Obstojni pomnilnik v vgradnih sistemih

Jaka Močnik
Active Tools d.o.o., Kranj, Slovenija

jaka@activetools.si

Andrej Brodnik
IMFM, Ljubljana, Slovenija
University of Technology, Luleå, Švedska
andrej.brodnik@imfm.uni-lj.si

Povzetek

V vgrajenih računalniških sistemih se pogosto pojavi potreba po hranjenju podatkov, ki morajo biti obstojni (persistent) med več zagoni sistema. V članku opisujemo sistem za upravljanje s takšnimi podatki, ki omogoča razvijalcu programske opreme kar se da preprosto delo. Za dostop do podatkov sistem nudi posplošene reference*, ki ohranjajo vrednost med zagoni. Poleg tega v prispevku opisujemo primer povezave med našim sistemom in strojno opremo, v kateri se shranjeni podatki med zagoni ne spreminjajo (baterijsko podprti pomnilnik).*

1 Uvod

Ena od osnovnih funkcij računalnikov je dolgoročno hranjenje podatkov, ki morajo biti obstojni med več zagoni sistema. Za zagotavljanje obstojnosti podatkov se običajno uporabljajo sekundarni pomnilniki (prim. [9]); pred obdelavo se podatki izrecno prenesejo v primarni pomnilnik, obdelajo in nato ponovno zapišejo v sekundarni pomnilnik. Zavoljo takšnega dela ima programer opraviti z dvema pogledoma na isti podatek. Predstavitev podatka se poleg tega razlikuje pri različnih tipih sekundarnih pomnilnikov. S sistemom, ki bi poenotil pogled na podatke in skril tip sekundarnega pomnilnika pred programerjem, bi močno poenostavili razvoj aplikacij in olajšali ponovno uporabo tako razvitega programja.

V naslednjem razdelku opisujemo nekatere znane tehnike metode za upravljanje s pomnilnikom ter njihove prednosti in slabosti. Sledi razdelek z opisom programske arhitekture našega sistema, in nato v četrtem razdelku še opis programskega vmesnika do sistema. V petem razdelku podajamo primer izvedbe sistema, ki uporablja pomnilnik z baterijskim napajanjem. Zaključili bomo s predlogi za možne razširitve sistema.

2 Upravljanje s pomnilnikom

Osnovna lastnost pomnilnika je, da lahko v njem hranimo podatke, jih pogledamo in, v splošnem, spreminjamo. Ohranjenost podatkov v pomnilniku zavisi od fizikalnih lastnosti pomnilnika. Fizikalne lastnosti pomnilnika tudi določajo kako hitro lahko dostopamo do pomnilnika – slednja lastnost definira tako imenovano *pomnilniško hierarhijo*. Običajna razdelitev hierarhije, glede na neposrednost procesorjevega dostopa do pomnilnika, je na *primarni pomnilnik* (registri, predpomnilnik in glavni pomnilnik) in na *sekundarni pomnilnik* (diskovje, tračne enote itd.). Sekundarni pomnilnik ima dodatno lastnost, da so podatki shranjeni v njem nespremenljivi med zagoni računalnika – rečemo, da so *obstojni* (*persitent*). V tem prispevku lastnost obstojnosti ločimo od lastnosti sekundarnosti.

Metode, ki naredijo delo s podatki v sekundarnem pomnilniku transparentno za programerja so bile deležne velike pozornosti. Predstavljajo kombinacijo orodij in rutin, ki jih nudita operacijski sistem (npr. [15]) in prevajalnik ter sistem jezika (npr. [2, 8]).

Z razširitvijo metod, ki se uporabljajo za podatke v primarnem pomnilniku na podatke v sekundarnem pomnilniku dobimo *navidezni* (*virtualni*) *pomnilnik* ([15, pp. 74–141] in [1]) oziroma, če zagotavljamo tudi obstojnost podatkov, enonivojsko shrambo podatkov ([7]). Osnovna prednost uporabe navideznega pomnilnika za programerja je, da so podatki v sekundarnem pomnilniku videti dosegljivi na enak način kot podatki v primarnem pomnilniku. Dosežemo *poenotenje* (*unifikacijo*) pomnilnika.

Uvedba navideznega pomnilnika sicer omogoča rabo konstruktov prirejanja za dostop do podatkov v sekundarnem pomnilniku, vendar zahteva uporabo strojne opreme, ki je običajno v preprostih vgrajenih računalniških sistemih ni. Tipičen primer izvedbe navideznega pomnilnika je *preslikovanja pomnilnika* (*memory mapping*) ([7]), ki pa zahteva posebno enoto za upravljanje s pomnilnikom, *MMU* ([9]). Takšne enote preprostejši procesorji, ki se najpogosteje uporabljajo v kontrolerjih, nimajo (npr. [16, 10]).

Prirejanje se pri sintaktični analizi prevede v dvo-parametersko funkcijo `assign`, ki jo nato ustrezno razširi prevajalnik v odvisnosti od jezika ([2]). Parametra funkcije sta referenca spremenljivke, kamor naj se shrani vrednost in sama vrednost. Za dostop do spremenljivke obstaja v prevajalniku podobna abstrakcija. Seveda ni nobene potrebe, da bi funkcijo razširjal prevajalnik, ampak je to lahko prepuščeno programerju – primerjaj predefiniranje (*overloading*) v funkcijskih in stvarno (predmetno) usmerjenih jezikih (npr. [6, 5, 8, 14, 13, 11, 12, 4]).

Sam pomnilnik moramo dodatno še upravljati in dober pregled algoritmov za upravljanje s pomnilnikom ponuja [3].

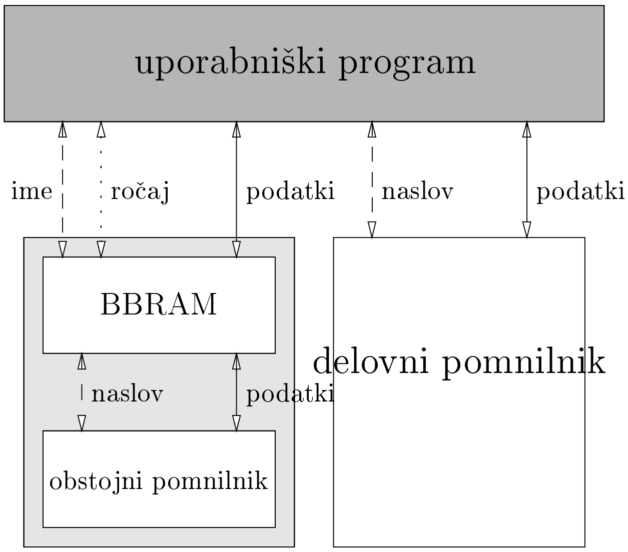
3 Pomnilniški sistem BBRAM

V tem razdelku opisujemo preprosto arhitekturo za upravljanje in delo z obstojnimimi podatki. Arhitekturo in samo izvedbo imenujemo *BBRAM*, kar je okrajšava za *Battery Backed-up RAM*. Čeravno ime namiguje na baterijsko podprti pomnilnik za hrambo podatkov, je to zgolj naša konkretna izvedba, medtem ko v splošnem ni razloga, da bi obstojnih podatkov ne hranili drugje, na primer na disku ali podobno.

Za upravljanje s fizičnim pomnilnikom in za dostop do podatkov v njem uporabljamo *naslov*. Naslovi se razlikujejo glede na uporabljen pomnilnik: pri delovnem pomnilniku je to (običajno) 32-bitno število, pri disku je lahko trojica (glava, sled, sektor), pri datotečnem sistemu datoteka in odmik v njej, pri flash ROM pomnilniku zopet število itd. Naslov dobimo bodisi z dinamičnim zaseganjem pomnilniškega prostora ali uporabimo stalen, absoluten naslov.

Zavoljo zahteve, da naj bo možno na enak način uporabljati poljuben pomnilnik, za dostop podatkov uporabljamo posplošene reference ali *imena*. V našem sistemu BBRAM so imena črkovni nizi, ki jih sistem imena pretvarja v fizične naslove (gl. sliko 1). Z drugimi besedami, uporabniški program dostopa do kakršnegakoli pomnilnika s pomočjo referenc (črtkane črte na sliki 1), ki so pri fizičnih pomnilnikih naslovi, kot smo jih opisali zgoraj, medtem ko so pri BBRAM sistemu imena. Pri delovnem pomnilniku nato prevajalnik poskrbi, da se reference pravilno uporabljajo v programju, medtem ko pri pomnilniku BBRAM uporabnik izrecno kliče funkcije, ki jih nudi sistem BBRAM.

Pri delovnem pomnilniku zna prevajalnik izboljšati kakovost kode tako, da seli podatke po pomnilniški hierarhiji. Pomnilnik BBRAM nudi možnost za podobno izboljšanje kode, saj uporabniku lahko vrne *ročaj* (*finger*, pikčasta črta na sliki 1), ki ga nato uporabniški program uporablja za hitrejši dostop do podatkov.



Slika 1: Preslikovanje imena

3.1 Nekaj podrobnosti o izvedbi

Informacijo o imenih in njihovo povezavo s podatki hranimo skupaj s podatki v istem pomnilniku, ker mora biti tudi ta obstojna. Obstojni pomnilnik torej vsebuje tudi opis svoje vsebine, metavsebino, ki jo imenujemo *imenik*.

Dostop do podatka preko njegovega imena je potrebno realizirati programsko, s preiskovanjem imenika podatkov, kar je – do neke mere odvisno od izvedbe imenika – tipično draga operacija. Pospešitev dosežemo tako, da uporabimo ročaj. Ročaj je prozorna (*opaque*) podatkovna struktura, katere ustroj ni znan uporabniškemu programu. Na ta način dosežemo dva cilja: najprej, če se izvedba ročaja spremeni, tega uporabniški program sploh ne zazna in drugo, uporabniški program ne more (nehote) pokvariti podatkov.

Delo s podatki je, mogoče le, kadar so ti v delovnem pomnilniku za kar imamo na voljo vrsto funkcij. Ko uporabniški program zahteva od sistema BBRAM prenos podatkov med obema pomnilnikoma, se le-ta zgodi brez njegovega posredovanja – podobno kot prenos DMA (*direct memory access*) pri strojni opremi. Sistem BBRAM pozna preproste, skalarne podatke, ki jih lahko prenesemo na enkrat in sestavljene podatke (vektorji), ki jih lahko prenašamo po delih.

V samem obstojnem pomnilniku ali pri zapisovanju vanj lahko pride do napak (izpad napajanja, ipd.). Pomnilnik BBRAM nudi zato dodatno zaščito podatkov in v primeru nepravilnosti pri vsebini pomnilnika obvesti uporabnika o tem. Zato je vsakemu podatku dodana še nadzorna informacija, ki jo, logično gledano, hranimo kot del imenika. Mogočih je več načinov izvedbe preverjanja: od preproste paritete do rabe zgoščevalnih funkcij, med katerimi si izberemo primerno glede na njihovo zahtevnost in stopnjo zanesljivosti, ki jo želimo.

4 Programski vmesnik

V tem razdelku je opisan definicijski modul (programski vmesnik), ki omogoča rabo sistema BBRAM. Modul BBRAM je zasnovan stvarno usmerjeno. Zatorej ima modul poleg vseh ostalih funkcij za delo s podatki tudi funkcijo, ki ga ustvari, in funkcijo, ki ga ugasne. Opis modula je razdeljen na dva dela: najprej so opisani osnovni tipi in zagon sistema in v drugem delu samo delo s sistemom.

4.1 Osnovni tipi in zagon modula

Za delo s sistemom BBRAM uporabniški program potrebuje definicije naslednjih tipov: sam obstojni pomnilnik (bbram), ime, ročaj in podatki. BBRAM je prozoren tip, katerega dejanska izvedba je odvisna od uporabljenega obstojnega pomnilnika:

```
typedef void *tBbram;
```

Kot rečeno, je ime poljuben črkovni niz:

```
typedef char *tBbramName;
```

Kot sam obstojni pomnilnik, je tudi ročaj prozoren, od posamezne izvedbe odvisen tip:

```
typedef void *tBbramFinger;
```

Končno, podatki so predstavljeni z večobličnim tipom:

```
typedef struct _tBbramData {
    unsigned char *data;
    unsigned int   size;
} tBbramData;
```

Tvorjenje (zagon) je odvisno od vrste uporabljenega pomnilnika, zato tudi zanj uporabimo večoblično obliko podpisa:

```
tBbram bbramInit(int nargs, ...);
```

V odvisnosti od vrste pomnilnika na različne načine navedemo, kje se nahajajo naši podatki: pri flash ROM pomnilniku začetni naslov, pri datotečnem sistemu pot do imenika...

4.2 Zaseganje in sproščanje pomnilnika

Za vsak podatek moramo najprej zaseči pomnilnik. Običajno je edini potrebni parameter velikost zaseženega pomnilnika, medtem ko moramo v našem sistemu ob zaseganju navesti tudi ime podatka, za katerega bo zasežen pomnilnik. To ime bomo kasneje uporabljali kot osnovno referenco za dostop do podatka. Pri zaseganju se najprej preišče imenik: ako podatek s tem imenom že obstaja, vrnemo njen ročaj ter programerja obvestimo, da je dobil ročaj že zaseženega podatka. Sicer poskušamo zaseči potreben prostor: če uspemo, dodamo podatek v imenik ter vrnemo njen ročaj, v nasprotnem primeru vrnemo neveljaven ročaj.

Zaseganje prostora za podatek zahteva intenzivno delo z imenikom in je zavoljo tega običajno draga operacija. Za zaseganje uporabimo funkcijo `bbramAlloc`:

```
tBbramFinger bbramAlloc(tBbram       *bbram,
                        unsigned int size,
                        tBoolean     *prealloced);
```

Vrednost na katero kaže `prealloced` se nastavi na `FALSE`, če smo pomnilnik zasegli na novo, oziroma na `TRUE`, če je bil pomnilnik za podatek s tem imenom v željeni velikosti že zasežen.

Sproščanje lahko zahtevamo preko imena ali preko ročaja: v vsakem primeru se poišče ustrezen vnos v imeniku ter sprosti prostor, ki ga zaseda. Za sproščanje pomnilnika uporabimo eno od funkcij:

```
void bbramFreeN(tBbram           *bbram,
                const tBbramName name);
void bbramFreeH(tBbram             *bbram,
                const tBbramFinger finger);
```

Ročaj, s katerim lahko pospešimo dostop do podatkov, dobimo s funkcijo:

```
tBbramFinger bbramGetFinger(tBbram           *bbram,
                            const tBbramName name);
```

4.3 Prenos podatkov v in iz obstojnega pomnilnika

Branje podatkov iz obstojnega pomnilnika izvedemo s klicem funkcije:

```
tBbramStatus bbramGetN(tBbram           *bbram,
                       const tBbramName name,
                       tBbramData       *dest);
```

Ponor podatkov `dest` ima lahko člana zapisa `data` nastavljen na neveljaven naslov (`NULL`): tedaj bo sistem sam zasegel primerno količino primarnega pomnilnika ter primerno nastavil vrednost `data`. Sicer mora `data` kazati na že zasežen pomnilnik in `size` vsebovati njegovo velikost: za prenos se bo uporabil ta pomnilnik. V obeh primerih bo po uspešnem prenosu član `size` vseboval količino prenešenih zlogov.

Pred branjem se preveri pravilnost shranjenih podatkov v obstojnem pomnilniku s pomočjo kontrolne informacije. Funkcija vrača vrednost `BBRAM_OK`, če branje uspe, `BBRAM_NOT_FOUND`, če podatek ne obstaja in BBRAM_DIRTY, če je bila njegova vsebina pokvarjena!

Za zapisovanje podatkov v sekundarni pomnilnik uporabimo funkcijo:

```
tBbramStatus bbramSetN(tBbram           *bbram,
                       const tBbramName name,
                       const tBbramData *src);
```

V izvoru podatkov `src` član zapisa `data` vsebuje referenco na prepisovane podatke in `size` njihovo velikost (v zlogih). Vrnjena vrednost funkcije je BBRAM_OK, če je zapisovanje uspelo, BBRAM_ERR, če ni, in BBRAM_NOT_FOUND, če podatek ne obstaja. Ob zapisovanju se ponovno izračuna in popravi kontrolna informacija podatka.

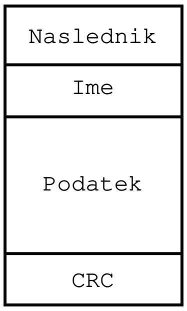
Branje in zapisovanje lahko izvršimo tudi z dostopom preko ročaja:

```
tBbramStatus bbramGetH(tBbram             *bbram,
                       const tBbramFinger finger,
                       tBbramData         *dest);
tBbramStatus bbramSetH(tBbram             *bbram,
                       const tBbramFinger finger,
                       const tBbramData   *src);
```

5 Primer izvedbe

Oglejmo si izvedbo z baterijsko napajanim RAM pomnilnikom, pogostim tipom pomnilnika v vgrajenih sistemih. Obstojni pomnilnik sestoji iz baterijsko napajanega RAM pomnilnika, ki ga lahko procesor neposredno naslavlja. Določen je z naslovom začetka BRAM_START_ADDRESS in velikostjo BBRAM_SIZE.

Reference, imena, so poljubno dolgi nizi črk ki so zaključeni z zlogom 0. Vsak podatek je v baterijskem pomnilniku zapisan kot prikazuje slika 2. Prvo polje



Slika 2: Zgradba podatka

vsebuje naslov naslednjega podatka, sledi poljubno dolg niz, prostor za sam podatek in 32-bitna kontrolna informacija, CRC vrednost vsebine prejšnjih treh.

S pomočjo naslovov v prvem polju so vsi podatki povezani v vrsto in tvorijo imenik. Predpostavimo še, da prvi podatek vedno zasežemo na naslovu BBRAM_START_ADDRESS, s čimer zagotovimo, da vedno poznamo prvi vnos v imenik. Poljuben podatek je, denimo, da je vseh podatkov n, dosegljiv preko njegovega imena v času $O(n)$. S pomočjo ročaja, ki je kar naslov podatka v baterijskem pomnilniku, je poljuben podatek dosegljiv v času $O(1)$.

Na najvišjem naslovu pomnilnika hranimo naslov prvega prostega bloka in proste bloke povežemo v vrsto podobno kot bloke s podatki. Prostor za podatke zasežemo z uporabo postopkov, ki so znani že iz klasičnega upravljanja s pomnilnikom (prim. [3] in [15, pp. 74-141]). Zaželjeno je, da pri izbiri postopkov premislimo tudi možnost razdrobljenosti pomnilnika, saj ne premoremo MMU, ki bi ta problem vsaj delno odpravil, ter se odločimo za morebitne postopke združevanja drobcev pomnilnika.

Takšno implementacijo lahko optimiziramo tako, da delamo s podatkom kar v baterijskem pomnilniku, namesto da bi ga prenašali v primarni pomnilnik. Seveda moramo še vedno izrecno poklicati funkciji `bbramGetN()` in `bbramSetN()`, ki preverita oziroma obnovita kontrolno informacijo.

Ker se naš sistem uporabljala v večopravilnem okolju, mora poskrbeti tudi varne sočasne dostope do podatkov. Sistem omogoča poljubno število sočasnih branj in eno samo pisanje.

6 Zaključek

V prispevku smo predstavili modul za delo z obstojnimi podatki. Ker sistem strogo loči med uporabnikovim pogledom na pomnilnik in uporabljenim obstojnim pomnilnikom, smo lahko modul definirali zelo splošno. Obstojne reference na podatke, s pomočjo katerih uporabnik dostopa do podatkov, smo definirali kot črkovne nize. Za pospešitev dostopa smo vpeljali še ročaje.

S praktičnega zornega koda bi v prihodnje veljalo modul razširiti z možnostjo postavitve določenih podatkov na v naprej določeno mesto v pomnilniku, kar se izkaže za zelo uporabno posebno pri vgrajenih sistemih. Zaradi te predoločenosti postavitve podatkov bo potrebno tudi spremeniti ustroj imenika.

S teoretičnega zornega kota obstaja kar nekaj zanimivih problemov: najprej, kako narediti izvedbo sistema čim bolj večoblično ter tako prenosljivo med sistemi z različnimi obstojnimi pomnilniki. Verjetno bi, zaradi poenostavitve dela s pomnilnikom, veljalo modul narediti kot rekurzivno podatkovno strukturo, kjer sta opis modula in celo imenik samo dva primera uporabe modula. Tako bi se opis sistema verjetno prevedel na eno samo vrednost (število, niz). Drugi problem nastopa pri pogostem spreminjanju vrednosti podatkov. Zahtevnost te operacije najbolj zavisi od uporabljene nadzorne funkcije (CRC). Kako izboljšati nadzorno funkcijo tako, da bo še vedno opravljala svojo funkcijo, a bo zahtevnost njenega računanja odvisna od velikosti vpisanega dela podatkov in ne od velikosti celotnega podatka. Slednje je posebej pomembno pri podatkih, ki niso preprosti, skalarni, podatki.

Literatura

[1] C.T. Clingen A. Bensoussanand. The multics virtual memory: Concepts and design. *Communications of the ACM*, 15(5):308–318, May 1973.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers - Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[3] J.E. Hopcroft A.V. Aho and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[4] R. Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, Englewood Cliffs, New Jersey, 1998.

[5] L. Cardelli. The semantics of multiple inheritance. *Information and computation*, 76:138–164, 1988.

[6] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[7] A.K. Goel. *Exact Positioning of Data Approach to Memory Mapped Persistent Stores: Design, Analysis and Modelling.* PhD thesis, University of Waterloo, 1996.

[8] S.P. Harbison. *Modula-3.* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[9] D. Kodek. *Mikroprocesorski sistemi.* BI-TIM d.o.o., Ljubljana, Slovenia, 1993.

[10] Motorola. Mpc8266ads – pci, user's manual, 2001. Board Rev. PROTOTYPE-B.

[11] H. Schildt. *Java 2: the complete reference.* Osborne/McGraw-Hill, Berkeley, CA, 2001.

[12] R. Stansifer. *ML Primer.* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[13] B. Stroustrup. *C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, 2^{nd} edition.

[14] Svensk standard SS 63 61 14. Databehandling programspråk – SIMULA, 1987.

[15] A.S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, Inc., Englewood Cliffs, NJ, 1992.

[16] Virata. Bd6210 (helium 210) development board hardware guide, 2001.