa spilvena drošības sistēma jaunai automašīnai. Šajā gadījumā neliela kļūda laikā kontrolē(izraisot spilvena atvēršanos par agri, vai par vēlu), varētu kļūt katastrofāla.

Reāllaika operētājsistēmu pretstatam operētājsistēmas domātas personāliem datoriem tiek sauktas par vispārēja lietojuma operētājsistēmām (piemēram, Linux). Neiedziļinoties detaļas ir svarīgi atcerēties, ka abiem operētājsistēmu tipiem ir savas vājas un stipras puses. Operētājsistēmas, piemēram, Windows, ir paredzētas, lai vairākas vienlaicīgi darbojošās programmas saglabātu atsaucību, bet reāllaika operētājsistēmas ir paredzētas darbībai ar kritiskam lietojumprogrammām, kur operāciju izpildes laiks ir no svara.

Kaut arī RTOS iegultas sistēmās pārsvara tika izmantotas augstas klases mikroprocesoros ar 32 bitu procesoriem, pastāv pieaugoša tendence, nodrošināt šos līdzekļus arī 8 un 16 bitu sistēmām.

2.1 Vispārēja pielietojuma operētājsistēmas un reāllaika operētājsistēmas

Operētājsistēmas pārvalda resursu sadali datoru sistēmas. Atšķirība no vispārēja pielietojuma operētājsistēmām RTOS ir speciāli izstrādāts, lai nodrošinās reāllaika atsaucību.

RTOS piedāvā apsteidzošo, prioritātes balstītu plānošanu. Plānošanas shēma attiecas uz to, kā operētājsistēma piešķir CPU(Central Processing Unit – latv. Centrāla apstrādes vienība, Procesors) ciklus uzdevumu veikšanai. Tādējādi, plānošanas shēma tieši ietekmē uz to kā tiks izpildītas palaistas lietojumprogrammas. Lielākā daļa vispārēja pielietojuma operētājsistēmu ir laika sadales sistēmas, kas piešķir procesiem vienādus laika intervālus (piemēram, round-robin plānošana). Reāla laika operētājsistēma bieži piešķir procesiem prioritātes, un augstākas prioritātes uzdevumi var

pārtraukt zemākas prioritātes uzdevumus to izpildes laikā (angl. Preemptive scheduling). Cita veida plānošana paredz, ka izpildāmais process izmanto plānotāju, lai pārslēgtos starp citiem procesiem, ko sauc par kooperatīvu plānošanu (angl. Cooperative scheduling).

Reāla laika operētājsistēma ļauj veikt prognozējamu uzdevumu sinhronizāciju. Vispārēja pielietojuma sistēmas, procesu sinhronizācija nav prognozējama, jo operētājsistēma var tieši vai netieši ieviest papildus aizskāvi. RTOS tas nav pieļaujams, jo sistēmas procesiem jābūt zināms un sagaidāms izpildes garums.

RTOS galvena atšķirība no vispārēja pielietojuma operētājsistēmām ir tāda, ka RTOS atbalsta deterministisku uzvedību. Reāla laika operētājsistēmā uzdevuma nosūtīšanas laiks, uzdevuma pārslēguma un pārtraukumu latentums, jābūt laika paredzamam un konsekventam, pat tad, ja uzdevumu skaits palielinās. Turpretī, vispārēja pielietojuma operētājsistēmas (galvenokārt saistībā ar to laika sadales pieeju) samazina sistēmas vispārējo atsaucību un negarantē plānotāja izpildi noteiktā laika sprīdī, kad uzdevumu skaits palielinās. Dinamiska atmiņas piešķiršana (malloc() C valodā), ko plaši pielieto vispārējas operētājsistēmas, nav ieteicams RTOS, jo tas rada neprognozējamo uzvedību.[1] Tā vietā, RTOS sniedz fiksēta izmēra atmiņas sadali, kurā katram pieprasījumam tiek piešķirts noteikta izmēra atmiņas bloks.

Daudzi sistēmas dizaineri uzskata, ka maza mēroga iegultām sistēmām, kuros izmanto mikrokontrollerus ar zemam specifikācijām (t.i., mikrokontrolleris ar maksimālo ROM 128 KB un RAM 4 KB dauzumu) nav nepieciešams RTOS. Tomēr, reāllaika operētājsistēmas piedāvā ievērojamas priekšrocības šāda veida ierīcēm.[2, 3] Piemēram, izstrādātāji var izmantot RTOS izstrādes procesa optimizācijai. Izmantojot nelielus mikrokontrollerus, programmatūras ražīgums ir būtisks jautājums, jo tiek dots ierobežots laiks sistēmas izstrādei un to ieviešanai tirgu (2004.gada Embedded System Design veikta aptauja, pieejama http://www.embedded.com/columns/survey vietnē). Sarežģītiem projektiem, RTOS ir efektīvs instruments, kas dod iespēju viegli pārvaldīt programmatūru un sadalīt uzdevumus starp izstrādātājiem. Tāda veida programmētāji var izstrādāt gala produktu pa daļām, kas gan atvieglos, gan paātrinās gala produkta izstrādi.

2.2 Reāla laika operētājsistēmas īpašības un stipras puses

Reāla laika operētājsistēmas nodrošina labāku un drošāku sinhronizāciju. Maza mēroga iegultas sistēmas bez RTOS izmantošanas izstrādātāji bieži izmanto globālus mainīgos, lai realizētu komunikāciju un sinhronizāciju starp vairākām koda daļām. Taču tas var novest pie grūti izsekojamam kļūdām un koda drošuma problēmām, jo īpaši uz pārtraukumiem bāzētas sistēmas (2006.gada

Embedded System Design veikta aptauja, pieejama http://www.embedded.com/columns/survey vietnē). Galvenokārt, tas ir tāpēc, jo bieži vien šie globālie mainīgie ir pieejami vairākas vietas un to vērtības tiek mainītas dažādas funkcijas, kas padara tos par neaizsargātiem no iespējamam kļūdām programmas izpildes laikā. Kodam palielinoties, šīs kļūdas tiek paslēptas dziļāk un tos pamanīt kļūst aizvien sarežģītāk. Kā rezultāta, izstrādes laiks var ievērojami palielināties, pat būvējot maza mēroga sistēmu. Izmantojot RTOS, procesi var viegli nosūtīt datus vai sinhronizēties viens ar otru bez kļūdu rašanas iespējas.

Lielāka daļa reāla laika operētājsistēmu izstrādātāju nodrošina ar lietojumprogrammas saskarsni (angl. Application Program Interface – API), kas ļauj viegli pārvaldit sistēmas resursus, piemēram, procesu plānotais, atmiņas kopas pārvaldība, laika pārvaldība, pārtraukumu kontrole, komunikācija un sinhronizācija. RTOS ir uzrakstīts tāda veidā, ka ir vienkārši rakstīt viegli lasāmu kodu, pārnest to uz dažādam aparatūras platformām. Tas, savukārt, dod iespēju īsa laika sprīdi izstrādāt kvalitatīvus maza mēroga projektus.

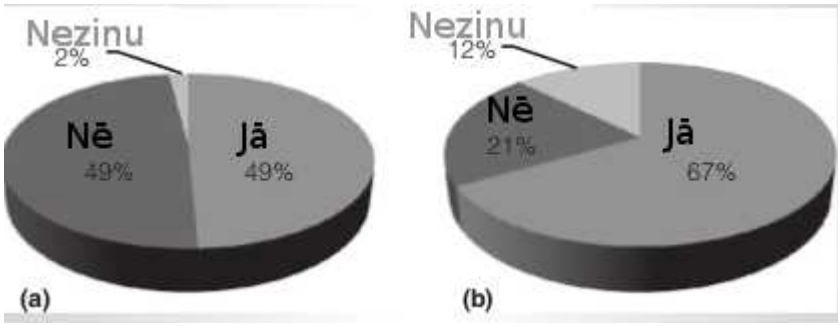
Laika kontroles funkcijas ļauj kontrolēt programmas izpildes laiku neiedziļinoties aparatūras īpašībās, piemēram, veikt uzdevumu aizkavi, palaist noteiktu uzdevumu konkrēta laika momenta. Mikrokontrolleros bez RTOS to izdarīt ir sarežģīti, jo vajag izprast perifērijas rīkus(piemēram, taimeri), kā tie strādā un kā tos apvienot ar pārējo programmu. Programmas koda izmaiņas rezultāta, piemēram aizkaves laika pagarināšana, izstrādātajam būs nepieciešams caurskatīt visu kodu un perifēriju no jauna, lai pārliecinātos kā izmaiņas ir veiktas pareizi un viss strādā kā vajag. Kodu uzrakstītu tāda veida būs sarežģīti pārnest uz sistēmu ar citu perifēriju un arhitektūru un programmētajam būs nepieciešams pārrakstīt iepriekš uzrakstītas laika kontroles funkcijas. Izmantojot RTOS lielāku daļu no iepriekšminēta nevajag darīt, un tas var ievērojami paātrināt izstrādes laiku.

Ganssle parāda RTOS nozīmīgumu maza mēroga iegultas sistēmās izmantojot printera sistēmas piemēru. Bez RTOS izmantošanas visu printera funkciju kontrole notiek viena koda gabalā – papīra padeve, lietotāja ievades nolasīšana, un drukas kontrole. RTOS dod iespēju noteikt katrai printera funkcijai savu atsevišķu procesu. Un šiem procesiem nav nepieciešams zināt par citu procesu darbību, izņemot tikai procesa darbības statusu. Tādējādi, izmantojot RTOS, var sadalīt programmu laika domēnos(lai tas vairākas daļas varētu strādāt vienlaicīgi) un\vai pēc to darbībās(lai katrs process veiktu specifisku uzdevumu). [3]

2.3 Pēdējo gadu tendence tirgu

Attēla 2.1 ir apskatāmi rezultāti no 2004 gada “Embedded Systems Design” veiktas

aptaujas iegulto sistēmu tirgu starp izstrādātajiem, kas izmantoja vai vēlas izmantot RTOS savos projektos. 2004 gadā, vairāk nekā 49 procenti izstrādātāju jau izmantoja reāla operētājsistēmas savos projektos. 2005. gadā šīs skaitlis pieauga līdz 80.9 procentiem un 2006.gada bija 71 procents. To izstrādātāju skaits, kas vēlas izmantot RTOS savos projektos bija 66.6 procentu 2004.gadā un 86% 2005.gadā, parādot pieaugošu interesi izmantot RTOS.



Attēls 2.1 (a) izmanto (b) vēlas izmantot

Tabula 2.1, kura parada 2006.gada "Embedded Systems Design" veiktas aptaujas rezultātus, ierosina pavisam citu tieksmi RTOS izvēle. Uzņēmumi sak pariet uz atklāta koda reāla laika operētājsistēmām – no 16 procentiem tekošos projektus uz 19 procentiem nākošos projektos, un virzas uz atklāta koda RTOS komerciālu izplatīšanu – no 12 procentiem uz 17 procentiem. Komerciālo un pašmājas operētājsistēmu izmantošana, kaut gan tagad vēl ir intensīva, sarūk no 51% līdz 47% un no 21% līdz 17% attiecīgi. 2007.gadā šīs skaitlis samazinās līdz 41% komerciāliem RTOS. 2007.gada aptauja parāda, ka galvenais komerciāla RTOS izvēles faktors ir tehniska atbalsta pieejamība un kvalitāte. Ja tāda nav, tad uzņēmumi varētu meklēt citu, rentablu risinājumu. [4]

Tabula 2.1: 2006 gada aptaujas rezultāti, tēma – projektos izmantoto operētājsistēmu tipi

Operētājsistēmas tips	Tekošais projekts(%)	Nākošais projekts(%)
Komerciāla OS	51	47
Pašmājas izstrādāta OS	21	17
Atvērta koda OS bez komerciāla atbalsta	16	19
Komerciāli izplatīta atvērta koda OS	12	17

2.4 Reālu operētājsistēmu vajās vietas

Izmantojot RTOS, tas patērē papildus resursu, tādus kā, atmiņu(gan ROM (Read Only Memory, latv. lasāmatmiņa), gan RAM (Random Access Memory, latv. brīvpiekļuves atmiņa)), skaitļošanas resursus, enerģiju. [5] Tāpēc, projektētajiem vajag noskaidrot vai sistēma var tikt gala ar

šiem virstēriņiem. Eksistē vairākas reālas operētājsistēmas, un daži no tiem ir pietiekami elastīgi, ka ļauj izstrādātajiem atslēgt visas nevajadzīgas funkcijas, tādējādi, atstājot pašu nepieciešamu un samazinot koda izmēru.[2, 6, 7, 8] Pietam, lielākai daļai RTOS ir nepieciešams taimeris(operētājsistēmas tiks (angl. Tick)[9]), lai darbotos plānotājs un citi svarīgi sistēmas servisi. RTOS servisu izpildes laikam jābūt zināmam(piemēram, procesu sinhronizācijai – jābūt zināms cik laika ir nepieciešams līdz nākamai procesu pārslēgšanai). Balstoties uz šiem faktoriem, un izmantotiem RTOS servisiem, sistēmas izstrādātajam ir rūpīgi jāprojektē sistēmas uzbūvi. Tādā veidā, izstrādātajiem jāzina reāla laika operētājsistēmas veiktspējas radītājus un jāsaprot kā tos ieguva.

3. Reāla laika operētājsistēmu veiktspējas etalonuzdevumi

Daudzi reāla laika operētājsistēmu veiktspējas etalonuzdevumi ir balstīti uz lietojumprogrammām vai uz biežāk izmantotam sistēmas servisa rutīnām(angl. Service routine).[10] Etalonuzdevumus nevar balstīt uz vienu vai dažam lietojumprogrammām, jo katrai programmai ir dažādas sistēmas prasības un šāds tests neatspoguļos RTOS vājas un stipras vietas. Veiktspējas etalonuzdevumi, kas ir balstīti uz biežāk izmantotiem sistēmas servisa rutīnām, iekļauj Rhealstone etalonuzdevumus, kuri mēra procesu pārslēgšanas laiku, prioritāra pārtraukuma izpildes laiku, pārtraukumu latentumu, semafora pārdalīšanas laiku, strupsaķeres(angl. deadlock) lūzuma laiku un datagrammu caurlaidspēju.[11] Taču arī Rhealstone testam ir minusi. Pirmkārt, tikai daži RTOS spēj iziet no strupsaķeres stāvokļa. Datagrammu caurlaides laiks ir balstīts uz ziņojumu sūtīšanu, kura procesā operētājsistēma kopē ziņojumus speciāla atmiņas apgabala. Taču, daži RTOS izmanto radītāju uz ziņojumu un, tādējādi, šīm nolūkam nav nepieciešams speciāls atmiņas apgabals. Tā ir piemērotākā pieeja izmantojot mazus mikrokontrollerus, jo ir nepieciešams mazāks pieejamas atmiņas daudzums. Pārtraukumu latentums šajā testa ir atkarīgs no procesora arhitektūras.[10]

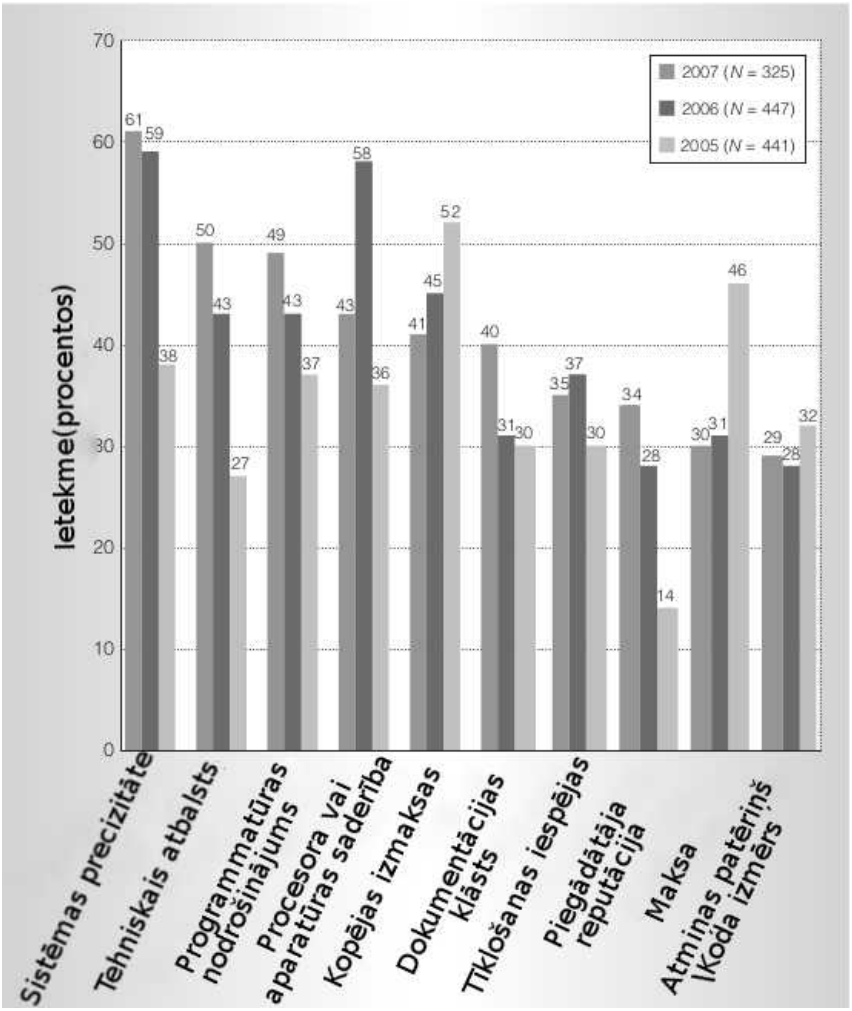
Garcia-Martinez un citi pētnieki piedāvā testu, kas ir balstīts uz biežāk izmantotam sistēmas izsaukumiem(angl. call): pārtraukumi, starpprocesu sinhronizācija un resursu koplietošana, starpprocesu datu pārraide(ziņojumu pārraide).[10] Starpprocesu datu pārraide ir līdzīga Rhealstone datagrammu caurlaides spējai. Pārtraukumu testā, pārtraukumu manipulators(angl. Interrupt handler) atmodina citu procesu izmantojot semaforu. Taču šajā gadījuma labākais risinājums būtu modināt procesu izmantojot sistēmas servisa izsaukumus(tādus kā sleep/wakeup izsaukumus), kas samazinās papildus aizkaves laiku.

3.1 Reāla laika operētājsistēmas īpašības un lietojumprogrammu salīdzinājums

Reāla laika operētājsistēmu salīdzinājums ir balstīts uz dokumentācijas, kas ir pieejama interneta vietnes. Tika apskatītas šādas sistēmas: uITRON[2,12], uTKernel[7], uC-OS/II[6], EmbOS[13], Free-RTOS[8], Salvo[14], TinyOS[15], SharcOS[16], eXtreme Minimal Kernel[17], Echidna[18], eCOS[19], Erika[20], Hartik[21], KeilOS[22] un PortOS[23].

3.1.1 Izmantotie kritēriji reāllaika operētājsistēmu salīdzināšanā

Izstrādātāji parasti izvēlas reāllaika operētājsistēmas balstoties uz valodas atbalsta, rīku saderības, sistēmas servisa lietojumprogrammu klāsta, izmantotas atmiņas daudzuma(ROM un RAM), veiktspējas, ierīču programmnodrošinājuma klāsta, atkļūdošanas rīku pieejamība, tehniska atbalsta pieejamība, koda izplatības līmeņa, licencēšanas veida un sistēmas piegādāja reputācijas līmeņa.[24,25] Kā tika noskaidrots pēc "Embedded System Design" veiktas aptaujas rezultātiem reāllaika operētājsistēmas laika precizitāte ir noteicošais faktors sistēmas izvēlē(sk. Attēls 3.1).[4]



Attēls 3.1 Ietekmējošie faktori operētājsistēmu izvēle pēc "Embedded Systems Design" veiktas aptaujas rezultātiem uz 2005, 2006, 2007. gadu. N – aptaujāto skaits.

Projekta mērķis – izstrādātajam vajag zināt reāllaika operētājsistēmas vēsturi un tā izstrādes motivāciju. RTOS var būt atvērta koda, pašdarināta(angl. Hobby based) vai biznesa projekts. Pašdarināta operētājsistēma var būt mazāk stabila nekā atklāta koda var komerciālas sistēmas.

Plānošanas shēma – ir svarīgi izvertēt RTOS plānošanas metodi: vai tā ir priekšlaicīga plānošana(angl. Preemptive scheduling), kooperatīva(angl. Cooperative) vai kāda cita plānošanas shēma.

Veiktspēja un sistēmas laika atbilstība – operētājsistēmas laika atbilstība apraksta vai sistēma var nodrošināt procesu izpildīšanu noteikta laika diapazonā. Procesora cikli un servisu patērēta laika daudzums jābūt mērojams un pietiekami zems. Visiem RTOS šī informācija nav pieejama, bet ja arī ir, tā var būt nepilnīga.

Atmiņas patēriņš – stabilākas un drošākas operētājsistēmu lietojumprogrammas parasti satur vairāk koda rindu, neka citas vienkāršākas operētājsistēmas. Tas tieši ietekmē uz atmiņas patēriņu un, tāpēc, pirms izvēles veikšanas projektētajiem jāiepazīstas ar šiem radītājiem.

Valodas atbalsts – programmēšanas valodas izvēles iespēja ir priekšrocība.

Lietojumprogrammatūras daudzējādība\Sistēmas izsaukumi – šīs kritērijs parāda konkrētas reāllaika sistēmas lietojumprogrammas nodrošinājuma kvalitāti salīdzinājuma ar citam RTOS. Katram RTOS arī tiek skaitīts kopējais sistēmas izsaukumu skaits.

Atkļūdošanas atbalsts – nosaka vai konkrēto RTOS var izmantot kopā ar integrētu izstrādes vidi(angl. Integrated development environment – IDE). Šīs iespējas pieejamība atvieglos izstrādes procesu, jo programmētajiem būs iespēja pārraudzīt sistēmas informāciju (piemēram, procesu stāvokli, sistēmas stāvokli, semaforus, un notikumu karodziņus).[3]

Licences tips – kāda veida tiek izplatīta reāllaika operētājsistēma: bezmaksas vai maksas, un kādiem mērķim, piemēram, apmācībai vai izmantošanai komerciālos projektos.

Dokumentācijas pieejamība – kāda tipa dokumentācija ir pieejama(detalizēts lietojumprogrammas apraksts, vienkārša pamācība, grāmata, vai sīks apraksts).

3.1.2 Salīdzinājuma rezultāti

Rezultāti ir apkopoti Tabula 3.1. Salīdzinājuma gaita tika noteiktas dažas svarīgas un līdzīgas iezīmes:

- Vairākums RTOS izmanto apsteidzošo, prioritātes balstītu plānošanas shēmu. Tikai divi no

apskatītiem – Salvo un TinyOS – izmanto kooperatīvo plānošanas shēmu.

- Visi RTOS atbalsta C programmēšanas valodu, kas ir biežāka izvēle iegulto sistēmu programmēšanā, jo īpaši maza mēroga sistēmu.
- Nedaudzi RTOS atbalsta atkļūdošanu IDE vidē: uC-OSII un EmbOS izmanto IAR izstrādes vidi, kurā var pieslēgt spraudņa(angl. Plug-in) moduļus; KeilOS tiek atbalstīts ar Keil kompilatoru; un uITRON un uTKernel tiek atbalstīts ar Renesas High Performance Embedded Workshop(HEW) kompilatoru.
- Reāllaika operētājsistēmai eCOS ir nepieciešama palaišanas programma(angl. Bootloader) - Redboot, kas aizņem vismaz 64 KB ROM atmiņas. Tā ielāde programmas RAM atmiņa saņemot komandas no termināla (parasti izmantojot seriālo portu). Tādējādi, eCOS ir nepieciešams vairāk ROM un RAM atmiņas nekā pārējam sistēmām.
- Dažam RTOS (KeilOS, PortOS un XMK) nav pieejama papildus informācija par sistēmas lietojumprogrammām (ir atzīmēti ar "N/A" "Sistēmas signāli/Lietojumprogrammu kvalitāte" kolonnā). SharcOS ir balstīts uz uC-OSII, tāpēc tiem ir vienādas lietojumprogrammas. To programmatūra nebija apskatīta salīdzinājumā.

Tabula 3.1: Mazo mikrokontrolleru reāllaika operētājsistēmu salīdzinājums

RTOS	**Izveides mērķis**	**Plānotāja shēma**	**Licences tips**	**Dokumentācija**	**Sistēmas signāli/Lietojumprogrammu kvalitāte**	**Programmēšanas valodu atbalsts**	**IDE atkļūdošanas atbalsts**
uITRON	Komerciāls	apsteidzoša, prioritātes balstīta	Maksas	Lietotāja instrukcija un specifikācijas	93	C	Renesas IDE
uTKernel	Komerciāls, izglītības, pētniecība	apsteidzoša, prioritātes balstīta	Bezmaksas izglītībā un komercijā	Lietotāja instrukcija un specifikācijas	81	C	Renesas IDE
uC-OSII	Komerciāls, izglītības, pētniecība	apsteidzoša, prioritātes balstīta	Bezmaksas izglītībā	Grāmata	42	C	IAR
EmbOS	Komerciāls	apsteidzoša, prioritātes balstīta	Maksas	Tiešsaistes dokumentācija	56	C	IAR

Free-RTOS	Hobijs	apsteidzoša, prioritātes balstīta	Bezmaksas izglītībā un komercijā	Tiešsaistes dokumentācija	27	C	Nav
Salvo	Komerciāls	Kooperatīva	Maksas	Tiešsaistes dokumentācija	31	C	Nav
TinyOS	izglītības, pētniecība	Kooperatīva	Bezmaksas izglītībā un komercijā	Pamācība	N/A	C, NesC	Nav
SharcOS	Komerciāls	apsteidzoša, prioritātes balstīta	Maksas	Lietotāja instrukcija	N/A	C	Nav
eXtreme Minimal Kernel (XMK)	izglītības, pētniecība	apsteidzoša, prioritātes balstīta	Bezmaksas izglītībā un komercijā	Tiešsaistes dokumentācija (nepilna)	N/A	C	Nav
Echidna	izglītības, pētniecība	apsteidzoša, prioritātes balstīta	Bezmaksas izglītībā un komercijā	Tiešsaistes dokumentācija (nepilna)	18	C	Nav
eCOS	Komerciāls, izglītības, pētniecība	apsteidzoša, prioritātes balstīta	Bezmaksas izglītībā un komercijā	Grāmata un tiešsaistes dokumentācija	N/A	C	Nav
Erika	izglītības, pētniecība	apsteidzoša, prioritātes balstīta	Bezmaksas izglītībā un komercijā	Tiešsaistes dokumentācija	19	C	Nav
Hartik	izglītības, pētniecība	apsteidzoša, prioritātes balstīta	Bezmaksas izglītībā un komercijā	Tiešsaistes dokumentācija	33	C	Nav
KeilOS	Komerciāls	apsteidzoša, prioritātes balstīta	Maksas	Tiešsaistes dokumentācija	N/A	C	Keil IDE
PortOS	pētniecība	apsteidzoša, prioritātes balstīta	Maksas	Tiešsaistes dokumentācija	N/A	C	Nav

Nākamajā attēla tika salīdzināts pieejamo lietojumprogrammatūras funkciju skaits noteikta uzdevuma izpildei katrai reāllaika operētājsistēmai(sk. Attēls 3.2). Funkcijas tika sagrupētas pēc šādiem uzdevumiem(kopā ar piemēriem):

- Sistēmas pārvaldība: operētājsistēmas inicializācija\izslēgšana\ieslēgšana, procesora aizture(angl. Lock);
- Pārtraukumu pārvaldība: ieeja\izeja no funkcijām, ieeja\izeja no kritiskiem apgabaliem;
- Procesu pārvaldība: izveidot\izdzēst\pārtraukt procesu;
- Procesu sinhronizācija: atmodināt\pārslēgt miega režīma\turpināt procesu;
- Komunikācija un sinhronizācija: semafori, datu pieprasījums, notikumu karogi, pastkastīte, resursu bloķēšana un ziņojumu buferis;
- Atmiņas pārvaldība: noteikta un mainīga izmēra atmiņas bloka izdalīšana;
- Laika pārvaldība: iegūt sistēmas laiku, sistēmas taimeris;
- Programmas trasējums: piestiprināt lietotāja funkciju kopā ar operētājsistēmas rutīnu, piemēram, plānotāju. No salīdzinātiem RTOS – uITRON, uTKernel, uC-OS/II un EmbOS ir diezgan plašs klāsts šāda tipa rutīnu.

Lielāka daļa apskatīto komerciālo reāllaika operētājsistēmu atbalsta iepriekšminētas kategorijas izslēdzot trasēšanas funkcijas. Pie tam tie ir labākie no visam apskatītam sistēmām, jo tiek attīstīti un uzlaboti un atrodas tirgū jau krietnu laiku.

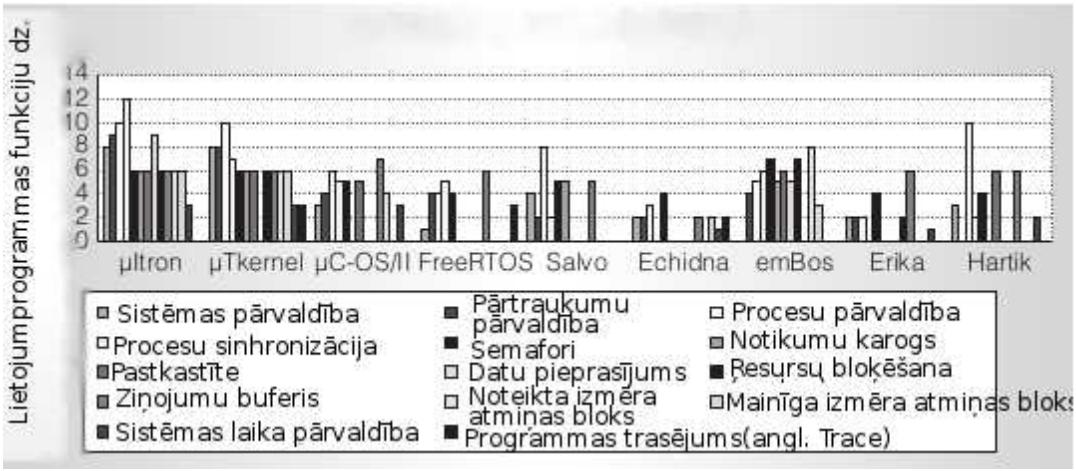
Savukārt, atvērta koda RTOS, tādi kā, Free-RTOS, Echidna, Erika un Hartik, ir minimāli implementēti un vislabāk der maza mēroga sistēmu izstrādei. Pretstatam, uC-OS/II ir pirmā pēc pieejamo funkciju klāsta. Bezmaksas RTOS - uTKernel, atbalsta gandrīz tikpat daudz funkciju, jo to izstrādi kontrolēja uITRON arhitektūras projektētājs Ken Sakamura. EmbOS ir implementētas gandrīz visas funkcijas izņemot trasēšanu, sistēmas laika pārvaldi, sistēmas pārvaldi, un ziņojumu bufera.

Balstoties uz šiem argumentiem, tālākos testos un salīdzinājumos piedalās tikai pirmā četrinieka reāllaika operētājsistēmas: uITRON, uTKernel, uC-OS/II un EmbOS.

Papildus attēla 2.1 minētam funkcijām, daži no apskatītiem reāllaika operētājsistēmām atbalsta šādas iespējas:

- Noildze(angl. Timeout). Piemēram, process var gaidīt semaforu maksimāli n milisekundes. uITRON un uTKernel ļauj norādīt noildzi absolūtas vērtības. Citas RTOS, kā uC-OS/II, Free-RTOS, Salvo un EmbOS, ļauj norādīt noildzi taimera tiku izteiksmē.

- Atkļūdošanas lietojumprogrammas ļauj dabūt informācija, kas tiek pārvaldīta ar sistēmas kodola palīdzību. Šobrīd, tikai uTKernel atbalsta šāda veida programmatūru(piemēram, dabūt procesa reģistru un uzstādīt procesa reģistru).
- Ciklisks apdarinātājs(angl. Cyclic handler) nosaka, ka funkcija bus izpildīta noteikta laika perioda, un trauksmes apdarinātājs(angl. Alarm handler) ļauj izpildīt funkcija pēc noteikta laika intervāla. Šīs funkcijas pašlaik ir pieejamās uITRON un uTKernel operētājsistēmas.
- Satikšanas mehānisms(angl. Rendezvous mechanism) atļauj starpprocesu komunikāciju un sinhronizāciju(līdzīgi Ada valodai, bet reāllaikā). Šo mehānismu atbalsta uITRON un uTKernel operētājsistēmas.



Attēls 3.2 Funkciju daudzums noteikta uzdevuma izpildei dažādiem RTOS

Dažas uITRON un uTKernel lietojumprogrammas sniedz labāku kontroli un pielāgojamību. Pēc tabulas 3.2 procesi, gaidot pēc semafora, uC-OS/II un EmbOS operētājsistēmas nokļūst rindā pēc FIFO(angl. First in – First out, latv. - Pirmais iekšā – Pirmais arā) metodes un izstrādātāji nevar mainīt šo kartību. Taču, uITRON un uTKernel operētājsistēmas, var norādīt vai izmantot FIFO vai prioritātēs secību. Šī pielāgojamība attiecas arī uz pārējiem servisiem(ieskaitot pastkastīti, ziņojumu rindām, atmiņas pārvaldību un notikumu karogiem). Taču šādas iespējas atstāj savu negatīvo ietekmi uz atmiņa patērēšanu un veiktspēju.

3.2 Veiktspējas un atmiņas lietošanas testi

Testi tika veikti ar četrām iepriekšminētam operētājsistēmām: uITRON, uTKernel, uC-OS/II un EmbOS. Lai nomērītu izpildes laiku, tika izmantoti oscilogrāfs un loģiskie analizatori kombinējot tos ar izejas\ieejas portu pārslēgšanu, lai sasniegtu lielāku rezultātu precizitāti(mikrosekunžu robežas).

Bija izmantota Renesas M16C/62P izstrādes plate(sk. Attēls 3.1) ar šādam specifikācijām:

- 24 MHz takts frekvence
- 512 KB ROM
- 31 KB RAM bez kešatmiņas vai atmiņas pārvaldības bloka(angl. Memory Management Unit – MMU)
- Renesas HEW IDE, versijas 4.03.00.001
- NC30 kompilators, versijas 5.43.00

Attēls 3.3: Renesas M16C/62P izstrādes plate

Šai platei ir 16 bitu CISC(complex instruction set computer, latv. sarežģītas instrukcijkopas dators) arhitektūras procesors ar, kopumā, 91 pieejamam instrukcijām. Lielāka daļa instrukciju izpildās divu, trīs takts laikā. Mikrokontrolierī ir instrukciju četrpakāpju rindas buferis, kas ir līdzīgs vienkāršākam konveijeram lielākos 32 bitu procesoros.

Lai nodrošinātu, ka visi RTOS strādā šajā platformā ar vienādu laika precizitāti, katrai operētājsistēmai atsevišķi bija uzstādīts taimeris ar 10 sekunžu frekvenci. Papildus, katra sistēma izmantoja vienādu steka atmiņas daudzumu. Un bija izslēgtas īpatnējas operētājsistēmu iespējas(piemēram, ziņojumu vai semaforu rinda).

Sistēmas signālu un kritiska apgabala signālu implementācijas atšķirības ir parādītas

Tabula 3.2. Kad uC-OS/II un EmbOS veic sistēmas izsaukumu, tie izsauc funkciju tieši no lietotāja procesa. Šīs metodes priekšrocība ir tāda, ka funkcija tiek izsaukta bez papildus aizskāves, taču tā izmantos šī procesa steku. Pretstatam, uITRON un uTKernel, izraisa nemaskējamo koda pārtraukumu(INT instrukcija M16C/62P arhitektūrā). Šajā veida notiek parēja uz kodola telpu(angl .kernel space), jeb atsevišķa steka, kurā tiek izpildīta izsaukta funkcija. Šīs metodes trūkums ir papildus aizskāves rašanas.

Tabula 3.2: Sistēmas signālu un kritiska apgabala signālu implementācija

Signāls	**uC-OS/II**	**uTKernel**	**EmbOS**	**uITRON**
Sistēmas izsaukums	Tiešais	Izmantojot koda pārtraukumu	Tiešais	Izmantojot koda pārtraukumu
Kritiskais apgabals	Izslēgt visus pārtraukumus	Izraisa pārtraukumu ar maskas līmenis 4	Nevajag izslēgt pārtraukumus(balstīts uz iekšēja mainīga)	Izraisa pārtraukumu ar maskas līmenis 4

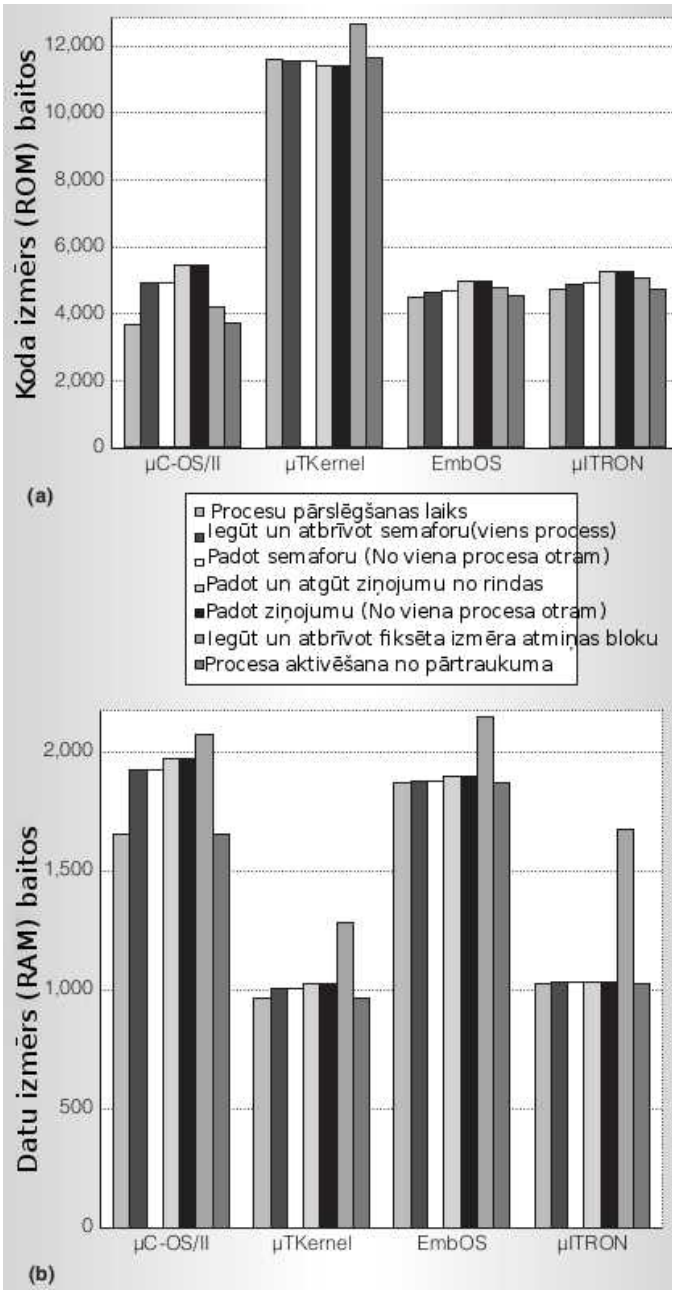
Ieejot kritiskajā apgabala uC-OS/II izslēdz visus pārtraukumus, kas ļauj izpildīt visas darbības šajā apgabala droši, bez ārējas iejaukšanas(piemēram, ārēja pārtraukuma izsaukums). Taču pārtraukumi ar lielāku prioritāti, kas var būt nozīmīgāki par izsaukto, nebūs ņemti vēra. uITRON un uTKernel operētājsistēmas kritiska apgabala apstrāde tiek īstenota izraisot pārtraukumu ar maskas līmeni 4, kas dod iespēju pieņemt pārtraukumu ar lielāku prioritāti. EmbOS operētājsistēmā kritiska apgabala apstrāde ir programmētāja ziņā, kas dod iespēju pieņemt un apstrādāt jebkurus pārtraukumus, taču tas palielina realizācijas sarežģītību.

3.2.1 Etalonuzdevumu kritēriji

Kritēriji tika izvēlēti tāda veida, lai tos būtu viegli pārnest uz dažādam izstrādes platformām. Katram kritērijam tika savākti izpildes laika un atmiņas lietojuma informācija.

Procesa pārslēgšanas laiks – laiks, kurā tiek pārnests izpildāmais konteksts no viena procesa uz citu. Attēla 3.5 ir apskatāmi kritēriju aprēķina metodes.

Tas iekļauj divus procesus, procesu 1 ar lielāku prioritāti par otru procesu. Katrs RTOS no sākuma izpilda procesu 1, kurš pēc tam ieiet miega vai bezdarbības stāvokli. Pēc tam izpildes process pariet uz procesu 2, kas atmodina vai aktivizē procesu 1. Uzreiz pēc atmošanās izpilde pariet atpakaļ uz procesu 1, jo tam ir lielāka prioritāte. Dažādam reāllaika operētājsistēmām tiek izmantoti dažādi termini, lai aprakstītu miega\ne aktīvo(angl. sleep\inactive) un gatavs\aktīvais(angl.ready\active) stāvokļus, piemēram, uC-OS/II un EmbOS izmanto suspend/resume, bet uITRON un uTKernel



Attēls 3.4 Testa rezultāti salīdzinot (a)koda un (b)datu izmēru

izmanto sleep/wakeup. Tabulā 3.3 ir apskatāmi izmantotie sistēmas signāli katrai operētājsistēmai.

Tabula 3.3: Izmantotas funkcijas procesu pārslēgšanas laika etalonuzdevumiem

API	UC-OS/II	uTKernel	EmbOS	uITRON
Nodot ziņojumu	OSTaskSuspend()	tk_slp_tsk()	OS_Suspend()	slp_tsk()
Iegūt ziņojumu	OSTaskResume()	tk_wup_tsk()	OS_Resume()	wup_tsk()

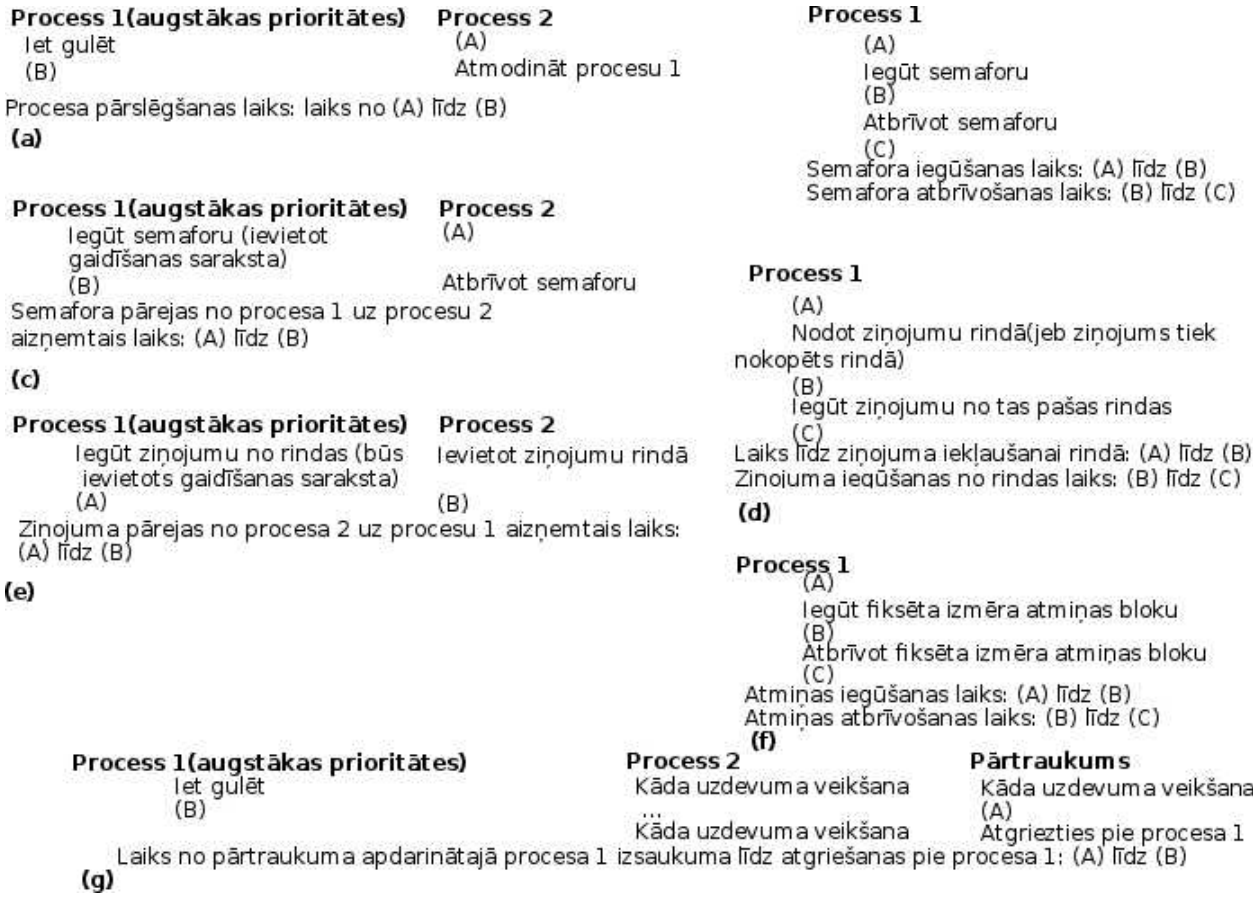
Iegūt/atbrīvot semaforu uzdevuma izpildes laiks. Bieži vien mainīgo sinhronizācijai izmanto semaforus. Šajā etalonuzdevumā tiek mērīts laiks, ko aizņem semafora iegūšanai un atbrīvošanai, kā arī tā padošanai citam procesam. Attēla 3.5(b) ir attēlota laika aprēķināšanas metode.

Pastāv viens process(process 1) un viens binārai semafors(tā sākuma vērtība ir 1). Process 1 iegūst un pēc tam atbrīvo semaforu. Tabula 3.4 ir apskatāmi izmantotie sistēmas signāli katrai operētājsistēmai.

Tabula 3.4: Izmantotas funkcijas semafora iegūšanas\atbrīvošanas etalonuzdevumos

API	UC-OS/II	uTKernel	EmbOS	uITRON
Iegūt semaforu	OSSemPend()	tk_wai_sem()	OS_WaitCSema()	wai_sem()
Atbrīvot semaforu	OSSemPost()	tk_sig_sem()	OS_SignalCSema()	signal_sem()

Semafora padošanas laiks citam procesam. Lai noteiktu semafora padošanas efektivitāti starp procesiem tika izmantota metode, kas attēlota attēlā 3.5(c).



Attēls 3.5 Etalonuzdevumu kritēriju mērījumi: procesa pārslēgšana(a), iegūt/atbrīvot semaforu(b), semafora padošana(c), padot/iegūt ziņojumu(d), padot ziņojumu(citam procesam) (e), iegūt/atbrīvot fiksēta izmēra atmiņas bloku(f), procesa aktivācija izmantojot pārtraukumu(g)

Šajā testa piedalās divi procesi, process 1 un 2, kur pirmajām procesam ir lielāka prioritāte, un binārai semafors(inicializēts uz nulli). Sistēma izpilda procesu 1, kurš mēģina iegūt semaforu. Tā kā semafora vērtība ir 0, process ieej miega režīmā, gaidot, kad semafors tiks atbrīvots.

Izpilde pariet pie otra procesa, kurš atbrīvo semaforu. Uzreiz pēc tā atbrīvošanas, tas atmodina pirmo procesu un izpilde pariet pie procesa 1.

Ziņojuma iegūšanas/padošanas izpildes laiks. Papildus semaforu izmantošanai, ziņojumu padošana kļūst populāra sinhronizācijas uzdevumos. Šīs tests izmanto ziņojuma padošanas mehānismu balstītu uz ziņojuma radītāja kopēšanu, nevis visu ziņojumu, jo ne visi RTOS atbalsta šādu metodi. Attēlā 3.5(d) ir redzams kā tiek izpildīts šīs mērījums.

Tas iekļauj tikai vienu procesu, kas padod ziņojuma radītāju(parasti uz ārējo ziņojumu rindu), un pēc tam iegūst šo pašu ziņojuma radītāju. Tabula 3.5 var apskatīt izmantotas funkcijas katrai RTOS.

Tabula 3.5: Izmantotas funkcijas ziņojumu padošanas etalonuzdevumos

API	UC-OS/II	uTKernel	EmbOS	uITRON
Nodot ziņojumu	OSQPost()	tk_snd_mbx()	OS_Q_Put()	snd_mbx()
Iegūt ziņojumu	OSQPend()	tk_rcv_mbx()	OS_Q_GetPtr()	rcv_mbx()

Starpprocesu ziņojuma padošanas laiks. Attēlā 3.5(c) ir parādīts kā tiek mērīts laiks šajā testā. Tests iekļauj divus procesus(1 un 2). Process ar lielāku prioritāti(1) tiek izsaukts pirmais un mēģina iegūt ziņojuma rādītāju no rindas. Tā kā neviena ziņojuma vēl nav, tas ieiet gaidīšanas režīma. Izpilde pariet uz procesu 2, kurš ievieto jaunu ziņojumu rindā, kā rezultāta pamostas pirmais process un izpilde pāriet pie tā. Šī testa atšķirība no viena procesa ziņojuma iegūšanas un padošanas ir tāda, kā šī metode iekļauj sistēmas papildus aizkāvi, kas rodas ziņojumu rindas apstrāde un procesa pamošanās signāla sūtījumā rezultāta.

Fiksēta izmēra atmiņas bloka iegūšana un atbrīvošana. Reāllaika operētājsistēmas vajag izdalīt tikai fiksēta izmēra dinamisku atmiņu. Attēlā 3.5(f) ir redzams testa mērījumu gaitā.

Šīs tests iekļauj vienu procesu, kurš iegūst atmiņas bloku(128 baitu) un pēc tam atbrīvo to. Izmantotas funkcijas ir redzamas Tabula 3.6.

Tabula 3.6: Izmantotas funkcijas atmiņas etalonuzdevumiem

API	UC-OS/II	uTKernel	EmbOS	uITRON
Iegūt fiksēta izmēra atmiņas bloku	OSMemGet()	tk_get_mpf()	OS_MEMF_ Alloc()	get_mpf()
Atbrīvot ...	OSMemPut()	tk_rel_mpf()	OS_MEMF_ Release()	rel_mpf()

Procesa aktivācija no pārtraukuma. RTOS jāspēj strādāt ar ārējiem pārtraukumiem, kas

var tikt izsaukti jebkurā laikā. Pie tam pārtraukuma kodam jābūt pēc iespējas īsākam, jo citādi apdarinātājs var aktivēt citu procesu tā izpildes gaitā. Laiks, kurā pārtraukuma apdarinātājs izsauc procesu līdz tā izpildes sākumam ir kritisks sistēmas projektēšanā.

Attēls 3.5(g) parāda mērījuma uzstādni, kurš ietver divus procesus un ārējo pārtraukumu ar raksturīgu apdarinātāju. Process ar lielāku prioritāti(process 1) tiek izpildīts pirmais. Tas ieiet miega režīmā un izpilde pariet pie otra procesa, kas darbojas cikliski. Kad notiek ārējais pārtraukums, izpildās pārtraukuma apdarinātais un atmodina pirmo procesu. Pēc tam izpilde pariet pie procesa 1. Tabula 3.7 ir apskatāmi izmantotie sistēmas signāli katrai operētājsistēmai.

Tabula 3.7: Izmantotas funkcijas procesu aktivācijas etalonuzdevumiem

API	UC-OS/II	uTKernel	EmbOS	uITRON
Iet gulēt	OSTaskSuspend()	tk_slp_tsk()	OS_Suspend()	slp_tsk()
Atgriezties no pārtraukuma	OSTaskResume()	tk_wup_tsk()	OS_Resume()	iwup_tsk()

3.2.2 Testu rezultāti

Izmantotais kods tika nokompilēts katram etalonuzdevuma kritērijam, lai noskaidrotu ROM un RAM izmantotas atmiņas daudzumu. Attēls 3.4(a) parāda katrai apskatītai reāllaika operētājsistēmai izmantotas ROM atmiņas vidējo daudzumu, palaižot septiņus testus.

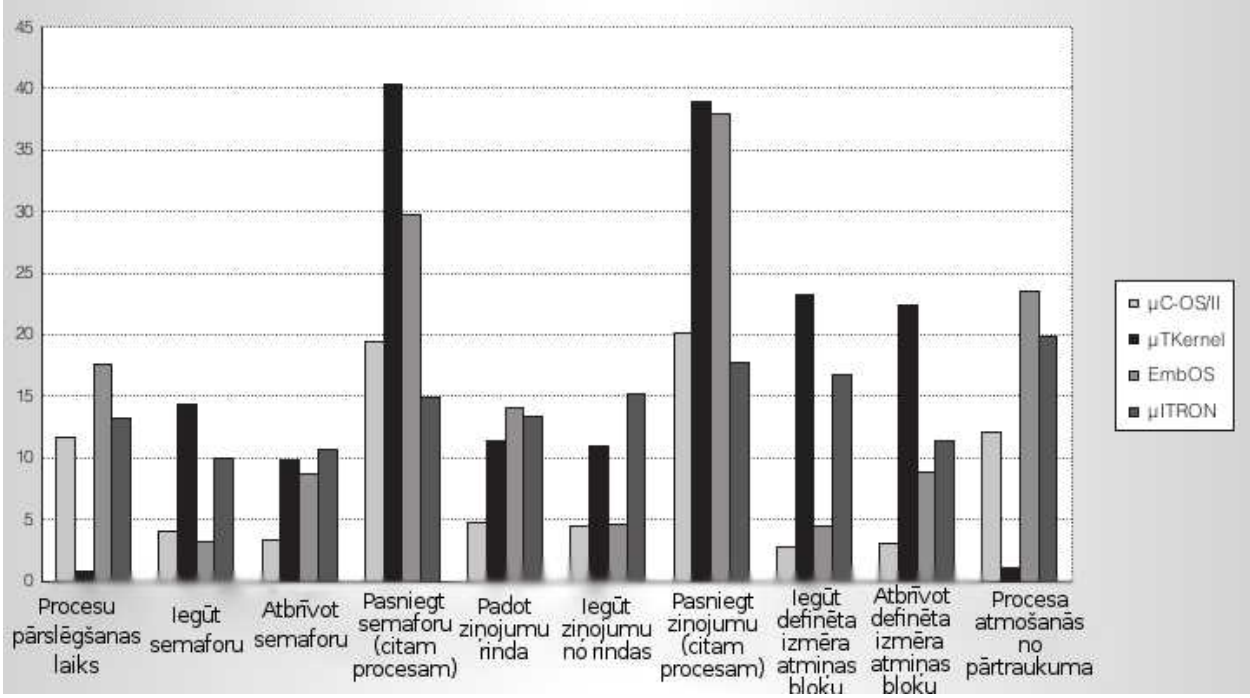
Ka redzams no grafika, uTKernel operētājsistēma patērē lielāku ROM atmiņas daudzumu. Tas ir saistīts ar tā lietojumprogrammatūras pielāgojamību un plašu atbalstu. Savukārt, komerciāli produkti, kā uITRON un EmbOS, piedāvā nosacīti kompaktu koda izmēru. Neskatoties uz to, visi četri RTOS der mikrokontrolleriem ar ierobežotu ROM atmiņas daudzumu.

Attēls 3.4(b) rāda informāciju par RAM atmiņas patēriņu. uITRON un uTKernel patērē mazāku RAM atmiņas daudzumu salīdzinājuma ar uC-OS/II un EmbOS. Saskaņa ar katra etalonuzdevuma prasībām katrai reāllaika operētājsistēmai tika uzstādīts vienāds procesu skaits, steka izmērs, un sistēmas objektu skaits(piemēram, semaforu un notikumu karodziņu). Izmantotas RAM atmiņas atšķirība katram RTOS ir 7-10 baitu robežas, kas var būt iekšējas realizācijas atšķirības rezultāts.

Kopumā, visi RTOS rāda nosacīti labus rezultātus un der maziem mikrokontrolleriem, taču uITRON salīdzinājuma ar pārējiem patērē gan ROM, gan RAM minimālu daudzumu.

3.3 Izpildes laiks

Attēls 3.6 parāda dažādu uzdevumu izpildes laika mērījumus dažādiem reāla laika operētājsistēmām(microC-OS/II - Micro-Controller Operating Systems Version 2 no Micrium(sk. http://micrium.com/), atvērta koda microTKernel no T-Engine(sk. http://en.wikipedia.org/wiki/T-Engine), EmbOS no Segger(sk. http://www.segger.com/) un atvērta koda microITRON - Industrial "The Real-time Operating system Nucleus"). Tā kā taimera pārtraukuma laiks ir vienīgais, ko ir grūti prognozēt, tika veikti vairāki etalonuzdevumu mērījumi(vismaz divas reizes katram uzdevumam), lai garantētu ticamus rezultātus.



Attēls 3.6 Izpildes laika etalonuzdevumu rezultāti četriem RTOS

Procesa atmošanās no pārtraukumā etalonuzdevumam ārēja pārtraukumu izsaukums bija papildus variants. Tādējādi, ārējais pārtraukums var tikt un var arī netikt izsaukts, nokļūstot operētājsistēmas kritiskajā segmentā(kuras laikā operētājsistēma atslēdz visus pārtraukumus). Jā pārtraukumu izsaukums notika šajā brīdi, tad sistēmas atbildes laiks būs mazliet garāks. Tādā veidā, šīs mērījums var nesaturēt sliktāko gadījumu. Kā ir redzams no attēla 3.6, uTKernel visātrāk pārslēdzas starp procesiem, pēc tam nāk uITRON, uC-OS/II un EmbOS. Tas arī parāda, ka uC-OS/II visātrāk tiek galā ar semaforu iegūšanu un atbrīvošanu. uITRON visātrāk padod ziņojumus starp procesiem, turpretim uC-OS/II un uTKernel labāk iegūst un padod ziņojumus no rindas nekā uITRON un EmbOS. Strādājot ar atmiņas bloku izdalīšanu vislabāko laiku parādīja uC-OS/II. Atklāta koda operētājsistēmu

kategorija uC-OS/II ir noderīga kā kompakta RTOS pēc aizņemtas ROM atmiņas. Taču pēc API atbalsta uTKernel ir labāka izvēle(upurējot koda kompaktumu). No cita skatu punkta, ja izstrādātājs labāk izvēlēsies komerciālu operētājsistēmu, tad jāskatās uITRON vai EmbOS virziena, pietam uITRON izmanto mazliet mazāk RAM atmiņas.

Šo testu kritēriji ir vienkārši un tos ir viegli pārnērst uz citam platformām, un, kas nebūt nav mazsvarīgi, tie parāda tipisku RTOS lietojumu. Tie parāda kā uzdevumu un operētājsistēmu dažādībā nav skaidra uzvarētāja un katrai reāllaika operētājsistēmai ir savas vājas un stipras puses.

Atsauces

1 D. Kalinsky, ‘‘Basic Concepts of Real-Time Operating Systems,’’ Linux Devices, Nov 2003; http://www.jmargolin.com/uavs/jm_ rpv2_npl_16.pdf.

2 K. Sakamura and H. Takada, ‘‘mITRON for Small-Scale Embedded Systems,’’ IEEE Micro vol. 15, no. 6, Nov./Dec. 1995, pp. 46-54.

3 J. Ganssle, ‘‘The Challenges of Real-Time Programming,’’ Embedded System Programming, vol. 11, July 1997, pp. 20-26.

4 R. Nass, ‘‘Annual Study Uncovers the Embedded Market,’’ Embedded Systems Design, 2 Sept. 2007; http://www.embedded.com/ design/opensource/201803499;jsessionid= NGMOMOIGE5ZNNQE1GHOSKHWATMY 32JVN?printable=true.

5 K. Baynes et al., ‘‘The Performance and Energy Consumption of Embedded Real-time Operating Systems,’’ IEEE Trans. Computers, vol. 52, no. 11, 2003, pp. 1454-1469.

6 J.J. Labrosse, MicroC/OS-II: The Real-Time Kernel, R&D Books, 1999.

7 T-Engine Forum, ‘‘mTKernel specification, 1.00.00,’’ Mar. 2007; http://www.t-engine.org.

8 R. Barry, ‘‘A Portable, Open Source Mini Real-Time Kernel,’’ Oct. 2007; http://www. freertos.org.

9 K. Curtis, ‘‘Doing Embedded Multitasking with Small Microcontrollers, Part 2,’’ Embedded System Design, Dec. 2006; http://www. embedded.com/columns/technicalinsights/ 196701565?_requestid=242226.

10 A. Garcia-Martinez, J. F. Conde, and A. Vina, ‘‘A Comprehensive Approach in Performance Evaluation for Modern Real-Time Operating Systems,’’ Proc. 22nd EuroMicro Conf., IEEE CS Press, 1996, p. 61.

11 R.P. Kar and K. Porter, ‘‘Rhealstone: A Real- Time Benchmarking Proposal,’’ Dr. Dobb’s J. of Software Tools, vol. 14, no. 2, Feb. 1989, pp. 14-22.

12 K. Sakamura and H. Takada, mITRON 4.0 Spec- ifications, TRON Assoc., 2002; http://www. ertl.jp/ITRON/SPEC/FILE/mitron-400e.pdf.

13 EmbOS Real-Time Operating System, User & Reference Guide, Segger Microcontroller, 2008; http://www.segger.com/cms/admin/ uploads/productDocs/embOS_Generic.pdf.

14 http://www.pumpkininc. com

15 http://www.tinyos.net

16 http://www.sharcotech.com

17 XMK, http://www. shift-right.com/xmk/index.html

18 http://www.ece.umd.edu/serts/research/ echidna/index.shtml

19 A.J. Massa, Embedded Software Develop- ment with eCos, Prentice Hall, 2002

20 P. Gai et al., E.R.I.K.A.: Embedded Real-tIme Kernel Architecture; ERIKA Educational User Manual, Realtime System (RETIS) Lab, Scuola Superiore Sant’Anna, Italy, 2004; http://erika. sssup.it/download.shtml#Doc.

21 G.C. Buttazzo, ‘‘Hartik: A Hard Real-Time Kernel for Programming Robot Tasks with Explicit Time Constraints and Guaran- teed Execution,’’ Proc. IEEE Int’l Conf. Robotics and Automation, IEEE Press, 1993, pp. 404-409.

22 http://www.keil.com/ rtos

23 R. Chrabieh, ‘‘Operating System with Prior- ity Functions and Priority Objects,’’ Tech- Online, Feb. 2005; http://www.techonline. com/learning/techpaper/193101942.

24 G. Hawley, ‘‘Selecting a Real-Time Operat ing System,’’ Embedded System Design, vol. 12, no. 3, 1999, http://www.embedded. com/1999/9903.

25 M. Timmerman and L. Perneel, ‘‘Understand- ing RTOS Technology and Markets,’’ Dedicated Systems RTOS Evaluation proj- ect report, 2005; http://www.dedicated- systems.com/vpr/layout/display/pr.asp?PRID= 8972.