INFORMATICA 1/1982

PREGLED JEZIKOVNIH ELEMENTOV ZA OPIS SINHRONIZACIJE PARALELNIH PROCESOV

MONIKA KAPUS

UDK: 681.3.012

INŠTITUT „JOŽEF STEFAN", LJUBLJANA

Članek podaja pregled razvoja jezikovnih elementov za opis sinhronizacije paralelni procesov. Povdarek je na predstavitvi logičnih osnov paralelnega procesiranja in na opazovanju pojava, da postajajo jeziki za multiprogramiranje vedno bolj homogeni.

The paper is an overview of the development of synchronization primitives. It presents the logical basis of parallel processing and describes the homogenization of multiprogramming languages.

0. Uvod

V začetku je bil osnovni namen paralelnega procesiranja ekonomično izkoriščanje hardvara v skladu s pravilom, da naj proces zaseda nek računalniški vir le v tistih časovnih intervalih, ko ga zares uporablja. Operacijski sistem je moral delovati tako, da so procesi čim manj čutili prisotnost drugih procesov v sistemu. Zato dela programske kode, ki sta se izvajala paralelno, nista smela biti logično povezana.

Najpreprostejša povezava med procesi se je pojavila takrat, ko so se programerji začeli zavedati, da lahko programe, ki so se prej izvajali kot en proces, na nekaterih odsekih razdelijo na paralelne procese in s tem skrajšajo čas obdelave. Taki paralelni procesi so bili logično povezani v tem smislu, da se je izvajanje programa lahko nadaljevalo šele takrat, ko so biti znani rezultati vseh paralelnih procesov, ki so se spet združili v en proces. Pravzaprav ni šlo za medsebojno povezavo paralelnih procesov, temveč za povezavo paralelnih procesov s svojim očetom, ki je med njihovim izvajanjem miroval.

Z razvojem velikih računalniških sistemov se je pogled na paralelno procesiranje spremenil. Predvsem se je večina logičnega načrtovanja prenesla s hardwarskega na programski nivo. Še več! Po splošni sistemski teoriji sta dva sistema ekvivalentna, če dajeta pri enakih vhodnih signalih enak odziv. Zato se je na programski nivo preneslo tudi funkcionalistično pojmovanje sistema. Uveljavilo se je pravilo: ena funkcija-en sistemski blok. Sistemski blok v računalništvu je navidezni (logični) stroj, ki je izveden hardwarsko, programsko ali mešano. Programsko izveden logični stroj predstavlja v sistemu poseben proces, ki je stalno prisoten, tako kot bi bil hardwarsko izveden logični stroj, ali pa ga poženemo takrat, ko ga potrebujemo. Razbitje naloge na funkcijske bloke je posebno primerno za pisanje obsežnih programov, kakršen je operacijski sistem.

1. Logična odvisnost paralelnih procesov

Funkcionalistična opredelitev procesa je najbolj naravna, saj je proces deklariran kot logični objekt-nosilec aktivnosti. Zaradi takega načina dela pa procesi niso več nujno neodvisni. Dva procesa sta medsebojno logično odvisna, če uporabljata skupne računalniške vire, ki so lahko pasivni ali aktivni predmeti, tako da se morata tega zavedati. Ker je vse dogajanje v računalniku v bistvu branje in spreminjanje podatkov, so procesi soodvisni natanko takrat, kadar izvajajo operacije na istih podatkovnih predmetih.

Podatkovne predmete, ki jih obravnava proces, delimo v tri skupine: vhodne spremenljivke, izhodne spremenljivke in spremenljivke stanja. Kot pri vsakem sistemu smemo meje procesa premikati, pri čemer zajamemo med spremenljivke stanja več ali manj njegovih operandov. Ker pa želimo obravnavati proces kot logični stroj, uvrstimo med spremenljivke stanja le tiste spremenljivke, do katerih nima neposrednega dostopa noben drug proces. Vhodne in izhodne spremenljivke procesa so v splošnem tudi vhodne ali/in izhodne spremenljivke drugih procesov.

2. Sinhronizacija procesov

Za pravilno delovanje sistema velja naslednje pravilo:

Ko eden od procesov nastavi vrednost neke spremenljivke, ki naj bi jo brala skupina procesov-čitalcev (lahko tudi proces, ki je vrednost vpisal), smejo vsi procesi to vrednost poljubno spreminjati pod pogojem, da bo imela spremenljivka pravilno vrednost v vseh tistih trenutkih, ko jo bo bral kak proces iz skupine čitalcev.

Vidimo, da imajo procesi skoraj popolno svobodo. Na spremenljivkah smejo izvajati poljubne operacije, ki niso prepovedane s statičnim zaščitnim mehanizmom podatkovnih predmetov, to je s pravili, kateri aktivni predmeti imajo pravico dostopa do nekega pasivnega predmeta in kakšen sme biti ta dostop. Od procesa zahtevamo le to, da spremenljivke po potrebi vrača na stare vrednosti. Omenimo še, da zgodovine spremenljivk ne predstavlja v vsakem primeru ena sama vrednost. Spremenljivka ima navidezno toliko vrednosti, kolikor je procesov, ki so vanjo vpisali neko vrednost z namenom, da ostane konstantna. Lahko bi

ugovarjali, da je tako razmišljanje nepotrebno, saj bi lahko dali vsakemu procesu svojo spremenljivko. Vendar vseh spremenljivk ne moremo enostavno razmnožiti. To velja predvsem za aktivne spominske celice, kakršne so registri. Organizacija sistema, v katerem obstajajo spremenljivke z več navideznimi vrednostmi, je izredno težka, posebno, kadar imamo enakopravne procese.

Problem sinhronizacije procesov zapletajo časovni pogoji. V preprostih sistemih zadošča, da uskladimo hitrosti posameznih procesov kar z vstavljanjem ukazov, ki porabijo določeno število urinih ciklov. Taka sinhronizacija procesov je statična in se ne zna prilagajati nepredvidenim motnjam v sistemu. Splošna rešitev je sinhronizacija procesov z medprocesno komunikacijo, ki je lahko neposredna preko skupnih spremenljivk ali pa posredna preko pomožnega procesa. Pri sinhronizaciji preko skupnih spremenljivk proces, ki prvi doseže sinhronizacijsko točko, v zanki testira sinhronizacijsko spremenljivko, dokler mu drug proces ne pošlje statičnega signala. Komunikacija preko pomožnega procesa je lahko taka, da partnerja pač izvajata operacije nad sinhronizacijsko spremenljivko s posredovanjem pomožnega procesa. Najpogosteje pa je posrednik sistemski proces, ki mu hitrejši proces naroči, naj ga požene, ko bo partner prišel do sinhronizacijske točke. Proces čaka pasivno, kar je zelo ugodno, ker tudi golo testiranje troši računalniški čas. Sporočilo počasnega procesa posredniku, da je dosegel sinhronizacijsko točko, predstavlja kratkotrajen signal, ki požene čakajoči proces, če seveda obstaja. Če je čakanje simetrično, torej če čaka vedno tisti proces, ki je prvi prišel na sinhronizacijsko točko, partner pa vedno pošlje signal, se bosta procesa vedno uspešno srečala (princip rendez-vous-a). Če pa vnaprej predpišemo, kdo pošilja signale in kdo jih čaka, je nevarno, da se kak signal izgubi, kar je usodno, če proces, ki je poslal signal, kar nadaljuje izvajanje. V tem primeru se procesa ne srečata. Isto velja, če proces dinamično signalizira poljuben dogodek, na katerega čaka partner. Pogoji za delo so lahko popolnoma izpolnjeni, pa proces tega ne opazi.

2.1. Problem vzajemnega izključevanja

Osnovni problem sinhronizacije je dostop več procesov do skupnega podatkovnega predmeta. Dovoljeno je paralelno branje, ni pa dovoljeno paralelno pisanje ali paralelno branje in pisanje. Dele programske kode, ki naslavljajo obravnavani predmet, imenujemo z ozirom na ta predmet kritične dele programa. Recimo, da želimo zagotoviti, da bo v vsakem trenutku izvajal svoj kritični del največ en proces. Nalogo imenujemo problem vzajemnega izključevanja. Od rešitve zahtevamo še naslednje lastnosti:

- -simetrijo (procesi ne smejo imeti statične prioritete);
- -neodvisnost od relativnih hitrosti procesov;
- -neobčutljivost na pojav, da se kak proces ustavi izven kritičnega dela;
- -kadar več procesov hkrati zahteva vstop v kritični del, se mora algoritem odločiti za enega od njih v končnem času.

Problem vzajemnega izključevanja ni direktna posplošitev vsakega sinhronizacijskega problema. V nekaterih pogledih so zahteve prestroge (npr. ne dovolimo paralelnega branja), v nekaterih preblage (npr. ne predpisujemo, da moramo vrednost podatka najprej vpisati, preden jo lahko beremo). Algoritem vzajemnega izključevanja je samo orodje, s katerim lahko rešimo katerikoli sinhronizacijski problem. Z vpeljavo dodatnih spremenljivk za medprocesno sinhronizacijo brez težav omejimo vzajemno izključevanje na tiste dele programov, kjer je res potrebno.

Problem vzajemnega izključevanja je prvi rešil danski matematik J.Dekker. Zapišimo algoritem za vzajemno izključevanja poljubnega števila procesov, ki ga je predstavil Dijkstra v [3] v Algolu 60:

```
begin integer array b,c [0:N];
      integer turn;
      for turn := 0 step 1 until N do
          begin b[turn] := 1; c[turn] := 1
end;
      turn := 0;
      parbegin
       process 1: begin.......end;
       process 2: begin.......end;
       .
       .
       process N: begin.......end
      parbegin
     end.

process i: begin integer j;
           Ai: b[i] := 0;
           Li: if turn <> i then
               begin c[i] := 1;
                     if b[turn] = 1
                     then turn := i;
                     goto Li
               end;
               c[i] := 0;
               for j := 1 step 1 until N do
                   begin if j <> i and
                                c[j] = 0
                         then goto Li end;
               kritično področje;
               turn := 0; c[i] := 1;
               b[i] := 1;
               ostanek cikla i; goto Ai
           end.
```

Namesto dokaza, da algoritem zadošča vsem zahtevam, opišimo njegovo delovanje v naravnem jeziku. Zanimivo je, da v algoritmu, ki omogoča medsebojno izključevanje procesov, vsi procesi prosto uporabljajo skupne spremenljivke. Pomisliti pa moramo, da je digitalni računalnik sekvenčni stroj, v katerem se ukazi za pisanje in čitanje po naravi izvajajo zaporedno. Če zahtevamo vpis dveh različnih vrednost a in b v isto spremenljivko, bo vrednost spremenljivke v vsakem trenutku ali a ali b, nikakor pa ne mešanica obeh vrednosti.

Konkurenčni procesi sklepajo na naslednji način:

Označim, da želim vstopiti v kritično področje. Preverim, če sem navrsti. Če nisem, pogledam, ali je tisti, ki je navrsti, končal delo. Če ne dela več, označim, da sem navrsti jaz. Preverim, ali ni morda kak drug proces tudi ugotovil, da prejšnji proces ne dela več in se je poskusil postaviti na začetek vrste. Če me je izrinil, začnem postopek za preboj na čelo vrste od začetka z vljudnim čakanjem, da proces na čelu konča delo. Ko sem na čelu vrste, označim, da vstopam v kritično področje. Spet preverim, ali morda kdo ne dela. To je čisto mogoče. Med trenutkom, ko sem ugotovil, da se imam pravico postaviti na čelo vrste, in med mojim vpisom na čelo vrste je minilo nekaj časa. Medtem je lahko kak posebno hiter proces, ki se prej še ni zanimal za kritično področje, prišel in začel z delom. Če je to res, se mu umaknem in ponovim postopek za preboj na začetek vrste, sicer pa vstopim v kritično področje. Po izstopu označim, da me ktiriično

področje ne zanima več in ustvarim možnosti za enakopraven konkurenčni boj vseh procesov.

2.2. Nedeljivost vzroka in posledice. Semaforji

Iz tega algoritma smo se naučili, da je vir težav naslednje preprosto dejstvo: Proces dobi informacijo o stanju drugih procesov in se odloči, kako bo ukrepal. Toda ko začne ukrep izvajati, nima zagotovila, da stara informacija sploh še velja. Morda je dobil signal že novo vrednost. Problem je v preveliki dinamiki procesov, ki pošiljajo signale.

Vsak signal mora izpolnjevati naslednjo zahtevo:

Signal naj traja dovolj dolgo, da ga procesi, ki jim je namenjen, opazijo in preberejo, kolikorkrat želijo, stanje, ki ga opisuje signal, pa mora trajati, dokler niso uresničeni vsi ukrepi, ki so posledice tega stanja in ne bi več ustrezali, če bi se stanje spremenilo.

To pravilo nedeljivosti vzroka in posledice upošteva Dijkstrova definicija binarnega semaforja. Ker pravilo v celoti zajema bistvo sinhronizacije, lahko rešimo z binarnimi semaforji katerikoli sinhronizacijski problem.

Binarni semafor je poseben primer splošnega semaforja, ki je nenegativna cela spremenljivka, binarni semafor pa lahko zavzame vrednosti 0 in 1. Semafor je statični signal, ki drži vrednost, dokler je eksplicitno ne spremenimo. Zato ni nevarnosti, da procesi ne bi opazili signala. Princip splošnega semaforja je redundanten, ker se da realizirati z binarnim semaforjem in navadno celo spremenljivko. Uvedli so ga zato, ker na eleganten način rešuje problem dodeljevanja enakih virov skupini procesov. Nad semaforjem sta definirani operaciji P in V. Procesi, ki čakajo, mehanizem semaforja uvršča v implicitno vrsto. Čakanje je po definiciji pasivno. Semafor mora zagotoviti enakopravnost procesov in izključiti možnost, da bi kak proces (po semaforjevi krivdi) čakal neskončno dolgo. Iz zadnje zahteve sledi, da nima prednosti proces na začetku vrste, ampak tisti prvi proces v vrsti, pri katerem so izpoljnjeni pogoji za izvedbo operacije. Pri operaciji V navidezno ni čakanja, dejansko pa se tudi na njej odraža sekvenčna narava računalnika. Zapišimo definicijo splošnega semaforja v jeziku Alphard:

```
s: semaphore (začetne vrednost);
   comment semafor je nenegativna cela
           spremenljivka za posebne
           namene, ki ji je pridružena
           implicitna procesna vrsta;
procedure V (s: semaphore) =
begin [s:= s+1;
       če je v vrsti proces,
       ki izpolnjuje potrebne pogoje,
       ga zbudi] end;
procedure P (s: semaphore) =
begin [if s>0
       then s:= s+1
       else begin
            vključi proces v vrsto];
            wait; P(s)
       end
end.
```

Operacije v oklepajih so nedeljive.

Definicija semaforja ne pove nič o njegovi izvedbi. Tudi na področju sinhronizacije procesov težimo k ločitivi deklarativnega in proceduralnega obravnavanja problema. Če zapišemo obsežno proceduro za sinhronizacijo, sicer lahko pravilno deluje, vendar njej namen ni jasno viden in preverjanje pravilnosti je težko. Zato vključujemo ukaze za sinhronizacijo v visoke programske jezike, v katerih opisujemo, kakšen naj bo odnos med procesi in podatkovnimi predmeti, gradnjo potrebnih procedur pa prepustimo prevajalniku.

2.3. Razvoj jezikovnih elementov za sinhronizacijo

Pred definicijo semaforjev je moral programer vedno znova pisati procedure, kakršen je Dekkerjev algoritem. Ukazi, ki bi združevali testiranje in spreminjanje podatkov, praktično niso obstajali. Semafor pomeni prvi podatkovni predmet, ki ni navadna spremenljivka in je namenjena izključno sinhronizaciji.

Po uvedbi semaforjev so si avtorji najprej prizadevali izboljšati sam koncept semaforjev. Operacije na semaforjih so testiranje različnosti od 0, zmanjšanje in povečanje vrednosti. Wodon je leta 1972 namesto P in V operacij uvedel operaciji up in down. Dovolil je kombiniranje pogojev in operacij na več semaforjih npr. Sa.Sb.Sc pomeni, da naj se operacije izvršijo, če so semaforji a,b,c večji od -1 (druga varianta pogoja >0); down Sa,Sq pomeni zmanjšanje, up Sa,Sq pa povečanje semaforjev a in q. Na ta način lahko realiziramo bolj kompleksne nedeljive pare pogojev in posledic (npr. Sa.Sb.: down Sa,Sb). Dijkstrov semafor je le poseben primer (s: down s = P(s); up s = V(s)). Izboljševanje P in V ukazov se je končalo s predlogi bolj sestavljenih pogojev za izvedbo operacij (AND/OR izrazi) in z razmišljanjem o organizaciji implicitne vrste.

Z razvojem teorije programskih jezikov se je pogled na ukaze za sinhronizacijo spreminjal. Prodrlo je spoznanje, da moramo ponuditi programerju taka jezikovna sredstva, ki ga bodo spodbujala k strukturiranemu programiranju. Hoare in Brinch Hansen sta leta 1972 definirala "kritično področje". Novost je v tem, da prevajalniku deklariramo, kateri podatkovni premeti so skupni. Kadar želimo delati s skupnimi podatkovnimi predmeti, to eksplicitno izrazimo in sistem sam poskrbi za vzajemno izključevanje.

```
var v : shared;
region v do operacija na v od
```

Prevajalnik pomaga programerju na ta način, da javi napako, če naslovimo skupne podatke izven kritičnega področja.

Še bolj strukturirana so pogojna kritična področja (prav tako Hoare in Brinch Hansen), ki so naravna razširitev pojma kritičnega področja in vključujejo še čakanje procesa na dogodek.

```
var v shared;
region v when B do operacija na v od
```

Ko proces vstopi v kritično področje, se testira pogoj B. Če je pogoj izpolnjen, se zahtevana operacija izvede, sicer pa proces začasno zapusti kritično področje in se postavi v procesno vrsto spremenljivke v. Vrsta je ena sama za vse procese, ne glede na to, na kakšen pogoj čakajo. Kadarkoli nek proces uspešno zaključi operacije v kritičnem področju, sistem testira pogoje, na katere čakajo procesi v vrsti, in požene enega od procesov, katerih pogoj je izpolnjen. Princip pogojev je ekvivalenten statičnem signalom.

Danes so jezikovni elementi za sinhronizacijo dosegli stopnjo, ko predstalja sinhronizacija le del opisa abstraktnega podatkovnega tipa in programerju o njej sploh ni treba razmišljati, kadar piše proceduro, v kateri naslavlja skupni podatkovni predmet. Abstraktni podatkovni tip združuje opis podatkovne strukture in operacij, ki so na njej definirane. Zunaj dela programa, kjer opišemo podatkovni tip, lahko opazujemo samo učinek definiranih operacij na predmete tega tipa, nikakor pa ne načina izvajanja teh operacij. Lastnosti abstraktnega podatkovnega tipa lahko preverjamo, ne da bi upoštevali uporabo posameznih predmetov tega tipa. Preverjati abstraktni podatkovni tip pomeni preverjati, ali izvedba operacij na podatkovni strukturi ustreza deklaraciji. Tako pojmovanje sinhronizacije omogoča hierarhično strukturiranje obsežnih programov. Iz operacij, ki smo jih definirali in preverili na nižjih nivojih, lahko gradimo operacije na višjih nivojih, ki jih testiramo le glede na deklaracije višjih nivojev.

Najenostavnejši primer podatkovnega tipa z vgrajeno sinhronizacijo so monitorji (Dijkstra, Brinch Hansen, Hoare). Monitor je konstrukt, ki vsebuje lokalne spremenljivke in procedure, ki definirajo operacije nad temi spremenljivkami. Monitorske spremenljivke lahko dosežemo le preko monitorskih procedur. Vsaka monitorska procedura sme imeti tudi svoje lokalne spremenljivke. Monitorske procedure ne smejo naslavljati spremenljivk zunaj monitorja. Vse monitorske procedure se vzajemno izključujejo, le v primeru, da kaka procedura čaka na signal, sme teči druga procedura do takrat, da prva procedura sprejme signal in spet pravzame kontrolo. Navadno monitorske procedure ne sprejemajo zunanjih signalov, ampak čakajo na signal tiste monitorske procedure, ki teče. Signali so dinamični, to pomeni, da se izgubijo, če jih nihče ne čaka. Hkrati sme čakati na signal ene vrste tudi več procedur, po signalu pa steče ena sama in sicer tista z najvišjo prioriteto ali z najdaljšim časom čakanja. Operacije wait so edine operacije, ki se smejo izvajati paralelno z drugimi operacijami. Monitorske procedure ne smejo klicati nestandardnih zunanjih procedur, še manj pa lokalne procedure drugih monitorjev.

Monitorji sicer združijo sinhronizacijo s podatki, na katere se nanaša, vendar mora programer še vedno pisati monitorske procedure s klasičnimi jezikovnimi sredstvi, iz katerih ni razvidno, kaj pravzaprav želi. Kadar uporabljajo procesi skupne spremenljivke, je vzajemno izključevanje pogosto samo tehnično vprašanje, ki le v primeru, da gre za golo dodeljevanje virov, predstavlja bistvo dogajanja. V vseh zahtevnejših aplikacijah je namen skupnih spremenljivk komunikacija med procesi, ki morajo brati in pisati vrednosti spremenljivk v določenem zaporedju. Programer potrebuje jezikovno sredstvo, s katerim bo ta zaporedja opisal. Rešitev sta podala Campbell in Habermann leta 1974 z uvedbo "opisov poti".

Opis poti je izraz, ki podaja vsa dovoljena zaporedja operacij na nekem podatkovnem predmetu. Preproste poti so ciklične, osnovni operatorji v izrazih pa so sekvenciranje, izbira in ponavljanje. Specifikacija podatkovnega tipa ima naslednjo obliko:

```
type ime =
begin
 opis podatkovne strukture
 path opis poti end
 operation opis možnih operacij
end
```

Izraz path f1 (g* + h)i i end na primer opisuje cikel, ki se obvezno začne z eno operacijo f, sledi poljubno število ponovitev operacije g ali ena operacija h, zaključi se z eno operacijo i. Vse operacije se vzajemno izključujejo. To lahko preprečimo z operatorjem paralelnosti ali z definicijo več poti na istem podatkovnem predmetu. Zanimiv element poti je tudi numerični element, ki ima obliko

path (f1-f2-...-fm)n end

No(fi) naj pomeni število ponovitev operacije fi. Z numeričnim elementom dosežemo, da velja:

No(f1)>=No(f2)>=....>=No(fm)>=No(f1)-n ali
D=<No(f1)-No(f2)=<...=<No(f1)-No(fm)=< n

Primer m=2 ilustrira pošiljanje sporočil preko vmesnika z dolžino n.

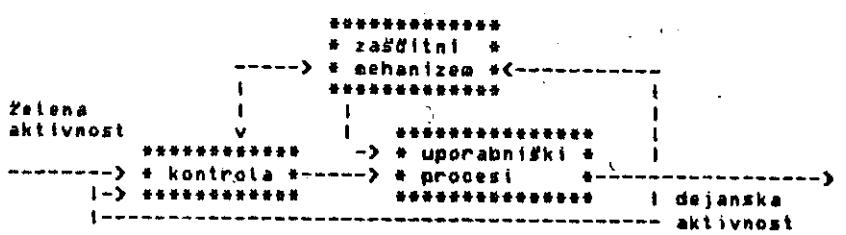
Razvoj jezikovnih elementov za opis sinhronizacije oz. komunikacije med procesi pa s tem še ni dosegel končne stopnje. Elementi naj bi imeli veliko izrazno moč, biti morajo razumljivi in enostavni za uporabo. Predvsem pa morajo biti taki, da jih bo programer uporabljal pravilno, da bo pravilnost brez težav dokazal in da ga bo sistem sam opozarjal na morebitne napake.

Opis sinhronizacijskih zahtev, ki je del opisa podatkovnega predmeta, predstavlja dinamično komponento zaščitnega mehanizma tega predmeta; zato smemo združiti opis sinhronizacijskih zahtev z opisom statičnega zaščitnega mehanizma, ki odgovarja na vprašanje, kateri procesi imajo pravico do neke stalne množice operacij nad danim podatkovnim predmetom.

Poskusimo definirati splošni zaščitni mehanizem podatkovnega predmeta!

Zaščitni mehanizem podatkovnega predmeta je funkcijski blok, ki deluje ravno obratno kot funkcijski bloki uporabniških procesov, ki jih zaščitni mehanizem nadzoruje.

1. Zaščitni mehanizem je proces, ki je aktiven, dokler obstaja podatkovni predmet, katerega ščiti. Uporabniški procesi so navadno vezani samo na obstoj predmeta, ki vsebuje njihovo programsko kodo, in na kontrolne informacije.



2.Uporabniški procesi so bloki, pri katerih kontrolne informacije sprožijo aktivnost, ki jo predpiše uporabnik. Zaščitni mehanizmi so bloki, pri katerih določena aktivnost uporabniških procesov sproži generiranje kontrolnih informacij. Torej ima računalniški sistem povratno zanko.

3.Uporabniški proces posredno ali neposredno izvaja tiste aktivnosti, v katerih (s stališča uporabnikov) nastopa kot subjekt. Zaščitni mehanizem nadzoruje oz. izvaja tiste aktivnosti, v katerih njegov podatkovni predmet nastopa kot objekt.

Zaščitni mehanizem mora predvidevati na svojem vhodu poljubno kombinacijo zahtev za izvedbo operacij nad podatki. V vsakem trenutku se lahko odloči in sproži izvajanje skupine zahtev, odloži skupino zahtev za določen čas ali skupino zahtev zavrne, s tem da generira kontrolne informacije. To pomeni, da smemo v zaščitnem mehanizmu definirati poljubno stopno paralelnosti izvajanja operacij nad podatkovnim predmetom. Zaščitni mehanizem ima tudi svoje spremenljivke stanja, v katerih hrani zgodovino operacij. Ker se vsaka aktivnost računalniškega sistema odraža na vrednosti spremenljivk, za opis odločitvene verige splošnega mehanizma za zaščito podatkov zadošča, da rečemo, da se mehanizem odloča na podlagi vrednosti vseh spremenljivk v sistemu. Pri tem pa moramo vedeti, da upošteva razen vrednosti spremenljivk, ki jih obravnavajo uporabniški procesi, tudi spremenljivke, ki povedo, kateri procesi obstajajo v sistemu, v kakšnem stanju so, kako teče realni čas in predvsem, kdo je sprožil zahtevo za izvajanje neke operacije. Identiteto procesa predstavlja njegovo lastno ime skupaj z imeni vseh prednikov.

Ves čas zanemarjamo važno dejstvo, da pomeni klicanje procedure za operacijo nad podatki komunikacijo med uporabniškim procesom in zaščitnim mehanizmom. Komunikacija poteka seveda preko skupne spremenljivke, ki jo tudi nadzoruje zaščitni mehanizem itd.. Vedno pridemo do komunikacije na nekem nižjem nivoju, za katero smemo reči, da je že zadovoljivo realizirana.

Morda se komu zdi trditev, da je zaščitni mehanizem proces, preozka. Vendar pri tem ne mislimo, da ima zaščitni mehanizem v sistemu nujno enak položaj kot uporabniški procesi. Proces nam pomeni le ime za spreminjanje skupine spremenljivk v odvisnosti od stopenjske spremenljivke - realnega oz. računalniškega časa. Od teh spremenljivk so le nekatere iz uporabniškega področja, torej deklarirane s programom. Zato se, kot smo že rekli, zaščitni mehanizem (vsaj v nekaterih jezikih) ne pojavlja v programu kot deklaracija procesa, temveč - kot del deklaracije pripadajoče podatkovne strukture. Ko pa prevajalnik jezikovni konstrukt realizira, dejstva, da je zaščitni mehanizem proces ali (v primeru, da definiramo paralelne operacije) celo družina procesov, ne moremo zanikati.

Delovanje zaščitnega mehanizma se kaže v dveh fazah:

1.Med prevajanjem programa kot zaščita, da ne zahtevamo operacij, ki nad zaščiteno podatkovno strukturo niso definirane; ali da eksplicitno ne zahtevamo nedefiniranih paralelnosti.

2.Med izvajanjem programa kot zakasnitev ali zavrnitev operacij, ki niso takoj izvedljive zaradi dinamike celotnega sistema, ki iz samega programa ni razvidna, ker prevajalnik normalno ne izvaja kompletnega vrednotenja časovnih razmer (kar bi bilo skoraj nemogoče ali neekonomično).

Če sme zaščitni mehanizem uporabljati le svoje lokalne in globalne spremenljivke, smo pri definiciji bolj kompleksnih operacij prisiljeni združevati več podatkovnih predmetov v enega. Opis dovoljenih zaporedij operacij postane zelo nepregleden, ker zaščitni mehanizem nadzoruje več kot eno funkcijo sistema ali pa se na vmesniku zaščitni mehanizem - uporabniški procesi pojavljajo operacije z različno stopnjo kompleksnosti. To ne pomeni, da prostor podatkovnih predmetov hierarhično širimo, ker predmeti, ki jih združimo, izginejo.

Boljše rezultate dobimo, če zaščitnemu mehanizmu dovolimo uporabo tudi tistih spremenljivk, ki jih ščitijo drugi zaščitni mehanizmi. Pridemo do tega, da se zaščitni mehanizem pojavlja kot enakovreden partner uporabniških procesov na vhodu podatkovnih predmetov. Edina razlika med zaščitnimi mehanizmi in uporabniškimi procesi je ta, da so zaščitni mehanizmi deklarirani kot pasivni procesi, ki tečejo samo takrat, kadar skušajo uporabniški procesi izvesti neko operacijo nad njihovim podatkovnim predmetom. V sistemu je dovoljeno dinamično generiranje procesov. Dovolimo še dinamično generiranje novih podatkovnih predmetov, pa ni več nobene ovire, da ne bi pasivnih in uporabniških procesov obravnavali kot enotno vrsto procesov v sistemu, le da imajo pasivni procesi posebno nalogo. Dovoljeno naj bo tudi spreminjanje in ukinjanje podatkovnih predmetov in njihovih zaščitnih mehanizmov.

Spreminjanje lastnosti prostora podatkovnih predmetov sme zahtevati vsak proces v sistemu, ne glede na to, ali je pasiven ali aktiven (uporabniški). Sistemski program sme izvesti zahtevo po spremembi le v primeru, da so izpolnjeni pogoju zaščite prostora podatkovnih predmetov. Proces, ki zahteva generiranje novega predmeta, mora navesti naslednje podatke:

-katere spremenljivke bo vključil novi podatkovni predmet. Vse spremenljivke morajo biti globalne, torej izven vseh obstoječih od uporabnika definiranih podatkovnih predmetov. Smisel ima tudi to, da predmet ne vsebuje nobenih lokalnih spremenljivk. V tem primeru ga generiramo samo zaradi implementacije bolj kompleksnih operacij nad spremenljivkami v drugih podatkovnih predmetih.

-katere globalne spremenljivke bo nastavljal zaščitni mehanizem novega predmeta;

-katere tuje podatkovne predmete bo nastavljajl zaščitni mehanizem in kakšne operacije bo izvajal na njih;

-katere operacije nad novim podatkovnim predmetom naj sestavljajo vmesnik podatkovni predmet - okolje;

-kakšna so legalna zaporedja operacij nad podatkovnim predmetom in pod kakšnimi pogoji.

Zaščitni mehanizem je proces, ki sme posegati v tuje podatkovne predmete med odločanjem in med izvajanjem definiranih operacij. Tako se zgodi, da procedure enega zaščitnega mehanizma kličejo procedure drugih zaščitnih mehanizmov in operacije na vmesniku z okoljem posredno posegajo tudi v druge podatkovne predmete. Zato niti ni več nujno, da je legalno zaporedje operacij nad enim podatkovnim predmetom popolno v tem smislu, da bi zaščitni sistem reševal vse probleme sinhronizacije procesov - uporabnikov na predmetu, ki ga nadzoruje. Če na tistih odsekih zaporedij, ki še niso popolni, ne dovolimo direktnega dostopa uporabnikov in vključimo pred vhod predmeta drug podatkovni predmet oz. njegov

zaščitni mehanizem, se uporabnikove zahteve večkrat filtrirajo. Tako dosežemo, da zahteve, ki so na nekem vhodu prepovedane in jih vhod ne zna izločiti, filtrira filter pred tem vhodom in prepovedane zahteve (npr. interferirajoče paralelne operacije) se nikoli ne pojavijo. Programerju ni več treba direktno graditi kompleksnih podatkovnih struktur in zaščitnih mehanizmov, gradi jih hierarhično od spodaj navzgor ali celo rekurzivno iz bolj enostavnih gradnikov.

Zaščitni mehanizem izvaja definirane operacije kot podprograme ali kot posebne procese. Zato za vsako spremembo obstoječih podatkovnih predmetov veljajo pravila za spreminjanje aktivnih podprogramov oz. procesov, ki so morda očetje ali sinovi drugih procesov in podprogramov.

Poseben jezik za opis legalnih zaporedij operacij, kakršnega predstavljajo znani izrazi za opis poti, je odveč, ker se pri velikih zaščitnih mehanizmih razraste v prav tako kompleksnost, ki je značilna za programske jezike, s katerimi opisujemo uporabniške procese. Zato naj vse dogajanje v sistemu opisuje enoten jezik po zgledu jezika ADA. Če obravnavamo zaščitne mehanizme kot posebnosti, ki jih šele pri prevajanju realiziramo z normalno programsko kodo, potrebujemo tudi posebne tehnike za preverjanje pravilnosti delovanja paralelnih procesov.

Naravo objekta za kontrolo sinhronizacije karakterizira invariantna relacija med spremenljivkami ki je izpolnjena vedno, kadar element ni aktiven. To relacijo uporabljamo skupaj z drugimi relacijami v sistemu za formalno dokazovanje pravilnosti algoritmov. Aktivnost objekta za sinhronizacijo precizno opišemo še s tem, da navedemo pogoje, ki morajo biti izpolnjeni, preden se lahko izvede dana operacija. Za objekt posebne vrste, ki ga nanovo vpeljemo v jezik, moramo odkriti njegovo invariantno relacijo, kar ni zmeraj enostavno. Res pa je, da pozneje invariantne relacije ni treba ponovno iskati, medtem ko moramo objekte za kontrolo sinhronizacije, ki niso tipizirani, opisovati sproti. Vendar to ni problem, če je objekt za kontrolo sinhronizacije zgrajen po enakih zakonih, kot ostali objekti v sistemu. S tem se ne ravna po standardu za posebno področje sinhronizacije, ampak po standardu za celoten sistem. Podrobna zgradba objekta za sinhronizacijo ni vnaprej znana, znane pa so metode za konstruiranje objekta, ki so povsem splošne.

Zaščitni mehanizem v modernem programskem jeziku je proces kot vsak drug, ima skupino lokalnih podatkov, ki jih ščiti, in skupino procedur za dostop do podatkov, ki jih drugi procesi kličejo preko procesnih vhodov. Razen podatkov z dinamično zaščito pa še vedno obstajajo podatki s statično zaščito, pri katerih predpisuje in nadzira deljivost ali nedeljivost operacij implicitni sistemski proces. Tako sploh ne moremo več govoriti o direktnem dostopu do spremenljivk in smemo znotraj računalniškega sistema vse spremenljivke, ki jih uporabljamo, vključiti v različne funkcijske bloke. Vsa sinhronizacija se pravzaprav odvija implicitno s sistemsko kontrolo procesnih vhodov po principu rendez-vous-a, klicani ali kličoči proces avtomatično čakata drug na drugega v določenih točkah algoritma.

Zaščitni mehanizem, ki je implementiran kot normalen proces, lahko predstavlja aktivni filter vhodnih klicev. Vse do sedaj omenjene vrste zaščitnih mehanizmov so imele edino funkcijo, da zahteve za izvedbo operacije ali sprejmejo ali zavrnejo ali zakasnijo. Ko je mehanizem neko zahtevo sprejel, je operacijo vedno izvedel na enak način, neodvisno od stanja sistema. Seveda bi v procedure, ki opisujejo izvajanje operacij, lahko vključili upoštevanje stanja sistema, vendar je bil osnovni namen ta, da uporabnikova zahteva kadarkoli sproži isto proceduro. Zaščitni mehanizem je bil izključno pasiven filter. Zaščitni mehanizem, ki je normalen proces, sprejema enake zahteve na več različnih procesnih vhodih, v odvisnosti od tega, do katerega vhoda je pritekel algoritem zaščitnega mehanizma. Vsak procesni vhod predstavlja stavek

when klic sprejet do akcija

in vsak procesni vhod ima svojo lastno posledico klica. Zaščitni mehanizem ni več pasivni filter, ki spušča uporabnikove klice lokalnih procedur v notranjost, temveč te klice aktivno preoblikuje. Prav tako je lahko zaščita lokalnih spremenljivk samo stranska naloga procesa. Aktivni filter ni samo nadzornik uporabnikove aktivnosti, ampak servisni proces.

Razvoj jezikovnih elementov za sinhronizacijo oz. komunikacijo med paralelnimi procesi je s tem dosegel tisto točko, ko se obravnavano posebno področje vključi v jezik na tako popoln način, da neha biti posebnost in ga ni več treba principielno raziskovati z jezikovnega stališča. Oblikovalci jezikov za paralelno programiranje se ukvarjajo v glavnem s podrobnostimi izvedbe in manjšimi spremembami jezikovnih konstruktov, ki ne vplivajo na bistvo rešitve.

Oglejmo si še problem sinhronizacije procesov, ki jih implicitno sprožajo programi v jezikih za nepostopkovno programiranje. Tudi nepostopkovne programe sestavljajo zaporedja ukazov in sicer za vnašanje novih dejstev v bazo znanja, za spreminjanje starega znanja ter za iskanje podatkov in izpeljevanje logičnih sklepov na osnovi znanja v bazi. Sistem pogosto realiziramo kot pomnilnik brez vnaprej deklarirane logične strukture, kjer so podatki spravljeni v obliki sintaktičnih konstruktov. Osnovni proces v sistemu je administrator baze, ki sprejema ukaze uporabnikov in za procesiranje vsakega ukaza sproži poseben proces. Procesi posegajo v bazo znanja kot pisci ali čitalci. Zaščitni mehanizem baze dodeli vsakemu piscu del praznega pomnilniškega prostora ali obstoječi sintaktični konstrukt, ki ga proces želi spremeniti. Proces - pisec je izključni lastnik dodeljenega objekta, dokler ne konča dela. Procesi smejo prosto brati vse konstrukte v bazi znanja, ki niso last procesov - piscev, ker samo ti konstrukti predstavljajo veljavno znanje. Čitalci se obračajo na zaščitni mehanizem baze, kadar iščejo v bazi konstrukte z dano strukturo. Zaščitni mehanizem mora sinhronizirati procese tako, da ne dovoli spreminjanja tistih konstruktov, ki jih že bere kak proces, in branja tistih konstruktov, ki jih kak proces že spreminja. Imamo klasičen primer piscev in čitalcev. Ukrepi procesov, ki uporabljajo znanje iz baze, so posledice prebranega znanja. Zato zahtevamo tudi tu določeno stopnjo nedeljivosti vzroka in posledice. Baza znanja naj bi bila slika nekega sveta. Ne glede na to, ali je ta svet realen ali abstrakten, smemo reči, da se celotno znanje spreminja v odvisnosti od stopenjske spremenljivke in logika sveta predpisuje, da se morajo nekateri elementi znanja spremeniti sočasno, to pomeni, da mora staro znanje v

trenutku preiti v novo. Zaradi vsaj delno sekvenčnega delovanja računalnika pa to ni res in obstaja nevarnost, da bo uporabnik brat kombinacije elementov znanja, ki se v svetu, ki ga opisuje baza znanja, ne bi pojavile. Zato je važna sinhronizacijska naloga zaščitnega mehanizma baze, da simulira realno spreminjanje sveta in prepreči dostop uporabnikov baze do podatkov, ki se spreminjajo, med trajanjem prehodnega pojava. Pri tem se naloga ne ujema popolnoma z blokiranjem čitalcev, enega sintaktičnega konstrukta med njegovim spreminjanjem. Obstajati mora ukaz za blokado poljubne skupine sintaktičnih konstruktov, dokler vsi ne dobijo nove vrednosti.Procesi - uporabniki baze znanja morajo za smiselno upoštevanje blokade ločiti med primeroma, da je neka izjava v bazi zanikana ali da v bazi ne obstaja niti trditna niti nikalna oblika izjave. Če izjave ni v bazi, mora proces avtomatično ponovno preiskovati bazo do preteka dovoljenega časa iskanja in šele nato podati uporabniku končno poročilo. Označevanje sintaktičnih konstruktov, ki naj se navidezno spremenijo istočasno, in njihovo vključevanje v bazo po spremembi je nedeljiva operacija, med katero procesi lahko spreminjajo konstrukte, ki so jih že prej rezervirali kot pisci, ni pa dovoljeno branje že označenih konstruktov. Sinhronizacijske potrebe v sistemu še narastejo, če en ukaz uporabnika sproži izvajanje večih procesov.

Kot vidimo, se kaže postopkovnost nepostopkovnih programskih jezikov kot komunikacija sistema in uporabnika, implicitno pa tudi med izvajanjem programov. Postopkovni in nepostopkovni jeziki se po sintaksi bistveno razlikujejo. S postopkovnimi eksplicitno opisujemo procese v računalniku, z nepostopkovnimi samo deklariramo relacije med podatki in prepustimo gradnjo procedur in organizacijo procesov za iskanje odgovorov na naša vprašanja sistemu. Med samim izvajanjem so problemi komunikacije in sinhronizacije v obeh primerih enako pomembni in se rešujejo z enakimi pristopi, le da se pri nepostopkovnem programiranju ne odražajo na jeziku in ker procese organizira sistem, so procesni sklopi preprostejši in manj raznoliki.

3. Sklep

Paralelno procesiranje se podreja majhni skupini temeljnih zakonitosti, ki jih brez težav odkrijemo in opišemo. Kljub temu, da je bil razvoj jezikovnih elementov za opis sinhronizacije paralelnih procesov precej počasen, je danes dosegel zadovoljivo stopnjo. Zato se je težišče dela preneslo na optimiranje programov v multiprocesnem okolju ob podpori visokih programskih jezikov.

4. Viri

N.Wirth
Toward a Discipline if Real-Time Programming,
Comm. ACM 20 (1977), No.8, 577-583

N.Wirth
MODULA: A Language for Modular Multiprogramming,
Software Practice and Experience 7 (1977), 3-35

E.W.Dijkstra
Co-operating Sequential Processes,
Programming Languages,
ed. F.Genuys, Academic Press, 1968

C.A.R. Hoare
Monitors: An Operating System Structuring Concept,
Comm. ACM 17 (1974), 10, 549 - 557

S. Andler
Synchronization Primitives and the Verification
of Concurrent Programs,
Carnegie-Mellon University

M.Exel, F.Prijatelj
Programiranje sprotnih in vgnezdenih sistemov:
Procesi v ADI.
Informatica 1981/2