SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2245

ALGORITMI RASPOREĐIVANJA ZADATAKA U OPERACIJSKIM SUSTAVIMA ZA RAD U STVARNOM VREMENU

Marin Bralić

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2245

ALGORITMI RASPOREĐIVANJA ZADATAKA U OPERACIJSKIM SUSTAVIMA ZA RAD U STVARNOM VREMENU

Marin Bralić

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 13. ožujka 2020.

DIPLOMSKI ZADATAK br. 2245

Pristupnik:	**Marin Bralić (0036481753)**
Studij:	Računarstvo
Profil:	Računalno inženjerstvo
Mentor:	prof. dr. sc. Davor Petrinović

Zadatak: **Algoritmi raspoređivanja zadataka u operacijskim sustavima za rad u stvarnom vremenu**

Opis zadatka:

Za diplomski rad potrebno je implementirati jednostavnu jezgru operacijskog sustava za rad u stvarnom vremenu s osnovnim funkcijama te nekoliko algoritama raspoređivanja zadataka. U operacijskim sustavima za rad u stvarnom vremenu (engl. real-time systems), da bi mogli reći da rade ispravno, potrebno je osim logičke ispravnosti, zadovoljiti i vremensku točnost svih zadataka, posebno ako se radi o sustavu sa striktnim vremenskim ograničenjima (engl. hard real time). U slučaju višezadaćnog rada, raspoređivanje zadataka zahtljeva algoritme koji će što efikasnije iskoristiti procesorsko vrijeme te zadovoljiti vremenske zahtjeve svih zadataka. Zbog toga se ovisno o zahtjevima koriste različiti algoritmi, a nekada i više njih istovremeno. Za izvedbu sustava potrebno je koristiti računalo Raspberry Pi. Potrebno je izmjeriti i analizirati vremenska svojstva operacijskog sustava te ponašanje i odzive uz različita opterećenja i primjenu različitih algoritama raspoređivanja.

Rok za predaju rada: 30. lipnja 2020.

Sadržaj

1. Uvod

Sustavi za rad u stvarnom vremenu su sustavi koji osim logičke ispravnosti moraju zadovoljiti i sva vremenska ograničenja svih dodijeljenih poslova. Svi računalni sustavi obavljaju svoj posao u nekom vremenu, a za računala koja ljudi koriste najčešće vrijeme nije nešto kritično, niti od velike važnosti. Iako naravno svi želimo da računala obave svoj posao u što kraćem vremenu. Za primjer možemo uzeti nekoliko različitih poslova koje računalo mora obaviti. Kao prvi primjer uzet ćemo pisanje dokumenta u nekom od programa te njegovo spremanje na disk. Korisnik unosi podatke putem tipkovnice te po završetku sprema unesene podatke na disk. Ako pritiskom na neku tipku računalo „zakasni“ te prikaže slovo na zaslonu par milisekundi kasnije ili možda sekundu kasnije, korisnik to najčešće neće niti primijetiti. Čak i da primijetimo to nam neće puno smetati, kao niti vrijeme koje će računalu biti potrebno da taj dokument spremi na disk. Dakle ovdje se radi o poslu za koji vrijeme nije od presudne važnosti i bez obzira na brzinu računala, možemo reći da sustav radi dobro, iako naravno korisnik želi da poslovi koji dulje traju računalo obavi što prije. Kao drugi primjer uzet ćemo reprodukciju zvuka. Jednom kada pokrenemo reprodukciju računalo mora tijekom cijelog vremena dohvaćati podatke s diska, dekodirati ih te slati na digitalno-analogni konverter koji generira zvučni signal. Vidimo da u ovakvom sustavu računalo ipak mora neke poslove obavljati u zadanom vremenu, kao što je dekodiranje podataka i slanje na izlaznu jedinicu (audio sustav) kako bi zvuk bio u toku bez prekidanja. Ako dođe do kratkotrajnih zastoja koji su lako mogući zbog npr. kratkotrajnih preopterećenja, korisnik će te smetnje čuti i neće biti dobro ako se to često dešava. Sve dok su te smetnje i vremenski ispadi rijetki i kratkotrajni, za sustav možemo reći da radi dobro. Ukratko, vrijeme je bitno za ispravan rad, ali male i rijetke ispade možemo tolerirati. Kao treći primjer uzeti neki sigurnosni kontrolni sustav unutar aviona. Za ovaj sustav možemo reći da je jako bitno da bude raspoloživ u svakom trenutku i to s vrlo brzim odzivom. Samo jedan zastoj ovakvog sustava u krivom trenutku može dovesti do težih posljedica. Na temelju ova tri primjera možemo podijeliti sve računalne sustave u tri kategorije: sustav koji ne rade u

stvarnom vremenu (prvi primjer), sustavi koji rade u stvarnom vremenu ali nisu kritični (drugi primjer) i oni koji jesu kritični (treći primjer). U praksi RTS (engl. Real Time Systems, sustavi za rad u stvarnom vremenu) se dijele u 3 kategorije:

Strogi sustavi

Strogi RTS imaju vrlo striktno zadana vremena početka, trajanja i završetka pojedinih zadataka, a svaki vremenski ispad može uzrokovati velike štete. Kao primjere ovakvih sustava možemo navesti računala u avionima, medicinska oprema, kontrolni sustavi u tvornicama itd.

Čvrsti sustavi

Čvrsti RTS imaju također zadane stroge vremenske rokove, ali ispadi koji se dogode neće uzrokovati veliku štetu, već će samo narušiti kvalitetu rada sustava. Kao primjer možemo navesti razne multimedijske uređaje: kamere, mobiteli, televizori itd.

Ublaženi sustavi

Ublaženi RTS prihvaćaju kašnjenja za određene poslove, ali i dalje postoji vremenski raspon (tolerancija) unutar kojeg bi bilo dobro da posao bude obavljen. Ako i dođe do ispada iz vremenskog intervala, to se može tolerirati sve dok ta kašnjenja nisu česta i prevelika. Kao primjere možemo navesti: sustav internetskih transakcija, bankomati itd.

Složeniji RTS obavljaju istovremeno više poslova te svaki od njih ima svoje zahtjeve i vremenska ograničenja. Takve probleme rješavamo pomoću višezadaćnog rada koji omogućuje da lakše ostvarimo upravljanje svim poslovima. Implementacija složenih sustava bez toga bi bila vrlo komplicirana, a za neke probleme gotovo neizvediva. Zbog toga se poslovi razdjeljuju u nezavisne cjeline koje nazivamo zadatci (engl. task). Tako možemo svakoj dretvi ili procesu pridijeliti jedan zadatak. Jednoprocesorski sustavi mogu u jednom trenutku obavljati samo jedan zadatak, tj. izvoditi samo jednu dretvu. Višeprocesorski sustavi mogu odabrati više dretvi

(onoliko koliko ima procesora) te ih izvoditi istovremeno. Međutim, višezadaćni rad moguće je ostvariti i u jednoprocesorskom sustavu tako da zadatci dijele procesor te svaki dobije dio vremena za izvršavanje. Raspoređivač poslova određuje kada i koji zadatak će dobiti priliku za izvođenje, te koliko će tog vremena dobiti. Problem odabira dretve iz skupa dretvi naziva se problemom raspoređivanja. Najčešće se susrećemo s takvim problemima u RTS-ovima zbog njihovih specifičnih zahtjeva, tj. vremenskim ograničenjima postavljenim prema zadatcima koja moraju biti zadovoljena. Algoritama za raspoređivanje zadataka ima više, a svaki ima svoje prednosti i nedostatke, što će dalje u nastavku biti razmatrano. Neki od uobičajenih algoritama raspoređivanja u računalima su: raspoređivanje prema prioritetu, raspoređivanje prema redu prispijeća, raspoređivanje podjelom vremena. RTS-ovi često podržavaju i neke druge metode raspoređivanja. Problem raspoređivanja može biti riješen i predimenzioniranjem sustava, tj. korištenjem bržeg sklopovlja koje ima veću moć obrade podataka i time se smanjuje ili uklanja problem nemogućnosti raspoređivanja dretvi u najkritičnijima situacijama. Takav način je skuplji, a i često nam nije praktičan zbog većeg utroška energije i problema hlađenja.

2. Izvedbe upravljanja u sustavima za rad u stvarnom vremenu

Sve zadaće RTS-a najčešće možemo svesti na nadzor i upravljanje. Postoji više načina upravljanja, a svaki ima svoje prednosti i nedostatke. U ovom poglavlju će biti navedeni neki osnovni načini upravljanja s obzirom na strukturu programskog koda i način pokretanja određenih aktivnosti. Složeniji načini nude više mogućnosti, ali i zahtijevaju primjenu složenijih postupaka, više resursa i naprednije sklopovlje koje je potrebno za njihovu implementaciju. Algoritme raspoređivanja nalazimo u naprednijim mehanizmima upravljanja koji koriste jezgru operacijskog sustava gdje su poslovi podijeljeni u zadatke.

Upravljačka petlja

Najjednostavniji način upravljanja je upravljačka petlja. Ovakav tip upravljanja moguć je na svim računalnim sustavima, čak i na najjednostavnijim mikrokontrolerima. Ne zahtijeva nikakvo posebno sklopovlje. Procesor tijekom cijelog vremena izvršava petlju u kojoj obavlja određene poslove. Možemo odmah uočiti da je ovakav način prikladan za mali broj periodičkih poslova koje možemo redoslijedom obavljati u petlji jedan po jedan. Na sljedećoj slici prikazan je primjer izvedbe upravljačke petlje. Na početku rada sustava obavljamo inicijalizaciju (sklopova koji se koriste: UART, SPI, I2C, brojila itd.), a nakon toga započinje petlja iz koje sustav ne izlazi. Unutar petlje procesor redom očitava potrebne podatke, ispituje njihove vrijednosti te na temelju toga obavi neki posao. Nedostatak ovog načina je kada sustav mora obrađivati više periodičkih i sporadičnih zadataka. To je izvedivo upravljačkom petljom, ali je komplicirano i vrlo lako se napravi pogreška. Također, vanjske jedinice se poslužuju redom, što može rezultirati slabijim odzivom jer svaka jedinica mora čekati završetak obrade prethodnih. Za obradu sporadičnih događaja možda je prikladniji sljedeći način, a to je upravljanje zasnovano na događajima.

```
void main() {
    inicijalizacija();

    while (1) {
        var = ocitaj_senzor1();
        if (potrebna_akcija1(var))
            obavi_akciju1();

        var = ocitaj_senzor2();
        if (potrebna_akcija2(var))
            obavi_akciju2();

        var = ocitaj_senzor3();
        if (potrebna_akcija3(var))
            obavi_akciju3();


        // ...
        pricekaj(ms);
    }
}
```

Slika 1. Upravljačka petlja

Upravljanje zasnovano na događajima

Ovaj način upravljanja izvodi se pomoću prekida, čime dobivamo i brži odziv sustava. Upravljanje zasnovano na događajima funkcionira tako da događaji koji zahtijevaju neku akciju izazovu prekid na procesoru, a procesor onda obavi zadani posao. Ovaj način daje još jednu prednost, a to su prioriteti. Sada možemo važnijim događajima dati veći prioritet i time dati prednost za obavljanje zadatka ako se više događaja desi istovremeno. Također ako neke akcije zahtijevaju dulju obradu, postoji mogućnost njihovog privremenog prekidanja. To za razliku od prethodnog načina zahtjeva i malo složenije sklopovlje, što podrazumijeva mogućnost procesora da obrađuje prekide i u nekim slučajevima prekidni kontroler. Dok nema zahtjeva za prekidima, procesor može raditi nešto korisno kao npr. obavljati upravljačku petlju (čime dobivamo kombinaciju ovog i prethodnog načina upravljanja) ili može čekati prekid, što često podrazumijeva stavljanje procesora u nisko potrošni rad popularnije kao „sleep mod“. Kod ovakve obrade valja paziti i na pristup zajedničkim strukturama podataka kako bi i dalje ostale konzistentne

tj. da određene operacije nad podatcima budu obavljene do kraja bez njihovog prekidanja. Zbog toga je potrebno imati i mogućnost povremene zabrane prekida, te ponovno omogućavanje prekida, što možemo nazvati i najjednostavnijim sinkronizacijskim mehanizmom. Sljedeća slika prikazuje primjer upravljanja temeljeno na događajima. Prekidne funkcije najčešće obavljaju vrlo kratke poslove kao npr. promjena stanja nekih varijabli. Cilj je da procesor što manje vremena provede u prekidnoj funkciji radi boljeg odziva na druge događaje koji se u tom trenutku mogu pojaviti.

```
void main() {
    inicijalizacija();
    registriraj_prekide();

    // čekaj prekid
    // ili radi nešto korisno
    while (1)
        čekaj_prekid();
}

void obrada_prekida1() {
    obavi_posao1();
}

void obrada_prekida2() {
    obavi_posao2();
}

void obrada_prekida3() {
    obavi_posao3();
}
```

Slika 2. Upravljanje zasnovano na događajima

Jezgra operacijskog sustava

Prethodni način upravljanja pokazuje se praktičnim za obradu sporadičnih događaja uz mogućnost obavljanja manjeg broja periodičkih zadataka u petlji. Problem se komplicira ako sustav treba obavljati više poslova. Takve probleme je nekad moguće riješiti prethodnim metodama upravljanja, ali to postaje vrlo zahtjevno kada imamo veći skup periodičkih poslova. Zbog toga se povećava mogućnost greške, programski kod postaje nepregledan, teže ga je održavati, te se javlja problem kako najbolje iskoristiti procesorsko vrijeme, a da svi zadatci budu obavljeni u skladu sa svojim vremenskim zahtjevima. Rješenje može biti korištenje operacijskog sustava (skraćeno OS) koji nudi mnogo mogućnosti vezanih za upravljanje zadatcima, ali i ostalim resursima računalnog sustava. Značajno jednostavnije upravljanje postiže se razdvajanjem upravljačkog programa u nezavisne entitete – dretve. Dretve se u ugradbenim računalnim sustavima najčešće nazivaju zadatcima (engl. task). Svaki zadatak obavlja svoj posao tj. brine se za jedan proces upravljanja. Dretve zahtijevaju višezadaćni rad sustav. Također postavlja se pitanje kojim redoslijedom je najbolje obaviti određene zadatke kada ih ima više, a da svi budu obavljeni na vrijeme. Sve to se uvelike olakšava korištenjem operacijskog sustava.

Operacijski sustav koristi sklopovlje računala preko skupa funkcija koje nazivamo upravljački programi (engl. device drivers). S druge strane, OS nudi korisničkim programima sučelje preko jezgrinih funkcija (API, engl. Application Programming Interface). To su najčešće funkcije za upravljanje zadatcima, sklopovima i ostalim mehanizmima koje jezgra nudi. Neke od funkcija za upravljanje zadatcima su: pokretanje zadatka, odgoda izvođenja, završetak zadatka, čekanje na neki događaj, te funkcije sinkronizacijskih mehanizama kao što su: binarni semafori, brojački semaforu, zastavice događaja, signali, barijere, monitori itd. Na sljedećoj slici možemo vidjeti primjer korištenja operacijskog sustava. Korisnik pokreće 3 periodička zadatka koji obave posao, a zatim pozivaju jezgrinu funkciju za odgodu izvođenja čime se zadatak stavlja na čekanje, a za to vrijeme procesor može obavljati drugi zadatak.

```
void main() {
    inicijalizacija();
    pokreni_zadatak(z1);
    pokreni_zadatak(z2);
    pokreni_zadatak(z3);
}

void z1() {
    while (1) {
        obavi_posao1();
        odgodi_izvođenje(ms);
    }
}

void z2() {
    while (1) {
        obavi_posao2();
        odgodi_izvođenje(ms);
    }
}

void z3() {
    while (1) {
        obavi_posao3();
        odgodi_izvođenje(ms);
    }
}
```

Slika 3. Korištenje operacijskog sustava

Ako zamislimo da imamo puno više zadataka, problem bi bio gotovo nemoguće efikasno riješiti upravljanjem pomoću prethodne dvije metode. Ovim načinom dobivamo neke prednosti: programski kod je čitljiviji te ga je lakše razumjeti, samim time programski kod je lakši za održavanje. Također programskog koda ima manje jer jezgra operacijskog sustava obavlja dobar dio posla. Zbog toga je korisniku lakše projektirati sustav jer ne mora razmišljati o nekim detaljima kao što je raspoređivanja poslova (o tome se brine OS). Nedostatak je što je sustav složeniji i zauzima više računalnih resursa (mehanizmi OS-a zauzimaju određenu količinu memorije, izvršavanje funkcija OS-a troši procesorsko vrijeme) te ga je teže implementirati na slabijim ugradbenim računalima.

3. Operacijski sustavi za rad u stvarnom vremenu

Operacijski sustav (skraćeno OS) je skup funkcija i programa koji olakšava korištenje računala. OS maskira specifičnosti sklopovlja kroz sučelje preko kojega pristupamo jezgrinim funkcijama. Također operacijski sustav se brine da računalni resursi budu što efikasnije iskorišteni, a to se ostvaruje pomoću višezadaćnog rada. Višezadaćni rad na jednom procesoru ostvaruje se podjelom procesorskog vremena, npr. kada neki zadatak za dovršetak operacije mora nešto pričekati (protok vremena, dovršetak I/O operacije na vanjskoj jedinici, završetak drugog zadatka, itd.), procesor može taj zadatak staviti na čekanje, a za to vrijeme obavljati drugi zadatak i tako korisno potrošiti vrijeme. Time se povećava iskoristivost računalnog sustava jer procesor neće izvoditi radno čekanje nego će cijelo vrijeme obavljati koristan posao. Takvom podjelom procesorskog vremena dobivamo prividni višezadaćni rad na jednom procesoru. Naravno, da bismo imali pravi višezadaćni rad potreban je veći broj procesora koji paralelno mogu izvoditi programe. Operacijski sustavi koje koristimo svakodnevno u osobnim računalima i prijenosnim računalima nazivamo općenamjenskim operacijskim sustavima. Oni su najčešće građeni generički i namijenjeni su za obavljanje mnoštvo različitih stvari uz zadovoljavajuću kvalitetu i nisku cijenu. Takvi operacijski sustavi zbog nepredvidivog ponašanja i sporijih odziva često nisu pogodni za upravljanje procesima koji imaju kritična vremenska ograničenja. Zbog toga u takvim sustavima koristimo operacijski sustave za rad u stvarnom vremenu (RTOS, engl. Real Time Operating System). Za razliku od općenamjenskih operacijskih sustava, RTOS imaju neka drugačija svojstva kao što su: determinističko ponašanje, brzi odziv na određene događaje, više različitih metoda raspoređivanja poslova te mala latencija jezgrinih funkcija. Neke od osnovnih razlika između općenamjenskih i RT operacijskih sustava prikazani su u sljedećoj tablici.

Tablica 1. Razlika između RT i općenamjenskih operacijskih sustava

Općenamjenski operacijski sustavi	**RT operacijski sustavi**
Koriste se u osobnim računalima	Koriste se u ugradbenim računalima
Pravedno raspoređivanje poslova	Raspoređivanje bazirano na vremenu
Kašnjenje obrade nije presudno	Obrada mora biti obavljena do roka
Nema inverzije prioriteta	Koristi se inverzija prioriteta
Jezgrine funkcije su neprekidive	Jezgrine funkcije su prekidive

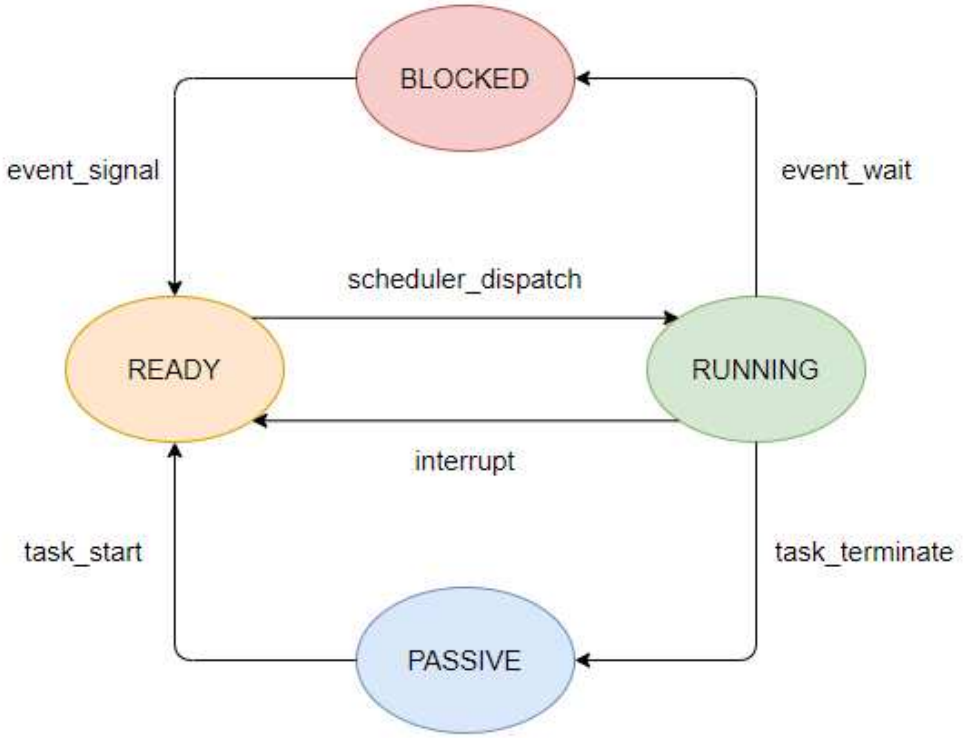
Arhitektura RT operacijskog sustava ovisi o njegovoj složenosti i namjeni. Za najjednostavnije slučajeve operacijski sustav se može sastojati od svega par jezgrinih funkcija koje pružaju višezadaćni rad, dok kod složenih sustava mogu sadržavati mnoštvo različitih mehanizama podijeljenih u podsustave kao što su: podsustav za upravljanje zadatcima, podsustav za upravljanje vremenom, podsustav za upravljanje memorijom, podsustav za upravljanje iznimkama/prekidima, podsustav za upravljanje ulazno-izlaznim jedinicama, mrežni podsustav, datotečni sustav te razne biblioteke koje mogu sadržavati pomoćne funkcije i algoritme za obradu podataka. Također RT operacijski sustav možemo implementirati sami po svojim potrebama ili možemo uzeti jedan od gotovih komercijalnih sustava, što ubrzava razvoj. Mnogi gotovi RTOS su modularni te omogućuju da implementiramo samo one mehanizme koji su nam potrebni, za razliku od općenamjenskih OS koji sadrže sve što bi nekad nekome moglo zatrebati, kao npr: podrška za različite uređaje/sklopove, podrška za nekoliko datotečnih sustava, različite biblioteke koje sadrže algoritme za obradu podataka (kompresija, multimedija, kriptografija, itd.) te mnoštvo pomoćnih programa i aplikacija. U nastavku su objašnjene neke od osnovnih komponenti RT operacijskih sustava koje većina njih sadrži.

Slika 4. Jezgra operacijskog sustava

3.1. Upravljanje zadatcima

Podsustav za upravljanje zadatcima, ujedno i podsustav za višezadaćni rad. Sadrži raspoređivač i algoritme za raspoređivanje zadataka te sučelje funkcija pomoću kojih korisnik može stvarati, pokretati i zaustavljati zadatke. Najprimitivnije jezgre operacijskog sustava često sadrže samo ovaj podsustav kako bi omogućile višezadaćni rad. Osnovna zadaća ovog sustava je pobrinuti se da svi zadatci dobiju dovoljno procesorskog vremena u određenom trenutku kako bi zadovoljili sve vremenske kriterije. Rad ovog sustava možemo opisati petljom u kojoj raspoređivač odabere jedan zadatak iz skupa pripravnih koji čekaju u redu za izvođenje, te mu dodjeli procesor na neko vrijeme ili do nekog događaja. Nakon toga raspoređivač prekida izvođenje te uvrsti zadatak ponovo u red čekanja, te izabere sljedeći i tako u krug. Dakle zadatak može biti u stanju čekanja na izvršavanje (pripravan) i u stanju izvođenja na procesoru (aktivan). U operacijskim sustavima postoji

više stanja u kojima se pojedini zadatak može nalaziti i ona se razlikuju u svakom OS-u. Međutim generalno svi se mogu svesti na pojednostavljeni model kojeg prikazujemo dijagramom stanja. Sljedeća slika prikazuje stanja zadataka u RT operacijskim sustavima:



Slika 5. Stanja zadataka u operacijskom sustavu

Na slici možemo vidjeti 4 stanja u kojima se zadatak može nalaziti, a to su:

Pripravno stanje (engl. ready), ovdje se nalaze svi zadatci koji čekaju priliku za izvođenje na procesoru. Najčešće su organizirani u red čekanja koji je implementiran listom ili stablom. Zadatak se nakon pokretanja uvrštava u red pripravnih zadataka.

Blokirano stanje (engl. blocked), ovdje se nalaze svi zadatci koji čekaju na neki događaj. U praksi zadatak može čekati različite događaje pa zbog toga najčešće imamo više redova čekanja ovisno o tome što zadatak čeka. To može biti signal, dovršetak I/O operacije na nekom uređaju, oslobađanje nekog resursa, istek vremenskog intervala i drugi. Zbog jednostavnosti, na slici su svi ti redovi čekanja predstavljeni jednim stanjem: blokirani zadatci.

Aktivno stanje (engl. running), ovdje se nalaze zadatci koji se trenutno izvode, tj. trenutno su na procesoru. Na jednoprocesorskom sustavu samo jedan zadatak može biti aktivan, dok u višeprocesorskom sustavu možemo imati aktivnih zadataka koliko i procesora.

Pasivno stanje (engl. passive), ovdje se nalaze zadatci koju su završili svoj posao i u sustavu više ne postoje (ostaju samo deskriptori s povratnim vrijednostima i slično). Također ovdje se nalaze i zadatci koji još nisu niti bili pokrenuti, a one koji su završili možemo po potrebi ponovo pokrenut.

U nastavku su navedene neke osnovne funkcije koje podsustav za upravljanje zadatcima može nuditi:

pokreni_zadatak (funkcija, argumenti, prioritet, veličina_stoga);

pozovi_raspoređivač ( );

završi_zadatak (id_zadatka);

odgodi_izvođenje (vrijeme_ms);

pauziraj_zadatak (id_zadatka);

nastavi_zadatak (id_zadatka);

Pokretanje zadatka izvodi se funkcijom „pokreni_zadatak“ koja najčešće prima parametre kao što su: adresa funkcije/procedure koja će se izvršiti, argumenti koji se predaju funkciji/proceduri, prioritet zadatka, veličina stoga i drugi. Na prethodnoj slici (slika 5) funkcija je nazvana „task_start“. Pozivom ove funkcije zadatak prelazi iz pasivnog u pripravno stanje (dodaje se na kraj reda pripravnih zadataka) i smatramo da je pokrenut. Kada dođe vrijeme za izvođenje novog zadatka, raspoređivač prekida trenutni aktivni zadatak i stavlja ga na kraj reda čekanja pripravnih zadataka (na slici strelica „interrupt“). Nakon toga uzima sljedeći zadatak iz reda pripravnih i počinje ga izvršavati (na slici strelica „scheduler_dispatch“). Ne mora se nužno uzeti prvi iz reda pripravnih, to ovisi o algoritmima raspoređivanja koji će biti detaljnije

obrađeni u sljedećem poglavlju. Zadatak koji se upravo izvršava (nalazi u aktivnom stanju) može iz tog stanja izaći iz više razloga. Jedan od mogućih razloga je da zadatak završi svoj posao (kraj glavne „main“ funkcije) ili da sam pozove funkciju za završetak „završi_zadatak“ (na slici strelica „task_terminate“). Nakon toga zadatak prelazi u pasivno stanje. Drugi razlog napuštanja aktivnog stanja može biti i prekid od strane raspoređivača (na slici strelica „interrupt“), također zadatak može i sam pozvati raspoređivač kako bi prepustio procesor drugom (obavezno se koristi kod nekih algoritama raspoređivanja). Takva funkcija se često naziva „task_yield“. Preostaje još i treći način napuštanja aktivnog stanja, a to je čekanje na neki događaj što rezultira blokiranjem zadatka (na slici funkcija „event_wait“) i prelazak u stanje blokiranih. Zadatak najčešće završava u ovom stanju pozivom blokirajuće funkcije nekog od sinkronizacijskih ili komunikacijskih mehanizama kao što su: semafor, monitor, barijera, zastavica događaja, obavljanje I/O operacije i drugi. Za primjer ćemo uzeti poziv funkcije za odgodu izvođenja koja se u praksi često naziva „task_sleep“, a kao argument se predaje vremenski interval za koji zadatak želi odgoditi izvođenje. Pozivom ove funkcije zadatak se uvrštava u red čekanja za događaj: prekid vremenskog sklopa (što bi bio signal da je interval istekao). Kada se događaj desi, zadatak se oslobađa iz reda čekanja i ponovo se uvrštava na kraj reda pripravnih (na slici strelica „event_signal“). Mnogi operacijski sustavi daju i mogućnost da zadatak bude prisilno zaustavljen od strane drugog zadatka, to bi značilo prijelaz iz blokiranog ili pripravnog stanja u pasivno stanje. Ovo je pojednostavljeni prikaz RT operacijskog sustava te će se takav model koristiti dalje u razmatranju.

3.2. Sinkronizacija i komunikacija zadataka

Zadatci iako imaju određeni stupanj neovisnosti, to ne znači da nemaju baš ništa zajedničkoga. Često se javlja potreba da zadatci međusobno moraju komunicirati, razmjenjivati neke podatke te signalizirati jedan drugom neke događaje. Zadatci ponekad dijele određene strukture podataka ili koriste iste resurse pa je potrebno osigurati da ne dođe do istovremenih pristupa od strane više različitih zadataka, što se može desiti kod višezadaćnog rada. Mehanizama sinkronizacije i komunikacije ima više, a za primjer bit će navedeni samo neki koji se najčešće koriste u RT operacijskim sustavima.

Dijeljene varijable i memorijski prostor

Ovo je jednostavno ostvarenje komunikacije među zadatcima koristeći memorijski prostor kojem mogu pristupiti svi zadatci. Međutim operacije izmjena podataka često nisu atomarne već se sastoje od više strojnih instrukcija. Pri tome izmjena podataka može biti prekinuta, a zapis podataka ne dovršen. Zato je potrebno osigurati kontrolirani pristup kako bi podatci ostali konzistentni. Jedan od načina je iskoristiti jednu varijablu kao zastavicu zaključavanja. Neke procesorske arhitekture sadrže instrukciju čitaj-piši što omogućuje atomarno čitanje i pisanje podatka u memorijsku lokaciju jednom instrukcijom. Drugi način može biti privremena zabrana prekida. Problem s idejom zabrane prekida je ako se radi o većoj količini podataka, tada ta operacija može potrajati duže. Dugotrajna zabrana prekida može narušiti brzinu odziva sustava.

Signali

Signali su najjednostavniji mehanizam komunikacije kojeg RT operacijski sustavi nude, ali ne nužno svi. Svaki zadatak u svom deskriptoru može imati nekoliko zastavica (8, 16, 32 ovisno o implementaciji). Signalnu zastavicu (ili nekoliko njih) može postaviti bilo koji drugi zadatak pomoću logičke operacije ILI. Svaki zadatak može čitati samo stanja svojih zastavica. Proces čitanja zastavica je destruktivan, tj. svakim čitanjem zastavice se brišu. U nekim sustavima ovaj mehanizam je bolje sofisticiran, a funkcionira tako da se

posebna funkcija (koju zadatak registrira) automatski pozove kada se postave bilo kakve signalne zastavice. To uklanja potrebu da zadatak sam prati svoje zastavice, čime je taj mehanizam analogan prekidima na procesoru. Primjer funkcija za rad sa signalima:

registriraj_signal (funkcija, signal);

čekaj_signal (signal);

pošalji_signal (id_zadatka, signal);

Zastavice događaja

Zastavice događaja (engl. event flags) su također komunikacijski mehanizam, a razlikuju se od signala po tome što su neovisni jezgrini objekti, dakle ne pripadaju niti jednom zadatku. Bilo koji zadatak može postaviti i očistiti zastavice pomoću logičkih operacija ILI i I. Također bilo koji zadatak može ispitivati stanja zastavica. U mnogim operacijskim sustavima moguće je izvršiti blokiranje zadatka, što znači da zadatak ostaje blokiran dok se ne postave određene zastavice događaja na koje on čeka. Primjer funkcija za rad sa zastavicama:

čekaj_događaj (id_grupe, događaj, or/and);

postavi_događaj (id_grupe, događaj);

Semafori

Semafori su također nezavisni jezgrini objekti, a koriste se za kontrolu pristupa određenom resursu (sinkronizaciju). Općenito postoje dvije vrste semafora: binarni (imaju samo 2 stanja) i brojački (imaju proizvoljan broj stanja). Funkcije za korištenje semafora su: čekaj_semafor i postavi_semafor. Bilo koji zadatak može pokušati proći kroz semafor kako bi dobio pristup određenom resursu. Ako je trenutna vrijednost semafora veća od 0, zadatak prolazi kroz semafor i smanjuje mu vrijednost za 1. Ako je vrijednost semafora 0, zadatak ostaje blokiran u redu čekanja koji pripada

tom semaforu. Svaki semafor sastoji se od varijable stanja i reda čekanja za blokirane zadatke. Zadatak može biti oslobođen iz reda čekanja na semafor ako bilo koji drugi zadatak oslobodi (postavi) semafor. Funkcija za oslobađanje semafora oslobađa jedan zadatak iz reda čekanja, a ako je red čekanja prazan, uvećava se varijabla stanja semafora. Kod binarnog semafora varijabla stanja može imati maksimalnu vrijednost 1 (daljnje postavljanje semafora ne mijenja ništa), dok kod brojačkog može svakim postavljanjem biti veća za jedan. Primjer funkcija za rad sa semaforima:

inicijaliziraj_semafor (sem_id, pocetna_vrijednost);

prođi_semafor (sem_id);

postavi_semafor (sem_id);

Redovi poruka

Redovi poruka su mehanizam za komunikaciju između zadataka, također su neovisni jezgrini objekti. Veličine poruka ovise o implementaciji, a najčešće su fiksne. Poruke mogu sadržavati podatke ili pokazivače na strukture koje sadrže više podataka. U nekim operacijskim sustavima poruke su implementirane tako da se podatak zapisuje u običnu varijablu, a jezgra samo kontrolira pristup. Sučelje za korištenje poruka sastoji se od funkcija: pošalji_poruku i pročitaj_poruku .Svaki zadatak može poslati poruku pri čemu može završiti blokiran ako je pretinac poruka pun. Također zadatak može pročitati poruku ili završiti blokiran ako je pretinac poruka prazan. Poruke mogu biti organizirane po redu prispijeća (FIFO) ili prioritetno (važnije poruke će uvijek biti prije pročitane). Ponekad postoji i treća funkcija za baratanje porukama: šalji_svima. Ova funkcija oslobađa sve zadatke koji čekaju u redu na nekom od pretinaca, pri čemu svi zadatci dobivaju istu poruku. Primjer funkcija za korištenje redova poruka:

pošalji_poruku (red_id, poruka);

pošalji_svima (red_id, poruke);

*pročitaj_poruku (red_id, *poruka);*

Uzajamno isključivi semafor (mutex)

Ovaj semafor funkcionira gotovo identično kao binarni semafor, uz malu razliku. Uzajamno isključivi semafor ima još jednu varijablu u kojoj se pohranjuje „vlasnik“ semafora, tj. zadatak koji ga je zaključao. To znači da samo zadatak koji je zaključao semafor može otključati semafor, dok kod binarnog semafora bilo koji zadatak može otključati (postaviti) semafor. Uzajamno isključivi semafor nudi i mogućnost rekurzivnog poziva, što znači da zadatak koji je zaključao semafor može više puta pozvati funkciju za zaključavanje bez da završi blokiran. Nakon toga dovoljno je samo jednom pozvati funkciju za otključavanje kako bi se semafor oslobodio. Primjer funkcija za korištenje uzajamno isključivog semafora je:

```
zaključaj_resurs ( mutex_id );
otključaj_resurs ( mutex_id );
```

Kod RT operacijskih sustava često postoji ekvivalentna funkcija blokirajućim funkcijama koja obavlja isti posao, ali neće blokirati zadatak, već će vratiti grešku. Tako možemo imati dvije vrste funkcija „čekaj_semafor“, blokirajuću i ne-blokirajuću. Također blokirajuća funkcija može nuditi zadavanje maksimalnog vremenskog intervala za kojeg zadatak može biti blokiran. Ako vrijeme istekne, a resurs nije oslobođen, zadatak se odblokira, a funkcija vraća grešku. Primjer takvih funkcija je sljedeći:

```
// blokirajuća funkcija
čekaj_semafor ( sem_id );

// ne-blokirajuća funkcija, ako je vremenski interval jednak 0
čekaj_semafor (sem_id, vremenski_interval );
```

3.3. Upravljanje memorijom

Složeniji operacijski sustavi imaju podsustav za upravljanje memorijom kojem je glavna svrha efikasno upravljanje radnom memorijom. To uključuje dodjeljivanje i oslobađanje blokova memorije koju koriste zadatci, a također je bitan mehanizam straničenja memorije kojeg koriste sustavi koji implementiraju potpunu memorijsku izolaciju zadataka (procesi). Kod općenamjenskih operacijskih sustava rad je nezamisliv bez procesa, međutim kod ugradbenih računala i RT sustava, procesi su zamijenjeni dretvama (zadatcima) koje dijele memoriju. Procesi kod takvih sustava nisu pogodni jer zahtijevaju složene sklopove potrebne za straničenje memorije koje mikrokontroleri najčešće nemaju, a također su i zahtjevni jer troše više računalnih resursa (procesor ima više posla oko procesa i sporije barata s njima nego s dretvama). Ipak mnogi RT operacijski sustavi sadrže mehanizme za dinamičko dodjeljivanje memorijskih blokova. Upravljati spremnikom možemo na dva načina: statički i dinamički.

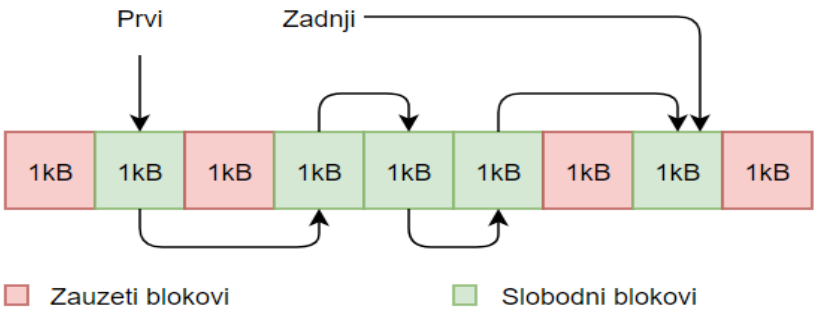
Statičko upravljanje memorijom

Operacijski sustav za svoj rad treba određene strukture podataka. To mogu biti globalne varijable i strukture podataka koje zauzimaju mjesto prilikom pokretanja sustava. Takvim varijablama se memorijski prostor određuje prilikom prevođenja programskog koda, te one ostaju na tom mjestu tijekom cijelog vremena rada sustava. Za njih kažemo da su statički zauzele memorijski prostor. Možemo odmah zaključiti da nam za ovakav način ne treba nikakav poseban mehanizam i algoritmi za upravljanje memorijom jer smo na početku zauzeli memoriju koja će nam trebati i to se ne mijenja. Ovakav način možemo koristiti u jednostavnim sustavima, dok u složenijim sustavima često trebamo određeni memorijski prostor samo privremeno, te je zbog toga teško predvidjeti koliko će nam memorije u kojem trenutku trebati. Ako zauzmemo previše prostora, trebat će nam sklopovlje koje sadrži više memorije (viša cijena opreme), ako zauzmemo premalo, može se desiti da sustav ne može obaviti svoju zadaću u nekim situacijama jer nema dovoljno

memorije. Taj problem rješava podsustav za dinamičko upravljanje memorijom.

Dinamičko upravljanje memorijom

Dinamičko upravljanje memorijom ostvaruje se podsustavom koji gospodari memorijom. Umjesto da memorije bude na početku statički zauzeta, ona će se dodjeljivati po potrebi prema zahtjevima sustava tijekom rada. Prilikom pokretanja sustava zauzima se dio memorije koji se naziva gomila (engl. heap). Tijekom rada sustava gomila se određenim algoritmima dijeli na blokove (zauzete i slobodne), te se blokovi dodjeljuju zadatcima na zahtjev. Algoritam za dinamičku dodjelu memorije definira način podjele memorijskog prostora, način traženja slobodnog bloka, način organizacije blokova itd. Algoritmi se razlikuju po svojstvima kao što su: složenost dodjele i oslobađanja (koliko će operacija trajati), utjecaj fragmentacije (moguća iskoristivost prostora, efikasnost) i svojstva zahtjeva (zahtjevi za velikim ili malim blokovima, utjecaj zaglavlja blokova). Najjednostavniji algoritam je podjela prostora na blokove fiksne duljine te organizacija u listu. Kod zauzimanja memorije uzima se prvi blok iz liste i daje na korištenje, dok kod oslobađanja, zauzeti blok se ponovo uvrštava na kraj liste te time postaje opet slobodan. Ovaj algoritam ima vrlo nisku složenost operacija O(1), ne postoji fragmentacija prostora i vrlo je jednostavan za izvedbu. Nedostatak mu je što se prostorom ne upravlja racionalno. Blokovi su fiksne veličine npr. 1 kB, što ako nam treba polje od 100 bajtova, ili blok od nekoliko kB, kako odrediti optimalnu veličinu blokova ? Dio prostora ostaje ne iskorišten.



Slika 6. Dinamičko upravljanje memorijom fiksnim blokovima

3.4. Upravljanje vremenom

Upravljanje vremenom je vrlo bitna značajka RT operacijskih sustava. Zahtjevi za upravljanje vremenom mogu dolaziti iz jezgre ili od zadataka. Jezgra zahtijeva vrijeme zbog mehanizama raspoređivanja zadataka od kojih mnogi zahtijevaju neke vremenske parametre. Zadatci mogu zahtijevati očitanje trenutnog vremena, odgodu izvođenja za određeno vrijeme, periodičke signale (postavljanje alarma) itd. Za upravljanje vremenom u operacijskim sustavima potrebna je podrška sklopovlja koja se sastoji od barem jednom vremenskog brojila koje može postavljati zahtjev za prekid procesoru. Pomoću jednog brojača možemo programski implementirati više njih, ovisno o potrebi sustav. Funkcionalnost ovog podsustava temelji se na tome da brojilo u vremenskim intervalima (unaprijed postavljen period) postavlja prekide, čime se svaki put poziva procedura za obradu tog prekida. Obrada prekida sadrži: ponovno konfiguriranje brojila za sljedeći prekid, ažuriranje varijabli proteklog vremena u sustavu (sat sustava), odblokiranje blokiranih zadatka kojima je istekao interval čekanja, te poziv raspoređivača zadataka (ovisno o načinu raspoređivanja). Funkcije koje ovaj podsustav nudi mogu biti:

*dohvati_trenutno_vrijeme (*vrijeme);*

postavi_trenutno_vrijeme (vrijeme);

odgodi_izvođenje_zadatka (vremenski_interval);

postavi_alarm (kada, period, funkcija_aktivacije, parametri_funkcije);

Alarme možemo smatrati virtualnim brojilima. Mikrokontroleri najčešće imaju po nekoliko brojila, dok je za jezgru operacijskog sustava dovoljan samo jedan, a ostali (proizvoljan broj) mogu biti programski implementirani, što najčešće imaju svi RT operacijski sustavi. Vremenski interval brojila određuje osnovni kvant vremena u operacijskim sustavima. Što je interval kraći, sustav ima preciznije vrijeme i brži odziv, ali zato procesor više vremena troši na obradu prekida. Period tog intervala je reda veličine 1 ms, dok kod bržih procesora može biti i oko 100 us.

3.5. Upravljanje prekidima

U operacijskim sustavima javljaju se tri vrste prekida: vanjski, unutarnji i programski. Vanjski prekid postavljaju periferni sklopovi i uređaju da bi signalizirali da je neka operacija završena, ili da neku operaciju treba obaviti. Takav mehanizam koristi se zbog efikasnijeg korištenja procesorskog vremena s obzirom na to da je većina vanjskih jedinica znatno sporija od procesora. Procesor može poslati komandu za obavljanje neke operacije (koja može potrajati) i za to vrijeme raditi nešto drugo, a kada operacija završi, vanjska jedinica će signalizirati prekidom procesoru da je operacija završila. Unutarnji prekidi su prekidi koje procesor može sam sebi postaviti (iznimke). To se može desiti kada dođe do dijeljenja s nulom, preljeva stoga, nedozvoljeni pristup memoriji ili izvršavanje nepostojeće instrukcije. Programski prekid koristi se kod poziva jezgrinih funkcija. Kod operacijskih sustava koji koriste višezadaćni rad temeljem na procesima, proces može pristupati samo svom memorijskom prostoru. Zbog toga poziv jezgrine funkcije zahtjeva ulaz u privilegirani način rada što se postiže programskim prekidom. Proces na svoj stog sprema parametre i postavlja određeni programski prekid. Jezgra operacijskog sustava ulazi u privilegirani način rada, te obavlja operaciju koju je proces zatražio. Kod operacijskih sustava koji višezadaćni rad temelje isključivo na dretvama, jezgrine funkcije mogu se direktno pozivati pa je to moguće ostvariti bez programskih prekida. Ovakav način je i brži jer ne zahtijeva ulaz u privilegirani način rada. Neki jednostavniji mikrokontroleri nemaju podršku za unutarnje i programske prekide, već samo za vanjske prekide od perifernih sklopova. Podsustav za upravljanje prekidima rukovodi svim prekidima i prosljeđuje njihovu obradu drugim podsustavima kao što su: podsustav za upravljanje zadatcima, vremenom, vanjskim jedinicama i ostali. Neke od funkcija koje ovaj podsustav nudi mogu biti sljedeće.

registriraj_prekid (naprava_ n, procedura_obrade);

dozvoli_prekide_od (naprava_n);

zabrani_prekide_od (naprava_n);

3.6. Upravljanje napravama

Podsustav za upravljanje napravama radi apstrakciju računalnog sklopovlja i svih vanjskih jedinica te pruža jedinstveno sučelje kojim možemo pristupiti njima. Već je ranije navedeno kako je jedna od zadaća operacijskog sustava maskirati specifičnosti sklopovlja te korisniku omogućiti njihovo korištenje bez potrebe za ulaženjem u detalje sklopa ili uređaja. Radi pojednostavljenja upravljanja napravama, u operacijskim sustavima se definira sučelje prema kojem se izgrađuju upravljački programi (engl. device driver). Prednosti korištenja sučelja su jednostavnost zamjene jedne naprave drugom (ako obavljaju sličnu funkciju) i lakša integracija novih upravljačkih programa. Sučelje prema korisničkim programima koje jezgra operacijskog sustava nudi mogu biti:

inicijaliziraj_napravu (id_naprave, postavke_konfiguracije);

*pošalji_podatke (id_naprave, *podatci, duljina);*

*pročitaj_podatke (id_naprave, *podatci, duljina);*

isključi_napravu (id_naprave);

registriraj_prekid_naprave (id_naprava, callback_funkcija);

Podsustav za upravljanje napravama omogućuje i efikasnije korištenje sklopova i njihovu integraciju u višezadaćni rad. Kod nekih jednostavnijih RT operacijskih sustava, jezgra ne sadrži podsustav za upravljanje napravama već se one koriste direktno pozivom njihovih funkcija kao što su npr: uart_init, spi_send, flash_read i ostale. Što znači da su funkcije za upravljanje napravama implementirane izvan jezgre operacijskog sustava. Prednost ovoga može biti što ne zahtijeva pozive jezgrinih funkcija (programske prekide), pa time dobivamo efikasniji i brži kod.

4. Algoritmi raspoređivanja zadataka

Na jednoprocesorskom sustavu, procesor u nekom trenutku može obavljati samo jedan posao. Višezadaćni rad na jednom procesoru postiže se kontrolom procesora tako da se izmjenjuju poslovi koje procesor obavlja tj. više poslova dijeli isti procesor tako da svaki posao dobije određeni vremenski interval za izvršavanje. Tu izmjenu poslova obavlja raspoređivač (engl. scheduler). Dakle raspoređivanje je odabir koji će zadatak u određenom trenutku dobiti priliku za izvršavanje na procesoru, dok će ostali čekati. Raspoređivač ne mora biti kompliciran, niti mora zahtijevati složeno sklopovlje, ali u nekim sustavima može biti i vrlo složen. Način raspoređivanja određuje algoritam po kojem se zadatci raspoređuju. Svaki algoritam raspoređivanja ima svoje prednosti i nedostatke, tako neki način može biti dobar za jednu skupinu problema, a loš za drugu i obratno. Algoritme raspoređivanja možemo podijeliti prema nekoliko kriterija:

- Prema vremenu i načinu donošenja odluka: statički ili dinamički
- Prema postupanju sa zadatcima u izvođenju: prekidivi ili neprekidivi

Statičko raspoređivanje

Kod statičkog raspoređivanja, odluke se donose prije pokretanja sustava. Korisnik prilikom stvaranja sustava definira određene parametre koji se neće mijenjati tijekom rada. Npr. kod statičkog prioritetnog raspoređivanja, korisnik unaprijed zadaje prioritet svakom zadatku na temelju svoje procjene. Nakon toga sustav će raspoređivati zadatke na temelju tog prioriteta tako da se prednost u izvršavanju uvijek daje zadatku s većim prioritetom. Prednost statičkih algoritama je da su jednostavni za implementaciju, troše vrlo malo resursa, promjena konteksta traje vrlo kratko. Međutim korisnik mora dobro procijeniti parametre za raspoređivanje, u ovom slučaju prioritet svakom zadatku tako da svi budu raspoređeni na najbolji način.

Dinamičko raspoređivanje

Dinamički raspoređivač mijenja parametre raspoređivanja tijekom rada sustava. Kod prioritetnog dinamičkog raspoređivanja, prioritete ne zadaje korisnik, već sustav. Također sustav tijekom rada može mijenjati svakom zadatku prioritet, ovisno o situaciji, događajima, vremenu i drugim parametrima. Dinamički raspoređivač zbog toga kod svake promjene zadatka mora analizirati određene parametre te na temelju njih donijeti odluku koji zadatak bi bilo najbolje početi izvršavati. Promjena konteksta kod dinamičkog raspoređivanja može potrajati dulje, a često vrijeme promjene konteksta može biti varijabilno, ovisno o opterećenju sustava. Zbog toga se dinamički raspoređivači manje koriste u sustavima za rad u stvarnom vremenu, a više u operacijskim sustavima opće namjene jer takvi sustavi često koriste jače procesore, a njihovi zadatci nisu kritični po pitanju vremena. Prednost dinamičkog raspoređivanja je što korisnik ne mora unaprijed analizirati zadatke kako bi zadao ispravan parametar raspoređivanja, već će se sustav sam pobrinuti o tome.

Neprekidivo raspoređivanje

Neprekidivo raspoređivanje podrazumijeva da zadatak koji se počne izvoditi ne može biti prekinut. Svaki zadatak se izvršava dok ne završi ili dok dobrovoljno ne prepusti procesor drugom zadatku. Kod takvog načina raspoređivanja promjena konteksta se događa rijetko, samo kada je potrebno. Zbog toga ovakav način raspoređivanja može rezultirati slabijim odzivom sustava na bitne događaje. Prednost neprekidivog raspoređivanja je smanjeno trošenje vremena na „kućanske“ poslove (promjene konteksta zadataka) zbog čega je procesor efikasniji. Neprekidivo raspoređivanje se često naziva kooperativni višezadaćni rad, kod kojeg se zadatci uglavnom raspoređuju po redu prispijeća (FIFO), ali može i po drugim kriterijima kao što su prioriteti. Izgladnjivanje zadataka je moguće ako zadatak koji se trenutno izvodi traje dugo, te zbog toga ostali dugo čekaju. Također može nastati potpuna blokada sustava ako je jedan zadatak loše implementiran, te ostane blokiran.

Prekidivo raspoređivanje

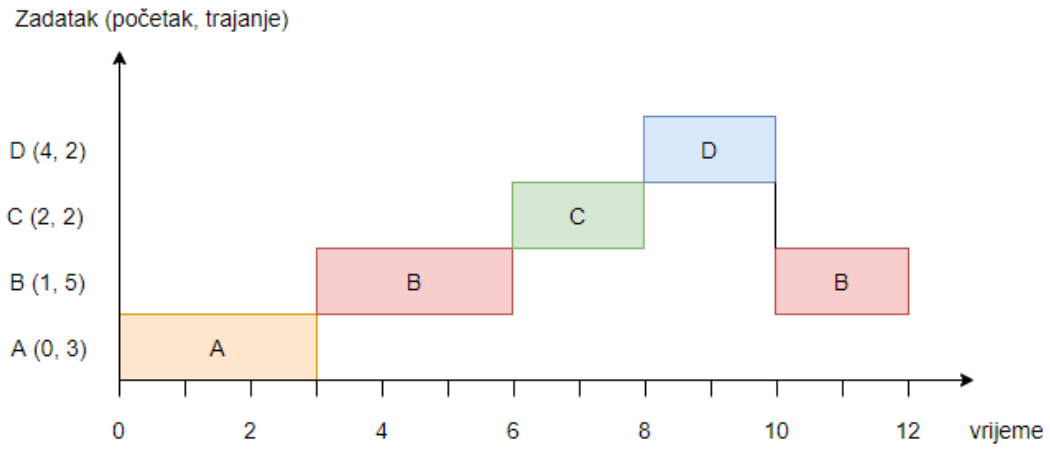
Prekidivo raspoređivanje dopušta da zadatak bude prekinut tijekom izvršavanja kako bi drugi zadatak dobio mogućnost izvršavanja. Zadatak može biti prekinut iz više razloga kao što su: pristigao je zadatak s višim prioritetom, trenutnom zadatku je isteklo vrijeme koje je dobio za izvršavanje i drugi razlozi. Zbog mogućnosti prekidanja, sustav može dati prednost važnijim zadatcima i pobrinuti se da budu prije obavljeni. Također, odziv na neke događaje može biti bolji jer zadatci s višim prioritetom ne moraju čekati. Međutim kod čestog prekidanja iskoristivost procesora može biti manja jer se više vremena troši na promjenu konteksta. Izgladnjivanje zadataka je moguće ako se koriste dodatni kriteriji raspoređivanja kao što su: prioriteti, prednost kraćim poslovima i drugi. Kod korištenja prioriteta izgladnjivanje može nastati ako zadatak s visokim prioritetom dugo traje. Za razliku od neprekidivog raspoređivanja, loše implementirani zadatak ne može zaustaviti rad cijelog sustava jer ga raspoređivač može prekinuti i dati procesor ostalima na korištenje. Korisnik ne mora voditi računa o tome da zadatak prepusti procesor sljedećem, već se o tome brine sustav. U većini operacijskih sustava koristi se prekidivo raspoređivanje.

Tablica 2. Razlike između prekidivog i neprekidivog raspoređivanja

Neprekidivo raspoređivanje	**Prekidivo raspoređivanje**
Zadatak ne može biti prekinut tijekom izvršavanja	Zadatak može biti neočekivano prekinut
Zadatak dobiva resurse na neograničeno	Zadatak dobiva resurse na ograničeno vrijeme
Izgladnjivanje moguće ako neki zadatak dugo traje	Izgladnjivanje moguće ako zadatak većeg prioriteta dugo traje
Promjene konteksta su rijetke	Promjene konteksta su česte

4.1. Raspoređivanje po redu prispijeća

FCFS (engl. First Came First Serve) ili poznatiji kao FIFO (engl. First In First Out) raspoređivač. Zadatci se obavljaju redom kojim pristignu. Ovaj algoritam spada u statičke, neprekidive algoritme. Pripravni zadatci koji čekaju na izvođenje poredani su u redu čekanja koji je najčešće implementiran listom, stablom, a može biti i statičkim poljem. Zadatci koji se pojave svrstavaju se uvijek na kraj reda. Procesor uzima prvi iz reda i počinje ga izvršavati. Zadatak se izvršava sve dok ne završi ili svojevoljno prepusti procesor sljedećem (npr. pozivom funkcije za odgodu izvođenja). U slučaju prepuštanja procesora sljedećem zadatku, trenutni (nedovršeni) se ponovo stavlja na kraj reda čekanja. Osim što je trošak raspoređivanja minimalan jer procedure za manipuliranje redom čekanja imaju složenost O(1), raspoređivanje se obavlja rijetko, samo kada je potrebno. Zbog toga se vrlo malo vremena troši na poslove raspoređivanja. Za primjer pretpostavimo da imamo 4 zadatka koji pristižu abecednim redom. Sljedeća slika prikazuje izvođenje zadataka na procesoru. Pretpostavka je da proces promjene zadatka ima trajanje 0 (u praksi nekoliko mikro sekundi).



Slika 7. Raspoređivanje po redu prispijeća

U trenutku t=0 pristiže zadatak A te ga procesor počinje izvršavati. Red čekanja ostaje prazan do trenutka t=1 u kojem pristiže i zadatak B. Zadatak B ostaje u redu čekanja sve dok zadatak A ne završi. U trenutku t=2 pristiže zadatak C. Sada u redu čekanja imamo B i C, dok se A još uvijek izvršava. U trenutku t=3 zadatak A završava izvođenje te procesor preuzima zadatak B, a C ostaje u redu. U trenutku t=4 pristiže zadatak D te se uvrštava u red. Sada imamo zadatak B na procesoru, a C i D čekaju. U trenutku t=6 zadatak B poziva funkciju (često ima naziv „task_yield“) za prepuštanje procesora sljedećem zadatku. Procesor vraća zadatak B na kraj reda čekanja te uzima sljedeći, a to je zadatak C. Sada imamo zadatak C na procesoru, a D i B u redu čekanja. U trenutku t=8 zadatak C završava te procesor uzima sljedeći, zadatak D, a u redu čekanja ostaje zadatak B. Zadatak D se izvodi do t=10 kada završava, a procesor uzima sljedeći preostali, zadatak B koji nije bio dovršen. U trenutku t=12 svi zadatci su dovršeni.

Dakle vidimo da kod ovog algoritma nema prekidanja zadataka zbog čega ovaj algoritam spada u neprekidivo raspoređivanje. To ne znači da sustav ne može reagirati na neke vanjske događaje kao što su prekidi, međutim ti prekidi ne mogu promijeniti redoslijed zadataka. Dakle u slučaju prekida, procesor ulazi u prekidnu rutinu, obavlja posao, izlazi iz prekidne rutine te nastavlja zadatak na kojem je stao, sve dok ga ne završi. Iznimno zadatak može sam prepustiti procesor sljedećem zadatku pozivom funkcije „task_yield“, što se često koristi kod raspoređivanja ovakvog tipa i time se postiže višezadaćni rad. Sučelje funkcija za upravljanje kod ovog algoritma raspoređivanja može biti sljedeće:

pokreni_zadatak (funkcija, argumenti);

završi_zadatak ( );

propusti_sljedeći ( );

FCFS algoritam raspoređivanja nema nikakvih zahtjeva prema sklopovlju. Nije potrebno imati podršku za prekide, niti vremenska brojila za potrebe implementacije samog algoritma zbog čeka ovaj algoritam spada u najjednostavnije algoritme. Za razliku od ostalih izvedbi višezadaćnog rada, kod ovog načina nema potrebe za sinkronizacijskim mehanizmima s obzirom na to da drugi zadatak ne može sporadično prekinut trenutni. Ako se koriste dijeljene varijable s prekidnim procedurama, sinkronizaciju možemo jednostavno ostvariti funkcijama za omogućavanje i zabranu prekida. Zabrana prekida u niti jednom operacijskom sustavu ne bi smjela potrajati dugo jer to narušava kvalitetu odziva sustava. Možemo navesti neke prednosti i nedostatke ovog algoritma.

Prednosti:

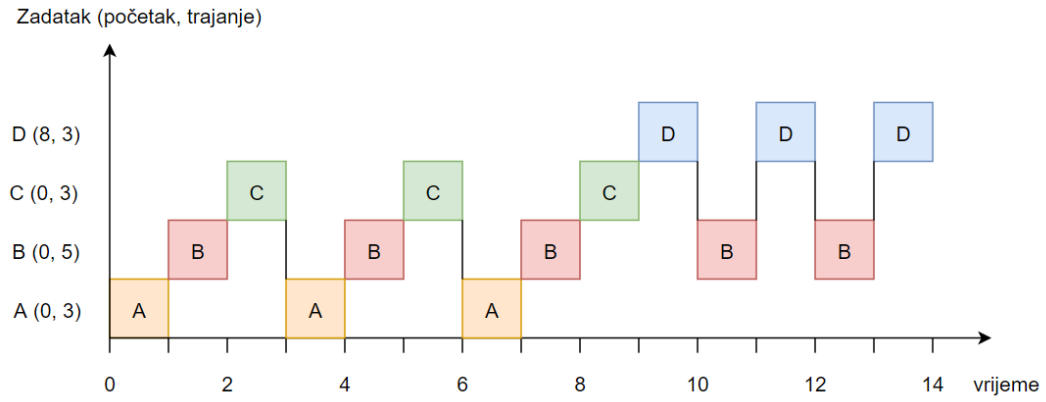
- Algoritam ne zahtijeva nikakvu kompliciranu logiku, već samo mehanizam reda čekanja (umetanje i uzimanje zadataka)
- Vrlo jednostavno funkcionira i jednostavno ga je implementirati
- Rijetke promjene konteksta, trošak raspoređivanja minimalan

Nedostatci:

- Ne postoji mogućnost prekidanja zadataka, zbog čega neki kritični zadatci možda neće stići na vrijeme obaviti svoj posao
- U slučaju da jedan zadatak zapne, cijeli sustav se blokira
- Propusnost slaba kod pojave dugih zadataka
- Slab odziv na sporadične događaje

4.2. Kružno raspoređivanje

Kružno raspoređivanje (engl. Round Robin ili skraćeno RR) funkcionira po principu podjele procesorskog vremena. Pripravni zadatci su kao u prethodnom FCFS algoritmu, poredani u red čekanja. Zadatci koji pristižu ubacuju se na kraj reda čekanja, a procesor uvijek uzima prvog u redu i počinje ga izvršavati. Razlika je što svaki zadatak dobije kvant vremena (reda veličine 1 do 10 ms) za izvođenje te nakon što zadani interval istekne, zadatak se prekida i ubacuje ponovo na kraj reda čekanja. Zbog toga ovaj algoritam spada u prekidivo raspoređivanje. Dakle zadatak ne mora biti dovršen da bi sljedeći došao na red čime se sprječava izgladnjivanje ako se pojavi zadatak koji dugo traje. Svi zadatci dobivaju jednako vremena za izvođenje, a brzina odziva ovisi o količini zadataka u sustavu, ne vezano koliko dugo traje njihova obrada. Zbog potrebe za prekidanjem i odbrojavanjem vremena, za implementaciju je potrebno imati barem jedan vremenski brojač koji može generirati prekid procesoru. Procesor svaki put prije pokretanje sljedećeg zadatka konfigurira brojač koji će generirati prekid kad vremenski interval istekne. Vremenski interval je uvijek jednak za sve zadatke, a ključno pitanje je koliki vremenski interval bi bio najbolji. Što je vremenski interval dulji, zadatci će se rjeđe prekidati, procesor će trošiti manje vremena na promjenu konteksta te će vrijeme biti efikasnije potrošeno. Nedostatak duljeg intervala bit će slabiji odziv na događaje koji zahtijevaju brzu reakciju sustava. Ako zadamo kraći interval prekidanja, sustav će imati bolji odziv, ali će promjena konteksta biti česta pa će procesor više vremena trošiti na promjene konteksta zadataka, čime dobivamo manju efikasnost procesora. Uzmimo kao primjer da procesoru treba 100 us za promjenu konteksta, a kvant vremena je 1 ms. U tom slučaju procesor skoro 10% vremena troši samo na promjenu konteksta. Za vremensku vizualizaciju rada ovog algoritma uzet ćemo sličan primjer kao u prethodnom algoritmu uz pretpostavku da je kvant vremena jednak 1, a vrijeme potrebno za promjenu konteksta zanemarivo.



Slika 8. Kružno raspoređivanje

U trenutku t=0 u sustav pristižu 3 zadatka, A, B i C. Svaki zadatak dobiva jednak interval za izvođenje i tako u krug, zbog čega je algoritam dobio naziv kružno raspoređivanje. U trenutku t=7 zadatak A završava te se više neće pojavljivati. Zadatak D dolazi u trenutku t=8, te se ubacuje na kraj reda čekanja koji u tom trenutku sadrži: C, D, B (pretpostavka da je zadatak D stigao malo prije nego li je zadatku B istekao kvant vremena, pa je ubačen u red čekanja prije zadatka B). U trenutku t=9 zadatak C završava s izvođenjem te se preostali zadatci B i D izvode izmjenično dok ne završe. Ako u sustavu postoji samo jedan zadatak, on će se izvoditi dok ne završi ili dok ne pristigne neki drugi zadatak, ali će svejedno biti prekidan svaki put kad istekne kvant vremena. U situaciji da ne postoji niti jedan pripravni zadatak, procesor se može zaustaviti i čekati sljedeći prekid.

S obzirom na to da svaki zadatak može biti prekinut, uz ovaj algoritam potrebno je imati sinkronizacijske mehanizme kako bi se mogao ostvariti kontrolirani pristup određenim resursima koji zahtijevaju takav pristup. Sučelje za upravljanje zadatcima kod kružnog raspoređivanja može biti sljedeće:

pokreni_zadatak (funkcija, argumenti);

odgodi_izvođenje (vrijeme);

završi_zadatak ( );

Dakle korisnik više ne treba voditi brigu oko prepuštanja procesora sljedećem zadatku pozivanjem funkcije „task_yield“ kao što je kod FCFS algoritma. Time je otklonjen problem da jedan zadatak može blokirati cijeli sustav, kao i problem izgladnjivanja, jer svi zadatci dobivaju jednako vremena za izvođenje. Funkcija „odgodi_izvođenje“ stavlja zadatak u red blokiranih koji čekaju da istekne zadani vremenski interval. Interval se može pohraniti u deskriptor zadatka kao trenutak u kojem čekanje završava. Na svaki prekid od vremenskog sklopa (kvant vremena), ažurira se vrijeme sustava te provjerava treba li neki zadatak osloboditi iz reda čekanja i ponovo prebaciti u red pripravnih zadataka.

Prednosti:

- Svi zadatci dobivaju jednako vremena za izvođenje
- Jednostavna implementacija
- Ne postoji mogućnost izgladnjivanja

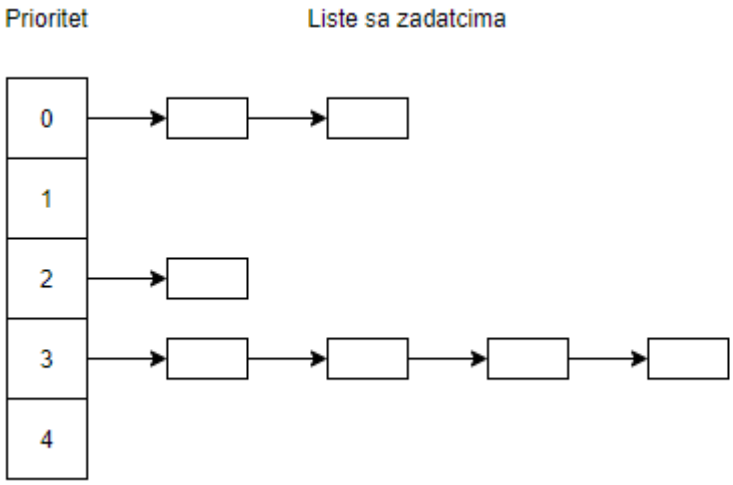
Nedostatci:

- Teško je odrediti najbolji vremenski kvant
- Predugi kvant vremena narušava brzinu odziva i propusnost
- Prekratki kvant vremena umanjuje efikasnost procesora
- Prosječno vrijeme čekanja za izvršavanje može biti predugo

4.3. Prioritetno raspoređivanje

Jedan od nedostataka prethodnog algoritma je što vrijeme čekanja u slučaju većih opterećenja može biti predugo. S obzirom na to da se svi zadatci tretiraju jednako, odziv će ovisiti o broju zadataka u sustavu. Način za sprječavanje čekanja nekih važnih zadataka i povećanja odziva na važnije događaje možemo riješiti prioritetima. Takav sustav bi davao prednost zadatcima s višim prioritetom nad zadatcima s nižim prioritetom. Svakom zadatku korisnik pridjeljuje određeni prioritet, a raspoređivač osigurava da u bilo kojem trenutku na procesoru bude zadatak koji ima najveći prioritet. Ako se pojavi zadatak s većim prioritetom, raspoređivač prekida izvršavanje trenutnog i započinje onaj s najvišim prioritetom. Prioritetno raspoređivanje može biti izvedeno statički ili dinamički, prekidivo ili neprekidivo. Kod statičkog prioritetnog raspoređivanja korisnik zadaje svakom zadatku fiksni prioritet pri pokretanju, te zadatak cijelo vrijeme ima isti prioritet. Kod dinamičkog prioritetnog raspoređivanja prioritete zadataka ne daje korisnik već raspoređivač na temelju nekih parametara. Raspoređivač tijekom rada sustava može mijenjati prioritete ovisno o stanju sustava, zbog čega se takav način i naziva dinamičko raspoređivanje. Svaki zadatak može imati svoj prioritet, ali često više zadataka ima isti prioritet. U slučaju da više zadataka ima isti prioritet postavlja se pitanje na koji način njih obavljati. Zbog toga prioritetno raspoređivanje nikad ne dolazi samo, već dolazi s nekim sekundarnim kriterijem. Sekundarni algoritam raspoređivanja može biti npr. neki od prethodna dva: kružno raspoređivanje ili raspoređivanje po redu prispijeća. Ako se radi o kružnom raspoređivanju kao sekundarnom algoritmu, govorimo o prekidivom prioritetnom raspoređivanju. U operacijskim sustavima za rad u stvarnom vremenu najčešće se koristi prioritetno raspoređivanje izvedeno statički.

Prioritetni raspoređivač može biti implementiran na više načina. Jedan način je da su svi pripravni zadatci sortirani u listi po prioritetu. Raspoređivač uzima uvijek prvog koji ima najviši prioritet, a trenutnog vraća u listu. Uzimanje prvog zadatka iz sortirane liste ima složenost O(1), ali ubacivanje zadatka u sortiranu listu ima složenost O(n), što nije efikasno kod većeg broja zadataka. Drugi, malo bolji način može biti da za svaki prioritet imamo jednu listu, kao što sljedeće slika prikazuje.



Slika 9. Redovi pripravnih zadataka kod prioritetnog raspoređivanja

Sada su opet sve operacije nad listom minimalne složenosti, O(1). Međutim kada raspoređivač uzima sljedeći zadatak, on mora ići od početka (najvišeg prioriteta koji je 0) i provjeravati da li je lista neprazna. Na prvoj listi koja nije prazna, uzima prvi zadatak i počinje ga izvoditi. Ovo pretraživanje je složenosti O(n), ali ako nemamo velik broj prioriteta već svega nekoliko razina, ovo pretraživanje će trajati vrlo kratko (možemo ga smatrati O(1)), pa ovaj način radi efikasnije od prethodnog s jednom sortiranom listom. Također, red pripravnih zadataka nije jedini koji zahtjeva složeniju implementaciju. Kod sinkronizacijskih mehanizama koji sadrže redove čekanja, želimo da zadatci s većim prioritetom budu prije odblokirani, iako su možda kasnije pristigli u red čekanja. To znači da svi redovi čekanja u

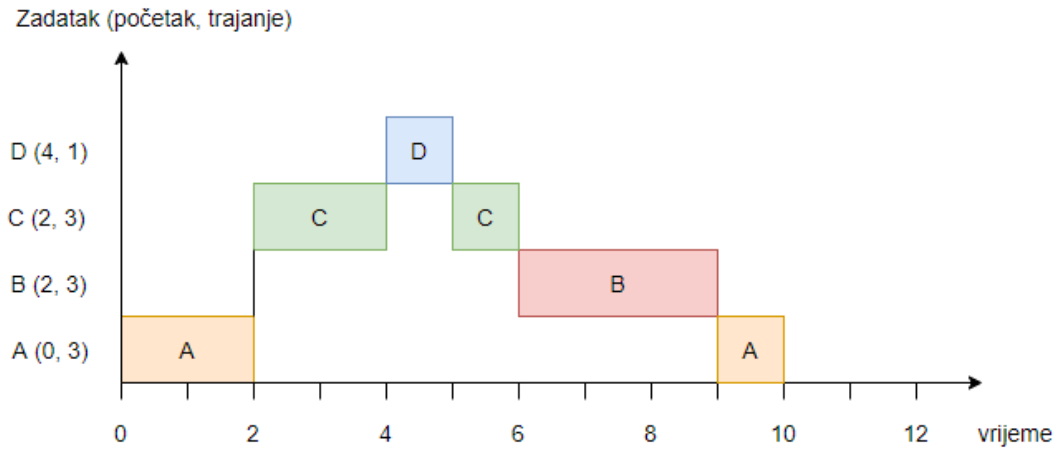
sustavu moraju imati prioritetnu organizaciju. Zbog toga strukture kod prioritetnog raspoređivanja zauzimaju više memorije, ali i jezgrine funkcije imaju veće latencije. Kod prioritetnog raspoređivanja postoji još jedan problem, a to je inverzija prioriteta. U sustavu se može desiti da zadatak s nižim prioritetom zauzme neki resurs. Ako se nakon toga pojavi zadatak s višim prioritetom, on ostaje čekati da zadatak s nižim prioritetom završi svoj posao i oslobodi resurs. Time dobivamo situaciju da zadatak s višim prioritetom čeka zadatak s nižim, dakle privremeno je došlo do prividne zamjene prioriteta. Rješenje za ovakve situacije može se implementirati zamjenom prioriteta. Zadatak koji ima niži prioritet, a zauzeo je resurs zadatku s višim prioritetom, bit će mu privremeno dodijeljen veći prioritet (prioritet zadatka koji čeka) kako bi dobio više procesorskog vremena i time što prije završio sa svojim poslom i oslobodio resurs. Kada oslobodi resurs, vraća mu se njegov niži prioritet kojeg je prije toga imao.

Funkcije za upravljanje zadatcima kod prioritetnog raspoređivanja su gotovo identične kao kod prethodnog RR algoritma, dodatno imaju samo parametar za zadavanje prioriteta svakom zadatku kod stvaranja.

pokreni_zadatak (funkcija, argumenti, prioritet);

odgodi_izvođenje (vrijeme);

završi_zadatak ( );



Slika 10. Prioritetno raspoređivanje

Zadatci imaju prioritet redom: D, C, B, A, pri čemu zadatak D ima najviši prioritet, a zadatak A najmanji. Zadatak A počinje prvi, zatim u trenutku t=2 dolaze B i C. C zadatak počinje s izvođenjem jer ima najviši prioritet u tom trenutku, sve do t=4 kada dolazi zadatak D. Zadatak B počinje s izvođenjem tek kada C i D završe, a zadatak A nastavlja tek kada svi završe jer ima najniži prioritet. Zadatak s višim prioritetom uvijek istiskuje zadatak s nižim.

Prednosti:

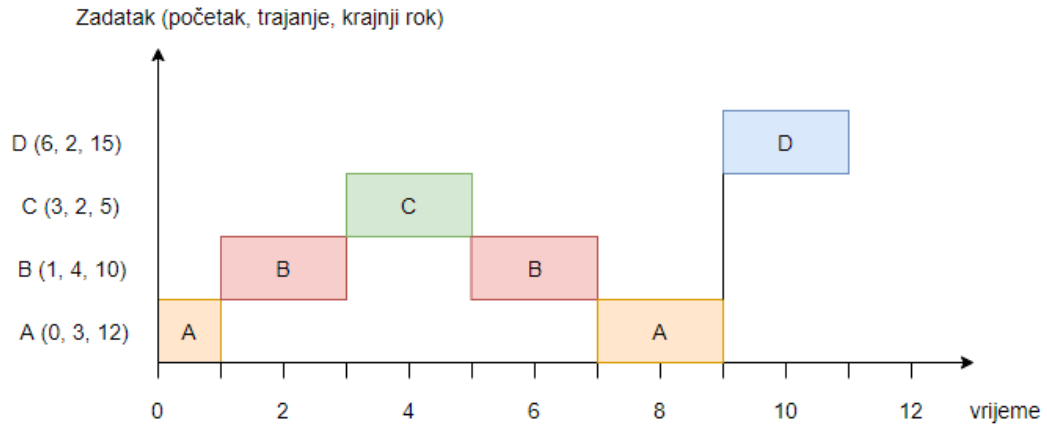
- Mogućnost davanja prednosti važnijim zadatcima
- Rijetke promjene konteksta
- Brz odziv na važne događaje

Nedostatci:

- Mogućnost izgladnjivanja kada zadatci višeg prioriteta dugo traju
- Potreban sekundarni algoritam za zadatke istog prioriteta
- Složenije strukture podataka
- Inverzija prioriteta

4.4. Raspoređivanje po trenutcima krajnjih završetaka

Raspoređivanje po trenutcima krajnjih završetaka, skraćeno EDF (engl. Earliest Deadline First) je dinamički algoritam raspoređivanja koji daje prednost zadatcima s bližim rokom krajnjeg završetka. Kod strogih sustava za rad u stvarnom vremenu, može postojati zahtjev da zadatak mora biti obavljen do određenog trenutka. U takvim sustavima je nedopustivo da neki zadatak ne obavi posao do krajnjeg roka jer bi to značilo da sustav ne radi ispravno. Algoritam raspoređivanja po trenutcima krajnjih završetaka pokušava zadatke rasporediti tako da svi stignu biti dovršeni prije krajnjeg roka. U svakom trenutku izvršavat će se zadatak koji ima najbliži krajnji rok završetka kako bi što prije obavio posao. To podrazumijeva da sustav može dinamički tijekom rada mijenjati prioritete zadataka, zbog čega spada u grupu dinamičkih algoritama. Jezgra sustava nudi funkciju s kojom zadatak može odrediti krajnji rok završetka, te nakon toga počinje obavljati posao. Raspoređivač će se pobrinuti da svi zadatci uspiju na vrijeme završiti tako što će uvijek prednost davati zadatcima s bližim rokom završetka. Također zadatak može biti prekinut ako se pojavi drugi zadatak koji ima još bliži rok završetka (ako se radi o prekidivom raspoređivanju, EDF kao i mnogi drugi algoritmi mogu biti implementirani da budu prekidivi i neprekidivi). Algoritam može garantirati da će svi zadatci uspjeti obaviti svoj posao prije roka pod uvjetom da opterećenje procesora bude uvijek manje od 100%. Ako opterećenje procesora prijeđe 100%, sustav neće moći rasporediti sve zadatke te će najmanje jedan od njih zakasniti. Na sljedećem grafu možemo vidjeti primjer raspoređivanja zadataka EDF algoritmom.



Slika 11. Raspoređivanje po trenutcima krajnjih završetaka

U trenutku t=1 u sustav pristiže B zadatak koji ima bliži krajnji rok nego trenutni A zadatak (10 < 12). Raspoređivač prekida A zadatak te počinje izvoditi zadatak B. U trenutku t=3 u sustav dolazi C zadatak koji ima krajnji rok još bliži nego zadatci A i B (5 < 10 < 12). Raspoređivač uzima C zadatak i izvršava ga do kraja. Nakon toga nastavlja sljedeći zadatak s najbližim krajnjim rokom, zadatak B. Tijekom izvođenja zadatka B u trenutku t=6 u sustav pristiže zadatak D. Zadatak D ima dalji krajnji rok nego zadatak B, pa se B ne prekida već nastavlja s izvođenjem do kraja. U trenutku t=7 zadatak B završava te ostaju još zadatci A i D. Zadatak A ima bliži rok od zadatka D pa će prvi biti obavljen do kraja.

Za analizu i izračun da li je sustav rasporediv pretpostavimo da imamo isključivo periodičke zadatke. Pretpostavimo da je rok krajnjeg završetka ujedno i početak sljedeće periode. Ako period svakog zadatka označimo sa slovom T, a vrijeme potrebno da zadatak obavi svoj posao slovom C (uzimamo vrijeme obavljanja za najgori slučaj), tada će *n* zadataka biti rasporedivo ako vrijedi sljedeća jednadžba:

$$\mathrm{U} = \sum_{i=0}^{n} \frac{C_i}{T_i} \leq 1$$

U obzir treba uzeti i to da promjena konteksta ima neko trajanje, kao i obrada prekida i ostali događaji koji se mogu dogoditi, a troše procesorsko vrijeme. Za primjer pretpostavimo idealan slučaj da promjena konteksta traje zanemarivo kratko te da ne postoje drugi događaji u sustavu. Neka se u sustavu nalaze 3 zadatka sa sljedećim podatcima:

Zadatak A: $C_A = 2, T_A = 4$

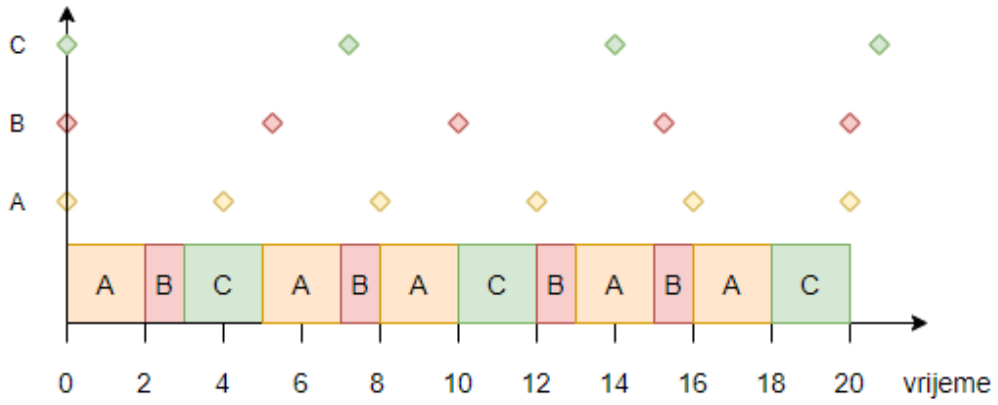
Zadatak B: $C_B = 1, T_B = 5$

Zadatak C: $C_C = 2, T_C = 7$

Uvrštavanjem zadanih parametara u jednadžbu dobivamo sljedeći izraz:

$$U = \frac{2}{4} + \frac{1}{5} + \frac{2}{7} = \frac{69}{70} = 0.9857 \leq 1$$

Što znači da je sustav raspodjeljiv. Ipak u praksi iznos koji je ovako blizu granice ne bismo prihvatili jer postoje i ostali poslovi koje će procesor raditi kao što je promjena konteksta koja će se tijekom rada sustava konstantno događati i uzeti jedan dio vremena. Na sljedećem grafu prikazan je izgled raspoređivanja prethodna 3 zadatka u vremenskoj domeni. Pretpostavka je da su sva 3 zadatka počela izvršavanje u trenutku t=0. Pravokutnicima su označeni početci perioda svakog zadatka.



Slika 12. Grafički prikaz raspoređivanja EDF algoritmom

Funkcije koje nudi jezgra operacijskog sustava koji koristi EDF raspoređivač mogu biti sljedeće:

pokreni_zadatak (funkcija, argumenti);

postavi_krajnji_rok (vrijeme);

čekaj_sljedeću_periodu ( );

završi_zadatak ( );

Nakon što se zadatak pokrene, na početku se poziva funkcija „postavi_krajnji_rok“ kojom se određuje budući trenutak do kojeg zadatak mora završiti svoj posao. Nakon toga slijedi posao kojeg zadatak mora obaviti. Zadano vrijeme može biti apsolutna točka u budućnosti ili relativno vrijeme od trenutka pozivanja funkcije, ovisno o implementaciji. Za slučaj da je zadatak periodičan, postoji funkcija koja se poziva nakon što zadatak obavi posao: „čekaj_sljedeću_periodu“. Pozivom te funkcije zadatak će čekati do početka sljedeće periode, a za to vrijeme se može neki drugi zadatak izvršavati. Završetak periode se u ovom slučaju smatra krajnjim rokom za periodičke zadatke. Nakon što trenutna perioda završi i počne sljedeća, zadatak ponovo postaje pripravan te će uskoro dobit vrijeme za izvršavanje. U ovom primjeru sustav ne garantira da će zadatak početi s izvođenjem odmah na početku periode, ali garantira da će zadatak biti završen prije početka sljedeće periode, odnosno prije kraja trenutne. Ako zadatak nije uspio obaviti svoj posao prije početka sljedeće periode (zbog preopterećenja sustava u tom trenutku), funkcija može vratiti grešku čime dobivamo informaciju koji zadatak je zakasnio. Sljedeći primjer prikazuje način korištenja navedenih funkcija za periodički zadatak i neperiodički zadatak.

```
void periodički_zadatak() {
    // Krajnji rok ujedno
    // određuje i trajanje periode
    postavi_krajnji_rok(200 ms);

    while (1) {
        obavi_posao();
        čekaj_sljedeću_periodu();
    }
}


void NEperiodički_zadatak() {
    postavi_krajnji_rok(200 ms);
    obavi_posao();
}
```

Slika 13. Primjer korištenja EDF algoritma raspoređivanja

Implementacija EDF raspoređivača zahtjeva dodatnu listu pripravnih zadataka u kojoj će bili zadatci kojima je zadan krajnji rok završetka. Zadatci u toj listi moraju biti sortirani na temelju krajnjeg roka. Raspoređivač prvo provjerava da li ima zadataka u listi pripravnih sa zadanim krajnjim rokom, ako ima, uzima prvog i izvršava ga dok ne završi. Ako nema niti jednog zadatka sa zadanim krajnjim rokom, sustav može uzeti zadatak iz druge liste pripravnih koji nemaju zadan krajnji rok završetka. Raspoređivanje tih zadataka može biti nekim drugim algoritmom kao što je npr. RR ili prioritetni. Ako sustav nudi podršku za periodičke zadatke kao što je navedeno tj. nudi funkciju „čekaj_sljedeću_periodu“, implementacija mora sadržavati još jednu listu u kojoj se nalaze zadatci koji su završili posao i čekaju da počne sljedeća perioda. Oni također moraju biti sortirani, na početku oni koji će najmanje čekati tj. oni kojima će perioda uskoro započeti. Raspoređivač na svaki istek kvanta vremena provjerava ima li zadataka kojima počinje sljedeća perioda, ako ima uzima prvi i prebacuje ga u listu pripravnih sa zadanim krajnjim rokom i tako dok ne prebaci sve kojima je počela nova perioda. Možemo zaključiti da implementacija EDF algoritma raspoređivanja zahtjeva jednu ili dvije sortirane liste, a ubacivanje novog člana u sortiranu listu ima složenost O(n). Ta složenost nije idealna pa rješenje može biti

korištenje stabla umjesto liste čime se složenost smanjuje na O(log n) što je i dalje veća složenost nego kod algoritama FCFS i RR kod kojih su sve operacije složenosti O(1).

Prednosti:

- Nije potrebno definirati prioritete zadataka
- Nema čestih promjena konteksta
- Garancija da će svi zadatci biti na vrijeme izvršeni (ako vrijedi navedena jednadžba)

Nedostatci:

- Složenija izvedba
- Manja kontrola izvršavanja, nepredvidivo ponašanje

5. Implementacije sustava

Zbog jednostavnosti, svaki algoritam raspoređivanja je implementiran zasebno. U svakom od 4 direktorija nalazi se jezgra operacijskog sustava koja koristi jedan algoritam za raspoređivanje zadataka. Implementirani algoritmi su: FCFS (neprekidivi raspoređivač), te ostali prekidivi raspoređivači: RR, EDF i prioritetni. Svi navedeni su statički raspoređivači, osim EDF koji dinamički raspoređuje zadatke. Ciljana platforma za implementaciju je ugradbeno računalo Raspberry Pi Zero. Svaka implementacija sadrži jedan korisnički program u datoteci „main.c“, za demonstraciju korištenja jezgrinih funkcija. Svaki sustav sastoji se od nekoliko datoteka sa „C“ programskim kodom, eventualno jedne datoteka s asemblerskim kodom koja sadrži inicijalizaciju procesora, prekidnih vektora, funkcije za upravljanje prekidima, promjenu konteksta i ostalo. Tu su i skripta za prevođenje „Makefile“, te skripta za povezivanje prevedenih datoteka „linker.ld“. Sustav možemo blokovski podijeliti na 3 razine: korisnički program, jezgra operacijskog sustava, te funkcije za pristup sklopovima (ovisne o platformi). Jednostavniji RT operacijski sustavi koji su namijenjeni za mikrokontrolere ne sadrže upravljačke programe (engl. device drivers), već se sklopovlju pristupa direktno pozivajući odgovarajuće funkcije. Te funkcije se nalaze izvan jezgre i neovisne su o operacijskom sustavu. U ovim implementacijama koristit će se samo neko sklopovlje koje je nužno za rad same jezgre i za demonstraciju rada sustava, a to je: serijska veza (UART), jedan vremenski brojač koji odbrojava mikrosekunde (System Timer), programabilni prekidni kontroler (PIC) te pinovi opće namjene (GPIO). Implementacija sustava sastoji se od sljedećih datoteka:

main.c

Datoteka „main.c“ sadrži korisnički program, ali korisnik može po potrebi stvoriti još datoteka sa svojim programima. Datoteka „main.c“ obavezno mora sadržavati funkciju setup koja obavlja inicijalizaciju svih potrebnih resursa kao što su: korišteni sklopovi, I/O jedinice, jezgrini mehanizmi i pokretanje zadataka. Nakon inicijalizacije potrebno je pokrenuti raspoređivač pozivom

funkcije „sched_start“ (ili „task_yield kod FCFS raspoređivanja), nakon čega počinje raspoređivanje i izvršavanje zadataka.

start.S

Datoteka sadrži funkcije za inicijalizaciju procesora, prekidnih vektora, prihvat prekida, promjenu konteksta i maskiranje prekida. Sve funkcije unutar datoteke napisane su u asembleru te je ova datoteka jedina koja sadrži asemblerski kod. Prilikom dovođenja napajanja u sustav, programski kod se počinje izvoditi od procedure „start“ koja se nalazi u navedenoj datoteci. Procedura „start“ postavlja ARM procesor u „system mod“, zabranjuje prekide, inicijalizira stog i prekidne. Nakon toga poziva se funkcija za inicijalizaciju sustava „setup“ koja se nalazi u prethodnoj „main.c“ datoteci.

core.c

Datoteka „core.c“ sadrži jezgru sustava, mehanizam raspoređivanja zadataka te ostale mehanizme potrebne za upravljanje zadatcima. Tu su smještene i strukture podataka kao što su: deskriptori svih zadataka, statički alociran memorijski prostor za stog svakog zadatka, red pripravnih zadataka, te ostali redovi čekanja potrebni za rad sustava.

queue.c

Datoteka sadrži funkcije za manipuliranje redovima čekanja, a to su: inicijalizacija reda, ubacivanja entiteta na kraj reda i uzimanje entiteta s početka reda. Radi se o dinamičkoj implementaciji FIFO strukture koja je izvedena jednostruko povezanom listom. Ove funkcije koristi isključivo jezgra sustava i pozivaju se iz datoteke „core.c“. Sve funkcije imaju složenost O(1).

squeue.c

Kod nekih algoritama raspoređivanja potrebno je imati zadatke sortirane po nekom kriteriju. Kao i prethodna datoteka, ova sadrži funkcije za manipulaciju sortiranim redom čekanja, a to su: inicijalizacija reda, sortirano ubacivanje entiteta, uzimanje s početka reda, te dohvat pokazivača na prvi entitet, ali ne i njegovo uzimanje iz reda. Red je također implementiran dinamički, jednostruko povezanom listom. Sve operacije imaju složenost O(1), osim

funkcije za sortirano ubacivanje entiteta koja ima složenost O(n). Za RT operacijske sustave složenost O(n) nije idealna te se uvijek nastoji smanjiti. To se može napraviti korištenjem stabla umjesto liste, što spušta složenost na O(log n), ali zbog jednostavnosti u ovoj implementaciji koristi se lista.

uart.c

Datoteka sadrži funkcije za korištenje serijske veze, a to su: inicijalizacija sklopa (koristi se brzina prijenosa 115200 bps, bez pariteta, 8-bitne riječi), ispis znakovnog niza, ispis brojeva u heksadekadskom zapisu i ostale.

timer.c

Za odbrojavanje vremena koristi se jedan vremenski brojač „system timer“. Ovaj brojač neprekidno uvećava vrijednost svake mikrosekunde, te nudi mogućnost generiranja prekida nakon isteka zadanog vremenskog intervala. Jezgra koristi brojač za generiranje kvanta vremena i prekidanje zadataka kod prekidivih algoritama raspoređivanja.

gpio.c

Datoteka sadrži funkcije za korištenje pinova opće namjene. Također nudi se mogućnost postavljanja pull-up i pull-down otpornika, kao i generiranje prekida promjenom stanja na određenom pinu.

pic.c

Datoteka sadrži funkcije za upravljanje programabilnim prekidnim sklopom, a to su: povezivanje određenih prekida s procedurom obrade, omogućavanje i zabrana određenih prekida (maskiranje prekida).

api.h

Datoteka sadrži deklaracije funkcija koje čine sučelje prema jezgri operacijskog sustava. Korisnički program poziva jezgrine funkcije isključivo navedene u ovoj datoteci, a to su: pokretanje novog zadatka, odgoda izvođenja za vremenski interval, završetak zadatka, sinkronizacijski mehanizmi i ostalo.

rpi.h

Kao što prethodna datoteka sadrži sučelje prema jezgri sustava, ova datoteka sadrži sučelje prema sklopovlju (UART, Timer, GPIO, itd.).

core.h

Datoteka sadrži definicije struktura podataka koje jezgra koristi, a to su: deskriptori zadataka, struktura konteksta, redovi čekanja i sinkronizacijski mehanizmi. Korisnički program ne poziva funkcije iz ove datoteke, već je namijenjena isključivo jezgri sustava. Također na početku datoteke se nalaze neke konfiguracije sustava koje korisnik može podesiti po potrebi. To su: broj zadataka, količina memorije za stog svakog zadatka, trajanje kvanta vremena i broj sinkronizacijskih mehanizama.

Makefile

Skripta za prevođenje i generiranje slike sustava. Prevođenje se pokreće unosom „make“ naredbe u konzoli (na linux operacijskim sustavima). Datoteka je podešena tako da korisnik može dodavati svoje datoteke s korisničkim programima bez potrebe za izmjenama unutar datoteke „Makefile“.

linker.ld

Skripta za povezivanje prevedenih datoteka (*.o), raspored podataka i programskog koda u memoriji računala.

5.1. Implementacija FCFS raspoređivača

FCFS je statički neprekidivi raspoređivač. Zadatci se izvršavaju po redu prispijeća bez prekidanja. Na sljedećoj slici možemo vidjeti strukture podataka: active_task – pokazivač na aktivni zadatak koji se trenutno izvodi na procesoru, ready_queue – red s pripravnim zadatcima koji čekaju da trenutni završi, task[ ] – statičko polje koje sadrži deskriptore svih zadataka, stack [ ] – memorijski prostor rezerviran za stogove svih zadataka.

```
static task_t *active_task;
static queue_t queue_ready;
static task_t task[TASK_COUNT];
static char stack[TASK_COUNT][STACK_SIZE] \
    __attribute__((aligned(8)));
```

Slika 14. Strukture podataka kod FCFS raspoređivanja

Datoteka „core.c“ sadrži sljedeće funkcije:

core_init – inicijalizacija jezgre, poziva se jednom na početku. Funkcija u for petlji prolazi sve deskriptore zadataka te ih označava kao neaktivne. Nakon toga inicijalizira se red čekanja za pripravne zadatke, te se stvara neradni zadatak (engl. idle task). Neradni zadatak izvršava se kada nema niti jednog drugog zadatka u redu pripravnih zadataka.

task_create – funkcija stvara i pokreće novi zadatak sa sljedećim parametrima: fn – procedura zadatka, arg – argumenti koji će se predati zadanoj proceduri. Prvo se prolazeći redom u for petlji pronalazi slobodan deskriptor. Ako nema slobodnog deskriptora, nije moguće stvoriti novi zadatak. Nakon pronalaska deskriptora, obavlja se njegova inicijalizacija (označava se kao aktivan, alocira se stog, te kontekst procesora). Nakon toga deskriptor zadatka se ubacuje na kraj reda pripravnih zadataka. Zadatak postaje aktivan i počinje se izvršavati kada dođe na red.

task_yield – funkcija omogućuje prepuštanje procesora sljedećem zadatku. Trenutni zadatak ne mora završiti da bi sljedeći došao na red, već može dobrovoljno prepustiti procesor sljedećem. Ovime se postiže

kooperativni višezadaćni rad. Funkcija sprema aktivni zadatak na kraj reda pripravnih, zatim uzima drugi (prvi iz reda) te promjenom konteksta nastavlja se izvršavati sljedeći zadatak.

task_exit – pozivom ove funkcije zadatak završava s izvođenjem. Funkcija se poziva automatski kada glavna procedura zadatka završi, a može ju korisnik pozvati u bilo kojem trenutku ako želi trajno zaustaviti izvođenje trenutnog zadatka. Prvo se označava u deskriptoru trenutnog zadatka da zadatak više nije aktivan (deskriptor postaje slobodan), nakon toga zadatak se ne ubacuje ponovo na kraj reda, već se odmah uzima sljedeći koji nastavlja s izvođenjem nakon promjene konteksta. Nakon poziva ove funkcije, zadatak više ne postoji u sustavu.

```
#define TICK_TIME   1000
#define TASK_COUNT  8
#define STACK_SIZE  (4 * 1024)

typedef void (*func)();
typedef unsigned int uint;

typedef struct {
    uint r0, r1, r2, r3, r4, r5, r6, r7;
    uint r8, r9, r10, r11, cpsr, lr, pc;
} context_t;

typedef struct task_t {
    uint *sp;
    uint id;
    uint status;
    struct task_t *next;
} task_t;

typedef struct {
    task_t *first;
    task_t *last;
} queue_t;


void core_init();
void queue_init(queue_t *q);
void queue_push(queue_t *q, task_t *t);
task_t* queue_pop(queue_t *q);
```

Slika 15. Sadržaj datoteke core.h

```
void task_create(func fn, void *arg) {
    int i;
    SYS_ENTRY();

    for (i = 0; i < TASK_COUNT; ++i)
        if (!task[i].status)
            break;

    if (i >= TASK_COUNT) {
        SYS_EXIT();
        return;
    }

    task[i].id = i;
    task[i].sp = (uint*)(stack[i] + STACK_SIZE - 4);
    task[i].sp -= sizeof(context_t);
    task[i].status = TASK_READY;

    context_t *ctx = (context_t*)task[i].sp;
    ctx->cpsr = CPSR_MODE_SYS;
    ctx->lr = (uint)task_exit;
    ctx->pc = (uint)fn;
    ctx->r0 = (uint)arg;

    queue_push(&queue_ready, &task[i]);
    SYS_EXIT();
}

void task_yield() {
    SYS_ENTRY();
    task_t *prev = active_task;

    if (prev && prev->status == TASK_READY)
        queue_push(&queue_ready, prev);

    active_task = queue_pop(&queue_ready);
    if (active_task)
        cpu_switch(prev, active_task);
    SYS_EXIT();
}

void task_exit() {
    cpu_interrupts(1);
    active_task->status = TASK_UNUSED;
    task_yield();
}
```

Slika 16. Sadržaj datoteke core.c (1)

```
#define TASK_UNUSED     0
#define TASK_READY      1

#define SYS_ENTRY()     int ei = cpu_interrupts(1)
#define SYS_EXIT()      cpu_interrupts(ei)


static task_t *active_task;
static queue_t queue_ready;
static task_t task[TASK_COUNT];
static char stack[TASK_COUNT][STACK_SIZE] \
    __attribute__((aligned(8)));


static void idle() {
    while (1) {
        task_yield();
    }
}

void core_init() {
    int i;
    for (i = 0; i < TASK_COUNT; ++i)
        task[i].status = TASK_UNUSED;

    active_task = 0;
    queue_init(&queue_ready);
    task_create(idle, 0);
}
```

Slika 17. Sadržaj datoteke core.c (2)

Na slici 15 možemo vidjeti sadržaj datoteke „core.h“ s definicijama jezgrinih struktura podataka, te na slikama 16 i 17 možemo vidjeti sadržaj datoteke „core.c“ koja sadrži prethodno opisane funkcije. Makro funkcija SYS_ENTRY zabranjuje prekide, a poziva se na početku jezgrine funkcije. Time se osigurava kontrolirani pristup jezgrinim strukturama podataka. na kraju jezgrinih funkcija poziva se funkcija SYS_EXIT koja obnavlja zastavicu prekida.

5.2. Implementacija RR raspoređivača

FCFS algoritam ima glavni nedostatak, a to je izgladnjivanje zadataka ako trenutni dugo traje ili zbog greške ne prepusti procesor sljedećem. Zbog zadataka koji dugo traju odziv sustava može biti spor jer ostali zadatci ne mogu dobiti dovoljno vremena da obave posao. Taj problem možemo riješiti kružnim raspoređivanjem koje garantira da će svi zadatci dobiti jednako vremena za izvođenje. Količinu tog vremena određuje zadani kvant vremena koji će u ovom slučaju biti podešen na 1 milisekundu. Po potrebi korisnik može postaviti drugačiju vrijednost. Mehanizam raspoređivanja je gotovo isti kao u prethodnom algoritmu, osim što zadatci ne moraju završiti niti ručno prepuštati procesor sljedećem, već to obavlja raspoređivač automatizirano. Za prekidanje i promjenu trenutnog zadatka koristi se vremenski sklop koji postavlja prekid na svaki istek kvanta vremena. Procedura obrade prekida obavlja promjenu konteksta. Ovaj mehanizam ostvaren je dodavanjem sljedećih funkcija u datoteci „core.c“.

task_tick – Slika 18. Procedura koja se poziva svaki put kada istekne kvant vremena. Unutar procedure vremenski sklop se ponovo konfigurira za budući prekid i ažurira se vrijeme sustava (varijabla sys_time, pomnožena s trajanjem kvanta vremena daje vrijeme proteklo od pokretanja sustava). Nakon toga se provjerava da li je nekom zadatku istekao interval čekanja, ako je zadatak se ponovo ubacuje u red pripravnih zadataka. Zadatak može biti uvršten u red čekanja na istek vremenskog intervala pozivom funkcije „task_sleep“ (više u nastavku). Na kraju se poziva raspoređivač koji će obaviti promjenu zadatka „task_yield“.

task_sleep – Slika 19. Dodatna jezgrina funkcija koja nije implementirana u prethodnom raspoređivaču, a služi za odgodu izvođenja zadatka za neki vremenski interval. Zadatak može zatražiti, nakon što obavi neki posao, nastavak izvođenja nakon isteka nekog vremenskog intervala (Često se koristi kod periodičkih zadataka). U tom slučaju u deskriptor zadatka se zapisuje apsolutno vrijeme završetka čekanja (trenutno vrijeme sustava uvećano za interval odgode izvođenja). Zatim se zadatak označava kao

neaktivan (stanje TASK_SLEEP) i ne uvrštava se u red pripravnih zadataka. Nakon toga poziva se raspoređivač „task_yield“ koja će obaviti promjenu konteksta i nastaviti izvoditi sljedeći zadatak. Nakon isteka svakog kvanta vremena generira se prekid i poziva procedura „task_tick“. Kako je prethodno opisano, ažurira se vrijeme sustava (varijabla sys_time) te se provjerava da li je kojem zadatku isteklo vrijeme čekanja. Ova provjera obavlja se u petlji prolazom kroz sve deskriptore zadataka što ima složenost O(n). Ovo nije dobra praksa, pogotovo u funkciji obrade prekida koja se često poziva kod RR raspoređivača, svaki kvant vremena. Zbog toga je preporučljivo koristiti sortirani red čekanja pri čemu bi se provjeravao samo prvi član u redu čekanja. Zbog jednostavnosti ostavljena je implementacija „for“ petljom.

```
static void task_tick() {
    int i;
    timer_set(TICK_TIME);
    ++sys_time;

    if (!sched_run)
        return;

    for (i = 0; i < TASK_COUNT; ++i) {
        if (task[i].status == TASK_SLEEP) {
            if (sys_time >= task[i].sleep) {
                task[i].status = TASK_READY;
                queue_push(&queue_ready, &task[i]);
            }
        }
    }
    task_yield();
}
```

Slika 18. Funkcija task_tick

```
void task_sleep(uint ms) {
    SYS_ENTRY();
    if (ms) {
        active_task->sleep = sys_time + ms;
        active_task->status = TASK_SLEEP;
    }
    task_yield();
    SYS_EXIT();
}
```

Slika 19. Funkcija task_sleep

5.3. Implementacija prioritetnog raspoređivača

Uvođenjem prioriteta omogućujemo brži odziv na važnije zadatke. Kod statičkog prioritetnog raspoređivanja korisnik kod pokretanja zadatka zadaje prioritet. Na procesoru će uvijek biti zadatak koji trenutno ima najviši prioritet. Prioritetno raspoređivanje možemo implementirati tako da red pripravnih zadataka bude sortiran. Time se zadatci ne ubacuju na kraj reda pripravnih kao u prethodnim algoritmima, već se ubacuju sortirano, na temelju prioriteta, a raspoređivač uvijek uzima prvog (jer ima najviši prioritet). Druga izvedba koja će se koristiti u ovoj implementaciji je da za svaki prioritet postoji zaseban red čekanja. Ovime se omogućuje brže ubacivanje zadataka u red čekanja O(1), za razliku od sortiranog reda O(n). Međutim uzimanje novog zadatka ima složenost O(n) jer sada algoritam mora pretražiti redove od najvišeg prioriteta do najnižeg i uzeti prvi zadatak iz reda koji nije prazan. Za manji broj prioriteta, ovaj način je vrlo efikasan i složenost O(n) postaje sumjerljiva sa O(1).

```
static void task_yield() {
    int i;
    task_t *prev = active_task;

    if (prev && prev->status == TASK_READY)
        queue_push(&queue_ready[prev->prio], prev);

    for (i = 0; i < PRIO_COUNT; ++i){
        active_task = queue_pop(&queue_ready[i]);
        if (active_task)
            break;
    }

    if (active_task)
        cpu_switch(prev, active_task);
}
```

Slika 20. Prioritetni raspoređivač

U programskom kodu možemo uočiti malu promjenu (slika 20), a to je da se kod uzimanja novog zadatka u „for“ petlji pretražuje od reda s najvišim prioritetom (prioritet 0) do reda s najnižim prioritetom. Iz prvog reda koji nije prazan uzima se zadatak i nastavlja izvršavati. To je ujedno i zadatak s najvišim prioritetom u sustavu. Također u funkciji za pokretanje novog zadatka potrebno je dodati na kraju poziv raspoređivača „task_yield“, za slučaj da novi zadatak koji je upravo pokrenut ima veći prioritet. U tom slučaju on se počinje odmah izvršavati. U situaciji da postoji više zadataka s istim prioritetom, algoritam radi po principu prethodnog RR algoritma (zadatci s jednakim prioritetom raspoređuju se kružno). Alternativa može biti i FCFS algoritam za zadatke s istim prioritetom. Korisnik u datoteci „core.h“ može odabrati broj mogućih prioriteta. Najviši prioritet je uvijek 0, a najniži N-1 pri čemu je N zadani broj mogućih prioriteta. Neradni zadatak (engl. idle task) uvijek ima najniži prioritet. U deskriptor zadataka (datoteka core.h) dodana je varijabla „prio“ koja označava prioritet zadatka (slika 21).

```
typedef struct task_t {
    uint *sp;
    uint id;
    uint prio;
    uint sleep;
    uint status;
    struct task_t *next;
} task_t;
```

Slika 21. Deskriptor zadataka kod prioritetnog raspoređivanja

5.4. Implementacija EDF raspoređivača

EDF raspoređivač dinamički raspoređuje zadatke na temelju krajnjeg roka završetka. Algoritam tijekom rada sustava određuje prioritete zadataka tako da najviši prioritet ima zadatak sa najbližim rokom krajnjeg završetka. Krajnji rok završetka određuje sam zadatak pozivom određene jezgrine funkcije. Zadatak može biti periodički ili neperiodički. Kod periodičkih zadataka krajnjim rokom se smatra završetak periode tj. početak iduće. Jezgra sustava nudi sljedeće funkcije (slika 22).

```
void task_start(func fn, void *arg);
void task_setdl(uint period);
uint task_wait();
void task_exit();
```

Slika 22. Jezgrine funkcije za korištenje EDF raspoređivača

Jezgrine strukture podataka potrebno je proširiti s dva reda čekanja. U prvom redu će se nalaziti pripravni zadatci koji imaju zadani krajnji rok završetka (slika 23. red čekanja je nazvan „q_edl_ready“). Zadatci će biti sortirani tako da na početku budu oni kojima je krajnji rok najbliži. Raspoređivač uvijek provjerava ima li zadataka u tom redu čekanja, ako nema, uzima prvi iz „običnog“ reda pripravnih i dalje se raspoređuju po RR algoritmu (ili bilo kojim drugim algoritmom). Ako postoji zadatak sa zadanim krajnjim rokom završetka, raspoređivač ga uzima i počinje izvoditi (jer takvi zadatci imaju prednost kako bi što prije obavili posao). Slično kao i kod prioritetnog raspoređivanja, na procesoru će se uvijek nalaziti zadatak s najbližim krajnjim rokom završetka, ako takav zadatak postoji. Zadatak sa zadanim rokom može biti prekinut jedino ako se pojavi drugi zadatak sa još bližim krajnjim rokom završetka. Drugi red čekanja odnosit će se na periodičke zadatke koji su obavili svoj posao i čekaju početak iduće periode (slika 23. red čekanja je nazvan „q_edl_wait“). Krajnji rok završetka zadaje se funkcijom „task_setdl“. Ako se radi o periodičkom zadatku, nakon što obavi posao, zadatak poziva funkciju „task_wait“. Time se zadatak premješta u

drugi red čekanja gdje čeka na početak iduće periode. Kada počne iduće perioda, sustav će sam premjestiti zadatak ponovo u prvi red čekanja. Zadatci su također sortirani tako da su na početku oni kojima će uskoro početi sljedeća perioda. Nakon početka nove periode nije potrebno ponovo postavljati krajnji rok završetka funkcijom „task_setdl“, već se podrazumijeva da je krajnji rok ujedno i završetak tekuće periode. U nastavku je prikazan programski kod dviju funkcija koje se odnose na EDF algoritam raspoređivanja (slika 24).

```
static volatile uint sched_run;
static volatile uint sys_time;
static task_t *active_task;
static queue_t queue_ready;
static queue_t q_edl_wait;
static queue_t q_edl_ready;
static task_t task[TASK_COUNT];
static char stack[TASK_COUNT][STACK_SIZE] \
    __attribute__((aligned(8)));
```

Slika 23. Strukture podataka EDF raspoređivača

Funkcija „task_setdl“ zapisuje period zadatka (koji je ujedno i krajnji rok) u deskriptor zadatka, nakon toga računa se i zapisuje apsolutni trenutak krajnjeg završetka (trenutno vrijeme uvećano za period), te se u varijablu status zadatka označava da zadatak treba uvrstiti u red pripravnih zadataka sa zadanim krajnjim rokom „q_edl_ready“. Nakon toga se poziva raspoređivač. Nakon što zadatak obavi posao može završiti pozivom funkcije „task_exit“. Ako je zadatak periodičan, tada ne završava već poziva funkciju „task_wait“, čime se zadatak nakon poziva raspoređivača, uvrštava u red zadataka koji čekaju početak iduće periode „q_edl_wait“. Slika 24 prikazuje programski kod jezgrinih funkcija „task_setdl“ i „task_wait“.

```
void task_setdl(uint period) {
    SYS_ENTRY();
    active_task->period = period;
    active_task->time_p = sys_time + period;
    active_task->status = TASK_EDL_R;
    task_sched();
    SYS_EXIT();
}

uint task_wait() {
    uint ret = 0;
    SYS_ENTRY();
    active_task->status = TASK_EDL_W;
    if (sys_time > active_task->time_p) {
        ret = 1;
    }
    task_sched();
    SYS_EXIT();

    if (ret)
        uart_print("ERROR: Deadline fail.\r\n");

    return ret;
}
```

Slika 24. Funkcije task_setdl i task_wait

U nastavku (slika 25) možemo vidjeti funkciju raspoređivača te funkciju „task_tick“ koja se poziva svaki put kada istekne kvant vremena, kao u prethodnim algoritmima uz neke izmjene. Raspoređivač sprema trenutni zadatak u red čekanja, ovisno o varijabli „status“ u deskriptoru zadatka koja je prethodno postavljena. Jezgra sadrži 3 reda čekanja: pripravni zadatci bez zadanog roka krajnjeg završetka (queue_ready), pripravni zadatci sa zadanim rokom krajnjeg završetka (q_edl_ready), te zadatci koji su obavili posao i čekaju sljedeću periodu (q_edl_wait). Nakon toga raspoređivač provjerava ima li pripravnih zadataka sa zadanim krajnjim rokom završetka, ako ima uzima prvog iz reda i počinje ga izvršavati. Ako nema, uzima prvog iz reda pripravnih (queue_ready) u kojem će se uvijek nalaziti barem jedan zadatak, a to je neradni zadatak.

```
static void task_yield() {
    task_t *prev = active_task;
    if (prev) {
        switch (prev->status) {
        case TASK_READY:
            queue_push(&queue_ready, prev);
            break;
        case TASK_EDL_W:
            squeue_put(&q_edl_wait, prev);
            break;
        case TASK_EDL_R:
            squeue_put(&q_edl_ready, prev);
            break;
        }
    }

    active_task = squeue_pop(&q_edl_ready);
    if (!active_task)
        active_task = queue_pop(&queue_ready);

    if (active_task)
        cpu_switch(prev, active_task);
}

static void task_tick() {
    int sched = 0;
    task_t *tmp;

    timer_set(TICK_TIME);
    sys_time += 10;

    if (!sched_run)
        return;

    while (1) {
        tmp = squeue_peek(&q_edl_wait);
        if (!tmp || (sys_time < tmp->time_p))
            break;

        sched = 1;
        tmp = squeue_pop(&q_edl_wait);
        tmp->status = TASK_EDL_R;
        tmp->time_p += tmp->period;
        squeue_put(&q_edl_ready, tmp);
    }

    if (sched || !squeue_peek(&q_edl_ready))
        task_yield();
}
```

Slika 25. EDF raspoređivač

Svaki put nakon što istekne kvant vremena poziva se funkcija „task_tick“. Unutar funkcije prvo se ažurira varijabla trenutnog vremena sustava „sys_time“. Nakon toga provjerava se da li nekom zadatku počinje nova perioda, ako da, zadatak se premješta iz reda „q_edl_wait“ u red „q_edl_ready“, te se poziva raspoređivač koji obavlja promjenu konteksta i počinje izvoditi sljedeći zadatak. Strukture podataka u datoteci core.h izgledaju vrlo slično kao u prethodnim implementacijama.

```
#define TASK_COUNT      8
#define STACK_SIZE      4*1024

typedef void (*func)();
typedef unsigned int uint;

typedef struct {
    uint r0, r1, r2, r3, r4, r5, r6, r7;
    uint r8, r9, r10, r11, cpsr, lr, pc;
} context_t;

typedef struct task_t {
    uint *sp;
    uint id;
    uint status;
    uint period;
    uint time_p;
    struct task_t *next;
} task_t;

typedef struct {
    task_t *first;
    task_t *last;
} queue_t;


void core_init(func fn);

void queue_init(queue_t *q);
void queue_push(queue_t *q, task_t *t);
task_t* queue_pop(queue_t *q);

void squeue_init(queue_t *q);
void squeue_put(queue_t *q, task_t *t);
task_t* squeue_pop(queue_t *q);
task_t* squeue_peek(queue_t *q);
```

Slika 26. EDF raspoređivač - datoteka core.h

6. ZAKLJUČAK

Od četiri navedena algoritma raspoređivanja, kao najbolji pokazao se algoritam prioritetnog raspoređivanja. Prioritetno raspoređivanje se koristi u skoro svim operacijskim sustavima za rad u stvarnom vremenu. Raspoređivanje po redu prispijeća i kružno raspoređivanje, iako su vrlo jednostavni, ne nude jednu bitnu stvar, a to je istiskivanje manje važnih zadataka kako bi oni važniji dobili prednost. Zbog toga i jedan i drugi algoritam imaju slab odziv na važnije događaje kod većeg broja zadataka, a to nije poželjno kod rada u stvarnom vremenu. Raspoređivanje po trenutcima krajnjih završetaka nudi dobar odziv, ali nije praktično za upotrebu. Kada želimo da neki važan zadatak ima prednost nad svim ostalim zadatcima, lakše je dati mu veći prioritet od ostalih nego određivati krajnji trenutak završetka. Stvari se dodatno kompliciraju kada imamo više zadataka koji nisu periodički pri čemu nije lako odrediti krajnje završetke. Zbog toga se ovaj algoritam rijetko koristi, a dodatni nedostatak mu je kao i svim dinamičkim algoritmima, složenija implementacija i dulje vrijeme odluke prilikom promjene zadataka. Najpraktičniji algoritam ostaje prioritetno raspoređivanje. Važni zadatci imaju brzi odziv bez obzira na opterećenje sustava i broj zadataka. Prioritetno raspoređivanje je pogodno za periodičke i neperiodičke zadatke, nudi jednostavno sučelje za dodjelu prioriteta, te se zbog toga i najviše koristi. Jedina stvar na koju moramo paziti je to da zadatci s visokim prioritetom ne traju dugo kako ne bi došlo do izgladnjivanja ostalih zadataka.

7. Literatura

Leonardo Jelenković, Sustavi za rad u stvarnom vremenu,

Zagreb, 2019

Leonardo Jelenković, Operacijski sustavi za ugrađena računala,

Zagreb, 2019

https://www.guru99.com/cpu-scheduling-algorithms.html

https://www.guru99.com/real-time-operating-system.html

https://en.wikipedia.org/wiki/Real-time_operating_system

https://en.wikipedia.org/wiki/Scheduling_(computing)

https://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling

https://www.tutorialandexample.com/cpu-scheduling/

https://www.tutorialandexample.com/scheduling-algorithms/

https://www.tutorialandexample.com/process-schedulers-and-process-queue/

https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm

https://www.embedded.com/inter-task-communication-and-synchronization/

https://www.studytonight.com/operating-system/comparision-scheduling-algorithms

https://www.engineersgarage.com/article_page/rtos-real-time-operating-system/

https://automaticaddison.com/round-robin-vs-function-queue-scheduling-embedded-software-architecture/

Sažetak

Algoritmi raspoređivanja zadataka u operacijskim sustavima za rad u stvarnom vremenu

Sustavi za rad u stvarnom vremenu moraju osim logičke ispravnosti zadovoljiti i zadana vremenska ograničenja. Za jednostavne sustave može se koristiti upravljačka petlja ili upravljanje zasnovano na događajima (pomoću prekida). Kod složenijih sustava praktičnije je koristiti operacijski sustav za rad u stvarnom vremenu. Takvi operacijski sustavi nude više različitih algoritama raspoređivanja zadataka koji mogu pomoći da sustav što efikasnije obavlja poslove na vrijeme. Raspoređivanje može biti prekidivo, pri čemu zadatak u izvođenju može biti prekinut od strane drugog zadatka, ili neprekidivo, pri čemu zadatak koji se izvodi ne može biti prekinut već mora biti obavljen do kraja. Također algoritme raspoređivanja možemo podijeliti na statičke (raspoređivanje na temelju unaprijed zadanih parametara) i dinamičke (sustav tijekom rada mijenja parametre raspoređivanja). Neki od najčešće korištenih algoritama raspoređivanja su: FCFS (raspoređivanje po redu prispijeća), RR (kružno raspoređivanje podjelom vremena), prioritetno raspoređivanje, te EDF (raspoređivanje po vremenu krajnjeg završetka).

Ključne riječi:

Algoritmi raspoređivanja, operacijski sustav, sustavi za rad u stvarnom vremenu, raspoređivanje zadataka

Summary

Task scheduling algorithms in real-time operating systems

In addition to logical correctness, real-time systems must also meet the set time limits. For simple systems, a control loop or event-based control (using interrupts) can be used. For more complex systems, it is more convenient to use the real-time operating system. Real-time operating systems offer a number of different task scheduling algorithms that can help the system perform tasks on time as efficiently as possible. Task scheduling can be preemptive, whereby a task in progress can be interrupted by another task, or cooperative, whereby the task being performed cannot be interrupted but must be completed. Also, scheduling algorithms can be divided into static (scheduling based on predefined parameters) and dynamic (the system changes scheduling parameters during operation). Some of the most commonly used scheduling algorithms are: FCFS (First Come First Serve), RR (Round Robin), priority scheduling, and EDF (Earliest Deadline First).

Keywords:

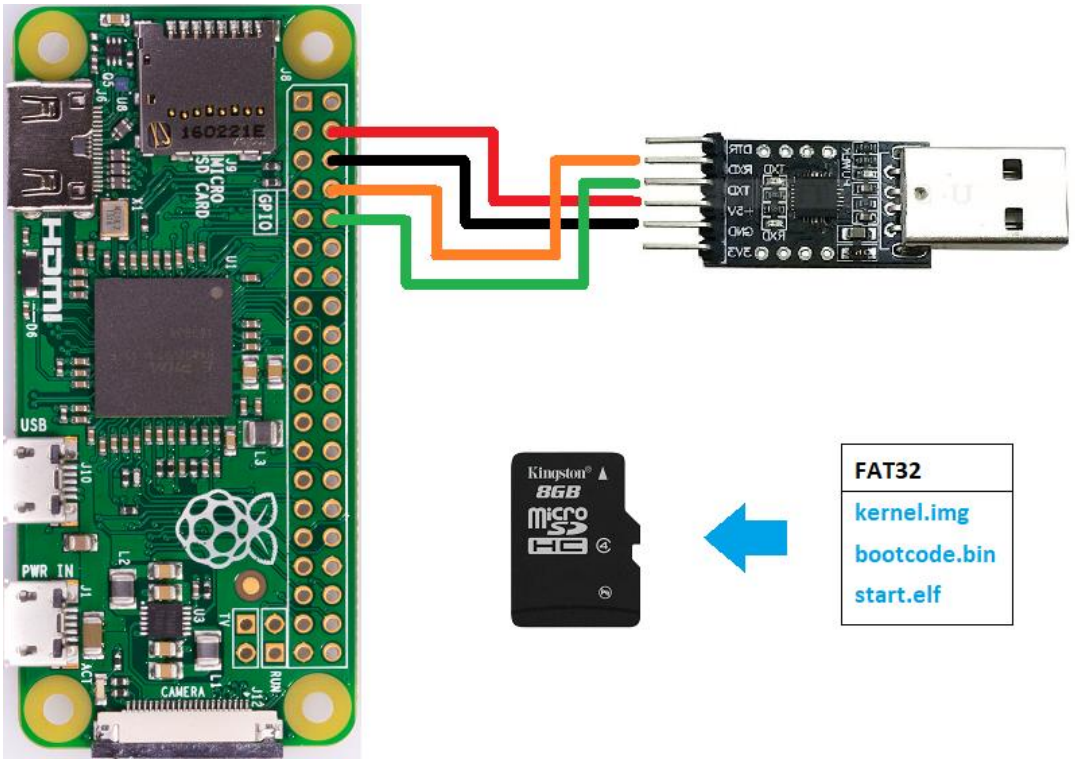
Scheduling algorithms, operating system, real-time systems, task scheduling

Skraćenice

RTS	Real Time System (sustav za rad u stvarnom vremenu)
OS	Operating system (operacijski sustav)
IO	Input/Output (ulaz/izlaz)
RPi	Raspberry Pi (ugradbeno računalo)
FCFS	First Came First Serve (raspoređivanje po redu prispijeća)
RR	Round Robin (kružno raspoređivanje)
EDF	Earliest Deadline First (raspoređivanje po trenutcima krajnjih završetaka)

Dodatak A: Upute za instalaciju

Sustav je implementiran na ugradbenom računalu Raspberry Pi Zero (skraćeno RPi), te se preporučuje upotreba isključivo tog modela. Ne postoji garancija da će sustav ispravno raditi na drugim Raspberry Pi modelima. Za pokretanje sustava osim RPi računala potrebna je memorijska SD kartica bilo kojeg kapaciteta, te USB-UART konverter kako bi se mogla ostvariti komunikacija sa PC računalom. USB-UART konverter ujedno služi i kao izvor napajanja za RPi. Sljedeća slika prikazuje način spajanja.



Slika 27. Priprema sustava za rad

Prevođenje programskog koda preporučuje se obavljati na operacijskim sustavima Linux. U direktoriju s programskim kodom nalazi se nekoliko poddirektorija, u svakom je implementiran po jedan algoritam raspoređivanja. Procedura za instalaciju je sljedeća:

1) Prevođenje programskog koda: U direktoriju s programskim kodom (nakon što se pozicioniramo u poddirektorij odabranog algoritma) postoji datoteka s nazivom 'Makefile'. To je skripta za prevođenje koja se pokreće naredbom „**make**“. Ako sve prođe dobro, izgenerirat će se datoteka s nazivom „kernel.img“ koja sadrži sav izvršni kod (operacijski sustav i korisnički program). Za prevođenje je potrebno imati instaliran prevoditelj za ARM arhitekturu. Ako nije, to na Linux operacijskim sustavima možemo obaviti naredbom:

   sudo apt install gcc-arm-none-eabi

2) Instaliranje na Raspberry Pi Zero: SD memorijsku karticu potrebno je formatirati u FAT32 datotečni sustav. Nakon toga na SD karticu potrebno je kopirati datoteke:

   kernel.img - izvršni kod iz prethodnog koraka

   start.elf - bootlaoder

   bootcode.bin - bootloader

   Datoteke 'start.elf' i 'bootcode.bin' možemo preuzeti s web stranice:

   https://github.com/raspberrypi/firmware/tree/master/boot

3) Nakon kopiranja prethodnih datoteka na SD karticu sustav je instaliran. Sada možemo umetnuti SD karticu u računalo Raspberry Pi, te dovođenjem napajanja sustav počinje s radom.