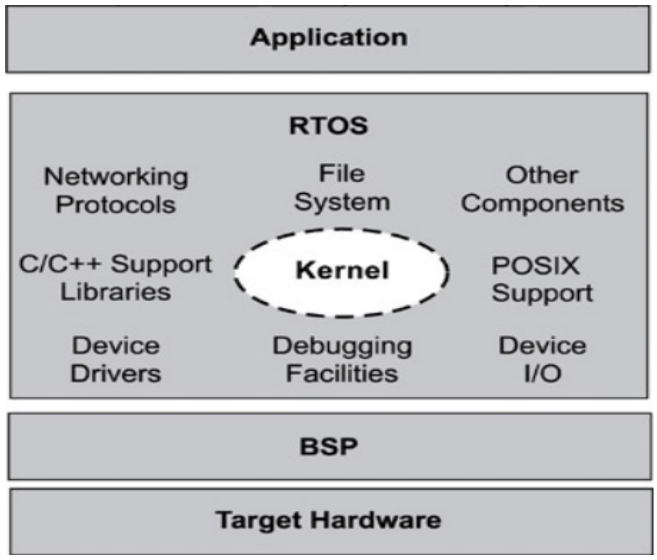
Predavanje 2

Operativni sistemi za rad u realnom vremenu (RTOS)

2.1 Definicija RTOS-a

Real-time operativni sistem (Real time operating system, RTOS) je program koji upravlja izvršavanjem procesa sa stanovišta vremena, upravlja sistemskim resursima i omogućava konzistentan interfejs za razvoj aplikacija. Kod aplikacije razvijen u okviru RTOS-a može biti različite složenosti i strukture: od jednostavnih aplikacija koje upravljaju ulazno-izlaznim portovima do složenih aplikacija sa zahtevnom obradom signala i ogromnim zahtevima za sistemskim resursima. Uloga operativnog sistema je u takvim slučajevima ključna, oni moraju da budu skalabilni u cilju zadovoljavanja različitih zahteva postavljenih pred njih. Na primer, u nekim aplikacijama, RTOS se sastoji samo od kernela, koji kao jezgro operativnog sistema, i pokriva samo minimalne algoritme za dodeljivanje vremena (scheduling), upravljanje resursima i vršenje kontrolne logike. Svaki RTOS mora da sadrži kernel. Sa druge strane RTOS može biti kombinacija više relativno složenih modula osim kernela, koji su zaduženi za fajl sistem, implementaciju mrežnog protokol steka i ostali komponenti neophodnih za određenu aplikaciju, kao što je prikazano na slici 1.



Slika 1. Struktura RTOS-a

Iako je većina RTOS-a skalabilna u cilju zadovoljavanja određenih aplikacija sa minimalnim zauzećem resursa, ovde ćemo akcenat staviti na minimalan podskup funkcionalnosti kernela RTOS-a, koji gotovo uvek sadrži:

1) Planer
je zadužen da obezbedi skup algoritama koji određuju koji proces je aktivan u kom vremenskom trenutku. Neki tipični primeri algoritama koji se koriste za planiranje su prioritetni algoritam sa prekidanjem (preemptive scheduling with priorities) i algoritam za kooperativno planiranje (round -

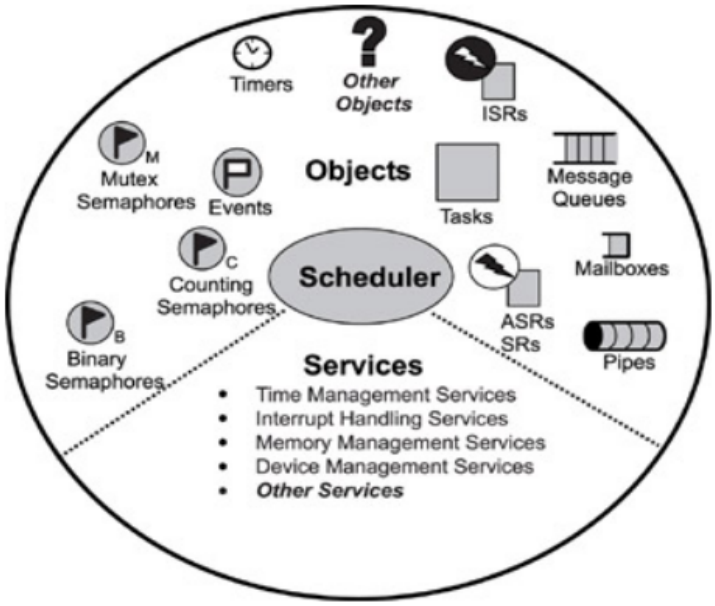
robin scheduling).

2) Objekti

su specijalne kernel strukture koje omogućavaju razvoj aplikacija za embedded sisteme u realnom vremenu. Tipični kernel objekti su semafori, procesi i redovi poruka (message queues).

3) Servisi

predstavljaju operacije koje kernel izvršava na objektima, operacije kao što su informacije o vremenu (system tick), manipulisanje prekidima, raspodela resursa, itd.



Slika 2. Uprošćena struktura kernela

2.2 Planer

Planer je najosnovniji blok unutar svakog RTOS-a. On obezbeđuje algoritme koji određuju koji proces se izvršava u kom vremenskom intervalu. Razumevanje rada planera ovde ćemo najpre opisati:

- rasporedive entitete
- multitasking
- zamenu konteksta (context switching)
- dispatcher
- algoritme za raspodelu procesorskog vremena

2.2.1 Rasporedivi entiteti

Rasporedivi entiteti (schedulable entities) su kernel objekti koji su ravnopravni sa stanovišta planera u pogledu dodele vremenskih resursa, a sve u skladu sa predefinisanim algoritmom dodeljivanja istih. Primeri ovih entiteta su task-ovi i procesi koji se uglavnom mogu pronaći kao sastavni deo većine kernela.

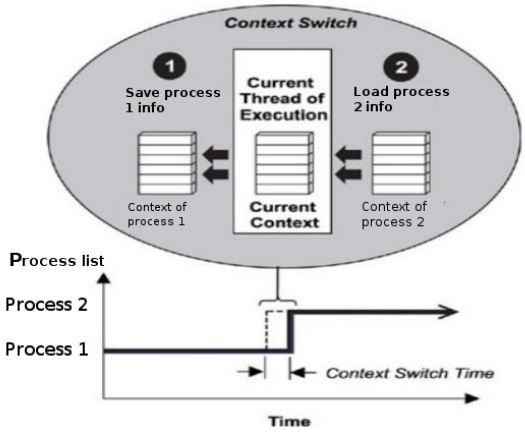
Task je nezavisna nit (thread) koja se izvršava i koja se sastoji od nezavisne sekvence instrukcija.

Određeni kerneli predstavljaju različit entitet koji se zove proces. Procesi su slični task-ovima sa stanovišta činjenice da ravnopravno i nezavisno konkurišu za dodelu procesorskog vremena. Ipak, procesi se razlikuju od taskova jer generalno omogućavaju bolje mehanizme zaštite memorije, po cenu degradacije performansi. U nastavku teksta, uprkos različitim karakteristikama procesa i task-ova, koristićemo termin proces kao zajednički naziv i za task-ove i za procese.
Treba primetiti da semafori i redovi poruka nisu rasporedivi entiteti. Oni predstavljaju objekte zadužene za inter-procesnu komunikaciju i sinhronizaciju.

Postavlja se pitanje kako planera manipuliše više rasporedivih entiteta koji treba da se izvršavaju simultano? Odgovor je: multitasking.

2.2.2 Multitasking

Multitasking predstavlja sposobnost operativnog sistema da istovremeno manipuliše i kontroliše više aktivnosti pri čemu će svaka od njih biti izvršena u za to predviđenom vremenu. Real-time kernel može sadržati više procesa koji se izvršavaju u paraleli kao što je prikazano na slici 3.



Slika 3. Multitasking na primeru dva procesa koji se izvršavaju u paraleli

U ovakvom scenariju multitasking omogućava izvršavanje više procesa koji se naizgled izvršavaju istovremeno. Ipak, planera obezbeđuje da procesorsko vreme dobijaju naizmenično svi procesi koji učestvuju u multitasking izvršavanju, u skladu sa unapred određenim algoritmom.
Planer mora da obezbedi da odgovarajući task počne sa izvršavanjem u adekvatnom trenutku. Ovde je važno spomenuti da se procesi izvršavaju u skladu sa algoritmom za raspodelu procesorskog vremena, dok se prekidne rutine (ISR) izvršavaju u zavisnosti od hardverskih prekida i prioriteta u kontekstu prekida, nezavisno od izvršavanja procesa. Kako se povećava broj procesa koji se izvršavaju, povećavaju se i potrebe za performansama procesorskog jezgra (CPU). Ovo je rezultat propagacije zamene konteksta prilikom zamene procesa koji se izvršava.

2.2.3 Zamena konteksta

Svaki proces koji se izvršava, ima svoj kontekst, koji predstavlja stanje CPU registara zahtevano svaki put kada je tom procesu dodeljen procesor i kada taj proces kreće sa izvršavanjem.
Zamena konteksta se dešava uvek kada planer menja proces koji će se izvršavati od strane

procesora. Tipičan kernel u ovom slučaju vrši sledeće operacije:
prilikom kreiranja svakog pojedinačnog procesa, kernel kreira i kasnije održava takozvani Proces Control Block (ili PCB).PCB-ovi su sistemske strukture podataka koje kernel koristi da bi upravljao informacijama vezanim za svaki konkretan proces. Oni sadrže sve informacije koje kernel treba da zna u vezi sa svakim pojedinačnim procesom. Dok se proces izvršava, njegov kontekst se takođe dinamički osvežava, što je podržano dinamičkim osvežavanjem odgovarajućeg PCB-a. Kada se proces ne izvršava, njegov kontekst je "zamrznut" u okviru PCB-a, kako bi se mogao restorirati sledeći put kada proces opet bude aktiviran. Tipičan scenario prilikom zamene konteksta je prikazan na slici 3.

Kao što se vidi na slici 3, kada kernel odluči da treba zaustaviti proces 1 i pokrenuti proces 2, preduzimaju se sledeći koraci:
1) kernel čuva kontekst procesa 1 u okviru njegovog PCB-a
2) kernel učitava kontekst procesa 2 iz odgovarajuće PCB-a, koji postaje proces koji će sledeći da se izvršava
3) kontekst procesa 1 je zamrznut dok se proces 2 izvršava, ali ukoliko planer odluči da ponovo aktivira proces 1, on će nastaviti tamo gde je stao prilikom prethodnog zaustavljanja od strane kernela, neposredno pre nego što se desila zamena konteksta.

Vreme koje je potrebno da bi planer izvršio zamenu procesa, uglavnom ne predstavlja problem sa stanovišta izvršavanja svakog od procesa. Ipak, u slučajevima kada se zamena konteksta vrši veoma često, pogotovo u aplikacijama koje su izuzetno vremenski zahtevne, zamene konteksta mogu rezultirati u ozbiljnijem narušavanju performansi sistema. Dakle, aplikacija uvek treba da se dizajnira tako da se minimizuje broj zamena konteksta, koliko je tako nešto moguće.
Svaki put kada aplikacija izvrši sistemski poziv, planer ima mogućnost da odluči da li je neophodno izvršiti zamenu konteksta. Kada planer odluči da je neophodna zamena konteksta, planer se oslanja na modul koji se naziva dispečer i koji je zadužen za samu zamenu konteksta.

2.2.4 Dispečer

Dispečer je modul koji je usko povezan sa planerom i on je zadužen za zamenu konteksta i toka izvršavanja aplikacije. U bilo kojem trenutku rada RTOS-a, tok izvršavanja aplikacije, takođe poznat i kao flow control, prolazi kroz jedno od tri stanja: aplikacija, ISR ili kernel. U trenutku kada proces ili ISR vrši sistemski poziv, kontrola toka izvršavanja se prepušta kernelu kako bi se izvršila jedna od sistemskih rutina u skladu sa sistemskim pozivom. U trenutku napuštanja kernela, dispečer ima ulogu da odluči kojem procesu (u okviru aplikacije) će biti prepuštena kontrola nad procesorom.
Nije neophodno da se kontrola vrati procesu koji je pozvao sistemski poziv. Algoritam za planiranje procesorskog vremena ima ulogu da odluči koji će proces sledeći da se izvršava, dok je dispečer zadužen za samu zamenu konteksta i prepuštanje kontrole nad CPU-om.
U zavisnosti od toga kako se pozvao kernel, dispečer može preduzeti različite akcije. Kada je kernel kod pokrenut na zahtev procesa koji je izvršio sistemski poziv, dispečer se koristi za napuštanje kernela svaki put kada se sistemski poziv završi. U ovom scenariju dispečer se koristi na tzv call-to-call bazi tako da može da koordinira stanjima procesa i tranzicijama stanja kao posledicama sistemskog poziva koji se upravo izvršio (na primer, jedan ili više procesa su možda postali spremni za izvršavanje). Sa druge strane, ukoliko je ISR izvršila sistemski poziv, izvršavanje dispečera se odlaže dok ISR u potpunosti ne završi svoje izvršavanje. Ovo važi i u slučaju da se npr. oslobodio neki od resursa koji bi inače izazvao aktiviranje određenog procesa i zamenu konteksta. Ipak, ovakva zamena konteksta se ne dešava jer ISR mora završiti svoje izvršavanje bez prekida od stane procesa. Tek kada se ISR završi, kernel prepušta kontrolu dispečeru koji odlučuje kom procesu

prepustiti kontrolu nad procesorom.

2.2.5 Algoritmi za planiranje procesorskog vremena

Kao što je pomenuto ranije, planer i dispečer odlučuju koji će se proces sledeći izvršavati kao rezultat algoritma koji donosi tu odluku (takođe poznato i kao scheduling policy).
Većina savremenih kernela podržava dva standardna algoritma:

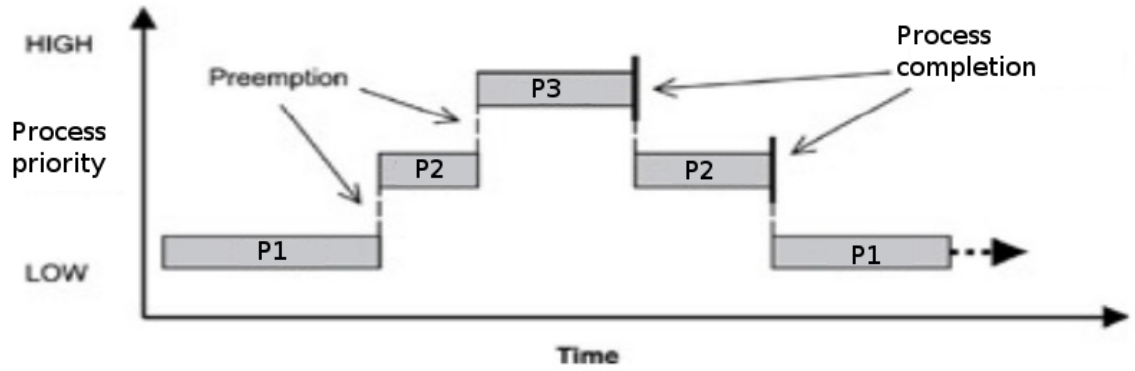
- prioritetno planiranje sa prekidanjem
- ko-operativno planiranje

U nekim slučajevima korisnici mogu kreirati svoje algoritme.

Prioritetno planiranje sa prioritetima

Većina real-time operativnih sistema koristi ovaj algoritam podrazumevano. Kao što je prikazano na slici 4, proces koji će se sledeći izvršavati je proces sa najvećim prioritetom u poređenju sa svim ostalim procesima koji su spremni za izvršavanje.

RTOS generalno podržava 256 nivoa prioriteta pri čemu je 0 najveći a 2 najniži. Neki kerneli definišu obrnuto označavanje nivoa prioriteta (inverzno) pri čemu procesi sa prioritetom 255 imaju najviši a sa prioritetom 0 najniži prioritet, što svakako ne menja logiku i mehanizam ovog algoritma. Ovim algoritmom, svaki proces ima prioritet i onaj koji ima najviši prioritet će se izvršavati sledeći. Ukoliko proces sa višim prioritetom postane spreman za izvršavanje, dok se izvršava proces sa nižim prioritetom, kernel momentalno čuva kontekst procesa koji se trenutno izvršava u njegovom PCB-u, vrši zamenu konteksta i prepušta kontrolu nad CPU-om procesu sa višim prioritetom. Na primeru sa slike 4, proces 1 je prekinut od strane procesa 2 sa višim prioritetom, a nakon toga je opet proces 2 prekinut od strane procesa sa prioritetom višim od njega. Kada se proces 3 završi, proces 2 nastavlja sa svojim izvršavanjem, a na sličan način, čim se proces 2 završi, proces 1 nastavlja da se izvršava, sve dok ne završi ili ponovo ne bude prekinut od strane nekog procesa sa višim prioritetom.
Iako se prioritet procesa dodeljuje prilikom kreiranja procesa, prioritet procesa može biti promenjen dinamički korišćenjem sistemskih poziva omogućenih od strane kernela. Mogućnost dinamičke promene prioriteta svakog od procesa takođe omogućava embedded sistemu fleksibilnost sa stanovišta eksternih događaja i trenutaka kada se oni pojavljuju. Kao rezultat, moguće je dizajnirati istinski real-time sistem. Ipak, treba imati na umu da se pogrešnim korišćenjem ovih mogućnosti dinamičke promene prioriteta procesa, mogu izazvati ozbiljni problemi kao što su inverzija prioriteta (o kojoj će biti više reči kasnije), mrtve petlje i greške koje vode ka neminovnom padu sistema u celini.

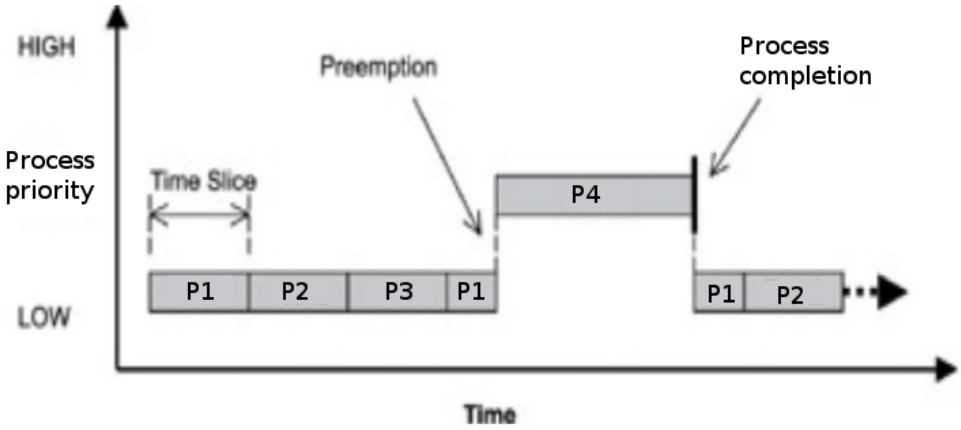


Slika 4. Prioritetno planiranje sa prekidanjem

Problem prioritetnog planiranja sa prekidanjem leži u činjenici da se procesi sa malim prioritetom možda nikada neće izvršiti (tzv. Problem izgladnjivanja, *starvation*). Problem se rešava pomoću *Aging* modifikacije algoritma po kojoj se prioriteti procesa menjaju proticanjem vremena.

Ko-operativno planiranje

Ko-operativno planiranje procesorskog vremena obezbeđuje jednako trajanje procesa i jednaku raspodelu CPU vremena. U svojoj originalnoj formi, ovaj algoritam ne može zadovoljiti real-time zahteve jer kod sistema u realnom vremenu, obavljanje procesa se vrši u skladu sa prioritetima istih (koji su često i promenljivi). Umesto toga, prioritetno planiranje bazirano na prioritetima može biti korišćeno zajedno sa ko-operativnim pristupom u smislu da se ravnopravna raspodela CPU vremena vrši između procesa istog prioriteta, dok se oni višeg prioriteta izvršavaju u “prekidačkom” maniru.



Slika 5. Prekidačko ko-operativno planiranje

Raspodelom vremena, svaki proces se izvršava tokom intervala predefinisanog trajanja. Tajmer prati trajanje vremenskog slota za svaki od procesa, inkrementirajući vrednost nakon svakog clock tik-a. Onog trenutka kada vreme istekne za određeni proces, njegov tajmer se briše, i taj proces se stavlja na kraj reda čekanja. Takođe, naknadno dodati procesi se uvek dodaju na kraj reda, sa dodeljenim tajmerima inicijalizovanim na 0. Ukoliko je neki proces u round-robin maniru prekinut od strane procesa sa višim prioritetom, njegov run-time counter (tajmer) se pamti i njegova vrednost se ponovo učitava kada je prekinut proces ponovo određen za izvršavanje. Ova ideja je prikazana na slici 5 na kojoj vidi kako je proces 1 prekinut od strane procesa 4 višeg prioriteta, ali proces 4 nastavlja tamo gde je stao kada za to dođe vreme (kada proces 4 završi izvršavanje).

Osim ova dva najčešće korišćena algoritma, postoji i nekolicina dodatnih algoritama koji se koriste.

First Come First Served (FCFS) planiranje

Kao što mu samo ime kaže, ovaj pristup favorizuje procese u skladu sa tim kada postaju spremni za izvršavanje. Na primeru tri procesa, sa trajanjima
T1 24unit
T2 3unit
T3 3unit

ukoliko pristižu u redosledu T1, T2, T3, izvršavanje je

T1	T2	T3
0	24	27

Pri čemu je prosečno vreme čekanja:
T1 = 0,
T2 = 24
T3=27
Tavg = (0+24+27)/3=17
Međutim, ukoliko procesi pristižu drugačijim redosledom T2, T3,T1 situacija je:

T2	T3	T1
0	3	6

i prosečno vreme čekanja je tada:
T1=6,
T2=0,
T3=3
Tavg=(6+0+3)/3=3

što je mnogo prihvatljivije nego u prethodnom slučaju. Efekat koji je zapažen u ovom slučaju se naziva konvoj efekat (*Convoy effect*) i nastaje kada se kratak proces izvršava nakon što se duži procesi izvrše.
Očigledno, ovakav nemodifikovan FCFS algoritam ne može da se koristi u embedded real-time sistemima.

Shortest Job First (SJF) planiranje

Bazira se na pristupu gde se svakom procesu dodeli vremena neophodno da bi se on izvršio. Ovo trajanje se koristi prilikom određivanja koji će se proces sledeći izvršavati, pri čemu će biti privilegovan proces sa najkraćim trajanjem.

Postoje dva pristupa u ovom algoritmu:
1) ne-prekidajuće: proces kome je jednom dodeljeno procesorsko vreme ne može biti prekinut sve dok se kompletan proces ne završi
2) prekidajuće: ukoliko se pojavi novi proces sa trajanjem kraćim od preostalog trajanja procesa

koji se izvršava, prekini proces koji se izvršava, u suprotnom proces nastavlja da se izvršava do samog završetka. Ova shema je poznata kao Shortest Remaining Time First (SRTF).
Ovakav pristup je optimalan i uvek daje najkraće srednje vreme čekanja za dati skup procesa. Primer:

Proces	Ready	Duration
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

SJF (non-preemptive)

P1	P3	P2	P4
0	7	8	12	16

Srednje vreme čekanja je Tavg=[0+(8-2)+(7-4)+(12-5)]/4 =4

SJF (preemptive)

P1	P2	P3	P2	P4	P1
0	2	4	5	7	11	16

Srednje vreme čekanja je Tavg=[9+1+0+2]/4 =3

Red čekanja sa više nivoa (Multilevel queue)

- red procesa koji su spremni za izvršavanje (ready) se deli na nekoliko nezavisnih redova, npr foreground, background,...
- Svaki red ima svoj algoritam za planiranje procesorskog vremena, npr. foreground round-robin, backgroung - FCFS
- ostaje problem što treba opet razrešiti planiranje između redova. Ovo se može uraditi kao
  - fiksan prioritet redova: usluži sve iz foreground reda, a zatim pređi na background. Problem: starvation.
  - Time slice: svaki red dobija vremenski slot u kome radi po unapred predefinisanom algoritmu. Npr: 80% foreground RR, 20% background FCFS.

Dinamički red čekanja sa više nivoa (Multilevel feedback queue)

- Proces može da menja poziciju u različitim redovima: aging može biti implementiran na ovaj način
- Planiranje u dinamičkom redu čekanja sa više nivoa je definisano pomoću sledećih parametara:
  - broj redova
  - algoritam za planiranje procesorskog vremena za svaki od redova
  - metod na osnovu koga se odlučuju kada da se proces prebaci u red “iznad”
  - metod na osnovu koga se odlučuje kada da se proces prebaci u red “ispod”
  - metod na osnovu koga se odlučuje u koji red se smešta proces koji treba da se opsluži

Planiranje krajnjeg roka (Deadline scheduling)

Ovakav pristup se bazira na činjenici da real-time aplikacije nisu fokusirane na brzinu, već na izvršenje određenih zadataka u predviđenom roku.

U ovom scenariju, planer dobija informacije o krajnjem roku za svaki od procesa.

Planer će u ovom slučaju obezbediti da će određeni proces krenuti sa izvršavanjem ukoliko je ugrožen krajnji rok njegovog izvršenja, tako što će biti prekinuti procesi koji inače imaju viši prioritet od njega.

Algoritam za planiranje monotonom stopom

Koristi fiksiran prioritet za svaki od procesa baziran na njegovom *periodu izvršavanja.*

Najviši prioritet ima proces sa najkraćim periodom.

Prekida se svaki proces izvršavanjem procesa sa višim prioritetom.

Prednost je jednostavna implementacija, relativno malo “dodatne” implementacije (system overhead).

Mana je činjenica da se zahteva statička prioritizacija pre nego što krene da se izvršava, što nije jednostavno ustanoviti.

EDF (Earliest Deadline First) planiranje

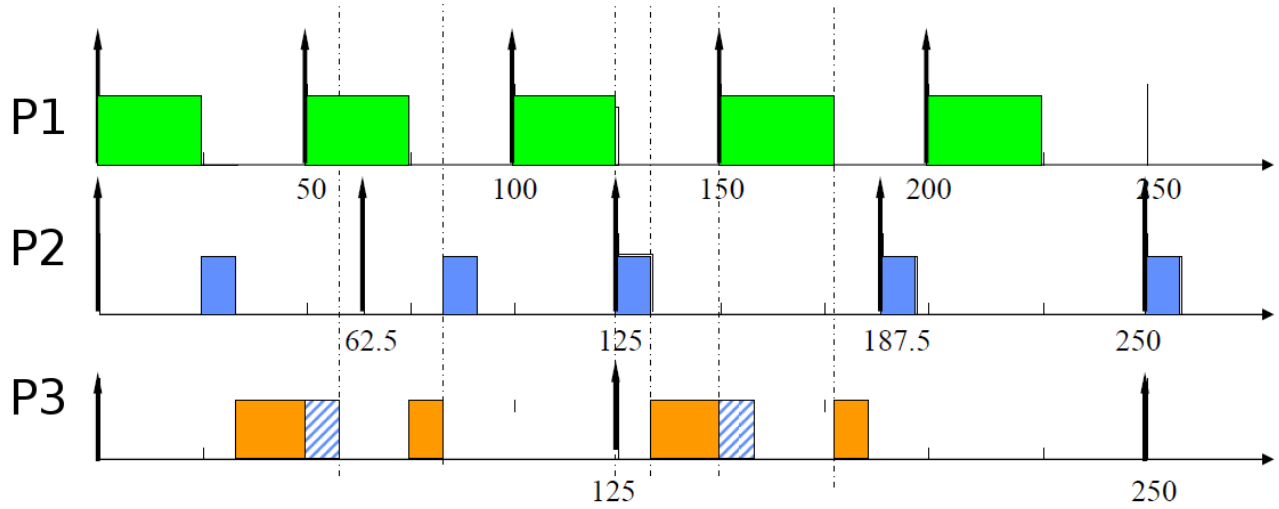
Koristi dinamičko planiranje sa prioritetima.

Najveći prioritet je dodeljen procesu sa najbližim krajnjim rokom, pri čemu se koristi prekidanje izvršavanja ostalih.

Teoretski je superioran u odnosu na algoritam za planiranje sa monotonom stopom i garantuje raspoređivanje za opterećenje procesora od 100% i manje.

Mane su: složenija implementacija, više “dodatne” implementacije, ponašanje preopterećenog sistema je nepredvidivo, ne garantuje se da će procesi višeg prioriteta biti izvršeni pre krajnjeg roka.

Na slici ispod je prikazan primer sa tri procesa (T-perioda procesa, C-trajanje izvršenja)



Slika 6: Primer rada EDF algoritma

LLFT (Least Laxity Time First) planiranje

Koristi dinamičko dodeljivanje prioriteta.

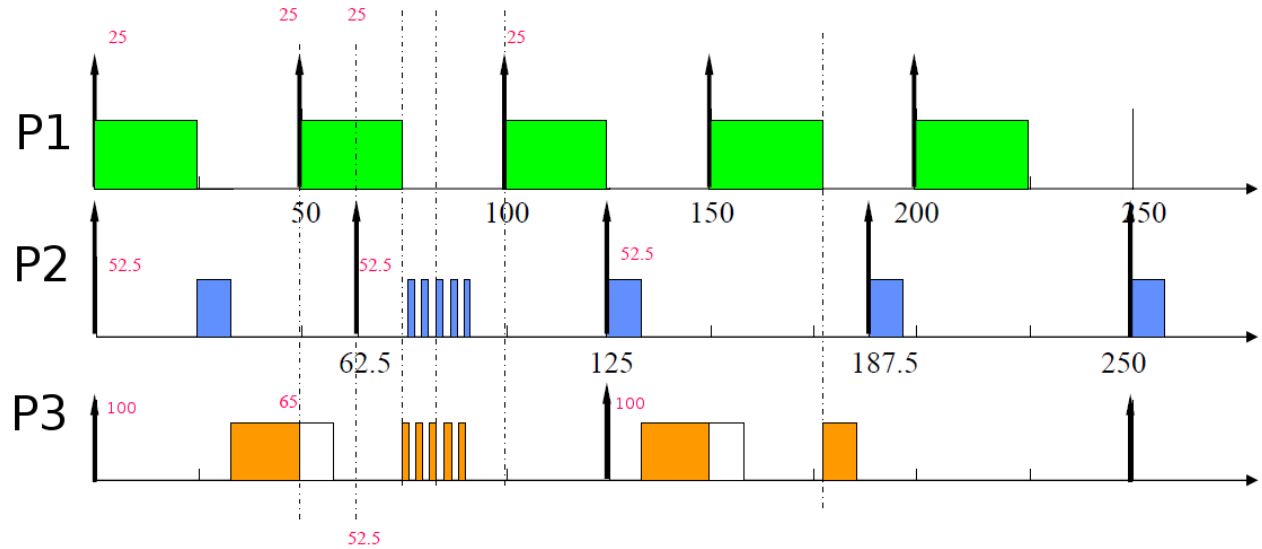
Najviši prioritet ide najmanje **relaksiranom procesu** , pri čemu se relaksiranost procesa L meri sa

L = Td - Tre

Gde je **Td** trenutak isteka krajnjeg roka, a **Tre** preostalo vreme potrebno da bi se proces završio.

Za razliku od prošlog algoritma, uzima u obzir ne samo krajnji rok, već i preostalo vreme potrebno da se proces završi.

Za primer procesa od malopre, dobijamo:



Slika 7: Primer rada LLTF algoritma

MLLF (Modified Least Laxity First) planiranje

LLF algoritam je nepraktičan za implementaciju jer u situacijama kada postoji "jednaka relaksiranost" dva ili više procesa vodi ka čestim zamenama konteksta i, globalno, znatnom opadanju performansi sistema.

MLLF algoritam rešava ovaj problem tako što smanjuje broj zamena konteksta.

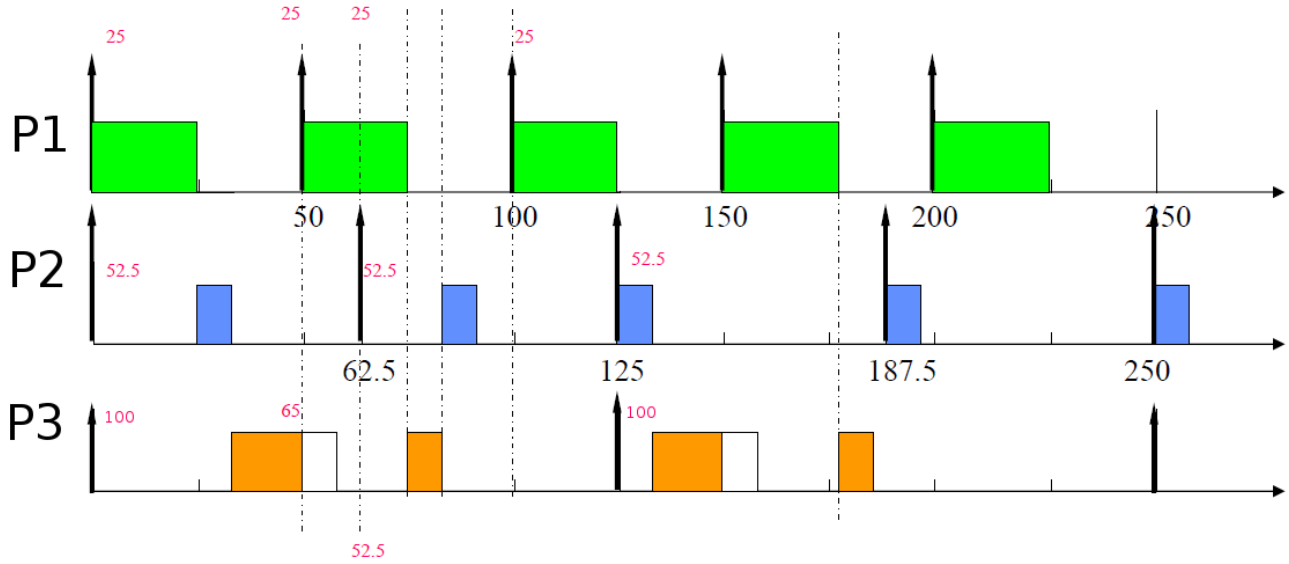
Sve dok nema "jednake relaksiranosti" radi kao LLF.

U trenutku kada se izjednače relaksiranosti dva ili više procesa, proces koji se trenutno izvršava nastavlja da se izvršava sve dok nije ugrožen krajnji rok ostalih procesa.

Na ovaj način MLLF odlaže zamenu konteksta sve dok to nije neophodno i potpuno je bezbedan čak i u situacijama kada nastane "izjednačena relaksiranost" procesa.

Ovaj algoritam dozvoljava ***inverziju relaksiranosti*** , pri kojoj proces sa najmanjom relaksiranošću ne mora trenutno biti odabran za izvršavanje.

Na sledećem slajdu je prikazano kako se ovaj algoritam ponaša u situacijama kada dođe do "izjednačavanja relaksiranosti" procesa.



Slika 8: Primer rada MLLF algoritma