SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

ZAVRŠNI PROJEKT

# Nadogradnja operacijskog sustava FreeRTOS za primjenu u kontrolnim aplikacijama

Luka Jengić

Zagreb, lipanj 2022.

*Umjesto ove stranice umetnite izvornik Vašeg rada.*
*Kako biste uklonili ovu stranicu, obrišite naredbu* `\izvornik`.

*Hvala!*

# SADRŽAJ

# 1. Uvod

Sustavi za rad u stvarnom vremenu (engl. real time operating systems) sustavi su u kojima nije bitan samo rezultat operacije, nego je jednako važna i pravovremenost njezina izvršenja. Najvažnija svojstva koja takvi sustavi moraju zadovoljavati su pouzdanost i predvidivost u izvršavanju zadataka. Zbog navedenog svojstva, sustavi za rad u stvarnom vremenu (u daljnjem tekstu SRSV) neizostavan su dio svih ugradbenih računalnih sustava u kojima postoje kritični poslovi čije neizvršavanje ili prekasno izvršavanje izaziva katastrofalne posljedice ili znatnu degradaciju performansi. Primjeri takvih sustava su automobili i ostala prometna sredstva, vojna industrija, multimedijski sustavi, sustavi automatskog upravljanja, roboti itd.

Ponekad SRSV zbog iznenadne pojave dodatnih zadataka mogu ući u stanje preopterećenja. To je stanje u kojemu procesor ne može izvršiti sve poslove na vrijeme i neke od njih mora preskočiti. Međutim, u nekim primjenama preskakanje zadataka ne smije biti nasumično, nego mora biti kontrolirano i predvidivo. Kod sustava automatskog upravljanja preskakanje kritičnih zadataka može sustav dovesi do nestabilnosti dok u multimedijskim sustavima izaziva smanjenje performansi.

Za primjer možemo uzeti autonomni mobilni robot koji ima sustav za izbjegavanje prepreka. Za siguran rad ključno je uspješno izvođenje zadataka vezanih uz lokalizaciju i kontrolu motora. Ako takav robot uđe u stanje preopterećenja, prioritet moraju dobiti zadatci koji su vezani uz detekciju prepreke i kontrolu motora kako bi se robot na vrijeme zaustavio ukoliko je to potrebno. Ostale funkcije koje robot obavlja manje su kritične te se mogu propustiti bez štete za sustav.

U ovom radu razmatraju se rješenja koja bi osigurala deterministično ponašanje SRSV-a u uvjetima preopterećenja. Što je posao kritičniji, to treba dobiti veći prioritet pri raspoređivanju. Tako se izbjegava šteta koja bi potencijalno nastala propuštanjem takvih zadataka. SRSV-ovi koji su najčešće zastupljeni u industrijskim primjenama obično ne osiguravaju sigurnost sustava kada je on u stanju preopterećenja. Stoga je ideja ovog rada modificirati SRSV otvorenog koda, konkretno FreeRTOS, kako bi se ugradio mehanizam za smanjenje trajnog preopterećenja kontroliranim propuštanjem nekritičnih poslova. Točnije, modifikacijom jezgre FreeRTOS-a implementiran je strogi sustav za rad u stvarnom vremenu s ublaženim uvjetima (engl. weakly hard real time system). Kod takvog sustava povremeno se dopušta da se posao

ne izvede, ali u određenom broju slijedno aktiviranih poslova postoji striktno definirana donja granica na broj poslova koji se moraju pravovremeno izvesti. Takvim pristupom preskakanja poslova osigurava se da se niti jedan zadatak trajno ne blokira. Nadalje implementirana je strategija prekidanja zadataka, prema kojoj se zadatci ne nastavljaju izvršavati nakon što su došli do krajnjeg roka za izvršavanje.

U literaturi se može pronaći mnogo različitih algoritama za raspoređivanje zadataka, s obzirom na optimalnost, složenost i model zadataka na koji se odnose. Neki od njih implementirani su u raspoređivač zadataka FreeRTOS-a te međusobno uspoređivani nad istim skupovima zadataka. Cilj je pronaći algoritam koji će u uvjetima preopterećenja osigurati deterministično ponašanje uz što veću kvalitetu usluge, to jest uz što više pravovremeno izvršenih poslova.

# 2. Operacijski sustavi za rad u stvarnom vremenu

## 2.1. Sustavi za rad u stvarnom vremenu

Sustavi za rad u stvarnom vremenu (engl. real time system) danas su neizostavan dio mnogih sustava korištenih u svim granama ljudske djelatnosti. Kod njih nije bitan samo rezultat izvođenja operacije, nego je jednako važno i vrijeme u kojem se ta operacija izvede. Zbog toga se svakom zadatku pridjeljuje krajnji rok do kojeg se mora izvršiti. Izvršenje posla nakon njegovog krajnjeg roka za izvršavanje ekvivalentno je tome da se posao nije niti izvršio.

Kako bi sustav radio pouzdano, mora se osigurati predvidivo raspoređivanje zadataka koje će osigurati da se što više zadataka izvrši na vrijeme. SRSV mora imati implementiranu kontrolu zadataka, kontrolu vremena, raspoređivač zadataka i sustav komunikacije i sinkronizacije među zadatcima.

S obzirom na posljedice koje izaziva propuštanje roka izvršavanja zadatke dijelimo u dvije skupine :

- strogi zadatci (eng. hard real time tasks),
- opcionalni zadatci (eng. soft real time tasks).

Strogi zadatci niti jednom ne smiju propustiti krajnji rok izvršavanja jer su to zadatci koji obavljaju važan posao. Njihovo propuštanje izaziva katastrofalne posljedice koje mogu biti pogubne za cijeli sustav i njegovu okolinu. Dobar primjer strogog zadatka je detekcija pritiska papučice kočnice na automobilu. Taj zadatak je kritičan i propuštanje njegovog izvršenja za posljedicu može imati ozbiljnu nesreću s ljudskim žrtvama. S druge strane, propuštanje nekih zadataka nije kritično i neće nanijeti štetu sustavu, nego će smanjiti performanse ili mogućnosti sustava. Primjer ovakvog zadatka je paljenje signalne lampice ili prikaz podataka na pokazniku.

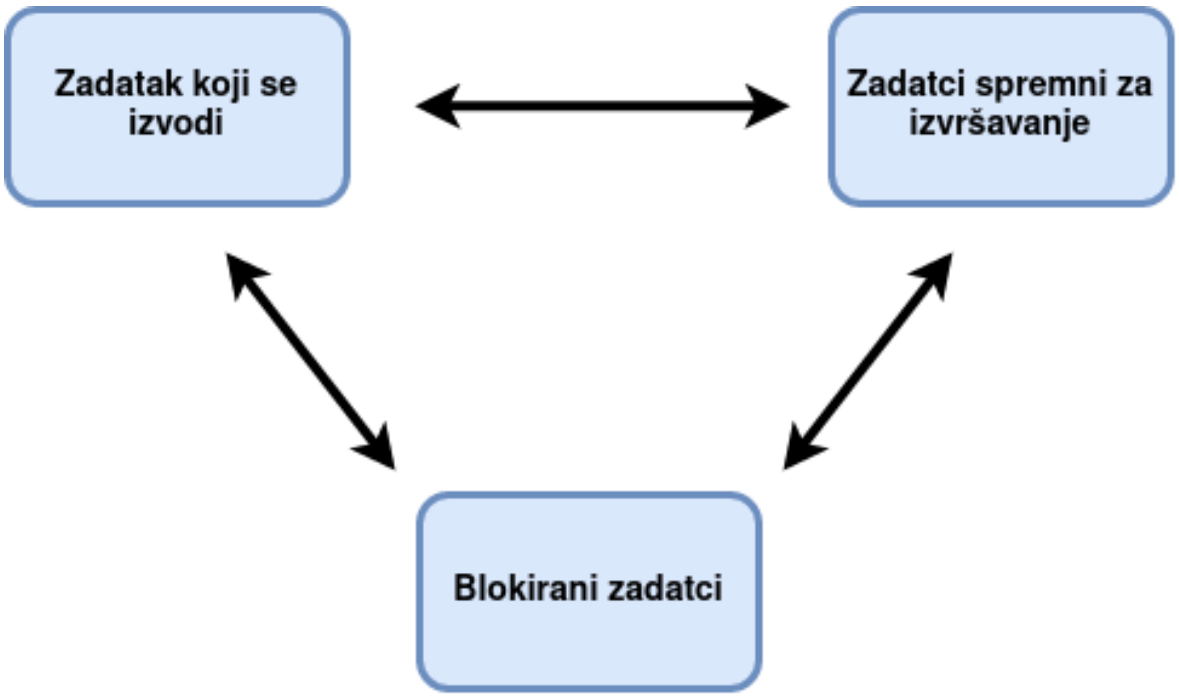
Zadatci u operacijskim sustavima za rad u stvarnom vremenu općenito se mogu naći u jednom od 3 stanja, a to su:

- zadatak koji se trenutno izvodi (engl. running task),

– zadatci koji su spremi za izvođenje (engl. ready tasks),

– zadatci koji su blokirani i čekaju određeni događaj (engl. blocked tasks).

Kod sustava s jednom procesorskom jezgrom samo, jedan zadatak se u određenom trenutku može izvoditi. Zadatci koji su spremni za izvođenje čekaju oslobođenje procesora i jedan od njih se odabire za izvršavanje. Dio jezgre SRSV-a koji je zadužen za izbor zadatka koji je spreman za izvođenje i koji će se idući izvršavati naziva se raspoređivač zadataka (engl. task scheduler). To je najvažniji dio jezgre SRSV-a jer o raspoređivanju zadataka ovisi kolika će biti kvaliteta usluge i koliko pouzdan će biti cijeli sustav. Neki zadatci mogu biti privremeno ili trajno blokirani i dok su u tom stanju raspoređivač zadataka ih ne uzima u obzir. Zadatci se tijekom rada prebacuju između opisanih stanja, no treba napomenuti da je ovo generalna podjela koja se razlikuje u implementaciji konkretnih SRSV-ova.




**Slika 2.1:** Stanja zadataka u SRSV-u

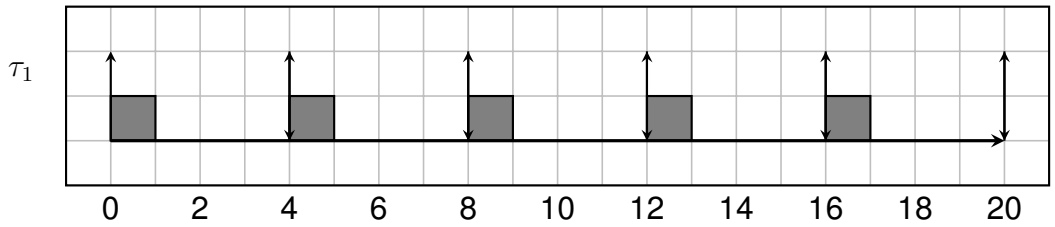
## 2.2. Periodični zadatci u sustavima za rad u stvarnom vremenu

Periodični zadatci su zadatci koji se iznova ponavljaju u istim vremenskim intervalima $T_i$. Taj interval naziva se period zadatka. Dio zadatka izvršen u jednom periodu naziva se posao. Svaki zadatak opisuje i vrijeme njegova izvršavanja $C_i$ (engl. computation time). Period i vrijeme izvršavanja zajedno daju veličinu koju nazivamo faktor opterećenja (engl. utilization factor) koja daje postotak zauzeća procesora pojedinog zadatka u jednom periodu. S obzirom da se poslovi ponavljaju jedan za drugim, to je ujedno i veličina koja govori koliko

procesorskog vremena se ukupno troši na taj zadatak.

$$U_{\mathrm{i}} = \frac{C_{\mathrm{i}}}{T_{\mathrm{i}}}$$

Nadalje, bitna veličina koja opisuje svaki zadatak je krajnji rok njegova završetka (engl. deadline). Krajnji rok izvršavanja je vrijeme do kojeg se posao mora izvršiti kako bi donio korist sustavu. U ovom radu razmatrani su isključivo zadatci čiji je krajnji rok završetka jednak periodu. Takav krajnji rok naziva se implicitni krajnji rok završetka. Kao primjer dan je vremenski dijagram s jednim periodičnim zadatkom. Period zadatka iznosi 4 vremenske jedinice, a vrijeme izvršavanja pojedinog posla 1 vremensku jedinicu. Pomoću ta dva podatka i ranije dane formule proizlazi da je faktor opterećenja zadatka 0.25. Drugim riječima, ovaj zadatak zauzima 25 % ukupnog procesorskog vremena.




**Slika 2.2:** Primjer periodičnog zadatka

Ukupno opterećenje sustava dobiva se kao zbroj faktora opterećenja svih zadataka. Ako je sustav preopterećen, tj. ukupno opterećenje mu je veće od 1, neki od zadataka se neće izvršiti do svog roka za izvršavanje. U tom slučaju koriste se algoritmi koji će osigurati da se zadatci ne preskaču nasumično, već na kontroliran način kao bi se sustav zaštitio od potencijalnih oštećenja nastalih preskakanjem kritičnih zadataka. Postoje dvije vrste preopterećenja sustava.

- trajno preopterećenje (eng. Permanent Overload),
- prolazno preopterećenje (eng. Transient Overload).

Kod prolaznog opterećenja sustava ima ukupno opterećenje manje ili jednako 1, no u nekom trenutku može doći do aktivacije aperiodičnog zadatka koji onda sustav gura u stanje preopterećenja. Nakon što prođe određeno vrijeme i aperiodični zadatak se izvede, sustav se vraća u prijašnje stanje i svi zadatci se izvršavaju pravovremeno. Drugi slučaj je kada je sustav u stanju trajnog preopterećenja, kada je ukupno opterećenje sustava trajno veće od 1. Tada se svi poslovi neće moći izvršiti i kontinuirano će se pojedini poslovi morati propuštati.

U ovom radu ispitivat će se ponašanje sustava s ublaženo-strogim uvjetima (eng. weakly hard constraints). Za razliku od kritičnih zadataka, kod ublaženo-kritičnih zadataka povremeno dopuštamo da se zadatak ne izvede na vrijeme, ali na predvidiv i kontroliran način. Na temelju važnosti pojedinog zadatka za rad cjelokupnog sustava određuje se u koliko slijednih

| Broj Zadatka | Period | Vrijeme izvršavanja | Faktor opterećenja |
|---|---|---|---|
| 1 | 6 | 2 | izracunaj |
| 2 | 8 | 2 | izracunaj |
| 3 | 4 | 2 | izracunaj |

**Tablica 2.1:** Skup zadataka korišten u primjerima.

perioda se zadatak može jednom propustiti. Ovaj pristup osigurava znatno bolje ponašanje u uvjetima preopterećenja jer se propuštanje pojedinih poslova odvija deterministički.

## 2.3. Algoritmi za raspoređivanje zadataka

U ovom radu ispitivat će se različiti algoritmi za raspoređivanje zadataka u sustavu koji se nalazi u stanju trajnog preopterećenja. U navedenom slučaju važno je osigurati preskakanje zadataka na predvidiv i za sustav siguran način.

Mjera kojom će se uspoređivati učinkovitost pojedinih algoritama naziva se kvaliteta usluge (eng. Quality of Service). Računa se kao omjer broj zadataka koji su se izvršili do krajnjeg roka završetka i ukupnog broja svih zadataka.

$$QoS = \frac{broj\ izvrenih\ poslova\ do\ krajnjeg\ roka}{ukupan\ broj\ poslova}$$

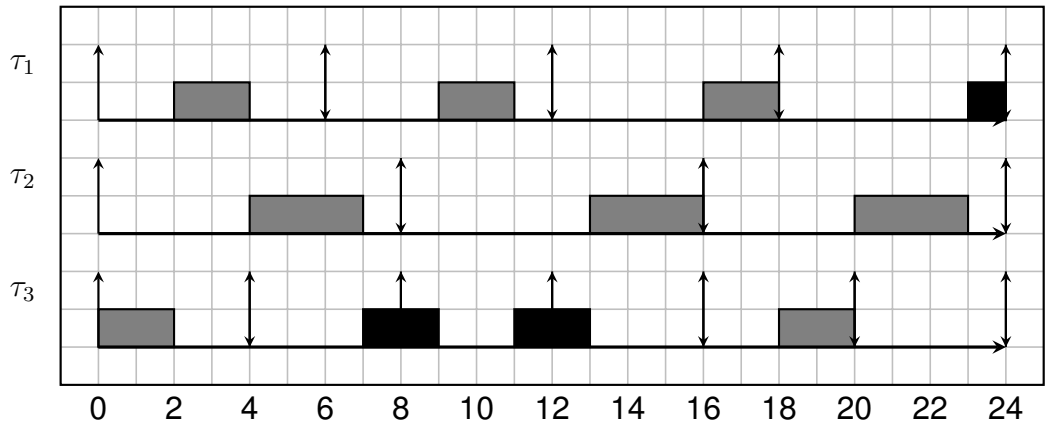
U nastavku su opisani korišteni algoritmi za raspoređivanje zadataka. Za svaki algoritam dan je primjer generiranog rasporeda na jednostavnom primjeru skupa zadataka koji je dan u tablici 2.1. Raspored se prikazuje vremenskim dijagramom u kojem je vidljivo koji zadatak se u danom trenutku izvršava. Za dani set zadataka zadano je da je prvi zadatak kritičan i mora se uvijek izvršiti, drugi zadatak ne utječe na sigurnost sustava, a treći zadatak mora se izvršiti svaki drugi put.

Dani set zadataka nalazi se u preopterećenju, i neki od zadataka neće zadovoljiti svoj krajnji rok završetka. Ukupni faktor opterećenja sustava je 1.08.

### 2.3.1. EDF algoritam

Algoritam EDF (engl. earliest deadline first) je algoritam koji pri raspoređivanju zadataka najveći prioritet daje onim poslovima koji imaju najbliži krajnji rok završetka. Ukoliko sustav nije preopterećen (ako je ukupni faktor opterećenja manji ili jednak od 1) ovim algoritmom optimalno će se rasporediti zadatci i svi će se izvršiti. Ovaj algoritam nije pogodan za primjenu sa ublaženo strogim uvjetima, ali u ovom radu je istražen jer se u praksi

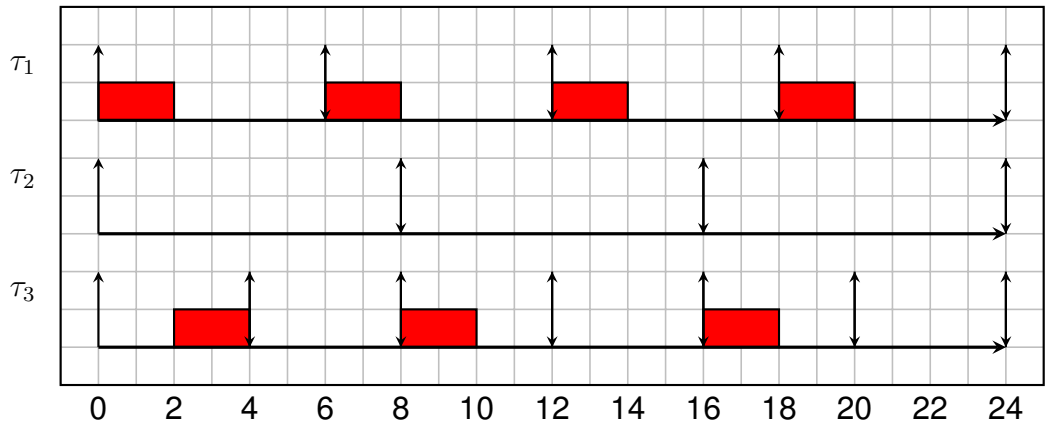
često koristi. Raspored generiran EDF algoritmom prikazan je na slici 2.3. Na vremenskom dijagramu crnom bojom su prikazani poslovi koji su propustili svoj krajnji rok izvršenja.




**Slika 2.3:** Raspored zadataka EDF algoritmom

### 2.3.2. RTO algoritam

Algoritam RTO (engl. red tasks only) je algoritam prema kojemu se samo izvršavaju strogi zadatci, dok se oni koji se mogu preskočiti uvijek preskaču. Prema ovom algoritmu osigurano je poštivanje zadanih uvjeta, no pri manjim faktorima opterećenja ovaj algoritam nije optimalan. Razlog tomu je to što postoji slobodno procesorsko vrijeme u kojem bi se mogli izvršiti zadatci koje nije nužno izvršiti, no oni se automatski izbacuju iz rasporeda. Kritični zadatci raspoređuju se prema ranije opisanom EDF algoritmu.

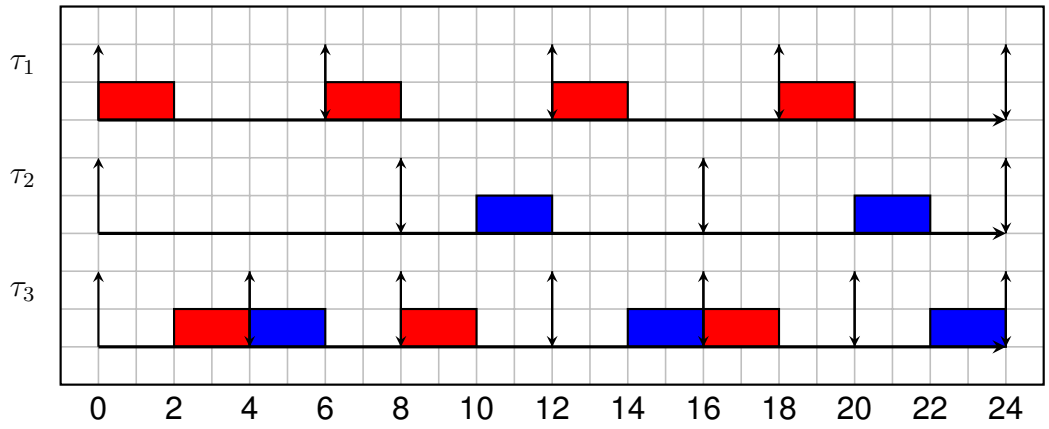



**Slika 2.4:** Raspored zadataka RTO algoritmom

### 2.3.3. BWP algoritam

Algoritam BWP (engl. blue when possible) je poboljšanje ranije opisanog RTO algoritma. Kod BWP algoritma prioritet imaju strogi zadatci, no raspoređuju se i zadatci koji se ne

moraju nužno izvesti. Na taj način, ukoliko se svi kritični zadatci izvrše, na red će doći i opcionalni zadatci. Ovom modifikacijom znatno se poboljša kvaliteta usluge (engl. quality of service), pogotovo pri manjim faktorima opterećenja. Zadatci su u dijagramu obojani prema nazivu algoritma, crvena boja za stroge zadatke, a plava za zadatke čije izvršavanje je opcionalno.




**Slika 2.5:** Raspored zadataka BWP algoritmom

# 3. Modifikacija jezgre FreeRTOS-a

## 3.1. Operacijski sustav FreeRTOS

FreeRTOS je operacijski sustav za rad u stvarnom vremenu otvorenog koda (eng. open source) namjenjen primjeni u ugradbenim računalnim sustavima. U ovom potpoglavlju bit će objašnjena njegova implmenetacija i izvedba raspoređivanja zadataka. Kod FreeRTOS-a organiziran je u nekoliko datoteka i zauzima svega nekoliko kilobajta. Datoteka u kojoj je implementirano upravljanje zadatcima i njihovim raspoređivanjem je `tasks.c`.

Zadatci u FreeRTOS-u su opisani strukturom za upravljanje zadatcima (eng. TCB - Task Control Block). U toj strukturi nalaze se sve informacije koje opisuju pojedini zadatak (ime zadatka, prioritet,broj zadatka...). Zadati se stvaraju funkcijom `xTaskCreate()`. Pri stvaranju novog zadatka u alocira se prostor u memoriji za novu inačicu ove strukture u koju se potom upisuju parametri novokreiranog zadatka.

Zadatci su smješteni u različite liste, ovisno o tome u kojem stanju se zadatak nalazi. Postoje liste za zadatke koji su spremni za izvršavanje, za zadatke koji su blokirani te za zadatke koji čekaju određeno vrijeme kako bi se ponovo mogli izvršavati. U TCB-u svakog zadatka nalazi se informacija kojoj listi zadatak u danom trenutku pripada. Upravljanje listama zadataka ostvareno je strukturama `ListItem_t` i `List_t`. `ListItem_t` je struktura koja sadrži podatke o pojedinom članu liste (pokazivači na prethodni i sljedeći `ListItem_t` te pokazivač na TCB zadatka na kojeg se odnosi). U `List_t` spremljeni su podatci o listi (broj elemenata, pokazivač na trenutni element te pokazivač na kraj liste).

Informacija o tome koji zadatak se trenutno izvršava pohranjena je u pokazivaču na TCB strukturu `pxCurrentTCB`. Zadatci koji su spremni za izvršavanje podijeljeni su u različite liste ovisno o tome kojeg su prioriteta (postoji posebna lista za sve razine prioriteta zadataka). Stoga se pri raspoređivanju zadataka naprije pronađe lista najvišeg prioriteta koja nije prazna. Zadatci iz te liste dijele procesorsko vrijeme na način da se svakom zadatku iz liste dodjeljuje jedan vremenski odsječak procesorskog vremena (eng. tick). Nakon što se zadatak izvede u jednom odsječku, vraća se na kraj liste čekanja. Ovaj način raspoređivanja zadataka naziva se dijeljenje procesorskog vremena među zadatcima (eng. Time Slicing).

FreeRTOS se konfigurira putem datoteke `FreeRTOSconfig.h` i korisnik treba mijenjati

samo tu datoteku. U toj datoteci nalaze se sve konstante preko kojih se uključuju pojedine funkcionalnosti ili postavljaju konstante bitne za rad sustva.

Podrška za upravljanje zadatcima u FreeRTOS-u ne podržava definiranje i kontrolu periodičkih zadataka, stoga je potrebno proširiti podsustav za upravljanje zadatvima u FreeRTOS-u. Također navedni pristup ne rješava problem preopterećenja sustava te je potrebno modificirati jezgru FreeRTOS-a kako bi se omogučilo predvidivo ponašanje ukoliko sustav uđe u trajno preopterećenje.Potrebne modifikacije detaljno su opisane u narednim podglavljima.

dodaj za xtaskgetticks() !!!!!!!!!!!!!!

## 3.2. Programska potpora za kontrolu izvršavanja periodičnih zadataka

Uključenje funkcionalnosti za upravljanje periodičnim zadatcima ostvareno je putem konstante `configUSE_PERIODIC_TASK` dodane unutar datoteke `FreeRTOSconfig.h`. Korisnik postavljanjem navedene konstante u 1 na brz i jednostavan način uključuje podršku za periodične zadatke. Svi djelovi programskog koda zaduženi za periodične zadatke pisani su u odsječcima koji se uključuju u proces prevođenja samo ako je vrijednost `configUSE_PERIODIC_TASK` jednaka 1. Pritom je korištena pretprocesorska naredba #if.

```
#if ( configUSE_PERIODIC_TASK == 1 )

#endif
```

**Isječak koda 3.1:** Pretprocesorska naredba za uključenje periodičnih zadataka

### 3.2.1. Stvaranje periodičnih zadataka

Prvi korak pri implementaciji programske podrške za izvršavanje periodičnih zadataka je proširenje TCB-a veličinama koje opisuju periodičan zadatak. Ovdje su definirane sve veličine bitne za kontrolu periodičkih zadataka koje su opisane u ranijim poglavljima.

```
#if ( configUSE_PERIODIC_TASK == 1 )
  // variables used for periodic task control
  uint8_t xTaskId;
  TickType_t xTaskPeriod;
  TickType_t start_time;
  TickType_t xTaskDuration;
  TickType_t xDeadline;
  TickType_t xRemainingTicks;
#endif
```

**Isječak koda 3.2:** Varijable dodane u strukturu za kontrolu zadataka

Nadalje, napisana je funkcija `xTaskCreatePeriodic()` koja se koristi za stvaranje i inicijalizaciju periodičnih zadataka. Navedena funkcija je proširenje `funkcije xTaskCreate()` s novim varijablama potrebnim za opis periodičnih zadataka.

```
BaseType_t xTaskCreatePeriodic( TaskFunction_t pxTaskCode,
                        uint8_t id,
                        const char * const pcName,
                        const configSTACK_DEPTH_TYPE usStackDepth,
                        void * const pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t * const pxCreatedTask,
                        TickType_t period,
                        TickType_t duration,
                        int weakly_hard_constraint);
```

**Isječak koda 3.3:** Prototip funkcije xTaskCreatePeriodic

### 3.2.2. Kontrola izvršenja priodičnih zadataka

Izvršavanje zadataka u SRSV-ovima podjeljeno je u vremenske odsječke. U određenim vremenskim intervalima (eng. ticks) prekida se izvođenje poslova i određuje se koji posao se treba dalje izvršavati. U FreeRTOS-u navedena funkcionalnost implementrana je u `xTaskIncrementTic` funkciji. Prekidni sustav periodički poziva navedenu funkciju i u njoj je potrebno dodati funkcionalnost kontrole periodičnih zadataka.

Kontrola izvođenja periodičkih zadataka realizirana je pomoću varijable `xRemainingTicks` koja u svakom trenutku pamti koliko je vremenskih odsječaka ostalo do potpunog izvršenja posla. Svaki vremenski odsječak u kojem se posao izvršava ta varijabla se umanjuje za 1. Ukoliko se vrijednost `xRemainingTicks` smanji na 0, posao je gotov i zabilježava se njegovo pravovremeno izvršavanje. Svaki puta kada se posao kreće izvršavati, varijabla `xRemainingTicks` postavlja se na vrijeme njegovog izvršavanja.

Za potrebe ovog projekta zadatci su spremani u dvije liste. Jedna lista u kojoj se čuvaju zadatci spremni za izvršavanje, ranije implementirana u FreeRTOS-u i novododana lista nazvana `xWaitTaskList` u kojoj su zadatci koji su na čekanju. Za dodavanje zadataka u liste napisane su dvije funkcije, za jedna po svakoj listi.

```
void addTaskToReadyList(TCB_t * const pxItemToAdd);
void addTaskToWaitList(TCB_t * const pxItemToRemove);
```

**Isječak koda 3.4:** Deklaracije funkcija za dodavanje zadataka u opisane liste

Vraćanje zadataka iz liste za čekanje u listu zadataka spremnih za izvršavanje implementirano je u funkciji `wakeTasks()`. U njoj se petljom iterira po svim zadatcima koji su u stanju čekanja i za svakog se provjerava treba li ga prebaciti u listu zadataka spremih za izvršavanje.

| Broj Zadatka | Period | Vrijeme izvršavanja | Faktor opterećenja |
|---|---|---|---|
| 1 | 6 | 1 | 0.17 |
| 2 | 8 | 6 | 0.75 |
| 3 | 4 | 2 | 0.50 |

**Tablica 3.1:** Skup zadataka korišten u opisu strategije prekidanja poslova.

Ukoliko je zadatak u listi za čekanje i ukoliko se program nalazi na višekratniku njegova perioda, zadatak se dodaju u listu zadataka spremih za izvršavanje. Pri tome je potrebno u varijablu `start_time` upisati trenutno vrijeme (trenutak u kojemu je zadatak posao spreman za izvršavanje).Također moramo osvježiti vrijednost varijable texttttxRemainingTicks. Vrijeme početka izvršavanja je važno za provjeru treba li zadatak prekinuti i je li se izvršio do krajnjeg roka završetka što će detaljnije biti objašnjeno u tesktu koji slijedi.

## 3.3. Strategija prekidanja poslova

Ponekad posao dođe na red za ivršavanje, ali čak i ako dobije svo procesorsko vrijeme ne stigne se izvršiti do krajnjeg roka za izvršavanje. kako kod strogih zadataka nema koristi od posla koji se djelomično izvršio, ukoliko posao dođe do krajnjeg roka za izvršavanje, a nije se izvršio treba prekinuti njegovo daljnje izvršavanje. Ova strategija zove se prekidanje poslova koji se ne mogu izvršiti do svog krajnjeg roka izvršenja (engl. job killing). Time se podiže kvaliteta usluge, jer je osigurano da poslovi koji se nikako neće izvršiti na vrijeme ne zauzimaju proesorsko vrijeme ostalim zadatcima. Strategiju prekidanja poslova dodatno možemo poboljšati ukoliko unaprijed prolazimo po svim spremnim poslovima i provjeravamo mogu li se izvršiti do roka ako im je na raspolaganju svo procesorsko vrijeme od datog trenutka. Ako se posao nebi stigao obaviti treba ga odmah prebaciti ga u listu za čekanje i detektirati propuštanje roka izvršavanja. Uz opisani pristup zadatci koji se ne mogu izvesti neće se niti kretati izvoditi, nego će samo biti detektirano njihovo propuštanje čime se dodatno poboljšava rasporedivost zadataka.

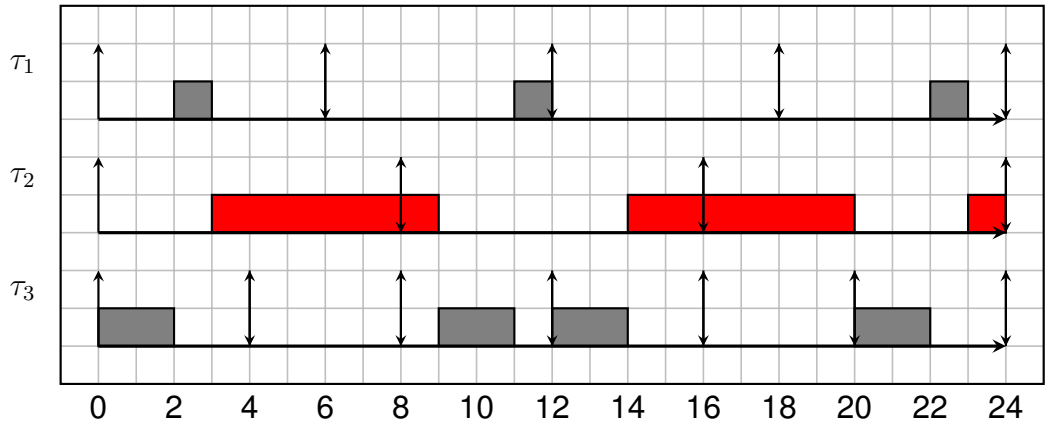
Opisana situacija prikazana je na vremenskim dijagramima na slikama 3.1 i 3.2. U primjeru imamo 3 zadatka jednakih prioriteta. Radi jednostavnosti je pretpostavljeno da su svi zadatci jednako kritični za sustav i da je korišten EDF algoritam za raspoređivanje zadataka. Periodi zadataka i vremena njihova izvršavanja dani su u tablici 3.1. Sustav je u stanju trajnog preopterećenja sa ukupnim faktorom opterećenja 1.42.

Treba primjetiti da je bez korištenja strategije prekidanja poslova zadatak $tau_2$ potrošio 13 vremenskih odječka procesorskog vremena, a na pritom se nije uspio niti jednom izvršiti

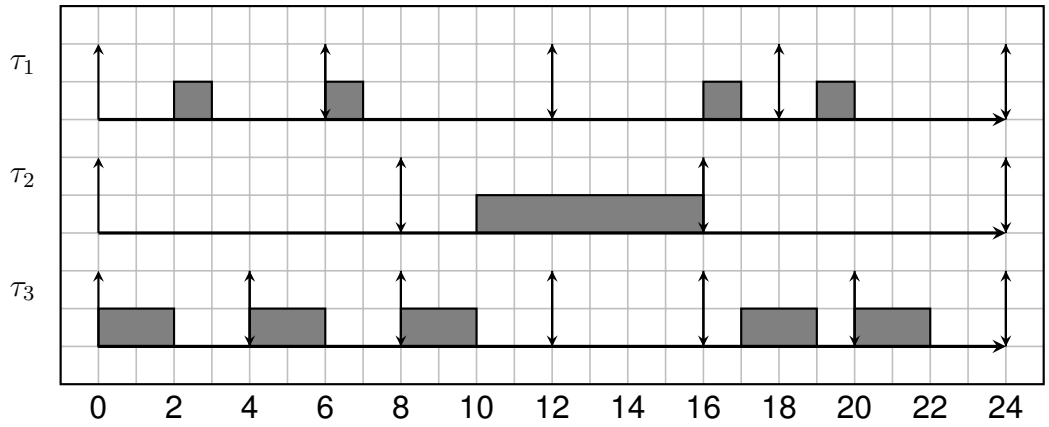
na vrijeme. Predložena modificirana strategija prekidanja poslova uvodi provjeru svaki vremenski odječak može li se zadatak izvesti do krajnjeg roka za izvršavanje. U navedenom primjeru, u trenutku t = 4 sustav bi obavio provjeru može li se zadatak broj 2 izvršiti na vrijeme, zaključio bi da ne može i nebi ga niti krenio izvoditi. Tada se oslobađs procesorsko vrijeme za zadatke 1 i 3 i raspored poslova izgleda kao na dijagramu 3.2.

Kako bi vidjeli kakav utjecaj na sustav ima prekidanje poslova za oba slučaja potrebno je izračunati kvalitetu usluge. Bez korištenja strategije prekidanja poslova izvršilo se 7 od ukupno 13 poslova. Kvaliteta usluge u tom slučaju je 0.54. Uz primjenu opisanog poboljšanja uspješno se izvelo 10 poslova, što daje kvalitetu usluge 0.67. Iz prikazanog primjera zaključujemo da korištenje prekidanja poslova u uvjetima preopterećenog sustava znatno podiže kvalitetu usluge i poboljšava rad sustava.

Važna napomena uz navedeni primjer je da služi samo za objašnjenje strategije prekidanja poslova te je stoga pretpostavljeno da su svi zadatci jednako kritični.




**Slika 3.1:** Raspored zadataka bez prekidanja poslova




**Slika 3.2:** Raspored zadataka sa primjenjnim prekidanjem poslova

Funkcionalnost prekidanja poslova implementirana je u funkciji `killTasks()`. U njoj u svakom vremenskom osdječku procesorskog vremena iteriramo po svim zadatcima koji

se nalaze u listi zadataka spremnih za izvršavanje i na temelju podataka o trenutnom stanju posla odlučujemo treba li ga prekinuti ili ne. Uvjet za prekid dobije se tako da usporedimo vrijeme u kojem bi posao bio obavljen sa krajnjim rokom za izvršavanje. Vremena su uspoređivana relativno u odnosu na vrijeme kada je posao postao spreman za izvršavanje (višekratnik perida). U nastavku je priložen navedeni uvjet.

```
if((xTaskGetTickCount() - TaskTcb->start_time + TaskTcb->xRemainingTicks)
    >
TaskTcb->xTaskDeadline)
```

**Isječak koda 3.5:** Uvjet za prekidanje izvođenja zadatka

## 3.4. Strogi sustav za rad u stvarnom vremenu s ublaženim uvjetima

Kod strogog SRSV-a s ublaženim uvjetima na zadatke postavimo uvjet da se smije propustiti jedan posao u nekoliko slijednih perioda. Navedeno je implementirano pomoću dvije cjelobrojne varijable dodane u TCB strukturu zadatka.

```
#if ( configUSE_PERIODIC_TASK == 1 )
  // variables used for weakly hard conditions control
  uint8_t weakly_hard_constraint;
  uint8_t previous_deadline_met;
#endif
```

**Isječak koda 3.6:** Varijable dodane u strukturu za kontrolu zadataka

Varijablom `weakly_hard_constraint` zadano je u koliko slijednih perioda se jedan posao smije propustiti. Na primjer, ukoliko je zadana vrijednost ublaženo-strogog zadatka 5, to znači da se jedan posao smije propustiti unutar pet perioda koji slijede jedan za drugim. Poseban slučaj su vrijednosti 0 i 1. Vrijednost 0 zadavat će se za najkritičnije zadatke koji se nikada ne smiju propustiti. S druge strane, vrijednost 1 daje se manje bitnim zadatcima koji se uvijek mogu propustiti i čije neizvršavanje nije pogubno za sustav. Postavljanjem ove vrijednosti točno se zadaje kako se poslovi mogu propuštati, a bez štete za sustav. Vrijednosti se određuju temeljem kritičnosti zadataka i posljedica koje nose propuštanja poslova.

Kako bi sustav u svakom trenuku znao smije li propustiti neki posao potrebna je još jedna varijabla. To je varijabla `previous_deadline_met` u kojoj je upisan broj poslova koji su se prethodno izvršili do svog krajnjeg roka za izvršavanje. Svaki puta kada se posao izvrši na vrijeme ova varijabla se povečava za jedan, a kada se proputi izvršavanje posla ili ga se prekine tada se vrijednost varijable postavlja na 0.

Poznavajući vrijednosti ovih dviju varijabli u svakom trenutku sustav zna mora li se zadatak izvršiti ili ne. Posao se mora izvršiti ako se prethodno zadatak izvršio manje puta od vrijednosti `weakly_hard_constraint` umanjene za 1. Ovaj uvjet bit će korišten prilikom implementacije raspoređivanja zadataka.

```
1 if( TaskTcb->previous_deadline_met < (TaskTcb->weakly_hard_constraint-1))
```

**Isječak koda 3.7:** Uvjet za slučaj kada se zadatak mora izvršiti

Zbog usporedbe različitih algoritama, na početku simulacije (pri stvaranju zadataka) će varijabla `previous_deadline_met` biti postavljen na 0, kako bi dobili najgori mogući slučaj i kako bi usporedbe bile valjane.

## 3.5. Implementacija algoritama za raspoređivanje zadataka

Algoritmi za raspoređivanje zadataka podrazumijevaju logiku kojom se odabire koji zadatak će se idući poslati na izvršavanje. To je najvažniji dio SRSV-a ... DODAJ !!

Liste implementirane u FreeRTOS-u imaju mogućnost sortiranja po vrijednosti varijable `xItemValue` koju sadrži svaki član liste (u `ListItem_t` strukturi). To svojstvo je iskorišteno kako bi zadatke koji su spremni za izvršavanje poredali po željenom redosljedu. Element s vrha liste uvijek će biti onaj sa najmanjom vrijednosti `xItemValue` i time će imati najveći prioritet pri izvršavanju. Vrijednost varijable `xItemValue` mijenja se putem macro-a `listSET_LIST_ITEM_VALUE()` implementiranog u FreeRTOS-u od ranije.

Prvi korak u modifikaciji raspoređivača zadataka FreeRTOS-a bio je promjena macro-a `taskSELECT_HIGHEST_PRIORITY_TASK()`. U njemu se određuje koji zadatak će se idući poslati na izvršavanje iz liste koja trenutno sadrži zadatke najvećeg prioriteta. Prije izmjene zadatci su dijelili procesorsko vrijeme, svaki zadatak po jedan vremenski odsječak. To je bilo realizirano pozivom macro-a `listGET_OWNER_OF_NEXT_ENTRY` koji je iterirao po svim članovima listi. Umjesto toga, potrebno je svaki vremenski odsječak dohvatiti prvi element liste. Za to u FreeRTOS-u postoji macro `listGET_OWNER_OF_HEAD_ENTRY` koji vrati pokazivač na prvi element liste. To će biti zadatak koji se treba izvršavati jer je lista sortitana u uzlaznom poretku. U svakom vremenskom odsječku u varijablu `pxCurrentTCB` treba pohraniti pokazivač na prvi element liste. Time je osigurano da se uvijek izvodi zadatak najvećeg prioriteta, skroz dok se ne izvede, dok ga ne prekine neki kritičniji zadatak ili dok ga sustav ne prekine strategijom prekidanja poslova.

Nadalje je pri stavljanju zadatka u listu zadataka spremih za izvršavanje potrebno generirati i upisati vrijednost `xItemValue` prema kojoj će se zadatci sortirati. To je implementirano u macro funkciji `prvAddTaskToReadyList`. Za svaki algoritam potrebno je imati posebno implementiranu funkciju `prvAddTaskToReadyList`. To je ostvareno preko konstanti u

datoteci FreeRTOSconfig.h. Za svaki algoritam definirana je jedna konstanta, i onaj algoritam kojem je vrijednost konstante 1 se koristi. Prije pokretanja simulacije korisnik treba odabrati koji algoritam se koristi, vrijednost njegove konstante postaviti u 1, a svih ostalih u 0. Time je omogućeno da se macro `prvAddTaskToReadyList` napiše za svaki algoritam zasebno, ali unutar pretprocesorske naredbe #if, te će na kraju samo jedna verzija biti uključena u program.

```
#define configEDF_ALGORITHM 0
#define configRTO_ALGORITHM 0
#define configBWP_ALGORITHM 1
```

**Isječak koda 3.8:** Primjer uključenja algoritma BWP

### 3.5.1. Implementacija EDF algoritma

Kako je ranije opisano, EDF algoritam sortira zadatke prema vremenu krajnjeg roka za izvršavanje. Pri stavljanju zadatka u listu za spremne zadatke, potrebno je izračunati apsolutno vrijeme roka za izvršenje posla. Pošto su nam rokovi za izvršavanje uvijek višekratnici perioda, zapravo trebamo izračunati vrijeme sljedećeg perioda. To radimo tako da od trenutnog vremena oduzmemo vrijeme proteklo od početka perioda (time smo dobili početak perioda) te još nadodamo jedan period. Kod EDF algoritma u obzir ne uzimamo ublažene uvjete kod strogog SRSV-a, nego sve poslove promatramo jednako važnima.

ZAMJENI OVO SA NEKIM PSEUDO KODOM !!!

```
listSET_LIST_ITEM_VALUE(&(( pxTCB )->xStateListItem),(( pxTCB )->
    xTaskPeriod+xTaskGetTickCount()-(xTaskGetTickCount()%(( pxTCB )->
    xTaskPeriod))));
```

**Isječak koda 3.9:** Računanje `xItemValue` vrijednosti za EDF algoritam

### 3.5.2. Implementacija RTO algoritma

Kod RTO algoritma izvršavaju se samo zadatci koji se moraju izvršiti, a svi ostali se preskaču. Drugim rječima, čim neki posao možemo preskočii, preskočiti ćemo ga. Zadatci koji se šalju na izvršavanje raspoređuju se po EDF algoritmu. Jedina razlika u odnosu na EDF algoritam je ta što provjeravamo ublažene uvjete u strogom SRSV-u te ukoliko se zadatak može prekočiti ne dodajemo ga u listu zadataka koji čekaju na izvršavanje, nego on ostaje u listi za čekanje.

DODAJ PSEUDO KOD OVOG !!!!

### 3.5.3. Implementacija BWP algoritma

Kako je ranije opisano, BWP algoritam je proširenje RTO algoritma. Sada se svi zadatci stavljaju u listu zadataka spremnih za izvršavanje, ali opcionalni zadatci na red dolaze tek kada su svi strogi poslovi obavljeni. To je potrebno osigurati sortiranjem prema vrijednosti `xItemValue`. Svi poslovi i dalje će biti raspoređivani EDF algoritmom, ali podjeljeni u dvije skupine (poslovi koji se moraju izvršiti i oni čije je izvršavanje opcionalno). Pošto je `xItemValue` je 32-bitni cijeli broj, a manja vrijednost daje prioritet za izvršavanje, poslovima koji se ne moraju izvršiti postavit će se najznačajniji bit u 1. Time je osigurano da takvi poslovi budu zadnji na redu za izvršavanje.

DODAJ PSEUDO KOD OVOG !!!!

# 4. Implementacija simulatora

## 4.1. Generiranje skupova testnih zadataka

Generiranje skupova zadataka implementirano je u datoteci `taskSetGenerator.c`. U strukturi `periodic_task` sadržane su sve informacije potrebne za opis pojedinog zadatka, kontrolu njegovog izvođenja te prikupljanje podataka o poštivanju rokova izvršavanja tijekom simulacije. Za svaki zadatak definirana je jedna inačica ove strukture.

```
struct periodic_task{
    TaskHandle_t handler;
    char * name;
    double u;
    TickType_t period;
    TickType_t duration;
    int weakly_hard_constraint;
    int numOfPeriods;
    bool report[MAX_PERIOD_CNT];
    int missed_deadlines;
    int times_killed;
} Task_Set[MAX_TASK_CNT];
```

**Isječak koda 4.1:** Struktura periodic_task

Vrijednosti perioda generirane su uniformnom razdiobom. Gornja i donja granica intervala iz kojeg se nasumično biraju vrijednosti zadane su konstantama. U konkretnom slučaju provedenih simulacija vrijednosti perioda su iz intervala [20,100]. Male vrijednosti izbjegnute su zbog nepreciznosti kod zaokruživanja pri računanju vremena izvršavanja, što u najgorem slučaju ima za posljedicu znatnu promjenu faktora opterećenja sustava. Sljedeća veličina koja je potrebna za provedbu simulacije je hiperperiod. To je najmanji zajednički višekratnik vrijednosti perioda svih zadataka. Nakon vremena hiperperioda, raspored poslova se ponavlja jer se krajnji rokovi završetka svih zadataka poklope. Zbog toga je simulaciju potrebno provesti od trenutka t=0 do vrijendosti hiperperioda. Vrijednosti hiperperioda mogu biti prevelike te bi zbog toga simulacije trajale predugo. Kako bi se spriječio opisani problem, nakon generiranja perioda računa se vrijesnot hiperperioda i ukoliko je veća od zadane konstante generiranje perioda se pokreće iznova. Konstanta je izračunata tako da se simulacija

ne izvodi više od 10 sekundi.

Za generiranje faktora opterećenja za skup testnih zadataka korišten je algoritam UUniFast. Algoritam prima ukupan broj zadataka i sumu faktora opterećenja svih zadataka te uniformno raspoređuje faktore opterećenja. Ovaj algoritam korišten je jer pri generiranju nasumičnih vrijednosti ne daje prednost niti jednoj vrijednosti. Vremenska složenost algoritma je O(n). Kako bi generirani skup zadataka bio što sličniji stvarnim uvjetima , nakon generiranja faktora opterećenja ugrađena je provjera da neki zadatak ne zauzima previše procesorskog vremena. Konkretno, ako neki zadatak ima faktor opterećenja veći od 0.75, sve vrijednosti se odbaciju i algoritam se ponavlja. Ovu provjeru bilo je potrebno napraviti i iz razloga što je moguće da faktor opterećenja zadatka bude veći od 1, što nema smisla razmatrati.

Vrijeme izvršavanja pojedinog zadatka dobiveno je kao umnožak perioda i faktora opterećenja. Imena zadataka generiraju se u obliku `Task_xx`, gdje xx predstavlja id pojedinog zadatka, počevši od 1 do broja zadataka.

Ublaženo-strogi uvjeti zadataka, kao i periodi, generirani su uniformnom razdiobom. Vrijednosti su iz intervala [0,5]. DODAJ OPIS ZA UVJETE RASPOREDIVOSTI

Prije početka svake simulacije poziva se funkcija `startTaskSetGenerator()` koja poziva potrebne funkcije kako bi se izvršili svi ranije navedeni koraci za generiranje zadataka. Navedena funkcija kao argumente prima ukupan broj zadataka koje je potrebno generirati, zadani faktor opterećenja i putanju do datoteke u koju će se pohraniti rezultati simulacije.

```
void startTaskSetGenerator(double utilization,int n, char * report_file){
    TASK_CNT = n;
    total_utilization = utilization;
    file_path = report_file;
    calculateUtilization(utilization);
    readTaskPeriods();
    readWeaklyHardConstraint();
    calculateTaskDuration();
    calculateHiperperiod();
    calculateNumOfPeriods();
    generateTaskNames();
    resetTimesKilled();
    resetReports();
}
```

**Isječak koda 4.2:** Funckija startTaskSetGenerator

## 4.2. Tijek simulacije

Tijekom izvođenja simulacije za svaki se zadatak pamti je li izvršen do roka završetka u svakom periodu. Na taj način se dobije informacija o izvođenju svakog zadatka u svakom

pripadajućem periodu. To je implementirano pomoću polja varijabli tipa bool čije su vrijednosti na početku simulacije postavljene na 0 i nakon što se posao pravovremeno izvede ta vrijednost se postavi u 1. Simulacija se prekida nakon hiperperioda (najmanjeg zajedničkog višekratnika perioda svih zadataka).

DODAJ SLIKU JEDINICA I NULA ONIH SILNIH !!!

Iz polja nula i jedinica, prikazanog na slici x.y, koje predstavljaju izvršavanje poslova, potrebno je interpertirati podatke o provedenoj simulaciji. Za svaki zadatak potrebno je pobrojiti propuštene rokove završetka, kao i broj kršenja ublaženo strogih ujeta postavljenih nad generiranim skupom. Također je bitno znati jesu li bili zadovoljeni svi ublaženo-strogi uvjeti postavljeni nad skupom zadataka te kolika je kvaliteta usluge. Za sve navedeno implementirane su funkcije u datoteci `taskSetGenerator.c`.

OPISI CSV DATOTEKU te se u csv (eng. comma-separated values) datoteku izvještaja dodaje novi redak koji opisuje trenutno provedenu simulaciju.

## 4.3. Pokretanje simulacije

Radi usporedbe različitih algoritama nad generiranim setom zadataka potrebno je pokrenuti program uz različite faktore opterećenja. Za to je implementirana bash skripa kojom se najprije stvara csv datoteka u kojoj će biti pohranjen izvještaj sa podatcima o svim simulacijama pokrenutim za određeni set testnih zadataka. Za stvaranje datoteke izvještaja napisan je poseban c program koji osim što stvori datoteku zapiše u nju prvi red sa nazivima pojedinih stupaca. Skripta nadalje pokreće simulaciju od faktora opterećenja 0.9 do 1.5, uz korake 0.01 te se nakon svakog pokretanja simulatora u datoteku izvještaja upisuje jedna linija koja sadrži ranije opisane podatke. Pri pokretanju simulacije programu se preko argumenata prosljeđuju 3 parametra, broj zadataka, faktor opterećenja te datoteka na čiji će se kraj upisivati izvještaj nakon završene simulacije. Faktor opterećenja se radi jednostavnosti implementacije prosljeđuje pomožen sa 100, jer bash ne podržava rad sa decimalnim brojevima. U nastavku je priložena slika datoteke izvještaja.

DODAJ SLIKU IZVJEŠTAJA !!!

OPISI KAKO SE IZ IZVJEŠTAJA DOBIJU GRAFOVI!!!

IZNOS TICKA NA KONKRETNOM SIMULATORU

# 5. Rezultati

tu ce biti rezultati

# 6. Zaključak

tu ce nam biti zakljucak, kad ga naravno napisem

## 6.1. First section

ovo je prvo odlomak

## 6.2. Second section

ovo je drugi. laku noc !

# BIBLIOGRAPHY

**Nadogradnja operacijskog sustava FreeRTOS za primjenu u kontrolnim aplikacijama**

## Sažetak

Sažetak na hrvatskom jeziku.

**Ključne riječi:** Ključne riječi, odvojene zarezima.

**Title**

## Abstract

Abstract.

**Keywords:** Keywords.