ELEKTROTEHNIŠKI VESTNIK 81(4): 209–213, 2014
ORIGINAL SCIENTIFIC PAPER

RTOS za ARM7

Janez Puhan

Univerza v Ljubljani, Fakulteta za elektrotehniko, Tržaška 25, 1000 Ljubljana, Slovenija
E-pošta: janez.puhan@fe.uni-lj.si

Povzetek. Predstavljen je predkupni operacijski sistem, ki teče v realnem času. Prirejen je za mikrokrmilnike s procesnim jedrom ARM7. Izvorna koda za družino mikrokrmilnikov LPC123x je dostopna na svetovnem spletu [1]. Posamezne sklope operacijskega sistema, kot so npr. razvrščanje procesov, dinamično dodeljevanje pomnilnika in binarni semaforji, smo implementirali na preprost in modularen način. Odločili smo se za prioriteten algoritem razvrščanja procesov, kar procesu z najvišjo prioriteto zagotavlja trdo realno časovno razvrščanje. Poudarjeni so primeri, ki povzročijo smrtni objem. Za dinamično dodeljevanje pomnilnika smo uporabili algoritem prvega primernega prostora, dodeljen pomnilnik pa se nahaja v urejenem povezanem seznamu. Razvrščevalnik procesov je prožen s prekinitvijo časovnika *timer0*, medtem ko so sistemski klici izvedeni s programsko prekinitvijo. Nesistemske prekinitve programska oprema prestreže. Obdelane so lahko v okviru operacijskega sistema ali mimo njega. Izmerjene lastnosti sistema podajamo za mikrokrmilnik LPC2138.

Ključne besede: operacijski sistemi v realnem času, razvrščanje procesov, vgrajeni sistemi

RTOS for ARM7

The paper presents a new preemptive real-time operating system for the ARM7 core. The source code for the LPC213x processors with the ARM7TDMI-S core is available [1]. The required mechanisms, such as process handling, dynamic-memory management and binary semaphores, are implemented both in a simple and modular manner. The developed scheduling algorithm is priority-based to ensure hard real-time scheduling of the highest-priority process. Deadlock situations are pointed out. The first-fit dynamic-memory management algorithm is used. The memory allocations are organized in an ordered linked list. To invoke the scheduler *timer0*, an interrupt request is used, while the system calls are performed by the software interrupt exceptions. The non-system exceptions are caught and can be handled by or without the system. The system properties are measured on the LPC2138 processor.

1 UVOD

Programska oprema za mikrokrmilniške vgrajene sisteme je napisana na različne načine. Najpogoštejše programske arhitekture so: izvajanje nalog v neskončni zanki, servisiranje s prekinitvami proženih dogodkov, razporejanje procesorskega časa med opravili s pomočjo operacijskega sistema ali kombinacija med njimi. Potreba po uporabi operacijskega sistema se pokaže ob večjem številu nalog, ki jih opravlja mikrokrmilniški sistem. Programska oprema postane obsežna, zapletena in posledično neobvladljiva. Operacijski sistem omogoči navidezno sočasnost izvajanja nalog (ang. *time sharing*) za ceno lastne režije. Operacijski sistemi na vgrajenih sistemih morajo zaradi svoje interakcije z okoljem navadno teči v realnem času. Za vgrajene sisteme je na voljo več operacijskih sistemov v realnem času, kot npr. FreeRTOS [2], ThreadX [3] ipd., ki navadno želijo zadostiti vsem željam uporabnikov. Tako tečejo na množici različnih mikrokrmilnikov, ponujajo različne politike razvrščanja, sheme dinamičnega dodeljevanja pomnilnika, sinhronizacije med procesi ipd. Splošnost je sicer dobrodošla, a lahko postane nepregledna.

V članku predstavljamo nov prioritetni predkupni operacijski sistem v realnem času. Napisali smo minimalen operacijski sistem, ki omogoča prioritetno predkupnost poljubnega števila dinamično ustvarjenih končnih ali neskončnih procesov. Sistem smo implementirali za 32-bitne mikrokrmilnike družine LPC213x [4] s procesnim jedrom ARM7TDMI-S [5].

2 DATOTEKE Z IZVORNO KODO OPERACIJSKEGA SISTEMA

Izvorna koda jedra operacijskega sistema [1] je v datotekah s predpono *os*. Jedro obsega razvrščevalnik procesov s funkcijami sistemskih klicev. Del jedra je tudi datoteka *exceptions.s*, ki prestreže izjeme* (ang. *exception*) [6]. Procesno jedro ARM7TDMI-S pozna več vrst izjem s pripadajočimi vektorji in načini delovanja (ang. *mode*) [5]. Vsak način delovanja ima svoj kazalec sklada†. Za inicializacijo kazalcev sklada, kakor tudi za postavitev incializiranih in neinicializiranih globalnih in statičnih spremenljivk, poskrbi zagonska koda v datoteki *ctr0.s*, ki jo povezovalnik (ang. *linker*) postavi na naslov

Prejet 17. junij, 2014
Odobren 30. september, 2014

*Izjema je prekinitev normalnega toka progama, ki je ni mogoče maskirati. Vse prekinitve zaradi zahtev perifernih enot (npr. *timer*, *uart* itd.) so izjeme tipa *irq* ali *fiq*, ki jih vektorski nadzornik prekinitev (ang. *Vector Interrupt Controller*) ne maskira.

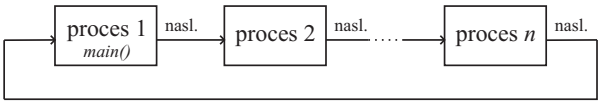
†Razen načinov *usr* in *sys*, ki imata skupen kazalec sklada [5].

`0x00000000`[‡]. Druge datoteke vsebujejo spremljajočo kodo, potrebno za inicializacijo mikrokrmilnika (nastavitev fazno sklenjene zanke, smeri pinov ipd.).

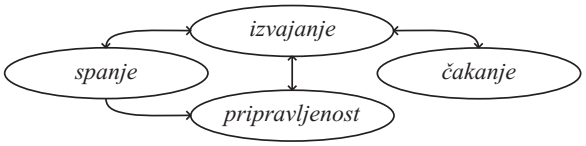
3 ZAGON OPERACIJSKEGA SISTEMA IN PROCESOV

Po izvršitvi zagonske kode v *crt0.s* se program nadaljuje v funkciji *start_up()*. Tu se s klicem funkcije *os()* inicializira operacijski sistem. V tem trenutku postane *start_up()* začetni proces. Na operacijskem sistemu tako po zagonu teče en proces (tj. funkcija *main()*, klicana iz *start_up()*).

Proces je funkcija tipa *void* brez argumentov. Nov proces se ustvari ob sistemskem klicu *os_process_create ()*. Operacijski sistem podatke o procesih hrani v cikličnem povezanem seznamu, prikazanem na sliki 1. Pomnilnik za sklad (ang. *stack*) je procesu dodeljen na kopici (ang. *heap*), kar ne velja za začetni proces, ki uporablja sklad načina *usr*. Za nov proces je privzeta najnižja prioriteta (spremeniti jo je mogoče s sistemskim klicem *os_process_priority()*) in stanje *pripravljenost*i na izvajanje. Začetni proces je zopet izjema. Njegovo začetno stanje je *izvajanje*. Poleg stanj *izvajanje* in *pripravljenost* se proces lahko nahaja še v stanjih *spanje* (proces določen časovni interval miruje) in *čakanje* (proces je blokiran) (slika 2).



Slika 1: Ciklični seznam procesov (nasl. = naslednji proces)



Slika 2: Prehodi med stanji procesa

Proces se konča, ko se njegova funkcija izvede do konca. Predčasno ga je mogoče končati s sistemskim klicem *os_process_kill()*. Zaključitev zadnjega procesa povzroči smrtni objem (ang. *deadlock*) operacijskega sistema.

Kazalec na funkcijo procesa je hkrati tudi identifikator procesa. Sočasno lahko teče le en primerek posameznega procesa. Operacijski sistem ne dovoljuje vzporednega teka več primerkov istega procesa.

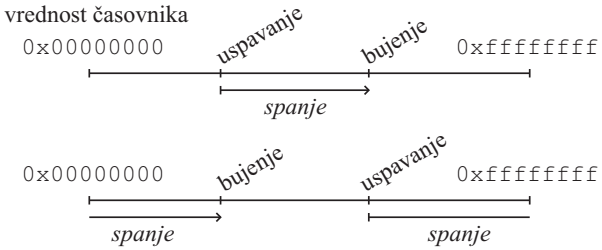
Procesne spremembe vplivajo na razvrščanje procesov. Zato procesni sistemski klici, kakor tudi končanje procesa, prožijo razvrščevalnik procesov operacijskega sistema.

[‡]Ob resetu mikrokrmilnika se prvi ukaz izvede iz naslova `0x0000 0000`.

4 STANJI SPANJA IN ČAKANJA

Časovnik operacijskega sistema šteje mikrosekunde od zagona naprej. Trenutno stanje časovnika vrne sistemski klic *os_timer()*. Časovnik je 32-bitni števnik. Njegovo stanje se ponovi na vsakih $2^{32}\mu\mathrm{s} \approx 1\mathrm{h}\ 11\mathrm{min}\ 35\mathrm{s}$.

Proces gre iz stanja *izvajanje* v *spanje* (slika 2) s sistemskim klicem *os_timer_sleep()*. Sprememba sproži razvrščevalnik procesov. Zaradi ciklične narave časovnika je trenutek bujenja določen tudi s trenutkom uspavanja, najdaljši čas spanja pa je $(2^{32} - 1)\mu\mathrm{s}$ (slika 3). Iz stanja *spanje* se proces prebudi v *izvajanje* (proces ima najvišjo prioriteto) ali *pripravljenost* (proces nima najvišje prioritete).



Slika 3: Trajanje stanja *spanje*

Operacijski sistem omogoča zaklepanje virov s pomočjo binarnih semaforjev [7]. Binarni semafor je realiziran kot spremenljivka tipa *int* z vrednostjo nič (deklarirano z *UNLOCKED*), ko je vir na voljo, oziroma z vrednostjo kazalca na funkcijo procesa, ki je vir zasedel. Poskus zasedbe semaforja/vira (ang. *wait semaphore operation*) izvede sistemski klic *os_mutex_lock()*. V primeru neuspeha je proces blokiran (stanje *čakanje*), kar sproži razvrščevalnik procesov operacijskega sistema, ki centralno procesno enoto dodeli drugemu procesu.

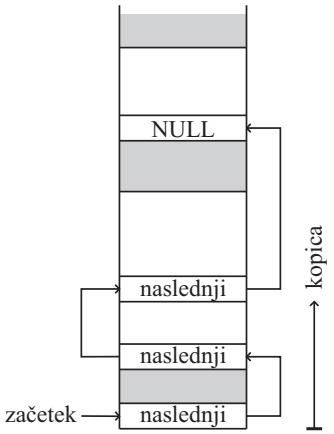
Sprostitev semaforja (ang. *post semaphore operation*) naredi sistemski klic *os_mutex_unlock()*. Pri tem se sproži razvrščevalnik procesov. Blokiran proces iz stanja *čakanje* nazaj v stanje *izvajanje* postavi razvrščevalnik procesov, ko izbere blokiran proces, katerega semafor je bil sproščen. Sveže odblokiran proces najprej ponovno poskusi z zasedbo semaforja.

5 DINAMIČNO DODELJEVANJE POMNILNIKA

Dinamično dodeljevanje pomnilnika (ang. *dynamic memory managment*) na kopici izvajata funkciji *os_allocate ()* in *os_free()*. Za preprečitev hkratnega dostopa do podatkov o dodeljenem pomnilniku funkciji uporabljata binarni semafor *alloc*. Dinamično dodeljevaje pomnilnika je uporabljeno tudi ob ustvarjanju in končanju procesa. Poleg prostora s podatki o pocesu se vsakemu procesu dinamično dodeli še prostor za sklad.

Dodeljen pomnilnik (ang. *allocated memory*) je organiziran v povezan seznam, prikazan na sliki 4. Seznam

je urejen po naslovih dodeljenih kosov (ang. *allocation*) pomnilnika. Prvi kos označuje začetek seznama in ni nikdar sproščen. Pomeni navidezno dodelitev nič zlogov (ang. *byte*) pomnilnika. Funkcija *os_allocate()* implementira strategijo prvega primernega prostora za dodelitev (ang. *first fit algorithm*) [8].

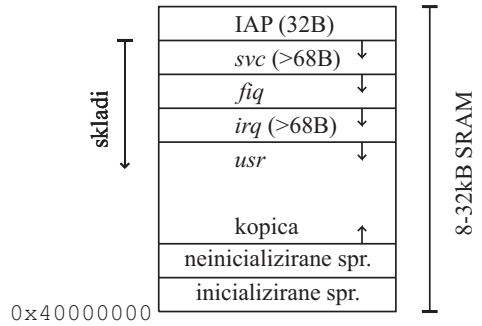


Slika 4: Urejen povezan seznam z dodeljenim pomnilnikom. Prost pomnilnik je osenčen.

Mikrokrmilniki družine LPC213x nimajo enote za upravljanje delovnega pomnilnika (ang. *Memory Management Unit*). Tako je dinamično dodeljen pomnilnik viden vsem procesom, česar operacijski sistem ne more omejiti.

6 RAZDELITEV DELOVNEGA POMNILNIKA

Operacijski sistem uporablja prekinitev časovnika *timer0* (tj. izjemo/način *irq*) za časovno rezinjenje in programsko prekinitev (tj. izjemo *swi* v načinu *svc*) za sistemske klice. Na vsakem izmed skladov načinov delovanja *irq* in *svc* mora biti na voljo vsaj 68 zlogov namenjenih trenutnemu stanju jedra (sedemnajst 32-bitnih delovnih registrov). Razporeditev je prikazana na sliki 5. Sklad *fiq* neodvisno od operacijskega sistema uporabljajo prekinitve, deklarirane kot izjeme tipa *fiq*, sklad *usr* pa uporablja začetni proces. 32 zlogov na vrhu delovnega pomnilnika je na voljo ukazom za zapisovanje v pomnilnik FLASH (ang. *In Application Programming*).



Slika 5: Razdelitev delovnega pomnilnika SRAM

Na drugi strani delovnega pomnilnika se nahajajo inicializirane in neinicializirane globalne in statične spremenljvke, ki jim sledi kopica. Naslovi skladov s slike 5 so postavljeni v datoteki *crt0.s*.

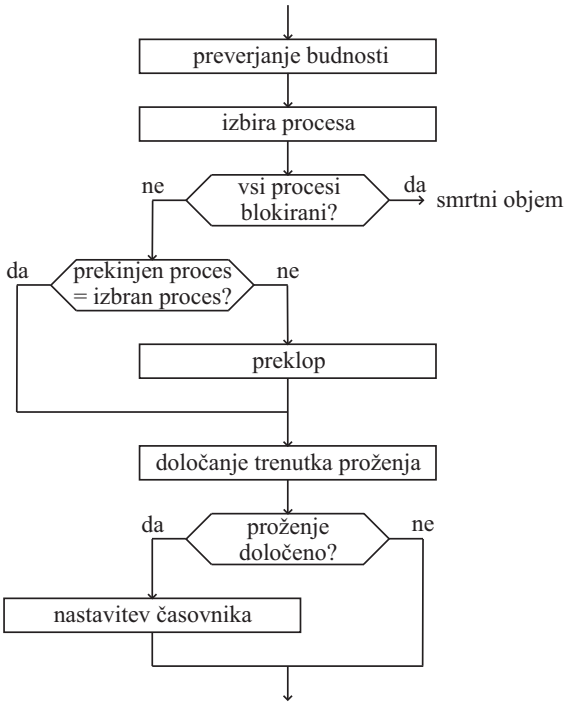
```
.equ svc_stack, 0x40007fe0 /* 128B */
.equ fiq_stack, 0x40007f60 /*  64B */
.equ irq_stack, 0x40007f20 /* 128B */
.equ usr_stack, 0x40007ea0
```

Skladi procesov, razen sklada začetnega procesa, se nahajajo na kopici. Pri poplavljanju katerega izmed skladov (ang. *stack overflow*) ni mehanizma, ki bi napako avtomatično zaznal. Prav tako ni mogoče zaznati poplavljanja sklada *usr* začetnega procesa prek kopice ali nasprotno. Zasedenost delovnega pomnilnika je treba oceniti vnaprej, pri čemer je treba upoštevati učinek razdrobljenosti kopice (ang. *heap fragmentation*).

7 RAZVRŠČEVALNIK PROCESOV

Razvrščevalnik procesov je srce operacijskega sistema. Njegova naloga je izbira procesa, ki naj v nadaljevanju začne oziroma nadaljuje izvajanje. Prožen/klican je ob spremembah, ki vplivajo na razvrščanje procesov. Prožijo ga sistemski klici ali potek časovnega intervala.

Poznanih je več politik razvrščanja procesov [9], [10]. Predstavljen operacijski sistem spada med predkupne operacijske sisteme v realnem času (ang. *preemptive real-time operating system*) s prioritetnim razvrščanjem. Algoritem razvrščanja (slika 6) je v osnovi preprost.



Slika 6: Algoritem razvrščevalnika procesov

Izbran je proces z najvišjo prioriteto. Če je takšnih procesov več, se čas centralne procesne enote med njimi enakomerno porazdeli.

Razvrščevalnik na sliki 6 procese, katerih čas spanja je potekel (slika 3), prestavi v stanje *pripravljenost* (slika 2). Nato izbere buden, neblokiran proces z najvišjo prioriteto. Če vsi procesi spijo, se pogoj budnosti umakne. Če je semafor blokiranega procesa sproščen, se tak proces pri izbiri šteje za neblokiranega. Če obstaja več enakovrednih procesov, je izmed njih izbran prvi v cikličnem seznamu (slika 1) od prekinjenega procesa naprej.

Izbira naslednjega procesa ne uspe, če so vsi procesi medsebojno blokirani. Razvrščevalnik se ujame v neskončno zanko oziroma v smrtni objem. Sicer je izbran proces prestavljen v stanje *izvajanje*, prekinjen pa v stanje *pripravljenost*. Razvrščevalnik izvede preklop med procesoma (ang. *context switch*). Če je izbran prekinjen proces, preklopa med procesoma ni.

Sledi izračun naslednjega časovno pogojenega proženja razvrščevalnika. Trenutek proženja je enak:
- trenutku prvega bujenja (vsi procesi v stanju *spanje*),
- trenutku bujenja procesa z višjo prioriteto ali
- poteku ene časovne rezine (več procesov z enako prioriteto).

Časovnik *timer0*, ki proži razvrščevalnik, se postavi na prvega izmed omenjenih časov. Če ni izpolnjen noben od zgornjih pogojev, se časovnik ne postavi (npr. izbran proces je buden in ima najvišjo prioriteto).

Iz zgradbe razvrščevalnika procesov sledi, da je operacijski sistem trd realnočasovni sistem (ang. *hard real-time operating system*) le za proces z najvišjo prioriteto. Procesu z nižjo prioriteto pravočasnost ni zagotovljena.

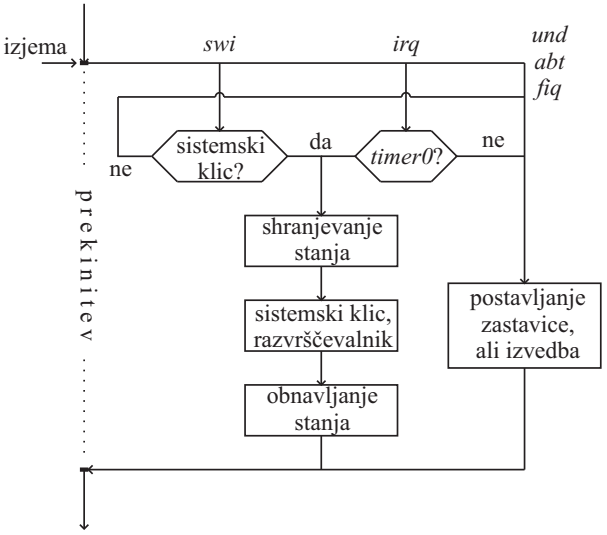
Razvrščevalnik procesov začne delovati ob inicializaciji operacijskega sistema, tj. ob klicu funkcije *os()*.

8 PREKINITVE IN SISTEMSKI KLICI

Izjemo oziroma prekinitev prestreže koda v datoteki *exceptions.s*, ki je del jedra operacijskega sistema. Od tod naprej je izjema/prekinitev lahko izvedena neodvisno od operacijskega sistema ali pa čaka na dodelitev centralne procesne enote kot nov proces. V prvem primeru je operacijski sistem ustavljen za čas izvajanja prekinitve. Takšna prekinitev je podobna procesu z najvišjo prioriteto. V njej ni dovoljeno uporabljati sistemskih klicev operacijskega sistema (npr. ustvarjanje novega procesa, dodeljevanje pomnilnika ipd.).

Če se izjema/prekinitev izvede pozneje kot proces v okviru operacijskega sistema, potem se postavi le zahteva po njej. Poseben proces periodično preverja zahteve po prekinitvah (ang. *interrupt polling*) in na zahtevo ustvari nov proces, ki prekinitev izvede. Zakasnitev začetka izvajanja prekinitve (ang. *latency*) je odvisna od frekvence preverjanja prekinitvenih zahtev. Takšna prekinitev lahko uporablja sistemske klice operacijskega sistema.

Za sistemske klice je uporabljena izjema *swi* (način *svc*), za proženje razvrščevalnika procesov pa prekinitev *timer0* (izjema/način *irq*). Stanje prekinjenega procesa se shrani na sklad pripadajočega načina delovanja. Sledi sistemski klic. Posebno mesto ima razvrščevalnik procesov, ki izvede preklop med procesoma. To naredi tako, da zamenja vsebino na skladu s podatki prekinjenega procesa. Princip delovanja izjem je prikazan na sliki 7.



Slika 7: Izvajanje izjem

Ker je gnezdenje prekinitev *irq* onemogočeno, so sistemski klici, kakor tudi razvrščevalnik procesov atomske operacije.

9 IZMERJENE LASTNOSTI SISTEMA

Izvorna koda operacijskega sistema je napisana v programskem jeziku C, razen strojno specifičnih odsekov, ki so napisani v zbirniku. Pri prevajanju s prevajalniško zbirko GCC brez optimizacije velikosti kode zavzame 5108 zlogov, od tega inicializacija sistema 588 zlogov, sistemski klici 2324 in razvrščevalnik 2196 zlogov.

Lastnosti sistema so bile izmerjene po metodi Thread-Metric [11] na mikrokrmilniku LPC2138 s taktom 60MHz. Rezultati v tabeli 1 podajajo število ponovitev testirane lastnosti v 30 sekundah.

Tabela 1: Lastnosti sistema po metodi Thread-Metric

Meritev	**Št. ponovitev**
relativna hitrost mikrokrmilnika	30818
sodelujoče razvrščanje	566150
predkupno razvrščanje	348228
prekinitve brez preklopa med proc.	1490065
prekinitve s predkupnim preklopom	328587
zaklepanje virov	1669756
dinamično dodeljevanje pomnilnika	662250

Meritve so bile opravljene na kodi, prevedeni brez optimizacije na hitrost. Sodelujoče razvrščanje (ang. *cooperative scheduling*) je bilo izvedeno s postavitvijo zahteve po prekinitvi *timer0* (razvrščevalnik) znotraj procesa. Podobno so prožene prekinitve brez in s predkupnim preklopom. Metoda Thread-Metric izvaja predkupno razvrščanje (ang. *preemptive scheduling*) z ekspicitnim postavljanjem procesov v stanje čakanja oziroma pripravljenosti, česar predstavljeni operacijski sistem ne pozna. Enak učinek je bil dosežen s spreminjanjem prioritet. Test dinamičnega dodeljevanja pomnilnika je bil izveden konservativno z dodeljevanjem in sproščanjem pomnilnika na koncu povezanega seznama (slika 4).

10 SKLEP

Predstavljen operacijski sistem za vgrajene sisteme s procesnim jedrom ARM7 teče v realnem času s prioritetnim razvrščanjem. Je univerzalen, hkrati pa modularen in preprost za uporabo. Modularna zgradba omogoča prilagoditev ali zamenjavo dela sistema glede na potrebe aplikacije (npr. spreminjanje politike razvrščanja, zamenjava cikličnega seznama procesov s prioritetnim, uporaba bitne predstavitve za dinamično dodeljevanje pomnilnika ipd.). Proženje razvrščevalnika procesov je dogodkovno. Operacijski sistem je napisan v programskem jeziku C, kar omogoča relativno preprost prenos na mikrokrmilnike z drugačnim jedrom. Dostopna izvorna koda je implementirana za 32-bitne mikrokrmilnike iz družine LPC213x.

ZAHVALA

Raziskavo je sofinanciralo Ministrstvo Republike Slovenije za izobraževanje, znanost in šport v okviru programa P2-0246 - Algoritmi in optimizacijski postopki v telekomunikacijah.

LITERATURA

[1] SEOS: Simple Educational Operating System, del knjižnice *http://fides.fe.uni-lj.si/~janezp/libraries.zip*, Jul. 2014.
[2] FreeRTOS, *http://www.freertos.org*, Jul. 2014.
[3] ThreadX Real Time Operating System, *http://rtos.com/products/threadx*, Jul. 2014.
[4] *LPC2131/2/4/6/8 User manual*, Rev. 4, Doc. ID UM10120, 23 Apr. 2012, NXP.
[5] *ARM7TDMI-S Technical Reference Manual*, Rev. r4p3, DDI 0234B, 2001, ARM Ltd.
[6] A.N. Sloss, D. Symes, C. Wright, J. Rayfield, *ARM System Developer's Guide: Designing and Optimizing System Software*, Elsevier, 2004.
[7] E.W. Dijkstra, *Cooperating Sequential Processes*, Technological University, Eindhoven, The Netherlands, Sep. 1965, ponatisnjeno v *Programming Languages*, ured. F. Genuys, str. 43-112, Academic Press, NY, 1968.
[8] D.E. Knuth, *The Art of Computer Programming Vol. 1, Fundamental Algorithms*, 3rd ed., Adison-Wesley, 2008.
[9] J.W.S. Liu, *Real-Time Systems*, Prentice-Hall, NJ, 2000
[10] A. Silberschatz, P.B. Galvin, G. Gagne, *Operating System Concepts*, 8th ed. update, Wiley, NJ, 2012.
[11] Thread-Metric Benchmark Suite, *http://rtos.com/PDFs/MeasuringRTOSPerformance.pdf*, Jul. 2014.

Janez Puhan je leta 2000 doktoriral s področja elektrotehnike na Univerzi v Ljubljani. Je asistent na Fakulteti za elektrotehniko. Njegovo področje raziskovanja obsega modeliranje, simulacijo in optimizacijske postopke pri računalniškem načrtovanju vezij, kakor tudi programsko in strojno opremo vgrajenih sistemov.