Primjena operacijskog sustava za rad u stvarnom vremenu

Puđa, Josip

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Josip Puđa

Primjena operacijskog sustava za rad u stvarnom vremenu

DIPLOMSKI RAD

Varaždin, 2020.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Josip Puđa

Matični broj: 46407/17–R

Studij: Informacijsko i programsko inženjerstvo

Primjena operacijskog sustava za rad u stvarnom vremenu

DIPLOMSKI RAD

Mentor/Mentorica:

Izv. prof. dr. sc. Ivan Magdalenić

Varaždin, siječanj 2020.

Josip Puđa

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Razrada teme operacijskih sustava u realnom vremenu. Objašnjavanje komponenti potrebnih za razumijevanje i implementaciju operacijskih sustava u realnom vremenu. Ovo se proteže od procesa i dretvi, te alata za manipuliranje istih, do planera koji se koriste u sustavima. Opis sustava i algoritama koji su se koristili u prošlosti kao i onih koji se koriste danas. Razrada teme ugrađenih sustava koji služe kao glavna okolina za operacijske sustave ove vrste. Izrađena implementacija operacijskog sustava u realnom vremenu pomoću FreeRTOS operacijskog sustava kao međusloja. Prikaz implementacije kroz kod i alate koji su na raspolaganju za razvoj. Na ovaj način se omogućuje zaokružen prikaz razumijevanja operacijskih sustava u realnom vremenu te implementacija istih.

Ključne riječi: Operacijski sustavi, RTOS, Planer poslova, ugrađeni sustavi, mikrokontroler, procesi, dretve.

Sadržaj

1. Uvod

U našoj svakodnevici koristimo razne alate i pomagala koja služe kako bi nam omogućila ili olakšala obavljanje nekog posla. Vrijeme obavljanja nekog posla je od relativne važnosti, drugim riječima imamo prioritete koliko je bitno da se posao obavi u određeno vrijeme u određenom trenutku. Uzmimo u obzir poslove koje možemo naći unutar automobila te zamislimo da je cijeli automobil jedan veliki sustav gdje se poslovi obavljaju u nekom redoslijedu te konstantno izmjenjuju. Jedan od takvih poslova je traženje radio stanice. Takav posao, iako je možda nekim osobama bitan, nije nužno dovoljno kritičan da ima visok prioritet nad drugim poslovima, te možemo oprostiti ako se ne obavlja brzo ili u točno određenom trenutku. Sada u obzir uzmimo posao automatskog mjenjača. Automatski mjenjač je namijenjen da automatski mijenja brzinu automobila u određenom trenutku, kada se dostigne dovoljno visok broj okretaja za podizanje brzine, tj. dovoljno nizak broj okretaja za spuštanje brzine. Ovaj proces je bitan, jer ukoliko ne prebacimo auto u veću brzinu pri visokim okretajima automobil će biti ograničen brzinom i neće raditi optimalno. Zato je nužno da sustav unutar auta regulira brzinu što točnije kako bi postigao optimalan rad automobila. Takav posao ima veći prioritet od traženja stanice te nam je bitnije da automobil promijeni brzinu u određeno vrijeme nego što nam je bitnije da automobil promijeni radio stanicu. U ovom slučaju mi želimo da sustav obavi jedan posao prije nego li nastavi s drugim. Naravno, možemo imati više razina prioriteta. Za zadnji slučaj uzmimo sustav za aktiviranje zračnih jastuka. Posao aktiviranja zračnih jastuka na vrijeme je jako bitno i razlike u milisekundama mogu značiti razliku života ili smrti. Ovaj posao zauzima najveći stupanj prioriteta, te ukoliko je pozvan da se obavlja, želimo da se prekine posao mijenjanja brzine i posao mijenjanja radio stanice kako bi se obavio. Primjer koji smo sad naveli je samo jedan, no dobro predstavlja rad operacijskog sustava u realnom vremenu (RTOS).

Kroz ovaj rad proći ćemo kroz pojmove i dijelove koji čine operacijski sustav u realnom vremenu. Počevši s procesima i dretvama postaviti ćemo razliku te načine na koji možemo manipulirati procesima i dretvama što ćemo koristiti u našem programskom rješenju. Ustanoviti ćemo ulogu operacijskih sustav kao i razne vrste operacijskih sustava kako su se razvijali kako bi ustanovili što čini elemente modernog operacijskog sustava. Proći ćemo kroz ugrađene sustave gdje pronalazimo operacijske sustave u realnom vremenu. Obraditi ćemo pojmove kao što su

planer za obavljanje poslova te razne algoritme koji se koriste za planiranje poslova. Na kraju ćemo iskoristi sve to kako bi napravili vlastitu implementaciju operacijskog sustava. Uz to ćemo još usporediti sustav s operacijskim sustavima u realnom vremenu koji postoje na tržištu kao što je FreeRTOS.

Slika 1 uređaji s ugrađenim sustavima [1]

2. Procesi i dretve

Pri radu, računalo obavlja neki posao (*task*) tako što izvodi programe. Te programe možemo nazvati računalnim procesima ili samo procesima. Procesi mogu nastati na nekoliko načina, prilikom inicijalizacije sustava, poziv za sustav kreiranja poslova od strane procesa koji se obavlja, korisnički zahtjev ili inicijacija hrpe poslova. Oni primaju vremenske parametre pri pokretanju kao što su trenutak početka, trajanje izvođenja i trenutak završetka. Kako procesi mogu nastati tako mogu i biti terminirani. Proces može biti terminiran namjerno, s normalnim završetkom rada ili nastankom greške, ili nenamjerno, s fatalnom greškom ili 'ubijanjem' od strane drugog procesa. Na kraju imamo stanja u kojima procesi mogu biti a to su pokrenut, kada obavlja svoja posao i koristi procesor, u čekanju, kada je trenutačno zaustavljen kako bi se drugi proces obavljao i blokiran, gdje proces čeka vanjsko djelovanje kako bi bio pokrenut. Ova stanja se odnose na glavnu memoriju, model za prikaz stanja se može još proširiti. [2]

Uz procese postoje još i dretve. Dretve imaju brojač koji im govori koju instrukciju trebaju izvesti sljedeću. Svaki proces ima jednu dretvu premda više dretvi se može pokrenuti u jednom procesu, to se zove *multithreading*. Dretve dijele adresni prostor tako da mogu manipulirati istim podacima u procesu. Prilikom *multithreadinga* procesi počinju s jednom dretvom te ta dretva može izvršiti instrukciju koja generira još dretvi. Svaka dretva ima svoj stog instrukcija koje mora izvršiti. Naravno ukoliko nema pravilne kontrole, ovo može uzrokovati kaos u obliku da se neke vrijednosti prepišu prije nego li druga dretva ju stigne iskoristit u oblik u kojem smo mi htjeli. Ovaj se problem pojavljuje u slučaju više dretvi te još više u slučaju kada imamo više procesa. Tada koristimo sustave zaključavanja resursa. [2]

2.1. Kritični odsječak

Kako bi procesi međusobno komunicirali moraju koristiti zajedničke resurse. Naravno, bitno je kada koji proces koristi resurs. Ukoliko je raspored neispravan, možemo dobiti pogrešne rezultate pri čemu mogu nastati veliki problemi. Kritični odsječak je pojam koji predstavlja trenutak kada proces koristi zajednički resurs pri čemu drugi procesi ne mogu koristiti taj isti resurs. Naravno i dalje imamo pitanje kako ćemo ostvariti višeprocesni rad u kojem dva procesa ne ulaze u kritični odsječak u isto vrijeme. Tu stupaju semafori i mutexi kao metode zaključavanja. [2]

2.1.1. Semafor

Semafor je cjelobrojna varijabla (*integer*) koja se koristi kao mehanika signaliziranja. Na početku se postavlja kao pozitivni broj te pri radu se koriste dvije operacije za semafor, povećavanje i smanjivanje. Prilikom operacije smanjivanja, program gleda da li je semafor veći od nula. Ukoliko nije proces se pauzira te čeka dok ga se ne probudi. Ukoliko je semafor veći od nula, proces ulazi u kritični odsječak i obavlja svoju operaciju. To radi sve dok ne dođe do operacije za smanjivanje kada proces napušta kritični odsječak te program odabire koji proces se budi iz svoje pauze. Semafori kojeg koriste samo dva stanja pri čemu je moguće da samo jedan proces ulazi u kritični odsječak zovu se binarni semafori. [2]

2.1.2. Mutex

Mutex je varijabla koja može imati dva stanja: otključan i zaključan. Ukoliko je vrijednost nula tada je mutex otključan, ukoliko je bilo koji drugi broj mutex je zaključan. U slučaju da je mutex otključan, proces povećava vrijednost mutexa i ulazi u kritični odsječak. U slučaju da je mutex zaključan proces daje kontrolu nad procesorom drugoj dretvi. Mutex može biti otključan samo od strane dretve koja ju je zaključala, na ovaj način imamo sigurniji oblik kritičnog odsječka gdje osiguravamo da resursu nije moguće pristupiti slučajno iz druge dretve dok je prva dretva u kritičnom odsječku.[2]

3. Operacijski sustavi

Računalo se sastoji od raznih komponenti koji čine računalno sklopovlje, Izvođenje računalnih operacija zahtjeva odgovarajuću programsku opremu, tu u pomoć dolaze operacijski sustavi. Operacijski sustav je skup programa koji olakšavaju korištenje računala, ili konkretnije, izvođenja određenih operacija računala. No olakšavanje korištenja računala je samo jedan zadatak operacijskog sustava. Drugi veoma bitan zadatak je djelotvorna organizacija iskorištavanja svih dijelova računala. To se odnosi na mogućnost da se u računalu odvija više procesa istovremeno. Procesor se može brinuti o izvođenju jednog niza instrukcija, no istovremeno mora biti u stanju se prebaciti na izvođenje drugog niza instrukcija. Ovo se još naziva višeprogramski rad. U ovom radu ćemo se više posvetiti drugom zadatku. [3]

3.1. Pristupi obradi poslova

Kada pričamo o obradi poslova, imamo nekoliko pristupa koji su se koristili kroz evoluciju računalstva. Neki od njih su: serijsko procesiranje (*Serial Processing*), jednostavan batch sustav (*Simple Batch System*), višeprogramirani batch sustav (*Multiprogrammed Batch System*), vremensko dijeljeni sustav (*Time-Sharing System)*. Svaki sustav ima svoju svrhu, kao npr. obavljanje poslova s manjim opterećenjem sustava, obavljanje više poslova paralelno za skraćeno vrijeme rada ili smanjeno vrijeme odaziva. S vremenom neki pristupi su postali zastupljeniji zbog potreba koje nalazimo u stvarnim slučajevima. [4]

3.1.1. Serijsko procesiranje

Najstariji oblik obavljanja poslova. Korišteno u starim računalnim sustavima koji nisu imali operacijske sustave, programeri su direktno utjecali na računala u serijama. Računala su primala ulaz u obliku čitača kartica, a izlaz u obliku ispisa na papir. Problemi koji su nastali pri ovom pristupu su bili:

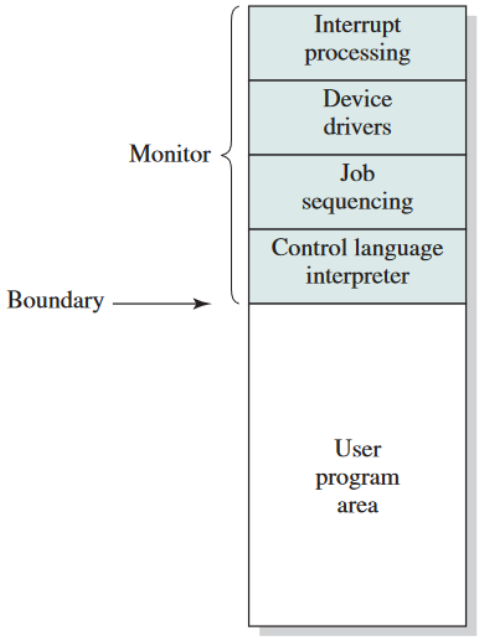
- Raspored poslova: Korištenje računala je zahtijevalo prijavu na određeno vrijeme s time je zahtijevalo da se željeni posao obavi u određeno vrijeme. Obavljanje posla prerano bi rezultiralo u neiskorištavanju resursa računala, dok obavljanje poslova prekasno bi rezultiralo u prekidu obavljanja posla.

- Vrijeme postavljanja: Svaki posao bi zahtijevao ponovno postavljanje i učitavanje *compiler*-a i programskog jezika u memoriju. U slučaju greške bilo je potrebno ponavljati cijeli proces od umetanja kartice do učitavanja programa i obrade posla.

Pristup se zvao serijskim procesiranjem zato što su korisnici pristupali i koristili računalo u serijama. U suštini pristup je bio obavljanje po jednog posla u jednom trenutku u raspoloživom vremenu. [4]

3.1.2. Jednostavan batch sustav (*Simple Batch System*)

Ograničenje po pitanju rasporeda poslova unutar serijskog procesiranja je predstavljao veliki problem. Količina neiskorištenog računalnog potencijala bila je preveliki trošak. Kako bi se poboljšala iskoristivost, koncept *batch* OS-a je bio razvijen. Koncept se temelji na skupljanju više poslova u hrpu (*batch*) i nakon toga bi se poslovi serijski obrađivali prema rasporedu u kojem su došli. Ovo se ostvarilo pomoću softwarea koji se zvao *monitor*. Korisnik više nije imao pristup računalnom sustavu, već bi predao posao koji se želi obaviti na karticama računalnom operatoru. Računalni operator bi zatim skupio sve poslove u hrpu (*batch)* i unosio sve u *monitor*. Svaki program je napravljen tako da se pri kraju vraća nazad na monitor koji bi pokrenuo sljedeći posao na redu. Tako da u suštini imamo dva dijela svakog programa, monitor i korisnički program. [4]



Slika 2 Simple Batch System [4]

Posao monitora je provođenje rasporeda obavljanja poslova. Uz to cilj je smanjivanje vremena izgubljenog između svakog posla tako što se mijenjanje obavlja što brže te s time što se ubrzava vrijeme postavljanja svakog posla. Sa svakim poslom uključene su upute uz pomoć oblika jezika za kontroliranje (*job control language*) ili *JCL*. Ovo je poseban oblik programskog jezika koji se koristi za davanje uputa monitoru za izvođenje. Nakon što je spreman da pokrene korisnički program, monitor prepušta kontrolu korisničkom programu te, nakon što se korisnički program obavi, ponovno preuzima kontrolu. Uz to poželjne su i druga svojstva:

- Zaštita memorije: Dok se korisnički program pokreće, on ne smije mijenjati memoriju na kojoj se nalazi monitor. U slučaju prave zaštite, slučaj mijenjanja memorije na kojoj se nalazi monitor bi trebao biti prepoznat od procesora te kontrola prebačena monitoru. Monitor bi zatim prekinuo program i izbacio grešku.
- *Timer*: Implementacija *timer*-a bi spriječilo slučajeve u kojem jedan posao uzima previše vremena i time monopolizira sustav. *Timer* bi započeo na početku posla te ukoliko posao traje predugo, prekinuo rad posla i vratio kontrolu monitoru.
- Privilegirane upute: Upute na razini strojnog jezika koje mogu biti pokrenute samo od strane monitora. Ukoliko korisnički program ima uputu koja izvodi uputu koju samo monitor može obaviti, tada se prekida program, izbacuje greška i vraća kontrola

monitoru. Primjer ovakve upute bile bi upute koje uključuju uređaje koji koriste priključke za ulaz i izlaz (*eng. I/O devices*) . Ukoliko korisnik želi koristiti privilegirane upute, on mora u vlastitoj uputi tražiti to od monitora.

- Prekidi: Prekidi omogućuju da operacijski sustavi lakše daju i preuzimaju kontrolu nad programima. Ovo se naravno čini prekidima programa.

Iz stvari kao zaštita memorije i privilegiranih uputa su nastali koncepti načina korištenja sustava. Korisnički programi su se pokretali u korisničkom načinu korištenja sustava, dok su monitori koristili jezgreni (*kernel*) način korištenja sustava. Načini korištenja su postali redovna stvar u modernim operacijskim sustavima. Kao rezultat žrtvovala se određena količina memorije i procesorskog vremena za rad monitora. [4]

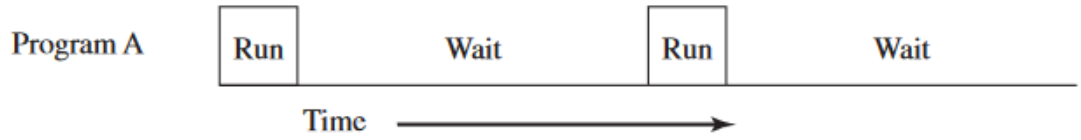
3.1.3. Višeprogramirani batch sustav (*Multiprogrammed Batch System*)

Premda je korištenje sustava bilo povećano pomoću jednostavnog batch sustava, procesorski potencijal je i dalje bio velikim dijelom neiskorišten. Razlog ovoga je bila brzina kojom su radili uređaji ulaza i izlaza (*I/O devices*). Ukoliko se posao sastoji od 100 uputa za obradu podataka koji zajedno traju 1 μs, čitanje datoteke pomoću I/O uređaja koje traje 15 μs i pisanje u datoteku pomoću I/O koje traje 15 μs. Cijeli posao će nam trajati 31 μs dok posao u kojem koristimo procesor traje 1 μs . Računanjem iskorištenost procesora dobivamo iskorištenost od 3.2%.

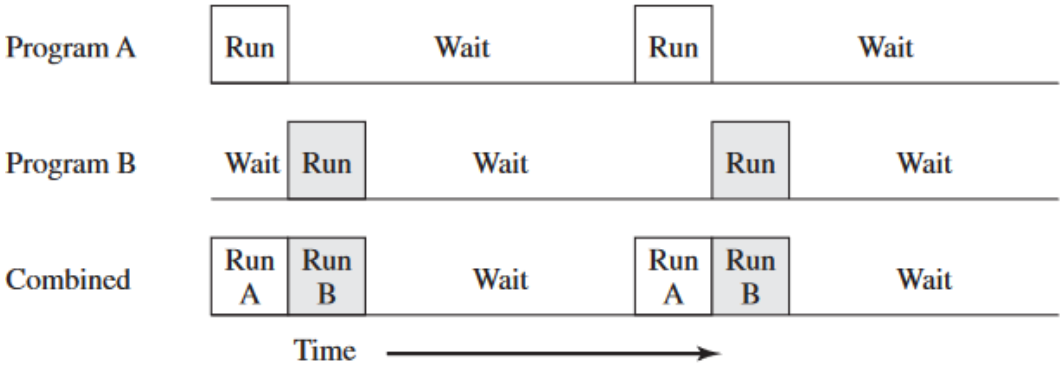
$$Postotak\ iskorištenosti\ Procesora = \frac{1\ \mu s}{31\ \mu s} = 0.032 = 3.2\%$$

Jednadžba 1 Postotak iskorištenosti procesora

Ovim vidimo da u većini vremena naš sustav uopće ne koristi procesor. U sustavu imamo dovoljno memorije da pohranimo monitor i jedan korisnički program, no pretpostavimo da možemo u memoriju pohraniti dva korisnička programa istovremeno. Ukoliko jedan posao čeka I/O, vjerojatnost je da drugi posao ne čeka I/O. Uzimajući ovo u obzir možemo proširivati memoriju da drži više poslova kako bi se povećala iskorištenost procesora. Ovo svojstvo nazivamo *Multiprogramming* ili *Multitasking* i ono čini centralnu temu modernih operacijskih sustava. Multiprogramiranje je znatno kompleksnije od uniprogramiranja. Za poslove koji se trebaju obaviti oni se moraju pohraniti u glavnu memoriju te se treba postaviti upravljanje memorijom. Ukoliko je više poslova spremno za pokretanje, onda je potrebno i postaviti neku vrstu *schedulera*. [4]



Slika 3 Uniprogramming [4]



Slika 4 Multiprogramming [4]

3.1.4. Vremensko dijeljeni sustav (*Time-Sharing System)*

Problem koji imamo s prijašnjim sustavima je pristup poslovima. Ukoliko želimo obaviti neki posao, uvijek moramo čekati da se prijašnji posao obavi ukoliko želimo utjecati na računalo. Kako je multiprogramiranje riješilo problem obavljanja *batch* poslova, ono se dalje unaprjeđivalo kako bi moglo raditi obavljanje više interaktivnih poslova. Ovo se zvalo dijeljenje vremena. Ovo je nastalo kao rezultat upravljanjem računala od strane više korisnika. Ukoliko je je bilo *n* korisnika, tada bi dijeljenje računala bilo *1/n* za svakog korisnika. Uzimajući u obzir brzinu ljudi u odnosu na računalo, vidimo da ovakav pristup dizajnu nije dobar. Cilj je bio smanjiti vrijeme koliko je korisnik radio na računalu te povećati odzivnost računala. U početku princip je bio prekinuti rad trenutnog korisnika i vratiti kontrolu OS-u kako bi dao kontrolu drugom korisniku. Poslovi i podaci s kojima je korisnik radio se spremaju. Prekidi su se obavljali u određenim ciklusima koji su bili diktirani na temelju sustavnog sata. Ovaj pristup se zvao *vrijeme rezanja* (*time slicing*). [4]

U vremenu kada je bilo razvijeno, vremensko dijeljenje je bilo namijenjeno kako bi se učinkovitije koristilo jedno računalo na više korisnika. Iako to je i danas korisno, računalstvo je postalo kompleksnije kao i njegova primjena tako da umjesto korisnika, primjenu još možemo

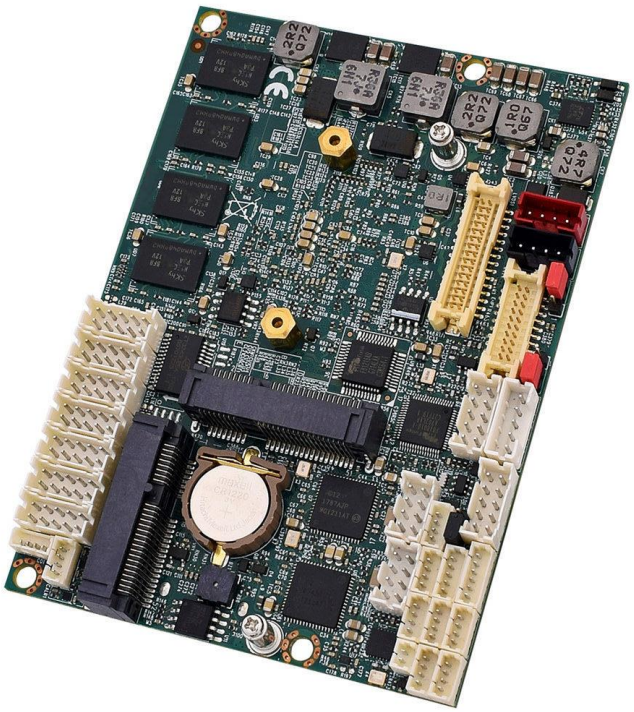
naći i u dijeljenu računalnih resursa među procesima ili konkretnije računalima, kao što to možemo naći u M2M sustavima danas. Ovdje također pronalazimo dio karakteristike operacijskih sustava u realnom vremenu. Operacijski sustav u realnom vremenu treba obaviti posao u određeno vrijeme i zbog se oslanja na raspoređivač posla i ispravan sustavni sat. [4]

4. Ugrađeni sustavi

Ugrađeni sustavi (*Embedded systems*) su primijenjeni računalni sustavi koji se razliku od osobnih računala na nekoliko načina.

- Po hardwareskim i softwareskim funkcionalnostima. Ugrađeni sustavi generalno imaju slabije performanse, manju potrošnju, manje memorije i tako dalje. Pored toga, ugrađeni sustav ne mora imati operativni sustav ili ima ograničeni sustav.
- Ugrađeni sustav je napravljen da izvodi određenu funkciju. Funkcije mogu varirati no sustavi su generalno napravljeni da rade manji broj funkcija.
- Ugrađeni sustav ima bolju kvalitetu i veću pouzdanost naspram drugih računalnih sustava. Kvaliteta i pouzdanost su veliki faktor za većinu funkcija. sustavi koji rade s kritičnim poslovima ne smiju zakazati, inače se štete mogu biti katastrofalne.

I dalje postoje rasprave oko definicije ugrađenih sustave i gdje se povlači crta između ugrađenog sustava i osobnog računala. [5]



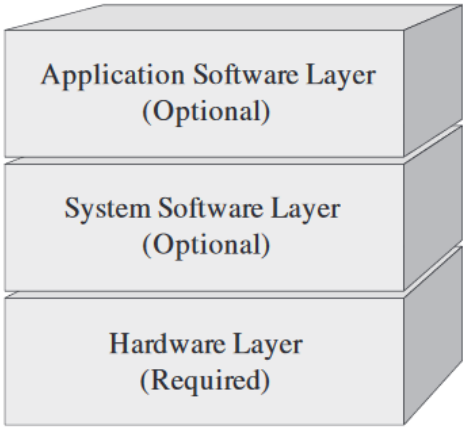
Slika 5 Ploča za razvoj uređenog sustava [6]

4.1. Model ugrađenih sustava

Arhitekture ugrađenih sustava variraju, no generalno možemo prikazati arhitekturu kroz model ugrađenih sustava. Model ugrađenih sustava je općeniti prikaz arhitekture koji se može primijeniti na pojam ugrađenih sustava umjesto na jedan specifični sustav. U suštini sustav ugrađenih sustava se ne razlikuje puno od prosječnog računalnog sustava. Naravno, ugrađeni sustav je vrsta računalnog sustava. U modelu imamo slojeve koje čine arhitekturu posložene u stogu.

- Hardwareski sloj – Nužan sloj za ugrađene sustave. Ugrađeni sustavi moraju biti razvijeni na računalnom hardwareu koji se sastoji od glavnih fizičkih komponenti.
- Sistemski sloj – Sloj na kojem se nalaze operacijski sustavi, driveri i međusloj. Ovaj sloj nije potreban za ugrađene sustave.
- Aplikacijski sloj – Sloj na kojem se nalaze aplikacije i funkcionalnosti ugrađenog sustava. Ovdje se obavljaju interakcije s korisnicima. Ovaj sloj nije potreban za ugrađene sustave.

Model se koristi kako bi se prikazao ugrađeni sustav te time olakšalo dizajniranje sustava. Svaki od ovih slojeva se može promijeniti prilikom faze planiranja. [7]



Slika 6 Model ugrađenog sustava [7]

4.2.Operacijski sustavi ugrađenih sustava

Operacijski sustavi se nalazi u sistemskom sloju ugrađenih sustava koji se stavlja na hardware sustava. Operacijske sustave koje možemo pronaći u ugrađenim slojevima se razlikuju na mnogo

načina, no kada se razvijaju ugrađeni sustavi obično se dijele na RTOS i embedded Linux. S vremenom su se razvile i Linux distribucije koje podržavaju svojstva u realnom vremenu no podjela je i dalje ostala ista. [8]

4.2.1. Operacijski sustav u realnom vremenu

Operacijski sustavi u realnom vremenu, ili **RTOS** (*Real-time operating system*), su operacijski sustavi koji posjeduju sljedeće karakteristike:

- Odlučnost
- Odzivnost
- Korisnička kontrola
- Pouzdanost
- Fail-soft rad

Odlučnost predstavlja svojstvo sustava da obavlja posao u određeno vrijeme, ili u određenom rasponu vremena. Nijedan sustav nije u potpunosti odlučan, no koliko je sustav u mogućnosti odgovoriti na prekide i koliko je u mogućnosti obaviti sve poslove u određenom vremenu, predstavlja koliko odlučno sustav može zadovoljiti sve zahtjeve. Jedna mjera za mjerenje odlučnosti je koliko vremena prolazi od kada je došao visoko prioritetni zahtjev za prekidom naspram koliko je dugo trajalo da zahtjev započne. Naravno u RTOS-u taj vremenski razmak može trajati samo par mikrosekundi. [4]

Odzivnost predstavlja koliko dugo vremena treba sustavu nakon prihvaćanja prekida za obavljanje zahtijevanog prekida. U odzivnost se uzimaju aspekti koliko je vremena potrebno sustavu da započne s prekidom, tj. da se započne *interrupt service routine* (**ISR**) koji predstavlja proces prekida, koliko je potrebno vremena za obavljanje ISR-a, ovo je ovisno o hardwareu, te ukoliko je moguć nastanak ugniježđena prekida, tj. prekid ISR-a zbog dolaska novog prekida. Odlučnost i odzivnost čine srž RTOS-a. [4]

Korisnička kontrola predstavlja količinu kontrole s kojom korisnik raspolaže. Za razliku od standardnih operativnih sustava, RTOS mora pružati znatno veću kontrolu nad sustavom. Korisnik mora biti u mogućnosti postavljati prioritete poslova te diktirati koji su poslovi moraju biti obavljeni u određenom poslu (*hard tasks*), a koji poslovi nisu vremenski ovisni (*soft tasks*).

Daljnje mogućnosti mogu biti definiranje procesa koji se izvode za izmjenu poslova, koji procesi se izvode u glavnoj memoriji, kakva prava imaju koje hijerarhije procesa itd. [4]

Pouzdanost je uobičajeno puno važnija za RTOS. Dok u slučaju većine sustava kvarovi mogu rezultirati neugodnostima koje mogu biti popravljene ponovnim pokretanjem, u slučaju da RTOS zakaže prilikom rada s poslovima u realnom vremenu posljedice mogu biti kritične pa čak i snositi životne opasnosti. Naravno, ovo je zbog poslova za koje se koristi RTOS, ali to je baš zato što se od RTOS-a očekuje visoka razina pouzdanosti.

Fail-soft rad je svojstvo sustava da prilikom zakazivanja sustav zakaže na način da zadrži što veću mogućnost rada i što veću količinu podataka. Prilikom zakazivanja tradicionalnih UNIX sustava, sustav ispiše grešku na konzoli, spremi što više podataka za daljnju analizu te prestaje s radom. U usporedbi, RTOS sustav mora ispraviti grešku ili umanjiti njen utjecaj te nastaviti s radom. U slučaju da je prekid rada potreban, nastoji se zadržati što veća konzistentnost podataka. Bitan aspekt fail-soft rada je stabilnost. Ukoliko sustav nije obavio sve zadatke na vrijeme, trebao bi barem obaviti najbitnije poslove u zadano vrijeme, iako drugi poslovi nisu obavljeni. [4]

4.2.2. Embedded Linux

Embedded Linux je korištenje jezgre Linuxa za razvijanje operacijskog sustava. Embedded Linux se često ne koristi u istim slučajevima kao i standardni sustavi već češće za kompleksnije sustave koji koriste mikroprocesore. S vremenom se razvio veliki broj operacijskih sustava temeljenim na Linux jezgri za ugrađene sustave kao i alati za implementaciju embedded Linuxa. Neki od Embedded Linux operacijskih sustava koje možemo naći na tržištu danas su Android, kojeg možemo pronaći na mnogim mobilnim uređajima, i OpenWrt, kojeg možemo pronaći na usmjerivačima. [9]

5. Planiranje poslova

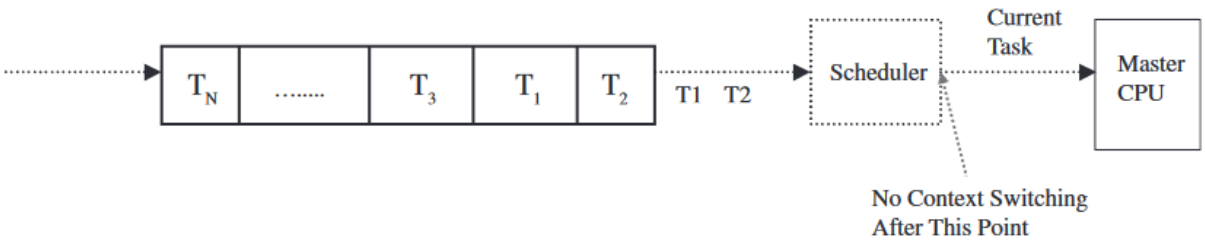
Za rad s više procesa operacijski sustavi imaju mehanizam koji se zove planer poslova (*scheduler*). Planer odlučuje redoslijed kojim se poslovi obavljaju te koliko ti procesi mogu trajati. Postavlja stanje u kojem se posao trenutno nalazi, to mogu biti čekanje, rad i pauzirano stanje. [5]

5.1. Algoritmi planiranja

Postoji mnogo algoritama za planiranje poslova i svaki algoritam ima svoje prednosti kao i nedostatke. Glavni faktori koje moramo uzeti u obzir jesu vrijeme odaziva, ili vrijeme koje je potrebno planeru da promijeni kontekst na posao koji je u spremnom stanju, vrijeme uzvrata, ili vrijeme potrebno da se proces obavi, vrijeme za odlučivanje sljedećeg procesa za pokretanje (*overhead*) te faktor po kojem se odlučuje koji posao se sljedeći pokreće. Cilj planera je što veća iskoristivost računala. To se ostvaruje s velikom propusnošću poslova, tj. procesiranjem što više poslova je moguće. Ujedno je cilj izbjeći situaciju u kojoj posao se nikad ne obavlja. Algoritmi se najčešće dijele na dva dijela: neprekidni i prekidni. neprekidni algoritmi su algoritmi u kojima poslovi koriste CPU dok ne završe, neovisno o trajanju ili važnosti drugih poslova. Prekidni algoritmi koriste hijerarhiju za planiranje poslova. Svaki posao ima određeni prioritet te ukoliko se pojavi posao većim prioritetom od posla koji se trenutno obavlja, planer može zaustaviti posao s manjim prioritetom kako bi pokrenuo rad posla s većim prioritetom. [5]

5.1.1. First-Come-First-Serve (FCFS)

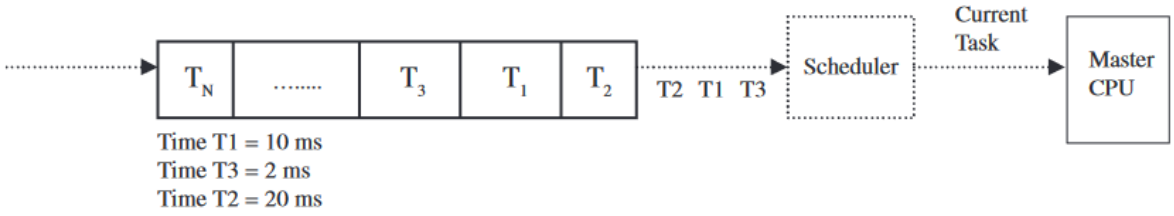
First-Come-First-Serve (*Run-To-Completion*) je algoritam gdje se poslovi obavljaju kako dolaze i obavljaju se dok nisu gotovi. U ovom slučaju nemamo blok za pauzirane poslove zbog čega je ovo nepreventivni algoritam. Vrijeme odzivnosti FCFS algoritma je sporije od drugih algoritama. Kao rezultat kraći poslovi na kraju reda čekaju duže zbog dužih poslova koji se mogu naći ispred njih. Prednosti ovoga pristupa je to što se svi poslovi prije ili kasnije moraju obaviti, tj. u normalnom radu nemoguće je da posao neće biti neobavljen. [5]



Slika 7 FCFS [5]

5.1.2. Shortest Process Next (SPN)

Shortest Process Next (*Run-To-Completion*) je algoritam gdje se poslovi obavljaju prema njihovom trajanju izvođenja. Prednosti algoritma su veća odzivnost prema poslovima s kraćim trajanjem izvođenja. S druge strane problem na koji nailazimo s ovim algoritmom je neobavljanje poslova s dužim trajanjem izvođenja u slučaju u kojem konstantno se pojavljuju poslovi s kraćim trajanjem izvođenja. [5]



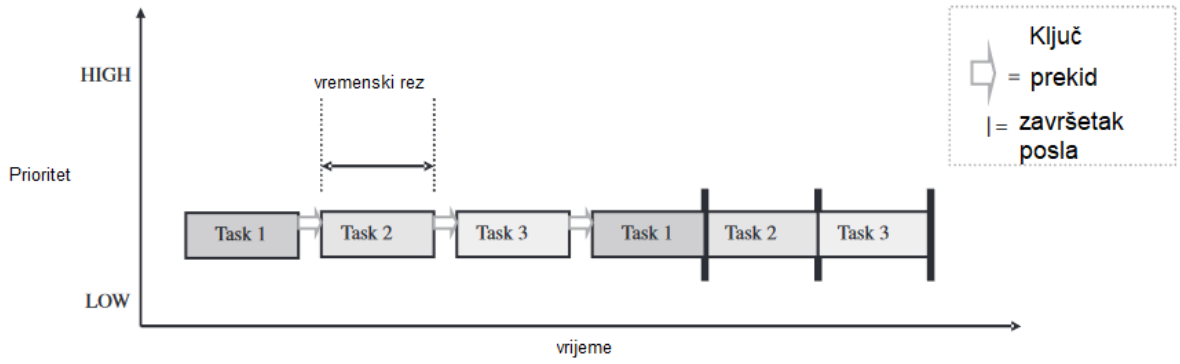
Slika 8 SPN [5]

5.1.3. Round Robin

Round Robin (FIFO) je algoritam koji dodaje poslove spremne za obradu u red. Procesi se dodaju na kraju reda a obrađuju se poslovi koji su na početku reda. Svi poslovi su jednaki neovisno o tome koliko opterećuju sustav ili koliko su interaktivni. Svakom poslu je dodijeljen jednaki rez vremena (*time slice*) gdje se poziva prekid na kraju isteka tog vremena. Nakon prekida slijedi posao koji sljedeći u redu. Ukoliko posao nije završen u vremenu reza, tada se on smješta na kraju reda kako bi se završio sljedeći put kada dođe na red. Ukoliko se posao završi na vrijeme, tada se kontrola procesora otpušta te planer poziva sljedeći posao u redu.

Prednosti su jednako izvođenje svih poslova, no problemi mogu se pojaviti ukoliko nailazimo na razne poslove s dužim izvođenjem koji se konstantno odgađaju pri čemu se više vremena provodi čekajući izmjene konteksta. Također problem može nastati pri međusobnoj interakciji

poslova gdje jedan posao čeka izvođenje drugog posla zbog podataka, pri čemu se poslovi provode nepotrebno vrijeme koristeći procesor. Protočnost poslova ovisi o rezu vremena, ukoliko je rez prekratak, veliki poslovi će se češće rotirati, dok prevelik rez stvara jednak problem kao i neprekidni algoritmi. Nemoguće je doći do situacije u kojem poslovi nikada neće biti izvršeni. [5]

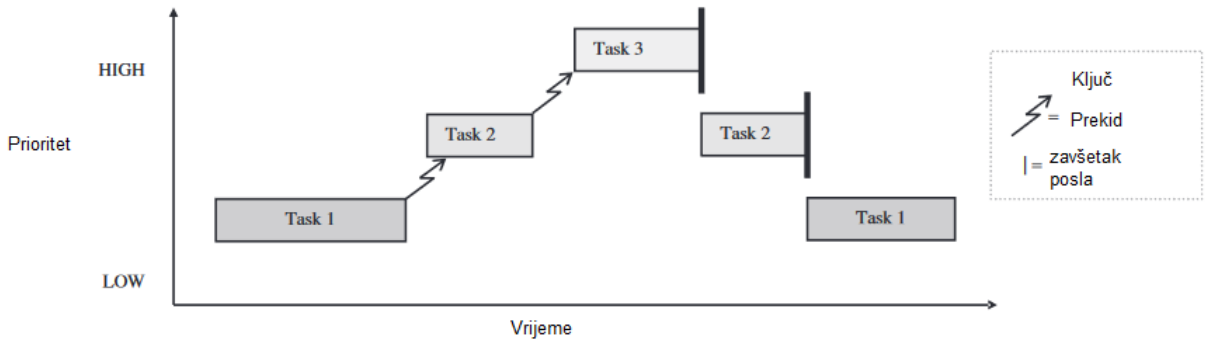


Slika 9 Round robin [5]

5.1.4. Prioritetno planiranje

Prioritetno planiranje, ili planiranje s prekidima, definira hijerarhiju između poslova pri čemu važniji posao ima prednost nad poslom s manjim prioritetom. Tako da ukoliko se pojavi posao s većim prioritetom može se pozvati prekid ukoliko posao koji se obavlja ima manji prioritet od posla koji je pozvan. Iako ovo rješava neke probleme koji nastaju s *Round robin* algoritmom, nastaju novi problemi.

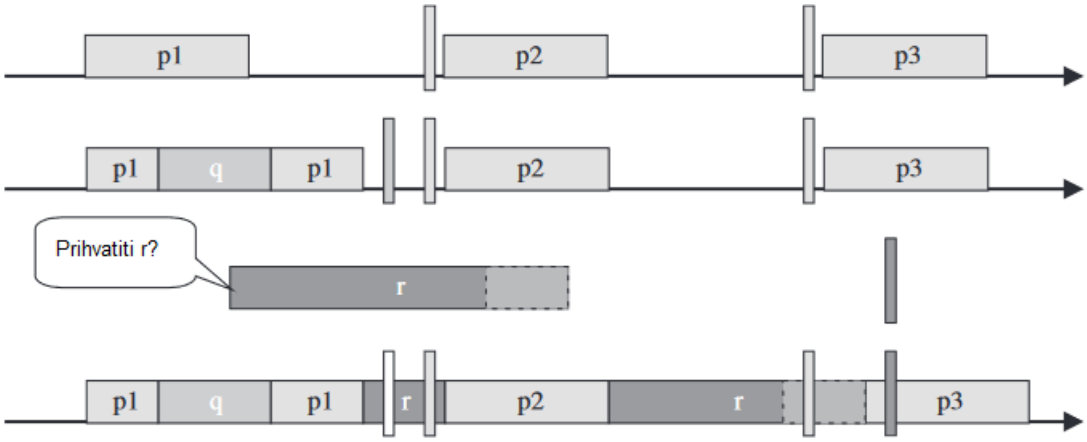
- Može doći do situacije u kojoj neki poslovi nikad se ne obavljaju zbog toga što konstantno nailaze poslovi s višim prioritetima. Ovo se može izbjeći tako da se povećava prioritet poslovima koji duže čekaju u redu.
- Inverzija prioriteta pri čemu poslovi višeg prioriteta su blokirani čekajući da se poslovi nižeg prioriteta obave dok poslovi prioriteta između imaju viši prioritet u pokretanju. Tako da posao nižeg prioriteta i višeg prioriteta se ne pokreću.
- Kako odrediti prioritete za razne poslove. U suštini, prioritet je vezan za važnost posla. Što je važniji posao, to je veći prioritet. No ukoliko su poslovi jednake važnosti moramo odlučiti koji posao ima veći prioritet. Jedan od pristupa za odlučivanje je *Rate Monotonic Scheduling* (RMS). [5]



Slika 10 Prioritetno planiranje [5]

5.1.5. Earliest Deadline First (EDF)

Earliest Deadline First (*Clock Driven Scheduling*) je algoritam koji se bazira na frekvenciji posla, tj. koliko puta je posao pokrenut, roku završetka posla i trajanju posla. EDF omogućava forsiranje izvođenja svakog procesa u određeno vrijeme, problemi nastaju kada treba definirati ispravno trajanje za različite procese. Obično se uzima prosječno trajanje posla. [5]



Slika 11 EDF [5]

6. Implementacija

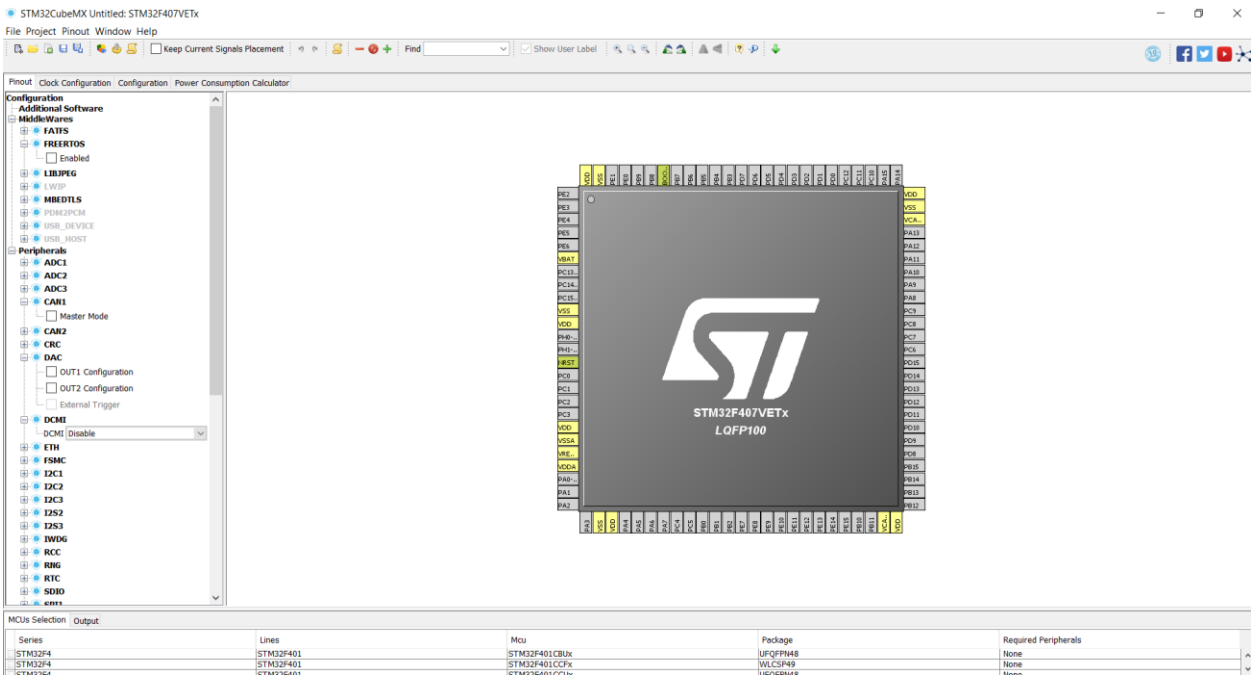
Na temelju prijašnjih poglavlja možemo napraviti implementaciju operacijskog sustava u realnom vremenu. Pristup koji ćemo koristiti je implementacija kao međusloj na razvojnoj ploči koja dolazi s ugrađenim operacijskim sustavom. Razlog zašto radimo pristup međuslojne implementacije je kako bi se fokusirali na bitnije aspekte operacijskih sustava u realnom vremenu što je pravovremeno obavljanje određenih poslova. Mnogi broj ploča dolazi sa svojim operacijskim sustavom koji radi u realnom vremenu, no kod mnogih operacijskih sustava bez dodavanja međusloja prekidi se pojavljuju samo u slučaju određenih priključaka. Također ćemo koristiti postojeće aspekte ovih operativnih sustava kao što su alokacija memorije. Na ovaj način možemo napraviti alat koji se primjenjuje na više razvojnih ploča bez da mijenjamo adrese na koje pišemo podatke.

6.1. Alati

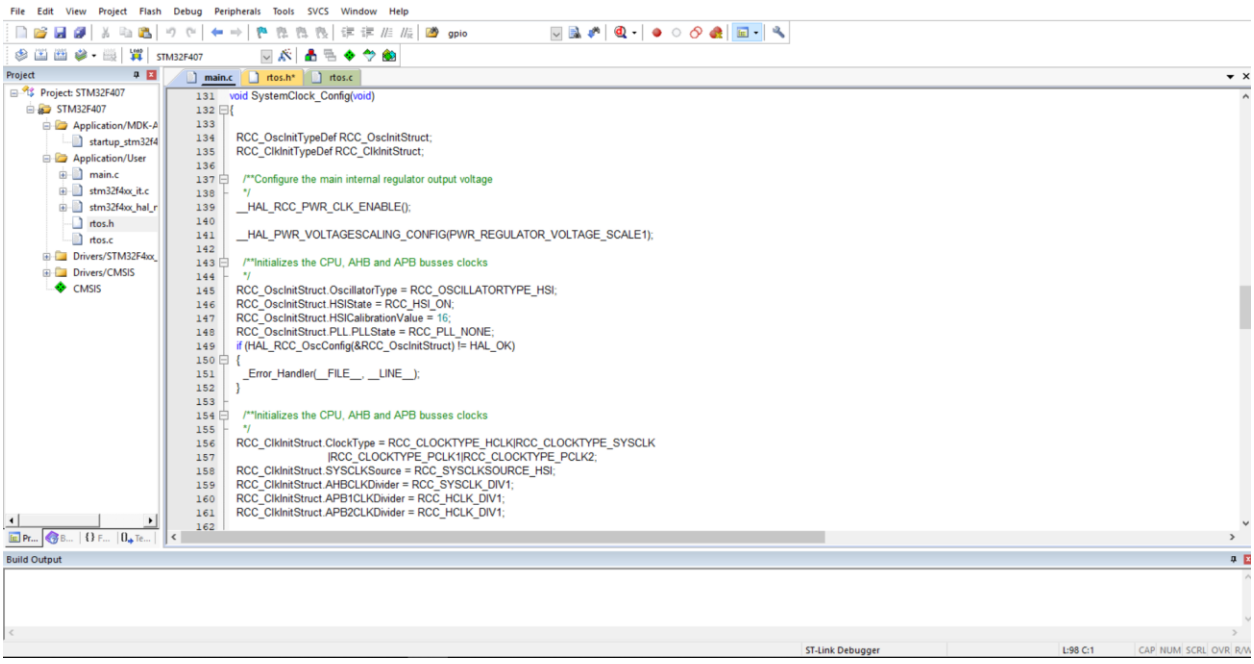
Za izvedbu koristiti ćemo nekoliko alata od hardwarea na kojem instaliramo programsko rješenje do raznih alata koje koristimo za razvoj programskog rješenja. Mikrokontroler na kojem ćemo instalirati programsko rješenje. Baziran na ARM Cortex M4 arhitekturi frekvencije od 168 MHz. Posjeduje 512 KB flash memorije i 192 KB SRAM.

Alat za postavljanje projekta je STM32CubeMX. Preko ovog alata mogu se postavi svi potrebni elementi za STM32 razvojne ploče. Mogućnosti variraju od izmjene individualnih priključaka za razne uporabe, dodavanja FreeRTOS operacijskog sustava kao međusloj, izmjene radnog takta mikrokontrolera te mnoge druge mogućnosti. Nakon postavljanja alat može generirati kod u obliku projekta u jednom od brojnih IDE-va na raspolaganju.

Okružje za razvoj (IDE) koje ćemo koristiti je Keil uVision 5. Ovo je IDE napravljen za rad s ugrađenim uređajima te nudi mogućnosti kao otkrivanje grešaka pomoću programera, sastavljanje (eng. *compiling*), te prevođenje u hex datoteke za instaliranje na razvojnu ploču. Programski jezik kojem ćemo se služiti je C. Hex datoteke ćemo instalirati pomoću alata FLASHER-STM32 koji omogućuje instaliranje putem USART mosta.



Slika 12 STM32CubeMX [autorski rad]



Slika 13 Keil uVision 5 [autorski rad]

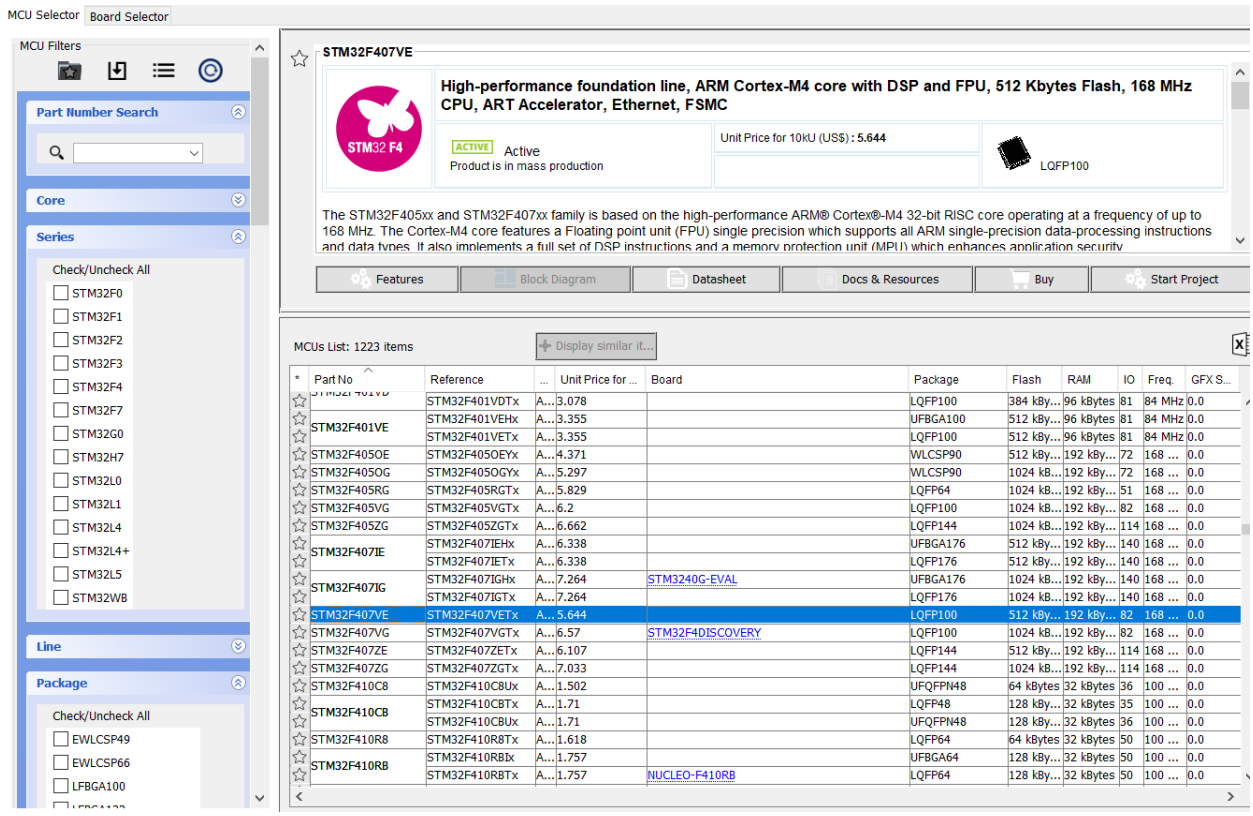
6.2. Primjena FreeRTOS-a

Problem koji smo predstavili na početku je bio vezan za sustave automobila. Naravno da bi to ostvarili trebalo bi nam puno vremena i resursa. Zato ćemo smanjiti problem na jednostavnije koncepte koje oni predstavljaju. Uzeti ćemo u obzir tri posla različitih prioriteta. Posao A će biti najnižeg prioriteta, posao B će imati prioritet srednje vrijednosti, dok posao C će imati visoki prioritet te će biti riješen pomoću prekida. Cilj nam je napraviti primjer sustava koji oponaša realnu situaciju. Primjer će biti malo drukčiji na način da će biti prikazan na način koji je lakše analizirati. Umjesto jako brzih poslova imati ćemo poslove koji traju nekoliko sekundi te prolaz vremena ćemo mjeriti u sekundama, gdje bi u realnoj situaciji vrijeme se mjerilo u otkucajima rada (*eng. tick*).

Bitno je ujedno i napomenuti vremenska ograničenja koja nastaju unutar RTOSa. Frekvencija otkucaja (*eng. tick*) je definirana unutar RTOS konfiguracijskih datoteka. Unutar FreeRTOSa ovo je definirano pomoću vrijednosti *configTICK_RATE_HZ* koja se kasnije ponovno koristi unutar određivanja vrijednosti *portTICK_PERIOD_MS* tako što se podijeli broj 1000 s vrijednosti od *configTICK_RATE_HZ*. Ukoliko postavimo vrijednost od *configTICK_RATE_HZ* na 500, tada dobivamo 1000/500 ili otkucaj svake dvije milisekunde. No problem nastaje ukoliko stavimo vrijednost *configTICK_RATE_HZ* veću od 1000. Tada vrijednost trajanja između otkucaja traje 0 milisekundi zbog toga što se vrijednosti zaokružuju. Ovo rezultira u greški u radu te neuspješnom izvođenju. Uzimajući to u obzir, najmanja brzina otkucaja je 1 milisekunda. Bez obzira koliko brzo se posao obavlja, ukoliko se koriste vremenska svojstva RTOSa, minimalno vrijeme će biti 1 milisekunda. Budući da je ovo potrebno za promjenu između poslova, korištenje UART priključka, pauziranje poslova te prekide. Bilo koja operacija koja traje manje od 1 milisekunde neće se primijetiti ranije budući da kao rezultat nemamo nikakvu interakciju sa sustavom. Postoje alternative za neke slučajeve, kao što je hardwareski prekid, no to ne rješava sve probleme, te ne možemo jednako precizno navesti vrijeme izvedbe prekida budući da to nije dio RTOS alata.

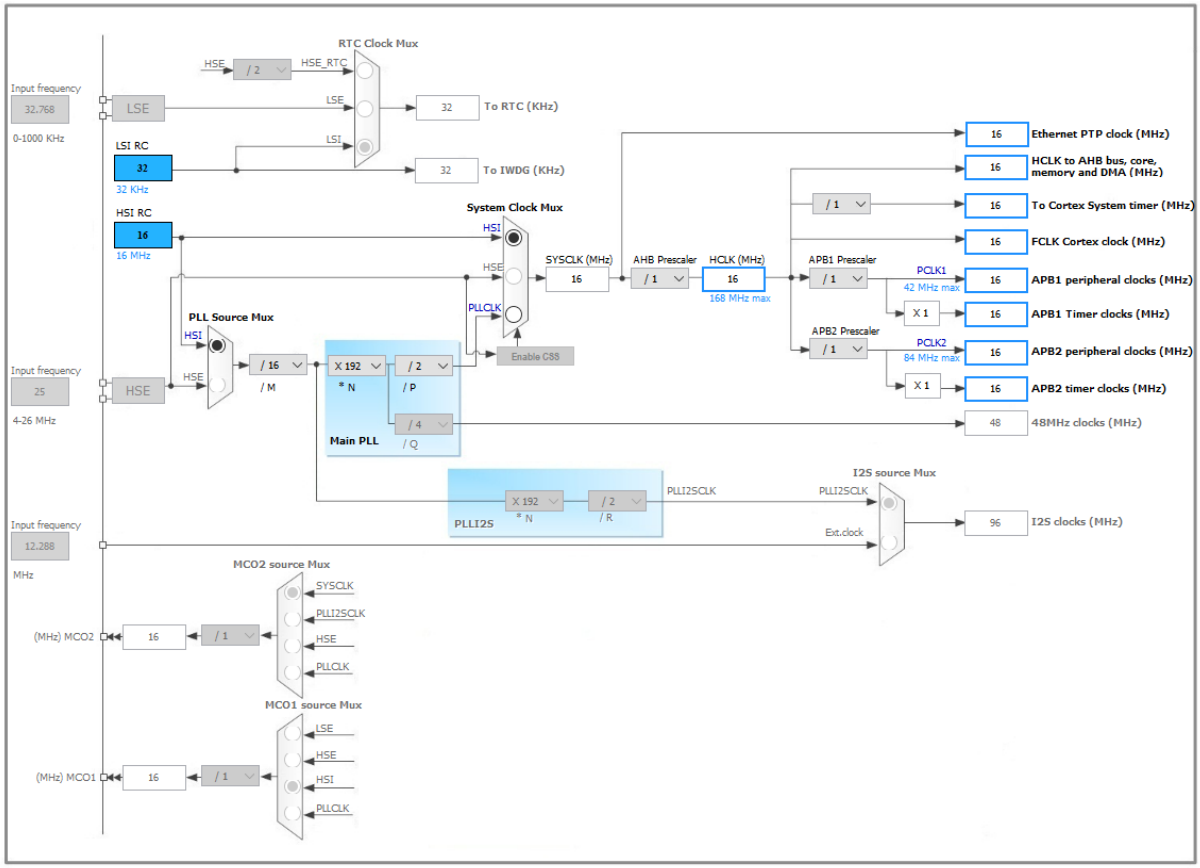
6.2.1. Postavljanje projekta

Kao što smo naveli, koristimo STM32CubeMX. Ovaj alat je namijenjen za STM32 mikorkontrolere te će nam olakšati implementaciju s mnogim dijelovima koje bi morali implementirati ručno. Za početak generiranje koda kroz STM32CubeMX će nam automatski implementirati HAL razvojni okvir u kod. HAL razvojni okvir služi za upravljanje hardwareom uređaja na način da olakšava korištenje komunikacije putem priključaka, korištenje sustavnog sata i drugih opcija. Za početak moramo odabrati mikrokontroler za koji ćemo koristiti. Budući da se radi o službenom STM32 proizvodu, STM32CubeMX sadrži sve STM32 mikrokontroler modele. U našem slučaju trebamo odabrati STM32F407VE budući da je to mikrokontroler s kojim ćemo raditi.



Slika 14 Izbornik za odabir mikrokontrolera [autorski rad]

Sljedeći korak je odrediti što želimo implementirati po pitanju hardwarea i, u našem slučaju, međusloja. Sve što se može implementirati u STM32CubeMXu se može naknadno implementirati u kodu, no na ovaj način uštedujemo na vremenu te izbjegavamo rizik od neispravnog podešavanja konfiguracije. Proći ćemo samo kroz bitne stavke. Za početak ćemo omogućiti FreeRTOS kao međusloj. Razlog zašto prvo omogućujemo FreeRTOS je zato što on ima određene zahtjeve za druge postavke kao što je postavka sistemskog sata. Za sistemski sat često možemo birati između SysTick opcije i TIM opcije. SysTick je jednostavan sistem koji se bazira na prekidima koji omogućavaju izmjenu konteksta u sustavu, no nije poželjan za održavanje vremena u okružjima s operacijskim sustavima. TIM, iako kompleksnije, omogućuje održavanje vremena u operacijskim sustavima, i samim time u FreeRTOSu, tako da ćemo odabrati TIM kao metoda držanja vremena. Daljnje upravljanje s radnim taktovima je moguće s velikom razinom kontrole. Svaka promjena koja se napravi u alatu će se ujedno primijeniti i u konfiguraciji u kodu nakon što se generira projekt. Početne postavke su prema dokumentaciji, što bi inače koristili za konfiguraciju koda ukoliko ne koristimo STM32CubeMX. U ovom alatu još možemo postaviti USART priključak koji će nam omogućiti serijsku komunikaciju s računalom. Svaki priključak se također može postaviti kao GPIO što omogućava korištenje priključka za slanje signala. Naravno u ovom slučaju moramo uzeti u obzir što sve želimo koristiti za naš projek. Budući da se svaki priključak može postaviti kao GPIO, možemo postaviti priključke koje podržavaju USART kao GPIO te time izgubiti USART priključak.



Slika 15 Izbornik za regulaciju rada mikrokontrolera [autorski rad]

Nakon što smo postavili sve kako smo htjeli možemo izgenerirati projekt. No prije nego što to napravimo moramo odabrati IDE koji ćemo koristiti za razvoj. IDE-evi koji su podržani su specifični za rad s ugrađenim sustavima. U našem slučaju ćemo odabrati Kiel uVision 5. Nakon toga možemo izgenerirati projekt koji će se pojaviti u prigodnom formatu.

6.2.2. Razvoj rješenja

Rješenje se bazira na zadacima različitih prioriteta. No prije nego li definiramo zadatke moramo definirati funkcije koji će zadatak pokrenuti. Zadatak mora biti nešto što možemo proučiti. Ovo zvuči očito no kada se radi s mikrokontrolerima često nemamo zaslon, dok operacije s komponentama kao LED diodama je teško bilježiti i uspoređivati. Kako bi prikazali rješenje, koristiti ćemo UART. UART (*eng. Universal asynchronus reciever-transmiter*) je komponenta koja služi za asinkronu serijsku komunikaciju. Ona se često koristi za prijenos podataka među

ugrađenim uređajima te pri samom programiranju istih. U našem slučaju koristiti ćemo UART za komunikaciju mikrokontrolera s računalom. Na ovaj način možemo prenositi poruke stanja poslova na mikrokontroleru te samim time možemo dobiti podatke za prikaz našeg rješenja. U našem rješenju možemo koristit UART pomoću HAL razvojnog okvira.

6.2.2.1. Poslovi

Za početak napraviti ćemo funkciju koja će služiti kao posao i slati će poruku putem UARTa kada task započinje s radom te kada završava s radom. Kako bi bolje predstavili primjer pravog rada dodati ćemo i vrijeme čekanja unutar funkcije. Ovaj posao ćemo nazvati posao broj 1.

```
void User_SendMessage_1(void *argument)
{
        uint16_t queued_time= a;
        uint8_t buffer_enqueue[128]="";
        sprintf(buffer_enqueue, "Task 1 Entered Queue: (t = %d, Priority = %d) \r\n",  queued_time, uxTaskPriorityGet( NULL ));
        taskENTER_CRITICAL();
        HAL_UART_Transmit(&huart1, buffer_enqueue, sizeof(buffer_enqueue), HAL_MAX_DELAY);
        taskEXIT_CRITICAL();
        while( xSemaphoreTake( xSemaphore, ( TickType_t ) 1000 ) != pdTRUE )
        {
          if (uxTaskPriorityGet( NULL ) < 5) {
            vTaskPrioritySet(NULL,uxTaskPriorityGet( NULL )+1);
          }
        }
          uint8_t buffer[128]="";
          sprintf(buffer, "Started Task 1: (t = %d, Priority = %d) \r\n",   queued_time, uxTaskPriorityGet( NULL ));

          uint8_t buffer_exit[128]="";
          sprintf(buffer_exit, "Finished Task 1: (t = %d)\r\n",   queued_time);



              taskENTER_CRITICAL();
              HAL_UART_Transmit(&huart1, buffer, sizeof(buffer), HAL_MAX_DELAY);
              taskEXIT_CRITICAL();
              osDelay(3000);

              taskENTER_CRITICAL();
              HAL_UART_Transmit(&huart1, buffer_exit, sizeof(buffer_exit), HAL_MAX_DELAY);
              taskEXIT_CRITICAL();
              xSemaphoreGive( xSemaphore );
              vTaskDelete(NULL);


}
```

Kod 1 Posao broj 1[autorski rad]

Napraviti ćemo još jedan posao koji je sličan poslu broj 1 koji će nam služiti kao posao većeg prioriteta. On će ujedno trajati kraće i nazvati ćemo ga posao broj 2. Bitno je napomenuti da svi

poslovi koji koriste UART koriste isti priključak kojeg smo postavili putem STMCubeMXa te da dva posla ne mogu istovremeno koristiti isti priključak. Također je bitno napomenuti da pri kraju svakog posla moramo dodati FreeRTOS funkciju *vTaskDelete* kako bi obrisali trenutni zadatak. U suprotnome možemo napuniti memoriju.

```
void User_SendMessage_2(void *argument)
{
	uint16_t queued_time= a;
	uint8_t buffer_enqueue[128]="";
	sprintf(buffer_enqueue, "Task 2 Entered Queue: (t = %d, Priority = %d) \r\n",  queued_time, uxTaskPriorityGet( NULL ));
	taskENTER_CRITICAL();
		HAL_UART_Transmit(&huart1, buffer_enqueue, sizeof(buffer_enqueue), HAL_MAX_DELAY);
		taskEXIT_CRITICAL();
	while( xSemaphoreTake( xSemaphore, ( TickType_t ) 1000 ) != pdTRUE )
	{
	  if (uxTaskPriorityGet( NULL ) < 5) {
	    vTaskPrioritySet(NULL,uxTaskPriorityGet( NULL )+1);
	  }
	}


	  char buffer[256]="";
	  sprintf(buffer, "Started Task 2: (t = %d, Priority = %d) \r\n",  queued_time, uxTaskPriorityGet( NULL ));

	  char buffer_exit[256]="";
	  sprintf(buffer_exit, "Finished Task 2: (t = %d)\r\n\r", queued_time);
	  taskENTER_CRITICAL();
	  HAL_UART_Transmit(&huart1, buffer, sizeof(buffer), HAL_MAX_DELAY);
	  taskEXIT_CRITICAL();


	  xSemaphoreGive( ySemaphore );
	  osDelay(2000);

	  taskENTER_CRITICAL();
	  HAL_UART_Transmit(&huart1, buffer_exit, sizeof(buffer_exit), HAL_MAX_DELAY);
	  taskEXIT_CRITICAL();
	  xSemaphoreGive( xSemaphore );

	  vTaskDelete(NULL);
}
```

Kod 2 Posao broj 2[autorski rad]

Tome ćemo još dodati posao broj 3. Posao broj 3 će biti poseban po tome što će imati mogućnost prekida rada drugih procesa. Specifičnije, koristiti ćemo ovaj posao kao primjer posla koji se izvodi kao posljedica ISR-a. Budući da je posljedica poziva prekida, ovaj posao je razine prioriteta definiranog pomoću konfiguracijske varijable *configTIMER_TASK_PRIORITY*.

```
void User_SendMessage_3(TimerHandle_t pxTimer)
{
	BaseType_t xHigherPriorityTaskWoken = pdFALSE;

	  char buffer[128]="";
	  sprintf(buffer, "Started Task 3: (t = %d, Interupt)\r\n",  (int*) a);
	  char buffer_exit[128]="";
	  sprintf(buffer_exit, "Finished Task 3: (t = %d)\r\n",  (int*) a);
	      taskENTER_CRITICAL();
	      HAL_UART_Transmit(&huart1, buffer, sizeof(buffer), HAL_MAX_DELAY);
	      taskEXIT_CRITICAL();
	      HAL_Delay(8000);
	      taskENTER_CRITICAL();
	      HAL_UART_Transmit(&huart1, buffer_exit, sizeof(buffer_exit), HAL_MAX_DELAY);
	      taskEXIT_CRITICAL();
	      xTimerResetFromISR( xSysMonTimer, &xHigherPriorityTaskWoken );

}
```

Kod 3 Posao broj 3[autorski rad]

Naposljetku ćemo dodati još jedan posao koji ćemo nazvati posao broj 4 ili brojač. Posao brojača je da ispisuje trenutno vremensko stanje od početka rada programa. Pomoću njega ćemo znati koliko vremena prolazi dok se obavljaju procesi. Bitno je napomenuti da ćemo ovaj posao pokrenuti samo jednom pri čemu će on kontinuirano raditi u petlji. Također, ovaj posao zauzima najveći prioritet i ne koriste semafor. Na ovaj način ostvarujemo kontinuirani ispis vremena unatoč drugih poslova koji se izvode.

```
void User_SendMessage_timer(int *argument){
      char filler[]="--------------------------------------------------\r\n";
      for(;;)
      {
        while( xSemaphoreTake( ySemaphore, ( TickType_t ) 1000 ) != pdTRUE )
        {

        }
        vTaskSuspendAll ();
        taskENTER_CRITICAL();
        HAL_UART_Transmit(&huart1, filler, sizeof(filler), HAL_MAX_DELAY);
        taskEXIT_CRITICAL();
        char buffer[128];
        a++;
        sprintf(buffer, "t= %d\r\n", (int*) a);
        taskENTER_CRITICAL();
        HAL_UART_Transmit(&huart1, buffer, sizeof(buffer) , HAL_MAX_DELAY);
        taskEXIT_CRITICAL();

        xTaskResumeAll ();
        xSemaphoreGive( ySemaphore );
        osDelay(1000);
      }
}
```

Kod 4 Posao broj 4 (Brojač) [autorski rad]

Ove poslove ćemo pozvati u poslu koji kontinuirano radi u petlji te u rasponima poziva poslove broj 1, 2 i 3, te jednom poziva posla broj 4, tj. brojača. Taj posao ćemo nazvati posao broj 5. Vrijeme poziva posla broj 1 i 2 je različit od posla broja 3 kako bi dobili bolji prikaz. U suprotnom posao broj 3 će češće koristiti procesor nego što želimo i neće omogućiti drugim poslovima korištenje procesora.

```
void User_TaskLoop(void const * argument)
{
        xSemaphore = xSemaphoreCreateBinary();
        xSemaphoreGive( xSemaphore );

        ySemaphore = xSemaphoreCreateBinary();
        xSemaphoreGive( ySemaphore );



        xTaskCreate(User_SendMessage_timer, "Task_Timer", 256, ( void * ) &a, 7,  NULL);

        while(1)
        {
           for(uint8_t i=0;i<3;i++)
           {
                 xTaskCreate(User_SendMessage_1, "Task_1", 256, ( void * ) &a, 1,  &xHandle1);
                 xTaskCreate(User_SendMessage_2, "Task_2", 256, ( void * ) &a, 2,  &xHandle2);
                 osDelay(3000);


           }
           osDelay(10000);
        }
}
```

Kod 5 Posao broj 5 [autorski rad]

Poslovi se kreiraju pomoću funkcije xTaskCreate u koju stavljamo šest argumenata. Prvi argument je funkcija koju posao obavlja, drugi argument je naziv posla, treći argument predstavlja broj riječi koje će se alocirati za korištenje stogu posla, četvrti argument je argument koji se prosljeđuje u funkciju posla, peti argument je argument koji predstavlja prioritet posla. Zadnji argument je *handle* ili referenca na kreirani posao. Nju možemo koristiti kako bi referencirali posao te njime upravljali van samog posla.

6.2.2.2. Planer

Sada kada smo postavili poslove možemo započeti s korištenjem FreeRTOSa. Možemo započeti s pokretanjem planera zadataka. U FreeRTOSu to se planer se pokreće pomoću funkcije *vTaskStartScheduler*. Nakon što se pozove funkcija planer počinje i obavljaju se svi poslovi koji su bili kreirani prije te se u planer stavljaju novi poslovi kako oni dolaze. Planer radi na *Round Robin* algoritmu. To znači da se poslovi kreću u ciklusu pri čemu svaki posao dobiva vremenski isječak (*eng. Time slice*) prilikom kojeg obavlja svoj posao. Nakon što vrijeme vremenskog

isječka prođe, posao vraća kontrolu nad procesorom planeru koji prosljeđuje kontrolu sljedećem poslu. Planer se ponaša ovisno o konfiguraciji. Ukoliko postavimo stavku *configUSE_PREEMPTION* na vrijednost 1 unutar konfiguracijske datoteke, planer će uvijek pokretati posao najvećeg prioriteta, te izmjenjivati između poslova istih prioriteta. Ukoliko stavimo tu istu stavku na vrijednost 0 izmjenjivati će se poslovi različitih prioriteta. Ovo naravno ne znači nužno da će vladati kaos u sustavu, već da kontrolu nad poslovima možemo obaviti na drukčiji način. Mnoge druge stavke se mogu promijeniti u konfiguracijskoj datoteci kao što je veličina hrpe koja drži zadatke, maksimalan broj prioriteta, korištenje mutexa te mnoge druge.

6.2.2.3. Kontroliranje i zaključavanje poslova

Najbitniji dio RTOS-a je kontrola poslova budući da je to način na koji ćemo osigurati da se poslovi obavljaju u pravom vremenu. No prije nego što dođemo do toga prvo moramo napraviti stvari vezane za prikaz našeg rješenja. Za početak želimo da se samo jedan posao obavlja u danom trenutku. Ovo se odnosi samo na poslove broj 1, 2 i 3. Posao broj 4 (brojač) i 5 će biti pozadinski poslovi te ćemo njima dopuštati rad kako bi bolje prikazali rješenje.

Način na koji možemo osigurati da se samo jedan posao obavlja u danom trenutku ovisi o našim potrebama. U našem slučaju koristiti ćemo binarne semafore. U FreeRTOSu semafori se mogu pozvati pomoću funkcije xSemaphoreCreateBinary pri čemu možemo koristiti xSemaphoreGive za davanje semafora te xSemaphoreTake za preuzimanje semafora. Za početak želimo inicijalizirati semafor te predati ga nakon čega ga možemo koristiti.

```
xSemaphore = xSemaphoreCreateBinary();
xSemaphoreGive( xSemaphore );
```

Kod 6 Inicijalizacija semafora [autorski rad]

Nakon što smo inicijalizirali semafor te predali ga, možemo ga implementirati. Način na koji to radimo je kreiranje beskonačne petlje u poslu gdje stavljamo preuzimanje semafora. Funkcija *xSemaphoreTake* ima dva argumenta. Prvi je semafor koji se uzima, dok drugi je vrijeme koje se čeka prije nego li se nastavi program, ili u ovom slučaju posao. Ukoliko se semafor uspije preuzeti, funkcija vraća *pdTRUE* nakon čega posao nastavlja sa svojim radom. Bitno je napomenuti da prilikom čekanja posao predaje kontrolu nazad planeru, što sprječava da se program svede na posao višeg prioriteta u beskonačnoj petlji nastaloj zbog deadlocka.

```
while( xSemaphoreTake( xSemaphore, ( TickType_t ) 1000 ) != pdTRUE )
{
}
```

Kod 7 preuzimanje semafora [autorski rad]

No problem koji sada može nastati je taj da redanje poslova višeg prioriteta može u potpunosti preuzeti procesor. U našem slučaju posao broj 1 je manjeg prioriteta od posla broja 2. To znači da može doći do slučaja gdje planer konstantno daje procesor poslu broj 2 nad poslom broj 1. U tom slučaj posao broj 1 nikada neće doći na vrijeme. Pored činjenice da trebamo obaviti svaki posao, trebali bi obaviti svaki posao u nekom roku. Tu nam može pomoći postepeno povećavanje prioriteta. Ukoliko stavimo da se nakon nekog vremena prioritet posla poveća možemo mu omogućiti da preuzme semafor ovisno o tome koliko je dugo čekao u redu. U FreeRTOSu funkcija za postavljanje prioriteta je *vTaskPrioritySet*, dok funkcija za dohvaćanje prioriteta posla je *uxTaskPriorityGet*. Naravno ovo ima i svoje nedostatke. Kako bi poslovi niskih prioriteta mogli raditi potrebno je da poslovi viših prioriteta predaju kontrolu nad procesorom, tj. da se pauziraju. U našem slučaju to je situacija no ukoliko se to ne događa potrebno je prioritete poslova u redu čekanja povećavati putem njihovih referenci, tj. *handlera*. U našem slučaju ograničiti ćemo razinu prioriteta na 5, što vrijedi za oba posla. Ograničenje je tu budući da imamo određeni broj prioriteta koji možemo imati u jednom trenutku u programu. Taj broj se može povećati u konfiguracijskoj datoteci pomoću stavke *configMAX_PRIORITIES*. Na ovaj način ostvarujemo takozvani mekani rok (*eng. Soft deadline*), tj. rok koji želimo postići al nije kritično da obavi točno na vrijeme koje želimo. Ovom slučaju želimo da se posao obavi u razumno vrijeme, no želimo zadržati i raspored našeg sustava i izvođenje po jednog posla u trenutku.

```
while( xSemaphoreTake( xSemaphore, ( TickType_t ) 1000 ) != pdTRUE )
{
  if (uxTaskPriorityGet( NULL ) < 5) {
    vTaskPrioritySet(NULL,uxTaskPriorityGet( NULL )+1);
  }
}
```

Kod 8 Povećavanje razine prioriteta [autorski rad]

Postoje alternative za kontroliranje poslova kao što je funkcija *taskYIELD*. Funckija *taskYIELD* predaje kontrolu nad procesorom nazad u planer pri čemu se odabire posao s najvećim prioritetom za daljnje izvođenje. Ova funkcija se koristi ukoliko je stavka

configUSE_PREEMPTION postavljena na vrijednost 0, tj. ukoliko se poslovi obavljaju neovisno o prioritetima. Ukoliko nema posla većeg prioriteta planer ponovno bira posao koji je pozvao funkciju *taskYield*.

Još jedna alternativa koja može koristit su funkcije su *vTaskDelay* i *vTaskDelayUntil*. Funkcija *vTaskDelay* odgađa rad posla dano vrijeme, dok funkcija *vTaskDelayUntil* odgađa posao do određenog trenutka. Vrijeme čekanja je u realnom vremenu. Ovo je bitno budući da s time možemo koristit *vTaskDelayUntil* kao mehanizam za obavljanje poslova do najkasnijeg vremena (*eng. deadlinea*). Naravno budući da koristimo semafore za ograničavanje broja poslova koji se mogu obavljati u određenom vremenu, ovo nije nužno najbolje rješenje za nas.

6.2.2.4. Prekidi

Prekidi su pozivi koji prekidaju rad poslova koji se trenutno izvode kako bi se obavio ISR. Najčešće ovakvi pozivi dolaze od strane hardware kroz omogućene kanale koji posjeduju svojstvo prekida. U tom slučaju procesor prekida rad svih poslova kako bi se izveo ISR. U našem slučaju prekid se vrši softwareski na razini međusloja, tj. na razini FreeRTOSa. Funkcija koja se koristi za inicijalizaciju prekida je *xTimerCreate*. Ova funkcija omogućava inicijalizaciju prekida koji se obavlja nakon određenog perioda te omogućuje ponovno postavljanje nakon svakog izvršavanja. Također je bitno napomenuti da svaka funkcija koja radi s FreeRTOSom i koja je pozvana iz prekida mora biti specifična funkcija koja radi unutar ISRa. U našem slučaju to je ponovno postavljanje brojača do sljedećeg prekida. U normalnoj situaciji to bi bila funkcija *xTimerReset*, dok u situaciji gdje se takva funkcija poziva iz ISRa, poziva se funkcija *xTimerResetFromISR*. Ovo je bitno napomenuti budući da standardne funkcije će izazvati rušenje programa. Na ovaj način ostvarujemo tvrdi rok (*eng. Hard deadline*), tj. želimo da se posao obavi u točno određeni trenutak. Na ovaj način možemo izvesti poslove koje želimo kada je to najbitnije.

Prekid koji koristimo inicijalizira se prije početka planera poslova. ISR handler je funkcija, a u našem slučaju je to posao broj 3. U praksi ISR handler je generalno funkcija koja ne traje dugo te je cilj da se izvede što brže. Zbog načina na koji radi naš sustav, staviti ćemo da je handler funkcija posla broj 3. Bitno je napomenuti da prekid posjeduje razinu prioriteta 7 što je jednako s poslom broj 4. Na ovaj način osiguravamo da poslovi broj 1 i 2 ne smetaju pri radu posla 3 i 4.

6.2.2.5. Kritični odsječak

Kritični odsječak služi kako bi osigurali sigurno iskorištavanje resursa bez ometanja od strane drugih poslova. Između poslova koji mogu biti istog prioriteta, kao što su poslovi broj 1 i 2, mi možemo osigurati sigurno korištenje određenog resursa bez ometanja. No budući da imamo poslove kao što su posao broj 3 koji se izvodi kao rezultat prekida, te posao broj 4 koji se izvodi u pozadini kako bi održavao vrijeme, potrebno je koristiti kritične odsječke. Dok u kritičnom odsječku, kontrola ne prelazi s procesa koji se izvodi što onemogućuje prekide i paralelno izvođenje.

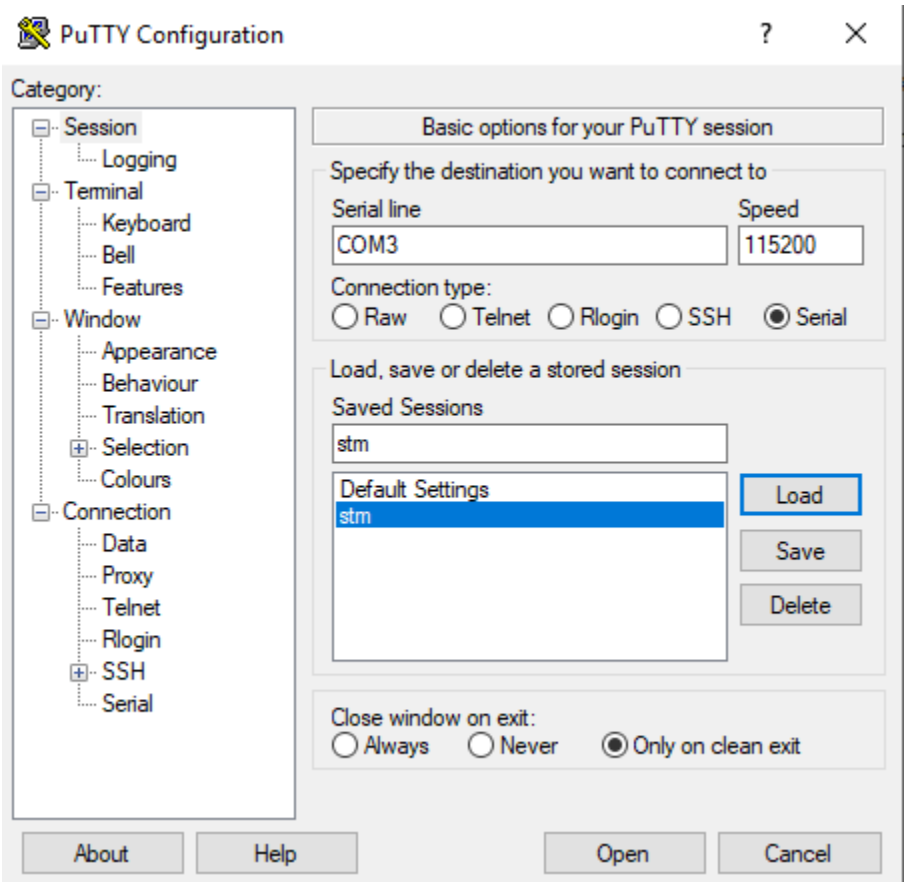
U našem slučaju posao resurs koji ne želimo dijeliti je UART priključak. Ukoliko dođe do prekida za vrijeme korištenja UARTa, poruke se više neće slati. Zbog toga moramo biti sigurni da se svaki posao koji koristi UART uspješno izvede. Ovo ćemo učiniti tako što ćemo pozvati funkciju *taskENTER_CRITICAL* prije nego li pozovemo funkciju za slanje poruke putem UARTa (*HAL_UART_Transmit*). Nakon što smo poslali poruku izlazimo iz kritičnog odsječka pomoću poziva funkcije *taskEXIT_CRITICAL*.

Postoji alternativa za ulazak u kritični odsječak pomoću funkcije *vTaskSuspendAll*. Nakon što obavimo posao iz kritičnog odsječka možemo izaći pomoću poziva funkcije *vTaskResumeAll*. Za razliku od prijašnje metode, ovakav ulazak u kritični odsječak može biti prekinut pomoću ISRa te samim time nije dovoljan za korištenje u slučaju osiguravanja UARTa. Ovaj pristup možemo koristiti u slučajevima gdje želimo osigurati neometanu izvedbu posla prilikom paralelnog izvođenja poslova s mogućnošću izvedbe prekida u slučaju da imamo kritični posao koji se ne smije ignorirati.

6.2.2.6. Ispis

Kao što smo naveli, cijeli sustav komunicira putem UARTa i na taj način dobivamo ispis. Računalo koje je povezano s mikrokontrolerom prima poruke od strane mikrokontrolera i to su poruke koje smo mi poslali koristeći UART. Kako bi prikazali te poruke koristiti ćemo alat Putty. Prilikom podešavanja komunikacije bitno je odabrati *Serial* opciju te unijeti naziv komunikacijskog porta koji koristimo za komunikaciju s mikrokontrolerom. U našem slučaju to je COM3 koji predstavlja USB u UART most. Također je potrebno odabrati brzinu modulacija u sekundi. Ovo je ovisno o uređaju s kojem radimo te za većinu je potrebno pogledati preporučenu

brzinu unutar popratne dokumentacije i priručnika mikrokontrolera. Nakon što smo sve to učinili, možemo otvoriti vezu između računala i mikrokontrolera.



Slika 16 Putty povezivanje s mikrokontrolerom [autorski rad]

Prilikom rada ispisuje nam se trenutno vrijeme označeno sa slovom „t“, koje nastaje kao rezultat posla broj 4. Posao koji ulazi u red čekanja za izvođenje, vrijeme kad je posao ušao u red čekanja (ovo se ujedno koristi i kao identifikacijski broj, budući da u red čekanja svake sekunde ne ulazi više od jednog posla iste vrste), te prioritet koji taj posao ima u tom trenutku. Ujedno vidimo i kada posao dolazi na red za izvedbu te mu se također prikazuju vrijeme kada je posao ušao u red čekanja te prioritet koji imaju u ovom trenutku. Naposljetku imamo poruku koja predstavlja završetak posla i ona u sebi sadrži vrijeme kada je posao došao u red čekanja.

```
t= 1
Task 2 Entered Queue: (t = 1, Priority = 2)
Started Task 2: (t = 1, Priority = 2)
Task 1 Entered Queue: (t = 1, Priority = 1)
-----------------------------------------------------
t= 2
-----------------------------------------------------
t= 3
Finished Task 2: (t = 1)
Started Task 1: (t = 1, Priority = 2)
Task 2 Entered Queue: (t = 3, Priority = 2)
Task 1 Entered Queue: (t = 3, Priority = 1)
-----------------------------------------------------
t= 4
-----------------------------------------------------
t= 5
Finished Task 1: (t = 1)
Started Task 2: (t = 3, Priority = 4)
```

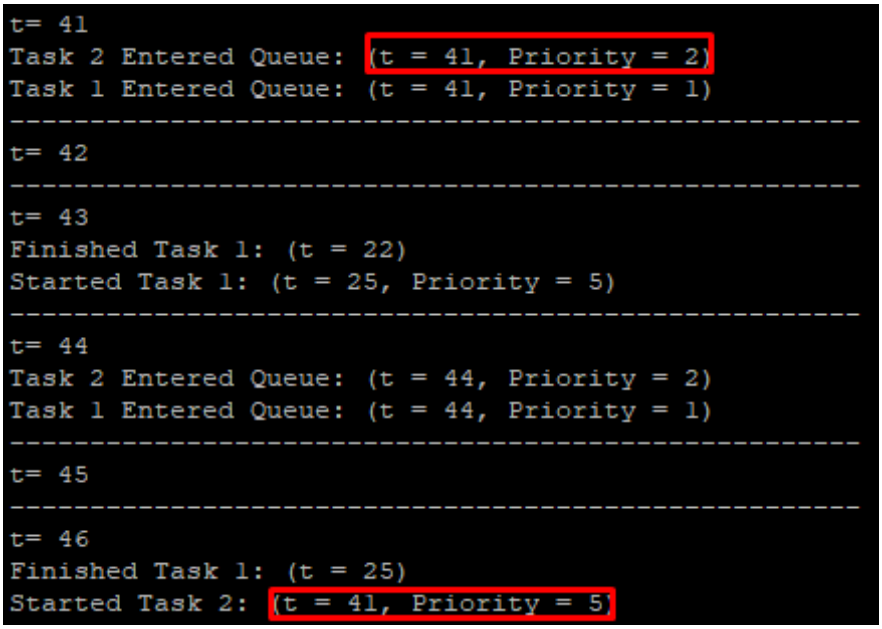
Slika 17 Ispis rada [autorski rad]

Na ovaj način možemo vidjeti događaje kao što su prekidi koji sprečavaju rad drugih poslova. Prilikom prekida posao 3 pokazuje vrijeme kada je prekid izvršen. U ovom slučaju posao 3 traje 8 sekundi za izvedbu i prekid se poziva svakih 12 sekundi.

```
t= 10
Finished Task 1: (t = 3)
Started Task 2: (t = 6, Priority = 5)
-----------------------------------------------------
t= 11
-----------------------------------------------------
t= 12
Started Task 3: (t = 12, Interupt)
-----------------------------------------------------
t= 13
-----------------------------------------------------
t= 14
-----------------------------------------------------
t= 15
-----------------------------------------------------
t= 16
-----------------------------------------------------
t= 17
-----------------------------------------------------
t= 18
-----------------------------------------------------
t= 19
-----------------------------------------------------
t= 20
Finished Task 3: (t = 12)
Finished Task 2: (t = 6)
Started Task 1: (t = 6, Priority = 5)
```

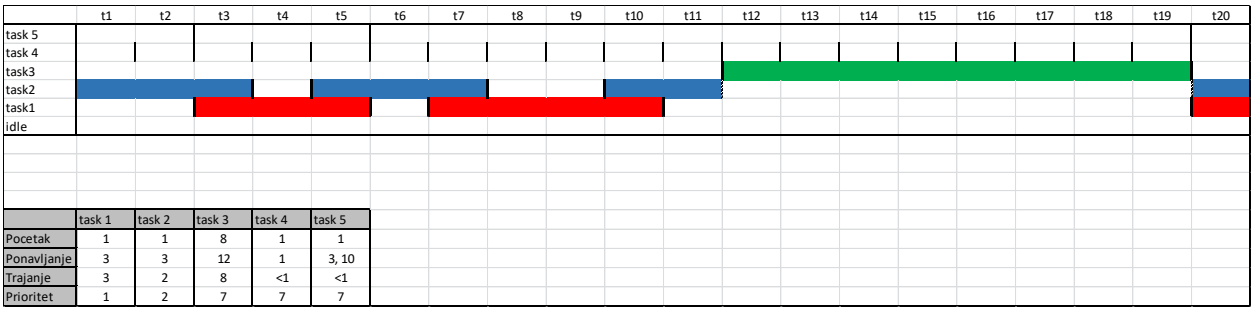
Slika 18 Prikaz prekida [autorski rad]

Druga stvar koju možemo zapaziti je postepeno povećavanje prioriteta poslova koji se zadržavaju u redu za čekanje. Ovo se odnosi na poslove 1 i 2, koji počinju s prioritetima razine 1 i 2. Maksimalan prioritet koji mogu dosegnuti je prioritet razine 5. Na ovaj način osiguravamo obavljanje svakog posla koji uđe u red za čekanje s time da nećemo imati veći prioritet od posla broj 3, koji predstavlja prekide, i posla broj 4, koji nam govori trenutno vrijeme.



Slika 19 Prikaz povećavanja razine prioriteta [autorski rad]

Rezultat je kontinuirani rad sustava gdje poslovi ulaze u red čekanja te počinju s radom kada dođu na red i ukoliko izvrše prekid ili rade paralelno. To možemo prikazati kroz graf gdje vidimo kada koji posao počinje s radom. Bitno je napomenuti da u ovom slučaju vremenske oznake označavaju sekunde, ne standardne vremenske isječke u procesoru koji se uobičajeno koriste. Zbog toga imamo situaciju gdje se pojavljuju dva posla u istom trenutku te poslovi koji se čine da traju duže ili kraće nego li je navedeno. Razlog tome je da paralelni poslovi mogu preuzet kontrolu nad procesorom pri čemu neki poslovi mogu započeti i završiti u sredini svake sekunde.



	task 1	task 2	task 3	task 4	task 5
Pocetak	1	1	8	1	1
Ponavljanje	3	3	12	1	3, 10
Trajanje	3	2	8	<1	<1
Prioritet	1	2	7	7	7

Slika 20 Tablični prikaz rada [autorski rad]

7. Zaključak

Operacijski sustavi u realnom vremenu igraju veliku ulogu u današnjoj tehnologiji. Premda je ovo tehnologija koja postoji godinama, mnogi problemi koji su postojali prije postoje i danas te ne postoji nužno jedno najbolje rješenje za svaki problem. Postoji razni sustavi s raznim potrebama i različitim specifikacijama. Algoritmi koji rješavaju jedan problem mogu prouzrokovati probleme u drugim situacijama. Paralelno izvođenje poslova, iako modernije rješenje, može prouzrokovati više problema od standardne, sekvencijalne, izvedbe. Ugrađeni sustavi, kao i mnogi drugi, imaju određene zahtjeve. Samo zato što se radi o ugrađenom sustavu ne znači da nam je potreban RTOS. Većina takvih uređaja, danas i sutra, operiraju i operirati će na jednostavnim programima koji obavljaju jednu funkciju na beskonačnoj petlji. Razlog tome je da ugrađeni sustavi bolje rade što su jednostavniji, budući da tako smanjujemo vjerojatnost da će nešto poći po zlu.

Također postoji velik broj operacijskih sustava koji su specijalizirani za razne situacije, bilo to za razvoj sustava unutar vozila, mrežnih uređaja, mobilnih uređaja ili nečeg drugoga. Postoje sustavi koji se specijaliziraju u području za mrežne alate dok neki drugi mogu biti specijalizirani u području ulazni i izlaznih sučelja. Ti određeni sustavi dolaze s alatima potrebnim za obavljanje potrebnog posla te mogu skratiti vrijeme razvoja sustava. Naravno ukoliko situacija nalaže, moguće je napraviti i vlastiti RTOS, no to može rezultirati u skupom i dugotrajnom razvoju, budući da RTOS sustavi, osim što rješavaju problem planiranja zadataka, obavljaju poslove kao što su ispravno upravljanje memorijom. Primjer situacije u kojoj je potrebno razviti vlastiti RTOS je ukoliko koristimo vlastite mikrokontrolere, budući da komercijalni RTOS su napravljeni za poznate proizvođače mikrokontrolera, no ova situacija je poprilično rijetka.

Budući da RTOS se radi za ugrađene sustave moramo odlučiti se i o mikrokontroleru koji ćemo koristiti. U većini slučaja mikrokontroler se odabire prije nego li se odabere RTOS, no to nije nužno u slučajevima ako imamo licencu za određeni sustav ili tražimo sustav sa specifičnim svojstvima. Pri izboru moramo uzeti u obzir koji sustav podržava koje mikrokontrolere. U slučaju samostalne izrade RTOSa moramo uzeti u obzir specifikacije mikrokontrolera. Moramo znati koji su priključci namijenjeni kojoj svrsi te adrese u memoriji koje smijemo zauzimati. Pogreške u radu s memorijom mogu rezultirati u trajnim oštećenjima uređaja.

Naposljetku moramo uzeti u obzir poslove koje želimo obavljati i način na koji želimo te poslove obaviti unutar našeg sustava. Kao što smo vidjeli u našem primjeru. Poslovi mogu biti raznoliki i način na koji možemo riješiti njihovu implementaciju može utjecati na sveukupni sustav. Želimo li koristiti paralelno izvođenje poslova gdje koristimo vremenske isječke za svaki posao u *Round robin* algoritmu koji može rezultirati dužim obavljanjem svakog posla, ili želimo li obavljati jedan posao u trenutku, kako bi češće izvršavali posao po cijeni zadržavanja drugih poslova u redu za čekanje? Želimo li imati fiksne prioritete poslova i time osigurati izvođenje poslova viših prioriteta ili želimo li imati rastuće prioritete gdje osiguravamo izvođenje poslova nižih prioriteta umjesto poslova koje smatramo bitnijima kako bi izbjegli vječno zadržavanje poslova u redu za čekanje? Želimo li koristiti *soft deadline* kako bi osigurali jednostavniji rad sustava ili *hard deadline* kako bi osigurali izvođenje posla u točno određeno vrijeme?

Svaki sustav je moguće obaviti na više načina i, kao što smo rekli, ne postoji nužno najbolje rješenje. Uzimajući ovo u obzir, i dalje je bitno napomenuti da sustavi ove vrste imaju jako bitan posao, i razlika između jednog dizajna i drugog može biti katastrofalna i snositi kobne posljedice. No ukoliko se napravi sustav koji je pouzdan, rezultati mogu pomaknuti granice obavljanja poslova u mnogim područjima koje prije je mogao samo čovjek obavljati ili poslove koje su uopće nisu mogli obavljati.

8. Literatura

[1] Java ugrađeni sutavi, Oracle, [Slika] (bez dat.) https://docs.oracle.com/javase/8/embedded/develop-apps-platforms/overview.htm. [Pokušaj pristupa 16. 1. 2020.].

[2] A. S. Tanenbaum, „Modern Operating Systems“, Harlow: Pearson Prentice Hall, 2009.

[3] B. Leo, G. Marin, J. Domagoj i J. Leonardo, „Operacijski sustavi“, Zagreb: Element d.o.o., 2010.

[4] S. William, „Operating systems: internals and design principles“, New Jersey: Prentice Hall, 1992.

[5] N. Tammy, „Embedded systems architecture: a comprehensive guide for engineers and programmers“, Newnes, 2013.

[6] Ugrađeni sustav, [Slika] (bez dat.) https://www.winsystems.com/winsystems-introduces-pico-itx-single-board-computer-with-ideal-functionality-for-embedded-industrial-iot-applications/. [Pokušaj pristupa 16. 1. 2020.].

[7] Science Direct, Embedded system model, https://www.sciencedirect.com/topics/engineering/embedded-system-model. [Pokušaj pristupa 16. 1. 2020.].

[8] M. Barr i A. Massa, „Programming Embedded Systems“, Sebastopol: O'Reilly, 2006.

[9] K. Yaghmour, J. Masters, G. Ben-Yossef i P. Gerum, „Building Embedded Linux Systems“, Sebastopol: O'Reilly Media, 2008.

[10] I. C. Bertolotti i G. Manduchi, „Real-Time Embedded Systems: Open-Source Operating Systems Perspective“, Boca Raton: CRC Press, 2012.

9. Popis slika