Anton Ružić, Aleš Klofutar
Inštitut Jožef Stefan, Ljubljana

UDK: 681.326

V članku podajamo pregled paralelnega programiranja. V uvodu opisujemo prednosti in potrebe po uporabi metod paralelnega programiranja ter načine doseganja paralelnega izvajanja. Na to opisujemo paralelno programiranje s prikazom razvoja tega področja. Ta je potekal skozi nezanesljivost začetne tovrstne programske opreme, postavljanje konceptualnih temeljev do razvoja programskih jezikov, ki vključujejo postavljene koncepte.

THE DEVELOPMENT OF CONCURRENT PROGRAMMING. In this article we describe concurrent programming. The methods and techniques are introduced through overviewing of the development of concurrent programming. This went through several stages: the initial development of complicated and unreliable software systems, the search for abstract concepts that simplified understanding and the incorporating of these concepts into new programming languages.

## 1 Uvod

Z nazivom paralelno programiranje zajamemo programske zapise in tehnike, ki izražajo možnost paralelnega izvajanja in ki omogočajo reševanje rezultirajočih sinhronizacijskih in komunikacijskih problemov. Posamezne programske enote, ki se paralelno izvajajo, imenujemo procesi. Imamo lahko navidezno in dejansko paralelnost. Navidezno paralelnost imamo, če tečejo vsi procesi na enem procesorju in je vsakemu procesu dodeljen določen čas. Dejansko paralelnost dosežemo, če uporabimo več procesorjev in vsak prevzame en ali več procesov.

Paralelni program omogoča, da računalnik izvaja več nalog istočasno. S paralelnim programiranjem zvečamo računalnikovo učinkovitost in enostavnejše obvladujemo okolja, v katerih se računalnik hkrati posveča različnim opravilom.

V prvem delu na kratko opisujemo prednosti, ki nam jih omogoča uporaba paralelnega programiranja pri izgradnji programskih sistemov. Podajamo možne načine za uvajanje paralelnosti in probleme, ki jih moramo pri tem rešiti.

V nadaljevanju širše opisujemo možne načine za predstavljanje paralelnih dejavnosti (procesov) in možne načine za komunikacijo in sinhronizacijo med procesi. Posamezne načine opisujemo skozi pregled razvoja paralelnega programiranja. Razvoj se je začel konec 60. let. Začetni motiv je bil izkoriščanje možnosti, ki jih je nudila nova materialna oprema.

## Področja uporabe

Paralelno programiranje se uporablja predvsem pri operacijskih sistemih in v sistemih za delo v realnem času.

Pri operacijskih sistemih lahko izboljšamo izkoristek materialne opreme, če imamo v pomnilniku več programov hkrati. Medtem ko en program čaka na izvedbo vhodne ali izhodne operacije s počasno periferno enoto, procesor računalnika izvaja nek drug program, naložen v pomnilniku. V tem primeru se procesor preklaplja med različne programe, ki se izvajajo navidezno istočasno. Več uporabnikov lahko hkrati dela na računalniku in uporablja vse računalnikove resurse.

Uporaba paralelnega programiranja je posebej koristna, če ne skoraj nujna, pri vgnezdenih sistemih za delo v realnem času, kot so na primer procesni sistemi. Pri teh je delitev globalne naloge na več procesov motivirana predvsem z enostavnejšim reševanjem naloge. Računalnik mora pravočasno sprejemati signale z različnih neračunalniških ali računalniških naprav in jih ustrezno krmiliti. Zahteve po sprejemanju in krmiljenju se v splošnem lahko pojavijo v različnih časovnih trenutkih,

saj je to pogojeno z dinamiko sistemov. Računalnik mora hkrati obravnavati in obdelovati podatke s številnih vhodov in izhodov. V tem primeru upravljalsko nalogo enostavneje in naravneje rešimo, če jo razdelimo na več delov, ki se paralelno izvajajo.

Paralelno programiranje uporabimo tudi zato, da zvečamo zmogljivost računalnika in skrajšamo čas izvajanja neke naloge. Namesto enoprocesorskega vzamemo večprocesorski računalniški sistem. Celotno nalogo razdelimo na podnaloge, ki se paralelno izvajajo na posameznih procesorskih elementih.

Delitev naloge na dele, ki se lahko paralelno izvajajo

Pri zgrajevanju sistema, ki uporablja paralelno programiranje, moramo najprej ugotoviti, katere aktivnosti se lahko opravljajo paralelno. Celotna naloga, ki jo rešujemo, je lahko postavljena tako, da jo enostavno razdelimo na paralelne aktivnosti. Primer je krmiljenje določenega števila perifernih naprav. Pri nekaterih nalogah je ta problem bolj zapleten in se ne da zadovoljivo rešiti z uporabo "ad hoc" načinov. Takšen primer so nekateri matematični izračuni, kjer moramo identificirati delne izračune, ki se lahko istočasno izvajajo, preden se izmenjajo vmesni rezultati izračunov. Paralelno preračunavanje moramo optimizirati pri omejitvah zaradi serijsko-paralelnih prednostnih razmerij.

Pomembno merilo za uspešnost delitve je čim večja neodvisnost procesov. V idealnem primeru so procesi popolnoma samostojni in operirajo le nad svojimi podatki. Ponavadi pa posamezni procesi opravljajo neko skupno nalogo, potrebujejo iste resurse ali kako drugače sodelujejo. Zato je potrebno, da procesi med sabo komunicirajo ali pa se v določenih točkah sinhronizirajo. Potem, ko smo razdelili nalogo na logične enote, moramo izbirati način za izražanje paralelnega izvajanja posameznih procesov in izbrati orodja oziroma načine, ki bodo omogočili nadzorovano in pravilno interakcijo procesov. Te načine širše opisujemo v naslednjem poglavju.

Načini paralelnega izvajanja

Nazadnje se moramo pri takšnih sistemih odločiti za primeren način paralelnega izvajanja in sinhronizacije ter komunikacije med procesi.

Če nimamo nobenega primernega sistemskega orodja, napišemo večposlovni izvrševalnik. To je pravzaprav inačica jedra pri multiprogramskih operacijskih sistemih. Celotno nalogo razdelimo na procese in jih skupaj z izvrševalnikom naložimo v pomnilnik. Izvrševalnik skrbi za izmenjevanje izvajanja poslov na procesorju (procesom, ki so del enega uporabniškega programa, lahko rečemo posli, angl. "tasks"). V izvrševalnik vgradimo tudi semaforje ali druge mehanizme za komunikacijo in sinhronizacijo.

Pri multiprogramskem ali večposlovnem operacijskem sistemu uporabimo za doseganje paralelnega izvajanja programov sistemski izvrševalnik in druge funkcije, ki jih nudi operacijski sistem. V tem primeru posamezne procese zapišemo kot posle ali samostojne programe v primernem programskem jeziku. S sistemskimi ukazi sprožimo paralelno izvajanje posameznih programov. Takšni operacijski sistemi podpirajo tudi nek način medprocesne komunikacije. To so lahko semaforji, dogodkovne zastavice, poštni predali in podobno.

Nazadnje imamo možnost, da uporabimo programske jezike, ki podpirajo paralelno programiranje. Osnova teh jezikov je jedro, ki omogoča paraleno izvajanje modulov in interakcijo med procesi. Te možnosti uporabljamo s primernimi stavki jezika. V to kategorijo lahko postavimo jezike PEARL, Concurrent Pascal, Modula, Edison, Ada in druge. Vključitev paralelnosti z uporabo takšnih jezikov je zelo primerna, saj se v tem primeru bolj posvetimo reševanju naloge in ne implementaciji mehanizmov paralelnosti.

## 2 Pregled razvoja in opis načinov in orodij paralelnega programiranja

V prejšnjem poglavju smo pokazali, kako uporabljamo paralelno programiranje za učinkovito izkoriščanje računalnikovih zmogljivosti, zvečevanje računalnikove zmogljivosti in za uspešno obvladovanje okolij, v katerih se mora računalnik hkrati posvečati različnim dogodkom. Paralelni sistemi so pri takšnih sistemih koristni in potrebni, izdelava pravilnih in zanesljivih programov pa je zahtevna. Najmanjša napaka lahko povzroči, da se paralelni program izvaja nepravilno in neponovljivo, kar onemogoča testiranje programa. Opisali bomo, kako so programski inženirji postopoma reševali ta problem.

Paralelno programiranje so uvedli, da bi izkoristili dosežke na področju materialne opreme. Po začetnem poskušanju so programerji ob pomanjkljivem znanju izdelali zapletene sisteme. Ti sistemi so zato bili tako nezanesljivi, da so sami načrtovalci uporabili izraz "kriza programske opreme". Računalniški znanstveniki so

spoznali pomen problema in začeli iskati abstraktne koncepte, s katerimi so poenostavili razumevanje paralelnih programov. Ko so razumeli bistvo problema, so postavili zapis za osnovne koncepte in jih definirali tako precizno, da so jih lahko vključili v nove programske jezike. Jezikovni zapis je omogočil napredek pri formalnem razumevanju problema.

V tem poglavju bomo opisali razvoj paralelnega programiranja. Ob tem bomo obravnavali načine izražanja paralelnega izvajanja procesov in načine za komunikacijo in sinhronizacijo med procesi.

## 2.1 Razvoj materialne opreme

Paralelno programiranje se je začelo razvijati pri reševanju problemov pri operacijskih sistemih, ki so sledili razvoju materialne opreme.

Prvi, enoposlovni operacijski sistemi so bili zelo primitivni in so naenkrat izvajali le en posel (uporabnikovo nalogo, angl. "single job"). Tipična vhodna enota je bila čitalec kartic in tipična izhodna enota je bila vrstični tiskalnik. Obe napravi sta bili veliko počasnejši od procesorja. Zaradi čakanja na počasnejše naprave je bil procesor veliko časa neizkoriščen. Poleg tega so bili prvi sistemi občutljivi na programerjeve napake, ker ni imel zaščitnih mehanizmov.

Z razvojem hitrejših pomnilniških perifernih naprav so izdelali operacijske sisteme za paketno obdelavo (angl. "batch processing"). Na manjših računalnikih so večje število poslov posneli s počasnejših čitalcev kartic na hitrejši magnetni trak ali disk. Sistem je potem s teh hitrejših pomnilniških enot zaporedno bral in izvajal posle, rezultate pa je zapisoval na hitrejše pomnilniške enote. Ko je bila obdelava končana, so rezultate natisnili. Sistemi za paketno obdelavo so bili boljši od prejšnjih, so pa še vedno zaporedno izvajali posle. Rezultati posameznih poslov so bili dostopni, ko so bili izvedeni vsi posli. Zato so rezultati krajših in daljših poslov bili dostopni ob istem času.

Naslednjo izboljšavo operacijskih sistemov je omogočila uvedba oziroma uporaba prekinitev, s katerimi so presegli sekvenčno naravo računalnika in dosegli delitev procesorjevega časa med različne programe. Pripadajoče tehnike, ki podpirajo takšno paralelno izvajanje programov imenujemo "multiprogramiranje". Tu se je začel razvoj paralelnega programiranja.

Multiprogramiranje so uvedli zato, da bi bil procesor in vhodno izhodne enote čim bolj izkoriščeni. Pri osnovnem načinu imamo istočasno v pomnilniku več poslov s pripadajočimi podatki. Časovnik v enakomernih časovnih presledkih prekinja trenutno izvajani posel in sproži izvajanje naslednjega posla. V kratkem času se izmenjajo vsi posli, tako da se izvajajo navidezno istočasno. Trenutno aktivni posel se ustavi tudi takrat, ko sproži izvajanje vhodne ali izhodne operacije s periferno enoto. Medtem ko se vhodna ali izhodna operacija izvaja (na primer zapisovanje na disk), se na procesorju izmenjujejo drugi posli. Ko periferna naprava konča operacijo, s prekinitvijo javi da se pripadajoči posel lahko nadaljuje.

Pozneje so takšne multiprogramske sisteme za paketno obdelavo dopolnili tako, da je lahko več uporabnikov hkrati interaktivno delalo z računalnikom preko terminalov. Takšne sisteme imenujemo operacijski sistemi s časovnim prepletanjem (angl. "time sharing operating system").

## 2.2 Nezanesljivost programske opreme

Možnost istočasnega izvajanja večjega števila programov na enem računalniku, ki so jo omogočili prvi multiprogramski sistemi, je zelo zvečala zmogljivosti računalnikov, povečala pa se je zapletenost. Pri teh sistemih je namreč zelo pomembno, da se posamezni programi pri izvajanju ne motijo in da potekajo na pravilen način. Programske napake so povzročile, da se je paraleleni program obnašal nepravilno in časovno odvisno. Ker so bile posledice napak različne od primera do primera, celo pri enakih vhodnih podatkih, jih je bilo zelo težko odkriti.

Zaradi omenjenih težav paralelnosti je bilo pomembno, da se uporabniku omogoči enostaven, zaporeden pristop do stroja. Operacijske sisteme so zgradili zato, da bi bili računalniški sistemi učinkoviti in zanesljivi ter enostavni za uporabo.

Zgodnji operacijski sistemi za paketno obdelavo, kot Atlas (1961) in Exec II (1962) so bili učinkoviti in enostavni, niso pa bili popolnoma zanesljivi. Prvi sistemi s časovnim prepletanjem, kot CTSS (1962) in SDCQ-32 (1964) so bili sorazmerno majhni. Operacijski sistemi naslednje generacije pa so bili veliko bolj obsežni in zapleteni. Razvoj sistema Multics (1965) je tako zahteval 200 človek let, OS 360 (1966) pa celo 5000 človek let. Zaradi svoje velikosti je bil OS 360 precej nezanesljiv. V vsaki verziji je bilo okrog 1000 napak.

Veliki operacijski sistemi so se dnevno podirali in postalo je dvomljivo ali resnično omogočajo učinkovito in zanesljivo delo računalnika. Postalo je jasno, da

takšne obsežne programe ni mogoče izdelati brez določenih konceptualnih temeljev, ki bi omogočili boljše razumevanje. Računalniški znanstveniki, ki so pri svojem delu prav tako uporabljali računalnike, so ugotovili pomembnost operacijskih sistemov in začeli delo na tem področju.

## 2.3 Konceptualni temelji

Najpomembnejši dosežek v nadaljnjem razvoju je bila ideja o delitvi paralelnega programa v sekvenčne procese, ki se asinhrono izvajajo. Obnašanje programa mora biti neodvisno od relativnih hitrosti procesov.

Proces je programski modul, sestavljen iz podatkovne strukture in zaporedja ukazov, ki operirajo nad njo. Če proces operira le na svojih podatkih, se bo obnašal popolnoma enako (ponovljivo) vsakič, ko bo pognan z enakimi podatki.

Procesi, ki si delijo računalniške resurse ali pa delajo na skupnih nalogah morajo biti zmožni pravilne interakcije, ki jo imenujemo procesna komunikacija ali procesna sinhronizacija. Takšne procese delimo glede na medsebojno razmerje do nekega resursa na tekmovalne procese, ki tekmujejo za neke stalne resurse (tračna enota, tiskalnik, spremenljivke itd.) in takšne, ki so v razmerju proizvajalca in potrošnika začasnih ali potrošnih resursov (sporočila, signali itd.). Če si proizvajalci in potrošniki izmenujejo sporočila, jih imenujemo komunicirajoči procesi, če pa si izmenjujejo sinhronizacijske signale, jih imenujemo sodelujoči procesi.

### Kritične sekcije

Dijkstra je ugotovil, da komunikacijo med procesi lahko prevedemo na izvrševanje operacij na podatkih, ki so skupni večim procesom. Pomembno je, da naenkrat le en proces izvaja določene operacije na nekem skupnem podatku, ker sicer lahko nastopijo nepredvidljive posledice. Vzrok je v tem, da nobeden od procesov ne ve, kakšne operacije izvajajo ostali procesi v istem času nad skupnimi podatki. Pripadajoče napake so časovno neodvisne in različne odvisno od tega, kako se procesi pri izvajanju prekrivajo.

Nazoren primer je, ko si dva procesa delita navadno spremenljivko, ki je števec dogodkov. Če se prekrije povečevanje števca dobimo:

| cikli | prvi proces | drugi proces |
|---|---|---|
| | (n=3) | |
| t0 | naloži n | |
| t1 | | naloži n |
| t2 | | n:=n+1 |
| t3 | | shrani n |
| t4 | n:=n+1 | |
| t5 | shrani n | |
| | (n=4) | |

Po dvojnem povečanju vrednosti smo iz n=3 dobili n=4 namesto n=5.

Tiste dele procesov, ki vršijo operacije na skupnih spremenljivkah, je Dijkstra imenoval kritične sekcije. Potrebno je zagotoviti, da bo samo en proces naenkrat izvajal kritično sekcijo, kar imenujemo problem vzajemnega izključevanja izvajanja kritične sekcije.

Vzajemno izključevanje zagotovimo, če proces sestavimo tako:

```
neodvisni del
....
vstopni protokol
kritična sekcija
izstopni protokol
....
neodvisni del
```

Vsak proces pred kritično sekcijo izvede nek vstopni protokol. Ta bo dovolil nadaljevanje samo takrat, kadar noben drug proces ne izvaja kritične sekcije. Na koncu kritične sekcije proces izvede izstopni protokol, ki omogoči, da v kritično sekcijo vstopijo drugi procesi.

Najbolj znan je Dekkerjev algoritem za zagotavljanje vzajemnega izključevanja. Algoritem za dva procesa, ki ga predstavimo v višjem programskem jeziku poteka tako:

```
program Dekker;
var
  turn : integer;
  c1,c2: integer;

procedure p1;                    (*prvi proces*)
begin
  repeat
    .............;               (*neodvisni del*)
    c1:=0;                       (*vstopni protokol*)
    while c2=0 do
      if turn=2 then
```

```
      begin
        c1:=1;
        while turn=2 do;
        c1:=0
      end;
    crit1;                  (*kritična sekcija*)
    turn:=2;                (*izstopni protokol*)
    c1:=1;
    .....                   (*neodvisni del*)
    forever
end;

procedure p2;               (*drugi proces*)
begin
  repeat
    .....;                  (*neodvisni del*)
    c2:=0;                  (*vstopni protokol*)
    while c1=0 do
      if turn=1 then
        begin
          c2:=1;
          while turn=1 do;
          c2:=0
        end;
    crit2;                  (*kritična sekcija*)
    turn:=1;                (*izstopni protokol*)
    c2:=1;
    .....                   (*neodvisni del*)
    forever
end;

begin                       (*glavni program*)
  c1:=1;
  c2:=1;
  turn:=1;
  cobegin           (*paralelno izvajanje p1 in p2*)
    p1;p2
  coend
end.
```

V programu smo uporabili paralelni stavek cobegin - coend, ki ga bomo podrobneje opisali v naslednjem poglavju. Vstop v kritično sekcijo proces p1 (p2) naznani z nastavljanjem c1 (c2) na 0. Če p2 (p1) hkrati izvaja vstopni protokol, spremenljivka turn odloča, kateri proces bo vstopil v kritično sekcijo.

Algoritmi za vzajemno izključevanje več kot dveh procesov so precej zapleteni, zato so neprimerni za praktično uporabo. Druga slabost takšnih algoritmov je v zaporednem testiranju vstopnega pogoja, kadar želi več procesov hkrati izvajati isto kritično sekcijo. To vidimo iz stavka: while pogoj do (* nič *). Medtem ko en proces vstopi v kritično sekcijo, drugi procesi zapravljajo računalniški čas s preverjanjem, ali je sekcija prosta (angl. "busy waiting").

## Semaforji

Dijkstra je uvedel podatkovni tip semafor, ki je primeren za prenašanje sinhronizacijskih signalov (1968). Semafor lahko uporabimo tudi pri reševanju ostalih problemov paralelnega programiranja. Na primer, z njim enostavno rešimo problem vzajemnega izključevanja.

Nad spremenljivko s tipa semafor sta definirani dve operaciji:

wait(s): če s>0 tedaj s:=s-1 sicer se izvajanje procesa, ki je klical wait(s) ustavi in proces se postavi v čakalno vrsto,

signal(s): če je bil nek drugi proces P z operacijo wait(s) nad tem semaforjem ustavljen in postavljen v vrsto ga zbudi in izvajaj, sicer s:=s+1

Operaciji "wait" in "signal" morata biti implementirani kot primitivni operaciji, torej se nedeljivo izvajata. Ko nek proces izvaja semaforsko instrukcijo, počakajo vsi ostali procesi, ki v tem času tudi zahtevajo izvajanje semaforske instrukcije. Semaforje, ki lahko zavzamejo poljubno pozitivno vrednost imenujemo splošni semaforji. Če dovolimo, da zavzamejo samo vrednosti 0 in 1, imenujemo takšne semaforje binarne semaforje.

Rešitev sinhronizacije dveh procesov s semaforji je preprosta:

```
program synchronisation;
var
  s: semaphore;
procedure p1;
begin
  .....
  wait(s);  (*cakanje na sinhronizacijo s p1*)
  .....
end;
procedure p2;
begin
  .....
  signal(s); (*sprozanje sinhronizacijskega signala*)
  .....
end;
begin       (*main program*)
  s:=0;
  cobegin   (*paralelno izvajanje p1 in p2*)
    p1;p2
  coend;
end.
```

Z operacijo "signal" proces preko semaforske spremenljivke odda sinhronizacijski signal drugemu procesu, ki oddani signal sprejme z "wait" operacijo. V paralelnem sistemu programer ne more predvideti relativne hitrosti asinhronih procesov. Ne moremo vedeti, če bo proces poslal signal preden ga bo drugi procesor pripravljen sprejeti. Semaforske operacije so definirane tako, da ni pomembno v kakšnem vrstnem redu se izvajajo. Proces, ki poskuša sprejeti sinhronizacijski signal še preden je ta oddan, se postavi v čakalno vrsto in zakasni, dokler nek drug proces ne bo oddal ustreznega signala. Če se signali oddajajo hitreje, kot se sprejemajo, se enostavno shranijo v semaforski spremenljivki, dokler ne bodo uporabljeni. Zaradi komutativnosti semaforskih operacij postane sinhronizacija procesov časovno neodvisna.

Rešitev navadne sinhronizacije s semaforji je enostavna, drugi problemi pa lahko zahtevajo bolj zapletene rešitve. Slabost semaforjev je tudi v tem, da se lahko sistem, zgrajen s semaforji, podre, če pozabimo na eno samo semaforsko operacijo.

Dijkstrin multiprogramirni sistem THE (1968) je uvedel večino konceptov na katerih temelji današnje razumevanje paralelnega programiranja. Njegov sistem je bil hierarhično zgrajen iz več programskih plasti, ki so fizični stroj postopoma pretvarjali v prijaznejši abstraktni stroj, ki je lahko izvajal številne procese. Ti so si delili obsežen homogen pomnilnik in številne virtualne naprave.

Pri sistemih, pri katerih paralelni procesi uporabljajo iste resurse so začeli raziskovati tudi načine urejanja zahtev po zaseganju resursov in prenašanja sporočil, da se preprečijo smrtni objemi (angl. "deadlock"). To je pojav, ko vsak izmed dveh ali več procesov zaseda nek resurs in čaka na resurs, ki ga zaseda drug proces. Napačna rešitev tega problema povzroči neskončno čakanje procesov.

## 2.4 Razvoj jezikov z elementi za paralelno programiranje

Okoli 1970. leta so raziskovalci začeli razvijati jezikovne zapise za opis novih konceptov.

Koncept programskega jezika mora predstavljati splošno idejo, ki se pogosto uporablja. Pomen in pravila koncepta programskega jezika morata biti natančno definirani. Predstavljen mora biti s kratkim in jedrnatim zapisom, ki omogoča enostavno spoznavanje elementov koncepta in njihove medsebojne odvisnosti. Pomembno je tudi, da je možna varna in učinkovita implementacija in da prevajalnik preverja, da so pravila gleda koncepta zadovoljena. Uporabljeni koncept mora omogočati programerju predvideti hitrost in velikost programa.

### Paralelni stavek

Dijkstra je uvedel zapis paralelnega stavka, ki določi, da se več sekvenčnih stavkov izvaja paralelno. Paralelni stavek se zaključi, ko se zaključijo vsi sekvenčni stavki. Primer paralelnega stavka je:

```
var
  this, next: line;
....
cobegin
  consume(this); input(next)
coend
```

Medtem ko stavek consume porablja vrstico this, stavek input sprejema naslednjo vrstico next.

Paralelni stavek ima predvidljiv učinek samo v primeru, če posamezni pripadajoči stavki, ki predstavljajo paralelne procese operirajo nad različnimi spremenljivkami (v našem primeru this in next) . če več stavkov operira nad istimi spremenljivkami, bo učinek paralelnega stavka časovno odvisten. Da bi preprečili časovno odvisne programske napake mora prevajalnik razpoznavati privatne spremenljivke procesa, ki morajo biti nedostopne drugim procesom.

### Kritične sekcije in pogojne kritične sekcije

čeprav je bistveno, da so nekatere spremenljivke dostopne samo enim procesom, je v primeru sodelovanja in komunikacije med procesi potrebno, da si procesi delijo nekatere spremenljivke.

Hoare in Brinch Hansen sta leta 1972 predlagala zapis za prirejanje deljene spremenljivke kritičnim sekcijam, ki operirajo z njo. Za primer lahko definiramo deljeno spremenljivko, ki jo uporabljamo kot uro:

```
var
  clock: shared integer
```

Procesi inkrementirajo in čitajo to spremenljivko s sledečimi stavki:

```
tick:    region clock do clock:=(clock+1) mod max
read(x): region clock do x:=clock
```

Deljena spremenljivka se lahko uporablja samo znotraj kritičnih sekcij (region), kar preverja prevajalnik. Kritične sekcije so tako implementirane, da je zagotovljeno, da se posamezne sekcije izvajajo brez prepletanja, ena za drugo.

Hoare in Brinch Hansen sta postavila tudi koncept in zapis pogojne kritične sekcije. Izvajanje sekcije se odlaša, dokler deljena spremenljivka ne izpolni nekega pogoja. Kot primer lahko prikažemo izravnalnik za sporočila, ki se sestoji iz vrstice "slot" in boolean vrednosti "full", ki kaže ali je vrstica polna:

```
var
  buffer: shared record
            slot:line;
            full:boolean
          end
```

V izravnalnik pišemo samo, ko je prazen in čitamo samo, ko je poln:

```
send(m):    region buffer when not full do
              begin slot:=m; full:=true end

receive(m): region buffer when full do
              begin m:=slot; full:=false end
```

Pri pogojnih kritičnih sekcijah nastopa problem učinkovite implementacije. Gre za omejevanje ponavljajočega testiranja boolean izraza, dokler pogoj ni izpolnjen. V ta namen so uvedli procesne vrste prirejene posameznim deljenim spremenljivkam. če proces testira v pogojni kritični sekciji vrednost izraza in pogoj ni izpolnjen, se postavi v procesno vrsto, prirejeno deljeni spremenljvki. Vsakič, ko neki proces konča izvajanje kritične sekcije, se testirajo izrazi procesov v vrsti. V primeru, da so izrazi nekaterih procesov izpolnjeni, se aktivira eden od teh procesov.

## Monitorji

Dijkstra je predlagal, da v en programski modul zajamemo vse operacije nad deljenimi podatkovni strukturami, namesto da jih raztresemo po celotnem programu. S tem se poveča jasnost interakcij med procesi. Brinch Hansen je predlagal zapis jezika za ta koncept monitorja (1973) Hoare je opisal monitorski koncept in podal primere (1974).

Za ilustracijo vzemimo izravnalnik za sporočila z operacijami zapisovanja in čitanja, ki ga zgradimo z monitorjem.

```
monitor buffer;

var
  slot: line;              (*monitorjeve spremenljivke*)
  full: boolean;

procedure send(m: line);        (*proceduri monitorja*)
when not full do
  begin slot:=m; full:=true end;
procedure receive(var m: line);
when full do
  begin m:=slot; full:=false end;

begin                    (*telo monitorja-inicializacija*)
  full:=false
end;


procedure producer;
var
  m: line;
begin
  ....
  produce(m);
  send(m);
  ....
end;
procedure consumer;
var m: line;
begin
  ....
  receive(m);
  consume(m);
  ....
end;

begin                                   (*glavni program*)
  cobegin
    producer; consumer
  coend
end.
```

Monitor je strukturirano orodje za medprocesno komunikacijo. Sestavljen je iz deklaracij globalnih spremenljivk in iz množice procedur, ki izvajajo določene operacije nad temi spremenljivkami. Procesi nimajo direktnega dostopa do globalnih spremenljivk, temveč samo preko klicanja monitorjevih procedur. Monitorjeve procedure imajo to lastnost, da lahko v nekem času samo en proces aktivno izvaja proceduro znotraj danega monitorja. Monitor ima tudi del, ki ga imenujemo telo in ki se uporablja za inicializacijo monitorjevih spremenljivk.

Monitorski koncept združuje dve ideji: vse kritične funkcije so zbrane na enem mestu in globalni podatki ter dostop do njih sta strukturirana.

Koncept monitorja omogoča, da prevajalnik testira, če se v programu nad deljenimi spremenljivkami izvajajo samo tiste operacije, ki so definirane v monitorju. Zgrajeni monitor lahko sistematsko in temeljito testiramo, potem pa prevajalnik preprečuje, da bi ga ostali programski moduli nepravilno uporabljali. Vsak proces s pripadajočimi lokalnimi spremenljivkami deklariramo kot poseben programski modul. Lokalne spremenljivke procesa morajo biti nedostopne drugim procesom (v našem primeru imata producer in consumer lokalno spremenljivko m). Prav tako testira prevajalnik pravilno uporabo lokalnih spremenljivk.

Testi pri prevajanju lahko nadomestijo teste pri izvajanju in zaščitne mehanizme v materialni opremi. Namen odpravljanje testov pri izvajanju ni samo doseganje učinkovitejše prevedene kode. Testi pri prevajanju preprečujejo nastop napak, testi pri izvajanju pa lahko samo sporočijo, zakaj se je sistem podrl. To je bistveno pri sistemih za delo v realnem času, ki opravljajo pomembne upravljalske funkcije.

Kot primer jezikov, ki temeljijo na monitorjih lahko navedemo Concurrent Pascal, ki so ga definirali in implementirali leta 1974 in Modula, ki so ga razvili leta 1977.

Jezikovni elementi so se razvijali v smer, kjer opis sinhronizacije med procesi podamo z opisom abstraktnega podatkovnega tipa. Pri zgoraj opisanem monitorju je na primer sinhronizacija med procesi v primeru praznega ali zapolnjenega izravnalnika implementirana v monitorjevih procedurah s stavkom "when pogoj do". Programer definira abstraktni podatkovni tip (izravnalnik) in dovoljene operacije nad njim. V opisu operacij je zajeta tudi potrebna sinhronizacija med procesi. Z uporabo takšnega podatkovnega tipa se izognemo opisu načina pravilne uporabe podatkovnega tipa.

Opis poti

Campbell in Habermann sta 1974 predlagala opis poti. To je abstraktni tip podoben monitorju, le da znotraj njega dodatno podamo izraz, ki določa zaporedje dostopa do podatkov. Oblika opisa poti je naslednja:

```
type
  ime = begin
          opis podatkovne strukture
          path opis zaporedja dostopa do podatkov end
          operation opis operacij nad podatki
        end
```

Izraz path opisuje dovoljeno zaporedje operacij nad podatki. V izrazu lahko opišemo tudi ponavljanje ali izbiro neke operacije.

## 2.5 Distribuirani sistemi

Doslej opisani mehanizmi za medprocesno komunikacijo so se večinoma zanašali na določeno implementacijo. čeprav izgleda, da višji programski jeziki z abstrakcijo prikrivajo razlike v implementaciji, implicitno predpostavljajo da se podatki med procesi prenašajo preko skupnega pomnilniškega območja. Na primer, vsak proces ima pri izvajanju monitorske procedure dostop do monitorjevnih spremenljivk. Pri tem mora biti zagotovljeno vzajemno izključevanje. To se da zagotoviti pri sistemih, ki imajo skupen pomnilnik, implementacija pa je težka pri distribuiranih sistemih, ki so povezani preko vhodnih in izhodnih linij, preko katerih si procesorji izmenjujejo sporočila. S pocenitvjo materialne opreme so se začeli takšni sistemi vse bolj uveljavljati.

Medprocesna komunikacija in sinhronizacija z rendezvous-jem

Hoare in Brinch Hansen sta 1978 leta pokazala, da je pri distribuiranih sistemih bolj naraven pristop k medprocesni komunikaciji, če sinhronizacijo in prenos podatkov med procesi jemljemo kot povezani dejavnosti. Osnovna ideja je prenos podatkov v določenih točkah posameznih procesov. Pred prenosom podatkov se oba procesa sinhronizirata. Ta način komunikacije imenujemo rendezvous. Na primer, če želi proces A poslati podatke procesu B, morata oba procesa soglašati s komunikacijo

tako, da pošljeta zahtevek za oddajanje oziroma za sprejem. če proces A prvi zahteva oddajanje podatkov, mora počakati, da proces B zahteva sprejem podatkov. če proces B prvi zahteva sprejem podatkov, mora počakati, da proces A zahteva oddajanje podatkov. Ko se procesa sinhronizirata, se podatki prenesejo in procesa nadaljujeta izvajanje.

Ta princip rendezvous-ja je izbran pri programskem jeziku Ada, za katerega so 1980 leta izdali standardni priročnik.

Ada nudi bogate možnosti za določanje in obvladovanje paralelnega izvajanja. Da dobimo predstavo o glavnih idejah, bomo v tem poglavju opisali nekatere možnosti. Zaradi jasnosti bomo ponekod podali samo nekatere oblike stavkov.

Za pojasnitev bomo vzeli primer omejenega izravnalnika za celoštevilčne podatke.

```
procedure PRODUCERCONSUMER is

      --specifikacija procesa izravnalnika
task BOUNDEDBUFFER is
  entry APPEND(V: in INTEGER); --deklaracija procesnih
  entry TAKE(V: out INTEGER);  --vhodov s parametri
end BOUNDEDBUFFER;             --konec specifikacije

      --specifikacija procesa, ki pise v izravnalnik
task PRODUCER;

      --specifikacija procesa, ki cita iz izravnalnika
task CONSUMER;

      --telo procesa izravnalnika
task body BOUNDEDBUFFER is
  SIZE: constant:= ...;
  B: array(0..SIZE) of INTEGER;
  INPTR,OUTPTR: INTEGER;
  N: INTEGER;
begin
  N:=0; INPTR:=0; OUTPTR:=0;   --inicializacija
  loop                         --omejenega izravnalnika
      --izbirni stavek
    select
      when N <= SIZE =>        --pogoj (varovalo)
        accept APPEND(V: in INTEGER) do  --prejemni st.
          B(INPTR):=V;         --zapisovanje v izravn.
        end APPEND;
        N:=N+1;
        INPTR:=(INPTR+1) mod SIZE;
    or
      when N > 0 =>            --pogoj
        accept TAKE(V: out INTEGER) do   --prejemni st.
          V:= B(OUTPTR);       --citanje elementa izrav.
        end TAKE;
        N:=N-1;
        OUTPTR:=(OUTPTR+1) mod SIZE;
    end select;                --konec izbirnega stavka
  end loop;
end BOUNDEDBUFFER;
      --telo procesa, ki pise v izravnalnik
task body PRODUCER is
  ELEM: INTEGER;
begin
  loop
    PRODUCE(ELEM);
    APPEND(ELEM);
  end loop;
end PRODUCER;
      --telo procesa, ki cita iz izravnalnika
task body CONSUMER is
  ELEM: INTEGER;
begin
  loop
    TAKE(ELEM);
    CONSUME(ELEM);
  end loop;
end CONSUMER;

      --glavni program
begin
  null;
end PRODUCERCONSUMER;
```

Prenos podatkov pri rendezvous-ju poteka preko parametrov in ne preko izravnalnika. Prenos podatkov med dvema asinhronima procesoma (v našem primeru PRODUCER in CONSUMER) zato izvedemo z uporabo dodatnega procesa (BOUNDEDBUFFER), v katerem programiramo omejeni izravnalnik. Rešitev s pasivnim monitorjem, ki se izvaja samo ko je klican, tu nadomestimo s procesom.

Procesi v Adi so programske enote, ki se izvajajo paralelno (task). Sestavljeni so iz specifikacije procesa in telesa procesa. Procesi se začnejo izvajati takoj po deklaraciji. Zato je v našem primeru glavni program lahko "null" (Ada uporablja "glavno proceduro" kot Pascal glavni program).

Komunikacija med procesi se izvaja na nadzorovan način preko procesnih vhodov (entry). V Adi je implementiran asimetričen rendezvous, pri katerem lahko ločimo kličoči proces (pri nas sta to PRODUCER in CONSUMER) in klicani proces (BOUNDEDBUFFER). Bistvo asimetričnosti je v tem, da v točkah rendezvous-ja samo kličoči proces navaja klicani proces oziroma njegove procesne vhode (APPEND, TAKE). Klicani proces ne navaja, kdo ga kliče, temveč izvede prejemni stavek (accept) na svoje vhode (APPEND, TAKE).

Oglejmo si obliko deklaracij in stavkov. Procesni vhodi so deklarirani v specifikaciji klicanega procesa:

```
entry identifikator [parametri_vhoda];
```

Pri tem so lahko parametri vhoda vhodni (in) ali izhodni (out).

V točki rendezvous-ja kličoči procesi navedejo ime vhoda klicanega procesa:

```
ime_vhoda [(dejanski_parametri)];
```

klicani proces pa navede prejemni stavek (accept):

```
accept ime_vhoda [parametri_vhoda]
  [do stavki
    end [identifikator]];
```

Asimetričnost omogoča, da procese razdelimo na uporabniške (kličoče) in servisne (klicane). Uporabniški procesi morajo poznati vhode servisnih procesov, ki jih kličejo. Servisnim procesom ni potrebno poznati uporabniških procesov. Prejemni stavek servisnega procesa namreč ne navaja imena uporabnika. Zato lahko zgradimo knjižnice servisnih procesov, ki jih lahko uporabljajo vsi uporabniki. Uporabniškim procesom zaradi njihove narave lahko rečemo tudi aktivni procesi, servisnim pa pasivni procesi.

Ada ima na tem področju še dodatno prednost: ker ima klic vhoda enako sintakso kot klic procedure, je lahko klicana operacija izvedena kot proces ali kot navadna sekvenčna procedura.

Na primeru izravnalnika vidimo, da mora imeti proces možnost rendezvous-ja na različnih vhodih, pri tem pa ne pozna vrstnega reda klicanja posameznih vhodov. Ne moremo namreč vnaprej predvideti zaporedja pisanja v izravnalnik in čitanja iz izravnalnika. Potrebujemo možnost nedeterminističnega prenosa podatkov. V Adi nam to možnost nudi izbirni stavek (select), ki omogoča alternativno izbiro med različnimi prejemnimi stavki.

Problem predstavlja tudi omejena velikost izravnalnika. če je izravnalnik poln, ne more sprejeti novega elementa. če je prazen, ne more oddati elementa. Zato pred posamezno alternativo v izbirnem stavku kot varovala postavimo pogoje (when). Takšen izbirni stavek z varovali in izbirami med alternativnimi prejemnimi stavki ima obliko:

```
select
  [when pogoj =>]
     alternativa s prejemnim stavkom
or
  [when pogoj =>]
     alternativa s prejemnim stavkom
.....

[else
     stavki]
end select;
```

Ob izvajanju izbirnega stavka se najprej ovrednotijo vsi pogoji in tako določijo odprte alternative. če je klican nek prejemni stavek v odprtih alternativah, se rendezvous izvede. če ni odprte alternative ali ni klicanih prejemnih stavkov odprtih alternativ in obstaja stavek else, se izvedejo pripadajoči stavki. če obstajajo odprte alternative, noben pripadajoči prejemni stavek pa ni klican in ni stavka else, proces počaka na zahtevo po randezvous-ju. če pa ni odprte alternative ne stavka else nastopi napaka.

Za raznovrstne probleme paralelnega programiranja mehanizem rendezvous-ja nudi primernejše rešitve od mehanizmov na osnovi vzajemnega izključevanja (kritične sekcije, semaforji, monitorji), ki so bolj prilagojeni reševanju določenih problemov. Rendezvous je primeren za implementacijo na enoprocesorskih sistemih, multiprocesorskih sistemih s skupnim pomnilnikom in distribuiranih multiprocesorskih sistemih. Ugotovimo, da rešitve z uporabo rendezvous-ja ponavadi zahtevajo več procesov, kot pri uporabi drugih mehanizmov. Procesi se zato pogosteje izmenjujejo, za kar se porabi določen čas. Pri našem primeru proizvajalca in potrošnika smo tako namesto izravnalnika, implementiranega s pasivnim monitorjem, pri uporabi rendezvous-ja vstavili servisni proces, ki vrši funkcijo izravnalnika. Pri uporabi novejših procesorjev je čas preklopa med procesi tako hiter, da ni kritičen.

## 2.6 Zaščitni mehanizmi

Videli smo, da monitor lahko definiramo kot abstraktni podatkovni tip. V monitorju so opisane podatkovne strukture in procedure, ki so edine dovoljene operacije nad podatki. V procedurah je zajeta tudi pravilna sinhronizacija med procesi. Procesi kličejo procedure monitorja, pri tem pa nimajo vpliva in jih tudi ne zanima način izvajanja operacij, temveč samo učinek operacij na podatkovne predmete. Vidimo, da monitor

deluje kot nekakšna zaščita podatkovnega predmeta. Pri uporabi izrazov za opis poti smo videli, kako lahko definiramo tudi dovoljeno zaporedje izvajanja operacij nad podatkovnimi predmeti.

Razvoj jezikovnih konceptov se je nadaljeval v smeri abstrakcije podatkovnih predmetov. Definiramo lahko splošni mehanizem za zaščito. Zaščitni mehanizem podatkovnega predmeta je proces, sestavljen iz skupine podatkov, ki jih ščiti in množice procedur za dostop do teh podatkov. Drugi procesi lahko izvajajo operacije nad podatki s klicanjem procedur procesa zaščitnega mehanizma preko njegovih procesnih vhodov.

Pri uporabi zaščitnega mehanizma lahko prevajalnik ugotavlja, ali procesi izvajajo nad podatkovno strukturo samo tiste operacije, ki jih mehanizem dovoljuje. Med izvajanjem programa zaščitni mehanizem izvede, zakasni ali zavrne operacije, ki niso takoj izvedljive zaradi dinamike celotnega sistema.

Vsak mehanizem za komunikacijo in sinhronizacijo med procesi, na primer monitor, izvaja ali zakasni zahtevane operacije. Enak moritorski klic bo vedno sprožil izvajanje enake procedure.

Zaščitni mehanizem lahko izvaja tudi aktivno filtriranje vhodnih klicov. Proces zaščitnega mehanizma izvaja nek program in ima lahko tudi lastne spremenljivke za shranjevanje stanja sistema. V različnih točkah program sprejema klice ostalih procesov na procesnih vhodih. Pri sprejemu klicev pa zaščitni mehanizem lahko izvede različne operacije nad podatkovnim predmetov, odvisno od tega, v katerem delu programa je klic sprejet in v odvisnosti od stanja sistema.

Pri uporabi jezika Ada lahko z elementi jezika zgradimo zaščitni mehanizem nekega podatkovnega predmeta na popoln način, tako da za ta namen ne potrebujemo kakšnega posebnega jezika.

Kot primer lahko vzamemo implementacijo omejenega izravnalnika, ki smo ga podali pri opisu rendezvous-ja. Formirali smo servisni proces (BOUNDEDBUFFER), v katerem smo programirali omejeni izravnalnik, ki ga uporabljamo za prenos podatkov med dvema asinhronima procesoma. Ta servisni proces je hkrati zaščitni mehanizem omejenega izravnalnika. Program procesa je neskončna zanka, v kateri se izvaja izbirni stavek (select). Znotraj izbirnega stavka se lahko izvedeta dva prejemna stavka (APPEND, TAKE) z rendezvous-jem. Ta prejemna stavka sta procesna vhoda. Pri rendezvous-ju na nekem procesnem vhodu ustrezni prejemni stavek izvede določeno operacijo nad omejenim izravnalnikom. To je edini način dostopa drugih procesov do izravnalnika. Če bi bilo potrebno, bi znotraj servisnega procesa zapisali enak prejemni stavek na različnih mestih programa, pri čemer bi se ob posameznem rendevous-ju izvedle različne operacije nad podatkovnim predmetom.

## 3 Zaključek

Paralelno programiranje uvedemo zato, da enostavneje in hitreje rešimo neko nalogo, ki ima možnost paralelnega izvajanja in da zvečamo izkoriščenost sistema. Če paralelni procesi med sabo sodelujejo, moramo zagotoviti pravilen način interakcije med procesi. Uporabimo lahko več ustreznih primitivov, kot so kritične sekcije, semaforji, monitorji, rendezvous-ji itd. Več jezikov vsebuje jezikovne konstrukte za deklariranje procesov in medprocesno komunikacijo in sinhronizacijo z opisanimi primitivi. Večina takšnih jezikov odraža stanje materialne opreme v času razvoja in podpira primitive (semaforji, kritične sekcije, pogojne kritične sekcije, monitorji), primerne za implementacijo na enoprocesorskih sistemih ali večprocesorskih sistemih s skupnim pomnilnikom. Takši jeziki so na primer Concurrent Pascal in Modula. Pri distribuiranih večprocesorskih sistemih, pri katerih so posamezni procesorji povezani preko vhodno-izhodnih linij, pa je nastal problem implementacije teh primitivov. V novejšem jeziku Ada so komunikacijo in sinhronizacijo med procesi povezali v mehanizmu rendezvous-ja, ki ga lahko enostavno implementiramo tudi v distribuiranih sistemih.

Uporaba višjih programskih jezikov poenostavi in skrajša čas razvoja programov. Pri tem nastopi problem učinkovitosti prevedene kode. Večina današnjih arhitektur ne podpira učinkovito abstraktne jezike v primerjavi s strojno kodo. Pri programih, ki morajo delati v realnem času smo zato soočeni z izbiro med učinkovitostjo, ceno in zanesljivostjo. Zato je prisoten trend razvijanja računalniških arhitektur, ki direktno podpirajo koncepte programskih jezikov. V procesorskih elementih se z materialno opremo oziroma mikroprogrami implementirajo specifične funkcije jezikov za paralelno izvajanje procesov: preklapanje procesorja med procesi, medprocesna komunikacija in sinhronizacija in podobne. S tem se občutno zveča hitrost izvajanja teh funkcij. Takšne funkcije ima na primer VAX procesor. Po drugi strani razvijajo sisteme z več procesorji, pri katerih posamezni procesi tečejo na lastnih procesorjih, s čemer se zveča hitrost izvajanja paralelnih programov. Takšno izvajanje na primer podpira arhitektura sistema iAPX 432.

Lahko povzamemo, da so nam danes na razpolago jeziki z jezikovnimi elementi, primernimi za paralelno programiranje, razvijajo pa se arhitekture, ki podpirajo učinkovito implementacijo jezikov na sistemih z več procesorskimi elementi.

Pri obravnavanju računalniških jezikov smo v članku upoštevali imperativne jezike. Programi v funkcionalnih jezikih vsebujejo visoko stopnjo inherentne paralelnosti, načine automatskega izkoriščanja in učinkovite implementacije te paralelnosti na večprocesorskih sistemih pa še raziskujejo.

Literatura


/1/ M. Ben-Ari, "Principles of Concurrent Programming", Prentice/Hall International, London, 1982

/2/ R.C. Holt, G.S. Graham, E.D. Lazowska, M.A. Scott, "Structured Concurrent Programming with Operating Systems Applications", Addison-Wesley, Reading, Mass., 1978

/3/ P. Brinch Hansen, "A Keynote Address on Concurrent Programming", IEEE Computer, May 1979, pp. 50-56

/4/ D.A. Anderson, "Operating Systems", IEEE Computer, June 1981, pp. 69-82

/5/ S.J. Young, "Real Time Languages: Design and Development", John Wiley & Sons, New York, 1982

/6/ I.C. Pyle, "The Ada Programming Language", Prentice/Hall International, 1981

/7/ M. Exel, F. Prijatelj, "Programiranje sprotnih in vgnezdenih sistemov: procesi v Adi", Informatica, No. 2, 1981, pp. 8-18

/8/ G. Štrkić, D. Novosel, "O jezicima za konkurentno programiranje kao sredstvu za projektovanje upravljačkih sistema", Informatica, No. 2, 1985, pp. 16-18

/9/ M. Kapus, "Pregled jezikovnih elementov za opis sinhronizacije paralelnih procesov", Informatica, No. 1, 1982, pp. 71-77

/10/ E.T. Fathi, M. Krieger, "Multiple Microprocessor Systems: What, Why and When", IEEE Computer, March 1983, pp. 23-32

/11/ J.L. Hennessy, "VLSI Processor Architecture", IEEE Trans. on Computers, Vol. C-33, No. 12, December 1984, pp. 1221-1246