UDK 681.3.02

Jože Rugelj
Institut »Jožef Stefan«

Sinhronizacijski mehanizmi v porazdeljenih računalniških sistemih so potrebni za definicijo in realizacijo urejenosti dogodkov v računalniškem sistemu. Članek podaja pregled mehanizmov za sinhronizacijo na različnih nivojih abstrakcije računalniškega sistema.

Synchronization mechanisms in distributed computing systems are necessary for the definition and the realization of event ordering in the computing system. This article gives an overview of mechanisms, used on different levels of abstraction.

1. UVOD

Porazdeljen računalniški sistem je množica procesnih elementov. Vsak procesni element ima lasten pomnilnik in procesne zmogljivosti. V procesnih elementih se izvajajo procesi. Z oznako porazdeljenost je mišljena porazdelitev nadzora in ne nujno tudi prostorska porazdeljenost. V tesno sklopljenih sistemih so procesi povezani med seboj s skupnim pomnilnikom, v šibko sklopljenih sistemih pa so procesna mesta povezana s komunikacijskimi kanali, ki skupaj tvorijo komunikacijsko mrežo. Skupnega pomnilnika v takih sistemih ni. V tem primeru gre tudi za prostorsko porazdelitev procesnih elementov, ki jim zaradi tega rečemo tudi procesna mesta.
Procesne elemente združuje v porazdeljen sistem porazdeljen operacijski sistem. Glavne naloge porazdeljenega operacijskega sistema so podpora medprocesne komunikacije, dodeljevnje virov in upravljanje z njimi, upravljanje z imeni ter reševanje iz napak. Jedra porazdeljenega operacijskega sistema na posameznih mestih so lahko implementirana kot jedra osnovnega operacijskega sistema na tem mestu ali kot procesi na uporabniškem nivoju /Trip87/.
V sistemih z večimi procesnimi elementi se procesi izvajajo sočasno, dokler ne potrebujejo medsebojnih stikov. Načrtovane in vodene stike med procesi imenujemo procesna komunikacija in sinhronizacija. Procesi morajo sodelovati zaradi omejevanja sočasnosti in zaradi medsebojnega razvrščanja /Verj83/. Odnose med procesi bi glede na način sodelovanja lahko razdelili v dve glavni kategoriji: lahko tekmujejo med seboj ali so v odnosu proizvajalec-potrošnik /Hwan85/.

2. KONSISTENTNOST IN NEDELJIVOST

Predpostavljamo, da se procesi izvajajo v diskretnih korakih in na vsakem koraku generirajo dogodek. Dogodek je lahko lokalen procesu in ga drugi procesi ne opazijo ali pa je viden vsem in vsebuje problem sinhronizacije.

Na vsakem nivoju abstrakcije lahko proces, ki se izvaja na nekem procesnem mestu, sproži operacije. Operacija na nivoju j je implementirana kot množica aktivnosti na nivoju i, pri čemer velja i < j. Kar vidimo z nivoja j kot aktivnost je definirano na nivoju i kot operacija. Tak model velja rekurzivno za vse nivoje.

Operacije oziroma aktivnosti vodijo in upravljajo vire. Viri so predstavljeni kot podatkovni objekti. Na primer, periferna naprava je lahko predstavljena s svojim trenutnim stanjem, ki je določeno z vrednostmi množice parametrov. Zapis na datoteki lahko predstavimo z njegovo vsebino. Primeri aktivnosti, ki delajo s podatkovnimi objekti, so pisanje, branje, kreiranje in brisanje.
Podatkovni objekti imajo med seboj semantične povezave, to pomeni, da morajo njihove vrednosti zadoščati nekim omejitvam. Ko objekt kreiramo, brišemo ali vanj pišemo, je pogosto potrebno kreirati, brisati ali pisati še v druge objekte, da bi zadostili konsistenčnim omejitvam /LeLa81/.

Vsaka operacija je definirana tako, da predpostavlja na vhodu množico objektov s konsistentnimi vrednostmi in zagotavlja kot rezultat množico novih vrednosti, ki so tudi konsistentne.
Torej, če so konsistentna stanja, ki se prenašajo med procesi ali so shranjena, je tudi procesiranje konsistentno.

Če ni posebnih predpostavk, lahko ohranimo konsistentnost samo z zagotavljanjem nedeljivosti operacij. Z definiranjem nekaterih drugih zahtev lahko zahtevo za nedeljivost opustimo. Definirajmo nedeljivost operacije. Operacija je nedeljiva, če zadošča naslednjima pogojema:

1. ali se izvedejo vse aktivnosti popolnoma ali pa se ne izvede nobena in

2. vmesna stanja pri izvajanju operacije

niso vidna nobeni drugi operaciji.

Nedeljivost je dobro poznan koncept /Lmps81/.

Običajne instrukcije na računalniku so nedeljive, ker se izvajajo strogo zaporedno na eni procesni enoti. V porazdeljenih sistemih se aktivnosti, ki pripadajo dani operaciji, lahko izvajajo na različnih procesnih enotah brez določenih časovnih povezav. Zato implementacija nedeljivih operacij zahteva specifične mehanizme, ki jih v centraliziranih sistemih ne potrebujemo.

Začetek izvajanja operacije šele po izvršitvi prejšnje operacije vodi k posebni vrsti dodeljevanja imenovanega zaporedno dodeljevanje (serial scheduling). V porazdeljenih sistemih pa je izključna uporaba zaporednega dodeljevanja zelo neučinkovita. Če aktivnosti, ki jih sproži dana množica operacij, delujejo nad različnimi objekti, vzporedno izvajanje aktivnosti ni le možno, ampak celo priporočljivo. Še več, če so aktivnosti, ki delujejo nad danim objektom in pripadajo različnim operacijam, sprožene sočasno in postavljene v čakalno vrsto procesnega elementa, ki vsebuje objekt, je nelzkoriščen čas med dvema zaporednima aktinostima krajši kot v primeru, če je naenkrat sprožena samo ena aktivnost.

Torej je zaželjeno, da dovoljujemo prepletanje aktivnosti (interleaving), kolikor je to mogoče, in s tem optimiziramo zmogljivosti. Odgovarjajoča dodeljevanja imenujemo dodeljevanja s prepletanjem in nekatera od njih so ekvivalentna zaporednim, torej so konsistentna. Toda v splošnem to ne velja. Če torej želimo čim večjo paralelnost in s tem veliko hitrost in dobro izkoriščenost virov moramo poiskati ustrezno prepletanje, ki ohranja konsistentnost. S sinhronizacijskimi mehanizmi pa lahko realiziramo potrebne odnose med procesi oz. operacijami.

3. RAZVRŠČANJE DOGODKOV

Določanje, kateri dogodek se je zgodil prej, je običajno intultivno zasnovano na razmišljanju v fizikalnem časovnem svetu. Tak pristop predpostavlja, da lahko definiramo nek univerzalen čas, ki je dosegljiv z različnih lokacij v sistemu. Vemo pa, da taka rešitev ne obstaja. V praksi lahko dobimo približke univerzalnega časa z dano natančnostjo. Vendar pa ni potrebno ali pa ni mogoče izraziti sistemskih specifikacij s fizikalnim časom /Kope87/.

Obstaja določena kronološka urejenost dogodkov v sistemu, ker procesi, ki generirajo te dogodke, spoštujejo specifična pravila (algoritme, protokole), ki izražajo relativno urejenost v množici dogodkov opazovanega sistema.

Taka urejenost dovoljuje procesom, ki opazujejo dogodke, da pravilno implementirajo sistemske aktivnosti. Glede na naravo teh aktivnosti je potrebna delna ali popolna urejenost dogodkov.

Delno urejenost, označeno z " --> " (beri "se zgodi pred") lahko definiramo nad katerokoli množico dogodkov, ki jih generirajo procesi, ki izmenjujejo sporočila /Lamp78/.
Za relacijo "se zgodi pred" velja:

1. če sta a in b aktivnosti istega procesa in se a zgodi pred b, potem je a --> b
2. če je a aktivnost za pošiljanje sporočila iz enega procesa in je b aktivnost, ki sprejme to sporočilo v drugem procesu, velja a --> b
3. če je a --> b in b --> c, potem velja a --> c

Če procesi komunicirajo med seboj z izmenjavanjem sporočil, lahko razvrstitimo nekatere dogodke, ki se zgodijo v različnih procesih. Urejenost " --> " je samo delna. Npr. če imamo dogodka a in b in velja a --> b in b --> a, potem je nemogoče reči, kateri od dogodkov se je zgodil prej. Za take dogodke rečemo, da so sočasni (concurrent). V splošnem dogodkov iz procesov, ki med seboj ne komunicirajo, ne moremo razvrstiti. V odvisnosti od omejitev, ki jih moramo upoštevati, je to dopustno ali pa tudi ne. Eden od načinov za dosego popolne urejenosti, kadar je le-ta nujna, je pridobitev popolne urejenosti iz delne z definicijo popolne urejenosti nad množico procesov.

Omeniti moramo enega od osnovnih problemov računalniških sistemov, to je problem zakasnitev pri procesiranju. V enoprocesorskih sistemih lahko zelo natančno določimo čas, ki ga procesna enota potrebuje za izvršitev danega ukaza. Seveda pa v splošnem operacijski sistemi niso zasnovani na osnovi takih podatkov. Še manj pa bi bila taka zasnova upravičena za porazdeljene operacijske sisteme, saj zakasnitve vključujejo še čas, ki je potreben za prenos podatkov in ukazov med posameznimi procesnimi enotami po komunikacijski mreži. V porazdeljenih sistemih, kjer se operacije izvajajo na različnih procesnih enotah, spreminjanje zakasnitev v komunikacijskem podsistemu lahko zmoti določeno urejenost dogodkov, ki jo pričakujemo. To se lahko zgodi celo v sistemih, kjer so zakasnitve stalne. Zato moramo uporabljati določene sinhronizacijske mehanizme pri procesih, kjer se dogodki zgodijo in pri procesih, ki opazujejo te dogodke.

Namen sinhronizacijskih mehanizmov je definiranje in realizacija razvrstitve poljubne množice dogodkov. Natančneje, rekli bomo, da je sinhronizacija način za definicijo in realizacijo delne all popolne urejenosti nad neko množico dogodkov.

Sinhronizacijskih mehanizmi nudijo procesom pripomočke, s katerimi se sistem ohranja v konsistentnem stanju. Stanje računalniškega sistema je konsistentno, če v vsakem trenutku ustreza nekim zunanjim določilom.

4. SPLOŠNE ZNAČILNOSTI SINHRONIZACIJSKIH MEHANIZMOV

Sinhronizacijski mehanizmi temeljijo na opazovanju in spreminjanju določenih skupnih sinhronizacijskih spremenljivk. V tesno sklopljenih sistemih so le-te v skupnem pomnilniku, v šibko sklopljenih sistemih pa so na nekaterih all na vseh procesnih mestih. Zahteve za branje ali spreminjanje spremenljivk in njihove vrednosti se prenašejo kot sporočila po komunikacijski mreži.

Pri porazdeljenih rešitvah so sinhronizacijske spremenljivke podvojene, razcepljene ali porazdeljene med mesta v sistemu.
Podvojenost ali celo pomnoženost spremenljivke pomeni, da so verzije iste spremenljivke v vseh ali vsaj v večih mestih v sistemu. Ustrezen mehanizem poskrbi za delno ali popolno konsistentnost v vsakem trenutku.
Razcepljenost spremenljivk je razdelitev vrednosti spremenljivke na več komponent. Komponente so porazdeljene po mestih, kjer se spreminja njihova vrednost. Prava vrednost spremenljivke je linearna kombinacija vrednosti njenih komponent.
Porazdelitev spremenljivk pa je rešitev, pri kateri so sinhronizacijske spremenljivke porazdeljene po različnih mestih v sistemu, vendar samo po ena verzija vsake.

Za izvedbo operacij nad sinhronizacijskimi spremenljivkami na katerem koli nivoju abstrakcije se lahko sklicujemo na določen osebek, ki ga imenujemo centralni sinhronizacijski osebek. Le-ta je dostopen vsakemu proizvajalcu vsakič, ko začne novo operacijo.
Termin "sinhronizacijski osebek" uporabljamo za usklajevalce procesov , semaforje, monitorje ipd.; to so torej aktivnosti, ki popravljajo in spremljajo stanje sinhronizacijskih spremenljivk.

Sinhronizacijski osebek je centralen, če velja:

- da ima edinstveno ime, ki ga poznajo vsi procesi, ki se sinhronizirajo med seboj,
- katerikoli od teh procesov ima dostop do sinhronizacijskega osebka v vsakem trenutku /LeLa81/.

Nekateri sistemi so zgrajeni tako, da preživijo napake, ki se pojavijo pri centralnem sinhronizacijskem osebku. Predvidene so tehnike za reševanje ob napakah, ki izberejo nov sinhronizacijski osebek, ko pride do napake.

Vsak sinhronizacijski mehanizem, ki temelji na centralnem sinhronizacijskem osebku imenujemo centraliziran.

Ostale mehanizme imenujemo porazdeljene.

Sinhronizacijski mehanizmi so lahko realizirani v elementih strojne opreme računalnika, kot primitivi v programskih jezikih ali v operacijskih sistemih.

5. ELEMENTI STROJNE OPREME ZA PODPORO SINHRONIZACIJE

Eden od najbolj enostavnih mehanizmov, ki omogočajo nedeljivost in medsebojno izključevanje procesov je onemogočanje prekinitev procesorja. Ta rešitev je bila uporabljena že na enoprocesorskih sistemih, ki so delovali na principu kvazi-paralelnosti. Na porazdeljenih sistemih nima posebnega pomena, saj procesorji med seboj ne morejo izvajati takih ukazov /Fink86/.

V tesno sklopljenih sistemih, kjer procesorji opazujejo in popravljajo vrednosti spremenljivk, je za ohranjanje konsistentnosti nujno zagotavljanje nedeljivosti branja in pisanja vrednosti spremenljivk glede na prebrano vrednost. Primera takih ukazov sta test-and-set in compare-and-swap /Hwan85/. Taki ukazi omogočajo realizacijo kompleksnejših sinhronizacijskih mehanizmov na višjem nivoju.

Obstaja še drug sinhronizacijski primitiv, ki je običajno realiziran v strojni opremi in dopušča določeno stopnjo sočasnosti pri uveljavljanju zaporednosti dostopa do skupnega pomnilnika. Imenuje se fetch-and-add. Primitiv ima obliko F&A(X,e) in vrne vrednost spremenljivke X ter poveča njeno vrednost za e. Če se izvede več takih ukazov hkrati, se vrednost spremenljivke poveča naenkrat za vsoto vseh e, vsak proces pa dobi vrnjeno vrednost, kot da bi se ukazi izvajali v naključnem vrstnem redu zaporedno.

V šibko sklopljenih sistemih uporabljajo kot osnovo za realizacijo nekaterih višjenivojskih mehanizmov posebne števnike, imenovane fizične ure. /Kope87/ predstavlja posebno časovno sinhronizacijsko enoto, ki poleg podatkov o realnem času vsebuje tudi mehanizme za usklajevanje in popravljane časovne baze, zasnovane na /Lamp78/. S tako enoto razbremenimo procesor, ki ga je izvajanje sinhronizacije preveč obremenilo.

6. SINHRONIZACIJSKI PRIMITIVI V PROGRAMSKIH JEZIKIH

Pomembna lastnost aplikacijske in sistemske programske opreme, ki povečuje preglednost in zmanjšuje kompleksnost, je njena modularnost. Navzven porazdeljen sistem tako izgleda kot množica programskih modulov , kjer je vsak modul zase enostaven zaporeden proces.
Procesi v tesno sklopljenih sistemih sodelujejo med seboj tako, da komunicirajo preko skupnih spremenljivk.

Osnovna rešitev problema medsebojnega izključevanje procesov, ki komunicirajo preko skupnih spremenljivk, je Dekkerjev algoritem /BenA83/. V njem so združene vse dobre lastnosti enostavnejših rešitev, kot so aktivno čakanje z opazovanjem skupne spremenljivke (busy-waiting) in uporabe spremenljivke za izmeničen dostop (switch-variable). Hkrati pa odpravlja pomanjkljivosti, zaradi katerih so te rešitve v praksi neuporabne. Mislimo predvsem na neodslednost pri medsebojnem izključevanju in na veliko odvisnost med procesi pri uporabi takega mehanizma.
Dekkerjev algoritem je precej kompleksen in ni primeren za implementacijo.

Veliko bolj enostavna rešitev je semafor /Dijk68/, ki ga je lahko implementirati in je dovolj močan, da lahko z njim elegantno rešimo probleme medsebojnega izključevanja in razvrščanja procesov. Dobra lastnost semaforjev, ki jo prej omnejeni mehanizmi nimajo, je tudi to, da so procesi, ki čakajo na doseg do določenega vira, blokirani. Zato medtem, ko čakajo, sprostijo procesne zmogljivosti za druge procese. Semaforje lahko uporabimo tudi pri implementaciji močnejših primitivov.
Semafor s je strogo pozitivna celoštevilska spremenljivka, ki ji je pridružena še vrsta

procesov.
Definirani sta dve operaciji nad semaforjem, P(s) in V(s), ki sta nedeljivi. Operacija P(s) zmanjša vrednost s za ena, če je s > 0, sicer pa odloži proces, ki je izvedel to operacijo v vrsto. Operacija V(s) pa zbudi prvi proces v vrsti, če pa je vrsta prazna, poveča vrednost s za ena.

Kritična področja odpravljajo edino slabo lastnost semaforjev, to je velika možnost za napake pri njihovi uporabi. Nepravilna razvrstitev operacij P in V je lahko usodna. Kritična področja so deli programa, kjer proces dosega skupne spremenljivke, ki so v tistem času nedostopne drugim procesom. Začetek in konec kritičnega področja označujejo posebni jezikovni konstrukti, ki so zelo enostavni za uporabo. Operacijski sistem potem sam poskrbi za dejansko realizacijo medsebojnega izključevanja.

Pogojna kritična področja so razširitev prejšnjega konstrukta. Vstop v krtično področje je dovoljen šele, ko je izpolnjen nek pogoj. Ovrednotenje pogoja in izvedba kritičnega področja sta nedeljiva.

Hoare in Brinch Hansen sta definirala monitor /Hoar74/, /BrHa73/. Podobno kot proces predstavlja koristno abstrakcijo pri mutiprogramiranju je monitor abstrakcija za medprocesno komunikacijo.
Monitor je razširitev pogojnih kritičnih sekcij. Monitor predstavlja telo, ki vsebuje skupne spremenljivke in procedure s *kritičnimi sekcijami za delo z njimi. S tem* postanejo te spremenljivke lokalne, skrite znotraj monitorja. Procesi, ki želijo dostop do takih spremenljivk, ga lahko dobijo samo preko monitorskih procedur. Monitor je pasiven v sistemu in se aktivira samo takrat, ko procesi želijo dostop do njegovih spremenljivk.

Pri šibko sklopljenih sistemih pa je sodelovanje med procesi v različnih programskih modulih povezano s pošiljanjem in sprejemanjem sporočil, ki služijo za sinhronizacijo in prenašanje podatkov /Slom87/. Komunikacija med procesi je lahko enosmerna ali dvosmerna.

Osnovna primitiva pri enosmerni komunikaciji sta 'pošlji' in 'sprejmi', pri dvosmerni pa zahtevek z odgovorom (request-reply), oddaljeni klic procedure (remote procedure call) in rendezvous.

Primitive za sinhrono sprejemanje podatkov najdemo v večini jezikov, ki so primerni za porazdeljene sisteme (ADA /USAD80/, CONIC /Slom85/, CSP /Hoar78/, SR /Andr81/, Pascal-m /Abra83/).
Modul, ki čaka na sprejem sporočila od drugega modula se blokira ter postavi v vrsto čakajočih in se aktivira šele po prejemu pričakovanega sporočila.
CSP ima podoben konstrukt tudi za pošiljanje sporočil.

Vsi dvosmerni primitivi omogočaju sinhronizacijo med procesi, ki jih izvajajo, čeprav se v podrobnostih in načinih izvedbe nekoliko razlikujejo.
Zahtevek z odgovorom je dvosmerni primitiv, ki je v bistvu kombinacija sinhronega pošiljanja in sprejema sporočila.

Oddaljeni klic procedure ima določene prednosti, saj omogoča transparentnost lokacije virov in procesov /Stan82/. To pomeni, da uporabniku ni treba vedeti, kje se nahajajo iskani viri niti mu ni treba sestavljati sporočil. Vse to zna narediti operacijski sistem.
Konstrukt v jeziku Ada /USAD80/, imenovan rendezvous, je kombinacija RPC in izmenjavanja sporočil. Omogoča sinhronizacijo procesov v času izvajanja konstrukta /BenA82/.

7. SINHRONIZACIJSKI MEHANIZMI V OPERACIJSKIH SISTEMIH

7.1. Centralizirani sinhronizacijski mehanizmi

Skupna lastnost centraliziranih sinhronizacijskih mehanizmov je v tem, da temeljijo na enem sinhronizacijskem osebku.

Fizična ura predstavlja osnovo za razvrščanje dogodkov podobno kot v centraliziranih sistemih. Operacijski sistem za realizacijo tega mehanizma uporablja posebne elemente strojne opreme. Procesi so razvrščeni na osnovi časovnih značk, ki jih dobijo, kadar je potrebna sinhronizacija.
Čeprav je ta metoda enostavna, ima mnogo pomanjkljivosti. Pravilno zaznamovanje dogodkov s časovnimi značkami je popolnoma odvisno od sprejema stanja ure ob vsakem dogodku. Napaka pri prenosu sporočila s tem podatkom je lahko usodna za pravilno razvrstitev. Potrebujemo tudi vnaprejšnje točno poznavanje zakasnitev v prenosnih kanalih. Natančnost je odvisna od zahtev sistema oziroma aplikacije.

Števec dogodkov je objekt, ki šteje dogodke, ki so se zgodili v določenem razredu (npr. aktivnosti). Definirani so trije primitivi: povečaj vrednost števca, preberi vrednosti števca in odloži klicoči proces, dokler ni vrednost števca vsaj enaka podani konstanti.
Pomembna prednost tega mehanizma je v tem, da dovoljuje sočasno izvajanje teh primitivov na istem števcu brez medsebojnega izključevanja. Pojem "števec z enim upravljalcem" je definiran kot števec dogodkov, ki deluje pravilno, dokler je prepovedano sočasno povečanje vrednosti števca. Vendar pa je sistem zelo občutljiv na izpad posameznih elementov, saj je vsaka napaka števca usodna za pravilno razvrščanje.

Statični razvrščevalnik uporabljamo za popolno razvrstitev dogodkov v danem razredu (števec dogodkov omogoča samo delno urejenost). Razvrščevalnik je celoštevilčna spremenljivka. Za delo z razvrščevalnikom je definiran samo en primitiv, imenovan vstopnica (ticket), ki vrne tekočo vrednost razvrščevalnika in poveča njegovo vrednost za 1. Razvrščevalnik zahteva ločen mehanizem za medsebojno izključevanje zato, da je primitiv vstopnica nedeljiv. Dva proizvajalca ne moreta dobiti vstopnice hkrati. Glavni problemi pri uporabi razvrščevalnika so pri izbiri mehanizma za medsebojno izključevanje in vpliv napak ali izpada razvrščevalnika na preživetje sinhronizacijskega mehanizma.

Jasno je, da centralizirani pristopi v porazdeljenih sistemih ne izpolnjujejo zahtev, ki so bistvene za take sisteme. Sistemi, zgrajeni na takih osnovah, ne morejo imeti visoke stopnje razpoložljivosti in imajo v splošnem slabše zmogljivosti zaradi

ozkega grla, ki ga predstavlja centralna enota.

7.2. Porazdeljeni sinhronizacijski mehanizmi

S porazdeljenimi mehanizmi dosegamo večjo stopnjo paralelnosti in s tem hitrejše delovanje, boljšo izkoriščenost opreme in večjo zanesljivost. Porazdeljeni sinhronizacijski mehanizmi so večkratne fizične ure, večkratne logične ure, abstraktni izrazi, skupne spremenljivke, krožeči žeton in krožeči razvrščevalnik.

Cilj uporabe fizičnih ur je v določitvi enotnega fizičnega časa v sistemu. Konsistentno dodeljevanje lahko dobimo iz popolne kronološke razvrstitve aktivnosti, ki se pojavljajo v sistemu. Pri uporabi več ur ni pomembna samo točnost vsake posamezne ure ampak tudi medsebojna usklajenost, tako da je razlika med poljubnima dvema urama manjša od vnaprej določene konstante. Rešitev tega problema je podal Lamport /Lamp 78/.
Sistem modeliramo kot povezan graf procesov s premerom d. Premer grafa predstavlja minimalno število procesnih mest, preko katerih mora iti sporočilo iz poljubnega procesnega mesta, da doseže poljubno drugo procesno mesto. Vsak proces ima uro in periodično (perioda t) pošilja sinhronizacijska sporočila vsekemu drugemu procesu. Vsako sinhronizacijsko sporočilo vsebuje fizično časovno značko. Po prejemu sinhronizacijskega sporočila proces pomakne svojo uro naprej, če je časovna značka večja od stanja ure. Predpostavljamo, da poznamo spodnjo mejo (u) in zgornjo mejo (u + z) zakasnitev na komunikacijski mreži. Naj bo k natančnost vsake ure ($k<10^{-6}$) in e dovoljen zamik med poljubnima dvema urama. Če je $e/(1-k) < u$ in $e << t$, potem je možno izračunati približno vrednost e, ki je približno d(2kt + z).
V odvisnosti od zahtev glede relativnih zamikov in veljavnosti predpostavk glede zakasnitev na komunikacijski mreži se lahko odločimo za tveganje, da bomo občasno izgubili sporočila zaradi prevelikih zakasnitev in tako dosegli verjetnostno sinhronizacijo ali pa ne bomo tvegali. Tak pristop bistveno zmanjša zmogljivosti sistema. Ključni parameter pri tem je razmerje z/u.

Večkratne logične ure so prvič opisane v /Lamp 78/. Implementirane so kot funkcije C, ki dodelijo število vsaki začeti lokalni aktivnosti. To so torej navadni števci. V sistemu, kjer ima vsak proizvajalec svojo logično uro je problem, kako zagotoviti globalno razvrstitev. Funkcija C ima lastnost, da velja $C(i,a) < C(j,b)$, če sta a in b aktivnosti v procesih i in j in velja a > b. Pri realizaciji logičnih ur moramo upoštevati dve pravili:

Pravilo 1: Vsak proces i poveča vrednost števca C(i) med dvema zaporednima aktivnostima.

Pravilo 2: Če aktivnost a v procesu i pošilja sporočilo in potem sporočilo m vsebuje časovno značko T(m) = C(i,a). Po prejemu sporočila m proces j postavi svoj števec C(j) na vrednost, ki je večja ali kvečjemu enaka trenutni vrednosti in je večja od T(m).

Vsako funkcijo C, ki ima zgoraj navedeno lastnost, lahko uporabimo za popolno razvrstitev poljubne množice aktivnosti. Za to potrebujemo še popolno ureditev procesov (npr. glede na njihova imena). Aktivnost a se je zgodila pred b, če je $C(i,a) < C(j,b)$ oziroma $C(i,a) = C(j,b)$ in je $i < j$. Sihronizacijski mehanizem definiran s pravili 1 in 2 in popolna razvrstitev dovoljujeta konsistentno dodeljevanje aktivnosti. Definirana popolna razvrstitev ni edinstvena in ni ekvivalentna kronološki razvrstitvi. To je razlog, da je včasih treba implementirati sistem logičnih ur na sistemu fizičnih ur.

Mehanizem z uporabo abstraktnih izrazov /Herm83/ temelji na uporabi logičnih ur. Vsaka ura ima poleg osnovne verzije, ki je pri procesu, ki povečuje njeno vrednost še dodatne verzije pri drugih procesih, ki samo spremljajo njeno vrednost. Predpostavimo, da mora vedno veljati abstrakten izraz

$$\sum c_i x_i < k$$

kjer sta c_i in k konstanti, x_i vrednost logične ure in i število logičnih ur.
Dodatne verzije logične ure imenujemo njene slike. Označene so z $m(x_i)$ in se lahko razlikujejo od osnovne verzije. Z uporabo dodatnih verzij pri delu s sinhronizacijskimi spremenljivkami zelo zmanjšamo komunikacijske potrebe in povečamo učinkovitost.
Zamenjava osnovne verzije logične ure z njeno sliko je dopustna, če pri zamenjavi upoštevamo tako stroge omejitve, da je kljub dopustni napaki slike izpolnjena zahteva osnovnega abstraktnega izraza.
To pomeni, da lahko pri slikah logičnih ur, kjer je konstanta c_i negativna popravljamo njihove vrednosti z zakasnitvijo, ne da bi s tem ogrozili pravilnost abstraktnega izraza. Podobno lahko vrednosti slik logičnih ur, ki imajo konstanto c_i pozitivno, popravljamo vnaprej.

Sinhronizacijski mehanizmi, ki temeljijo na fizičnih ali logičnih urah imajo skupno lastnost, da ne temeljijo na medsebojnem izključevanju. To je velika prednost pri uporabi v porazdeljenih sistemih, saj omogoča paralelnost izvajanja..

Sinhronizacijski mehanizmi lahko pri svojem delu uporabljajo dejstvo, da imajo procesi edinstvena in stalna imena. To določa popolno razvrstitev procesov, ki daje opazovalcu občutek, da so procesi povezani v neko verigo ali obroč. Vsak proces ima natančno določenega prednika in naslednika. Taka logična razvrstitev ni nujno povezana s fizično topologijo sistema.
Obroč služi kot osnova za prenašanje posebnih pravic med procesi v sistemu. Pri sinhronizaciji taka posebna pravica pomeni dostop do sinhronizacijskega osebka.

Skupne spremenljivke: Sinhronizacijski mehanizem, ki temelji na konceptu logičnega obroča je bil predstavljen v /Dijk74/. Ker predvideva uporabo skupnih spremenljivk je uporaben v tesno sklopljnih sistemih.
Lastništvo nad posebno pravico lahko zaznamo z opazovanjem spremenljivke, ki jo proces deli z enim od obeh sosedov v obroču. Z znanimi algoritmi lahko dosežemo stabilno stanje, kjer od večih posebnih pravic ostane samo še ena, ki potem kroži po obroču in zagotavlja medsebojno izključevanje.

Odkrivanje napak, reševanje iz napak in dinamično širjenje sistema so možni z metodami, opisanimi pri podajanju žetona.

Krožeči žeton je sinhronizacijski mehanizem zasnovan na enakih osnovah kot deljene spremenljivke, le da se posebna pravica prenaša med procesi s sporočilom, ki ima edinstveno obliko in ga imenujemo žeton. S tem principom dosežemo medsebojno izključevanje procesov v šibko sklopljenih sistemih. Po prejemu žetona lahko proces opravi želeno aktivnost. Žeton lahko obdrži največ nek naprej določen čas in potem ga mora predati naslednjiku v obroču. Mehanizem mora zagotoviti obnovitev logičnega obroča in podajanja žetona, če izpade trenutni lastnik žetona ali katerikoli element logičnega obroča, hkrati pa lahko dopušča v sistemu samo en žeton.

Protokoli za odkrivanje napak in njihovo reševanje so podani v /LeLa77/ in v /LeLa78/. Tak sinhronizacijski mehanizem je uporabljen tudi na logičnem nivoju lokalnih mrež in definiran v /ISO802/.

Učinkovitost sinhronizacijskih mehanizmov za medsebojno izključevanje med procesi je v veliki meri odvisna od časovne obsežnosti kritičnih sekcij. Če kritične sekcije vključujejo izmenjavo sporočil med proizvajalci in potrošniki, potem to daje nizke zmogljivosti.

Krožeči razvrščevalnik je razvrščevalnik, ki stalno kroži po logičnem obroču in uporablja mehanizem podajanja žetona.

Po prejemu žetona lahko proces aktivira več primitivov tipa vstopnica in potem pošlje žeton naslednjiku. Na ta način dosežemo medsebojno izključevanje uporabnikov razvrščevalnika. V članku /LeLa78/ je opisani protokol, ki uporablja opisani koncept.

Protokol uporablja številko obhodov, ki jo nosi žeton in se poveča vsakič, ko doseže proces, katerega naslednjik ima nižjo številko, kot je njegova.

8. KRITERIJI ZA VREDNOTENJE SINHRONIZACIJSKIH MEHANIZMOV

Sinhronizacijski mehanizmi nimajo identičnih lastnosti. V nadaljevanju podajamo važnejše kriterije za vrednotenje sinhronizacijskih mehanizmov. Pomembnost vsakega od njih je odvisna od omejitev, ki jih postavimo sistemu.

Odzivni čas in propustnost: Vsak mehanizem mora v največji možni meri upoštevati in izkoriščati vse prednosti paralelnosti delovanja porazdeljenega sistema. Paralelnost v sinhronizaciji in pri komuniciranju omogoča veliko propustnost in kratke odzivne čase.

Prožnost pomeni, da mora sinhronizacijski mehanizem biti odporen na pojav napak in zmanjšanja števila procesov v sistemu. Potrebujemo pravzaprav bolj natančno merjenje te lastnosti, ki bi izrazila število hkratnih napak oziroma izpadov procesov, ki jih sistem še preživi.

Dodatna poraba virov (overhead) je pravzaprav cena, ki jo moramo plačati, da lahko izvedemo sinhronizacijo. Kaže se v povečanju prometa med procesi in je odvisna od števila in velikosti paketov, v večji potrebi za procesiranje (dodatnih sporočil za sinhronizacijo) in v uporabi pomnilnika za shranjevanje potrebnih informacij.

Konvergentnost in poštenost sta lastnosti sistema, ki zagotavljata, da se v konfliktnih situacijah viri in zmožnosti enakomerno porazdelijo med tekmujoče procese.

Razširljivost je zahteva, da mora sinhronizacijski mehanizem dopuščati dodajanje ali ponovno vključevanje procesnih elementov, ne da bi s tem motil delovanje sistema.

Določenost. Sistem imenujemo deterministično, če je zasnovan tako, da v vsakem primeru doseže sinhroniziranost. Če pa doseže sinhronizacijo samo v večini primerov, se sistem imenuje verjetnosten.

Reševanje iz napak je zelo pomemben nabor funkcij. Posamezni mehanizmi se razlikujejo po količini pomoči, ki jo nudijo pri reševanju iz napak, po vplivu pokvarjenih elementov sistema na pravilno delujoče in po času, ki je potreben, da se popravljeni elementi spet vključijo v normalno delovanje sistema.

Povezanost med elementi sistema se zelo razlikuje glede na različne mehanizme. Slabi so taki mehanizmi, ki zahtevajo polno povezanost, saj je le-ta zelo draga v primerjavi s tipi povezanosti, kjer so elementi združeni v verigo ali obroč.

Vzpostavitve začetnega stanja se po kompleksnosti lahko zelo razlikujejo.

Razumljivost in enostavnost mehanizma nam olajšata delo pri snovanju, formalnem preverjanju, implementaciji, testiranju in vzdrževanju.

9. LITERATURA

/Andr81/ G.R.Andrews: The Distributed Programming Language, Software Practice and Experience vol.12 1982

/BenA83/ M. Ben-Ari: Principles of Concurrent Programming, Prentice Hall International 1983

/BrHa73/ P.Brinch Hansen: Operating Systems Principles, Prentice Hall 1973

/Dijk68/ E.W. Dijkstra: Cooperating Sequential Processes in Programming Languages, Academic Press 1968

/Dijk74/ E.W. Dijkstra: Self-stabilizing Systems in Spite of Distributed Control, Comm. ACM vol.17 no.11 1974

/Fink86/ R.A. Finkel: An Operating SystemsVade Mecum, Prentice-Hall 1986

/Herm83/ D. Herman: Towards a Systematic Approach to Implement Distributed Control of Synchronization, in Distributed Computing Systems, Academic Press 1983

/Hoar74/ C.A.R. Hoare: Monitors: An Operating Systems Structuring Concept, Comm. ACM vol.17 no.10 1974

/Hoar78/ C.A.R. Hoare: Communicating Sequential Processes, Comm. ACM vol.21 no.8 1978

/Hwan85/ K. Hwang, F.A. Briggs: Computer Architecture and Paralel Processing, McGraw-Hill 1985

/ISO802/ ISO IS 8802/4 Local Area Networks, Token Passing Bus Acces Protocol

/Kope87/ H. Kopetz, W. Ochsenreiter: Clock Synchronization in Distributed Real-Time Systems, IEEE Trans. on comp., vol.36 no.8 1987

/Lamp78/ L. Lamport: Time, Clocks, and the Ordering of Events, Comm. ACM, vol.21 no.7 1978

/Lmps81/ L. Lampson: Atomic Transactions, LNCS 105, Springer Verlag 1981

/LeLa77/ G. LeLann: Distributed Systems - Towards a Formal Approach, Proc.IFIP Congress Toronto, North-Holland 1977

/LeLa78/ G. LeLann: Algorithms for Distributed Data Sharing System which Use Tickets, Proc. 3.Berkley Workshop, August 1978

/LeLa81/ G. LeLann: Synchronization, LNCS 105, Springer verlag 1981

/Slom85/ M. Sloman et. al: The CONIC Toolkit for Building Distributed System Proc. 6.IFAC Workshop on Distr. Contr. Comp. Sys., Pergamon Press 1985

/Slom87/ M. Sloman, J. Kramer: Distributed Computer Systems and Computer Networks, Prentice Hall International 1987

/Stan82/ J.A. Stankovic: Software Communication Mechanisms: Procedure Calls vs. Messages, IEEE Computer, April 1982

/Trip87/ A.Tripathi: Distributed Operating Systems, Tutorial of 7th ICDCS Berlin, september 1987

/USAD80/ USA Departement of Defence: Reference Manual for the ADA Programming Language, Proposed Standard Document 1980

/Verj83/ J.P. Verjus: Synchronization in Distributed Systems, in Distributed Computing Systems, Academic Press 1983