Poglavlje 22

Uvod u operativne sisteme za rad u realnom vremenu

22.1 Softverske arhitekture za embeded sisteme

Više puta je pomenuto do sada, da se funkcionalnosti nekog sistema često mogu predstaviti kao niz zasebnih logičkih celina i na taj način značajno olakšati implementaciju takvog sistema. Međutim, ponekad je poželjno (ili čak neophodno) ove logičke celine organizovati tako da se postigne neki vid paralelnog izvršavanja. Pri tome, treba uzeti u obzir da, budući da govorimo o embeded platformama koje poseduju jedan procesor (mikrokontroler), ne postoji mogućnost pravog paralelizma. Iz tog razloga, koriste se različite tehnike koje omogućavaju naizgled istovremeno izvršavanje ovih logičkih celina – tzv. *pseudo-paralelni* rad. Ovakav vid izvršavanja se zasniva na vremenskom multipleksu, gde se u svakom trenutku izvršava samo jedan zadatak (eng. *task*), pri čemu se, pod unapred definisanim pravilima, izvršavanje prebacuje sa jednog zadatka na drugi.

Za implementaciju pseudo-paralelnog rada sistema koriste se različite softverske arhitekture. Pod softverskim arhitekturama podrazumevaju se upravljačke strukture koje čine da celokupan sistem radi ispravno.

Izbor arhitekture u najvećoj meri zavisi od odnosa kontrolabilnosti koja želi da se postigne i vremena reakcije samog sistema. Tako na primer, kod sistema koji ne zahteva značajnu kontrolabilnost, arhitektura će biti jednostavnija. Sa druge strane, ukoliko sistem treba da reaguje na niz različitih događaja u kratkim vremenskim intervalima, njegova arhitektura će tada biti složenija.

U literaturi se razlikuju četiri osnovne klase. Počevši od najjednostavnije ka najsloženijoj, to su:

- *round-robin* arhitektura
- *round-robin* arhitektura sa prekidima

- arhitektura sa planiranjem baziranom na redovima (*function-queue-scheduling*)
- arhitektura zasnovana na operativnom sistemu za rad u realnom vremenu (*real-time operating system* – RTOS)

U nastavku je dato kratko objašnjenje svake arhitekture ponaosob.

22.1.1 *Round-robin* arhitektura

Round-robin arhitekura predstavlja najjednostavniju moguću softversku arhitekturu. Prototip koda koji implementira *round-robin* arhitekturu je dat na listingu 22.1.

```
void main()
{
    // initialization

    while(1)
    {
        if (/*uredjaj_1 zahteva obradu*/)
        {
            /*obrada za uredjaj_1*/
        }

        if (/*uredjaj_2 zahteva obradu*/)
        {
            /*obrada za uredjaj_2*/
        }
        ...
        if (/*uredjaj_n zahteva obradu*/)
        {
            /*obrada za uredjaj_n*/
        }
    }

    return 0;
}
```

Listing 22.1: Prototip koda za round-robin arhitekturu

Kao što se može primetiti sa listinga 22.1 u pitanju je vrlo jednostavna struktura koja ne koristi prekide i ne generiše značajnije kašnjenje prilikom svog izvršavanja. Iz tog razloga, ova arhitektura, na prvi pogled, deluje kao prilično dobar izbor. Međutim, to u praksi vrlo često nije slučaj, budući da se pokazuje da je jednostavnost zapravo jedina prednost ovakvog tipa arhitekture. Neki od problema koji se susreću u praksi su:

- ukoliko neki od uređaja zahteva obradu brže nego što se izvrši prolazak kroz glavnu petlju, sistem neće funkcionisati;

- ukoliko obrada nekog uređaja zahteva značajno vreme, rad celokupnog sistema će biti usporen za taj iznos; ovo se ne mora striktno tumačiti kao neispravan sistem, ali u većini situacija ovakvo ponašanje nije poželjno;
- sama arhitektura je vrlo fragilna i teška za održavanje i podešavanje ukoliko je broj različitih zadataka varijabilan.

Prema tome, zaključak je, da se *round-robin* arhitektura koristi isključivo za aplikacije u kojima je jednostavnost jedini prioritet.

22.1.2 *Round-robin* arhitektura sa prekidima

Round-robin arhitektura sa prekidima predstavlja proširenje prethodne arhitekture. Prototip koda za round-robin arhitekturu sa prekidima je dat na listingu 22.2.

```
/*globalne promenljive*/
int8_t dev1_flag = 0;
int8_t dev2_flag = 0;
...
int8_t devN_flag = 0;

ISR(/*uredjaj_1 vektor prekida*/)
{
    /*obrada visokog prioriteta*/
    dev1_flag = 1;
}

ISR(/*uredjaj_2 vektor prekida*/)
{
    /*obrada visokog prioriteta*/
    dev2_flag = 1;
}

...

ISR(/*uredjaj_N vektor prekida*/)
{
    /*obrada visokog prioriteta*/
    devN_flag = 1;
}

void main()
{
    while(1)
    {
        if (dev1_flag)
        {
            dev1_flag = 0;
```

```
            /*zavrsetak obrade*/
        }

        if (dev2_flag)
        {
            dev2_flag = 0;
            /*zavrsetak obrade*/
        }
        ...
        if (devN_flag)
        {
            devN_flag = 0;
            /*zavrsetak obrade*/
        }

    }

    return 0;
}
```

Listing 22.2: Prototip koda za round-robin arhitekturu sa prekidima

Kod ove arhitekture, prekidne rutine vrše obradu visokog prioriteta, nakon čega postavljaju indikatorske promenljive (eng. *flag*). Glavna petlja zatim resetuje indikatorske promenljive i završava obradu zahteva.

Ovaj tip arhitekture omogućava nešto veću kontrolabilnost nad prioritetima, budući da se koriste prekidne rutine koje su uvek većeg prioriteta od ostatka programa. Na taj način je moguće, na neki način, napraviti raspored prioriteta u sistemu.

Problemi ovakve arhitekture jesu problemi deljenih resursa u sistemu. Pored toga, iako prioritet između prekidnih rutina i ostatka programa postoji, ono što ne postoji jeste prioritet unutar ostatka programa.

22.1.3 *Function-queue-scheduling* arhitektura

Naredna arhitektura koja će biti razmatrana jeste arhitektura sa planiranjem baziranom na redovima. Njen prototip je prikazan na listingu 22.3.

```
QUEUE function_ptr_queue;

ISR(/*uredjaj_1 vektor prekida*/)
{
    /*obrada visokog prioriteta*/
    /*postavi funkciju func1() u red*/
}

ISR(/*uredjaj_2 vektor prekida*/)
```

```
{
    /*obrada visokog prioriteta*/
    /*postavi funkciju func2() u red*/
}

...

ISR(/*uredjaj_N vektor prekida*/)
{
    /*obrada visokog prioriteta*/
    /*postavi funkciju funcN() u red*/
}

void main()
{
    while(1)
    {
        while(/*red je prazan*/);

        /*poziv prve funkcije u redu*/
    }

    return 0;
}

void func1()
{
    /*dalja obrada*/
}

void func2()
{
    /*dalja obrada*/
}

...

void funcN()
{
    /*dalja obrada*/
}
```

Listing 22.3: Prototip koda za arhitekturu sa planiranjem baziranu na prekidima

U okviru ove arhitekture, prekidne rutine postavljaju pokazivače na funkcije u red pokazivača (eng. *queue*, *First-In-First-Out* (FIFO) struktura podataka). Jedini zadatak glavnog programa jeste čitanje sadržaja tog reda (odnosno, pokazivača na funkcije), i pozivanje odgovarajućih funkcija.

Osnovna prednost ovakvog tipa arhitekture jeste ta, što ne postoji pravilo za re-

dosled smeštanja pokazivača na funkcije u dati red. U zavisnosti od situacije, moguće je izabrati određeni raspored (eng. *scheduling*) smeštanja tih pokazivača i na taj način uspostaviti odgovarajući redosled prioriteta.

Najduže čekanje na obradu visokog prioriteta je određeno dužinom najduže funkcije. Ovo je u opštem slučaju bolje od *round-robin* arhitekture sa prekidima. Ipak, u nekim situacijama ni ovo nije dovoljno. Sa druge strane, kompromis koji je morao biti napravljen, pored povećane kompleksnosti, jeste manja brzina odziva kod obrade sa nižim prioritetom. Drugim rečima, mogu se dogoditi slučajevi u kojima se funkcije sa nižim prioritetom ne izvrše nikada, ukoliko je učestanost izvršavanja funkcija sa višim prioritetom takva da zauzima celokupno procesorsko vreme.

22.1.4 Arhitektura bazirana na operativnom sistemu koji radi u realnom vremenu

Poslednja arhitektura koja će biti razmatrana jeste arhitektura bazirana na operativnom sistemu za rad u realnom vremenu. Prototip koda arhitekture bazirane na operativnom sistemu za rad u realnom vremenu je prikazan na listingu 22.4.

```
ISR(/*uredjaj_1 vektor prekida*/)
{
    /*obrada visokog prioriteta*/
    /*postavi signal 1*/
}

ISR(/*uredjaj_2 vektor prekida*/)
{
    /*obrada visokog prioriteta*/
    /*postavi signal 2*/
}

...

ISR(/*uredjaj_N vektor prekida*/)
{
    /*obrada visokog prioriteta*/
    /*postavi signal N*/
}

void Task1()
{
    while(1)
    {
        /*sacekaj na signal 1*/
        /*dalja obrada*/
    }
```

```
}

void Task2()
{
    while(1)
    {
        /*sacekaj na signal 2*/
        /*dalja obrada*/
    }
}

...

void TaskN()
{
    while(1)
    {
        /*sacekaj na signal N*/
        /*dalja obrada*/
    }
}
```

Listing 22.4: Prototip koda arhitekture bazirane na RTOS

Kod ove arhitekture, prekidne rutine preuzimaju obradu visokog prioriteta, nakon čega signaliziraju ukoliko postoji dalja obrada. Iako na prvi pogled deluje isto kao i prethodna arhitektura, postoje izvesne razlike:

- signalizacija između prekidnih rutina i ostatka programa je rešena od strane samog operativnog sistema (deo koda koji ovo obavlja nije prikazan na listingu 22.4);
- ne postoji glavna petlja koja određuje koji će biti naredni korak; takođe i ovo je rešeno od strane samog operativnog sistema;
- operativni sistem može da prekine izvršavanje neke procedure (zadatka, eng. *task*) kako bi započeo izvršavanje neke druge.

Prve dve navedene stavke predstavlju više programersku rutinu. Sa druge strane, treća stavka je ključna. Sistemi koji koriste ovu arhitekturu mogu da kontrolišu kako izvršavanje pojedinih zadataka tako i izvršavanje *prekidnih rutina*. Ovo znači da, ukoliko se trenutno izvršava neki zadatak (na primer *Task2()*) i prekidna rutina signalizira da je potrebno izvršiti zadatak višeg prioriteta (na primer *Task1()*), operativni sistem će zaustaviti izvršavanje prvog zadatka (*Task2()*) i preći na izvršavanje drugog (*Task1()*). Na taj način, kašnjenje na izvršavanje zadataka višeg prioriteta je svedeno na nulu. Ovo je zapravo i definicija *preemptive* operativnog sistema za rad u realnom vremenu. Nešto više o ovome je dato u nastavku.

Glavni nedostatak ovakvog pristupa jeste taj, što sam operativni sistem troši određeno procesorsko vreme.

22.2 Uvod u operativne sisteme koji rade u realnom vremenu

Iako sličnog imena, većina operativnih sistema za rad u realnom vremenu (eng. *Real Time Operating Systems* – RTOS) se u velikoj meri razlikuje od standardnih operativnih sistema opšte namene kao što su *Windows* ili *Linux*.

Kao prvo, kod operativnih sistema koji rade u realnom vremenu, sistem i konkretna aplikacija su u znatno užoj sprezi, nego što je to situacija kod standardnih operativnih sistema. Pored toga, standardni operativni sistemi pružaju značajno veću zaštitu od nepredviđenih situacija i na taj način izbegavaju rušenje čitavog sistema. Kod embeded aplikacija, u većini slučajeva, ukoliko dođe do neke nepredviđene situacije, pored aplikacije, "srušiće" se i operativni sistem. Na kraju, operativni sistemi za rad u realnom vremenu su značajno manji, kompaktniji, i uglavnom poseduju samo neophodne servise za datu aplikaciju. Takođe, često su i konfigurabilni, tako da dozvoljavaju korisniku da izostavi one funkcije koje mu nisu potrebne.

U nastavku su objašnjene osnovne celine koje sačinjavaju jedan operativni sistem za rad u realnom vremenu.

22.2.1 Zadaci (Taskovi)

Zadaci ili taskovi (eng. *Task*) predstavljaju osnovnu gradivnu jedinicu svakog operativnog sistema za rad u realnom vremenu. Posmatrano sa aspekta implementacije, zadaci su funkcije (procedure) i može ih biti definisano proizvoljno mnogo. Svaki zadatak se može naći u jednom od naredna tri stanja:

- **Aktivno stanje** (eng. *Running*) – u okviru ovog stanja, zadatak se izvršava, odnosno njegov sadržaj; budući da govorimo o jednom procesoru, u aktivnom stanju, može se naći isključivo jedan zadatak u jednom trenutku;
- **Spremno stanje** (eng. *Ready*) – u okviru ovog stanja, zadatak je spreman da započne svoje izvršavanje, ali budući da postoji drugi zadatak koji je u aktivnom stanju, mora da sačeka dok procesor ne postane dostupan; u spremnom stanju, može se naći proizvoljno mnogo zadataka;
- **Blokirano stanje** (eng. *Blocked*) – u okviru ovog stanja, zadatak se ne izvršava, čak ni kad procesor postane dostupan; u većini situacija, zadaci dospevaju u ovo stanju jer čekaju na neki spoljašnji događaj (na primer pritisak tastera); u blokiranom stanju, takođe se može naći proizvoljan broj zadataka.

Pored ovih stanja, često se, u zavisnosti od posmatranog operativnog sistema, mogu naći i još neka dodatna stanja, kao što su: stanja čekanja (eng. *Waiting*), suspendovano stanje (eng. *Suspended*) itd. koja su većinom potkategorije tri gorenavedena stanja.

Svaki zadatak poseduje svoj privatni *kontekst* (eng. *context*) što podrazumeva stek za lokalne promenljive i zaseban programski brojač.

Sa druge strane, svi ostali podaci se smatraju globalnim i mogu da se dele između ostalih zadataka. Gledano sa implementacione strane, to mogu biti neke globalne promenljive, nizovi ili, ređe, neke složenije strukture podataka. Međutim, ovakav pristup je podložan problemu *deljenih resursa*. Više o ovome, dato je u nastavku.

Problem deljenih resursa

Problem deljenih resursa se javlja u situaciji kada više zadataka (ili prekidnih rutina) pristupaju i modifikuju iste podatke. Ovo je ilustrovano primerom datim na listingu 22.5.

```
int16_t temperature;
int16_t pressure;

void TaskPrint() /*veci prioritet*/
{
    /*cekaj dok korisnik ne pritisne taster*/
    Print(temperature, pressure);
}

void TaskCollect()
{
    temperature = readTemp();
    /*ovaj red predstavlja los trenutak za prekid izvrsavanja
        zadatka*/
    pressure = readPressure();
}
```

Listing 22.5: Ilustracija problema deljenih resursa

Zadatak *TaskCollect()* vrši očitavanje temperature i pritiska sa odgovarajućeg senzora, a zadatak *TaskPrint()* vrši ispis ovih vrednosti. Budući da operativni sistem u svakom trenutku može promeniti trenutni zadatak koji se izvršava, to može biti i trenutak između dva očitavanja (naznačeno na listingu 22.5). U toj situaciji, ukoliko je i taster pritisnut, zadatak *TaskPrint()* će ispisati netačne podatke. Kako bi se ovaj problem prevazišao, koriste se različiti servisi u okviru operativnog sistema. Više o ovim servisima biće objašnjeno u nastavku.

22.2.2 Planer

Planer (eng. *Scheduler*) je važan deo operativnog sistema za rad u realnom vremenu koji vodi računa o stanjima u kojima se zadaci nalaze i odlučuje koji zadatak će sledeći biti u aktivnom stanju. Za razliku od kompleksnih planera na standardnim operativnim sistemima kao što su *Windows* ili *Linux*, planeri kod operativnih

sistema koji rade u realnom vremenu su uglavnom vrlo jednostavni i bazirani na prioritetima. Planer posmatra sve zadatke koji se ne nalaze u blokiranom stanju i na osnovu njih bira zadatak sa najvećim prioritetom i prebacuje ga u aktivno stanje. Ostatak zadataka zadržava u spremnom stanju. Pored toga, važe i sledeća pravila:

- planer ne može da prevede zadatak u blokirano stanje, već zadatak to mora učiniti samostalno;
- sve dok je zadatak blokiran, ne može da zauzme procesor; zadatak ostaje u blokiranom stanju, sve dok ga neki spoljašnji događaj (prekidna rutina ili drugi zadatak) iz tog stanja ne izvede;
- određivanje prelaska iz spremnog stanja u aktivno stanje (i obrnuto) je zadatak planera.

Na osnovu ovog, mogu se postaviti i sledeća tri pitanja:

1. Šta se dešava ukoliko se svi zadaci nalaze u blokiranom stanju?
2. Šta se dešava ukoliko postoje dva zadatka sa istim prioritetom?
3. Šta će se dogoditi ukoliko se jedan zadatak izvršava, a drugi, većeg prioriteta pređe u spremno stanje?

Ukoliko se svi zadaci nađu u blokiranom stanju, planer će najčešće izvršavati neku petlju u okviru koje će čekati na spoljašnje događaje, odnosno na trenutak u kom će bar jedan od zadataka preći u spremno stanje. Naravno, moguće je da se takav događaj nikada ne dogodi, ali, u toj situaciji, smatra se da je to greška onog ko je projektovao dati sistem i zanemario takav slučaj.

Odgovor na drugo pitanje jeste taj, da to zavisi od toga koji operativni sistem se koristi. Neki operativni sistemi jednostavno zabranjuju tu situaciju, dok će drugi izvršiti zadatke jedan za drugim, u određenom redosledu (koji ponovo zavisi od konkretnog operativnog sistema koji se koristi).

Konačno, na osnovu trećeg pitanja je zapravo moguće izvršiti podelu operativnih sistema za rad u realnom vremenu na:

- kooperativne (eng. *cooperative*) sisteme za rad u realnom vremenu
- *preemptive* sisteme za rad u realnom vremenu

Kod kooperativnih sistema neće doći do zaustavljanja tekućeg zadatka, bez obzira da li je novi zadatak višeg ili nižeg prioriteta, već će zadatak biti izvršen u celosti, a tek nakon toga će se preći na izvršavanje narednog.

U slučaju *preemptive* operativnih sistema, situacija je obrnuta. Ukoliko se prilikom izvršavanja tekućeg zadataka pojavi zadatak višeg prioriteta u spremnom stanju, tekući zadatak će odmah biti zaustavljen i preći će se na izvršavanje narednog zadatka.

22.2.3 Komunikacija između zadataka

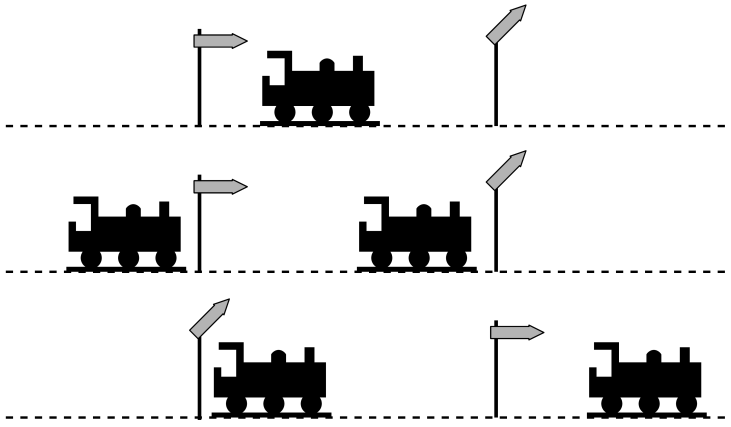
Kao što se moglo videti ranije, isključivo korišćenje globalnih promenljivih za komunikaciju između zadataka u većini situacija nije dobro rešenje. Kako bi se prevazišao problem deljenih resursa u okviru sistema za rad u realnom vremenu, koriste se različiti servisi. U okviru ovih vežbi, razmatraće se sledeći:

- Semafor (eng. *Semaphore*)
- Muteks (eng. *Mutex*)
- Red (eng. *Queue*)
- Događaj (eng. *Event*)

U nastavku je ukratko objašnjen svaki od navedenih servisa.

Semafor

Dobra analogija sa semaforima koji se koriste u sistemima za rad u realnom vremenu jesu semafori koji su se koristili za signalizaciju na železnicama. Ovo je ilustrovano na slici 22.1.



Slika 22.1: Primer upotrebe semafora u železničkom saobraćaju

Kao što se može primetiti na slici, u trenutku kada se prvi voz nađe na (kritičnom) delu pruge nakon prvog semafora, semafor se "spušta", onemogućavajući time drugom vozu da uđe u istu sekciju. Kada prvi voz napusti datu sekciju pruge, prvi semafor se "podiže" i propušta drugi voz. Na taj način je izbegnuta nezgoda.

Vrlo sličan koncept pomenutom, se koristi i kod semafora unutar operativnih sistema za rad u realnom vremenu. Naime, osnovna vrsta semafora – *binarni* semafori

se baziraju na pomenutom principu. Osnovni cilj je, analogno primeru sa vozovima, izvršavanje kritične sekcije od isključivo jednog zadatka. Implementacija samog semafora se vrši na sledeći način. Za svaki binarni semafor se pišu dve osnovne funkcije:

- funkcija za zauzimanje ("spuštanje") semafora – *TakeSemaphore()*
- funkcija za oslobađanje ("podizanje") semafora – *ReleaseSemaphore()*

Ukoliko jedan zadatak pozove funkciju za zauzimanje semafora, a pri tome ne pozove funkciju za njegovo oslobađanje, svaki drugi zadatak, koji pokuša da zauzme isti semafor će ostati blokiran sve dok dati semafor ne bude oslobođen, pri čemu taj semafor ne mora, striktno, biti oslobođen od strane istog zadatka koji ga je zauzeo. Prema tome, samo jedan zadatak ima pristup semaforu (kritičnoj sekciji) u jednom trenutku.

Čest slučaj je i da se, pored ove dve funkcije, implementira i funkcija za inicijalizaciju semafora – *InitSemaphore()*, koja se poziva jednom, na početku izvršavanja programa.

Na sledećem kodnom listingu je ilustrovana upotreba binarnog semafora kroz primer sa dva zadatka. Zadatak *TaskCollect()* vrši očitavanje podataka sa senzora (temperatura i pritisak), dok zadatak *TaskPrint()* čeka da korisnik pritisne taster, nakon čega ispisuje trenutnu vrednost temperature i pritiska. Pri tome, unapred je definisano da je zadatak *TaskPrint()* većeg prioriteta.

```
int16_t temperature;
int16_t pressure;

void TaskPrint() /*veci prioritet*/
{
    /*cekaj dok korisnik ne pritisne taster*/
    TakeSemaphore(sema);
    Print(temperature, pressure);
    ReleaseSemaphore(sema);
}

void TaskCollect()
{
    TakeSemaphore(sema);
    temperature = readTemp();
    pressure = readPressure();
    ReleaseSemaphore(sema);
}
```

Listing 22.6: Primer rada sa binarnim semaforom

Kao što se može uočiti sa listinga 22.6, prilikom rada sa globalnom promenljivom (deljeni resurs) vrši se zauzimanje semafora. Na taj način, sprečena je pojava

problema deljenih resursa, budući da samo jedan zadatak može pristupiti datim promenljivima u jednom trenutku.

U situaciji kada korisnik pritisne taster, dok drugi zadatak izvršava kritičnu sekciju (očitavanje temperature i pritiska), smenjuje se sledeći niz događaja:

1. operativni sistem će prebaciti izvršavanje na *TaskPrint()* zadatak, dok će *TaskCollect()* zadatak preći u spremno stanje;
2. *TaskPrint()* zadatak će pokušati da zauzme semafor, ali budući da je on već zauzet od strane *TaskCollect()* zadatka, ostaće blokiran;
3. operativni sistem će nakon toga pristupiti pokretanju zadataka u spremnom stanju; kako je *TaskPrint()* zadatak, koji je većeg prioriteta, u blokiranom stanju, *TaskCollect()* zadatak će se izvršiti do kraja i osloboditi semafor;
4. kada je semafor oslobođen, operativni sistem će vratiti izvršavanje na *TaskPrint()* zadatak, koji će ovog puta uspešno zauzeti oslobođeni semafor.

Prema tome, zadatak *TaskCollect()* će uvek završiti modifikaciju promenljivih `temperature` i `pressure` pre njihove upotrebe u zadatku *TaskPrint()*. Na ovaj način, izbegnuta je situacija u kojoj *TaskPrint()* zadatak koristi netačne podatke.

Pored pomenute namene, semafori se mogu koristiti i za komunikaciju (sinhronizaciju) između zadataka. Na primer, mogu se posmatrati dva zadatka *TaskCollect()* i *TaskPrint()*. Slično kao i u prethodnom primeru, prvi zadatak vrši očitavanje vrednosti sa senzora, dok drugi zadatak vrši njihov ispis. Pri tome, u ovom slučaju, zadatak *TaskPrint()* zauzima semafor i na taj način čeka sve dok odgovarajuće vrednosti ne budu spremne za ispis, odnosno sve dok se ne izvrši njihovo očitavanje u okviru zadatka *TaskCollect()*. Sa druge strane, zadatak *TaskCollect()*, nakon što izvrši očitavanje vrednosti, oslobađa semafor. Takođe, podrazumeva se da je semafor inicijalno zauzet, kako bi zadatak koji pokuša da ga zauzme ostao blokiran. Ovo je ilustrovano na listingu 22.7.

```
int16_t temperature;
int16_t pressure;

/*semafor sema inicijalno zauzet*/

void TaskPrint()
{
    TakeSemaphore(sema);
    Print(temperature, pressure);
}

void TaskCollect()
{
    temperature = readTemp();
    pressure = readPressure();
```

```
    ReleaseSemaphore(sema);
}
```

Listing 22.7: Primer sinhronizacije upotrebom binarnog semafora

U okviru sistema za rad u realnom vremenu, dozvoljen je proizvoljan broj semafora, pri čemu su ti semafori međusobno nezavisni. Ovaj pristup se najčešće koristi u situacijama gde postoji više deljenih resursa koji međusobno ne zavise jedni od drugih. U tom slučaju, različiti semafori bi se koristili za manipulaciju različitim deljenim promenljivima. Realizacija takvog sistema pomoću samo jednog semafora najčešće rezultuje njegovim usporavanjem, budući da bi postojali određeni zadaci koji čekaju na oslobađanje semafora od strane drugog zadatka, pri čemu taj zadatak koristi deljene resurse koji ostatku zadataka nisu od interesa.

Važna stvar prilikom rada sa više semafora jeste ta da, sistem za rad u realnom vremenu ne poseduje informaciju o tome koji semafor se koristi za koje deljene resurse. O ovim informacija računa mora voditi korisnik operativnog sistema, odnosno programer.

Iako deluju poprilično jednostavno, u radu sa semaforima često se javljaju najrazličitije greške. Neke osnovne greške prilikom rada sa semaforima su:

- zaboravljanje zauzimanja semafora – uzrokuje pojavu netačnih vrednosti na izlazu;
- zaboravljanje oslobađanja semafora – uzrokuje pojavu beskonačnog čekanja svih zadataka koji zavise od datog semafora;
- zauzimanje pogrešnog semafora;
- predugo zadržavanje semafora – uzrokuje usporavanje celokupnog sistema, što je loše, budući da sistem treba da radi u realnom vremenu;
- pojava inverzije prioriteta (eng. *priority inversion*) – nastaje u slučaju kada operativni sistem prebacuje izvršavanje sa zadatka nižeg prioriteta (zadatak C) koji je prethodno zauzeo semafor, na zadatak srednjeg prioriteta (zadatak B). Ukoliko zadatak višeg prioriteta (zadatak A) želi da zauzme dati semafor, mora da sačeka sve dok se izvršavanje ne vrati sa zadatka B na zadatak C; do tog trenutka, semafor će biti zauzet od strane zadatka nižeg prioriteta; prema tome, dešava se situacija u kojoj zadatak C naizgled ima veći prioritet od zadatka A, budući da zadatak A ostaje blokiran sve dok zadatak C ne oslobodi semafor.
- pojava blokade (eng. *dead-lock*, nekada se naziva i pojava mrtve petlje) – koja će biti objašnjena kroz sledeći primer.

```
void Task1()
{
    while(1)
    {
```

```
        ...
        TakeSemaphore(sema1);
        ReleaseSemaphore(sema2);
        ...
    }
}

void Task2()
{
    while(1)
    {
        ...
        TakeSemaphore(sema2);
        ReleaseSemaphore(sema1);
        ...
    }
}
```

Listing 22.8: Problem blokade

Kako se može videti sa listinga 22.8, oba zadatka prepuštaju kontrolu operativnom sistemu, čekajući na oslobađanje odgovarajućih semafora; naime, *Task1()* čeka na semafor `sema1`, koji se aktivira od strane *Task2()*, dok *Task2()* čeka na semafor `sema2`, koji se aktivira od strane *Task1()*; kako su oba zadatka u stanju čekanja, sistem je blokiran, budući da nijedan od dva semafora nikada neće biti oslobođen.

Pored osnovnog, binarnog, postoje i druge varijante semafora koje se često koriste u praksi. Neke od njih su:

- **Brojački semafor** (eng. *counting semaphore*) – predstavlja tip semafora koji može biti više puta zauzet. Implementiraju se kao brojači, koji se na početku inicijalizuju sa nekom početnom vrednošću; prilikom zauzimanja semafora, trenutna vrednost brojača se smanjuje za jedan, dok se prilikom oslobađanja semafora ova vrednost povećava za jedan; ukoliko je vrednost semafora jednaka nuli, sledeći zadatak koji pokuša da ga zauzme će ostati blokiran, sve dok neki od zadataka ne oslobodi semafor (ne uveća brojač); pri tome, podrazumeva se da se operacije uvećavanja i smanjivanja vrednosti semafora izvršavaju *atomički*, odnosno da se zabranjuje situacija u kojoj dva zadatka modfikuju istu vrednost brojača.
- **Muteks** (eng. *MUTual EXclusion - mutex*) – za razliku od semafora koji mogu biti zauzimani i oslobađani od strane različitih zadataka, kod muteksa je situacija nešto drugačija; muteks može biti oslobođen *isključivo* od zadatka koji ga je prethodno zauzeo; ono što je onemogućeno ovim putem jeste upotreba muteksa za komunikaciju između više zadataka (što je bilo moguće kod klasičnih semafora); sa druge strane, muteksi rešavaju problem

inverzije prioriteta koji je bio prisutan kod binarnih semafora, tako što implementiraju protokol o nasleđivanju prioriteta (eng. *priority inheritance*); prema ovom protokolu, ukoliko zadatak višeg prioriteta pokuša da zauzme već zauzet muteks od strane zadatka nižeg prioriteta, vrši se trenutno nasleđivanje prioriteta, u okviru kog se prioriteti zadataka izjednačavaju, sve dok zadatak, koji je prethodno zauzeo muteks, ne oslobodi isti; ovakav mehanizam je dizajniran sa ciljem zadržavanja zadatka sa višim prioritetom u blokiranom stanju tokom što je moguće kraćeg vremenskog intervala.

Na kraju, ukoliko više zadataka čeka na isti semafor, od implementacije operativnog sistema zavisi koji od njih će prvi dobiti pristup oslobođenom semaforu. Neki sistemi će prednost dati zadatku sa najvišim prioritetom, dok će drugi dati prednost onom zadatku koji je najduže čekao na dati semafor. Neki operativni sistemi će, ipak, ovu odluku ostaviti korisniku.

Red

Drugi važan servis unutar sistema za rad u realnom vremenu jesu redovi. Red (eng. *Queue*) predstavlja servis za komunikaciju između zadataka ili između zadatka i prekidne rutine. Red je najčešće implementiran kao FIFO (eng. *First-In-First-Out*) struktura podataka, kod koje se novi podaci postavljaju na početak reda, dok se čitanje vrši sa kraja reda. Slično kao i kod semafora, u okviru sistema za rad u realnom vremenu dozvoljen je proizvoljan broj redova.

Tipična upotreba reda, može se ilustrovati na sledeći način. Na primer, neka zadaci *TaskCollectLoc1()* i *TaskCollectLoc2()* vrše očitavanje vrednosti temperature sa senzora na dve različite lokacije (u opštem slučaju, broj zadatak koji vrše očitavanje može biti proizvoljan). Sa druge strane, treći zadatak, *TaskNetworkPush()*, ima ulogu u prosleđivanju sakupljenih informacija ka serveru. U ovoj situaciji, prva dva zadatka postavljaju očitane podatke na kraj reda, dok treći zadatak preuzima date podatke sa vrha reda. Na ovaj način, umesto da svaki zadatak vrši zasebno slanje podataka ka serveru i time biva značajno usporen, upotrebom reda se situacija značajno pojednostavljuje.

Prilikom realizacije reda, u većini slučajeva se implementiraju sledeće funkcije:

- funkcija za kreiranje reda – *CreateQueue()*
- funkcija za dodavanje elementa na kraj reda – *Enqueue()*
- funkcija za uklanjanje elementa sa početka reda – *Dequeue()*

Redovi se najčešće implementiraju tako da su podaci, koji se smeštaju u red, najčešće pokazivači (i to tipa `void`, odnosno pokazivači na *bilo kakav* tip podataka, pa čak i funkciju) na poziciju u memoriji na kojoj se nalaze stvarni podaci. Osnovna ideja koja leži iza ovoga jeste ta, da zadatak sada može da pošalje proizvoljan broj podataka ka nekom drugom zadatku. Ovo je moguće učiniti tako što će se

odgovarajući podaci postaviti u neki memorijski bafer, nakon čega će se proslediti pokazivač na dati bafer u red.

Ukoliko zadatak pokuša da pročita podatke iz praznog reda, najčešće će ostati blokiran sve dok neki drugi zadatak ne postavi neki podatak u red i na taj način red više ne bude prazan. Kod nekih realizacija, pored ove funkcije, biće implementirana i funkcija za čitanje kod koje, ukoliko je red prazan, zadatak neće ostati blokiran, a kao povratna vrednost biće vraćen kod greške. Takođe, nekad je moguće specificirati, kao dodatni parametar, i *timeout*, odnosno vremenski interval, nakon kog će, čak i ako red ostane prazan, zadatak biti odblokiran i nastaviti izvršavanje.

U slučaju da je red popunjen, a zadatak pokuša da upiše neki podatak, u najčešćem broju implementacija, funkcija će vratiti kod greške, a dati podatak će biti odbačen. U manjem broju slučajeva, funkcija za upis će blokirati zadatak koji pokuša da upiše podatak u popunjen red.

Pitanje 22.2.1. Osmisliti način implementacije mehanizama kod koga će zadatak koji pokuša da upiše neki podatak u popunjen red ostati blokiran, sve dok drugi zadatak ne pročita nešto iz reda.

Pitanje 22.2.2. Na koji način se redovi mogu koristiti kao binarni semafori za zaštitu podataka? Na koji način se koriste kao brojački semafori?

Pored redova, u praksi se često mogu susresti i sledeće strukture:

- **Sanduče** (eng. *Mailbox*) – predstavlja restrikciju koncepta reda; naime, sanduče predstavlja red u kome je moguće skladištiti samo jedan element; poput reda, poseduje funkcije za kreiranje sandučeta, dodavanje i uklanjanje elemenata; neki operativni sistemi za rad u realnom vremenu dozvoljavaju veći broj elemenata u sandučetu (što ih u toj situaciji ne čini značajno drugačijim od običnog reda).
- **Cev** (eng. *Pipe*) – slično kao i sanduče, i cev je vrlo slična struktura redu; u cev, za razliku od reda i sandučeta, moguće je upisivati podatke različite dužine; pored toga, podaci u cevi se uglavnom posmatraju samo kao bajtovi (eng. *byte-oriented*); ovo znači da, ukoliko prvi zadatak upiše 13 bajtova u cev, drugi zadatak 10 bajtova, a treći zadatak pročita 16 bajtova, 13 od tih 16 bajtova će biti bajtovi koje je upisao prvi zadatak, dok će 3 bajta biti bajtovi koje je upisao drugi zadatak.

Događaji

Poslednji servis korišćen od strane operativnog sistema za rad u realnom vremenu, koji će biti razmatran, jeste događaj (eng. *event*). U osnovi, događaj predstavlja

običnu Bulovu promenljivu na čije postavljanje ili resetovanje mogu čekati drugi zadaci. Događaji se često koriste u situacijama kada više zadataka čeka na jedan događaj. Nakon što se on pojavi, ovi zadaci će biti odblokirani i izvršiće se u unapred definisanom redu.

Operativni sistemi za rad u realnom vremenu često formiraju skupove događaja, gde jedan zadatak može da čeka na pojavu proizvoljnog podskupa iz datog skupa događaja.

Način resetovanja događaja, nakon njegove pojave, u velikoj meri zavisi od samog operativnog sistema. Neki operativni sistemi ovo vrše automatski, dok drugi zahtevaju od zadataka da to učine.

Iako ne deluje tako, koncept događaja je nešto kompleksniji od semafora. Prednost korišćenja događaja u odnosu na semafore se ogleda u tome da jedan zadatak može čekati na više različitih događaja istovremeno. Sa druge strane, jedan zadatak može u jednom trenutku da čeka na isključivo jedan semafor.

22.3 RTOS Implementacija na primeru ArdOS

U okviru ovog poglavlja će biti prikazan rad sa jednom od postojećih implementacija operativnih sistema za rad u realnom vremenu koja se naziva *ArdOS* (eng. *The Arduino Operating System*). Ova implementacija operativnog sistema za rad u realnom vremenu sa više zadataka omogućava upotrebu većine navedenih servisa na velikom broju Arduino platformi, uključujući i Arduino UNO.

Odlike ovog operativnog sistema su:

- potpuna kompatibilnost sa standardnim bibliotekama za Arduino platforme i Arduino integrisanim razvojnim okruženjem *Arduino IDE*;
- malo, kompaktno jezgro (eng. *kernel*);
- prioritetni planer za rad sa više zadataka u okviru *hard real-time* aplikacija (aplikacija u kojoj postoji tačno određen vremenski interval u okviru kog je potrebno reagovati na neki događaj, a ukoliko se reagovanje izvrši kasnije, može doći do značajne štete);
- maksimalan broj zadataka je 8;
- implementaciju funkcije za "uspavljivanje" (eng. *sleep*) zadatka koja omogućava pauziranje izvršavanja tekućeg zadatka u dužini željenog vremenskog intervala pri čemu dozvoljava izvršavanja drugih zadataka u tom periodu;
- binarni i brojački semafor;
- muteks i uslovne promenljive;
- FIFO i prioritetni redovi;

- konfigurabilnost koja omogućava izopštavanje nepotrebnih delova operativnog sistema tokom procesa kompajliranja u cilju smanjenja upotrebe memorije.

Originalna verzija operativnog sistema za rad u realnom vremenu ArdOS se može preuzeti sa sledećeg linka. Ova verzija je namenjena upotrebi u okviru razvojnog okruženja *Arduino IDE*.

`https://bitbucket.org/ctank/ardos-ide/wiki/Home`

Kako je ova verzija bazirana na upotrebi biblioteka koje su dostupne u okviru razvojnog okružena *Arduino IDE*, ali ne i u okviru razvojnog okruženja *Eclipse* ili direktnom upotrebom *AVR–GCC* razvojnih alata, dostupna je modifikovana verzija na sledećem linku.

`https://github.com/rszes/biblioteke.git`

22.3.1 Struktura ArdOS

Operativni sistem *ArdOS* je implementiran u okviru nekoliko datoteka koje čine odgovarajuće funkcionalne celine. Datoteke koje sačinjavaju ovaj operativni sistem su:

- *kernel.c* i *kernel.h* – datoteke koje implementiraju jezgro operativnog sistema; sačinjavaju ih funkcije za inicijalizaciju i pokretanje operativnog sistema, inicijalizaciju sistemskog vremena, prekidna rutina za sistemsko vreme, funkcije za kreiranje zadataka, funkcije za promenu zadataka, asemblerski kod za sačuvavanje i učitavanje konteksta itd.;
- *task.c* – datoteka koja sadrži dodatne funkcije za upravljanje redom koji skladišti zadatke;
- *profiler.c* i *profiler.h* – datoteke koje implementiraju funkcije za merenje vremena izvršavanja zadataka i njihovo skladištenje u EEPROM memoriji;
- *sema.c* i *sema.h* – biblioteka koja sadrži funkcije za kreiranje i upravljanje semaforima;
- *mutex.c* i *mutex.h* – biblioteka koja implementira funkcije za kreiranje i upravljanje muteksima i uslovnim promenljivama;
- *queue.c* i *queue.h* – biblioteka koja realizuje funkcije za kreiranje i upravljanje FIFO i prioritetnim redovima.

Navedene datoteke sadrže veliki broj funkcija, promenljivih i konstanti koje su neophodne za ispravno funkcionisanje operativnog sistema. Međutim, za njegovu

ispravnu upotrebu, nije neophodno poznavanje i razumevanje svake od njih. Zbog ovog razloga će u narednim poglavljima biti opisane samo funkcije koje su nama, kao korisnicima, neophodne za upotrebu operativnog sistema.

Kako bi se omogućio rad operativnog sistema *ArdOS*, potrebno je izvršiti sledeće korake:

1. *Uključivanje neophodnih biblioteka* – Standardna biblioteka za rad sa operativnim sistemom ArdOS je *kernel.h*. Ukoliko je u okviru projekta potrebno koristiti posebne servise, neophodno je uključiti biblioteke koje ih implementiraju.
2. *Inicijalizacija operativnog sistema* – Pod inicijalizacijom operativnog sistema se podrazumeva izvršavanje funkcije *OSInit* koja omogućava specificiranje broja zadataka koje će operativni sistem morati da podržava.
3. *Kreiranje servisa* – Ukoliko je u okviru projekta neophodno upotrebiti neki servis, potrebno ga je definisati kao globalnu promenljivu i kreirati pomoću odgovarajuće funkcije.
4. *Kreiranje zadataka* – Potrebno je definisati onoliko funkcija, koje implementiraju funkcionalnosti koje se očekuju od zadataka, koliko je specificirano brojem zadataka prilikom inicijalizacije operativnog sistema. Takođe, funkcije je potrebno povezati sa odgovarajućim zadacima.
5. *Pokretanje operativnog sistema* – Nakon izvršenja prethodnih koraka, operativni sistem se može pokrenuti upotrebom funkcije *OSRun()*.

22.3.2 Funkcije za pokretanje operativnog sistema

Dve osnovne funkcije koje su na raspolaganju korisniku za pokretanje operativnog sistema su *OSInit()* i *OSRun()*. Obe funkcije su definisane unutar *kernel.h* zaglavlja.

```
void OSInit(uint8_t numTasks);
```

Opis: Funkcija koja inicijalizuje operativni sistem i specificira broj zadataka koji trebaju da se izvršavaju.

Parametri:

- `numTasks` – broj zadataka koji je potrebno da se izvršava istovremeno; maksimalna vrednost ovog parametra je 8, što znači da je moguće istovremeno izvršavanje 8 različitih zadataka

Povratna vrednost:
Nema.

```
void OSRun();
```

Opis: Funkcija koja započinje rad operativnog sistema.

Parametri:
Nema.

Povratna vrednost:
Nema.

22.3.3 Funkcije za kreiranje zadataka

Funkcije koje se upotrebljavaju pilikom kreiranja zadatak su *OSSetStackSize()* i *OSCreateTask()*. Obe funkcije su definisane unutar *kernel.h* zaglavlja.

```
void OSSetStackSize(uint8_t stackSize);
```

Opis: Funkcija koja podešava veličinu steka koji će biti formiran za svaki zadatak.

Parametri:

- `stackSize` – broj koji predstavlja veličinu steka, gde je svaki element steka tipa `uint32_t`; podrazumevana vrednost parametra iznosi 50, što znači da će stek zauzimati 200 bajtova

Povratna vrednost:
Nema.

```
uint16_t OSCreateTask(uint8_t prio, void (*fptr)(void *), void
    *param);
```

Opis: Funkcija koja se upotrebljava za kreiranje zadatka. Njena uloga je da registruje novi zadatak, podesi njegove parametre kao što su prioritet i stek. Zatim da poveže zadatak sa funkcijom koju treba da izvršava i definiše njene argumente.

Parametri:

- `prio` – broj koji predstavlja prioritet zadatka zadatka koji se kreira; vrednost može biti u opsegu od 0 do `numTasks`-1, gde je `numTasks` broj zadataka naveden prilikom inicijalizacije operativnog sistema; manja vrednost parametra `prio` označava veći prioritet zadatka, tako da vrednost 0 označava najveći prioritet, a `numTasks`-1 najmanji[1]
- `fptr` – pokazivač na funkciju koja se izvršava u okviru kreiranog zadatka

[1] U okviru operativnog sistema *ArdOS*, svaki zadatak mora imati različit prioritet!

- `param` – pokazivač na skup argumenata koje je potrebno proslediti funkciji prilikom pokretanja zadatka

Povratna vrednost:
Kod greške koji može imati vrednost:

- `OS_NO_ERR` – ukoliko je funkcija izvršena ispravno
- `OS_ERR_DUP_PRIO` – ukoliko je prethodno definisan zadatak sa navedenim prioritetom
- `OS_ERR_BAD_PRIO` – ukoliko je navedeni prioritet veći od broja zadataka

22.3.4 Funkcije za promenu izvršavanja zadataka

Prilikom rada operativnog sistema, zadaci se izvršavaju u tzv. *pseudo-paralelnom* režimu, gde se paralelnost postiže naizmeničnim izvršavanjem zadataka. Ovakav način rada operativnog sistema zahteva postojanje funkcija koje izvršavaju zamenu zadatka koji se izvršava. Ove funkcije mogu biti pozvane eksplicitno u okviru nekog zadatka ili implicitno, tj. u okviru neke druge funkcije, što i jeste najčešći slučaj. Na primer, prilikom pozivanja funkcije *OSSleep()* se tekući zadatak "uspavljuje", odnosno blokira, pri čemu se prelazi na izvršavanje nekog drugog koji je u tom trenutku spreman. U zavisnosti od načina izvršavanja zamene zadatka, definisane su četiri funkcije: *OSSwap()*, *OSPrioSwap()*, *OSSwapFromISR()* i *OSPrioSwapFromISR()*.

```
void OSSwap();
```

Opis: Funkcija koja omogućava zadatku da prepusti kontrolu mikrokontrolera. U slučaju kada postoji nekoliko zadataka spremnih za izvršavanje, prelaz se vrši na onaj sa najvišim prioritetom. Ukoliko ni jedan drugi zadatak nije spreman za izvršavanje, kontrola se vraća zadatku koji je pozvao ovu funkciju.

Parametri:
Nema.

Povratna vrednost:
Nema.

```
void OSPrioSwap();
```

Opis: Funkcija koja omogućava zadatku da prepusti kontrolu mikrokontrolera. U ovom slučaju izvršava prelaz na zadatak najvećeg prioriteta koji je spreman za izvršavanje. Takođe, prioritet zadatka na koji se prelazi *mora biti veći* od prioriteta zadatka koji je pozvao funkciju! U suprotnom, kontrola se vraća zadatku koji je pozvao ovu funkciju.

Parametri:
Nema.

Povratna vrednost:
Nema.

```
void OSSwapFromISR();
```

Opis: Funkcija koja prepušta kontrolu iz prekidne rutine na isti način kao u slučaju kod funkcije *OSSwap()*.

Parametri:
Nema.

Povratna vrednost:
Nema

```
void OSPrioSwapfromISR();
```

Opis: Funkcija koja prepušta kontrolu iz prekidne rutine na isti način kao u slučaju kod funkcije *OSPrioSwap()*.

Parametri:
Nema.

Povratna vrednost:
Nema.

22.3.5 Funkcije za rad sa vremenom

Operativni sistem ArdOS podržava dve vremenske funkcije *OSSleep()* i *OSticks()*.

```
void OSSleep(uint32_t ms);
```

Opis: Funkcija koja "uspavljuje", odnosno blokira, zadatak najkraće u vremenskom intervalu specificiranom parametrom *ms*. Kada predviđeni broj milisekundi prođe, zadatak se ponovo pokreće samo ukoliko je u tom trenutku najvećeg prioriteta. Ovo znači da će zadaci višeg prioriteta imati tačnije vreme spavanja.

Parametri:

- `ms` – broj milisekundi tokom kojeg zadatak treba da bude blokiran; broj milisekundi je neoznačenog 32-bitnog tipa

Povratna vrednost:
Nema.

```
uint32_t OSticks();
```

Opis: Funkcija koja kao povretnu vrednost vraća broj milisekundi proteklih od pokretanja operativnog sistema.

Parametri:
Nema.

Povratna vrednost:
Broj milisekundi proteklih od trenutka pokretanja operativnog sistema pozivom funkcije *OSRun()*. Broj milisekundi je neoznačenog 32-bitnog tipa.

22.3.6 Funkcije za rad sa semaforima

Kao što je već ranije rečeno, semafori su servisi kojima se upravlja upotrebom tri funkcije. U slučaju operativnog sistema *ArdOS*, njihova imena su *OSCreateSema()*, *OSTakeSema()* i *OSGiveSema()*. Kako u okviru aplikacije koja sadrži operativni sistem može postojati više semafora, potrebno je prvo kreirati objekte tipa *OSSema* kao globalnu promenljivu, nakon čega je kreirani semafor potrebno inicijalizovati.

```
void OSCreateSema(OSSema *sema, uint16_t initVal, uint8_t
    isBinary);
```

Opis: Funkcija koja inicijalizuje objekat semafora sa prosleđenim parametrima.

Parametri:

- `sema` – pokazivač na semafor koji je potrebno inicijalizovati
- `initVal` – vrednost kojom se inicijalizuje semafor; u slučaju da je semafor binarnog tipa, moguće vrednosti su 0 i 1, dok je u slučaju brojačkog semafora bilo koji nenegativan broj
- `isBinary` – parametar koji označava da li će semafor biti podešen kao binarni ili brojački; ukoliko je parametar pozitivan broj, semafor će biti podešen kao binarni, a ukoliko je 0, biće kreiran brojački semafor

Povratna vrednost:
Nema.

```
void OSTakeSema(OSSema *sema);
```

Opis: Funkcija koja pokušava da zauzme semafor. Ukoliko semafor ima vrednost različitu od 0, zadatak će uspešno zauzeti semafor dekrementovanjem njegove vrednosti, dok će, u suprotnom, zadatak biti blokiran.

Parametri:

- `sema` – pokazivač na semafor koji je potrebno zauzeti

Povratna vrednost:
Nema.

```
void OSGiveSema(OSSema *sema);
```

Opis: Funkcija koja oslobađa semafor i odblokira zadatak najvećeg prioriteta koji čeka na semafor, ukoliko postoji. U suprotnom inkrementuje vrednost semafora.

Parametri:

- `sema` – pokazivač na semafor koji je potrebno osloboditi

Povratna vrednost:
Nema.

22.3.7 Funkcije za rad sa muteksima

Po svom konceptu, funkcije za upravljanje muteksima su slične funkcijama za upravljanje semaforima. Operativni sistem podržava upotrebu više muteksa, tako da je potrebno kreirati i inicijalizovati željeni broj objekta tipa *OSMutex* kao globalne promenljive. Funkcije koje se upotrebljavaju prilikom rada sa muteksima su *OSCreateMutex()*, *OSTakeMutex()* i *OSGiveMutex()*.

```
void OSCreateMutex(OSMutex *mutex);
```

Opis: Funkcija koja inicijalizuje muteks objekat.

Parametri:

- `mutex` – pokazivač na muteks koji je potrebno inicijalizovati

Povratna vrednost:
Nema.

```
void OSTakeMutex(OSMutex *mutex);
```

Opis: Funkcija koja pokušava da zauzme muteks. Ukoliko je muteks slobodan, trenutni zadatak nastavlja sa izvršavanjem. Međutim, ako je muteks zauzet od strane nekog drugog zadatka, trenutni zadatak će biti blokiran dok se muteks ne oslobodi.

Parametri:

- `mutex` – pokazivač na muteks koji je potrebno zauzeti

Povratna vrednost:
Nema.

```
void OSGiveMutex(OSMutex *mutex);
```

Opis: Funkcija koja oslobađa muteks. Ukoliko u trenutku oslobađanja muteksa postoji nekoliko zadataka koji čekaju da se ovaj muteks oslobodi, zadatak najvišeg prioriteta će biti odblokiran i postaje spreman za izvršavanje. Ako zadatak koji oslobađa muteks ima viši prioritet u odnosu na sve zadatke koji čekaju da se muteks oslobodi, ovaj zadatak će nakon oslobađanja muteksa nastaviti sa izvršavanjem.

Parametri:

- `mutex` – pokazivač na mutex koji je potrebno osloboditi

Povratna vrednost:
Nema.

22.3.8 Funkcije za rad sa uslovnim promenljivama

Sinhronizacija unutar međusobno isključivih zona koda, koda ograničenog muteksom, je problem koji zahteva uvođenje novog servisa, koji se naziva uslovna promenljiva (eng. *conditional variable*). Ovaj servis omogućava zadacima da izvršavaju čekanje i signalizaciju. Naime, uslovna promenljiva je specijalna promenljiva koja se može iskoristiti u zadacima da kreiraju signal ili da čekaju na isti. Međutim, za razliku od semafora, gde se oslobađanje uvek izvršava, slanje signala pomoću uslovne promenljive ne mora biti uvek detektovano. U slučaju kada ni jedan zadatak ne čeka na signal, isti će biti izgubljen. Kako bi se ovo predupredilo, uslovne promenljive se pozivaju samo unutar koda koji je ograničen muteksom.

Uslovna promenljiva se definiše kao globalna promenljiva tipa *OSCond*. Ukoliko se uslovna promenljiva upotrebljava u programu, obavezno je prisustvo i muteksa koji obezbeđuje kod u kom se promenljiva koristi. Za rad sa uslovnim promenljivama, dostupne su tri funkcije: *OSCreateConditional()*, *OSWait()* i *OSSignal()*.

```
void OSCreateConditional(OSCond *cond);
```

Opis: Funkcija koja inicijalizuje objekat uslovne promenljive.

Parametri:

- `cond` – pokazivač na uslovnu promenljivu koju je potrebno inicijalizovati

Povratna vrednost:
Nema.

```
void OSWait(OSCond *cond, OSMutex *mutex);
```

Opis: Funkcija koja omogućava čekanje na pojavu signala. Funkcija oslobađa muteks i prelazi u blokirano stanje. Na uslovnu promenljivu može čekati samo jedan zadatak!

Parametri:

- `cond` – pokazivač na uslovnu promenljivu na koju je potrebno čekati
- `mutex` – pokazivač na muteks koji obezbeđuje deo koda koji sadrži ovu funkciju

Povratna vrednost:
Nema.

```
void OSSignal(OSCond *cond);
```

Opis: Funkcija koja izvršava signalizaciju uslovne promenljive. Prilikom pozivanja ove funkcije, zadatak koji čeka na signal će biti odblokiran, postaće spreman za izvršavanje i biće pokrenut kada bude imao najviši prioritet od svih spremnih zadataka. Kada zadatak koji čeka nastavi izvršavanje, ponovo će zauzeti muteks. Zbog ovoga je neophodno da svaki zadatak koji je zauzeo muteks u periodu od kako je zadatak koji čeka oslobodio muteks do sada oslobodi isti.

Parametri:

- `cond` – pokazivač na uslovnu promenljivu koja treba da se signalizira

Povratna vrednost:
Nema.

22.3.9 Funkcije za rad sa FIFO redovima

Kao što je već rečeno, za upravljanje redovima se koriste tri funkcije. To su u slučaju operativnog sistema *ArdOS* *OSCreateQueue()*, *OSEnqueue()* i *OSDequeue()*. Pre nego što se ove funkcije iskoriste, potrebno je kreirati neophodne

objekte. Sam red se instanicionira kao globalna promenljiva tipa *OSQueue*. Pored njega, neophodno je napraviti niz tipa `int16_t`, koji će predstavljati memorijski element ovog reda i koji će biti ugrađen u red prilikom njegove inicijalizacije.

```
void OSCreateQueue(int16_t *buffer, uint8_t length, OSQueue *
    queue);
```

Opis: Funkcija koja inicijalizuje objekat reda.

Parametri:

- `buffer` – pokazivač na niz koji predstavlja memorijsku osnovu reda, odnosno u kojem će biti skladišteni elementi koji se dodaju u red; elementi niza moraju biti tipa `int16_t`
- `length` – dužina reda koja ne sme biti duža od broja elemenata niza `buffer`
- `queue` – pokazivač na red koji je potrebno inicijalizovati

Povratna vrednost:
Nema.

```
void OSEnqueue(int data, OSQueue *queue);
```

Opis: Funkcija koja dodaje novi element u red ukoliko u njemu ima mesta, u suprotnom, element će biti izgubljen. Kako bi se ovakva situacija što manje puta desila, zadatak koji dodaje nove elemente bi trebao da bude nižeg prioriteta u odnosu na onaj koji uklanja elemente iz reda.

Parametri:

- `data` – vrednost koju je potrebno dodati u niz. Mora biti tipa `int16_t`
- `queue` – pokazivač na red u koji je potrebno dodati novi element

Povratna vrednost:
Nema.

```
int OSDequeue(OSQueue *queue);
```

Opis: Funkcija koja uklanja element iz reda. Kako je u pitanju FIFO red, element koji se uklanja je upravo onaj koji je prvi dodat. Ukoliko je red prazan, zadatak će biti blokiran dok se ne doda prvi element. Dobra praksa je da samo jedan zadatak ima mogućnost da pozove ovu funkciju.

Parametri:

- `queue` – pokazivač na red iz kojeg je potrebno ukloniti element

Povratna vrednost:
Element tipa `int16_t` koji je prvi dodat u red.

22.3.10 Funkcije za rad sa prioritetnim redom

Prioritetni redovi su po svom konceptu vrlo slični FIFO redu. Glavna osobina koja ih razlikuje je to što elementi niza nisu obične celobrojne vrednosti nego promenljive tipa *TPrioNode*. Pa je zbog toga neophodno definisati globalni niz koji će biti upravo ovog tipa. Ovaj niz će biti iskorišćen prilikom inizijalizacije kao memorijska osnova niza. Ovaj tip sadrži dva polja, prvo je tipa `int16_t` i sadrži vrednost elementa niza, dok drugo polje predstavlja prioritet ovog elementa i ima tip `uint8_t`. Funkcije za upravljanje prioritetnim redom su *OSCreatePrioQueue()*, *OSPrioEnqueue()* i *OSDequeue()*.

```
void OSCreatePrioQueue(TPrioNode *buffer, uint8_t length,
    OSQueue *queue);
```

Opis: Funkcija koja inicijalizuje objekat reda.

Parametri:

- `buffer` – pokazivač na niz koji predstavlja memorijsku osnovu reda, odnosno u kojem će biti skladišteni elementi koji se dodaju u red; elementi niza moraju biti tipa `TPrioNode`
- `length` – dužina reda koja ne sme biti duža od broja elemenata niza `buffer`
- `queue` – pokazivač na red koji je potrebno inicijalizovati

Povratna vrednost:
Nema.

```
void OSPrioEnqueue(int16_t data, uint8_t prio, OSQueue *queue)
    ;
```

Opis: Funkcija koja dodaje novi element u red ukoliko u njemu ima mesta. Ukoliko je red popunjen, element će biti izgubljen. Kako bi se ovakva situacija što manje puta desila, zadatak koji dodaje nove elemente bi trebalo da bude nižeg prioriteta u odnosu na onaj koji uklanja elemente iz reda. Elementi su u redu poređani po prioritetu, gde su elementi najvišeg prioriteta postavljeni na početak reda.

Parametri:

- `data` – vrednost koju je potrebno dodati u niz; mora biti tipa `int16_t`
- `prio` – vrednost koja reprezentuje prioritet elementa koji je potrebno dodati; što je vrednost veća, to je prioritet manji; mora biti tipa `uint8_t`

- `queue` – pokazivač na red u koji je potrebno dodati novi element

Povratna vrednost:
Nema.

```
int OSDequeue(OSQueue *queue);
```

Opis: Funkcija koja uklanja element iz reda. Kako je u pitanju prioritetni red, elementi će biti uklonjeni po proritetu, počevši od onih sa najvišim prioritetom. Ukoliko je red prazan, zadatak će biti blokiran dok se ne doda prvi element. Dobra praksa je da samo jedan zadatak ima mogućnost da pozove ovu funkciju.

Parametri:

- `queue` – pokazivač na red iz kojeg je potrebno ukloniti element

Povratna vrednost:
Element tipa `int16_t` koji je na početku reda.

22.4 Primeri aplikacija u ArdOS

U okviru ovog poglavlja će biti prikazani primeri aplikacija u ArdOS operativnom sistemu. Primeri su koncipirani na način da redom uvode nove servise. Za pokretanje svih primera je neophodan samo Arduino koji ima mogućnost serijske komunikacije sa računarom.

Osnovna aplikacija u ArdOS

Primer 22.4.1. Implementirati aplikaciju koja omogućava istovremeno treperenje diode frekvencijom od 1Hz i ispisivanje sistemskog vremena putem serijskog porta svakih 500ms.

Rešenje:

Kako je potrebno realizovati istovremeno izvršavanje dve funkcionalnosti, biće neophodno iskoristiti dva zadatka. Prvi zadatak će biti namenjen realizaciji treperenja diode. Funkcija koja će biti povezana sa ovim zadatkom je *task1*, a kao parametar ove funkcije je prosleđena perioda treperenja, koja u ovom slučaju iznosi 1000ms. Drugi zadatak će biti realizovan u okviru funkcije *task2*, koja će slati poruku u čijem je sadržaju sistemsko vreme svakih 500ms. Ovaj vremenski interval je, takođe, prosleđen kao parametar funkcije prilikom kreiranja zadatka. Rešenje zadatka je prikazano na listingu 22.9.

```
#include <stdio.h>
```