Osnove mikroprocesorske elektronike

Ciklični medpomnilnik

Pri programiranju vgrajenih sistemov se pogosto srečamo s problemom *obdelave podatkov*: na tak ali drugačen način *sprejemamo podatke*, ki jih je potem potrebno nekako *obdelati* in se morda nato še *ustrezno odzvati nanje*. Pogosto se ta proces obdelave podatkov razdeli v dva dela:

1) podatke *najprej hitro* sprejmemo in *shranimo*;
2) *kasneje*, ko utegnemo, sprejete in *shranjene* podatke še obdelamo.

Prvi korak moramo biti sposobni izvesti zelo hitro, da zagotovimo, da se podatki pri sprejemanju ne izgubljajo zaradi naše neodzivnosti. Drugi korak, ko pa podatke obdelamo in se nanje odzovemo, pa običajno izvedemo nekoliko kasneje, ko procesorski čas lahko namenimo obdelavi podatkov.

Vidite lahko, da je ključen del procesa obdelave podatkov ta, da *znamo sprejete podatke shraniti za krajši čas*, preden jih nato obdelamo. Za kratkotrajno shranjevanje sprejetih podatkov uporabljamo t. i. *medpomnilnike* (angl. buffer). Medpomnilniki so posebne *podatkovne strukture*, ki omogočajo *hitro shranjevanje in branje podatkov*.

Preden se lotimo prikaza uporabe medpomnilnika premislimo, kakšno podatkovno strukturo sploh potrebujemo v primeru, ko je *potrebno sprejeti podatke in jih shraniti za kasnejšo obdelavo* (npr. hranjenje informacije o pritisnjenih tipkah, hranjenje sprejetih podatkov preko serijskega vmesnika ipd.).

Podatkovna struktura mora očitno biti dovolj velika, da lahko shrani podatke, ki jih trenutno ne moremo obdelati, ker izvajamo neke druge rutine. Primerna bi torej bila na primer tabela oziroma polje (angl. array) ustrezne dolžine.

Ta podatkovna struktura mora biti očitno tudi organizirana tako, da *podatki izstopajo iz podatkovne strukture v istem vrstnem redu, kot so vanjo vstopili*. Če to ne bi veljalo, bi nam uporaba podatkovne strukture lahko na primer obrnila vrstni red sporočila (npr. sporočilo "`halo`" bi postalo "`olah`"). Taki podatkovni strukturi, ki jo potrebujemo, v žargonu pravimo tudi ***FIFO medpomnilnik*** (angl. First-In-First-Out buffer), saj podatek, ki je najprej prišel v medpomnilnik, je tudi podatek, ki bo najprej prebran iz medpomnilnika.

Tekom laboratorijskih vaj bomo uporabljali t. i. *ciklični medpomnilnik* (angl. cyclic buffer). Funkcionalnost cikličnega medpomnilnika smo implementirali v modulu `buf.c`. Pomembno je, da znate ta modul pravilno uporabiti in s tem namenom si bomo zato ogledali primer uporabe v naslednjem poglavju.

Uporaba cikličnega medpomnilnika `buf.c`

Poglejmo si ključne korake pri uporabi cikličnega medpomnilnika, ki ga implementira modul `buf.c`.

Da sploh dobimo funkcionalnost modula `buf.c`, je potrebno seveda najprej modul vključiti z `include` stavkom takole:

```
#include "buf.h"
```

Modul `buf.c` omogoča, da *katerokoli tabelo tipa `uint8_t` uporabimo kot ciklični medpomnilnik*. Zato najprej definiramo tabelo, ki jo bomo uporabljali kot medpomnilnik. In običajno dolžino tabele *parametriziramo* z makrojem. Poglejte primer:

```
#define BUF_LEN      64
uint8_t buffer_array[BUF_LEN];
```

Za delo s cikličnim medpomnilnikom potrebujemo še t. i. "*handle structure*" strukturo, s pomočjo katere bomo navadno tabelo `buffer_array[]` uporabljali kot ciklični medpomnilnik. Zato v naslednjem koraku definiramo "handle" strukturo cikličnega medpomnilnika:

```
buf_handle_t  buffer_handle;
```

V strukturni spremenljivki `buffer_handle` bomo torej hranili vse potrebne parametre za delo z medpomnilnikom. Vendar je zaenkrat ta "handle" struktura še neuporabna, saj še ni inicializirana. Zato v naslednjem koraku s pomočjo funkcije `BUF_init()`,

```
void BUF_init(buf_handle_t *buf_handle, uint8_t *buffer_ptr, uint32_t buf_length)
```

inicializiramo "handle" strukturo. Če nadaljujemo s primerom, sledi:

```
BUF_init( &buffer_handle, buffer_array, BUF_LEN);
```

Prvi vhodni argument funkcije je *naslov* "handle" strukture medpomnilnika `&buffer_handle`, drugi vhodni argument je tabela `buffer_array[]`, ki jo želimo uporabljati kot medpomnilnik in tretji argument je dolžina medpomnilnika `BUF_LEN`. Vidite, da je res ugodno to dolžino parametrizirati z makrojem.

Sedaj imamo pripravljeno vse, da pričnemo tabelo `buffer_array[]` uporabljati kot cikličnimi medpomnilnik. Poglejmo si kako to storimo na najbolj osnovni ravni, kjer shranjujemo in beremo *le po en bajt* podatkov.

V medpomnilnik shranimo en bajt podatkov s funkcijo `BUF_store_byte()`,

```
buf_rtrn_codes_t BUF_store_byte(buf_handle_t *buf_handle, uint8_t data)
```

V našem primeru bi torej v medpomnilnik shranili vrednost 10 takole:

```
BUF_store_byte(&buffer_handle, 10);
```

Prvi vhodni argument funkcije je naslov "handle" strukture, ki upravlja z medpomnilnikom, torej `&buffer_handle`. Drugi vhodni argument pa je podatek, torej število 10.

Če pogledate definicijo funkcije `BUF_store_byte()`, boste opazili, da funkcija pravzaprav *vrača vrednost* tipa `buf_rtrn_codes_t`. Funkcije za delo s cikličnim medpomnilnikom so namreč implementirane tako, da *vračajo kodo, ki sporoča, ali je bila izvedba funkcije uspešna* ali ne. V premislek: lahko bi se zgodilo, da bi želeli v medpomnilnik shraniti vrednost, pa bi bil medpomnilnik že poln. Ta primer bi lahko zaznali s pomočjo *vrnjene kode funkcije* (angl. *return codes*). V spodnji tabeli so navedene vrnjene kode in kratek opis njihovega pomena. Mimogrede, te kode so implementirane s pomočjo naštevnega tipa.

`BUFFER_OK`	operacija z medpomnilnikom je bila uspešna
`BUFFER_FULL`	medpomnilnik je poln, operacija je bila neuspešna (npr. shranjevanje podatka)
`BUFFER_EMPTY`	medpomnilnik je prazen, operacija je bila neuspešna (npr. branje podatka)
`BUFFER_NOT_ENOUGH_SPACE`	v medpomnilniku ni dovolj prostora za shranjevanje niza podatkov
`BUFFER_NOT_ENOUGH_DATA`	v medpomnilniku ni dovolj podatkov za branje niza podatkov

Za poenostavljeno uporabo medpomnilnika je pravzaprav dovolj, če le preverjamo, ali je funkcija vrnila kodo `BUFFER_OK` in tako preverjamo, ali je bila operacija uspešna ali ne. Včasih pa lahko situacijo še malenkost poenostavimo in pravzaprav v celoti ignoriramo vrnjene kode funkcije.

Ko želimo iz medpomnilnika prebrati naslednji podatek, uporabimo funkcijo `BUF_get_byte()`,

```
buf_rtrn_codes_t BUF_get_byte(buf_handle_t *buf_handle, uint8_t *data)
```

Prvi vhodni argument funkcije je *naslov* "handle" strukture za medpomnilnik, medtem ko je drugi vhodni argument *naslov, kamor naj se prebrani podatek shrani*. Če torej pokažemo na našem primeru:

```
uint8_t a;

BUF_get_byte( &buffer_handle, &a );
```

Najprej pripravimo spremenljivko "`a`", kamor bomo iz medpomnilnika prebrali vrednost, nato pa *naslov* spremenljivke "`&a`" posredujemo funkciji za branje podatka. Ne pozabite: tudi funkcija `BUF_get_byte()` vrne kodo, ki sporoča, ali je bilo branje iz medpomnilnika uspešno.

Za konec zberimo vso programsko kodo primera uporabe cikličnega medpomnilnika na enem mestu, da bo stvar preglednejša:

```
#include "buf.h"                    // vključitev knjižnice

#define BUF_LEN   64                // definicija dolžine bufferja
uint8_t buffer_array[BUF_LEN];      // def. tabele za implementacijo bufferja


buf_handle_t   buffer_handle;       // definicija "handle" strukture za buffer
BUF_init( &buffer_handle, buffer_array, BUF_LEN );    // init."handle" strukture

BUF_store_byte( &buffer_handle, 10 ); // shranimo število 10 v buffer


uint8_t a;              // spremenljivka, kamor preberemo podatek iz bufferja
BUF_get_byte( &buffer_handle, &a );   // branje podatka iz bufferja
```

V zgornjih vrsticah imate torej vso potrebno kodo, da poljubno tabelo uporabite kot ciklični medpomnilnik in vanj pišete in berete bajt po bajt. Zavedajte se le, da *po potrebi kodo lahko nadgradite tako, da pri funkcijah za pisanje in branje upoštevate še vrnjene kode o uspešnosti izvedbe*.

Če vas morda zanima podrobnejša razlaga delovanja cikličnega medpomnilnika, si lahko pogledate poglavje o implementaciji cikličnega medpomnilnika spodaj.

Implementacija *krožnega medpomnilnika* na podlagi tabele

Kakšen medpomnilnik sploh rabimo?

Preden se lotimo okvirne razlage implementacije *krožnega medpomnilnika* premislimo, kakšno podatkovno strukturo sploh potrebujemo v primeru, ko je *potrebno sprejeti podatke in jih shraniti za kasnejšo obdelavo* (npr. hranjenje informacije o pritisnjenih tipkah, hranjenje sprejetih podatkov preko serijskega vmesnika ipd.).

Podatkovna struktura mora očitno biti dovolj velika, da lahko shrani podatke, ki jih trenutno ne moremo obdelati, ker izvajamo neke druge rutine. Primerna bi torej bila na primer tabela oz. polje (angl. array) ustrezne dolžine.

Ta podatkovna struktura mora biti očitno tudi organizirana tako, da podatki *izstopajo* iz podatkovne strukture *v istem vrstnem redu, kot so vanjo vstopili*. Če to ne bi veljalo, bi nam uporaba podatkovne strukture lahko npr. obrnila vrstni red sporočila (npr. sporočilo `"halo"` bi postalo `"olah"`). Taki podatkovni strukturi, ki jo potrebujemo, v žargonu pravimo tudi *FIFO medpomnilnik* (angl. First-In-First-Out buffer).

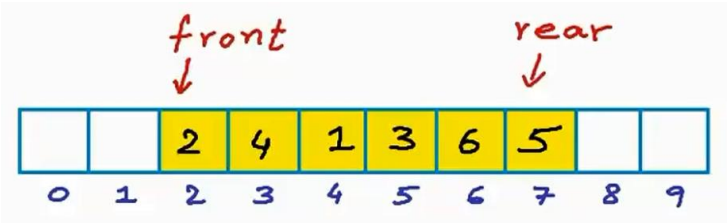
Ideja vrste (angl. queue)

Podatkovna struktura, ki smo jo nakazali zgoraj, se v splošnem imenuje *vrsta* (angl. queue). V našem primeru jo lahko *implementiramo* s tabelo oz. poljem (angl. array). Ideja je sledeča: tabelo "nadgradimo" tako, da jo "opremimo" s spremenljivko – kazalcem, ki pove, kje se nahaja *začetek vrste* (torej prvi element, ki je vstopil v vrsto) ter spremenljivko, ki pove, kje se nahaja *konec vrste* (tj. zadnji element, ki je vstopil v vrsto). Z začetka vrste očitno jemljemo podatke, na koncu vrste pa seveda dodajamo nove podatke.

Ti dva kazalca imata tipično sledeča imena:

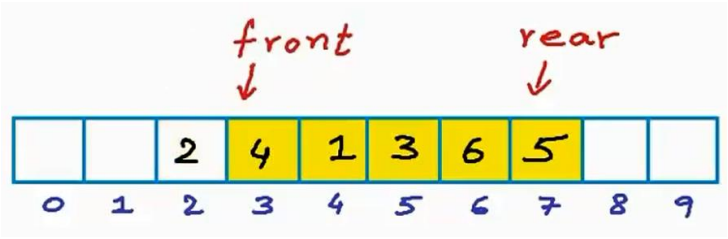
- "start", "front", "head", "out", "read" za začetek vrste ter
- "end", " rear", "tail", "in", "write" za za konec vrste.

Običajno sta ti dva kazalca kar *indeksa* (tj. celo pozitivno število) , ki povesta, *na katerem mestu v tabeli* imamo začetek in konec vrste (glejte sliko spodaj: indeksi so pisani pod tabelo z modro barvo).



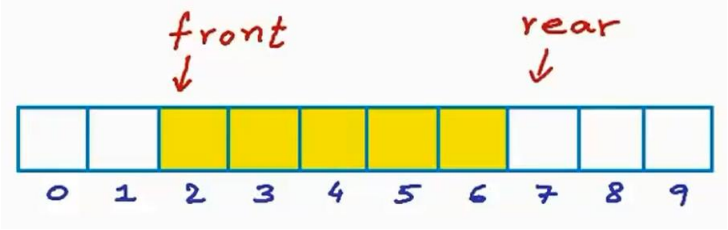
Slika 1 - indeksa "front" in "rear" kažeta na začetek in konec vrste

Prednost uporabe kazalcev na začetek in konec vrste je, da ko iz vrste preberemo naslednji element (tj. na poziciji "front"), ni potrebno premikati vseh preostalih podatkov v vrsti za en element naprej, pač pa le popravimo kazalec "front" tako, da ga povečamo za 1 – glejte sliko spodaj. Prebrana vrednost (število 2 v spodnjem primeru) se sicer še vedno nahaja v tabeli. Nič hudega, saj jo bomo čez čas prepisali. Vemo pa, da se celotna naša vrsta podatkov nahaja med kazalcema "front" in "rear".



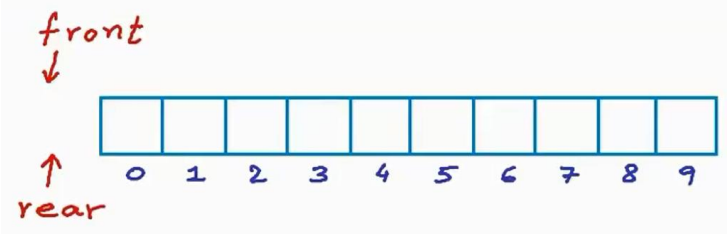
Slika 2 - branje elementa z vrste povzroči premik kazalca "front" za 1 naprej

Ko pa v vrsto vstavljamo nov element, ga vstavimo na mesto, ki ***sledi mestu***, na katerega kaže "rear". Glejte spodaj.



Slika 3 - vstavljanje novega elementa v vrsto

Poudarimo naj, da je potrebno ***posebej obravnavati primer, ko* je *vrsta prazna*** in vanjo vstavljamo prvi element! Zakaj? Ker takrat vrednosti kazalcev "front" in "rear" nimata smisla, saj vrsta ne obstaja.



Slika 4 - ko je vrsta prazna, kazalca "front" in "rear" nimata smisla in ju zato v tem primeru ponastavimo

Iz povedanega sledi, da bo potrebno v funkciji za realizacijo vstavljanja novega elementa v vrsto preverjati, ali ni morda vrsta prazna. Če je prazna, nastavimo kazalca "front" in "rear" tako, da oba kažeta na prvi element v tabeli in nato tja vstavimo prvi element.

S psevdokodo lahko trenutno idejo algoritma za dodajanje novega elementa zapišemo takole:

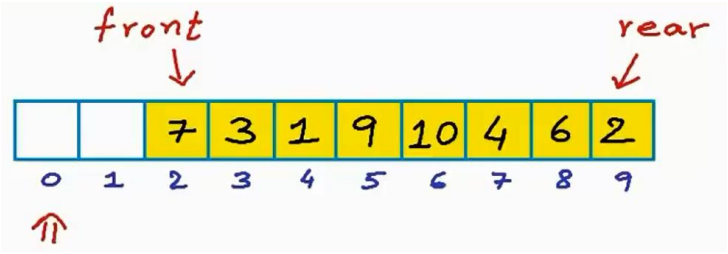
```
Add to queue(x)
{
     if IsFull()
          return

     else if IsEmpty()
     {
          front = rear = 0
     }
     else
     {
          rear = rear + 1
     }

     Array[rear] = x
}
```

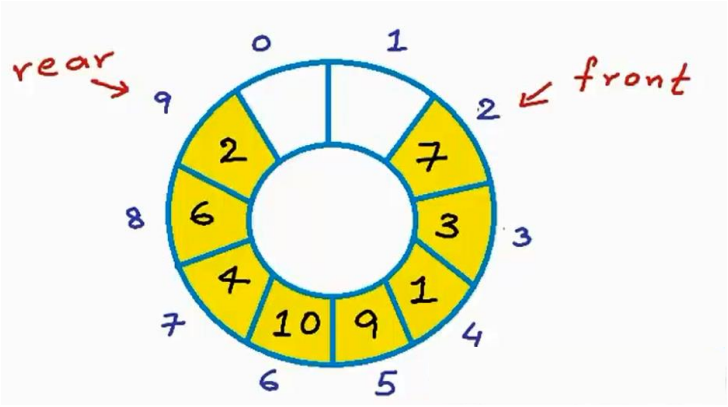
Ko vrsta postane krožna

Kaj pa storiti, ko pa pridemo do konca tabele?



Slika 5 - situacija, ko vrsta pride do konca tabele

Vidimo, da imamo na začetku tabele še prosta mesta. Torej ideja: s konca tabele nadaljujemo sedaj vrsto na začetku tabele! Tako se konec in začetek tabele pravzaprav skleneta (glejte sliko spodaj) in dobimo *krožno (ciklično) tabelo* in posledično naš *krožni medpomnilnik*!



Slika 6 - če konec tabele sklenemo z začetkom tabele, dobimo krožno (ciklično) tabelo

Vidimo lahko, da ideja *ciklične* tabele vpelje *cikličnost* tudi v aritmetiko, ki se tiče kazalcev – indeksov. Indeksi "front" in "rear" tako tudi "*tečejo v krogu*". Za naš primer: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, **0, 1** itd. Če smo matematično bolj spretni, opazimo da se pravzaprav indekse sedaj računa z ***aritmetiko po modulu***, kjer je modul enak kar dolžini tabele.

Na tem mestu lahko popravimo idejo algoritma za vstavljanje novega elementa v vrsto:

```
Add to queue(x)
{
     if IsFull()
          return

     else if IsEmpty()
     {
          front = rear = 0
     }
     else
     {
          rear = (rear + 1) % N
     }

     Array[rear] = x
}
```

Vidimo, da algoritem najprej preveri, če je krožna tabela polna.

Če ni, v naslednjem koraku pripravi ustrezno vrednost za kazalec "rear", pri čemer posebej obravnava primer, ko je vrsta prazna.

Šele nato vpiše vrednost na konec vrste.

Pohitritev algoritma z implementacijo brez računanja po modulu

Na tem mestu je smiselno opozorilo: uporaba aritmetike po modulu je elegantna, vendar pa je *lahko* za preprostejše mikrokrmilnike velik računski zalogaj. Zato smo v našem primeru aritmetiko po modulu izvedli kar z uporabo if-then stavka, ker bo tako delovanje hitrejše. Idejo pohitritve si poglejte spodaj.

```
rear = (rear + 1) % N
```

➔

```
rear = rear + 1
if (rear = N)
      rear = 0
```

Še podrobnejša razlaga implementacije krožnega medpomnilnika

Če vas zanima še podrobnejša razlaga implementacije krožnega medpomnilnika, si lahko na spletu ogledate video lekcijo z naslovom "Data structures: Array implementation of Queue".