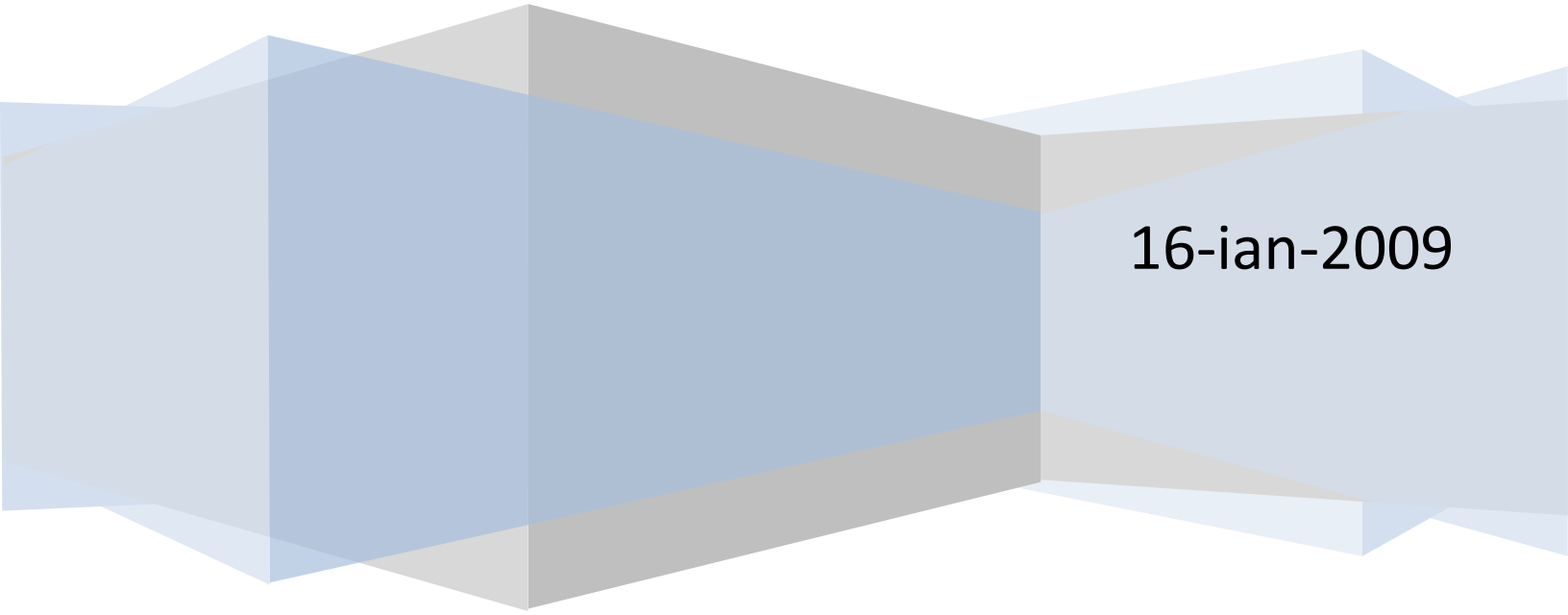
Facultatea de Automatică și Calculatoare
Sisteme Multiprocesor

Programarea multiprocesoarelor

Concurența în sistemul de operare și accesul la regiștri

Andrei Ciorba – 342 C1



16-ian-2009

Andrei Ciorba – 342 C1

PROGRAMAREA MULTIPROCESOARELOR

Concurența în sistemul de operare și accesul la regiștri

Introducere

Industria calculatoarelor suferă avansuri din ce în ce mai spectaculoase odată cu trecerea timpului. Totodată, o schimbare oarecum surprinzătoare a intervenit în momentul în care producătorii majori de cipuri au renunțat la tendința de a crea procesoare mai rapide, cel puțin pentru momentul de față. Legea lui Moore nu a fost scoasă încă din "uz", drept dovadă stând faptul că în fiecare an din ce în ce mai multe tranzistoare sunt incluse în aceleași cipuri, dar de data aceasta fără scopul de a crește viteza de calcul prin creșterea frecvenței. Degajările de căldură din astfel de sisteme ce lucrează la frecvențe din ce în ce mai mari devin o problemă prea greu de rezolvat.

Producătorii se axează pe tehnologiile ce exploatează paralelismul, spre arhitecturi multicore în care mai multe procesoare, sau core-uri comunică direct prin memorii chache implementate în hardware. Adevărata încercare pentru producători, dar și pentru programatori, devine acum conceperea, designul și construirea unor astfel de arhitecturi, pentru a putea utiliza puterea de calcul distribuită pe mai multe procesoare sau core-uri pentru a lucra împreună și a rezolva același task. Evident, scopul este acela de a maximiza gradul de paralelism atât pentru a maximiza gradul de utilizare a puterii de calcul disponibile, cât și de a minimiza timpul de rulare al calculelor, al aplicațiilor.

Răspândirea arhitecturilor multiprocesor are, indiscutabil, un efect profund și asupra programării aplicațiilor și sistemelor de operare destinate să lucreze cu un astfel de hardware. Până nu demult, evoluția tehnologiei însemna aproape întotdeauna creșterea frecvențelor de lucru ale procesoarelor și, ca efect, aplicațiile deveneau automat mai rapide, fără intervenția programatorilor. Acum, însă, creșterea performanței unei aplicații trebuie făcută având în vedere creșterea gradului de paralelism hardware, chiar fără o creștere a frecvenței. Exploatarea unui hardware orientat spre paralelism se poate dovedi o sarcină dificilă pentru programarea multiprocesoarelor.

Multiprocesoarele, în general, comunică printr-o memorie comună, numindu-se astfel multiprocesoare cu memorie partajată. Dificultătile în programarea lor apar la o multitudine de niveluri: la scară mică, multiprocesoarele trebuie să coordoneze între ele accesul la memoria partajată, în timp ce, la o scară mai mare, un supercalculator trebuie să coordoneze rutarea datelor. Programarea multiprocesoarelor este dificilă și din cauza faptului că sistemele de calculatoare moderne sunt, inerent, asincrone: activitățile pot fi întrerupte, întârziate, anulate în orice moment și fără a putea ști în prealabil evoluția unor astfel de evenimente, din cauza cererilor de întreruperi (IRQ), sistemelor de operare preemptive, accese nevalide la memoria cache, erori sau alte evenimente. Aceste întârzieri nu pot fi prevăzute cu acuratețe, iar scara la care efectele acestora se pot întinde poate varia de la un cache miss, care poate întârzia execuția unui program cu doar câteva zeci de instrucțiuni, până în momentul în care se realizează sincronizarea, la un page fault care poate induce o întârziere de câteva milioane de instrucțiuni, până când se aduce pagina respectivă din memoria virtuală mapată pe disc, iar un sistem de operare preemptiv poate intârzia execuția aproape oricât de mult, ajungându-se la miliarde de instrucțiuni, dacă sistemul de operare a decis să acorde timp de procesor unui alt proces.

Sisteme de operare

Scopul unui sistem de operare orientat spre arhitecturi multiprocesor este maximizarea paralelismului, pentru a putea face uz cât mai bine de întreaga putere de calcul disponibilă. Câteva dintre aspectele generale de care sistemele de operare trebuie să țină cont în vederea optimizării paralelismului sunt:

- Variabile definite pe fiecare CPU în parte, pentru a evita sincronizarea între aceastea
- Variabile atomice, al căror acces se face non-blocant
- Scrieri în memorie de tip write-through
- Blocarea resurselor în funcție de numărul de accese și tipul lor (citiri sau scrieri)
- Semafoare
- Mutex-uri
- Dezactivarea preferentiala a intreruperilor

Într-un sistem multiprocesor, două sau mai multe CPU-uri au acces partajat la aceeași memorie principală. De asemenea, de regulă, fiecare astfel de procesor deține și un cache propriu, astfel apărând problema sincronizării cache-urilor față de memoria partajată și, în final, a cache-urilor între ele.

- Scrierile efectuate de unul dintre CPU-uri trebuie să fie vizibile pentru toate celelalte CPU-uri
- Scrierile succesive în aceleași locații de memorie trebuie să fie văzute în aceeași ordine de către toate CPU-urile (serializarea scrierii)
- Bus snooping cu invalidarea cache-ului, o tehnică folosită în sistemele multiprocesor bazate pe memorie distribuită partajată, pentru obținerea corenței cache-urilor; fiecare controller de cache analizează magistrala pentru a detecta mesajele de difuzare (broadcast-uri) ce privesc cache-urile celorlalte procesoare, putând obține astfel informații care i-ar putea invalida anumite linii din cache-ul propriu.

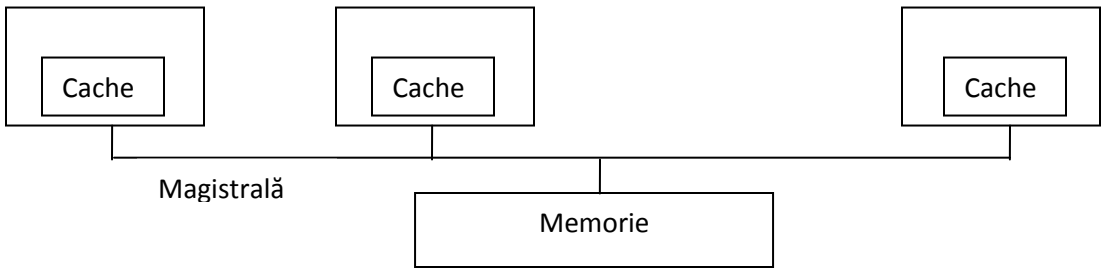


Fig 1: Schema fundamentală a unui sistem multiprocesor

Observație: pentru ușurința înțelegerii și comoditatea abstractizării conceptelor prezentate în continuare, exemplele următoare vor fi reprezentate în Java.

Spațiul registrelor

La nivelul hardware-ului, thread-urile unui sistem de operare comunică între ele prin scrierea într-o memorie partajată. Practic, abstractizând conceptele hardware, comunicarea între aceste fire de execuție se face prin niște obiecte ce permit accesul concurent, partajate între entități (thread-uri).

Un registru de tip citire-scriere (sau, în general, un registru pur și simplu) reprezintă un obiect de tip container care încapsulează o valoare accesibilă printr-un apel de tip read() și care poate fi modificată printr-un apel al unei metode write(). În realitate, de cele mai multe ori aceste tipuri de apeluri sunt denumite load și store. Codul de mai jos ilustrează o interfață implementată de toți acești regiștri:

```
1    public interface Register<T> {
2      T read();
3      void write(T v);
4    }
```

Fig 2: Definirea unei interfețe pentru un registru

Tipul T al valorii stocate în registru este, în mod normal, fie Integer sau Boolean sau o referință spre un alt obiect. Un registru care implementează interfața Register<Boolean> se numește un registru boolean. Un registru care implementează interfața Register<Integer> pentru un interval de M valori se numește un registru de valoare M. Restul registrelor, cele care stochează referințe, pot fi tratate ca și tipurile anterioare, prin considerarea valorii de referință ca pe un întreg.

În cazul ideal în care apelurile metodelor nu se suprapun, o implementare a unui astfel de registru ar trebui să funcționeze așa cum este descrisă mai jos:

```
1    public class SequentialRegister<T> implements Register<T> {
2      private T value;
3      public T read() {
4        return value;
5      }
6      public void write(T v) {
7        value = v;
8      }
9    }
```

Fig 3: Implementarea unei clase de tip registru secvențial

Totuși, pe un sistem multiprocesor, e de așteptat ca apelurile să se suprapună, deci trebuie avute în vedere funcționarea apelurilor concurente și modul lor de tratare. Una dintre abordările disponibile este folosirea excluderii mutuale prin alocarea unui mutex fiecărui registru, în urma apelurilor metodelor de read() și write(). Totuși, pe lângă faptul că regiștrii sunt cei care sunt, de regulă folosiți în implementarea excluderilor mutuale, folosirea excluderii mutuale în cazul regiștrilor ar însemna că execuția programului depinde de planificatorul (scheduler-ul) sistemului de operare pentru a garanta faptul că execuția vreunui fir nu rămâne blocată definitiv într-o regiune critică.

O altă abordare ar fi cea în care se ține cont de faptul că implementarea obiectelor este de tip wait-free[1], astfel că fiecare apel al metodelor este terminat într-un număr finit de pași, indiferent de felul în care execuția lor este suprapusă cu pașii altor metode concurente. Condiția de wait-free este, într-

[1] O implementare wait-free a unui obiect de date înseamnă că acesta garantează că orice proces poate să ducă la bun sfârșit orice operație sau serie de instrucțiuni într-un număr finit de pași, indiferent de viteza de execuție a altor procese.

adevăr, simplă, dar consecințele sale pot fi majore. În mod particular, această condiție elimină orice excludere mutuală si garantează progresul independent al execuției, fără a depinde de sistemul de operare și de planificatorul său. În consecință, implementarea anterioară a registrelor trebuie să fie de tip wait-free[2].

Un registru atomic reprezintă o implementare liniarizabilă a clasei regisrtului secvențial descris în implementarea anterioară. În mod informal, un registru atomic funcționează exact așa cum e de așteptat: returnează "ultima" valoare scrisă. Un model în care thread-urile comunică prin utilizarea registrelor atomice este, de asemenea, mult mai intuitiv și a reprezentat multă vreme standardul pentru modelul calculelor paralele.

De asemenea, este important de avut în vedere numărul de entități care vor citi din și care vor scrie în acest registru. Bineînțeles, este mult mai ușor de conceput un model care suportă un singur scriitor și un singur cititor decât unul pentru mai mulți.

Pentru simplificare, notația folosită cuprinde următoarele acronime:
- SRSW = "single reader, single writer";
- SRMW = "single reader, multiple writers";
- MRSW = "multiple readers, single writer";
- MRMW = "multiple readers, multiple writers";

De notat faptul că orice implementare a comunicării între threaduri trebuie să fie persistentă și consistentă, să nu depindă de participarea activă a niciunuia dintre cei ce participă la sincronizare. Cu alte cuvinte, operațiile de sincronizare și comunicare nu trebuie să se supună acelorași rigori ca și entitățile cărora aceste operații le facilitează comunicarea. Spre exemplu, cea mai simplă formă de sincronizare persistentă poate fi considerată posibilitatea de a stoca un singur bit în memoria partajată și cea mai simplă formă de sincronizare simplă este chiar lipsa oricărei forme de sincronizare. Dacă operația de a citi un bit nu se poate suprapune cu operația care scrie acel bit, atunci valoarea citită este în mod sigur cea scrisă. Altfel, însă, simultaneitatea acestor operații poate returna orice valoare a bitului, la citirea sa.

Diferite tipuri de registre vin cu diferite tipuri de "garanții" care le fac mai mult sau mai puțin puternice. Spre exemplu, s-a arătat ca registrele pot să difere prin intervalul de valori pe care le encapsulează (de tip Boolean sau de valoare M), dar și prin numărul de cititori și scriitori pe care îi suportă. În fine, ei pot să difere prin gradul de consistență pe care îl oferă.

O implementare a unui registru single writer, multiple readers se consideră sigură ("safe") dacă:
- Un apel read() care nu se suprapune cu un apel de tip write() returnează valoarea scrisă de cel mai recent apel write();
- Altfel, dacă un apel read() se suprapune cu un apel write(), atunci apelul lui read() poate returna oricare dintre valorile cuprinse în marja de valori suportate de acel registru (spre exemplu, de la 0 la M-1 pentru un registru de valoare M).

Este evident, în acest caz că termenul de "sigur" ("safe") nu cuprinde și sensul său propriu pentru că registrul, oferă, de fapt, garanții foarte slabe, deci registrele "sigure" sunt, de fapt, complet nesigure.

[2] O implementare wait-free este, automat și lock-free.

Dacă se consideră evoluția datelor prezentată în figura 4, de mai jos, atunci cele trei apeluri se pot comporta în felul următor:

- R^1 returnează 0, cea mai recent scrisă valoare;
- R^2 și R^3 sunt concurente cu W(1), deci pot returna orice valoare din marja de valori a registrului.

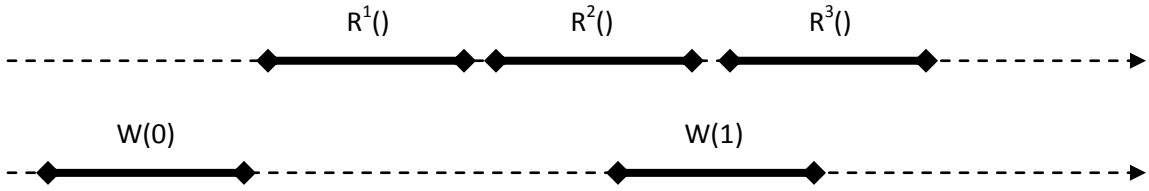


Figura 4

Figura de mai sus prezintă execuția unui registru de tip single-reader single-writer. R^i este citirea cu numărul i a registrului iar W(v) reprezintă o scriere în registru a valorii v. Axa timpului este de la stânga spre dreapta. Indiferent de tipul acestuia, dacă este "safe", normal sau atomic, R^1 trebuie să returneze 0 cea mai recent scrisă valoare. Dacă registrul este normal, atunci atât R^2 cât și R^3 pot returna atât 0 cât și 1. Dacă registrul este atomic, atunci , dacă R^2 returnează 1 și R^3 returnează 1 și dacă R^2 returnează 0, atunci R^3 poate returna 0 sau 1.

Un registru "normal" ca cel menționat mai sus este considerat ca un nivel intermediar de consistență între un registru "safe" si unul atomic. Un registru normal este un registru care suportă mai mulți cititori simultan și un singur scriitor, ale cărui operații de scriere nu au loc automat. De fapt, pe măsură ce apelul lui write() se execută, valoarea citită poate oscila între cea veche și cea nouă înainte de a fi înlocuită definitiv cu cea finală. Cu alte cuvinte:

- Un registru normal este totodată și "safe", în sensul că orice apel read() care nu se suprapune cu un apel write() trebuie să întoarcă întotdeauna ultima valoare scrisă.
- Presupunând că un apel read() se suprapune cu unul sau mai multe apeluri write(), dacă v^0 este ultima valoare scrisă de write()-ul precedent și $v^1,..,v^k$ este secvența valorilor scrise de write()-urile suprapuse, atunci apelul read() poate returna orice valoare v^i, pentru i în intervalul 0..k.

Pentru execuția din figura 4, un registru cu un singur cititor și un singur scriitor se poate comporta în felul următor:

- R^1 returnează vechea valoare, 0.
- R^2 și R^3, ambele returnează fie vechea valoare, 0, fie noua valoare, 1.

Pentru execuția din figura 4, un registru atomic cu un singur cititor și un singur scriitor poate returna următoarele:

- R^1 returnează vechea valoare, adică 0.
- Dacă R^2 returnează 1, atunci și R^3 returnează 1.
- Dacă R^2 returnează 0, atunci R^3 ar putea returna 0 sau 1.

Figura 5 reprezintă o vedere schematică a intervalelor posibile pentru tipurile regiștrilor ca un spatiu tridimensional: mărimea registrului definește una dintre dimensiuni, numărul de cititori și de scriitori definește o alta, iar propietatea de consistență a registrului o definește pe a treia. Bineînțeles, reprezentarea nu trebuie luată ca atare, din moment ce pot exista și registre "safe" ce permit mai mulți scriitori, a căror utilitate este aproape nulă.

Precedența unui obiect reprezintă o secvență de apeluri și răspunsuri, în care un eveniment de tip apel are loc atunci când un thread apelează o metodă, iar răspunsul aferent este evenimentul ce are loc atunci când apelul întoarce un rezultat (sau, mai transparent "se termină"). Apelul unei metode reprezintă intervalul dintre apelul efectuat de thread și răspunsul corespunzător. Orice precedență a unor astfel de evenimente definește și o relație de implicare, în sensul că, dacă m_0 și m_1 sunt apeluri de metode, putem spune ca m_0 implică m_1 ($m_0 \rightarrow m_1$) dacă evenimentul de răspuns al lui m_0 precede apelul lui m_1.

Orice implementare a registrelor ("safe", normală sau atomică) definește o relație de ordine totală asupra apelurilor write(), ordinea în care fiecare apel write() este "considerat" de către registru. Pentru registrele "safe" și normale, ordinea de scriere este pur și simplu trivială, pentru că ele permit doar un singur scriitor la un moment dat. Pentru registrele atomice, apelurile prezintă o ordine cu scopul liniarizării lor. De notat că pentru registrele de tip SRSW și MRSW ordinea de scrieri este aceeași cu precedența apelurilor.

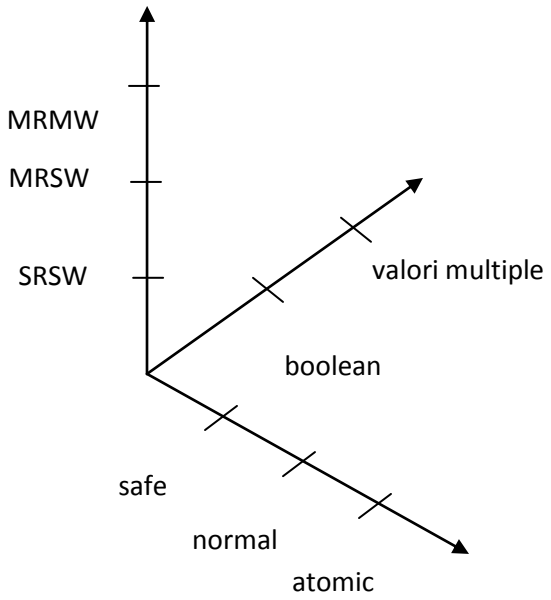


Figura 5

Dacă definim notația R^i pentru a desemna orice apel de citire care returnează valoarea v^i, valoarea unică scrisă prin W^i. Precedența serializată poate conține doar un singur apel W^i, dar poate conține mai multe apeluri R^i. În acest fel, se poate defini natura registrelor normale. În primul rând, niciun apel de citire nu poate returna o valoare din viitor, deci este imposibil ca:

$$R^i \rightarrow W^i$$

În al doilea rând, niciun apel de citire nu poate returna o valoare din trecutul distant, adică orice valoare care precede cea mai recentă valoare scrisă și nesuprapusă, deci e imposibil ca:

$$W^i \rightarrow W^j \rightarrow R^i$$

Tot în acest, context, un registru atomic satisface, în plus, și următoare condiție:

$$\text{dacă } R^i \rightarrow R^j \text{ atunci } i \leq j$$

Expresia arată faptul că un apel de read() nu poate returna o valoare ce urmează a fi returnată de un alt apel read(), din viitor.

Implementarea registrelor

Registre "safe" de tip MRSW

Figura 6 arată construcția unui registru "safe" MRSW dintr-un registrul "safe" SRSW.
Dacă apelurile de citire ale lui A nu se suprapun cu apelurile write(), atunci apelul lui read() returnează valoarea s_table[A], care este cea mai recent scrisă valoare. Pentru apelurile concurente, read()-ul poate returna orice valoare, deoarece implementarea este "safe".

```
1   public class SafeBooleanMRSWRegister implements Register<Boolean> {
2     boolean[] s_table;                          //vector de regiștri "safe" SRSW
3     public SafeBooleanMRSWRegister(int capacity) {
4       s_table = new boolean[capacity];
5     }
6     public Boolean read() {
7       return s_table[ThreadID.get()];
8     }
9     public void write(Boolean x) {
10      for (int i = 0; i < s_table.length; i++)
11        s_table[i] = x;
12    }
13  }
```

Figura 6

Registre normale boolene MRSW

Figura 7 arată construcția unui registru normal boolean de tip MRSW dintr-un registru "safe" boolean MRSW. Pentru registre boolene, singura diferență dintre registrele "safe" și normale apare atunci când noua valoare scrisă (x) este aceeași ca și precedenta. Un registru normal poate returna întotdeauna x, pe când un registru "safe" poate returna oricare dintre valorile boolene. Această excepție poate fi rezolvată prin simpla analiză și scriere a a valorii doar dacă valoarea scrisă este diferită de cea precedentă.

```
1   public class RegBooleanMRSWRegister implements Register<Boolean> {
2     ThreadLocal<Boolean> last;
3     boolean s_value; // safe MRSW register
4     RegBooleanMRSWRegister(int capacity) {
5       last = new ThreadLocal<Boolean>() {
6         protected Boolean initialValue() { return false; };
7       };
8     }
9     public void write(Boolean x) {
10      if (x != last.get()) {
```

```
11              last.set(x);
12              s_value =x;
13             }
14           }
15           public Boolean read() {
16             return s_value;
17             }
18         }
```

Figura 7

Orice apel de tip read() care nu se suprapune cu un apel write() returnează cea mai recent scrisă valoare. Dacă apelurile se suprapun, se iau în considerare două situații:

- Dacă valoarea scrisă este aceeași ca și valoarea scrisă anterior, scriitorul evită scrierea în registrul "safe" astfel asigurând cititorul că va obține valoarea corectă la citire.
- Dacă valoarea ce urmează a fi scrisă este diferită de ultima valoare scrisă, atunci ele trebuie să fie "true" și "false". O citire concurentă poate returna oricare dintre aceste valori.

Registre normale MRSV de valoare M

Cea mai simplă și totodată cea mai ineficientă reprezentare a unui registru de valoare M este reprezentarea unară, ca un vector de M registre boolene. Inițial, registrul este setat la valoarea zero, lucru indicat de bitul de pe poziția 0, care e setat la "true". O metodă care scrie valoarea x în registru, va scrie "true" la poziția x și "false" la toate pozițiile de index inferior. Evident, în acest caz, funcționarea unei metode de citire se bazează pe citirea liniară a valorilor boolene până la întâlnirea primei valori "true", returnând drept rezultat citit indexul acestei valori din vector.

Figura 8 arată o implementare pentru un astfel de registru de valoare 8:

```
1          public class RegMRSWRegister implements Register<Byte> {
2            private static int RANGE = Byte.MAX_VALUE - Byte.MIN_VALUE + 1;
3            boolean[] r_bit = new boolean[RANGE]; //registrul normal boolean MRSW
4            public RegMRSWRegister(int capacity) {
5              for (int i = 1; i < r_bit.length; i++)
6              r_bit[i] = false;
7              r_bit[0] = true;
8            }
9            public void write(Byte x) {
10             r_bit[x] = true;
11             for (int i = x - 1; i >= 0; i--)
12               r_bit[i] = false;
13           }
14           public Byte read() {
15             for (int i = 0; i < RANGE; i++)
16               if (r_bit[i]) {
17                 return i;
18               }
19             return -1; // caz exceptat și imposibil
20           }
21         }
```

Figura 8

În figura 9 este prezentat modul de executie al unui registru normal MRSW, de valoare 8. Valorile "true" și "false" sunt reprezentate prin 1, respectiv 0:

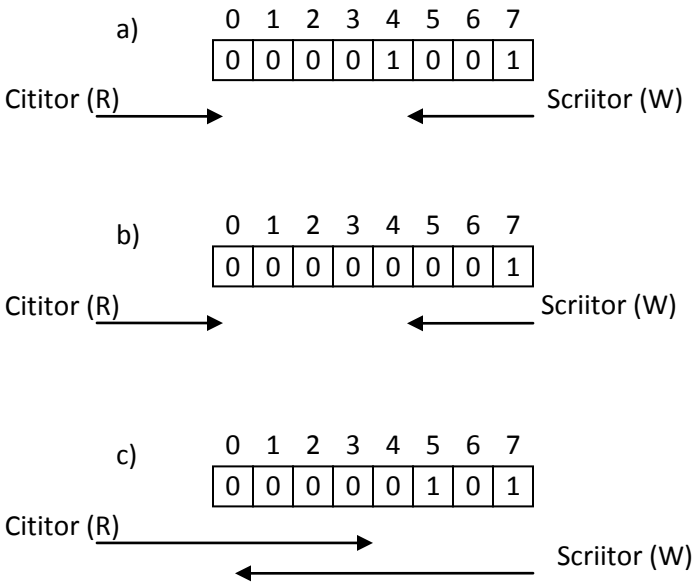


Figura 9

În partea (a), valoarea anterioară scrierii era 4, dar valoarea 7 a scrierii threadului W nu este citită de către threadul R, deoarece threadul R ajunge să citească poziția 4 a vectorului înainte ca threadul W să apuce să suprascrie valoarea "false" la acea locație.

În partea (b), valoarea 4 este suprascrisă de threadul W înainte ca acea locație să fie citită, deci citirea returnează corect valoarea 4.

În partea (c), threadul W începe să scrie valoarea 5. Din moment ce a scris valoarea 5 înainte ca aceasta sa fie citită, citirea returnează 5, deși valoarea 7 încă există, deoarece bitul să era setat pe "true".

Registre atomice SRSW

Conform condițiilor descrise anterior, excepția în cazul registrelor atomice apare în situația în care două citiri simultane cu o scriere returnează valori neordonate, prima returnând v^i și următoarea returnând v^j, unde $j < i$.

În figura 10 se descrie un mecanism bazat pe etichete de timp. Implementarea va folosi aceste etichete pentru a putea ordona corespunzător apelurile concurente de citire. Fiecare citire memorează cea mai recentă (cea mai mare valoare a etichetei de timp) citită pentru a putea fi folosită și de citirile următoare. În cazul în care o citire ulterioară primește o etichetă de timp cu o valoarea mai mică, aceasta va fi ignorată, și va fi folosită în continuare ultima valoare citită. În mod similar, thread-urile de scriere vor memora ultima valoare a etichetei de timp folosite la scriere și vor eticheta scrierile următoare cu valori consecutive (valori mai mari decât 1).

Algoritmul prezentat mai sus presupune posibilitatea de a scrie și de a citi valori ale etichetei de timp ca unități indivizibile. În C, spre exemplu, aceste informații pot fi interpretate ca simpli biți, fără un tip anume de date, folosind operații de shiftare asupra lor.

```
1       public class StampedValue<T> {
2         public long stamp;
3         public T value;
4         // valoarea inițială, cu eticheta de timp zero
5         public StampedValue(T init) {
6           stamp = 0;
7           value = init;
8         }
9         //valorile ulterioare alte etichetelor de timp
10      public StampedValue(long stamp, T value) {
11        stamp = stamp;
12        value = value;
13      }
14      public static StampedValue max(StampedValue x, StampedValue y) {
15        if (x.stamp > y.stamp) {
16          return x;
17        } else {
18          return y;
19        }
20      }
21      public static StampedValue MIN_VALUE = new StampedValue(null);
22      }
```

Figura 10

Bibliografie:

- Cursul de Sisteme Multiprocesor al departamentului Computer Science al Universității din Rochester
- The Art of Multiprocessor Programming – Maurice Herlihy, Nir Shavit – Morgan Kaufman Publishers, 2008