tanulmányok 128/1981

MTA Számitástechnikai és Automatizálási Kutató Intézet Budapest

MAGYAR TUDOMÁNYOS AKADÉMIA
SZÁMITÁSTECHNIKAI ÉS AUTOMATIZÁLÁSI KUTATÓ INTÉZETE

MULTI-TASK RENDSZEREK FEJLESZTÉSE MAGASSZINTÜ NYELVEN

Irta:

BÖSZÖRMÉNYI LÁSZLÓ

Tanulmányok 128/1981

A kiadásért felelős:

DR. VÁMOS TIBOR

ISBN 963 311 125 O

ISSN O324-2951

TARTALOMJEGYZÉK

Mottó: "Ha nem tudjuk müködőképessé tenni a konkurrens programokat ellenörző olvasás vagy tesztelés segitségével, akkor egyetlen más hatásos módszert látok: hogy valamennyi konkurrens programot egy olyan programnyelven irjuk, amely lehetővé teszi, hogy pontosan specifikáljuk a közös adatokhoz való hozzáférés feltételeit, és amely rendelkezik egy olyan forditóval, amelyik ellenőrzi is ezeket a feltételezéseket".

(Per Brinch Hansen - The architecture of concurrent programs 1977)

BEVEZETÉS

Tiz-tizenöt évvel ezelőtt, a megszakitással rendelkező számitógépek megjelenése előtt, a számitógép programok valamennyien szekvenciálisak voltak, a gép a program utasitásait a programozó által előirt sorrendben hajtotta végre. Ebben az időben nagy valószinüséggel lehetett számitani arra, hogy egy program azonos bemenet mellett többszöri futtatás esetén is ugyanugy müködik (hacsak nem tartalmaz a tártartalomtól függő hibát). A megszakitás megjelenése után a programok egy részét már nem lehetett egyetlen szekvenciális müveletsornak tekinteni, hanem annak egyes részei egyidőben, vagy időben átlapolódva, kerültek végrehajtásra. Az addigi szekvenciális programozás mellett kialakult a multiprogramozás is [Wir77d]. Ha egy multiprogram olyan részeket (processzeket - ld. később) is tartalmaz, amelyeknek futási idejére valamilyen korlátozó feltételnek is teljesülnie kell (azon tulmenően, hogy pozitiv), akkor a feladat már a valós idejü (real-time) programozás körébe tartozik. [Wir77d]. A multi- és a valós idejü programok közös tulajdonsága, hogy azonos bemenet esetén is különböző módon müködhetnek, és a lehetséges müködések egy része helyes, a másik része hibás. Az ilyen programok müködése általában nem determinisztikus. Tartalmazhatnak olyan hibákat, amelyek nem reprodukálhatók, és igy szinte lehetetlen megtalálni őket. Ezért a multi- és a valós idejü programok készitésénél még óvatosabbnak kell lennünk, mint a szekvenciális esetben, meg kell kisérelni a lehetetlent: hogy hibátlan programot irjunk.

Ebben a törekvésben kétféle segitséget is kaphatunk. Egyrészt a programozási elméletektől elvi, másrészt a programnyelvektől gyakorlati segitséget. A dolgozat célja, hogy bemutassa egyrészt a helyes programozás elveinek egy részét, másrészt pedig néhány ujabb programozási nyelv szolgáltatásait, mindkét esetben különös tekintettel a multiprogramozásra. (A multiprogramozás helyett azonos értelemben használom a párhuzamos, parallel programozás kifejezéseket, az ilyen programok által leirt rendszerekre pedig a multi-task, párhuzamos, parallel, konkurrens rendszer elnevezéseket.)

A helyes programok irásának kérdése már a szekvenicális programozás kapcsán fölmerült, és létrehivta a strukturált programozás elméletet. Ennek alapvető kérdéseivel foglalkozik [Dij68c, Dah78, Wir78]. A strukturált programozással külön nem foglalkozom a továbbiakban, de hatása a később tárgyalt összes programnyelvben jelen van.

A helyes programozás elősegitésére kialakult a programok axiomatikus tárgyalása, a programbizonyitás elmélete is. Ez állitásokat tesz egy program(rész) tulajdonságaira, amelyek helyességét matematikai módszerekkel bizonyitja. Ennek a kérdéskörnek átfogó tárgyalása található [Bar79]-ben. A multiprogramok bizonyitása elsősorban Hoare axiomatikus rendszerén alapul [Hoa69, Hoa72, Hab74, OwG76]. Programbizonyitással a továbbiakban nem foglalkozom, de hatása a később tárgyalt összes programnyelvben jelen van.

Az elmult évtizedben a párhuzamos rendszerek tervezése önálló tudományággá fejlődött. Ez a folyamat, ugy tünik, a hardware oldaláról indult, a megszakitásos gépek megjelenése vetett fel először olyan kérdéseket a software terve-

zésben amelyekre mintegy válaszképpen született meg a párhuzamos programozás elmélete. Az utóbbi években a multi-mikroprocesszoros rendszerek megjelenése hasonló kihivást jelentett, és már megszülettek az első válaszok is. Érdekes mindenesetre megfigyelni, hogy a párhuzamos tervezés elmélete a számitógép technikai lehetőségeinek bővüléséből ered, noha maga a kérdéskör ennél sokkal nagyobb hatósugaru. Ma már a párhuzamos tervezés többet jelent, mint a megszakitások megfelelő lekezelését, és számos olyan feladatot is érdemes párhuzamosság bevonásával megfogalmazni, amelynek semmi köze a számitógép architekturájához.

A párhuzamos rendszerek elméletének kialakulása, a strukturált programozás elméletével karöltve, lehetővé tette, hogy olyan magasszintü nyelvek készüljenek, amelyek az elméletek által előirt szabályokat figyelembe veszik, a programozót támogatják, szinte kényszeritik a helyes tervezői és programozói stilusra. Az Assembly szintü nyelveket minden fronton támadás érte, ma már léteznek olyan magasszintü nyelvek, amelyekkel a hardware közvetlen vezérlése is elegáns módon lehetséges. A magasszintü nyelvek hatékonysága, a generált kód minősége helyenként még indokolttá teheti Assembly szintü nyelvek használatát, de ez nem utolsósorban annak a következménye, hogy a számitógép architekturák eleve az Assembly programozók támogatására készültek. Például a magasszintü nyelvi fordítók támogatására épült "Lilith" gépre a MODULA-2 nyelv fordítója több mint háromszor olyan tömör kódot generál, mint a hagyományos architekturáju PDP/11-re [Wir81].

A magasszintü nyelvek alkalmazásának előnyei közül az első helyen a megbizhatóság növekedését emlitem. Rögtön hozzá kell tenni, hogy ez nem érvényes minden magasszintü nyelvre, csak azokra, amelyeket a helyes programozás elméletének ismeretében készitettek. Ezek viszont lehetővé teszik azt, hogy a hibák jelentős részét már forditási időben kiszürjük, valamint, hogy az egyszer már hibátlannak itélt modulokat később már ne lehessen elrontani (információ elrejtés). A következő előny, hogy a program elkészülési ideje egy nagyságrenddel csökken, olvashatósága, karbantarthatósága jelentősen nő. Számos nyelv erősen támogatja, hogy egy feladatot több részre osszunk, és a részfeladatok közötti interface-eket egyértelmüen definiáljuk. Végül, de nem utolsósorban egy jól tervezett magasszintü nyelven dolgozni kellemes, és a programozó megbizhat a saját programjában.

A dolgozatot 4 fejezetre osztottam. Az 1. fejezet a párhuzamos rendszerek alapfogalmaival foglalkozik. A második az ujabb keletü javaslatokkal. A 3. fejezetben 7 modern magasszintü nyelvről adok áttekintést a modularitás és a párhuzamosság szempontjából. Az utolsó fejezet egy esettanulmányt tartalmaz, amelynek során egy számitógép hálózati szint implementációját készitettem el MODULA-2 nyelven.

A dolgozatban a feladat jellegéből következően sokféle jelölést kell használnom, amelyeket nem mindenhol magyarázok meg teljes részletességgel, számitva arra, hogy önmaguktól érthetők. A példaprogramok azonositói részint magyar, részint angol eredetüek, utóbbiak esetenkénti használatát az indokolja, hogy a programnyelvek amugy is az angol nyelvre épülnek, és emiatt az angol azonositók használata világszerte elterjedt.

1. ALAPFOGALMAK

1.1 A PROCESSZ

Ha egy folyamat (processz) müveletei időben egymás után, előre megadott sorrendben játszódnak le, akkor a folyamatot szekvenciálisnak nevezzük. A "művelet" jelentését tovább nem definiáljuk, annyit tételezünk fel, hogy a művelet maga is szekvenciális és véges időn belül befejeződik. Az "egymás után" történő lejátszódást ugy is fogalmazhatjuk, hogy egy művelet akkor és csak akkor kerül végrehajtásra, ha egyértelmüen definiált elődje már befejeződött [Wir77c]. Ha a folyamat müveletei részben egyidejüleg (parallel) vagy egymás után, de előre meg nem adható sorrendben (kvázi-parallel) játszódnak le, akkor a folyamat nem szekvenciális. Az ilyen folyamatok leirása önmagában nagyon bonyolult (inkább reménytelen) lenne. Ezért az ilyen folyamatokat egymással versenyző (konkurrens) szekvenciális folyamatok (processzek) formájában ábrázoljuk. Valószinü, hogy létezik olyan nem szekvenciális folyamat, amely nem képezhető le együttmüködő konkurrens szekvenciális processzekre, de az ilyen folyamatokkal a továbbiakban nem foglalkozom. Az egyszerüség kedvéért és az irodalmi gyakorlatnak megfelelően a továbbiakban a processz mindig a szekvenciális processzt jelenti [BrH73, Dij68a, Dij68b, stb.].

1.2 A PROCESSZOR

A processzor az az aktiv egység, amely a processz müveleteit végrehajtja [Lal76, Hop78]. A processzor lehet egy tényleges központi egység (CPU), egy periféria vezérlő egység (például csatorna), vagy egy magasabb szintü, software által megvalósitott gép. Az ilyen magasabb szintü gép maga is állhat a processzek halmazából, amelyek processzora(i) ismét a fenti definiciónak felel(nek) meg. Egy hierarchikusan felépitett rendszerben igy processzek és processzorok tetszőleges szinjeit állithatjuk elő [Dij68b, Dij71]. Ahol szükséges hangsulyozni, hogy a processzort külön hardware egység valósitja meg, ott a fizikai processzor elnevezést alkalmazom.

1.3 DETERMINISZTIKUS VISELKEDÉS

Egy folyamatot determinisztikusnak nevezhetünk, ha kezdeti állapota egyértelmüen meghatározza a folyamat lefutását és eredményét. Ha ez a feltétel nem áll fenn, akkor a folyamat nem determinisztikus. A párhuzamos folyamatok eleve magukban hordják a nem determinisztikus viselkedés lehetőségét, és a helyes párhuzamos programozás célját ugy is megfogalmazhatjuk, hogy az nem más, mint a nem determinisztikus folyamatok determinisztikussá tétele. Egy algoritmus megfogalmazásakor kivánatos lehet, hogy annak egyes részeit, mint nem determinisztikus müveleteket fogalmazzuk meg, tehát, hogy az indeterminizmust az algoritmus részévé tegyük. Ezzel a kérdéssel később még részletesebben foglalkozom (a 2. fejezetben).

1.4 PROCESSZ KOMMUNIKÁCIÓ

A konkurrens processzek alapvetően egymástól térben és időben függetlenül müködnek (különböző processzorokban futhatnak, különböző időben). Ha müködésük funkcionálisan nem független, akkor esetenként térbeli és időbeli müködésüket is össze kell hangolniuk, szinkronizálniuk kell. A szinkronizáció állhat egyetlen jelzésből is, de jelentheti adatok cseréjét is.

Ha a kommunikáló processzek adatokat akarnak cserélni, akkor biztositani kell az adatok konzisztenciáját. Ez azt jelenti, hogy az adatoknak a processzek szempontjából mindig jól definiált értékkel kell rendelkezniük. Az adatok megváltoztatásának ezért a processzek felé atomi, oszthatatlan müveleteknek kell lenniök. Ennek a biztositása a processz kommunikáció sarkalatos kérdése. A megoldás erősen különböző lehet attól függően, hogy a processzek egy közös elérésü adatterületen keresztül kommunikálnak-e, vagy olyan különálló processzorokban futnak, amelyek csak valamilyen input/output müveletek révén kommunikálhatnak. (A processzor és az input/output nem feltétlenül fizikai berendezésre vonatkozik.)

Ha a processzek közös elérésü adatterületen keresztül kommunikálnak, akkor ugy biztosithatják az adatok konzisztenciáját, ha a közös adatokon való müveletek ideje alatt megtiltják a többi processznek, hogy a kritikus időszakban az adatokhoz hozzáférjenek. Ezt ugy is mondhatjuk, hogy a processzek kritikus szakaszokban kölcsönösen kizárják egy-

mást. A kölcsönös kizárás a konkurrens tervezés egyik legalapvetőbb fogalma. A kölcsönös kizárás megvalósitása a processzek valamilyen szinkronizációját igényli. Ez egyszersmind azt is jelenti, hogy a szinkronizáció általánosabb fogalom, mint a kölcsönös kizárás.

1.5 PROCESSZEK SZINKRONIZÁCIÓJA

A konkurrens tervezés elméletének kialakulásában döntő szerepe volt a szinkronizációs alapmüveletek (primitivek) megjelenésének. A kezdő lépést mindenképpen Dijkstra adta meg a szemafor fogalmának bevezetésével [Dij68a]. A szinkronizációs alapmüveletek fejlődéséről összefoglalást találhatunk [Bri73]-ban, [Hop78]-ban és Sten Andler-nél [And79]. A következő áttekintésben elsősorban Sten Andler csoportositása szerint haladok.

1.5.1 A SZEMAFOR ELŐTT

A szemafor megjelenése előtt a kölcsönös kizárás megvalósitása erősen magán viselte a probléma eredeti jelentkezésének helyét: a megszakitások megjelenését. A feladat ugy jelentkezett, hogy valahogy "el kell bánni" a megszakitásokkal [Dij72]. A megszakitással való elbánás legegyszerübb módja a megszakitások maszkolása egy kritikus időszakra. Ez az eljárás máig is nélkülözhetetlen egy operációs rendszer legalsó hierarchikus szintjén, de igen rossz megoldás mint általános szinkronizációs müvelet. (Jellemző a "maszkolásos korszak" szemléletére az, hogy sok tervező megszakitás engedélyezési szakaszokat iktatott

programjaiba, azokat a helyeket kereste, ahol a megszakitást be lehet engedni, és nem azt, ahol ki kell zárni. A maszkolások ilyenfajta használata teljesen áttekinthetetlenné tette a programokat, amelyekben nem rövid kritikus, hanem rövid "nem kritikus" szakaszok voltak.)

Az egyik első szinkronizációs alapmüvelet a LOCK-bit. Ha egy processz be akar lépni a kritikus szakaszba, akkor a LOCK müvelettel lefoglalja azt és kilépéskor az UNLOCK müvelettel feloldja. A LOCK-bit müveletei nem tartalmaznak implicit sorkezelést, igy a várakozás a LOCK-bit állandó lekérdezésével valósul meg, és a kritikus szakaszba való belépés sorrendje független az igény felmerülésének sorrendjétől (tehát egy "balszerencsés" processz örökre kiszorulhat). Az érdekesség kedvéért nézzük meg a LOCK-bit egy lehetséges implementációját Modula-2 jelölésben és föltéve, hogy a processzor rendelkezik egy EXCHANGE(x,y) nevü atomi (oszthatatlan) müvelettel, amely x és y értékét megcseréli:

```
MODULE LOCKbit;

EXPORT lock,unlock;

TYPE lockins = (locked,unlocked);
VAR lb: lockins;

PROCEDURE lock;
VAR x: lockins;
BEGIN  x:= locked;
  WHILE x = locked DO EXCHANGE(x,lb) END; (*WHILE*)
END lock;

PROCEDURE unlock;
BEGIN lb:= unlocked
END unlock;

BEGIN lb:= unlocked

END LOCKbit.
```

1.5.2 A SZEMAFOR

A szemafort E.W. Dijkstra vezette be [Dij68a,Dij71]. A szemafor egy adattipus, amelyen két müvelet értelmezett, a P és a V müvelet. (A müveletek elnevezése a holland Passeresn illetve Vrjigeven, elfoglalni, illetve felszabaditani szavakból ered). A müveletek jelentése a következő:

P(s): s:= s-1 (s > 0)

A P művelet 1-gyel csökkenti az s szemafor tipusu változó értékét, ha értéke 0-nál nagyobb. Ellenkező esetben a P müveletet végrehajtó processz "elalszik", várakozó állapotba megy. A szemafor lekérdezése és csökkentése oszthatatlan müvelet.

V(s): s:= s+1

A V művelet 1-gyel növeli s értékét, egyszersmind az s-re váró processzek közül (ha van ilyen) egyet "felébreszt". A V művelet nem irja elő, hogy melyik legyen ez a processz, csak azt köti ki, hogy egy processz sem várakozhat végtelen sokáig. Egy lehetséges implementáció a "first-in/first-out" stratégia, amely az érkezési sorrendben szolgálja ki a processzeket. A szemafor növelése oszthatatlan müvelet.

A szemafor számos szinkronizációs feladat helyes és elegáns megoldását lehetővé teszi. A kölcsönös kizárást például egyszerüen ugy valósithatjuk meg, hogy a kritikus

szakaszt egy összetartozó P-V pár közé fogjuk:

```
VAR mutex: semaphore;
  mutex:= 1;
LOOP
  P(mutex);
  kritikus szakasz műveletei;
  V(mutex);
  tetszőleges egyéb, nem kritikus műveletek;
END; (*LOOP*)
```

A mutex (mutual exclusion - kölcsönös kizárás) nevü szemafor kezdeti értéke 1. Ez azt jelenti, hogy a P(mutex) müvelet csak egy processzt enged be a kritikus szakaszba egyszerre. A V(mutex) hatására az esetleg várakozó processzszek közül belép az egyik (mondjuk a sorrendben következő). Az 1 kezdeti értékü szemafort szokás bináris szemafornak is nevezni.

A szemaforok alkalmazására tekintsünk egy, az irodalomban sokszor felhasznált példát, a véges körpuffer példáját. Adott egy véges méretü körpuffer, amelyen egy termelő és egy fogyasztó processz dolgozik. A feladat egy olyan algoritmus készitése, amely biztositja, hogy a körpufferhez egyszerre csak egy processz férhet hozzá, és egyszersmind a fogyasztó és termelő közötti esetleges sebesség-különbséget is kiegyensulyozza. A fenti leirásból érezhető, hogy itt kétféle szinkronizációra is szükség van. Egyrészt biztositani kell a puffer hozzáféréshez a kölcsönös kizárást. (Ez általában rövid ideig tartó müvelet, ezért a hozzátartozó szinkronizációs algoritmust rövidtávu ütemezésnek is szokták nevezni [BrH73].) Másrészt biztositani kell, hogy ha a puffer üres, illetve tele van, akkor a fogyasztó, il-

letve a termelő megvárhassa a másikat. (Ezt nevezhetjük középtávu ütemezésnek [BrH73].) Az alkalmazott jelölés Dijkstrá-tól ered. A szemafor változók neve magyarázza jelentésüket. A cobegin és coend utasitások azt jelentik, hogy a közöttük megadott utasitások (producer és consumer) egymással párhuzamosan, vagyis processzként müködnek.

```
BEGIN
  mestöltött-elemek-száma: semaphore(0); (*kezdetben 0*)
  üres-helyek-száma: semaphore(n); (*kezdetben n*)
  puffer-hozzáférés: semaphore(1); (*kezdetben 1*)
COBEGIN
  producer (*termelő*):
  REPEAT
  BEGIN
  következő elem előállitása;
  P(üres-helyek-száma);
    P(puffer-hozzáférés);
      puffer elem betöltése;
    V(puffer-hozzáférés);
  V(mestöltött-elemek-száma);
  END;

  consumer (*fogyasztó*):
  REPEAT
  BEGIN
  P(mestöltött-elemek-száma);
    P(puffer-hozzáférés);
      puffer elem kivétele;
    V(puffer-hozzáférés);
  V(ures-elemek-száma);
  a kivett elem feldolgozása;
  END;
COEND
END
```

A példában jól látszik a szemaforok kétféle felhasználása a kétféle ütemezésre. A puffer-hozzáférés nevü bináris szemafor azt biztositja, hogy egy adott pillanatban csak az egyik processz férhet hozzá a pufferhez. A másik két szemafor a középtávu ütemezést végzi, lehetővé teszik, hogy a két processz bevárja egymást. Nagyon lényeges a P müveletek sorrendje a processzekben. Ha a P(puffer-hozzáférés) művelet állna elől, akkor előfordulhatna, hogy mondjuk a fogyasztó tuljut rajta, s ekkor a puffert éppen üresnek találja, tehát a P(megtöltött-elemek-száma) müveletben várakozó helyzetbe kerül. A termelő azonban a P(puffer-hozzáférés) müveleten nem tud tuljutni, hiszen a fogyasztó még "benne van", nem adta ki a megfelelő V-t. Ebből a helyzetből többé nincs kiut, a két processz "örökre" várakozó állapotban marad,ez egy ugynevezett patt-helyzet. Az utóbbi megfontolások arra is rámutatnak, hogy a szemfor ugyan eszközt ad a szinkronizáció helyes megoldására, de biztositékot nem ad a hibátlanságra. A későbbiekben látni fogjuk, hogyan alakultak ki olyan uj nyelvi elemek, amelyek a szinkronizáció biztonságát és a kifejezés kényelmességét növelik.

1.5.3 A KRITIKUS RÉGIÓ

A kritikus régió fogalmát C.A.R. Hoare és P. Brinch-Hansen vezette be [Hoa72,BrH73]. A kritikus régió a rövidtávu ütemezés biztonságos megoldására szolgál. A kritikus régió egy adott változóra vonatkozik (egy közös-shared változóra) és biztositja a kölcsönös kizárást. Legyen a buffer nevü változó közös, akkor a rajta végzett müveletek

csak a hozzátartozó kritikus régiókban fordulhatnak elő, amelyek biztositják, hogy bennük egyszerre legfeljebb 1 processz tartózkodhat.

VAR buffer SHARED ... egyéb tipus megadások ... ;
REGION buffer DO müveletek a buffer nevü változón OD

1.5.4 FELTÉTELES KRITIKUS RÉGIÓ

A kritikus régió igen jól megoldja a kölcsönös kizárást, a rövidtávu ütemezést. A kommunikáló processzek sebességkülönbségét kiegyensulyozó középtávu ütemezést azonban továbbra is minden egyes programban szemaforok segitségével kell biztositani, ami könnyen eltéveszthető. Ezt a problémát oldja meg a feltételes kritikus régió [Hoa72, BrH73]:

REGION v WHEN b DO s1;s2; ... sn OD

Itt v egy közös (shared) változó, amelyre fenn kell állnia a kölcsönös kizárásnak, b pedig az a feltétel, amelynek teljesülnie kell, mielőtt a kritikus régió utasitásai végrehajtásra kerülnek. (Ilyen feltétel például a fogyasztó számára, hogy a puffer nem üres, illetve a termelő számára, hogy nincs tele). Ha a b feltétel nem teljesül, akkor a hivó processz azonnal kilép a kritikus régióból és egy várakozó sorba kerül. Valahányszor egy processz elhagyja a feltételes kritikus régiót, a b feltételt mindig ujra ki kell értékelni. Ezt ugy lehet például megvalósitani, hogy a közös várósorból valamennyi processz ismét belép a saját kritikus régiójába. Ez azt jelenti, hogy egyes processzek esetleg többször is kiértékelik a megfelelő b feltételt, mielőtt a kritikus utasitásokat végrehajtják. A feltételes

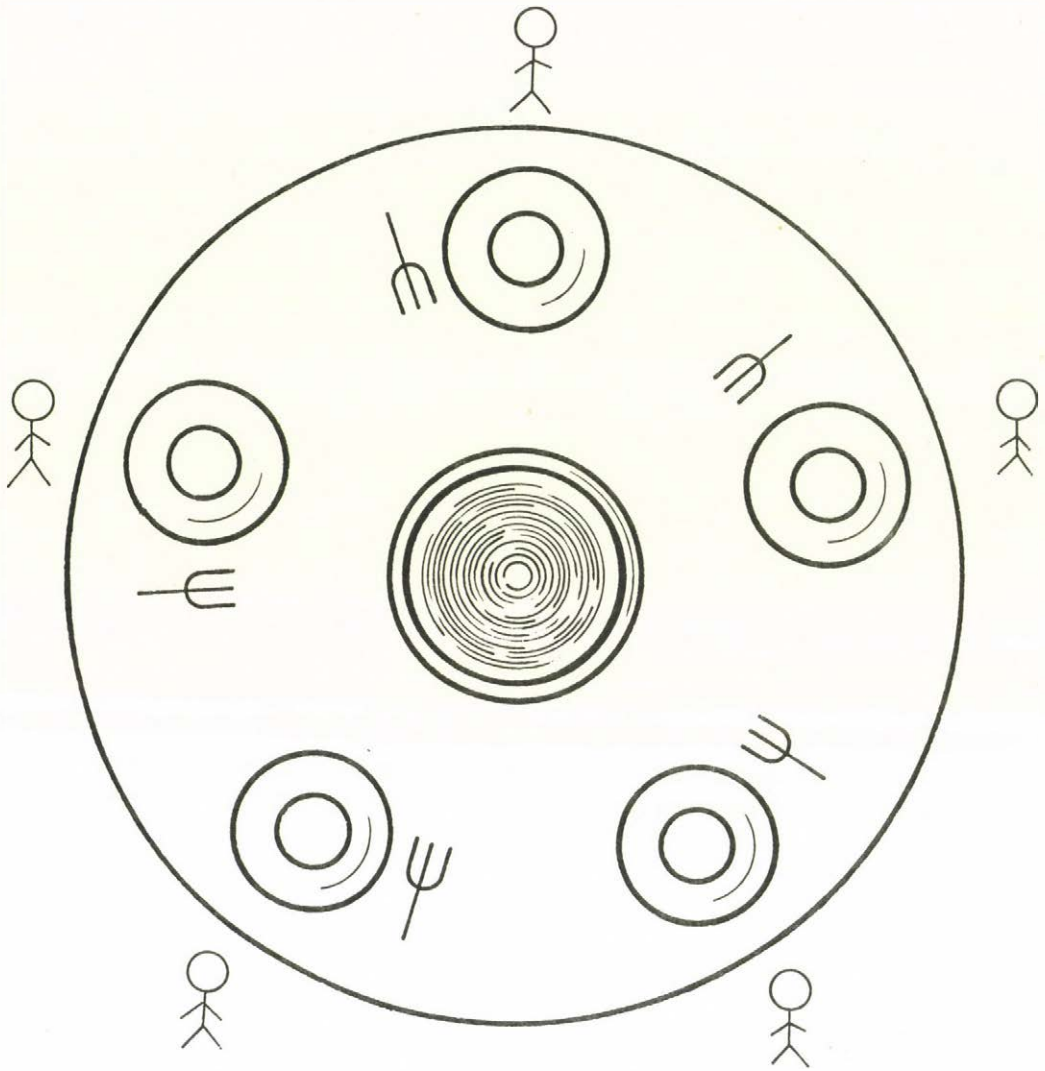
kritikus régiók egy fizikai processzoron ezért nem a legjobb hatásfokkal implementálhatók. (Később látjuk majd, hogy az EDISON nyelv mégis használja ezt a nyelvi konstrukciót, föltételezve, hogy minden processz külön fizikai processzoron fut.) A feltételes kritikus régió segitségével nagyon elegánsan oldhatjuk meg a már látott véges körpuffer példát:

```
VAR puffer SHARED
          RECORD info: információ-típus; tele: integer END;

REGION puffer WHEN tele < maximum DO
                        puffer-elem-betevése;
                        tele:= tele + 1
                              OD

REGION puffer WHEN tele > 0 DO
                        puffer-elem-kivétele;
                        tele:= tele - 1
                              OD
```

A feltételes kritikus régiók bonyolultabb ütemezési feladatok esetén válnak különösen vonzóvá. Példákat találhatunk [BrH73]-ban, amelyek jól mutatják, hogy bizonyos bonyolultság felett a szemaforok alkalmazása körülményessé válhat. Különösen elegáns és hires Hoare feltételes kritikus régiót alkalmazó megoldása az étkező filozófusok problémájára. A feladat Dijkstra-tól származik, és a következőképpen szól: Él valahol 5 filozófus, akik állandóan gondolkodnak, amig csak meg nem éheznek. Ha éhségüket eloltották, ismét gondolkodni kezdenek. Egy gazdag filozófiapártoló ezért a rendelkezésükre bocsátott egy csodálatos tálat, amelyből sose fogy ki a spagetti. Ezt elhelyezte az asztal közepén, köré 5 tányért, 5 villát, és az aztal köré 5 széket, minden filozófusnak egy saját széket.



Az étkező filozófusok

Sajnos kiderült, hogy a spagetti olyan különös természetü, hogy még a legügyesebb filozófus is csak 2 villával tudja megenni. Ebből viszont az következik, hogy az egymás mellett ülő filozófusok egyszerre nem ehetnek. A feladat olyan algoritmus készitése, hogy valamennyi filozófus véges időn belül ehessen, lehetőleg megéhezési sorrendben. Az evés, illetve a gondolkodás idejére semmilyen más kikötést nem tehetünk, mint hogy véges. Az első veszély, amelyet el kell kerülnünk az, hogy, ha valamennyi filozófus egyszerre éhezik meg, egyszerre ragadja meg mondjuk a baloldali villáját, akkor egyikük sem talál szabad villát, egyikük sem tud enni, s igy patt-helyzet áll elő. Ezt kiküszöbölhetjük például ugy, hogy az 5. filozófus mindig a jobboldali villához nyul először. Még igy is fennáll az a veszély, hogy két váltakozva evő filozófus kiéheztetheti a közöttük ülőt. Hoare megoldása a következő:

```
INTEGER ARRAY villak [0:4];
(*A villàk tömb i-dik eleme az i-dik filozófus számára
  rendelkezésre álló villák száma; ez kezdetben 2 *)

(*Az i-dik filozófus működését leíró processz: *)

LOOP
  gondolkodik;
  REGION villak WHEN villák[i] = 2 DO
            villák[(i-1) MOD 5]:= villák[(i-1) MOD 5]-1;
            villák[(i+1) MOD 5]:= villák[(i+1) MOD 5]-1
                                  OD;
  eszik;
  REGION villak DO
            villák[(i-1) MOD 5]:= villák[(i-1) MOD 5]+1;
            villák[(i+1) MOD 5]:= villák[(i+1) MOD 5]+1
                                  OD;
END (*LOOP*)
```

1.5.5 A MONITOR

A monitor előzetes ihletője egyrészt a SIMULA/67 nyelv [Dah78] osztály koncepciója, másrészt a Dijkstra által bevezetett titkár [Dij71]. A monitor végső megformálójának általában Brinch Hansen-t és Hoare-t tartják [Hoa73].

A monitor egy absztrakt adatstrukturát valósit meg [Dah78] a SIMULA osztályoz hasonlóan. Ez azt jelenti, hogy egyetlen szintaktikai egységbe foglal bizonyos adatstruktúrákat, a hozzájuk tartozó müveletekkel együtt. A szintaktikai egység környzete elől elrejti az adatstruktura és a müveletek részleteit és csak egy jól definiált interface-t tesz láthatóvá. Ezzel egyrészt mentesiti a környezetet a belső müveletek részletes ismeretétől, másrészt és ez a fontosabb, lehetetlenné teszi, hogy belső változóit a környezet szétrombolja. A monitor ezen tulmenően még azt is biztositja, hogy az általa definiált és a környezet számára elérhetővé tett eljárásokra teljesüljön a kölcsönös kizárás. Ez azt jelenti, hogy ha egy processz meghiv egy monitor eljárást, akkor a többi processz addig nem léphet be (várakozni kényszerül), amig az első processz a monitort (végleg vagy ideiglenesen) el nem hagyja. A monitor a kölcsönös kizárás mellett eszközt ad a középtávu ütemezésre is, egy uj adattipus, a feltétel (CONDITION vagy SIGNAL [Wir77a]) bevezetésével. A feltétel tipusu változókon kétféle müvelet értelmezett. A WAIT (vagy DELAY) művelet egyrészt feloldja a kölcsönös kizárást (a WAIT utasitást végrehajtó processz ideiglenesen elhagyja a monitort), másrészt lehetővé teszi, hogy egy processz egy későbbi jelzésre várakozzék. A SIGNAL (vagy CONTINUE) utasitás reaktivizálja az adott feltételre legrégebben várakozó processzt.

Ha a feltételre senki sem vár, akkor SIGNAL hatástalan. A SIGNAL müvelet Hoare eredeti definiciója szerint [Hoa73] azonnal elinditja a reaktivizált processzt, nehogy közben egy másik processz beléphessen és a felszabadult erőforrást "elorozhassa". (Később látjuk majd, hogy az eredeti koncepciót egyes nyelvek érdekes módon megváltoztatták, igy a MESA a fenti feltételt törölte, a MODULA pedig ezt megtartotta, de a SIGNAL müveletre is kiterjesztette a kölcsönös kizárás feloldását.)

A monitor alkalmazására először nézzük meg, hogyan implementálhatjuk segitségével a bináris szemafort:

```
szemafor: MONITOR
BEGIN foglalt: BOOLEAN;
       szabad: CONDITION;

PROCEDURE P;
BEGIN IF foglalt THEN szabad.WAIT;
      foglalt:= TRUE
END P;

PROCEDURE V;
BEGIN foglalt:= FALSE; szabad.SIGNAL
END V;

foglalt:= FALSE     (*kezdeti érték*)

END szemafor.
```

A P és V müveletek a monitoron kivülről a szemafor.P és szemafor.V formában hivhatók. A példa jól mutatja, hogy a feltétel tipus egyszerübb a szemafornál, ugy tekinthető, mint egy dinamikus jelzés, amelyhez semmilyen statikus, tárolt információ nem tartozik. A feltétel tipus igy a szemafornál nagyobb szabadságot ad, de önmagában, valami-

lyen tartalmilag kapcsolt tárolt információ nélkül használni nem szabad [Wir77b]!

A monitor alkalmazásának további szemléltetésére oldjuk meg ismét a véges körpuffer feladatot, ezuttal teljes részletességgel:

```
körpuffer: MONITOR
BEGIN puffer: ARRAY 0..n-1 of pufferelem;
      mutató: 0..n-1;
      elemszám: 0..n;
      nemüres,nemtele: CONDITION;

PROCEDURE betesz(x: pufferelem);
BEGIN IF elemszám = n THEN nemtele.WAIT;
  puffer[mutató]:= x;
  mutató:= (mutató + 1) MOD n;
  elemszám:= elemszám + 1;
  nemüres.SIGNAL

END betesz;

PROCEDURE kivesz(VAR x: pufferelem);
BEGIN IF elemszám = 0 THEN nemüres.WAIT;
  x:= puffer[(mutató - elemszám) MOD n];
  elemszám:= elemszám - 1;
  nemtele.SIGNAL
END kivesz;

elemszám:= 0; mutató:= 0;   (*kezdeti értékek*)

END körpuffer
```

A monitor koncepció nem igényli a logikai feltételek többszöri kiértékelését, mint a feltételes kritikus régió, viszont a monitor készitője felelős a logikai feltételek helyes állitásáért és a jelzés elküldéséért.

A monitor és általában az eddig tárgyalt szinkronizációs kérdések jobb megértéséhez nézzük meg, hogy hogyan valósithatjuk meg a monitort szemaforok segitségével [Hoa73]. (A forditottját már láttuk.) A kölcsönös kizárás biztositására bevezetjük a mutex nevü bináris szemafort. Erre minden monitor eljárás hivásakor ki kell adni a P(mutex), befejezésekor pedig a V(mutex) müveletet. Annak biztositására, hogy a SIGNAL müveletet kiadó processz megvárhassa, amig az általa reaktivizált processz őt továbbengedi, bevezetjük a sürgős nevü, 0 kezdeti értékü szemafort. A monitor elhagyása előtt mindig meg kell vizsgálni, hogy várakozik-e valaki sürgős-re, mert ebben az esetben azt kell továbbengedni a V(sürgős) utasitással. A sürgős-re várakozó processzszek számát a sürgősszám nevü INTEGER tipusu változóban (kezdeti értéke 0) tároljuk. A monitor eljárásokból való kilépés igy:

IF surgősszám > 0 THEN V(sürgős) ELSE V(mutex)

A monitor minden egyes lokális feltétel tipusu változóját egy-egy szemaforral és számlálóval ábrázoljuk, ezek neve legyen condsem és condszám (kezdeti értékük 0). Ekkor a rajtuk végezhető müveletek a következőképpen fejezhetők ki:

```
WAIT:
  condszám:= condszám + 1;
  IF sürgősszám > 0 THEN V(sürgős) ELSE V(mutex) END;
  P(condsem);
  condszám:= condszám - 1;

SIGNAL:
  sürgősszám:= sürgősszám + 1;
  IF condszám > 0 THEN V(condsem); P(sürgős) END;
  sürgősszám:= sürgősszám - 1;
```

Ez a megoldás nagyon általános, és lehet olyan értelmes megszoritásokat tenni, ami a valóságban sokkal egyszerübb megoldást tesz lehetővé. Ha egy eljárás nem tartalmaz, se WAIT se SIGNAL müveletet, akkor az eljárások befejezése egyszerüen V(mutex). További egyszerüsitések érhetők el, ha a SIGNAL és a WAIT müveletek mindig az eljárások végén állnak [Hoa73].

1.5.6 UTKIFEJEZÉSEK

Az utkifejezések egy szélesebb család, a reguláris kifejezéseken alapuló nyelvek körébe tartoznak, Ezeket az utóbbi időkig elsősorban specifikációs célra használták, ujabban léteznek implementációk is. A nyelvcsalád széles körü áttekintése található [Sha79]-ben. A reguláris kifejezések önmagukban nem alkalmasak párhuzamosság kifejezésére, ennek érdekében különböző kibővitéseket tettek (EE - event expression, FE - flow expression, PE - path expression). A kérdéskör sokkal bonyolultabb annál, semhogy részletekbe bocsátkozhatnék, szinronizációs szempontból talán a leglényegesebb közös tulajdonság az, hogy a programozónak nem kell explicite megadnia a szinkronizáció végrehajtását, hanem csak korlátozó követelményeket kell megfogalmaznia. Hogy ezt érthetővé tegyük, nézzünk meg egy egyszerü példát, az utkifejezések alkalmazására. Legyen a feladat egy 1 elemü puffer realizálása, amelyen egy termelő és egy fogyasztó dolgozhat:

```
TYPE puffer;
     f: message; (*message a cserélt információ típusa*)
     PATH Put Get END;
     OPERATIONS
      PROCEDURE Put(x: message); (*betesz*)
      BEGIN f:= x END;
      PROCEDURE Get(x: message); (*kivesz*)
      BEGIN x:= f END;
END TYPE
```

A PATH után megadott utkifejezés az OPERATIONS után szereplő eljárások hivására tesz korlátozást, mégpedig azt, hogy először mindig egy Put, azután egy Get utasitást kell kiadni. Látható, hogy ennek a korlátozásnak a betartatása nem terheli tovább a programozót. Az utkifejezések további vonzereje, hogy kapcsolatba hozhatók a Petri hálókkal, és az ottani elméleti eredmények (például a program patt-helyzet mentességének igazolása) alkalmazhatók. Még egyszer hangsulyozni kivánom, hogy a reguláris kifejezéseken alapuló nyelvek, sőt maguk az utkifejezések témája is már külön tudomány, és itt éppen csak megemlitettem.

2. A CSP ÉS A DP NYELV

Az előző fejezetben láttuk a szinkronizációs alapműveletek és a velük kapcsolatos nyelvi alapelemek (feltételes kritikus régió, monitor, processz stb.) kialakulását és alkalmazását. Ebben a fejezetben rátérünk néhány ujabb és összetettebb javaslatra a párhuzamos nyelvekre vonatkozólag. Ezek a javaslatok nem teljes nyelv definiciók, de mint később látni fogjuk, erősen befolyásolták a konkrét nyelvek (mindenek előtt az ADA nyelv tervezőit). Az ujabb javaslatokon érezhető, hogy kialakulásukra a hardware és a software fejlődése egyaránt hatott. A hardware főleg annyiban, hogy olcsó, sok processzoros, esetleg közös tár nélküli berendezések elterjedése várható, mégpedig olyan alacsony áron, hogy a kihasználtság másodlagossá válhat a megbizhatósághoz képest. A software pedig annyiban, hogy a jelenlegi javaslatok figyelembe vehették az elméleti kutatások eddigi eredményeit.

2.1 A CSP JAVASLAT

A CSP (Communicating Seguential Processes) nyelvet Hoare javasolta [Hoa78]. A javaslatnak talán legfontosabb gondolata az, hogy csökkenteni kivánja a felhasznált fogalmak számát. Igy a következő alapvető elemek alkotják a CSP fogalmi készletét:

1. A szekvenciális vezérlési strukturák megadására a Dijkstra féle őrzött parancs (guarded command [Dij75]) szolgál, némi szintaktikai módositással. Az őrzött parancsok adják az egyetlen eszközt a nem-determinisztikus viselkedések leirására. Pontos definiciójuk [Dij75]-ben és [Hoa78]-ban található. Müködésük lényege a következő: az őrzött parancs őrök és őrzött alparancsok sorozatából áll. Először mindig az őrök kerülnek végrehajtásra. Az őr Dijkstra eredeti definiciójában logikai (Boolean) kifejezés, a CSP-ben logikai kifejezések sorozata is megadható. Ha valamennyi őr végrehajtása sikertelen, akkor az egész őrzött parancs végrehajtása sikertelen. Ciklikus parancs esetén ez a ciklusból való kilépést, feltételes parancs esetén hibát jelent. Ha egyetlen őr végrehajtása sikeres, akkor a hozzátartozó őrzött alparancs, ha több is sikeres, akkor ezek közül egy tetszőlegeshez tartozó alparancs kerül végrehajtásra. Tekintsünk két példát, Hoare jelölésének megfelelően:

```
[x>=y -> m:=x # y>=x -> m:=y]
```

Ez egy feltételes őrzött parancs. A -> jel előtt található az őr, mögötte az őrzött alparancsok. A # jel az alternativákat választja el. Ha x>y, akkor m értéke x, ha y>x, akkor m értéke y lesz. Ha mindkét őr végrehajtása sikeres (x=y), akkor a kettő közül bármelyikhez tartozó őrzött alparancs hajtódik végre, a példában ez mindegy, m értéke mindenképpen ugyanaz lesz.

```
i:=0; *[i<határ;tartalom(i) <> n -> i:=i+1]
```

Ez egy ciklikus parancs, megkeresi a tartalom nevü tömbnek azt az elemét, amely n-nel egyenlő. A ciklust a * jelzi, a ciklusból való kilépésnek két feltétele van; ha i elér egy adott határt, vagy ha sikerült megtalálni a keresett elemet.

2. A párhuzamosságot parallel parancsokkal lehet kifejezni, ezek hasonló konstrukciók, mind Dijkstra COBEGIN és COEND utasitásai [Dij68a]. A parallel parancs szekvenciális elemei (processzek) "egyszerre" indulnak és egymással versenyezve futnak. Például:

[x!!ut1 || y!!ut2 || z!!ut3]

Az x y és z nevü processzek egymással versenyezve hajtják végre szekvenciális utasitás-sorozataikat. A || jel a processzeket választja el.

3. A processzek egymás közötti kommunikációjának céljára egyszerü input/output jellegü parancsok állnak rendelkezésre. Közös, globális adatokhoz való hozzáférés nincs! Az input/output processzek (és nem processzorok) között értelmezett müvelet. Ha egy rendszerben minden egyes processzhez külön fizikai processzor tartozik, akkor ez a kettő lényegében egybeesik. Példa:

kártyaolvasó?kártyakép

nyomtató!sor

A ? jel az inputot, a ! jel az outputot jelenti. A példában az első utasitás a kártyaolvasó nevü pro-

cessztől olvas be egy értéket a kártyakép nevü változóba, a második pedig a nyomtató nevü processznek adja át a sor változó értékét.

4. A processzek közötti kommunikáció akkor és csak akkor jön létre, ha a partnerek kölcsönösen megfelelő utasitást adtak ki, tehát p1 processz olvasni kiván p2-től és p2 adni kiván p1-nek. A müvelet végrehajtása abból áll, hogy az adóban megadott érték átmásolódik a vevőben megadott változóba, pontosan ugy, mint egy értékadó utasitásnál. Nincs automatikus pufferelés, az a processz, amelyik input/ouput utasitást ad ki, várakozó helyzetbe kerül addig, amig a megfelelő "ellen"-utasitást egy másik processz ki nem adja, Az input/output tehát ugy játszódik le, mint egy (esetleg megkéslelteteti) értékadó utasitás.

5. Input utasitások az őrben is előfordulhatnak. (Ujabb javaslatok szerint output utasitások is.) Ha az őrben megadott input utasitás párját még nem adták ki, akkor ez nem jelent sikertelenséget, hanem várakozást okoz, ha nincs egyéb sikeres őr. Ha egyszerre több input utasitást tartalmazó őr is rendelkezik adáskész partnerrel, akkor az őrzött parancsok elveinek megfelelően, ezek közül egy tetszőleges választható ki. (Egy lehetséges ésszerü implementáció, ha az érkezési sorrend szerint választódik a megfelelő rész.) Egy input (output) utasitást tartalmazó őr csak akkor tekinthető sikertelennek, ha az a processz, amelyre partnerként hivatkozik, már befejeződött. Ha egy ciklikus parancs őrei ebben az értelemben sikertelenek, akkor ez a cik-

lusból való kilépést okoz. (Ennek az a következménye, hogy egy ciklus utasitás után csak a tisztán logikai feltételeket tartalmazó őrökre vonatkozólag tudhatjuk biztosan, hogy a logikai feltételek hamissá váltak, ahol ezek inputtal keverve fordulnak elő, ott nem tudhatjuk a kilépési feltétel igazi okát.)

6. Az értékadásnál bizonyos tipus egyezéseknek kell teljesülinök, ellenkező esetben az sikertelen. Egyszerü és strukturált változók egyaránt léteznek a CSP-ben. Egy komponens nélküli strukturált változó egy input/output utasitásban egy jelzés szerepét töltheti be a feltételhez (CONDITION vagy SIGNAL) hasonló módon.

A CSP nyelv megértéséhez és szemléltetéséhez vegyünk néhány példát:

1. Irjunk egy másoló processzt, amely egy forrás processz által kibocsátott karaktereket egy nyelő processznek ad át:

```
másoló:: *[c:CHARACTER; forrás?c -> nyelő!c]
```

A c karakter tipus változó a másoló processz lokális változója. A másoló processznek akkor van vége, ha a forrás nevü processz befejeződik. Ekkor a forrás?c őr végrehajtása sikertelenné válik, a ciklus befejeződik és ezáltal a másoló is befejeződik. Következésképpen a nyelő processznek a másolóra vonatkozó további input utasitásai sikertelenek. A másoló processz egy 1 karakteres pufferként müködik a forrás és a nyelő között.

2. Implementáljunk egy monitort CSP-ben. A monitor itt nyilván egy processz, amely több felhasználó processzszel kommunikál. Egy ilyen feladat megoldásánál zavaró valamelyest, hogy a monitoron keresztül egymással kommunikáló processzeknek mind különböző névvel

kell rendelkezniük. Ez a kikötés azonban nem olyan sulyos, mert a CSP-ben lehetőség van processz-tömbök megadására is. Ebben az esetben a processz tömb elemeire indexelten lehet hivatkozni. A monitort megvalósitó processz programja igy:

```
*[(i:1..100)x(i)?(berar) ->
          ...; x(i)!(kirar)]
```

Az i változó értéke az 1-100 tartományban mozoghat, és egyértelmüen azonositja, hogy melyik processznek melyik kimenő paramétert kell átadni. Ha a monitor hivás elfogadására valamilyen megkötést akarunk tenni, akkor az őrt egyszerüen kibővithetjük egy logikai feltétellel. Tegyük fel például, hogy ugyanattól a processztől nem fogadunk el hivást kétszer egymás után:

```
j:=0; *[(i:1..100)i<>j;x(i)?(berar) ->
                ...; j:=i]
```

Az i<>j (i nem egyenlő j) őr csak akkor sikeres, ha nem az utoljára kiszolgált processz akar belépni.

3. Készitsünk egy általános szemafort 100 felhasználó processz kiszolgálására:

```
szem:: sz:INTEGER; sz:=0;
  *[(i:1..100)x(i)?V() -> sz:=sz+1
  #(i:1..100)sz>0; x(i)?P() -> sz:=sz-1]
```

A felhasználó processzek a szem!P illetve szem!V utasitásokkal hajtják végre a szemafor müveleteket. Ha szem!P kiadásakor a szemafor számlálója (sz) nem nagyobb mint 0, akkor a szem!P-t kiadó processz várakozó állapotba kerül, hiszen a megfelelő x(i)?P utasitásra addig nem kerül sor, amig az előtte lévő logikai kifejezés igazzá nem válik.

4. Implementáljunk egy véges körpuffert CSP-ben:

```
X:: puffer::(0..9) pufferelem;
ki,be:INTEGER; be:=0;ki:=0;
*[be<ki+10;termelő?puffer(be MOD 10) -> be:=be+1
*ki<be;fogyasztó?még() ->
  fogyasztó!puffer(ki MOD 10); ki:=ki+1]
```

A termelő az X!p (p pufferelem tipusu) utasitással adja át az adatait, a fogyasztó az X!még() és X?(p) sorozattal veszi át. A fogyasztó ilyen megoldásának előnye, hogy a fogyasztó külön jelzést kap arról, hogy adat áll rendelkezésére, és ekkor az adat kiolvasásával párhuzamosan még továbbdolgozhat. Ha a be<ki+10 feltétel nem teljesül, akkor a termelő, ha a ki<be feltétel nem teljesül, akkor a fogyasztó várakozik.

5. [Hoa78]-ban számos példa található arra, hogy az eddig ismertetett fogalmak igen sok ismert nyelvi elem megvalósitására alkalmasak, vezérlési- és adatstruktúrákra egyaránt. Ezek közül a példák közül tekintsünk egyet, amely a halmaz tipust és két rajta végzett müveletet valósit meg (halmazba való felvétel és annak lekérdezése, hogy egy adott elem tagja-e a halmaznak).

```
H:: tartalom(0..99)INTEGER;méret:INTEGER;
    méret:=0;
  *[n:INTEGER;X?eleme(n) -> keress;X!(i<méret)
  #n:INTEGER;X?veddfel(n) -> keress;
        [i<méret -> SKIP
       #i=méret;méret<100 ->
            tartalom(méret):=n; méret:=méret+1
   ]    ]
```

A keress utasitás az alábbiak röviditése:

```
i:INTEGER; i:=0;
*[i<méret;tartalom(j) <> n → i:=i+1]
```

Vagyis keress addig fut, amig meg nem találja a kivánt értéket(n), vagy i értéke a méret fölé nem fut. A felhasználó X processz a H!veddfel(n) utasitással veteti fel az n-nek megfelelő elemet a halmazba. A lekérdezést a H!eleme(n) és H?b sorozattal végzi el, ahol b egy logikai változó, amelynek értéke igazzá válik, ha a kérdéses n már eleme a halmaznak. A H!eleme(n) hivással a hivó processz csak átadja azt az értéket, amelyet keresni kell, a tényleges keresés mind a két müveletnél a hivóval párhuzamosan fut.

A CSP nyelv alapján készült egy javaslat egy operációs rendszer lehetséges felépitésére, amely a kommunikáló processzek elvén alapul [HoK77]. A cikk egyszersmind szép példa a lépésenkénti finomitás [Wir71] és a hierarchikus felépitès [Dij68a,Dij68b] elveinek alkalmazására is.

2.2 A DP JAVASLAT

A DP (Distributed Processes) nyelvet Brich Hansen javasolta [BrH78]. A DP nyelv igen sok tekintetben hasonlit a CSP-re. Mindenekelőtt abban, hogy szintén drasztikusan csökkenti a nyelvi elemek és fogalmak számát. A nem-determinisztikus viselkedés kifejezésére a DP is a Dijkstra féle őrzött parancsokat alkalmazza, az eredetihez valamivel közelebb álló szintaxissal. Az őrzött parancsok mellett őrzött régiók is léteznek, utóbbiak elhalaszthatnak egy processzt, de az előbbiek nem. A párhuzamosságot itt is egymással versenyző szekvenciális processzek reprezentálják. A processzek közötti kommunikáció eszköze az ugynevezett közös eljárások hivása. Közös, globális adatok használata a DP-ben sem lehetséges!

A processz felépitése a következő:

PROCESS név
saját változók
közös eljárások
kezdeti utasitás

A processz csak a saját változóit érheti el. Minden processz hivhat közös eljárásokat, a sajátjait és máséit egyaránt. Egy másik processztól érkező hivást külső hivásnak nevezünk. A külső hivás formája:

CALL processznév.eljárásnév

Egy processz a külső hivások és a kezdeti utasitás végrehajtását átlapolva végezheti. Először mindig a kezdeti utasitást hajtja végre. Ezt végzi, amig az be nem fejeződik, vagy el nem kezd várakozni egy feltételre. Ekkor lehetséges egy külső hivás teljesitésének végrehajtása. Ez ismét addig fut, amig be nem fejeződik, vagy várakozni nem kezd. A kezdeti utasitás és a külső hivások átlapolása örökké folyhat. Ha a kezdeti utasitás befejeződik (nem végtelen ciklus jellegü), a processz továbbra is képes külső hivásokat végrehajtani. Egy processz örökké végzi müveleteit, kivéve, ha valamennyi lehetséges müvelete egy őrzött régióban vár, vagy ha a processz egy külső hivást adott ki. Az első esetben a processz addig vár, amig egy külső hivás ki nem mozditja várakozó helyzetéből, a második esetben pedig addig, amig a másik (a hivott) processz végre nem hajtotta a kérést. (Ez lényeges különbség a CSP-vel szemben, mert itt ki kell várni a kért müvelet befejeződését, mig a CSP-ben az input/output "találkozása" után a két processz már ismét párhuzamosan futhat [Wel79].) Mivel a DP szintaktikus jelölései közelebb állnak a közismert programnyelvek jelöléseihez, a továbbiakban csak az őrzött parancsokat és régiókat ismertetem, ezután a konkrét példák már könnyen érthetőek lesznek.

A feltételes parancsok formája:

```
IF b1:S1|b2:S2| ... END
DO b1:S1|b2:S2| ... END
```

Ha a b1,b2... logikai feltételek (őrök) közül valamennyi hamis, akkor az IF (feltételes) utasitás hibás (a hiba kezelésének módját a nyelv nem specifikálja), a DO (ciklus) utasitás pedig befejeződik (ciklusból való kilépés). Ha egyetlen feltétel igaz, akkor a hozzátartozó utasitás (Si), ha több mint egy feltétel igaz, akkor ezek közül egy tetszőlegeshez tartozó utasitás kerül végrehajtásra.

A kritikus régiók általános formája:

```
WHEN  b1:S1|b2:S2| ... END
CYCLE b1:S1|b2:S2| ... END
```

A WHEN utasitás addig vár, amig a b feltételek közül legalább az egyik igazzá nem válik. Ha több ilyen is van, akkor egy tetszőlegeshez tartozó utasitást hajt végre. A CYCLE utasitás a WHEN utasitás végtelen ismétlése. Tekintsünk néhány példát:

1. A szemafor implementálása DP-ben:

```
PROCESS szemafor; s: INT;
PROC P WHEN s>0: s:=s-1 END
PROC V s:=s+1
S:=0
```

A szemafor nevü processz egy saját változóval (s) rendelkezik. A P és V müveleteket egy-egy közös eljárás hajtja végre. P hivásakor ellenőrzi, hogy a hivő belépheti-e. Ha nem (s<=0), akkor várakozik, és ebből az állapotból csak a V eljárás viheti tovább. A processz kezdeti utasitása nem végtelen ciklus (s:=0), ezért a szemafor processz valójában pontosan olyan, mint egy monitor. A külső hivások formája: CALL szemafor.P és CALL szemafor.V.

2. A véges körpuffer:

```
PROCESS puffer; p:SEQ[n] CHAR
PROC betesz(c:CHAR) WHEN NOT p.full: p.put(c) END
PROC kivesz(#v:CHAR) WHEN NOT p.empty: p.get(v) END
p:=[]
```

A megoldásnál feltételeztük, hogy létezik egy SEQ nevü előredefiniált tipus [BrH77], amely szekvenciális sorozatot jelöl. A SEQ tipuson értelmezett a get és put müvelet (a soronkövetkező elem kivétele, illetve a sorozat végére való beirás), valamint az empty és a full logikai függvények, amelyek igaz értéket adnak, ha a sorozat üres, illetve megtelt. A puffer nevü processz kezdeti utasitása nem ciklikus (a p sorozatot üres kezdeti értékkel látja el), igy ez is monitorként müködik. A két közös eljárás a megfelelő feltételvizsgálat eredményétől függően várakozást irhat elő. (A # jel a kimenő paramétert jelzi.)

3. Készitsünk egy "ébresztőórát", amely lehetővé teszi a felhasználók számára, hogy egy időre elaltassák, majd ébresztessék magukat:

```
PROCESS ébresztő; idő:INT
PROC alvás(ennyit:INT)
eddig:INT
BEGIN eddig:=idő+ennyit
  WHEN idő = eddig: SKIP END
END
PROC óraütés; idő:=idő+1
idő:=ø
```

Az idő nevü INTEGER változó az ébresztő processz, az eddig nevü az alvás eljárás tulajdona. A SKIP utasitás üres utasitást jelent. Az alvás nevü eljárás beállitja eddig-be azt az időpontot, amikor ébreszteni kell. Ha ez nem egyenlő az aktuális idővel, akkor az őrben megadott feltétel hamis, tehát várakozni kell. A WHEN utasitás akkor fejeződik be, amikor a kért idő (ennyit) letelt. A hivó processz ekkor folytathatja müködését.

A DP nyelv alkalmazására számos további példa található [BrH78]-ban. A DP processz koncepciója (a CSP-hez hasonlóan) igen sokféle adat- és vezérlési struktura kiváltására alkalmazható. Ez itt is különösen akkor előnyös, ha a párhuzamosság igazi, tehát a különböző processzeket különböző fizikai processzorok hajtják végre. [BrH78]-ban példát találhatunk arra, hogy hogyan lehet DP-ben kifejezni az alábbi elemeket:

- eljárás (procedure);
- korutin;
- osztály (class);
- monitor;
- processz;
- szemafor;
- üzenet-puffer;
- utkifejezés (path expression);
- input/output.

2.3 A CSP ÉS A DP ÖSSZEHASONLITÁSA

Kiváló és nagyon tanulságos összevetés található [Wel79]-ben. Mielőtt a főbb különbségeket áttekintenénk, érdemes megnézni a hasonlóságokat. A CSP és DP javaslatok külön érdekessége, hogy szerzőik egyben a monitor koncepció fő megalkotói, éppen azé a koncepcióé, amely e javaslatokban háttérbe szorul, és csak mint az általános processz kommunikáció speciális esete fordul elő. A monitor koncepció háttérbe szorulásának ugy tünik elsősorban nem elvi, hanem gyakorlati okai vannak. A monitor implementációja közös tárral nem rendelkező processzorokra elég körülményes. O.J. Dahl Budapesten tartott előadása arra is fényt vetett, hogy a CSP verifikációval szemben mutatott képességei is jobbak, mint a monitoréi. A processz kommunikáció visszavezetése (késleltetett) értékadó utasitásra minden bizonnyal nagyszerü gondolat, még akkor is, ha implementációja 1 fizikai processzoros esetben rossz hatékonyságu lehet.

A CSP és DP közötti legföltünőbb különbség, hogy a CSP-ben a kommunikáló processzek kölcsönösen megnevezik egymást, a DP-ben csak a hivónak kell tudnia a hivott nevet. Ez első látásra komoly érv a DP mellett, hiszen egy programkönyvtár létrehozásakor ez nagy előny. Valójában a helyzet nem ilyen egyértelmü. Egyrészt példán is láttuk, hogy a CSP-ben is elég könnyü olyan processzt irni, amelyre kivülről egy processz tömb tetszőleges tagja hivatkozhat. Másrészt a DP-ben is az implementációnak nyilván szüksége van a processzek valamilyen megnevezésére, vagy legalább sorszámozására [BrH78]. Bizonyos alkalmazásoknál a processzeknek az alkalmazás jellegéből következően kell azonositaniuk egymást (például egy "job" ütemező esetén [Hoa78,

BrH78]), és ekkor a DP-ben a felhasználóra hárul az azonositás feladata, mégpedig nemcsak redundáns, de esetleg az implementáció azonositási mechanizmusának ellentmondó módon. Könnyü észrevenni, hogy a CSP input/output müveletei alacsonyabb absztrakciós szinten állnak, mint a DP közös eljárás hivásai. Ebből következne, hogy a CSP rugalmasabb, a DP viszont kényelmesebb és biztonságosabb. A következő példa azt mutatja, hogy ez nincs mindig igy, a párhuzamosság kifejezésében a CSP ebben a tekintetben is fölülmulhatja a DP-t. A CSP-re adott 5. példánk a halmaz tipus és rajta végezhető két müvelet megvalósitása volt. Ugyanez a feladat DP-ben a következő módon oldható meg:

```
PROCESS H
  tartalom: ARRAY[100] INT; méret,i: INT;
  felvételkérés: BOOLEAN; fölveendő: INT;

  PROC  keress(n:INT)
    BEGIN i:=1; DO (i<=méret) & (tartalom[i] <> n):
                   i:=i+1
  END            END

  PROC eleme(n:INT #válasz:BOOLEAN)
    BEGIN CALL H.keress(n);
          válasz:=i<=méret
  END

  PROC veddfel(n:INT)
    BEGIN felvételkérés:=TRUE; fölveendő:=n
  END

  BEGIN  méret:=0; felvételkérés:= FALSE
    CYCLE felvételkérés:
      CALL H.keress(fölveendő);
      IF i<=méret: SKIP |
         (i>méret) & (méret<100):
          méret:=méret+1; tartalom[méret]:=fölveendő
      END;
      felvételkérés:= FALSE
    END
  END
```

A DP-ben a hivó processz addig várakozó állapotban van, amig a hivott közös eljárás vissza nem tér. Az eleme eljárás hivásakor tehát a hivó processz a keresési müveletet végigvárja. (Ez elfogadhatónak tünik, mert ilyenkor valószinüleg ugyse tud mit csinálni a hivó.) A felvétel idejére azonban mindenképpen kivánatos, hogy az időigényes keress eljárás és a halmazba való fölvétel a hivóval párhuzamosan fusson. A CSP rendkivül elegáns megoldásával szemben, a DP-ben ehhez két uj változót kellett bevezetni (felvételkérés és fölveendő), és a program egész struktúrája jóval bonyolultabbá vált. Annyira, hogy a közölt megoldás nem is hibátlan, mert nincs biztositva, hogy egy hivás után a kezdeti utasitásként megadott ciklus előbb lefut, mint egy következő hivás meg nem érkezik. Ezért a közös eljárások bemenetét explicite védeni kell egy

WHEN NOT felvételkérés:

őrrel. Ezt a könnyen elkövethető hibát az okozta, hogy a DP-ben a nem determinisztikus viselkedések leirása közel sem olyan egyértelmü, mint a CSP-ben. A CSP-ben az őrzött parancsok jelentik az egyetlen nem determinisztikus müveletet. A DP-ben ez csak látszólag van igy. A processz (ciklikus) kezdeti utasitása és a hivások közötti oszcilláció ugyanis még egy nem determinisztikus viselkedést visz a processz viselkedésébe. A kezdeti utasitásnak ráadásul "felemás" prioritása van, mert ha egyszer megszerezte a vezérlést, akkor addig nem engedi érvényre jutni a külső hivásokat, amig van futásképes alternativája, de ha egy külső hivás érvényre jut, akkor annak befejezése vagy felfüggesztése után nincs prioritása a hivásokkal szemben. Mind a CSP mind a DP nem determinisztikus vezérlési strukturákat ad a determinisztikus feltételes és ciklikus müve-

letek kifejezésére is. A CSP esetében azonban egy formális bővitéssel bevehető lenne determinisztikus IF vagy WHILE utasitás. A DP-ben ez elvi probléma, az előbb emlitett okok miatt. [Wel79]-ben a két javaslat további igen alapos elemzése, kvalitativ és kvantitativ összevetése található. A szerzők kimutatják, hogy bár a DP processz koncepciója magában foglalja a monitort, a CSP-nek a monitor funkcióját (és nem formalizmusát) megvalósitó egyetlen koncepciója sokkal világosabb. A végső következtetés az, hogy a CSP jobban választotta meg az absztrakciós szintet, mint a DP; kevesebbet markolt, de többet fogott.

3. NÉHÁNY MAGASSZINTŰ NYELV

A dolgozat egyik fő célja, hogy bemutassa, hogy az ujabb keletü programozási nyelvek mennyiben alkalmasak bonyolult és párhuzamosságot is tartalmazó rendszerek tervezésére és megvalósitására. A bonyolultság leküzdésének fő eszköze a dekompozició, vagyis a feladat részfeladatokra való bontása. A párhuzamosság kezelésének eszközeit elvben már áttekintettük.

A most bemutatásra kerülő programnyelveket elsősorban ebből a két szempontból vizsgálom meg és hasonlitom össze. Egy programnyelv jóságának megitélése ennél persze összetettebb feladat, már csak azért is, mert végső soron csak a gyakorlati alkalmazások során alakulhat ki meggyőző kép. Egy gyakorlati, és a triviálisnál bonyolultabb feladat megoldása található a következő fejezetben.

Az ismertetett programnyelvek mindegyike mutat némi rokonságot az Algol/60 nyelvvel [Loc67]. Ezen kivül mindegyik ismertetett programnyelvben megjelenik valamilyen formában a tipus koncepció. A tipus koncepció az Algol/68 tervezésekor merült fel, és a PASCAL nyelv révén terjedt el széles körben [JeW74]. A modularitás kifejezésére kinált megoldások közös ősе minden bizonnyal a Simula/67 [Dah78] osztály koncepciója. A tipus és osztály részletes és elmélyült tárgyalása található [Dah78]-ban.

3.1 KONKURRENS PASCAL

A Konkurrens Pascal [BrH77] volt az első olyan nyelv, amely komoly eredményt ért el abban, hogy a programozót segitse megbizható konkurrens programok irásában. A Konkurrens Pascal-t a hetvenes évek elején tervezték, a fordító már 1974-ben rendelkezésre állt, egyetlen ember irta (A. Hartmann) hét hónap alatt, PASCAL-ban. A Konkurrens Pascal a PASCAL nyelvre [JeW74] épül, kiegésziti azt mindenekelőtt a szekvenciális processz és a monitor fogalmával. A fordító képes biztositani, hogy egy processz saját adatait más processz nem érheti el. A processzek közötti kommunikáció csak monitorokon keresztül lehetséges. A Konkurrens Pascal nyelv tehát nemcsak eszközt ad a párhuzamosságok kifejezésére (mint mondjuk a PL/1), hanem ellenőrzi is bizonyos mértékig azok helyes használatát. Ez döntő jelentőségü lépés a párhuzamos rendszerek tervezésében, mert az ilyen és az azóta keletkezett hasonló nyelvek teszik lehetővé, hogy a párhuzamosságra vonatkozó szabályok betartásában a programozó segitséget kapjon egy fordítótól, és igy esélye legyen arra, hogy a futási időben ne kelljen megmagyarázhatatlan, szórványosan fellépő hibákra vadásznia.

A Konkurrens Pascal bevezette a rendszer tipus fogalmát. Ide tartozik az osztály, a monitor és a processz. Ezek közös tulajdonsága, hogy környezetükkel való kapcsolataikat, elérési jogaikat ugynevezett paraméterek formájában adják meg. Rendelkezhetnek saját objektumokkal; konstansokkal, tipusokkal, változókkal, eljárásokkal. Definiálhatnak a környezet számára elérhető eljárásokat (bemeneteket),

és olvasási célra láthatóvá tehetik változóikat (ENTRY rutinok és változók). A rendszer tipusu változókat egy inicializáló utasitással kell elinditani, ezután a rendszer tipusu változó saját változói és paraméterei örökre léteznek (permanensek). Az eljárások belső változói az eljárás befejeződésekor megszünnek (mint például az Algol/60-ban). Tekintsük át a rendszer tipusokat, és a Konkurrens Pascal-nak a PASCAL-tól eltérő egyéb sajátságait:

3.1.1 AZ OSZTÁLY

Az osztály a modularitás kifejezésének egyik eszköze, koncepciója a Simula/67 [Dah78] osztály fogalmára épül. Az osztály absztrakt adatstrukturák [Dah78] definiálására szolgál. Az osztályok tetszőleges mélységig egymásba skatulyázhatók, de a nyelv szintaktikus szabályai biztositják, hogy egy osztály(lánc) csak egyetlen processzhez tartozhat (egy osztályt egy rendszer tipuson belül kell deklarálni, és csak további osztályoknak lehet permanens paraméterként átadni). Az osztály tehát arra szolgál, hogy egyetlen processz teendőit egymás mellé- és alárendelt absztrakt adatstrukturák halmazával irhassuk le. Egy osztály bemeneteit csak egyetlen rendszer tipusu változó hivhatja.

3.1.2 A MONITOR

A monitor az osztályhoz teljesen hasonlóan absztrakt adatstrukturák definiálását teszi lehetővé. Ezen tulmanően, biztositja a saját bemeneteire a kölcsönös kizárást. Egy monitor bemeneteit több rendszer tipusu változó is hivhatja, de amig egy processz aktivan (nem várakozva) a monitor egyik

bemenetét hajtja végre, addig a többi processz nem léphet be a monitorba. A nyelv szintaktikus szabályai azt is biztositják, hogy a processzek csak monitor eljárásokon keresztül tudnak kommunikálni (processznek csak konstans, vagy monitor lehet permanens paramétere).

3.1.3 A PROCESSZ

A processz egy szekvenciális program, amely tartalmazhat végtelen ciklust is. Egy processz bemenetét csak egy, az ugyanahhoz a processz tipushoz tartozó processz által végrehajtott szekvenciális program hivhatja.

3.1.4 A QUEUE TIPUS

A QUEUE tipus a monitor-elv általános tárgyalásánál leirt CONDITION tipus alapján áll. A rajta végezhető müveletek: EMPTY(q) (logikai függvény, értéke igaz, ha a q QUEUE tipusu változóra nem vár senki), DELAY(q) (a hivó processz várakozni kiván) és CONTINUE(q) (a hivó kilép a monitor rutinból és egyszersmind azonnali hatállyal továbbinditja a q-ra várakozó processzt, ha van ilyen). A QUEUE tipus sajátsága, hogy csak egyetlen processz várakozhat benne, tehát két egymást követő DELAY között egy CONTINUE-t is ki kell adni. QUEUE tipusu változót csak monitorban lehet deklarálni. Ez a szabály biztositja, hogy a középtávu ütemezés monitorokra korlátozódjék.

Izelitőül a már jól ismert véges körpuffert vegyük példaképpen (a Konkurrens Pascal és a párhuzamosság, valamint a dekompozició egészen magasszintü példáját találhatjuk [Joa77]-ben, ahol egy nyilvános hálózatra való csatlakozást biztositó programrendszert ismertet a szerző):

```
TYPE buffer = MONITOR

VAR content: line; full: BOOLEAN;
    sender,receiver: QUEUE;

PROCEDURE ENTRY receive(VAR text: line);
BEGIN
  IF NOT full THEN DELAY(receiver);
  text:= content; full:= FALSE;
  CONTINUE(sender);
END;

PROCEDURE ENTRY send(text: line);
BEGIN
  IF full THEN DELAY(sender);
  content:= text; full:= TRUE;
  CONTINUE(receiver);
END;

BEGIN full:= FALSE END;
```

Figyeljük meg ezen a példán is, hogy a CONDITION jellegü szinkronizációs müveleteket önmagában nem használjuk, ezt mindig célszerü összekötni valamilyen statikus információval (a példában ez a full). Ennek az oka világos; ha előbb adjuk ki a CONTINUE-t mint a DELAY-t, akkor az hatástalan, tehát egy feltételhez nem kötött DELAY utasitás patt-helyzethez vezethet. A példában definiált buffer tipust tetszőleges számu változó definiálására használhatjuk, például:

VAR buf1,buf2: buffer

Ezeket a változókat az inicializáló utasitással "indithatjuk el":

INIT buf1,buf2

Ettől a pillanattól kezdve két buffer tipusu változó létezik, mindegyik külön permanens változókkal. A pufferekre ezután
buf1.send(text)
buf1.receive(text)
alaku utasitásokkal lehet hivatkozni.

3.2 A MODULA ÉS A MODULA-2

A MODULA nyelvet [Wir77a-c] és utódját a MODULA-2 nyelvet [Wir80] N. Wirth definiálta. A továbbiakban alapvetően a MODULA-2 nyelvet ismertetem, de hivatkozásokat teszek a MODULA-ra is. Helyenként tanulságos látni, hogy a MODULA egyes sajátosságai hogyan fejlődtek tovább.

A MODULA nyelv fő célja az volt, hogy olyan eszközt adjon a programozó kezébe, amely egyrészt megfelel a strukturált programozás követelményeinek és jó minőségü, jól olvasható programok irását teszi lehetővé, másrészt "hadat üzen" a gépi, Assembly szintü nyelveknek. Ez azt jelenti, hogy a nyelvnek hozzáférést kell biztositania a számitógép fizikai erőforrásaihoz, anélkül, hogy a program strukturált felépitésében törés keletkezne. Egyszersmind arra is törekedni kell, hogy a fordító által generált kód hatékonysága elfogadható legyen. Amint ezt Wirth [Wir81]-ben kimutatta, ez utóbbi kérdés erősen függ a számitógép architekturájától. A számitógépek tervezői mindmáig az Assembly szintü programozót tartják szem előtt, ezzel rendkivüli módon megnehezitve a fordító irók dolgát. Számos jel van azonban arra, hogy a közeljövőben a piacon is megjelennek olyan architekturáju gépek, amelyek már meglévő kisérleti rendsze-

rekhez hasonlóan [Wir81,Lam81,LaP81] a magasszintü nyelvek igényeinek figyelembevételével készülnek. A Palo Altoban épitett Alto gép csaknem teljes software rendszerét MESA nyelven [Mit79] irták, a Zürichben tervezett Lilith géphez [Wir81] pedig nem is készült Assembly szintü forditó (a kezdeti tesztelést leszámitva), a teljes rendszer MODULA-2-ben készült, illetve készül a jövőben is.

3.2.1 MODULARITÁS

A MODULA és MODULA-2 nyelvekben a modularitás és információ elrejtés eszköze a modul. A modul egy szintaktikus falat emel a modul belseje és környezete közé, amelyen keresztül csak az export/import szabályok betartásával lehet átjutni. A modul kifejezetten csak a forditási időben befolyásolja a hozzáférési szabályokat; egy leforditott programban nem lehetne megtalálni a modulok helyét. A modul általános formája konstansok, tipusok, változók és eljárások deklarációjából, valamint egy kezdeti utasitásból áll. A kezdeti utasitás akkor kerül végrehajtásra, amikor a modult tartalmazó eljárás (az inditó rendszer is egy pszeudo eljárásnak számithat) aktivvá válik. Ezután a modul saját változói (amelyek lokálisak a modulra, de nem lokálisak a modul eljárásaira nézve) mindaddig léteznek, amig a modult tartalmazó eljárás aktiv. A modul egyes eljárásainak is lehetnek saját (lokális) változói, és ezekre a például az Algol/60-ban vagy a PASCAL-ban meglévő szabály érvényes: ha az eljárás befejeződik, akkor az eljárás lokális változói megszünnek, uj hiváskor ujra keletkeznek, előző értéküket elvesztve. A modul változókra azonban nem ez a szabály érvényes, ezek megmaradnak azután is, ha a modul valamelyik eljárása befejeződött. A modul maga nem adható meg tipus-

ként, és nincs rá mód, hogy egy modul lokális adatait megsokszorozzuk, anélkül, hogy a teljes modult (program-kóddal együtt) meg ne sokszoroznánk. Ezért az ilyen jellegü feladatok megoldására kerülő megoldást kell keresni. Ebben segithet bizonyos esetekben az eljárás(PROCEDURE) tipus léte, amelynek segitségével a szóbanforgó absztrakt adatstruktúrát nem mint modult, hanem mint RECORD-ot irjuk le, és a benne definiált eljárás tipusu mezők dinamikusan változtatják értéküket. Egy másik megoldás, ha a modul változóit egy dimenzióval megnöveljük, amely a modul sokszorozási információt hordozza (erre láthatunk később példát a TTYS modulban). Ennek az elvnek egy elegáns továbbfejlesztése, hogy a modul változóit egyetlen rekordba fojuk össze, és a sokszorozási információt egy POINTER-tömb hordozza. Ebben az esetben a hivatkozási helyeken a WITH utasitást alkalmazhatjuk, és igy ott ezután már nincs különbség az egy- és a több-példányos modulok kezelése között. A modul objektumait általában csak a modulban definiált eljárások érhetik el. Ez alól a szabály alól kivételt képez az az eset, amikor egy modul valamilyen objektumot (konstanst, tipust, változót, eljárást) exportál, a külvilág számára láthatóvá tesz. Az ilyen objektumok elérhetővé válnak mindazon modulok számára, amelyek őket importálják. A MODULA-2 az eddig leirt általános modul koncepciót a definiciós/implementációs modul fogalmával egésziti ki. Ez azt jelenti, hogy egy modult két részre oszthatunk: az egyik rész, a definiciós modul, tartalmazza azokat a konstansokat, tipusokat, változókat és azoknak az eljárásoknak a fejét, amelyeket exportálni kivánunk. A definiciós modul igy a modulnak a környezet felé mutatott interface-ét tartalmazza. Az exportált eljárások (és bizonyos esetekben tipusok) kifejtését az implementációs modul tartalmazza. A definició és implementáció ilyenfajta szétválasztása lehetőséget teremtett

arra, hogy a MODULA-2-ben a modulok különforditását teljes elvi tisztasággal lehessen megvalósitani [Gei79]. A külön fordítás egysége a modul. A fenti elv lehetővé teszi, egyrészt, hogy a fordító könnyen elvégezhesse a külön forditott modulok között is a tipusellenőrzést (pontosan ugy, mintha együtt fordultak volna), másrészt, hogy bármelyik modul implementációs részét bármikor ujrafordithassuk, a többi modul ujraforditása nélkül. A modulok használatára tekintsünk most egy példát. Készitsünk egy olyan modult, amely érkezési sorrend szerinti sorbaállitást végez (FIFO QUEUE):

```
DEFINITION MODULE FIFOQ;

FROM SYSTEM IMPORT WORD;
EXPORT QUALIFIED QUEUE,CREATEQ,EMPTYQ,FULLQ,
        GETQ,PUTQ;

TYPE QUEUE;

PROCEDURE CREATEQ(VAR Q: QUEUE; LENGTH: INTEGER);
(*CREATES A NEW QUEUE HEAD*)

PROCEDURE EMPTYQ(Q: QUEUE): BOOLEAN;
(*GIVES TRUE IF THE Q IS EMPTY*)

PROCEDURE FULLQ(Q: QUEUE): BOOLEAN;
(*GIVES TRUE IF THE Q IS FULL*)

PROCEDURE GETQ(VAR Q: QUEUE; VAR THIS: WORD);
(*TAKES THE NEXT WORD FROM Q IF POSSIBLE*)

PROCEDURE PUTQ(VAR Q: QUEUE; THIS: WORD);
(*PUTS WORD IN Q IF POSSIBLE*)

END FIFOQ.
```

A modulban először egy import lista található, ezen a modul a SYSTEM nevü modulból [Wir80] importálja a WORD tipust. A WORD tipusnak az a különös sajátsága van, hogy ha egy eljárás formális paraméterének ilyen a tipusa, akkor a hivó helyen tetszőleges 1 szó hosszu aktuális paraméter állhat. A FIFOQ modul igy tetszőleges tipusu 1 szó hosszuságu elemek sorkezelésére alkalmas. Ezután következik az exportált objektumok felsorolása. A QUALIFIED minősités lehetővé teszi, hogy a külső hivó a modulnévvel kapcsolja össze az importált objektumok nevét, és igy a névütközést kizárja. A definiciós modul exportálja a QUEUE tipust, és a rajta végezhető müveleteket: CREATQ (tetszőleges maximális hosszuságu sor létrehozása), FULLQ (logikai függvény, igaz értéket ad, ha a sor megtelt), EMPTYQ (logikai függvény, igaz értéket ad, ha a sor üres) GETQ (kivesz a sorból egy elemet), és végül PUTQ (betesz a sorba egy elemet). A tipus listán (TYPE) a QUEUE tipusnak csak a neve szerepel. Ez egy ugynevezett rejtett tipus, felépitése a felhasználók számára rejtve marad. A QUEUE tipuson értelmezett müveletek definiciója következik ezután, ebből a hivás szabályai pontosan kiolvashatók. A fenti definicióhoz tartozó implementáció a következő:

```
IMPLEMENTATION MODULE FIFOQ;

FROM Storage IMPORT ALLOCATE,DEALLOCATE,SetMode;
FROM SYSTEM IMPORT WORD;

TYPE QUEUE = POINTER TO QHEAD;
     QP    = POINTER TO QPART;
     QHEAD = RECORD
               FIRSTELEM: QP;
               FREEPLACE,FULLPLACE: INTEGER;
             END; (*RECORD*)
     QPART = RECORD
               NEXTELEM: QP;
               INFO: WORD;
             END; (*RECORD*)

PROCEDURE CREATEQ(VAR Q: QUEUE; LENGTH: INTEGER);
(*CREATES A NEW QUEUE HEAD*)
BEGIN  NEW(Q);
       WITH Q^ DO
  FIRSTELEM:= NIL; FULLPLACE:= 0;
```

```
  IF LENGTH > 0 THEN FREEPLACE:= LENGTH
  ELSE FREEPLACE:= 0 END;
  END; (*IF*)
       END; (*WITH*)
END CREATEQ;

PROCEDURE EMPTYQ(Q: QUEUE): BOOLEAN;
(*GIVES TRUE IF THE Q IS EMPTY*)
BEGIN RETURN Q^.FULLPLACE = 0
END EMPTYQ;

PROCEDURE FULLQ(Q: QUEUE): BOOLEAN;
(*GIVES TRUE IS THE Q IS FULL*)
BEGIN RETURN Q^.FREEPLACE = 0
END FULLQ;

PROCEDURE GETQ(VAR Q: QUEUE; VAR THIS: WORD);
(*TAKES THE NEXT WORD FROM Q IF POSSIBLE*)
VAR FIRST: QP;
BEGIN  WITH Q^ DO
  IF FULLPLACE <= 0 THEN RETURN END; (*CANNOT GET OUT*)
  DEC(FULLPLACE); INC(FREEPLACE);
  FIRST:= FIRSTELEM; FIRSTELEM:= FIRST^.NEXTELEM;
  THIS:= FIRST^.INFO; DISPOSE(FIRST);
       END; (*WITH*)
END GETQ;

PROCEDURE PUTQ(VAR Q: QUEUE; THIS: WORD);
(*PUTS WORD IN Q IF POSSIBLE*)
VAR LAST: QP;
BEGIN  WITH Q^ DO
  IF FREEPLACE <= 0 THEN RETURN END; (*CANNOT PUT IN*)
  DEC(FREEPLACE); INC(FULLPLACE);
  IF FIRSTELEM = NIL THEN NEW(FIRSTELEM); LAST:= FIRSTELEM
  ELSE LAST:= FIRSTELEM;
    WHILE LAST^.NEXTELEM # NIL DO LAST:= LAST^.NEXTELEM END;
    NEW(LAST^.NEXTELEM); LAST:= LAST^.NEXTELEM;
  END; (*IF*)
  LAST^.NEXTELEM:= NIL; LAST^.INFO:= THIS;
       END; (*WITH*)
END PUTQ;

END FIFOQ.
```

Az implementációs modul importálja a Storage modul [Wir80] tárfoglaló müveleteit. A programban előforduló NEW(p) alaku utasitások ALLOCATE(p,TSIZE(T)), a DISPOSE(p) alakuak pedig DEALLOCATE(p,TSIZE(T)) utasitássá fordulnak, ahol p egy a T tipusra mutató POINTER változó és a TSIZE standard függvény a T tipus méretét adja. NEW és DISPOSE igy dinamikus tár foglalásra, illetve felszabaditásra használható. Az implementációs modul pontosanspecifikálja a QUEUE tipust, amely, mint látható, egy sorfejre (QHEAD) mutató pointer. Maga a sorfej három elemből áll, a sor első elemére mutató FIRSTELEM nevü pointerből, és két számlálóból, amelyek az üres, illetve telehelyek számát mutatják. Egy sorelem (QPART) két részből áll; a következő elemre mutató pointerből (NEXTELEM) és magából a besorolni kivánt információból (INFO). A CREATQ eljárás tárterületet foglal a sorfej számára, és beállitja a sorfej elemeit. A FIRSTELEM NIL értéket kap, ami azt jelenti, hogy "sehova se" mutat. [JeW74]. Az EMPTYQ és FULLQ függvények müködése azonnal látható. GETQ először ellenőrzi, hogy van-e kisorolható elem a sorban. Ha nincs, akkor visszatér, a THIS kimenő paraméter értéke ilyenkor definiálatlan, ez tehát a hivó felelőssége, hogy ilyen hivás ne forduljon elő. GETQ ezután kisorolja az első elemet és a felszabadult tárterületet visszaadja a dinamikus tárkezelőnek. Ez megbocsáthatóvá teszi a különben meglehetősen tárigényes adatábrázolást. PUTQ GETQ-hoz teljesen hasonlóan müködik, a besorolandó uj elemnek tárat foglal (ha a sor hossza megengedi). A FIFOQ modul nem tartalmaz a sorfej törlésére vonatkozó szolgáltatást, de ezzel bármikor kiegészithető. A modulon kivül azonban lehetetlen kitörölni egy sort, mert a QUEUE tipuson semmilyen más müveletet nem lehet végezni, mint amelyeket a

FIFOQ modul exportál. Ez a példa is jól mutatja az információ elrejtés hallatlan előnyét: ha egyszer a FIFOQ modult önmagában hibátlannak itélhetjük, akkor biztos, hogy kivülről már nem lehet az adatstrukturáját elrontani.

3.2.2 PÁRHUZAMOSSÁG

A párhuzamosság kezelésében a MODULA [Wir77a-c] és a MODULA-2 [Wir80] erősen eltér egymástól. A MODULA-ban alkalmazott megoldás meglehetősen hagyományos, rokon a Konkurrens Pascal [BrH77] vagy a MESA [Mit79] párhuzamos tulajdonságaival.

A MODULA-ban a párhuzamosság alapvető egysége a processz. A processz formálisan igen hasonlit egy eljárásra (mint majd látjuk, a MODULA-2-ben a processz és az eljárás szintaktikusan már nem tér el). Egy processz több példányban is elinditható, ezek algoritmusa azonos, de dinamikus müködése természetesen egyedi (minden processz-példány saját adatterülettel rendelkezik). A kölcsönös kizárás (a rövidtávu ütemezés) biztositására a MODULA-ban az interface modul áll rendelkezésre. Az interface modul biztositja, hogy amig egy processz aktivan benne tartózkodik (vagyis valamelyik eljárását hajtja végre), addig más processz oda belépni nem tud. A következő processz akkor léphet be, ha az előző befejezte az interface modul eljárást, vagy ideiglenesen elhagyta az interface modult (WAIT vagy SEND révén). Látható, hogy az interface modul eddigi definiciója megfelel a monitornak [Hoa73]. A processzek középtávu ütemezésére, szinronizálására a SIGNAL tipus és a rajta végezhető SEND, WAIT és AWAITED müveletek állnak rendelkezésre.

A WAIT(s) alaku utasitás révén egy processz az s SIGNAL-ra várhat. A SEND(s) alaku utasitás azonnal elinditja az s-re várakozó processzek közül az elsőt (a legrégebben várakozót). Ha egyáltalán nincs s-re váró processz, akkor SEND(s) hatástalan. Az AWAITED(s) logikai függvény, amely igaz értéket ad, ha az s SIGNAL-ra egy vagy több processz vár. A WAIT és a SEND ezen felül szinguláris pontot jelent az interface modulban a kölcsönös kizárás szempontjából, vagyis ha egy processz WAIT-et vagy SEND-et hajt végre, akkor lehetővé válik, hogy egy következő processz belépjen az interface modulba. A monitor Hoare féle definciójában [Hoa73] csak a WAIT jelent a fenti értelemben vett szinguláris pontot, a SEND-nek megfelelő müvelet nem. Ennek az a háttere, hogy Hoare egy olyan implementációt javasol, amelyben a SEND-et végrehajtó processzt az első lehetséges alkalommal tovább kell folytatni. (Egy 1 fizikai processzorra épülő implementációban a SEND implicit várakozást jelent, bár a SEND-et kiadó processz aktiv állapotban marad.) Wirth ezzel szemben olyan implementációt alkalmazott a MODULA-ban [Wir77c], amelyben a SEND-et kiadó és igy aktiv állapotban maradó, de mégis ideiglenesen várakozó processzt (egy-processzoros implementáció) nem inditja ujra az első lehetséges alkalommal. Ezért nyugotan meg lehet engedni, hogy a SEND is feloldja a kölcsönös kizárást. Annál is inkább, mert az irodalomban idézett példákban mindig, és a gyakorlatban is általában a SEND az utolsó utasitás az interface modul (monitor) eljárásban, ami ugyis feloldja a kölcsönös kizárást. Ez a választás drasztikusan egyszerüsitette az interface modul implementációját, rejtett processz-átkapcsolás nincs, csak a jól definiált WAIT és SEND pontokon van processz-átkapcsolás. A kölcsönös kizárás biztositása is

triviális egy 1 fizikai processzoros egységen, hiszen a WAIT és SEND jelenti az egyetlen lehetőséget a processz-átkapcsolásra, amely amugy is feloldja a kölcsönös kizárást. A megszakitásoktól természetesen eltekintettünk a fenti megfontolásokban, ezeket a MODULA az input/outputtal kapcsolatban kezeli le. Az input/output kezelésére a MODULA nyelv főleg implementáció függő tulajdonságok bevezetésével ad módot. Lehetővé teszi, hogy a készülék vezérlő processzekben (driverekben) a MODULA programozó közvetlenül hozzáférjen az input/output regiszterekhez. A készülék vezérlő processzek különleges interface modulokban az ugynevezett device modulokban helyezkedhetnek el. A készülék vezérlő processzek a készüléknek megfelelő prioritással futnak, az alacsonyabb szintü megszakitásokat kizárva. Az általuk kiadott SEND különlegesen müködik, nem inditja azonnal a SIGNAL-ra váró processzt, hanem tovább fut, amig csak várakozási müveletet nem kezdeményez. A készülék vezérlő processz kétféle várakozást tartalmazhat, a már ismertetett WAIT utasitást és az input/output müvelet befejeződését kiváró DOIO utasitást. A DOIO-t az input/output kezdeményezése után adja ki, és a DOIO akkor fejeződik be, amikor az adott készülék megszakitással jelezte az input/output végét. A DOIO tehát egy különleges WAIT utasitásnak számit, amely a készülékhez tartozó megszakitásra vár mint SIGNAL-ra. Mielőtt példát vennénk a MODULA párhuzamos lehetőségeinek használatára, áttekintjük a MODULA-2 hasonló sajátságait. Látni fogjuk, hogy a MODULA koncepciói kifejezhetőek MODULA-2-ben, s igy a kettőre együtt vehető példa.

A MODULA-2 a párhuzamosság támogatására alapvető változtatásokat eszközölt, jelentősen csökkentette az absztrakció szintjét. A párhuzamosság alapvető egysége továbbra is a processz, de ez valójában korutin értelemben [Dah78, Wir80]. A korutin abban különbözik a processztől, hogy a korutinok között nem tételezünk fel teljes párhuzamosságot, a korutinok közötti vezérlésátadás csak explicit kérésre történik. A MODULA-ra tett megfontolásokból már érezhető volt, hogy a MODULA törekedett egy olyan megfogalmazásra, amely kihasználhatja azt a körülményt, hogy a nyelvet előreláthatóan 1 fizikai processzorra implementálják. Ezt a burkolt feltételezést a MODULA-2 nyiltan vállalja. A MODULA-2-ben a hardware-prioritást nem a készülék meghajtó processzhez, hanem az azt tartalmazó modulhoz kell hozzárendelni. A processzek korutinként való kezelése maga után vonja, hogy a kölcsönös kizárás megvalósitása az azonos hardware szinten definiált modulokra nézve triviális. A különböző hardware szinten definiált modulokra nézve a hardware által nyujtott lehetőségek - maszkolás, processzorprioritás - biztosithatják a kölcsönös kizárást, az egyetlen szabály, hogy egy magasabb szintü modulból egy alacsonyabb prioritási szintü modul eljárásait közvetlenül meghivni nem szabad! A MODULA-2-ből kimaradt az interface és a device modul; bármelyik modul annak tekinthető. A processzek szintaktikusan azonosak az eljárásokkal, egy eljárásból külön utasitással lehet processzt létrehozni. (Csak paraméter nélküli, globális eljárásokból lehet processzt létrehozni.) Egy eljárásból a következő utasitással lehet processzt létrehozni:

```
NEWPROCESS(p: PROC; adr: ADDRESS;
           n: CARDINAL; var st: PROCESS)
```

A p eljárás-tipusu(PROC) paraméter a processzként (korutinként) létrehozandó eljárás, adr a processz rendelkezésére álló munkaterület cime, n a hossza. Az ilyen módon létrehozott processz hozzárendelődik az st nevü változóhoz (st a processz adat-stackjére mutat, amely NEWPROCESS hatására megfelelően inicializálódik is), de nem indul el. A processzek közötti átkapcsolás céljára a következő utasitás használható:

TRANSFER(VAR old,new: PROCESS)

A TRANSFER eljárás felfüggeszti az éppen futó processzt és old-hoz rendeli (stack-jébe lementi a regisztereket), majd a new által kijelölt processzt elinditja (a stack tetejéről betölti a regisztereket). A programozó felelőssége hogy new-t már egy előző NEWPROCESS vagy TRANSFER hozzárendelje egy processzhez. (Old és new kijelölheti ugyanazt a processzt is.) Az input/output kezelésére egyrészt továbbra is megvan az input/output regiszterekhez való hozzáférés lehetősége (implementáicó függő módon), valamint az

IOTRANSFER(VAR p1,p2; va: CARDINAL)

utasitás. Az IOTRANSFER a TRANSFER-rel analóg módon viselkedik (tehát az éppen futó processz futását felfüggeszti és p1-hez rendeli, a p2-vel kijelölt processzt pedig elinditja), de ezen tulmenően a megszakitás bekövetkezése után a megszakitott processzt p2-höz rendeli és továbbfolytatja p1-et, vagyis a készülék vezérlőt. A va (interrupt vector address) azt a cimet adja meg, ahová a megszakitás számára szükséges információkat (program-státusz és program-számláló) le kell menteni. A megszakitás után tehát a vezérlés

az IOTRANSFER utáni pontra adódik. A NEWPROCESS, TRANSFER és IOTRANSFER eljárásokat, valamint a PROCESS tipust a SYSTEM nevü (peszudo) modulból kell importálni. Az eljárások könnyen megvalósithatók mikroprogram segitségével is [Wir81].

Az ismertetett MODULA-2 alapmüveletek közvetlenül is használhatók input/output és processz-átkapcsolás végzésére, erre példa található [Wir80]-ban. Igazi jelentőségük azonban abban van, hogy eszközt adnak arra, hogy tetszőleges bonyolultságu ütemezőt készithessünk. A MODULA-2-nek az ütemezésre vonatkozó alapvetően uj koncepciója tehát az, hogy a magasszintü nyelv ne tartalmazzon semmilyen beépitett ütemezőt, de adjon eszközt annak megirására. Ezzel együttjár az is, hogy a MODULA-2-ben nagyobb a programozó szabadsága, de ezért cserébe kisebb a biztonsága. A MODULA-ban lehetőség volt rá, hogy a fordító ellenőrizze, hogy a szinkronizációs müveletek kiadása az interface modulokra korlátozódjék. MODULA-2-ben ez már csak ajánlott programozói szabály, amelynek betartását lehetetlen ellenőrizni. Ütemezőre találhatunk példát [Hop80]-ban, ahol egy üzenet orientált implementáció látható. Egy ütemező megirására bemutatjuk a MODULA beépitett ütemezőjének funkcióit megvalósító MODULA-2 program [Wir80] teljes, ténylegesen használható változatát:

```
DEFINITION MODULE PROCESSSCHEDULER;
(*N.W., Ch.J., S.E.K., HH.N., L.B. 30-Jan-81 *)
  FROM SYSTEM IMPORT ADDRESS;

  EXPORT QUALIFIED SIGNAL, STARTPROCESS, SEND,
         SENDDOWN,AWAITED,
         WAIT, DOJO, PAUSE, INITSIGNAL;

  TYPE SIGNAL;
   (* SIGNAL's must be initialised by INITSIGNAL *)

  PROCEDURE STARTPROCESS(P: PROC; A: ADDRESS; n: CARDINAL);
    (* start P with workspace A of length n  *)
```

```
  PROCEDURE AWAITED(S: SIGNAL): BOOLEAN;

  PROCEDURE SEND(VAR s: SIGNAL);
    (* resume first process waiting for s *)

  PROCEDURE SENDDOWN(VAR s: SIGNAL);
    (* mark first process waiting for s as ready *)

  PROCEDURE WAIT(VAR s: SIGNAL);

  PROCEDURE DOIO(va: CARDINAL);

  PROCEDURE PAUSE(n: CARDINAL);

  PROCEDURE INITSIGNAL(VAR s: SIGNAL);
    (* Initialisation of a SIGNAL *)

END PROCESSSCHEDULER.
```

Az ütemező (process-scheduler) definiciós része exportálja a SIGNAL tipust (rejtett tipus, kifejtése az implementációban található), és a rajta értelmezett müveleteket. Ezek közül a SEND, WAIT, AWAITED és a DOIO müveletekről már volt szó. A SENDDOWN müvelet a készülék vezérlő processzek különleges SEND-jét valósitja meg, nem hajt végre tényleges processz-átkapcsolást, csak futáskésznek jelöli meg a megadott SIGNAL-ra várakozó első processzt. A STARTPROCESS eljárás egy processz létrehozására és azonnali elinditására szolgál, az INITSIGNAL kezdeti értékkel lát el egy SIGNAL tipusu változót (ez kötelezően az első utasitás minden SIGNAL-ra!). A PAUSE nevü eljárás lehetővé teszi egy processz számára, hogy megadott időegységnyit várakozzék.

```
IMPLEMENTATION MODULE PROCESSSCHEDULER [6];
  (*$T- NW, CJ, SEK, HHN, LB  30-Jan-81 *)
  (* additional procedure INITSIGNAL CJ dec-79 *)
  (* elimination of import of storage handler CJ dec-79 *)
  FROM SYSTEM IMPORT WORD, PROCESS, ADDRESS,
      NEWPROCESS, TRANSFER, IOTRANSFER, LISTEN, SYSRESET;

  TYPE SIGNAL = POINTER TO ProcessDescriptor;
    ProcessDescriptor =
      RECORD pp: PROCESS;
```

```
      postponment: CARDINAL;
      (* postponment=0 means that the process is ready
         postponment>0 means that the process is waiting
                       for a signal.
         postponment>1 is used for PAUSE only *)
      next: SIGNAL; (* ties the descriptors in the ring *)
      queue: SIGNAL; (* ties the descriptors waiting for
                        the same signal *)
      interrupted: SIGNAL; (* pointer to descriptor of
                              interrupted process *)
    END ;

VAR cp: SIGNAL; (* current process *)
  tick: SIGNAL;
    (* head of the queue of the PAUSing processes*)
  intbyclock, clk: PROCESS;
  WSP: ARRAY [0..100] OF WORD;
    (* work space for clock process *)
  LCS [177546B]: CARDINAL;(* line clock status register *)
  StorageForCP: ProcessDescriptor;
  IDLESIGNAL: SIGNAL;
  IDLEWSP: ARRAY [0..100] OF WORD;

PROCEDURE STARTPROCESS(P: PROC; A: ADDRESS; n: CARDINAL);
  (* start P with workspace A of length n  *)
  VAR t: SIGNAL;
BEGIN t := cp; (*NEW(cp)*) cp := A;
  WITH cp^ DO
    next := t^.next; postponment := 0;
    interrupted := NIL;
    t^.next := cp;
    NEWPROCESS(P,A+TSIZE(ProcessDescriptor),
               n-TSIZE(ProcessDescriptor),pp);
    TRANSFER(t^.pp,pp);
  END;
END STARTPROCESS;

PROCEDURE AWAITED(S: SIGNAL): BOOLEAN;
BEGIN RETURN S # NIL
END AWAITED;

PROCEDURE SEND(VAR s: SIGNAL);
  (* resume first process waiting for s *)
  VAR t: SIGNAL;
BEGIN
  IF s # NIL THEN
    t := cp; cp := s;
    cp^.postponment := 0; s := cp^.queue;
    TRANSFER(t^.pp, cp^.pp)
  END
END SEND;

PROCEDURE SENDDOWN(VAR s: SIGNAL);
```

```
  (* mark first process waiting for s as ready *)
BEGIN
  IF s # NIL THEN
    s^.postponment := 0; s := s^.queue
  END
END SENDDOWN;

PROCEDURE nextready;
  (* activate next ready process *)
VAR this, strt: SIGNAL;
BEGIN this := cp;
  WITH this^ DO
    IF interrupted # NIL THEN
      cp := interrupted; interrupted := NIL;
    ELSE
      LOOP strt := cp;
        REPEAT cp := cp^.next
        UNTIL (cp^.postponment=0) OR (cp=strt);
        IF cp^.postponment = 0 THEN EXIT ELSE LISTEN END;
      END;
    END;
    TRANSFER(pp,cp^.pp);
  END;
END nextready;

PROCEDURE WAIT(VAR s: SIGNAL);
  VAR this, next: SIGNAL;
BEGIN (* insert current process at end of queue s *)
  IF s = NIL THEN s := cp
  ELSE
    this := s;
    LOOP next := this^.queue;
      IF next = NIL THEN EXIT END ;
      this := next
    END ;
    this^.queue := cp
  END ;
  cp^.postponment := 1;  cp^.queue := NIL;
  nextready;
END WAIT;

PROCEDURE DOIO(va: CARDINAL);
  VAR this, strt: SIGNAL; p: PROCESS;
BEGIN cp^.postponment := 1;
  this := cp;
  WITH this^ DO
    IF interrupted # NIL THEN
       cp := interrupted; interrupted := NIL;
    ELSE
      LOOP strt := cp;
        REPEAT cp := cp^.next
        UNTIL (cp^.postponment=0) OR (cp=strt);
        IF cp^.postponment = 0 THEN EXIT
```

```
        ELSE SENDDOWN(IDLESIGNAL) END;
      END;
    END;
    p := cp^.pp; IOTRANSFER(pp, p, va); cp^.pp := p;
    interrupted := cp; cp := this;
  END;
END DOIO;

PROCEDURE PAUSE(n: CARDINAL);
  VAR this: SIGNAL;
BEGIN
  IF n > 0 THEN
    cp^.queue := tick; tick := cp;
  END;
  cp^.postponment := n;
  nextready;
END PAUSE;

PROCEDURE INITSIGNAL(VAR s: SIGNAL);
  (* Initialisation of a SIGNAL *)
BEGIN
  s := NIL;
END INITSIGNAL;

PROCEDURE IDLE;
BEGIN INITSIGNAL(IDLESIGNAL);
  LOOP WAIT(IDLESIGNAL); LISTEN; END; (*LOOP*)
END IDLE;

PROCEDURE Clock;
  (* this procedure acts as a clock,
     ticking 50 times per sec *)
  VAR this,last: SIGNAL;
BEGIN LCS := 100B;
  LOOP
    intbyclock:= cp^.pp;
    IOTRANSFER(clk, intbyclock, 100B);
    cp^.pp:= intbyclock;
    this := tick; last := NIL;
    WHILE this # NIL DO
      WITH this^ DO
        DEC(postponment);
        IF postponment = 0 THEN
          IF last = NIL THEN tick := queue
          ELSE last^.queue := queue
          END;
        ELSE last := this;
        END;
        this := queue;
      END;
    END;
  END
END Clock;
```

```
  PROCEDURE INITSCHEDULER;
  BEGIN SYSRESET;
    (*NEW(cp)*) cp := ADDRESS(ADR(StorageForCP));
    WITH cp^ DO
      postponment := 0; next := cp;
      interrupted := NIL;
    END;
    tick := NIL;
    NEWPROCESS(Clock, ADR(WSP), SIZE(WSP), clk);
    TRANSFER(cp^.pp,clk);
  END INITSCHEDULER;

BEGIN INITSCHEDULER;
  STARTPROCESS(IDLE,ADR(IDLEWSP),SIZE(IDLEWSP))
END PROCESSSCHEDULER.
```

Az implementációs modul első sorában látható a processzor-prioritás megadása ([6]). Ez gépfüggő tulajdonság, megadása azt jelenti, hogy a modul összes eljárása ezen a processzor-prioritási szinten fut. Jelen esetben ez a legmagasabb engedélyezett szint, vagyis az ütemező és az óravezérlő szintje fölött már nem futhat processz. A közönséges (nem input/outputtal kapcsolatos) modulok prioritása általában 0. Az implementációs modulban látható, hogy a SIGNAL tipus egy processz leiróra (ProcessDescriptor) mutató POINTER. A processz leiró első eleme(pp) a processz adatterületére (stack-jére) mutat. A postponment nevü elem a processz futási állapotát irja le, értéke 0, ha a processz futáskész, 0-nál nagyobb, ha vár. Értéke 1-nél csak akkor lehet nagyobb, ha a processz az órasorban áll (PAUSE révén), postponment értéke ilyenkor a kért időegység száma. A next nevü, SIGNAL tipusu mező arra szolgál, hogy ezen keresztül a processzek egy gyürüre füződjenek fel (ld. STARTPROCESS). A processzek gyürüje már a MODULA implementációban is megvolt [Wir77c], a különbség csak annyi, hogy ott a készülék

vezérlő processzek nem füződtek fel a gyürüre. A queue nevü mező arra a célra szolgál, hogy egy processz fölfüződhessen egy tetszőleges SIGNAL-ra. Az INTERRUPTED nevü SIGNAL azt a célt szolgálja, hogy előnyben lehessen részesiteni a megszakitott processzeket. A várakozó jellegü müveletek (WAIT és DOIO) először mindig azt nézik meg, hogy van-e megszakitott processz, és csak ha nincs, akkor veszik a gyürüről a következő futáskész processzt (ld. nextready). A PROCESSSCHEDULER modul cp nevü változója az éppen futó processzre (current process) mutat. A modul további változói részint az órakezelést segitik, részint az ugynevezett IDLE-processzt támogatják. Az IDLE-processz semmi mást nem csinál, mint hogy vár egy jelre (IDLESIGNAL), és ha azt aktivizáljuk (ld. DOIO), akkor a LISTEN nevü SYSTEM-eljárás segitségével a processzor prioritását leszállitja, vagyis lehetővé teszi, hogy a megszakitások érvényre jussanak. Abban a reményben, hogy az ütemező eljárásainak müködése a programszövegből megállapitható, rátérünk egy következő példára, amely már azt mutatja, hogy hogyan lehet a most ismertetett ütemezőt használni, mégpedig egy tty (teletype) berendezés meghajtó moduljában. A modul lehetővé teszi, hogy egyszerre több azonos tipusu berendezés is müködjön, egymástól függetlenül.

```
DEFINITION MODULE TTYS;

EXPORT QUALIFIED CHOUT,CHIN,TELETYPES;

TYPE TELETYPES = (TT0,TT1,TT2);

PROCEDURE CHOUT(CH: CHAR; TTY: TELETYPES);
(*PRINTS A CHARACTER ON TTY*)

PROCEDURE CHIN (VAR CH: CHAR; TTY: TELETYPES);
(*READS A CHARACTER FROM TTY. CUTS THE EIGTH BIT*)

END.
```

A definiciós modul exportálja a TELETYPES nevü felsoroló tipust, amelynek lehetséges értékei a modul által fizikailag kezelhető berendezéseket adja meg. A CHOUT eljárás egy karakter kiirására, a CHIN pedig beolvasására szolgál

```
IMPLEMENTATION MODULE TTYS ;

IMPORT SYSTEM;
IMPORT PROCESSSCHEDULER;

TYPE EXISTENCE = SET OF TELETYPES;

VAR EXISTING: EXISTENCE;

MODULE TYPEWRITE [4];

IMPORT EXISTING,EXISTENCE,TELETYPES;
FROM SYSTEM IMPORT WORD;
FROM PROCESSSCHEDULER IMPORT SIGNAL,WAIT,
  SENDDOWN,INITSIGNAL,SEND,STARTPROCESS,PAUSE,DOIO;
EXPORT CHOUT;

CONST BUFL=32;
VAR T: TELETYPES; (*TELETYPE AND PROCESS IDENTIFIER*)
    NF: ARRAY TELETYPES OF INTEGER;
    INX,OUTX : ARRAY TELETYPES OF [1..BUFL];
    BUF : ARRAY TELETYPES,[1..BUFL] OF CHAR;
    WSP: ARRAY TELETYPES,[0..100] OF WORD;
    NONFULL,NONEMPTY: ARRAY TELETYPES OF SIGNAL;
```

```
PROCEDURE CHOUT(CH: CHAR; TTY: TELETYPES);
(*PRINTS OUT A CHARACTER ON TTY*)
BEGIN
   IF NOT (TTY IN EXISTING) THEN RETURN END; (*IF*)
   IF NF[TTY] = BUFL THEN WAIT(NONFULL[TTY]) END; (*IF*)
   BUF[TTY,INX[TTY]]:= CH;
   INX[TTY]:= INX[TTY] MOD BUFL + 1;
   INC(NF[TTY]);
   SEND(NONEMPTY[TTY]);
END CHOUT;

PROCEDURE DRIVER;
VAR TTY: TELETYPES;
    TWS0[177564B]: CARDINAL;
    TWS1[175614B]: CARDINAL;
    TWS2[175624B]: CARDINAL;
    TWB0[177566B]: CHAR;
    TWB1[175616B]: CHAR;
    TWB2[175626B]: CHAR;
BEGIN TTY:= T;
   LOOP
     IF NF[TTY] = 0  THEN WAIT(NONEMPTY[TTY])  END; (*IF*)
      CASE TTY OF
    TT0: TWB0:=BUF[TTY,OUTX[TTY]];
         TWS0:= 100B; DOIO(64B); TWS0:= 0;
   |TT1: TWB1:=BUF[TTY,OUTX[TTY]];
         TWS1:= 100B; DOIO(304B); TWS1:= 0;
   |TT2: TWB2:=BUF[TTY,OUTX[TTY]];
         TWS2:= 100B; DOIO(314B); TWS2:= 0;
      END; (*CASE*)
     OUTX[TTY]:= OUTX[TTY] MOD BUFL + 1;
     DEC(NF[TTY]);
     SENDDOWN(NONFULL[TTY]);
   END (*LOOP*)
END DRIVER;

BEGIN
  EXISTING:= EXISTENCE{TT0,TT2};
  FOR T:= TT0 TO TT2 DO
   IF T IN EXISTING THEN
    INX[T]:= 1; OUTX[T]:=1; NF[T]:=0;
    INITSIGNAL(NONFULL[T]); INITSIGNAL(NONEMPTY[T]);
    STARTPROCESS(DRIVER,ADR(WSP[T]),SIZE(WSP[T]));
   END; (*IF*)
  END; (*FOR*)
END TYPEWRITE;

MODULE KEYBOARD [4];

IMPORT EXISTING,EXISTENCE,TELETYPES;
FROM PROCESSSCHEDULER IMPORT SIGNAL,WAIT,
  SENDDOWN,INITSIGNAL,SEND,STARTPROCESS,PAUSE,DOIO;
EXPORT CHIN;
```

```
CONST BUFL=32;
VAR T: TELETYPES; (*TELETYPE AND PROCESS IDENTIFIER*)
    NF: ARRAY TELETYPES OF INTEGER;
    INX,OUTX : ARRAY TELETYPES OF [1..BUFL];
    BUF : ARRAY TELETYPES,[1..BUFL] OF CHAR;
    WSP: ARRAY TELETYPES,[0..100] OF WORD;
    NONFULL,NONEMPTY: ARRAY TELETYPES OF SIGNAL;

PROCEDURE CHIN(VAR CH: CHAR; TTY: TELETYPES);
(*READS A CHARACTER FROM THE KEYBOARD*)
BEGIN
   IF NOT (TTY IN EXISTING) THEN CH:= 0C; RETURN END; (*IF*)
   IF NF[TTY] = 0 THEN WAIT(NONEMPTY[TTY]) END; (*IF*)
   CH:= BUF[TTY,OUTX[TTY]];
   OUTX[TTY]:= OUTX[TTY] MOD BUFL + 1;
   IF INTEGER(CH) >= 200B
   THEN CH:= CHAR(INTEGER(CH)-200B)
   END; (*IF*)
   DEC(NF[TTY]);
   SEND(NONFULL[TTY]);
END CHIN;

PROCEDURE DRIVER;
VAR TTY: TELETYPES;
    KBS0[177560B]: CARDINAL;
    KBS1[175610B]: CARDINAL;
    KBS2[175620B]: CARDINAL;
    KBB0[177562B]: CHAR;
    KBB1[175612B]: CHAR;
    KBB2[175622B]: CHAR;
BEGIN TTY:= T;
   LOOP
     IF NF[TTY] = BUFL  THEN WAIT(NONFULL[TTY])  END; (*IF*)
      CASE TTY OF
    TT0:
          KBS0:= 100B; DOIO(60B); KBS0:= 0;
          BUF[TTY,INX[TTY]]:= KBB0;
   |TT1:
          KBS1:= 100B; DOIO(300B); KBS1:= 0;
          BUF[TTY,INX[TTY]]:= KBB1;
   |TT2:
          KBS2:= 100B; DOIO(310B); KBS2:= 0;
          BUF[TTY,INX[TTY]]:= KBB2;
      END; (*CASE*)
     INX[TTY]:= INX[TTY] MOD BUFL + 1;
     INC(NF[TTY]);
     SENDDOWN(NONEMPTY[TTY]);
   END (*LOOP*)
END DRIVER;

BEGIN
  EXISTING:= EXISTENCE(TT0,TT2);
```

```
  FOR T:= TTO TO TT2 DO
   IF T IN EXISTING THEN
    INX[T]:= 1; OUTX[T]:=1; NF[T]:=0;
    INITSIGNAL(NONFULL[T]); INITSIGNAL(NONEMPTY[T]);
    STARTPROCESS(DRIVER,ADR(WSP[T]),SIZE(WSP[T]));
   END; (*IF*)
  END; (*FOR*)
END KEYBOARD;

END TTYS.
```

Az implementációs modul az EXISTING nevü változót arra használja, hogy kijelölje a ténylegesen létező tty-tipusu készülékek halmazát, a TYPEWRITE nevüt a karakterek kirására és KEYBOARD-ot a beolvasásra. A modulok és a bennük elhelyezkedő készülék meghajtók (DRIVER processzek) ezen a szinten teljesen függetlenek, teljes duplex müködést engedélyeznek. A két modul müködése teljesen hasonló, mindkettő a már jól ismert fogyasztó/termelő elven dolgozó véges körpuffert valósit meg. A TYPEWRITE modulban a DRIVER a fogyasztó és a CHOUT eljárást kivülről hivó processz a termelő, a KEYBOARD esetében a DRIVER a termelő és a CHINt hivó processz a fogyasztó. A DRIVER nevü eljárásokat a modulok kezdeti utasitásai inditják el, STARTPROCESS segitségével (feltéve, hogy az adott tty-berendezés szerepel az EXISTING halmazban). A fenti példa fényt vet arra a sajnálatos körülményre is, hogy a modul sokszorozás hiánya miatt, a több, azonos tipusu berendezés kezelésére irt modul jóval bonyolultabb, mintha csak egyetlen berendezéskezelésére irunk modult; az összes modul-változónak eggyel több dimenziója van (a TELETYPES indexszel). A helyzetet itt még külön sulyosbitja, hogy a készülék regisztereket kijelölő különleges cimmegadások (például TWSO[177564B]) nem lehetnek változók, és ezért a DRIVER-ekben az egyes konkrét készülékekre való hivatkozásnál még az indexelés sem segit, hanem CASE utasitást kell alkalmazni. Ez a példa igy

különös módon egyszerre mutatja be a MODULA-2 egyik legnagyobb előnyét, hogy magasszintü nyelven, világos megfogalmazásban lehet input/output kezelő modulokat irni, és talán egyetlen komoly hátrányát, a modul sokszorozás hiányát.

3.3 AZ ADA

Az ADA nyelv [Ada80] sokéves nemzetközi munka eredményeképpen jött létre, azzal a céllal, hogy rendszerprogramozói és általános felhasználói jellegü programok egyaránt megfogalmazhatók legyenek benne. Az ADA nyelv tükrözi a software-technológia aktuális helyzetét, figyelembe veszi a strukturált programozás elméletének eredményeit is. Bizonyos kérdésekben - ugy tünik - az ADA tervezői meghajoltak olyan konvencionális szempontok előtt, amelyek nem váltak a nyelv előnyére. Az ADA összességében felülmulni látszik valamennyi eddigi nyelvet, és miután hathatós adminisztrativ támogatásra is számithat, bizonyára hamarosan világszerte elterjed.

3.3.1 MODULARITÁS

Az ADA a modularitást és információ elrejtést messzemenően támogatja. A modularitás egysége az ugynevezett package (csomag). A package általában két részre oszlik, egy specifikációs részre és egy törzsre. Ez lényegében megfelel a MODULA-2 vagy a MESA definiciós/implementációs moduljának. A specifikációs rész fő célja (a definiciós modulhoz hasonlóan), hogy a package megadja a környezet

felé láthatóvá tett objektumait, a környezet felé mutatott interface-ét. A specifikációs rész tartalmazhat olyan elemeket is (PRIVATE), amelyek csak belső használatra valók. Lehetőség van arra, hogy egy tipusnak csak a nevét tegyük láthatóvá, strukturáját azonban elrejtsük. Ezen belül is szabályozható még, hogy a rejtett tipuson az értékadást és egyenlőségre való összehasonlitást értelmezettnek tekintsük-e, vagy kizárólag a package által definiált müveleteket engedjük meg. A package törzse (az implementációs modulhoz hasonlóan) a specifikációs részben megadott elemek kifejtését tartalmazza. Tartalmazhat saját deklarációkat is, ezek kivülről hozzáférhetetlenek. Az ADA lehetővé teszi, hogy egy package-ből több példányt is létrehozzunk (generic), amelyek mind külön adattérrel, de közös programkóddal rendelkeznek. Ez a lehetőség hasonló ahhoz, ahogy a Konkurrens Pascal vagy a PORTAL megengedi osztályok, illetve modulok tipusként való definiálását. Szemléltetésül tekintsük egy veremtár (stack) megvalósitását ADA-ban:

```
GENERIC
  size: natural;
  TYPE elem IS PRIVATE;
PACKAGE stack IS
PROCEDURE push(e: IN elem);
PROCEDURE pop(e: OUT elem);
overflow,underflow: EXCEPTION;
END stack;

PACKAGE BODY stack IS

  space: ARRAY (1..size) OF elem;
  index: integer RANGE 0..size:=0;
```

```
PROCEDURE push(e: IN elem) IS
BEGIN
  IF index = size THEN
    RAISE overflow;
  END IF;
  index:= index+1;
  space(index):= e;
END push;

PROCEDURE pop(e: OUT elem) IS
BEGIN
  IF index = 0 THEN
    RAISE underflow;
  END IF;
  e:= space(index);
  index:= index-1;
END pop;

END stack;
```

A veremtár tényleges példányait létrehozó utasitások például:

```
PACKAGE stack_int IS NEW stack(size->200,elem->integer);
PACKAGE stack_bool IS NEW stack(100,Boolean);
```

Egy maximum 200, integer tipusu, és egy maximum 100, Boolean tipusu elem befogadására képes veremtárat definiáltunk. Az egyes müvelteket

```
stack_int.push(625);
stack_bool.pop(b);
```

alaku utasitásokkal lehet igénybevenni.

3.3.2 PÁRHUZAMOSSÁG

Az ADA nyelv igen komolyan és koncepciózusan támogatja a párhuzamosságot. A párhuzamosság alapegysége a task. A task szintaktikusan nagyon hasonlit a package-re, müködése pedig a CSP [Hoa78] és a DP [BrH78] processzeivel mutat erős rokonságot. A task, a package-hez hasonlóan két részre oszlik, egy specifikációs részre és egy törzsre. A specifikációs rész itt is elsősorban a látható objektumok megadására szolgál, a törzs pedig azok realizációjára. A task a specifikációs részben csak ugynevezett bemeneteket (ENTRY) tehet láthatóvá, másfajta objektumokat nem. Ez érdekes változás az ADA első definiciójához képest [Ich79a], amely még megengedte például közönséges eljárások láthatóvá tételét is. Ez bizonyos esetekben káros következményekkel járt, amint ezt [WeL81]-ben a szerzők kimutatták. A taskok közötti kommunikáció fő eszköze a bementek és az ACCEPT utasitások között létrejövő "randevu". A randevu igen közel áll a CSP összeillő input/output párjaihoz. A t1,t2 task között akkor jön létre randevu, ha mondjuk t1 meghivta t2 valamelyik bemenetét és t2 ugyanerre a bemenetre kiadott egy ACCEPT utasitást. Ha egy task kiad egy hivást egy másik task bemenetére, és az még nem adott ki arra ACCEPT-et, illetve ha egy task valamelyik saját bemenetére kiad egy ACCEPT-et (ACCEPT-et csak saját bemenetre lehet kiadni), de arra kivülről még nem jött hivás, akkor a task várakozó állapotba kerül. Innen akkor lép ki, amikor a megfelelő utasitás-párt is kiadták. Ekkor létrejön a randevu és a hivó felfüggesztett állapotban marad addig, amig a másik task az ACCEPT utasitást végre nam hajtja. Ezután a két task ismét párhuzamosan futhat tovább. A nem determinisztikus tulajdonságok ábrázolására az ADA a SELECT utasitást vezette be, amely erős rokonságot mutat a CSP és a DP őrzött parancsaival. Az

őröket a WHEN kulcsszó után lehet megadni, az alternatívákat OR választja el, illetve az utolsó alternatívát ELSE vezetheti be. Az őr után állhat ACCEPT, DELAY vagy TERMINATE.Maga az őr tetszőleges logikai kifejezést tartalmazhat. Ha egy őr kiértékelése sikeres és után ACCEPT áll, akkor létrejön egy randevu (ha ilyenmódon több randevu is létrejöhetne, akkor ezek közül egy tetszőleges jön létre). Ha nincs sikeres őrrel rendelkező ACCEPT, akkor a sikeres őr utáni DELAY vagy TERMINATE kerülhet végrehajtásra (utóbbi csak akkor, ha a task befejeződése egyéb feltételei biztositva vannak), vagy az ELSE utáni utasitások. Szemléltetésül tekintsük ismét a véges körpuffer problémáját:

```
TASK buffer IS
  ENTRY read(c: OUT character);
  ENTRY write(c: IN character);
END;

TASK BODY buffer IS
  pool_size: CONSTANT integer:=100;
  pool       : ARRAY(1..pool_size) OF character;
  count      : integer RANGE 0..pool_size:=0;
  in,out     : integer RANGE 1..pool_size:=1;
BEGIN
  LOOP
    SELECT
      WHEN count < pool_size ->
        ACCEPT write(c: IN character) DO
          pool(in):= c;
        END;
        in:= in MOD pool_size + 1;
        count:= count+1;
    OR WHEN count > 0 ->
         ACCEPT read(c: OUT character) DO
           c:= pool(out);
```

```
        END;
        out:= out MOD pool_size + 1;
        count:= count-1;
    OR TERMINATE
     END SELECT;
  END LOOP;
END buffer;
```

A fogyasztó és termelő taskok

buffer.write(ch)

buffer.read(ch)

alaku utasitásokkal hivhatják a buffer task szolgáltatásait. Figyelemreméltó, hogy a hivó taskoknak csak azt kell kivárniuk, amig a buffer a pool-on végzett müveletet végrehajtja, az ACCEPT záró END-je után a két task már ismét párhuzamosan futhat.

3.3.3 AZ ADA PÁRHUZAMOS TULAJDONSÁGAINAK ELEMZÉSE

Az ADA-ról kiváló elemzés található [WeL80]-ban, ahol a szerzők az ADA tulajdonságait összevetik a CSP-vel és a DP-vel, amelyek, mint az már eddig is látható volt, erősen hatottak az ADA párhuzamos tulajdonságaira. A cikk alapjául az ADA első változata szolgál [Ich79a,Ich79b], azóta néhány kedvező változás következett be a párhuzamos tulajdonságok vonatkozásában (például eljárást nem lehet láthatóvá tenni taskból). A CSP és a DP processz kommunikációja közötti legszembeötlőbb különbség az, hogy a CSP-ben a megnevezés szimmetrikus (mindkét processz megnevezi partnerét), a DP-ben pedig asszimetrikus. Ez utóbbi megoldás nyilván előnyös, ha olyan könyvtárakat akarunk létrehozni, amelyek nem ismerik előre felhasználóikat. Hátrányos azonban akkor, ha a felhasználók valamiféle azonositása szükséges. Tekintsünk erre egy igen egyszerü példát, egy olyan erőforrás ütemezőt egy bináris(szemafort), amely mondjuk száz felhasz-

náló processzt (taskot) tud kiszolgálni. A megoldás CSP-ben:

```
resource::
 *[(i:1..n)user(i)?request() → user(i)?release()]
```

Ez a megoldás nemcsak azt biztositja, hogy amig valamelyik felhasználó processz éppen használja az erőforrást (kiadta a resource!request() utasitást), addig más processz nem férhet hozzá, de azt, is, hogy az erőforrás felszabaditása (resource!release()) is csak ettől a processztől jöhet. Ha most ettől az utóbbi követelménytől eltekintünk, akkor egy igen egyszerü ADA megoldáshoz juthatunk [Ich79a]:

```
TASK resource IS
  ENTRY request;
  ENTRY release;
END;
TASK BODY resource IS
BEGIN
  LOOP
    ACCEPT request;
    ACCEPT release;
  END LOOP;
END resource;
```

Ez a megoldás is biztositja, hogy egyszerre csak egy processz használhassa az erőforrást, vagyis a hivókra rákényszeriti a resource.request, resource.release hivási sorrendet. Addig nem fogad el release-t, amig request nem volt, és addig nem fogad el második request-et, amig release nem volt. Azt viszont nem ellenőrzi, hogy ki adta ki a release-t, tehát hibás felszabaditást is megenged. Ha ezt is ellenőrizni kivánjuk, akkor bevezethetünk egy azonositó ellenőrzést:

```
TYPE user_id IS NEW integer RANGE 1..n;
TASK resource IS
  ENTRY request(id: IN user_id);  --ENTRY paraméterrel
  ENTRY release(user_id'FIRST..user_id'LAST); --ENTRY család
END;

TASK BODY resource IS
  user: user_id;
BEGIN
  LOOP
    ACCEPT request(id: IN user_id) DO
      user:= id;
    END;
    ACCEPT release(user);
  END LOOP;
END resource;
```

Ez a megoldás már kifogástalanul müködik abban az esetben, ha a felhasználó processzek helyesen azonositják önmagukat, és nem orozzák el egy másik task azonosságát (amit ADA-ban megtehetnek, CSP-ben nem). Ha ezt az utóbbi hibalehetőséget is ki akarjuk zárni, akkor olyan azonositási mechanizmust kell alkalmaznunk, amellyel nem lehet kivülről visszaélni. Ezt rejtett tipus alkalmazásával könnyen megtehetjük. Ebben az esetben egy védőkulcsot alkalmazhatunk, amelynek tipusát és a rajta végezhető müveleteket a hivók elől teljesen elrejtjük. A hivó request és release hivásakor köteles egy ilyen tipusu paramétert átadni a resource tasknak. A resource task a request-en bekövetkezett randevu során a védőkulcs tipusu paramétert beállitja valamilyen értékre, és ezt visszaadja a hivónak. A hivó ezt az értéket megváltoztatni nem tudja, mert a tipus teljesen rejtett (PRIVATE LIMITED). Első közelitésben egy közönséges logikai változó is megfelelne kulcsnak, amelyet request első hivójánál TRUE-ra állitunk (kezdeti értéke FALSE), és ezután csak TRUE értékü kulccsal lehet release-t hivni, ilyet viszont a hivó task előállitani nem tud. Ez a megoldás sajnos még mindig nem hibátlan. Ennek az az oka, hogy az ADA elő-

irása szerint egy ENTRY hivásakor az aktuális paraméterek kiértékelése még azelőtt megtörténik, hogy a hivó megkezdi várakozását az ACCEPT-re, illetve létrejön a randevu. Vagyis a paraméterek értékelése még a kölcsönös kizáráson kivül játszódik le. Megengedett viszont, hogy taskok közös globális (shared) adatokkal rendelkezzenek. Ez azt jelenti, hogy ha a release paraméterének átadása hivatkozás szerinti, akkor a leirt megoldás helyes, de ha érték szerinti akkor nem. Az ADA definitive hibásnak tekinti az olyan programokat, amelyek függnek a paraméterátadás módjától; az ismertetett megoldási séma tehát definitive hibás - a kérdés csak az, hogy van-e olyan forditó, amelyik egy ilyen hibát kijelez. Ez az utóbbi nehézség nem merült volna fel, ha a bemenetek paramétereinek értékelése már a randevuban történne, vagy ha legalább kizárt lenne, hogy a taskok közös, globális adatokkal rendelkezzenek. Ez utóbbit azzal az indokkal vezették be a nyelvbe, hogy "ennek kizárása bizonyos kritikus esetekben nem lenne bölcs" [Ich79b]. Teljes joggal válaszolják erre [WeL81] szerzői, hogy más esetekben a megengedésük "nem bölcs". Az azonositást tehát csak ugy valósithatjuk meg, ha a resource task minden egyes request híváskor egyedi védőkulcsot generál (ehhez elvileg végtelen tartományra lenne szükség). Még ekkor is marad egy szépséghibája a megoldásnak, az, hogy egy hibás release nemcsak a hibásan hivó taskot, de magát a resource taskot is hibahelyzetbe (EXCEPTION) hozza.

[WeL81]-ben a szerzők a task azonositás ismertetett nehézségei után vizsgálat alá veszik a szinkronizációs és nem determinisztikus sajátságokat. A főbb következtetések a következők:

1. Az ADA párhuzamos tulajdonságai közelebb állnak a CSP-hez, mint a DP-hez. A task (processz) azonositásában a DP-hez közelebb álló asszimetrikus megoldást választották, de a DP-től eltérően, nem kötötték ezt össze nem kivánatos, másodlagos indeterminizmusok bevezetésével. Az asszimetrikus megoldásnak - nyilvánvaló előnyei mellett - még igy is vannak hátrányai. Ezeket felerősiti a bemenetekre definiált paramétereátadási szabály és a közös (shared) változók lehetősége.

2. A nem determinisztikus tulajdonságok lekezelésére az ADA az őrzött parancsoknak megfelelő konstrukciót használ, de a CSP-ben megengedett tisztán logikai (Boolean) tipusu őrök helyett egy ELSE esetet vezet be. Ez bizonyos esetekben ciklikus várakozást idézhet elő (busy waiting), amely azonban megfelelő körültekintéssel elkerülhető, és a keletkező program sem szükséges, hogy bonyolultabb legyen, mint CSP megfelelője.

3. A CSP nagyon elegáns eszközt ad arra, hogy egy processz bizonyos feltételtől függően a partner processzeknek csak egy részhalmazára várakozzon. Az ADA-ban ilyen feladat csak nagyon nehézkesen oldható meg.

Végül is elmondható, hogy amig a DP fő hátránya a CSP-vel szemben, hogy annál fokozottabb indeterminizmust enged meg, az ADA a CSP-hez képest valamelyest csökkentette az indeterminizmus mértékét, ami szintén némi hátránnyal jár.

3.4 MESA

A MESA nyelv [Mit79] egy igen sokrétü software és hardware fejlesztés részeként jött létre a Xerox cég Palo Alto-i kutató részlegében. Az ismertetett nyelvek közül talán a legtöbbet használt és az egyik legrégebbi. Mivel a Xerox cég alapvetően saját használatra fejlesztette ki, széles körben nem lett ismertté, annál inkább hatott a nyelv- és fordító- tervezők körében.

A MESA-nak is elsősorban moduláris és parallel tulajdonságait vesszük szemügyre, de a nyelv minden részletében figyelmet érdemel.

3.4.1 MODULARITÁS

A feladat dekompoziciót a MESA messzemenően támogatja, annak egysége a modul. Alapvetően kétféle modult különböztethetünk meg; a definiciós modult, amely a külvilággal való interface-t irja le és az ugynevezett program modult, amely a tényleges viselkedést implementálja. Ez a koncepció erősen befolyásolta a MODULA-2 definiciós/implementációs modul fogalmát. A kétféle modul a MESA-ban is a különforditás elvileg tiszta és ellenőrzött(!) alapjául szolgál. A külön forditott modulokat egy szerkesztő müvelettel füzhetjük össze egyeteln programmá. Külön érdekes megemliteni, hogy a szerkesztés maga is a MESA egy erősen szükitett részhalmazában történik, és igy teljes jogu része a rendszernek, nem utólag, ad-hoc jelleggel készült. A modulok között export/import révén lehetséges kapcsolat,

az ebben részt nem vevő objektumok rejtve maradnak a modul környezete elől. A MESA explicit támogatást nyujt könyvtárak létrehozásához, ahol az egyes modulokat közhasználatuvá (PUBLIC) vagy csak egy adott felhasználói csoport közös használatára alkalmasnak (PRIVATE) lehet deklarálni. A modul sokszorozás és inicializálás dinamikusan, futás során történhet, teljesen analóg módon azzal, mint ahogy a PASCAL-ban vagy a MODULA-2-ben egy POINTER-rel azonositott RECORD tipusu változó uj értéket kap a NEW utasitással. A szemléltetés kedvéért tekintsünk egy igen egyszerü példát, egy irógép kezelő modul definicióit és egy modult, amelyik ennek szolgáltatásait felhasználva visszairja az irógépre (képernyőre) a begépelt karaktereket. Ha egy sor elején "." áll, akkor a másoló fejezze be müködését:

```
Iodefs: DEFINITIONS =
BEGIN
-- Interface definíciók
  ReadChar: PROCEDURE RETURNS[CHARACTER];
  ReadLine: PROCEDURE[input: STRING];  --input-ba olvas
  WriteChar: PROCEDURE[output: CHARACTER];
  WriteLine: PROCEDURE[output: STRING];

  IOPks: PROGRAM;
-- Nem interface definíciók
  cr: CHARACTER = 15C;  --kocsi vissza
END.
```

Az implementáció vázlatos felépítése:

```
DIRECTORY
    Iodefs: FROM "Iodefs"  --Iodefs file-nevére hivatkozik
IOPks: PROGRAM EXPORTS Iodefs =
BEGIN
TerminalState: {off,on,hung} <- off;  --kezdeti állapot off
ReadChar: PUBLIC PROCEDURE RETURNS[CHARACTER] = BEGIN...END;
ReadLine: PUBLIC PROCEDURE[input: STRING] = BEGIN...END;
WriteChar: PUBLIC PROCEDURE[output: CHARACTER] =
 BEGIN...END;
WriteLine: PUBLIC PROCEDURE[output: STRING] = BEGIN...END;
END.
```

A felhasználó modul ezekután a következőképpen nézhet ki:

```
DIRECTORY
     Iodefs: FORM "Iodefs";

Copier: PROGRAM IMPORTS Iodefs =
BEGIN OPEN Iodefs  --minősítés nélküli hivatkozást enged
input: STRING <- [256];  --256 hosszú string

DO
  ReadLine[input];
  IF input[0] = '. THEN EXIT;
  WriteLine[input];
ENDLOOP;
WriteChar[cr];
END.
```

3.4.2 PÁRHUZAMOSSÁG

A MESA a párhuzamosság támogatására számos eszközt ad. Érdekes, hogy támogatja a kvázi-parallelitást is, korutinok formájában. A korutint, (ld. a MODULA-2 leirásánál is) felfoghatjuk mint olyan processzt, amely partnereivel csak explicite megadott kérés formájában kommunikál. Amikor egy korutin visszaadja a vezérlést egy másik korutinnak, akkor nem szünik meg létezni (mint egy eljárás), de nem is fut tovább (mint egy processz). Amikor legközelebb ismét meghivják, akkor ott folytatja, ahol abbahagyta nem az elején, mint egy eljárás . A korutinok inditása nehéz feladat, az elsőnek inditott korutin hivatkozást tehet a másodikra, mielőtt az elindult volna. Ezt a nehézséget küszöböli ki a MESA korutin inditó mechanizmusa, amely lehetővé teszi az inditási tranziensek helyes kezelését. A korutinok kommunikációja ugynevezett PORT-okon keresztül történik. Példaképpen tekintsünk két modult, amelyek file másolást végeznek:

```
DIRECTORY
  FileDefs: FROM "filedefs" USING[NUL,FileHandle,
     FileAccess,OpenFile,ReadChar,EndofFile,ClosFile];

ReadFile: PROGRAM[name: STRING] IMPORTS FileDefs =
BEGIN OPEN FileDefs;
Out: PORT[ch: CHARACTER];
input: FileHandle;
input <- OpenFile[name: name,access: FileAccess[Read]];
STOP;
UNTIL EndofFile[input]
  DO
    Out[ReadChar[input]];  --PORT hivás(küld egy karaktert)
  ENDLOOP;
CloseFile[input];
Out[NUL];  --NUL küldésével jelzi a file végét
END.

DIRECTORY

  FileDefs: FROM "filedefs" USING[NUL,FileHandle,
     FileAccess,OpenFile,ReadChar,EndofFile,ClosFile];

WriteFile: PROGRAM[name: STRING] IMPORTS FileDefs =
BEGIN OPEN FileDefs;
In: PORT RETURNS[ch: CHARACTER];
char: CHARACTER;
output: FileHandle;
output <- OpenFile[name: name;access: FileAccess[New]];
STOP;
DO  -- amíg In egy NUL-t nem küld
  char <- In[];  --kap In-től egy karaktert
  IF char = NUL THEN EXIT;
  WriteChar[output,char];  --kiírja a file-ba
ENDLOOP;
ClosFile[output];
END.
```

A ReadFile program először megnyitja az input file-t (Read hozzáféréssel), majd STOP utasitást ad ki. Amikor ujrainditják, akkor egy ciklusba kerül, ahol addig olvas be és küld Out-nak karaktereket, amig az UNTIL-ban megadott feltétel (file vége) föl nem lép. Ekkor még küld egy NUL karaktert, és ha még továbbra is ujrainditják, akkor azonnal visszatér. WriteFile New hozzáféréssel nyitja meg az output file-t, és ezután ad ki STOP-ot. Ujrainditás után addig vesz el karaktereket In-től és küldi ki az output file-ra, amig csak NUL-t nem kap. A két program tehát elvileg képes korutinként dolgozni, most nézzük meg, hogyan kell ehhez elinditani. A teljes inditó modul helyett nézzük csak az inditó utasitásokat:

```
...
START reader[input];
START writer[output];
CONNECT writer.In TO reader.Out;
CONNECT reader.Out TO writer.In;
RESTART writer[!TrapDefs.PortFault => CONTINUE];
RESTART reader[!TRapDefs.PortFault => ERROR];
END.
```

A két START utasitás elinditja a két korutint, amelyek, mint láttuk egy inicializáló müvelet után STOP-pal megállnak. A két CONNECT utasitás ezután összerendeli a megfelelő PORT-okat (ez csak START után lehetséges), majd ujrainditja a korutinokat, amelyek a STOP utáni ponton folytatódnak. Ha az elsőnek inditott writer azt észleli, hogy partnere nem a vele kapcsolatos PORT-híváson áll (és indulás-kor ez lesz a helyzet), akkor hiba (trap) keletkezik, amelyet azonban CONTINUE megadásával egyszerüen ignorálunk.

A valódi párhuzamosság alapeszköze a processz. A processzek létrehozására és szinkronizálására több lehetőség is van. A legegyszerübb a FORK/JOIN mechanizmus. A FORK utasitással egy tetszőleges eljárást processzként indithatunk el. A JOIN utasitás egy randevu igényt jelent be, ami akkor teljesül, ha a processzkéntinditott eljárás befejezte müködését. A randevu során az eljárás eredményeit átadja inditójának, és egyszersmind megszünik processzként tovább létezni. A FORK/JOIN mechanizmust tehát arra lehet használni, hogy egy processz egyes részfeladatai végrehajtásához egy önmagával párhuzamosan futó al-processzt hozzon létre. Tekintsük meg ennek a mechanizmusnak a müködését egy egyszerü példán. Tegyük fel, hogy van egy eljárásunk, amely egy sort olvas be a ReadChar eljárás felhasználásával. A sort a CR (kocsi vissza) karakter zárja:

```
ReadLine: PROCEDURE[s:STRING] RETURNS[CARDINAL] =
  BEGIN
  c: CHARACTER;
  s.length <- 0;
  DO
    c <- ReadChar[];
    IF ControlCharacter[c] THEN DoAction[c]
    ELSE AppendChar[s,c];
    IF c = CR THEN RETURN[s.length];
  ENDLOOP;
END;
```

Ezt az eljárást közönségesen ugy hivhatjuk, ha n CARDINAL tipusu, és buffer STRING tipusu:

```
n ← ReadLine[buffer];
```

Ha a hivó ReadLine végrehajtását a hivóval párhuzamosan futó processzként akarja végrehajtani, akkor definiálnia kell egy PROCESS tipusu változót:

```
p: PROCESS[RETURNS CARDINAL];
```

Ezután a hivás formája:

```
p ← FORK ReadLine[buffer];
... ReadLine-nal parallel végezhető müveletek...
n ← JOIN p;
```

A FORK ponton ReadLine a hivótól független életre kell, JOIN-nal átadja eredményét és megszünik processzként létezni.

A FORK/JOIN müveletpár egy jellegzetes processz család kezelésére alkalmas, amelynek tagjai dinamikusan keletkeznek és szünnek meg. A processzek egy másik családjába tartoznak azok, amelyek keletkezésük után "örökké" élnek. Az ilyen processzek inditására nem áll rendelkezésre külön nyelvi eszköz, hanem egy rendszer eljárás (ehhez hasonló megoldást alkalmaztak a MODULA-2-ben).

A processzek egymás közti kommunikációjára a MESA nyelv a FORK/JOIN páron kivül lehetővé teszi monitorok és CONDITION tipusu változók használatát. A monitorok a jól ismert elv [Hoa73] alapján a kölcsönös kizárás, a rövidtávu ütemezés célját szolgálják, a CONDITION tipus és a rajta végezhető müveletek pedig a középtávu ütemezést. A monitorokon a kölcsönös kizárás az előre definiált

```
MONITORLOCK:TYPE = PRIVATE
 RECORD[locked:BOOLEAN,queue:QUEUE];
```

tipus segitségével kerül megvalósitásra. A PRIVATE megadás biztositja, hogy a MESA programozó sohasem érheti el egy MONITORLOCK tipusu változó mezőit. Módja van viszont arra, hogy definiáljon ilyen változót, és ezzel a kölcsönös kizárást bizonyos mértékig irányitsa. Ez akkor lehet előnyös, ha egy monitor tulságosan nagy méretüvé válhatna, és célszerübb több kisebb modulra szétvágni. A külön modulok eljárásaira egy programozó által definiált közös (az egyik modul deklarálja, a többi importálja) MONITORLOCK tipusu változó segitségével lehet a kölcsönös kizárást biztositani. A mo-

nitorok ENTRY eljárások révén nyujtanak szolgáltatást kifelé. A monitorok sokszorozására többféle lehetőség is van, hatékonysági szempontoktól függően. A CONDITION tipuson három müvelet értelmezett: WAIT, NOTIFY és BROADCAST. A WAIT a szokásnak megfelelően egy jelre való várakozást és egyszersmind a kölcsönös kizárás feloldását jelenti. A NOTIFY egy jel küldésére szolgál a jelre várakozó processzek közül az elsőnek. Ha egyáltalán nincs a jelre (CONDITION-ra) váró processz, akkor NOTIFY hatástalan. NOTIFY nem oldja fel a kölcsönös kizárást (mint a SEND a MODULA-ban), és nem biztositja, hogy a várakozó processz azonnal továbbinduljon. Ez utóbbi azt jelenti, hogy az a logikai feltétel, amelynek teljesülését NOTIFY jelzi, csak valószinüleg, de nem biztosan áll fenn a WAIT utasitást követő pillanatban. Ebből következik, hogy a WAIT utasitás teljesülése után a hozzátartozó logikai feltételt ujra le kell kérdezni, vagyis a WAIT-et egy IF jellegü utasitás helyett egy WHILE tipusu utasitásban kell mindig kiadni. (A Konkurrens Pascal vagy a MODULA biztositja, hogy a WAIT után a jelzett feltétel fennáll. A MESA NOTIFY ugy müködik, mintha a MODULA-2-ben ismertetett MODULA-processz-ütemezőt használva a SEND helyett mindenütt SENDDOWN-t alkalmaznánk.) Ez a megoldás nyilván kedvezőtlen abban az esetben, ha az ismételt feltétel vizsgálat nagyon munkaigényes. A BROADCAST utasitás a jelre váró összes processzt elinditja. Ki lehet mutatni, hogy BROADCAST használata mindig helyes, bár nem mindig a legjobb hatékonyságu. Ez nyilván nem lenne igaz, ha nem lenne amugy is kötelező a ciklikus feltételvizsgálat WAIT körül. Az a körülmény azonban, hogy BROADCAST igy helyességi szempontból ekvivalens NOTIFY-jal, komoly érv NOTIFY-nak a szokásostól eltérő megoldása mellett. A MESA monitor müködésének szemléltetésére nézzünk meg egy monitort, amely memória allokációt végez kötött méretü memória egységekből:

```
StorageAllocator: MONITOR =
  BEGIN
  StorageAvailable: CONDITION;
  FreeList: POINTER;

  Allocate: ENTRY PROCEDURE RETURNS[p:POINTER] =
    BEGIN
    WHILE FreeList = NIL DO
      WAIT StorageAvailable
    ENDLOOP;
    p <- FreeList; FreeList <- p^.next;
    END;

  Free: ENTRY PROCEDURE[p: POINTER] =
    BEGIN
    p.next <- FreeList; FreeList <- p;
    NOTIFY StorageAvailable
    END;
END.
```

A FreeList nevü POINTER a szabad memória egységek láncára mutat, ha értéke NIL, akkor nincs szabad memória; várni kell. Ezt a feladatot könnyen átfogalmazhatjuk ugy, hogy a monitor változó hosszuságu memóriadarabok felszabaditását, illetve lefoglalását engedje meg. Ebben az esetben csak azok a processzek kénytelenek várakozni, amelyek igénye a meglévő mennyiségnél nagyobb. Amikor valamelyik processz memóriát szabadit fel (Free), akkor az eljárás a memória visszavétele után BROADCAST-tal jelez NOTIFY helyett. Ekkor a WAIT-en várakozó valamennyi processz ujraindul, a ciklikus ellenőrzés révén ismét megkisérli lefoglalni a szükséges memóriát. Ha ez megint nem sikerül, akkor visszatér a WAIT-re:

```
StorageAllocate: MONITOR =
  BEGIN
  StorsaeAvailable: CONDITION;
  ...

  Allocate: ENTRY PROCEDURE[size:CARDINAL]RETURNS[POINTER] =
    BEGIN
    UNTIL <megfelelő memória áll rendelkezésre>DO
      WAIT StorageAvailable
    ENDLOOP;
    p <- <a megfelelő memóriadarab címe>;
  END;

  Free: ENTRY PTOCEDURE[p:POINTER;size:CARDINAL] =
    BEGIN
    ...<visszafűzi a szabad memóriát>...
    BROADCAST StorageAvailable
    END;
END.
```

3.5 PORTÁL

A PORTAL nyelv [Nag79] a Modula-val párhuzamosan készült egy svájci magáncég számára, ipari célokra. A PORTAL és a MODULA közötti különbség érdekesen mutatja, hogy a célkitüzés hogyan befolyásolja egy nyelv felépitését. A PORTAL ipari célokra készült, ennek egyik következménye, hogy a biztonság kétszeresen is fontos követelmény, egyrészt az alkalmazások várható "komolysága" miatt, másrészt mert programozási szempontból tapasztalatlan felhasználókra is számitani lehet. A MODULA nyelvek egyetemi felhasználók számára készültek, ahol az alkalmazások általában kevésbé "élesek", és magasszintü programozói tudásra lehet számitani. A biztonság növelésére vonatkozó törekvések a PORTAL szekvenciális tulajdonságaiban is megnyilvánulnak. (Például nem engedi meg a VARIANT RECORD-ok használatánál a TAG-mező elhagyását. Ezzel egyrészt védekezni tud futási időben hibás hozzárendelések ellen, másrészt mindig

teljesen szimbolikus post-mortem dumpot tud készitени.) A párhuzamos tulajdonságokra vonatkozóan a PORTAL a monitor koncepciónak MODULA-ban alkalmazott változatát valósitja meg. A PORTAL-ban igy a fordító ellenőrizni tudja, hogy egy SIGNAL tipusu változót csak egy monitorban szabad deklarálni. A MODULA modul fogalmának megalkotásánál érezhető, hogy a nyelv végső soron egy személyes számitógép [Wir81] nyelvének készült. Ebben kereshető annak az oka, hogy a modul sokszorozásra a nyelv semmilyen különleges támogatást nem ad. A PORTAL-ban a modulok és monitorok tipusként is megadhatók, és ezután tetszőleges számu változót deklarálhatunk az adott tipusból, teljesen hasonlóan, mint például a Konkurens Pascal-ban [BrH77]. Szemléltetésül vegyük ismét a véges körpuffer példáját:

```
TYPE circbuff = MONITOR(n: INTEGER; TYPE info);
  DEFINES get,put;

  TYPE bufftype = ARRAY 1..n OF info;
    buffind = INDEX OF bufftype;
  VAR inp,out: buffind; count: 0..n;
      buff: bufftype;
  SIGNAL nonfull,nonempty;

  PROCEDURE put(x: info);
    USES VAR buff,inp,count;
    USES nonfull,nonempty;
    CODE
      IF count = n THEN nonfull.WAIT END IF;
      buff[inp]:= x;
      INCR(inp);
      IF NOT INRANGE(inp) THEN inp:= 1 END IF;
      INCR(count);
      IF count = 1 THEN nonempty.SEND END IF;
```

```
  END put;

  PROCEDURE set(RESULT x: info);
    USES VAR out,count;
    USES buff,nonfull,nonempty;
    CODE
      IF count = 0 THEN nonempty.WAIT END IF;
      x:= buff[out];
      INCR(out);
      IF NOT INRANGE(out) THEN out:= 1 END IF;
      DECR(count);
      IF count = n-1 THEN nonfull.SEND END IF;
  END set;

  CODE
    count:= 0; inp:= 1; out:= 1;
END circbuffer;
```

Ezt a tipust felhasználhatjuk példányok definiálására, például:

```
MONITOR integerbuff: circbuf(n==100,info==integer);
  DEFINES get,put;
END integerbuf;
```

Ez a példány egy maximum 100, integer tipusu elem befogadására képes. Felhasználói az integerbuff.get(i), illetve integerbuff.put(625) alaku utasitásokkal hivhatják a szolgáltatásait. A

```
TYPE complex = RECORD re,im: REAL END RECORD;

MONITOR complexbuff: circbuff(n==50,info==complex);
  DEFINES get,put;
END complexbuff;
```

deklarációkkal egy maximum 50, complex tipusu elem befogadására képes körpuffert definiáltunk. A példából jól látszik a PORTAL nyelv még további jónéhány kellemes tulajdonsága, például, hogy tipust is lehet átadni paraméterként.

A PORTAL a multiprogramozás igényein tulmenően, célul tüzte ki, hogy valós idejü (real-time) alkalmazások programozására is kényelmesen használható legyen. A multiprogramozás egyik alapelve, hogy a processzek sebességére semmilyen kikötést nem tesz azon kivül, hogy az pozitiv. A real-time programozás kikötést tehet egy processz lefutásának abszolut idejére (adott időn belül le kell kezelni bizonyos jelzéseket), és processzek egymáshoz képesti relativ idejére is (bizonyos müveleteknek mindig elsőbbséget kell biztositan). A real-time programozás támogatására a PORTAL lehetővé teszi, hogy a processzeket prioritással lássuk el. Azt a követelményt, hogy egy adott programegység adott időn belül lefusson, a PORTAL nyelv közvetlenül nem támogatja (a forditót képessé lehet tenni arra, hogy egy adott programszakasz várakozáson kivüli futási idejét kiszámitsa), de támogatja azt, hogy a programozó kényelmesen készülhessen fel arra az esetre, ha valamilyen esemény adott időn belül nem következik be. (Ez a müvelet, egy time-out megadása, egyébként egyáltalán nemcsak a real-time programozásban fordulhat elő, és a legtöbb programnyelv eléggé "mostohán" bánik vele.)

Az input/output kezelésére a PORTAL a MODULA-hoz nagyon hasonló lehetőségeket biztosit, az eddigi többletekkel együtt persze.

Érdemes megemliteni, hogy a PORTAL definiálása és forditójának elkészitése során igen komoly erőfeszitéseket tettek egy magas foku hordozhatóság (protabilitás - innen ered a nyelv neve is) elérésére. A forditót lényegében két részre osztották, egy gépfüggetlen és egy gépfüggő menetre. A gépfüggő menetet minden ujabb célgépre ujra kell irni, a gépfüggetlen menetet természetesen nem. Ez igen érdekes közelitése a hordozhatóságnak, amely számos nehéz kérdést vet fel [Nag78].

3.6 EDISON

Az EDISON az egyik legujabban közzétett nyelv [BrH81a-c]. Az EDISON elsősorban real-time alkalmazások számára, és kifejezetten multi-mikroprocesszoros környezetre készült. Kiinduló feltételezés, hogy minden processz külön processzoron fut (hatékony implementáció esetén külön fizikai processzoron). Az EDISON leglényegesebb vonása, hogy extrém módon törekedett a leegyszerüsitésre, gyanitható módon azért is, mert elsősorban ipari programozóknak készült. Az EDISON nagymértékben támaszkodik korábbi nyelvekre, a PASCAL-ra [JeW74], a Konkurrens Pascal-ra [BrH77] és a MODULA-ra [Wir77a], de ezenkivül "felmelegit" néhány egészen régi fogalmat is, mint a feltételes kritikus régiót és a Dijkstra féle párhuzamos utasitást [Dij68a].

A modularitás tekintetében az EDISON nagyjából a MODULA modul fogalmát használja. Abban a vonatkozásban is, hogy az EDISON sem ad eszközt a modul sokszorozásra. Éppen csak a

szemléltetés kedvéért nézzük meg egy veremtár (stack) implementációját EDISON-ban:

```
MODULE
  CONST max = 10
  ARRAY stack[1:max] (int)
  VAR lifo: stack; size: int


*PROC push(x:int)
 BEGIN size:= size+1; lifo[size]:= x END

*PROC pop(VAR x: int)
 BEGIN x:= lifo[size]; size:= size-1 END

*PROC empty: Bool
 BEGIN empty := (size=0) END

BEGIN size:=0 END
```

A * az eljárások előtt azt jelzi, hogy ezeket a modul exportálja. A programból kiolvasható, hogy az EDISON megváltoztatta a tipus megadásnak a PASCAL-ban bevezetett és azóta igen elterjedt jelölését (a stack tipus megadása PASCAL jelölésben TYPE stack = ARRAY 1..max of integer lenne). Ez szerencsés módositásnak tünik, mert elejét veszi a tipus azonosságok körüli vitáknak, amelyek a különböző PASCAL implementációk során zajlottak.

A párhuzamosság alapegysége a processz. A processzek egy párhuzamos utasitáson belül helyezkednek el, és a párhuzamos utasitás végrehajtása azt jelenti, hogy a processzek egymással párhuzamosan futnak. Az utasitás általános ismertetése helyett álljon itt egy példa:

```
COBEGIN 1 DO buffer END
ALSO    2 DO producer
ALSO    3 DO consumer END
```

A DO előtt álló számok ugynevezett processz-konstansok, jelentésük gépfüggő, jelenthetik például a végrehajtásra kijelölt processzor számát. A buffer, producer és consumer egy-egy processzt jelöl. A processzek hozzáférhetnek közös adatokhoz. A közös adatokon való müveletek szinkronizálására szolgál a WHEN utasitás. A WHEN utasitásban a WHEN és END kulcsszavak között tetszőleges számu feltételes utasitás lista található. A feltételes utasitás lista formálisan teljesen azonos a feltételes utasitás (IF), a ciklus utasitás (WHILE) és a szinkronizációs utasitás (WHEN) esetében. (Ez a formális egyezés szép példa az EDISON frappáns egyszerüsitéseire.) A WHEN utasitás végrehajtása két fázisban történik:

1. Szinkronizációs fázis: a WHEN utasitást kiadó processz felfüggesztésre kerül addig, amig egyetlen más processz sem tartózkodik egy WHEN utasitás kritikus fázisában.

2. Kritikus fázis: a szinkronizációs utasitás végrehajtásra kerül, teljesen ugy, mint egy feltételes (IF) utasitás. Ha valamennyi feltétel hamis, akkor a processz visszakerül a szinkronizációs fázisba, ha nem, akkor a sorrendben első teljesülő feltételhez tartozó utasitások végrehajtása után a WHEN utasitás befejeződik. Egy processz csak véges ideig tartózkodik a szinkronizációs fázisban, feltéve, hogy valamennyi processz véges idő alatt befejezi kritikus fázisát.

A WHEN utasitás szemmelláthatóan a feltételes kritikus régió elvére épül. Példaként tekintsük ismét a fogyasztó/ /termelő elven dolgozó puffert:

```
MODULE "buffer"
  VAR slot: char; full: Bool

 *PROC put(c:char)
  BEGIN
    WHEN NOT full DO
      slot:= c; full:= TRUE
    END
  END

 *PROC get(VAR c: char)
  BEGIN
    WHEN full DO
      c:= slot; full:= FALSE
    END
  END

BEGIN full:= FALSE END
```

A megoldás kétségtelenül szebb, mint azok, amelyek a monitoron és explicit WAIT, SEND müveleteken alapulnak (MODULA, Konkurrens Pascal). A feltételes kritikus régió szép koncepciója azért merült sokáig feledésbe, mert a feltételek többszöri kiértékelése miatt rossz volt a hatásfokuk. Egy multi-mikroprocesszoros rendszerben azonban, ahol minden processz külön fizikai processzoron futhat, ez a körülmény nem zavaró többé, hiszen csak az a processzor hajt végre többszörös kiértékelést, amelyiknek amugy sincs más tennivalója. [BrH81c]-ben egy kis gyüjtemény található (meglehetősen egyszerü) EDISON programokból.

3.7 ÖSSZEGZÉS

A bemutatott hét programnyelv közös vonása a párhuzamosság és a modularitás támogatása. Valamennyi alkalmas rendszerprogramozási feladatok megoldására. Az egyes nyelvek pontos alkalmazhatósági területei függenek az itt nem tárgyalt, egyéb tulajdonságoktól. A PORTAL-t és az EDISON-t ipari környezetben használják számos célra, egyebek közt folyamatirányitási, real-time rendszerek készitésére. A MESA, személyes számitógép nyelveként, igen széles körü felhasználásra talál egy olyan kutatóközpontban, ahol kb. 1000 db személyes számitógép kapcsolódik egy lokális hálózatba. A Konkurrens Pascal és a MODULA nyelvek eddig elsősorban egyetemeken találtak alkalmazást, utóbbi elsősorban, mint egy személyi számitógép kizárólagos programnyelve. Az ADA forditói általában még csak előkészületben vannak, a nyelv talán a legnagyobb horderejü valamennyi közül, és igy bizonyára a legszélesebb körben is alkalmazható lesz, a hagyományos adatfeldolgozási feladatokat is beleértve. Ez az általánosság viszont elzárhatja a kisebb gépkategóriáktól, ahol a szerényebb nyelvek (EDISON, MODULA-2) könnyebben terjedhetnek.

Az áttekinthetőség kedvéért tekintsük át összegzésül egy táblázatban a tárgyalt nyelveket, a multiprogramozási sajátságaik szempontjából. (Egy teljes összehasonlitás, a hagyományosabb programnyelveket is bevonva, külön tanulmányt érdemelne.) A táblázat tartalmazza a CSP és DP javaslatokat is:

	párhuzamosság alapegysége	közös adatokhoz való hozzáférés	szinkronizáció
Konkurrens PASCAL	processz	monitor	QUEUE DELAY CONTINUE
MODULA	processz	interface modul	SIGNAL WAIT SEND
MODULA-2	korutin	modul	korutin-transzfer
ADA	task	nem javasolt	randevu
MESA	processz korutin	monitor	CONDITION WAIT, NOTIFY FORK/JOIN korutin-transzfer
PORTAL	processz	monitor	SIGNAL WAIT SEND
EDISON	processz	feltételes kritikus régió	feltételes kritikus régió
CSP	processz	nincs	input/output
DP	processz	nincs	külső eljáráshivás

4. A VIRTUÁLIS TERMINÁL MODELL

A multi-task rendszerek magasszintü nyelven történő tervezésére egy konkrét példát mutat be ez a fejezet. Az MTA-SzTAKI-ban az elmult év végén elkészitettük az Akadémiai Hálózat virtuális terminál protokolljának modelljét MODULA-2 nyelven. Erről a munkáról már külön cikkben beszámoltam [Bos81], de a továbbiakban részletesen ismertetem a modellt is, és a virtuális terminálra vonatkozó felismeréseinket is.

A számitógéphálózatok tervezésében teljesen elfogadottá vált a réteges tervezés [Dij68b,Dij71]. A réteges tervezés lényege, hogy rendszerünket hierarchikusan egymásra épülő rétegek, szintek formájában tervezzük. Minden szint csak a közvetlenül alatta lévő szint szolgáltatásait használja, és ujabb szolgáltatásokat nyujt a fölötte lévő rétegnek. A szintek kapcsolódási felületét a szintek közötti interface-nek is nevezzük. A legalsó szint minden esetben nyilván maga a hardware - a hardware is gyakran felfogható egy hierarchikus rendszernek. Az e fölött lévő szint lehet például az operációs rendszer, amely ismét rétegekre bomlik. Egy operációs rendszer réteges felépitésére találunk példát [Dij68b]-ben. Egy alkalmas felosztás található [Hop78]-ban. A szerző a legalsó szinten helyezi el az ugynevezett nucleust, amely a legalapvetőbb ütemezési feladatokat látja el. Az e fölötti szinten helyezkednek el a készülékeket meghajtó processzek (a driverek), e fölött a file kezelők, majd a felhasználói programok.

A számitógéphálózatok terén a szintek megfelelő megválasztására sok éve folyik nemzetközi erőfeszités. A viszonylag kis földrajzi területen egy üzemen belül elhelyezkedő hálózatok esetén meglehetősen heterogén a kép, bár van néhány olyan rendszer, mint például a DCS gyürü hálózat [FaM77,Far78], vagy az ETHERNET [MeB76,WeD78], amelyek igen sok egyéb rendszert alapvetően befolyásoltak. A nagyobb távolságokat átfogó hálózatok esetén a posták és szabványügyi hivatalok számára nagyon fontos, hogy nemzetközi szabványok és ajánlások álljanak rendelkezésre. Ezért ezen a területen a legutóbbi időkben meglehetősen kikristályosodott, hogy milyen szinteket kell megkülönböztetni egy ugynevezett nyilt hálózatban. Az ISO-OSI [ISO79] 7 szintes modellje olyan közismert már, hogy csak egészen röviden ismertetem az egyes szintek jelentését:

1. A legalsó szinten a fizikai szint helyezkedik el (például a V.24 vagy az X.21 postai ajánlásnak megfelelően [CCI78]).

2. A következő a vonali szint (például HDLC vagy LAP-B).

3. A harmadik szint az ugynevezett hálózati szint (például datagram vagy X.25).

4. Az átviteli, vagy transzport szint. Ez a szint minden átviteli részletkérdéstől tehermentesiti a fölötte lévő szinteket, hálózat független interface-t mutat.

5. Kapcsolat felépitési (session) szint, a kommunikáció módját irja elő, például azt, hogy melyik félnek mikor van joga adni.

6. Adatábrázolási (virtuális terminál) szint. A kommunikáció szintaktikai szabályait irja elő.

7. Felhasználói szint. A kommunikáció szemantikáját, tartalmát irja le.

A számitógéphálózat technikában is megtartották a szintek kapcsolódási felületére az interface elnevezést. A szintek belső müködése itt egészen különleges lehet. Ha egy hálózatban két processz nem a szintekre bontott architekturán keresztül kommunikál, akkor ezeket lokális, egy helyen lévő processzeknek tekintjük. Kommunikációjuk nem a hálózaton keresztül történik, noha müködésük szoros kapcsolatban lehet azzal (szerviz processzek például). Ha két processz ugy kommunikál, hogy ehhez a hálózati architekturában alatta lévő (azonos) szint szolgáltatásait veszikigénybe, akkor ezeket távoliaknak mondjuk, kommunikációjuk a hálózaton keresztül történik. (Semmi akadálya annak, hogy két, ugyanazon a processzoron futó processz egymással ne mint lokális, hanem mint távoli partnerrel kommunikáljon.) A távoli processzek kommunikációjának szabályait a hálózati irodalomban általában protokollnak nevezik.

Érdemes megjegyezni, hogy az utóbbi időkben történtek kisérletek arra, hogy a fenti értelemben vett lokális és távoli processzek kommunikációját egységesen kezeljék. Eb-

ben az irányban történtek egyrészt elméleti törekvések (ide sorolható a CSP és a DP is), és olyan gyakorlatiak is, mint az elosztott közös változó [Her77,Bos79c] és a távoli eljárás. Utóbbit Buttler Lampson ismertette az elmult évben egy előadáson (Atomic transactions) néhány más, egészen uj elvvel együtt. Bevezette a stabil tár, stabil processz és stabil monitor fogalmát, amelyek közös jellemzője, hogy állapotuk mindig definiált. A távoli eljárás lényege, hogy az egyik processzor felhivhat egy másik processzorban lévő eljárást, amely "ujraindítható": vagy teljesen lefut, vagy olyan, mintha el se indult volna. A távoli eljárás implementációja megtalálható az előadás kéziratában, MESA nyelven.

4.1 A VIRTUÁLIS TERMINÁL MODELL CÉLJA

1. A Virtuális Terminál Protokoll szöveges definiciójának megértése. Ilyen szöveges definició található például [DuS77a, Her77, INW78, Zim73]-ban és még sok egyéb helyen. A protokoll definiciók ilyen nagy száma is mutatja, hogy ez a terület még eléggé mozgásban van.

2. A szöveges definició formális megfogalmazása, és ezzel egyértelmüvé tétele.

3. A protokoll definicióban elvileg sem eldönthető implementációs kérdések fölvetése és a helyi igények szerinti, egyértelmü megoldása.

4. Vitaalapanyag szolgáltatása a virtuális terminál megvalósitóinak.

5. Egy kisérleti eszköz készitése, amivel bármilyen hibaállapot előállitható.

Az EIN hálózat VTP-jének megvalósitásához egy sokban hasonló, formális leirást készitettek [DuS77b] PASCAL nyelven. A PASCAL nem ad eszközt párhuzamosságok kifejezésére, és (gyanithatóan ezért) ez a modell a virtuális terminált egyetlen processzként irja le. A MODULA-2 lehetővé tette, hogy a virtuális terminált 4 együttmüködő processz formájában fejezzük ki, ami végülis a struktura nagymértékü egyszerüsödését jelentette.

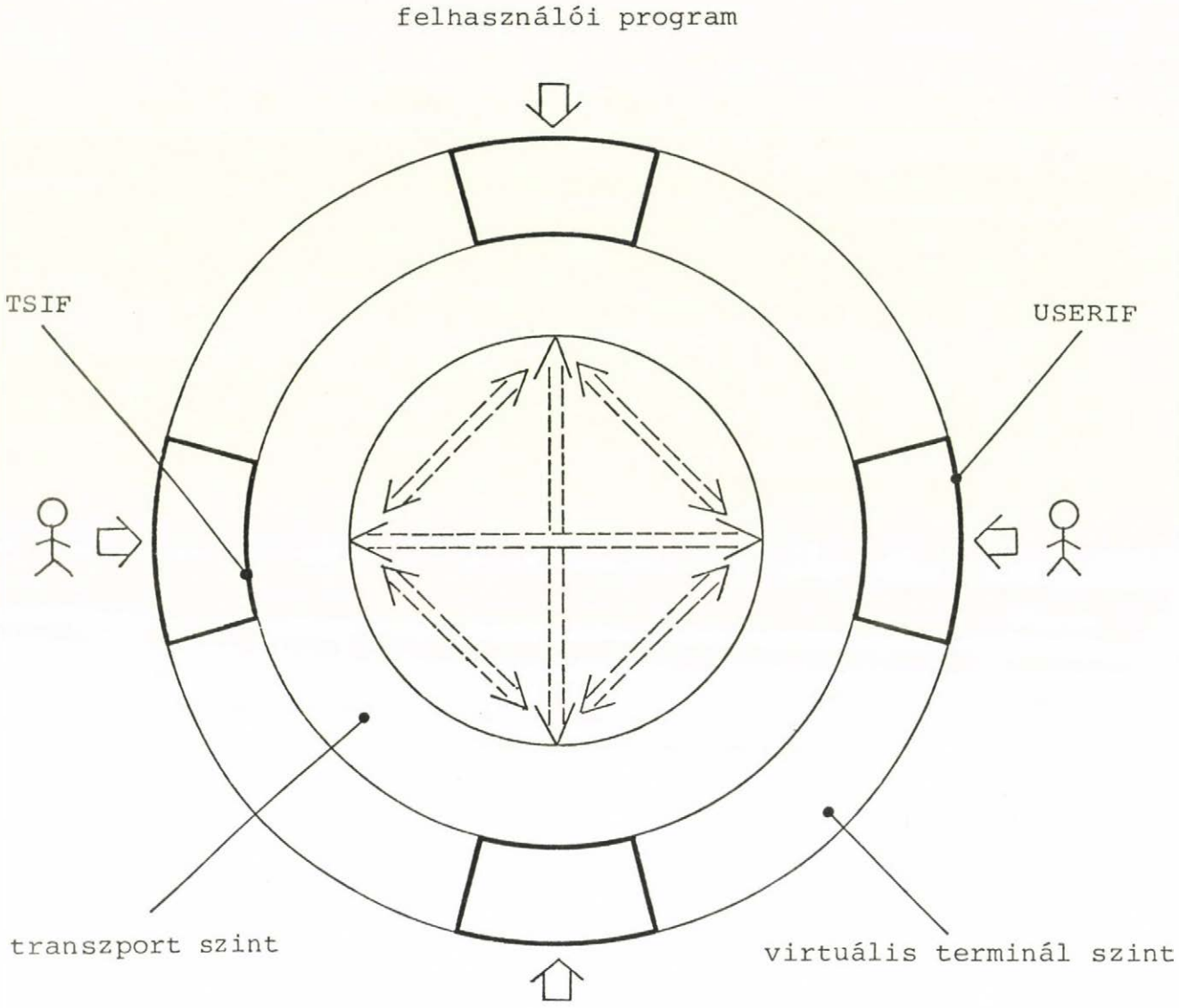
A modell tervezésében és strukturájának kialakitásában különösen értékes segitséget nyujtottak G.Bochmann és T. Joachim munkái [BoJ78, Joa77]. A szerzők az X.25 postai ajánlás [CCI78] 2. és 3. szintjét implementálták Konkurrens Pascal nyelven. Ennél valójában még sokkal többet is tettek, kidolgozták egy ilyen jellegü feladat tervezési és megvalósitási módszertanát. A feladatot hierarchikusan kapcsolódó lazán és szorosan kapcsolt egységek formájában fejezik ki, amelyeket világos módon képeznek le a Konkurrens Pascal osztály, monitor és processz fogalmaira. Az alkalmazott módszert egy korábbi tanulmányban részletesen ismertettem [Bos79c].

A virtuális terminál modell elkészülése után alapot szolgáltatott a különböző implementációk készitői közötti megbeszélésekhez. Ezen tulmenően, az R-10 csomóponti gép virtuális terminájának készitésében ténylegesen, mint formális specifikáció szerepelt. Ezzel meggyorsitotta az

R-10 implementációt, egyszersmind megkönnyitette olyan hibák kikerülését, amelyek más esetben ebben a fázisban talán még fel sem merülnek. Az R-10 operációs rendszere [Erc79a-b, Bos79a-b] különbözik valamelyest a MODULA-2-ben megvalósitott rendszertől, például jóval intelligensebb processz ütemezővel rendelkezik. Az általános elvek mégis könnyen átvihetők voltak, és a tapasztalat azt mutatja, hogy a magasszintü nyelven történt tervezés pozitivan befolyásolta az Assembly programozói stilust is.

4.2 A MODELL FELÉPITÉSE

A virtuális terminál modellhez az [INW78]-ban található virtuális terminál protokoll szolgált alapul. Ez a protokoll leirás még egy régebbi virtuális terminál koncepció szerint készült, olyan funkciókat is ellát, amelyeket az ISO modell ujabban a kapcsolat felépitési szintbe helyez. Miután ez a protokoll az ISO elképzeléshez képest ugy tünik csak többletet tartalmaz, az általunk készitett modell egy ISO virtuális terminál elkészitéséhez is jó alap. A virtuális terminál mindenképpen a felhasználói szint alatt helyezkedik el, annak nyujt szolgáltatásokat. A virtuális terminál alatti szintről modellünkben annyit tételeztünk fel, hogy képes egy maximális méret alatti blokkokat (leveleket) teljes duplex módon, sorrendtartással, hibátlanul továbbitani, valamint, hogy képes egészen rövid üzeneteket, (megszakitás, telegram) ettől az információáramtól függetlenül, nem feltétlenül hibátlanul és sorrendtartóan, de lehetőleg gyorsitva továbbitani.



A VT szint helye a hierarchiában

Ennek megfelelően a virtuális terminál modell a következő fő alapelemekből áll:

1. VT (Virtuális Terminál) modul, amely a protokoll által előirt módon tud kommunikálni egy távoli VT-vel.

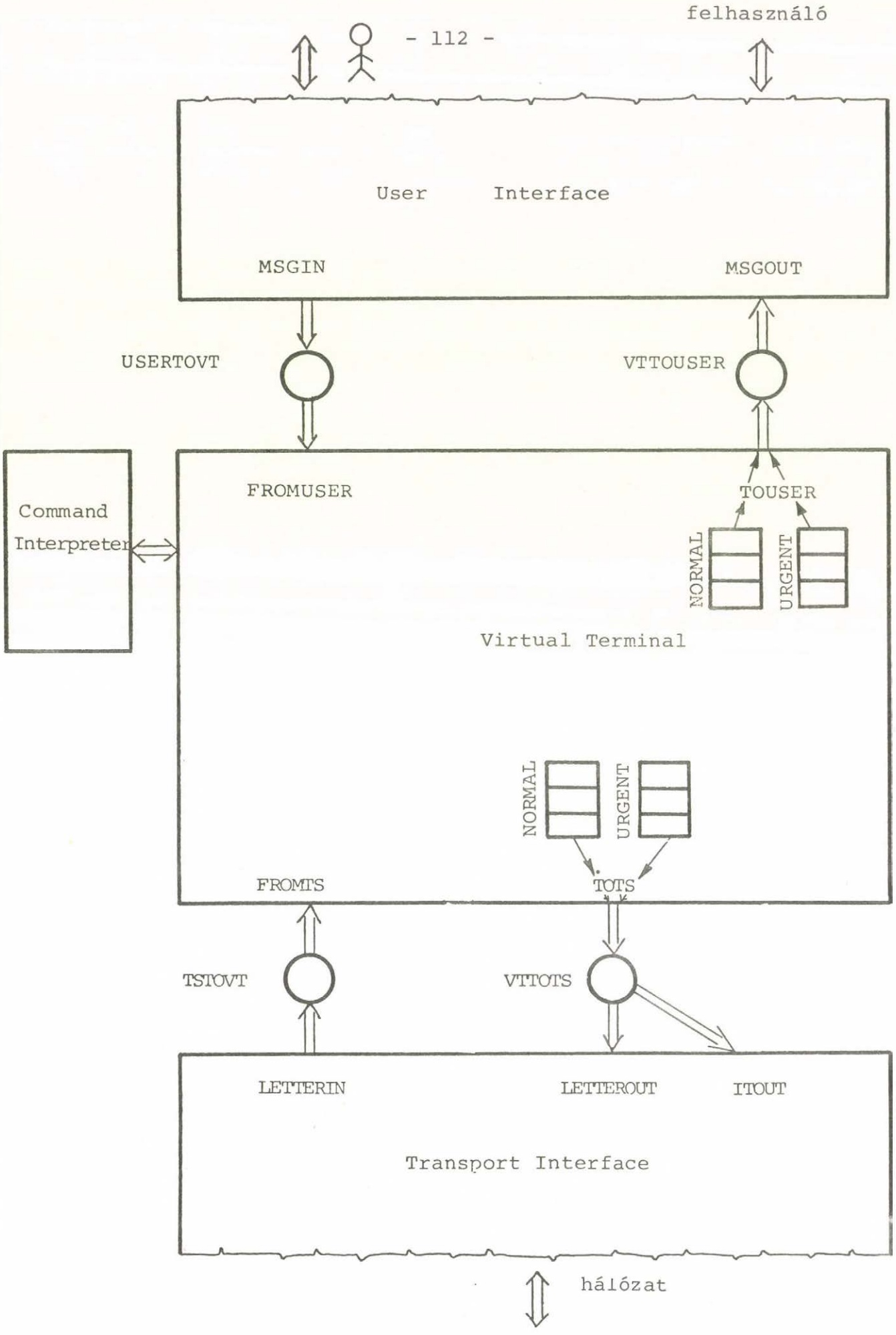
2. A USERIF (User Interface) modul, amely a felhasználóval való interface-t irja le és valósitja meg.

3. A TSIF (Transport Interface) modul, amely az imént ismertetett átviteli tulajdonságokkal rendelkező szinttel való interface-t irja le. A modell ennek az interface-nek az implementációját egy intelligens kiiró/beolvasó egységgel helyettesiti, amely lehetővé teszi, hogy láthatóvá és szimulálhatóvá tegyük a hálózaton áthaladó információkat.

4. A modell processzei, amelyek összekötik a VT modult az interface modulokkal.

5. A CMDINT (Command Interpreter) modul, amely a szövegesen begépelt parancsok értelmezését és átalakitását végzi.

6. Segédmodulok. Ütemezés, tárkezelés, pufferkezlés, véletlenszám generátor, sorkezelés stb.



A VT modell felépitése

4.3 SOROK ÉS SZINKRONIZÁLÁS

A virtuális terminál modell sorait, és a modul processzeinek szinkronizálását egyetlen modulba (a VTQ modul) koncentráltuk.

A virtuális terminál modell a bejövő információkat azonnal feldolgozza, a kimenetén pedig sorokat képez. Mindkét interface felé van kimenet (TS és USER), mégpedig mindkét irányba két sor, az egyik a sürgös (URGENT), a másik a normál (NORMAL) üzenetek számára. A VTQ modul lehetővé teszi, hogy ezekbe a sorokba be, illetve onnan ki lehessen sorolni. Ha valamelyik müveletet nem lehet végrehajtani (a sor tele van, illetve üres), akkor a hivó processzt várakozó állapotba helyezi a megfelelő feltétel teljesüléséig. Ez azt jelenti, hogy a VTQ be- és kisoroló müveletei mindig befejeződnek (feltéve, hogy az interface-ek az előirás szerint mükdönek), legfeljebb egy véges késleltetéssel, amelyből azonban a hivó processz semmit sem vesz észre. A VTQ modul ezenkivül lehetővé teszi a sorok állapotának lekérdezését, és 1 szónál hosszabb sorelemek megadását. A modul definciós része a következő:

```
DEFINITION MODULE VTQ;

FROM SYSTEM IMPORT WORD;
FROM FIFOQ IMPORT QUEUE;

EXPORT QUALIFIED ENQ,DEQ,QKIND,MSGINQ,MSGOUTQ,
       QDIRECTION,EXHAUSTEDQ,FEDUPQ;

TYPE QKIND = (URGENT,NORMAL,ANY);
     QDIRECTION = (USERQ,TSQ);
     QRANGE = [URGENT..NORMAL];

PROCEDURE ENQ(DIR: QDIRECTION; Q: QRANGE; THIS: WORD);
(*ENQUES THIS IF POSSIBLE. IF NOT, IT WAITS*)

PROCEDURE DEQ(DIR: QDIRECTION; VAR Q: QKIND; VAR THIS:
 WORD);
(*DEQUEUES THE NEXT ELEM IF POSSIBLE. IF NOT, IT WAITS*)

PROCEDURE EXHAUSTEDQ(DIR: QDIRECTION; Q: QRANGE): BOOLEAN;
(*GIVES TRUE IF THE CORRESPONDING QUEUE IS EMPTY*)

PROCEDURE FEDUPQ(DIR: QDIRECTION; Q: QRANGE): BOOLEAN;
(*GIVES TRUE IF THE CORRESPONDING QUEUE IS FULL*)

PROCEDURE MSGINQ(DIR: QDIRECTION; Q: QRANGE;
                 THIS,N: WORD);
(*CREATES A MESSAGE-HEAD RECORD AND ENQUEUES IT*)

PROCEDURE MSGOUTQ(DIR: QDIRECTION; VAR Q: QKIND;
                  VAR THIS,N: WORD);
(*DEQUEUES A MESSAGE-HEAD RECORD AND FREES ITS SAPCE*)

END VTQ.
```

A modul a már korábban ismertetett FIFOQ modul szolgáltatásait importálja, amely érkezési sorrend szerinti sorbaállitást végez. Exportál két tipust; QDIRECTION-t, amely azt mutatja, hogy a modultól igényelt szolgáltatás melyik irányba történjék (USERQ,TSQ), és QKIND-ot, amely azt mutatja, hogy a normál sorra, a sürgős sorra, vagy a

kettő bármelyikére vonatkozik-e a kért szolgáltatás (URGENT, NORMAL,ANY). A QRANGE tipus QKIND részintervalluma, és a teljesen meghatározott sorfajta megadására szolgál (ANY kizárva). Az ENQ eljárás egy adott sorba való besorolást végez, ha ez az adott pillanatban nem lehetséges, akkor kivárja, amig az adott sorban van hely. A DEQ eljárás a Q - QKIND tipusu - paramétertől függően, vagy egy meghatározott sorból, vagy egy adott irány bármelyik sorából végez kisorolást. Utóbbi esetben prioritást ad az URGENT sornak. A MSGINQ és MSGOUTQ eljárások ENQ-val és DEQ-val teljesen analóg müveletet hajtanak végre, csak a sorelemen bizonyos müveleteket is végrehajtanak (ld. az implementációnál). EXHAUSTEDQ logikai függvény, értéke igaz, ha a sor üres; FEDUPQ igaz, ha a sor tele van. Az implementáció:

```
IMPLEMENTATION MODULE VTQ;

FROM SYSTEM IMPORT WORD;
FROM PROCESSSCHEDULER IMPORT INITSIGNAL,SIGNAL,WAIT,SEND;
FROM Storage IMPORT ALLOCATE,DEALLOCATE,SetMode;
FROM FIFOQ IMPORT CREATEQ,FULLQ,EMPTYQ,GETQ,PUTQ,QUEUE;

TYPE MSG = POINTER TO MSGHEAD;
     MSGHEAD = RECORD NOTICE,TEXT: WORD; END; (*WORD*)

CONST QLENGTH = 3; (*ALL QUEUES HAVE THE SAME LENGTH*)

VAR I: QRANGE; J: QDIRECTION;
    VTQUEUES: ARRAY QDIRECTION,QRANGE OF QUEUE;
    NONFULL: ARRAY QDIRECTION,QRANGE OF SIGNAL;
    NONEMPTY: ARRAY QDIRECTION OF SIGNAL;

PROCEDURE ENQ(DIR: QDIRECTION; Q: QRANGE; THIS: WORD);
(*ENQUES THIS IF POSSIBLE. IF NOT, IT WAITS*)
BEGIN
  IF FULLQ(VTQUEUES[DIR,Q]) THEN WAIT(NONFULL[DIR,Q]) END;
  PUTQ(VTQUEUES[DIR,Q],THIS);
  SEND(NONEMPTY[DIR]);
END ENQ;
```

```
PROCEDURE DEQ(DIR: QDIRECTION; VAR Q: QKIND; VAR THIS:
 WORD);
(*DEQUEUES THE NEXT ELEM IF POSSIBLE. IF NOT, IT WAITS*)
BEGIN
  LOOP
    IF ((Q = ANY) OR (Q = URGENT))
        & NOT EMPTYQ(VTQUEUES[DIR,URGENT]) THEN
      Q:= URGENT; GETQ(VTQUEUES[DIR,Q],THIS); EXIT
    ELSIF ((Q = ANY) OR (Q = NORMAL))
          & NOT EMPTYQ(VTQUEUES[DIR,NORMAL]) THEN
      Q:= NORMAL; GETQ(VTQUEUES[DIR,Q],THIS); EXIT
    ELSE WAIT(NONEMPTY[DIR])
    END; (*IF*)
  END; (*LOOP*)

   SEND(NONFULL[DIR,Q]);
END DEQ;

PROCEDURE EXHAUSTEDQ(DIR: QDIRECTION; Q: QRANGE): BOOLEAN;
(*GIVES TRUE IF THE CORRESPONDING QUEUE IS EMPTY*)
BEGIN RETURN EMPTYQ(VTQUEUES[DIR,Q])
END EXHAUSTEDQ;

PROCEDURE FEDUPQ(DIR: QDIRECTION; Q: QRANGE): BOOLEAN;

PROCEDURE FEDUPQ(DIR: QDIRECTION; Q: QRANGE): BOOLEAN;
(*GIVES TRUE IF THE CORRESPONDING QUEUE IS FULL*)
BEGIN RETURN FULLQ(VTQUEUES[DIR,Q])
END FEDUPQ;

PROCEDURE MSGINQ(DIR: QDIRECTION; Q: QRANGE;
                 THIS,N: WORD);
(*CREATES A MESSAGE-HEAD RECORD AND ENQUEUES IT*)
VAR M: MSG;
BEGIN
  NEW(M);
      WITH M DO
    NOTICE:= N; TEXT:= THIS;
      END; (*WITH*)
  ENQ(DIR,Q,M);
END MSGINQ;

PROCEDURE MSGOUTQ(DIR: QDIRECTION; VAR Q: QKIND;
                  VAR THIS,N: WORD);
(*DEQUEUES A MESSEAG-HEAD, AND FREES THE STORAGE*)
VAR M: MSG;
BEGIN
  DEQ(DIR,Q,M);
      WITH M DO
    N:= NOTICE; THIS:= TEXT;
      END; (*WITH*)
  DISPOSE(M);
END MSGOUTQ;
```

```
BEGIN
  FOR J:= USERQ TO TSQ DO
    INITSIGNAL(NONEMPTY[J]);
    FOR I:= URGENT TO NORMAL DO
      INITSIGNAL(NONFULL[J,I]);
      CREATEQ(VTQUEUES[J,I],QLENGTH);
    END; (*FOR I*)
  END; (*FOR J*)

END VTQ.
```

Az implementációs modul definiálja a VT modul sorait tartalmazó tömböt (VTQUEUES, elemei QUEUE tipusuak). A NONFULL tömb SIGNAL tipusu elemeinek jelentése, hogy egy adott sorból kisoroltak egy elemet, a NONEMPTY elemeinek jelentése, hogy egy adott irány valamelyik sorába betettek egy elemet. A NONEMTY tömb ennek megfelelően nem rendelkezik a QRANGE dimenzióval, az elemei "több" jelentést hordoznak, mint a NONFULL tömb elemei. A VTQ modul a már többször tárgyalt termelő/fogyasztó elven dolgozik, a termelők azok a processzek, akik éppen besorolnak (ENQ), a fogyasztók, akik kisorolnak (DEQ). Az ENQ eljárás Q paraméterének tipusa QRANGE, ez biztositja, hogy besorolni csak pontosan specifikált sorba lehet. DEQ-ban Q tipusa QKIND, vagyis DEQ-t föl lehet hivni egyrészt URGENT vagy NORMAL értékkel, amely esetben csak az adott sorból próbál kisorolni, másrészt ANY-vel, aminek hatására a két sor bármelyikéből kisorolhat. Ebben az esetben először mindig a sürgős sort nézi meg, igy biztositva annak prioritást. A két logikai függvény müködése a listából könnyen kiolvasható. MSGINQ és MSGOUTQ némi elő-, illetve utófeldolgozást végez a sorelemeken, és meghivja ENQ-t, illetve DEQ-t.

4.4 PROCESSZEK ÉS INDITÁS

A virtuális terminál processzeit egyetlen modulba koncentráltuk. A virtuális terminál teljes duplex módon tart kapcsolatot mindkét interface-ével (felhasználó és hálózat), ez összesen 4 adatáramot jelent. Az egyes adatáramokat egy-egy processz "hajtja", igy a modell 4 processzből áll. Ezek többé-kevésbé függetlenek, a kapcsolódási pontok a VTQ modulban vannak (ld. Sorok, szinkronizálás). A felhasználó oldali két processz: USERTOVT (input) és VTTOUSER (output), a hálózati oldaliak: TSTOVT (input) és VTTOTS (output). A definiciós modul igen egyszerü, egyetlen eljárást exportál az inditáshoz:

```
DEFINITION MODULE VTPRCS;
EXPORT QUALIFIED STARTVT;
PROCEDURE STARTVT;
END VTPRCS.
```

Az implementáció tartalmazza a processzként indított eljárásokat, és maga az indító eljárást:

```
IMPLEMENTATION MODULE VTPRCS;
(*VIRTUAL TERMINAL PROCESSES*)

FROM Storage IMPORT ALLOCATE,DEALLOCATE;
FROM SYSTEM IMPORT WORD,ADDRESS;
FROM PROCESSSCHEDULER IMPORT STARTPROCESS;
FROM BUFFER IMPORT BUFHEAD,FREEBUF;
FROM VTQ IMPORT QKIND;
IMPORT VT;
IMPORT USERIF;
```

```
IMPORT TSIF;

TYPE WSP = ARRAY [0..250] OF WORD;
     POINTWSP = POINTER TO WSP;

VAR WORKSPACE : POINTWSP;

PROCEDURE (*PROCESS*) USERTOVT;
VAR M: BUFHEAD; MSGK: USERIF.MSGKIND;
BEGIN
  LOOP
    USERIF.MSGIN(M,MSGK);
    IF MSGK = USERIF.KILLEDLINE THEN
      FREEBUF(M)    (*THROWS AWAY KILLED LINES*)
    ELSE VT.FROMUSER(M,MSGK);
    END; (*IF*)
  END; (*LOOP*)
END USERTOVT;

PROCEDURE (*PROCESS*) VTTOUSER;
VAR M: BUFHEAD;
BEGIN
  LOOP
    VT.TOUSER(M);
    USERIF.MSGOUT(M);
  END; (*LOOP*)
END VTTOUSER;

PROCEDURE (*PROCESS*) TSTOVT;
VAR L: BUFHEAD;
BEGIN
  LOOP
    TSIF.LETTERIN(L);
    VT.FROMTS(L);
  END; (*LOOP*)
END TSTOVT;

PROCEDURE (*PROCESS*) VTTOTS;
VAR L: BUFHEAD; QK: QKIND;
BEGIN
  LOOP
     VT.TOTS(L,QK);
     IF QK = URGENT THEN TSIF.ITOUT(L)
     ELSE TSIF.LETTEROUT(L)
     END; (*IF*)
  END; (*LOOP*)
END VTTOTS;

PROCEDURE STARTVT;
BEGIN
  NEW(WORKSPACE);
  STARTPROCESS(USERTOVT,ADDRESS(WORKSPACE),TSIZE(WSP));
  NEW(WORKSPACE);
```

```
  STARTPROCESS(VTTOUSER,ADDRESS(WORKSPACE),TSIZE(WSP));
  NEW(WORKSPACE);
  STARTPROCESS(TSTOVT,ADDRESS(WORKSPACE),TSIZE(WSP));
  NEW(WORKSPACE);
  STARTPROCESS(VTTOTS,ADDRESS(WORKSPACE),TSIZE(WSP));
END STARTVT;

END VTPRCS.
```

Látható, hogy törekedtünk a processzeket a lehető legegyszerübbre irni. A hálózati oldali processzekből kiolvasható, hogy a feltételezett interface (TSIF) befelé nem különbözteti meg a megszakitást a többi üzenettől (minden a LETTERIN eljáráson át jön be), de kifelé igen (LETTEROUT és ITOUT). A STARVT eljárás processzként inditja el a felsorolt eljárásokat, az adatterületet (stack-et) a dinamikus tárterületen foglalja le.

Az egész rendszer inditását a START modul végzi:

```
MODULE START;
IMPORT VTPRCS;
FROM PROCESSSCHEDULER IMPORT PAUSE,
  SIGNAL,INITSIGNAL,WAIT,SENDDOWN;
VAR IDLESIG: SIGNAL;
PROCEDURE IDLE;
BEGIN INITSIGNAL(IDLESIG);
  WAIT(IDLESIG);
END IDLE;

BEGIN
  VTPRCS.STARTVT;
  IDLE;
END START.
```

Az IDLE eljárásra azért van szükség, mert a STARTVT eljárás a processzek elinditása után visszatér, és ezen a ponton a vezérlés visszakerülne az inditó rendszerbe. Ezt akadályozza meg IDLE inditása, amely igy a processzként is felfogható inditó rendszert "örökre" várakozó állapotba teszi.

4.5 A VT PROTOKOLL

A virtuális terminál protokollt (VTP) a VT modul valósitja meg. A VTP leirása [INW78]-ban található, szöveges definició formájában. A virtuális terminál modell egyik fő célja, hogy a leirásnak egyértelmü értelmezését adja. Nem állitom, hogy az itt következő értelmezés a legjobb, még azt sem, hogy hibátlan, de azt igen, hogy egyértelmü, és a listából viszonylag könnyen kiolvasható. Azok kedvéért, akik nem kivánnak teljes részletességgel foglalkozni a VTP implementációjával, külön felsorolom azokat a megjegyzéseket, amelyeket a modell készitése és a SzTAKI munkatársaival való konzultációk során a VTP jobb megértéséhez tettünk:

1. Terminál paraméterek. Az implementáció az irógép jellegü terminál osztályt valósitja meg (CLASS=SCROLL). A terminálhoz tartozó adatstruktura 1 dimenziós, sor orientált. A sorhossz (XSIZE) jelentése a következő: ha értéke 0, akkor a sorhossz definiálatlan, mindkét irányban tetszőleges hosszuságu szövegeket lehet küldeni (esetleg több blokkban). A virtuális terminál ebben az esetben köteles a sor végén automatikusan sort emelni. Ha XSIZE értéke nem 0 (0 < XSIZE < 256), akkor ennél hosszabb szöveget továbbitani nem szabad. A felhasználó a felelős azért, hogy a sor végére elhelyezze a soremelési parancsot. Ez az utóbbi üzemmód főleg gép-terminál kapcsolat esetén javasolt, amikor lehetővé teszi, hogy a terminálkezelő egy sor adott poziciója fölötti karakterek küldését letiltsa. Üzenetváltási szempontból a terminál kétféle üzemmóddal rendelkezhet: vagy mindkét

partner szabadon adhat-vehet bármikor (FREE), vagy pedig egyszerre az egyik terminál csak ad, a másik vesz (ALTERNATE). Utóbbi esetben a terminál aktuális paraméterei közül MODE = MYTURN, ha a terminálnak adási joga van, MODE = YOURTURN, ha nincs. Menetközben MODE vagy mindkét partnernél FREE, vagy az egyiknél MYTURN, a másiknál YOURTURN. Nem szabad olyan pillanatnak lennie, amikor MODE a két partnernél megegyezik, de nem FREE! A többi paraméter jelentése a leirásban [INW78] egyéttelmü.

2. Paramétercsere. A terminálok induláskor valamennyi paraméterre alapértelmezést tesznek, ami ugy is felfogható, mintha már lezajlott volna köztük egy cseretárgyalás (negotiation). Menet közben bármikor lehet cseretárgyalást kezdeményezni (SET) segitségével. A SET ütközésére a leirásban megadott véletlenszámos mechanizmust kell alkalmazni, ha a véletlenszámok megegyeznének, akkor SET-et mindkét oldalon el kell dobni. Az osztályra vonatkozólag a legkisebb közös értéket kell elfogadni. Az overprint/replacement paraméter közül általában csak az egyiket tudja egy terminál elfogadni (képernyőre replacement, irógépre overprint). Az XSIZE=0 értéket mindig el kell fogadni, egyéb értékre tett javaslat közül a legkisebb közös határértéket kell elfogadni. Az irányváltásra vonatkozó javaslatot mindig el kell fogadni, ez a normális (blokkfejen keresztül vezérelt) irányváltás felülbirálását jelenti, ezért nagy óvatossággal kezelendő! A másodlagos készülékekre vonatkozó javaslatot akkor szabad elfogadni, ha a terminál rendelkezik az adott perifériával (perifériákkal). A SET parancs ilyenkor azt is

jelenti, hogy a következő SET parancsig az input/output erre (ezekre) a perifériá(k)ra vonatkozik.

3. Megszakitáskezelés. A VTP kétféle megszakitást definiál. Az aszinkron megszakitás egyetlen byte mindenféle ellenőrzés nélküli átvitelét jelenti. Az aszinkron megszakitást át kell adni a felhasználónak, a VTP semmilyen automatikus intézkedést nem tesz. Igy például a PLEASE megszakitással egy felhasználó az adási jog átadását kérheti a másik felhasználótól (de nem a másik VT-től!). A szinkron megszakitás a leirásban megadott szabályok szerint történik, először megszakitásváltás a rendkivüli csatornán, és utána MARK jelek váltása a normál csatornán. A felhasználónak értesitést kell küldeni a megszakitás és a MARK bejöveteléről, de a leirásban nincs utalás arra, hogy lehetőséget kell-e adni MARK vezérlésére. Az itt közölt implementáció automatikusan küldi vissza MARK-ot. A leirás külön definiálja a PURGE szinkron megszakitást. A PURGE küldésével vagy vételével a VTP a PURGE fázisba lép, és ebben fázisban marad MARK vételéig. Ebben a fázisban a bejövő szöveg jellegü tételeket el kell dobni, de a vezérlő információkat fel kell dolgozni, a blokkfejben érkezőket is, amelyek például az adásirányt is vezérelhetik. A leirás szerint a PRUGE fázisban nem szabad szöveget küldeni. Ebből nem derül ki egyértelmüen, hogy az ez idő alatt a felhasználótól bejövő szöveget tárolni kell-e, vagy eldobni. Impmentációnk az utóbbi megoldást választotta. PURGE értelme a leirásból nem derül ki egyértelmüen, annál is kevésbé, mert függ a felhasználó tevékenységétől is. Egy értelmes felhasználói tevékenység lehet például, hogy a PURGE hatására az éppen kiküldendő egység (mondjuk file) végére megy, és MARK beérkezésére vár.

A VT modul definiciós része:

```
DEFINITION MODULE VT;
FROM BUFFER IMPORT BUFHEAD;

FROM VTQ IMPORT QKIND,QDIRECTION;
FROM USERIF IMPORT MSGKIND;
EXPORT QUALIFIED PARTYPES,ACTPAR,PARTOSET,SETSENT,
   CODEOVERPRINT,CODEREPLACE,
   CODEPURGE,CODERESUME,CODEPLEASE,XSIZEDEFAULT,
   CODEFREE,CODEMYTURN,CODEYOURTURN,
   NOTICEKIND,SDNOT,SENDNOTICE,SENDIT,SYNCITREQ,SENDSET,
   SHOWPARS,FROMUSER,TOUSER,TOTS,FROMTS;

CONST CODEPLEASE = 360B; CODERESUME = 361B;
      CODEPURGE = 351B; CODEMARK = 350B;
      CODEOVERPRINT = 1; CODEREPLACE = 0;
      CODEMYTURN = 2;  CODEYOURTURN = 3;
      CODEFREE = 0;  XSIZEDEFAULT = 0;

TYPE
     PARTYPES =(CLASS,AUXDEV,MODE,OVERPRINT,XSIZE);
     PARAMETERS = ARRAY PARTYPES OF INTEGER;
     NOTICEKIND =
       (NORMALTEXT,TEXTTURN,CMDNOTFOUND,
        CMDUNDERSPEC,GOODCMD,
        ITPROCESSING,SETPROCESSING,
        ITREC,AGREEREC,DISAGREEREC,MARKREC,
        AGREESENT,DISAGREESENT,
        ACTPARAMS,MYLIMITS,PARTNERSLIMITS);

VAR ACTPAR,PARTOSET: PARAMETERS;
    SETSENT: BOOLEAN;

PROCEDURE SDNOT(NOTICE: NOTICEKIND);
(*SENDS A NOTICE TO THE USER*)

PROCEDURE SENDNOTICE(NOTICE: NOTICEKIND; PARAM: CHAR);
(*SENDS A NOTICE AND A PARAMETER TO THE USER*)

PROCEDURE SHOWPARS(VAR P: PARAMETERS; VAR B: BUFHEAD);
(*PUTS P-S COMPONENTS INTO B IN A READABLE FORM*)

PROCEDURE SENDIT(ITCODE: INTEGER);
(*SENDS AN INTERRUPT TO THE TS*)

PROCEDURE SYNCITREQ(ITCODE: INTEGER);
(*THE USER REQUESTS A SYNCHRONOUS IT TO SEND*)
```

```
PROCEDURE SENDSET;
(*SENDS A SET-ITEM IN A SINGLE BLOCK*)

PROCEDURE FROMUSER(VAR M: BUFHEAD; MSGK: MSGKIND);
(*READS A MESSAGE FROM THE USER*)

PROCEDURE TOUSER(VAR M: BUFHEAD);
(*WRITES A MESSAGE TO THE USER. GIVES PRIORITY FOR MSGR*)

PROCEDURE TOTS(VAR LETTER: BUFHEAD; VAR RK: RKIND);
(*GIVES A LETTER TO THE TS*)

PROCEDURE FROMTS(VAR LETTER: BUFHEAD);
(*READS A LETTER FROM THE TS*)

END VT.
```

A definiciós modul 4 eljárást exportál az interface-ekkel való kapcsolatteremtés számára: FROMUSER, TOUSER, FROMTS, TOTS. A többi exportált eljárás a parancsértelmezővel való kapcsolatot biztositja. A modul exportálja néhány változóját is. Ez valójában nem helyes, mert ezzel lehetőséget teremt a külső modulok számára, hogy ezeket megváltoztassák (a régi MODULA vagy a Konkurrens Pascal nyelvek az exportált változókra csak kiolvasást engedélyeztek). Mivel a modellt egyedül készitettem, igy számithattam arra, hogy ilyenfajta visszaélés nem történik. Ennek ellenére hangsulyozni kell, hogy változók exportálása megengedett ugyan de kerülendő! Az exportált változók jelentését az implementációs modul ismertetése után a többi változóval együtt tárgyalom. A VT modul exportál még néhány olyan konstanst is, amelyek csak a TS interface helyén müködő intelligens megjelenitőt segiti, egy reális rendszerben ezeket nem kellene exportálni. Konstansok exportálása azonban semmiképpen nem okozhat hibát. Az implementációs modul:

```
IMPLEMENTATION MODULE VT;
(*VIRTUAL TERMINAL MAIN MODULE*)

FROM SYSTEM IMPORT WORD;
FROM CMDINT IMPORT CMDINTERPRET;
FROM GENRANDOM IMPORT RANDOM,INITRANDOM;
FROM BUFFER IMPORT BUFHEAD,BUFLENGTH,GETCHAR,
     PUTCHAR,PUTTEXT,FINI,INITBUF,FREEBUF,
     REMEMBER,FORGET,STRMAXB,RESETBUF;
FROM TSIF IMPORT MAXBLOCKL;
FROM USERIF IMPORT NL,STRTL,MSGKIND,BREAK;
FROM Conversions IMPORT ConvertInteger,ConvertHex;
FROM VTQ IMPORT ENQ,DEQ,QKIND,QDIRECTION,MSGINQ,MSGOUTQ;

CONST
  ITCTEXT = 300B; ITCNEWL = 301B; ITCSTRTL = 302B;
  ITCREQUEST = 340B; ITCINDICATE = 341B;
  ITCSET = 342B; ITCAGREE = 343B; ITCDISAGREE = 344B;
  BHTEXT = 0B; BHEOM = 2B; (*WITH ALARM*)
  BHEOMYRT = 22B; (*WITH ALARM*)
  BHCONTROL = 41B; BHNEGO = 101B;
  BHTSIT = 1B; (*PSEUDO BLOCK=HEAD*)
  CODESCROLL = 1; CODEPAGE = 2;
  MAXITEML = 255; (*MAXIMAL ITEM LENGTH*)
  PSEUDOIT = 0;
```

```
TYPE
  STATES = (DATA,WAITITREPLY,WAITMARKREPLY,
            MARKRECEIVED,WAITMARK);
  PARAMSET = SET OF PARTYPES;
  PARINDEX = [0..4];

VAR
  STATE: STATES;
  INX,OUTX: INTEGER; (*COUNTERS FOR FOLDING*)
  NEWPAR,PARTNER,LIMIT: PARAMETERS;
    (*ACTPAR  : ACTUAL PARAMETERS
                (DEFINED IN THE DEFINITION MODULE)
      NEWPAR  : RECOMMENDED BY SET
      PARTOSET: RECOMMENDED BY THE USER
                (DEFINED IN THE DEFINITON MODULE)
      PARTNER : PARTNER'S LIMITS; RECEIVED BY INDICATE
      LIMIT   : OWN LIMITS*)
  PINDEX: ARRAY PARINDEX OF PARTYPES; (*TYPE CONVERSION*)
  PURGSTATE: BOOLEAN;
  SENTRANDOM: CARDINAL; (*THE RANDOM NUMBER SENT BY SET*)

PROCEDURE SETLIMITS;
BEGIN
  LIMIT[CLASS]:= CODESCROLL;
  LIMIT[AUXDEV]:= 15;  (*ALL ARE ALLOWED*)
  LIMIT[MODE]:= CODEYOURTURN;  (*NO LIMIT*)
  LIMIT[OVERPRINT]:= 0;  (*UNDEFINED : NOT CHAGEABLE*)
  LIMIT[XSIZE]:= 255;
  PARTNER:= LIMIT;  (*AT START*)
END SETLIMITS;

PROCEDURE SETPINDEX;
VAR P: PARTYPES; I: INTEGER;
BEGIN I:= 0;
  FOR P:= CLASS TO XSIZE DO
    PINDEX[I]:= P; INC(I);
  END; (*FOR*)
END SETPINDEX;

PROCEDURE SETDEFAULT;
BEGIN
  ACTPAR[CLASS]:= CODESCROLL;
  ACTPAR[AUXDEV]:= 0;
  ACTPAR[MODE]:= CODEFREE;
  ACTPAR[OVERPRINT]:= CODEREPLACE;
  ACTPAR[XSIZE]:= 0;
  PARTOSET:= ACTPAR;
END SETDEFAULT;

PROCEDURE SHOWPARS(VAR P: PARAMETERS; VAR B: BUFHEAD);
VAR STR: STRMAXB;
BEGIN
```

```
    PUTTEXT('XSIZE =',B);
    ConvertInteger(P[XSIZE],4,STR); STR[4]:= 0C;
    PUTTEXT(STR,B);
    PUTTEXT(' CLASS = ',B);
    IF P[CLASS] = CODESCROLL THEN PUTTEXT('SCROLL',B)
    ELSE PUTTEXT('PAGE',B);
    END; (*IF*)
    PUTTEXT(' MODE = ',B);
    IF P[MODE] = CODEFREE THEN PUTTEXT('FREE',B)
    ELSIF P[MODE] = CODEMYTURN THEN PUTTEXT('MYTURN',B)
    ELSE PUTTEXT('YOURTURN',B);
    END; (*IF*)
    IF P[OVERPRINT] = CODEREPLACE
    THEN PUTTEXT(' REPLACEMENT',B)
    ELSE PUTTEXT(' OVERPRINT',B);
    END; (*IF*)
    PUTTEXT('  ',B);
    IF P[AUXDEV] = 0 THEN PUTTEXT('NOAUXDEV',B) ELSE
    IF P[AUXDEV] MOD 2 # 0 THEN PUTTEXT(' HARD COPY',B) END;
    IF P[AUXDEV] MOD 4 # 0 THEN PUTTEXT(' PAPER TAPE',B) END;
    IF P[AUXDEV] MOD 8 # 0 THEN PUTTEXT(' CASETTE',B) END;
    IF P[AUXDEV] MOD 16 # 0 THEN PUTTEXT(' FLOPPY',B) END;
    END; (*IF*)
    PUTCHAR(NL,B);
END SHOWPARS;

PROCEDURE SENDIT(ITCODE: INTEGER);
VAR L: BUFHEAD;
BEGIN
    INITBUF(L); PUTCHAR(CHAR(ITCODE),L);
    RESETBUF(L);
    ENQ(TSQ,URGENT,L);
END SENDIT;

PROCEDURE SYNCITREQ(ITCODE: INTEGER);
(*THE USER REQUESTS A SYNCHRONOUS IT TO SEND*)
BEGIN
    IF STATE = DATA THEN
      STATE:= WAITITREPLY;
      IF ITCODE = CODEPURGE THEN PURGSTATE:= TRUE END; (*IF*)
      SENDIT(ITCODE);
    ELSE SDNOT(ITPROCESSING)
    END; (*IF*)
END SYNCITREQ;

PROCEDURE SENDMARK;
VAR M: BUFHEAD;
BEGIN
    INITBUF(M);
    PUTCHAR(CHAR(RHCONTROL),M);
    PUTCHAR(CHAR(CODEMARK),M);
    RESETBUF(M);
    ENQ(TSQ,NORMAL,M);
```

```
END SENDMARK;

PROCEDURE SENDNOTICE(N: NOTICEKIND; PARAM: CHAR);
VAR M: BUFHEAD;
BEGIN
  INITBUF(M);
  PUTCHAR(PARAM,M);
  RESETBUF(M);
  MSGINQ(USERQ,URGENT,M,N);
END SENDNOTICE;

PROCEDURE USERNOT(N: NOTICEKIND);
VAR M: BUFHEAD;
BEGIN
  INITBUF(M);
  MSGINQ(USERQ,NORMAL,M,N);
END USERNOT;

PROCEDURE SDNOT(N: NOTICEKIND);
VAR M: BUFHEAD;
BEGIN
  INITBUF(M);
  MSGINQ(USERQ,URGENT,M,N);
END SDNOT;

PROCEDURE FROMUSER(VAR M: BUFHEAD; MSGK: MSGKIND);
VAR CH1,CH2: CHAR; IL: INTEGER; (*COUNTER FOR ITEM LENGTH*)
    LETTER,SAVE: BUFHEAD;
BEGIN
  GETCHAR(CH1,M);
  IF CH1 = 3C THEN HALT END; (*IF*)
   (*ONLY FOR TEST TIME!!!!!*)
  IF CH1 = BREAK
  THEN CMDINTERPRET(M); FREEBUF(M); RETURN; END; (*IF*)
  INITBUF(LETTER);
      CASE MSGK OF
   PARTIAL: CH2:= CHAR(BHTEXT);
  |WITHEOM: CH2:= CHAR(BHEOM);
  |WITHEOMYRT: CH2:= CHAR(BHEOMYRT);
      END; (*CASE*)
  PUTCHAR(CH2,LETTER);  (*BLOCK-HEAD*)
  IF NOT PURGSTATE THEN
    LOOP
    IF BUFLENGTH(LETTER) >= (MAXBLOCKL-2) THEN
      RESETBUF(LETTER);
      REMEMBER(LETTER,SAVE);
      PUTCHAR(CHAR(BHTEXT),SAVE);
      FORGET(SAVE);
      ENQ(TSQ,NORMAL,LETTER);
      INITBUF(LETTER);
      PUTCHAR(CH2,LETTER);
    END; (*IF*)
         CASE CH1 OF
```

```
        STRTL: PUTCHAR(CHAR(ITCSTRTL),LETTER);
               PUTCHAR(OC,LETTER); (*STRTL ITEM*)
               INX:= 1; (*RESET FOLDING*)
               GETCHAR(CH1,M);
        !NL   : PUTCHAR(CHAR(ITCNEWL),LETTER);
               PUTCHAR(OC,LETTER); (*NEWL ITEM*)
               INX:= 1; (*RESET FOLDING*)
               GETCHAR(CH1,M);
        !FINI : EXIT;  (*EXHAUSTED*)
        ELSE   PUTCHAR(CHAR(ITCTEXT),LETTER);
               REMEMBER(LETTER,SAVE);
               PUTCHAR(OC,LETTER);
                (*ONE PLACE FOR ITEM LENGTH*)
          IL:= 0;
          WHILE (CH1 # STRTL) & (CH1 # NL) &
          (CH1 # FINI) & (IL < MAXITEML)
          & (BUFLENGTH(LETTER) <= MAXBLOCKL) DO
           IF (ACTPAR[XSIZE] = 0) OR (INX <= ACTPAR[XSIZE])
           THEN
              PUTCHAR(CH1,LETTER); INC(IL); INC(INX);
            END; (*IF*)
            GETCHAR(CH1,M);
          END; (*WHILE*)
          PUTCHAR(CHAR(IL),SAVE); FORGET(SAVE);
            END; (*CASE*)
    END; (*LOOP*)
  END; (*IF NOT PURGSTATE,
          IF YES, THE TEXT MUST BE IGNORED*)
  IF BUFLENGTH(LETTER) = 1 THEN
     (*A MESSAGE WITH BLOCK-HEAD ONLY*)
    PUTCHAR(CHAR(ITCTEXT),LETTER); PUTCHAR(OC,LETTER);
  END; (*IF*)
  IF (INTEGER(CH2) = BHEOMYRT) & (ACTPAR[MODE] = CODEMYTURN)
  THEN (*THE USER WANTS TO GIVE HIS TURN*)
    ACTPAR[MODE]:= CODEYOURTURN;
    PARTOSET[MODE]:= ACTPAR[MODE];
  ELSIF PURGSTATE THEN FREEBUF(LETTER); RETURN
  END; (*IF*)
  FREEBUF(M);
  RESETBUF(LETTER); (*PREPARES LETTER FOR PROCESSING*)
  ENQ(TSQ,NORMAL,LETTER);
  (*ENQUEUES LETTER FOR TRANSPORT SERVICE OUTPUT*)
END FROMUSER;

PROCEDURE TOUSER(VAR M: BUFHEAD);
VAR N: WORD; QK: QKIND;
BEGIN
  IF STATE = MARKRECEIVED (*SUSPEND*) THEN
    QK:= URGENT
  ELSE QK:= ANY
  END; (*IF*)
  MSGOUTQ(USERQ,QK,M,N);
  IF QK = URGENT THEN MSGTOUSER(M,N); RETURN END; (*IF*)
```

```
    CASE NOTICEKIND(N) OF
 AGREEREC: ACTPAR:= PARTOSET; SETSENT:= FALSE;
           PUTTEXT('NEW AGREED PARAMETERS:',M);
           PUTCHAR(NL,M);
           SHOWPARS(ACTPAR,M);
|DISAGREEREC: SETSENT:= FALSE;
           PUTTEXT('WE CANNOT AGREE!',M); PUTCHAR(NL,M);
           PUTTEXT("ACT. PAR'S:   ",M); SHOWPARS(ACTPAR,M);
           PUTTEXT("REQU. PAR'S: ",M); SHOWPARS(PARTOSET,M);
|TEXTTURN: IF ACTPAREMODE] = CODEYOURTURN THEN
              ACTPAREMODE]:= CODEMYTURN;
              PARTOSETEMODE]:= ACTPAREMODE];
            END; (*IF*)
  ELSE  (*NOTHING TO DO*)
    END; (*CASE*)
END TOUSER;

PROCEDURE MSGTOUSER(VAR M: BUFHEAD; NOTICE: WORD);
VAR CH: CHAR;
    STR: STRMAXB; CARD: CARDINAL;
BEGIN
  GETCHAR(CH,M);
  FREEBUF(M); (*OLD CONTENT IN CH*) INITBUF(M);
      CASE NOTICEKIND(NOTICE) OF
  ACTPARAMS:  PUTTEXT('ACTUAL PARAMETERS:',M);
              PUTCHAR(NL,M);
              SHOWPARS(ACTPAR,M);
  |MYLIMITS: PUTTEXT('MY LIMITS:',M);
              PUTCHAR(NL,M);
              SHOWPARS(LIMIT,M);
  |PARTNERSLIMITS: PUTTEXT("PARTENER'S LIMITS:",M);
                   PUTCHAR(NL,M);
                   SHOWPARS(PARTNER,M);
  |GOODCMD: PUTTEXT('CI-OK',M);
  |CMDNOTFOUND: PUTTEXT('COMMAND NOT DEFINED',M);
                PUTCHAR(NL,M);
  |CMDUNDERSPEC: PUTTEXT('COMMAND UNDERSPECIFIED',M);
                 PUTCHAR(NL,M);
  |AGREESENT:
           PUTTEXT('NEW AGREED PARAMETERS:',M);
           PUTCHAR(NL,M);
           SHOWPARS(ACTPAR,M);
  |DISAGREESENT:
     PUTTEXT('I CANNOT ACCEPT THE RECEIVED PARAMETERS:',M);
     PUTCHAR(NL,M);
     SHOWPARS(NEWPAR,M);
  |ITPROCESSING: PUTTEXT('ITPROCESSING',M); PUTCHAR(NL,M);
  |SETPROCESSING: PUTTEXT('SETPROCESSING',M); PUTCHAR(NL,M);
  |MARKREC: PUTTEXT('MARK ARRIVED',M); PUTCHAR(NL,M);
  |ITREC: PUTTEXT('INTERRUPT ARRIVED: ',M);
          CARD:= CARDINAL(CH);
          ConvertHex(CARD,4,STR); STR[4]:= 0C;
          PUTTEXT(STR,M); PUTTEXT(' HEX',M); PUTCHAR(NL,M);
```

```
      END; (*CASE*)
  RESETBUF(M);
END MSGTOUSER;

PROCEDURE TOTS(VAR LETTER: BUFHEAD; VAR QK: QKIND);
VAR BH: CHAR;
BEGIN
  LOOP
    IF STATE = MARKRECEIVED (*SUSPEND*) THEN QK:= URGENT
    ELSE QK:= ANY
    END; (*IF*)
    DEQ(TSQ,QK,LETTER);
    GETCHAR(BH,LETTER);
    IF (BUFLENGTH(LETTER) = 1) & (INTEGER(BH) = PSEUDOIT)
    THEN FREEBUF(LETTER)
    ELSE RESETBUF(LETTER); EXIT
    END; (*IF*)
  END; (*LOOP*)
END TOTS;

PROCEDURE ITFROMTS(VAR IT: BUFHEAD);
VAR ITCODE: CHAR;
BEGIN GETCHAR(ITCODE,IT);
      CASE STATE OF
   DATA: SENDNOTICE(ITREC,ITCODE);
         IF SYNC(ITCODE) THEN
           STATE:= WAITMARK;
           IF INTEGER(ITCODE) = CODEPURGE
           THEN PURGSTATE:= TRUE END;
           SENDIT(INTEGER(ITCODE));
         END; (*IF*)
  |WAITITREPLY:
         SENDNOTICE(ITREC,ITCODE);
         IF SYNC(ITCODE) THEN
           STATE:= WAITMARKREPLY;
           SENDMARK;
         END; (*IF*)
  |WAITMARKREPLY: (*ERROR: IGNORE*)
  |MARKRECEIVED:
         SENDNOTICE(ITREC,ITCODE);
         PURGSTATE:= FALSE;
         STATE:= DATA;
         SENDIT(PSEUDOIT);
         SENDMARK;
  |WAITMARK: (*ERROR: IGNORE*)
      END; (*CASE*)
  FREEBUF(IT);
END ITFROMTS;


PROCEDURE FROMTS(VAR LETTER: BUFHEAD);
VAR BH: CHAR;
BEGIN
```

```
  IF BUFLENGTH(LETTER) = 1 THEN
    ITFROMTS(LETTER)
  ELSE
    GETCHAR(BH,LETTER);
        CASE INTEGER(BH) OF
     BHNEGO: NEGORECEIVED(LETTER);
    |BHCONTROL: CONTROLRECEIVED(LETTER);
     ELSE TEXTRECEIVED(LETTER,BH);
        END; (*CASE*)
  END; (*IF*)
END FROMTS;

PROCEDURE CONTROLRECEIVED(VAR L: BUFHEAD);
VAR CH: CHAR;
BEGIN
  GETCHAR(CH,L);
  IF INTEGER(CH) # CODEMARK
  THEN (*NOT DEFINED*) RETURN END; (*IF*)
      CASE STATE OF
   DATA: (*ERROR: IGNORE*)
  |WAITITREPLY:  STATE:= MARKRECEIVED; (*SUSPEND*)
                 SDNOT(MARKREC);
  |WAITMARKREPLY: PURGSTATE:= FALSE;
                  STATE:= DATA;
                  SDNOT(MARKREC);
  |MARKRECEIVED: (*ERROR: IGNORE*)
  |WAITMARK:  PURGSTATE:= FALSE;
              STATE:= DATA;
              SDNOT(MARKREC);
              SENDMARK;
     END; (*CASE*)
  FREEBUF(L);
END CONTROLRECEIVED;

PROCEDURE SYNC(ITCODE: CHAR): BOOLEAN;
BEGIN RETURN INTEGER(ITCODE) = CODEPURGE
END SYNC;

PROCEDURE TEXTRECEIVED(VAR L: BUFHEAD; BH: CHAR);
VAR CH,ITC: CHAR; I,ITL: INTEGER; M: BUFHEAD;
    NOTICE: NOTICEKIND;
BEGIN  INITBUF(M);
  IF NOT PURGSTATE THEN
    LOOP
      GETCHAR(ITC,L);
      IF ITC = FINI THEN EXIT END; (*IF*)
      GETCHAR(CH,L); ITL:= INTEGER(CH);
        CASE INTEGER(ITC) OF
     ITCTEXT:
       FOR I:= 0 TO ITL-1 DO
         GETCHAR(CH,L);
         IF (ACTPAR[XSIZE] = 0) OR (OUTX <= ACTPAR[XSIZE])
         THEN PUTCHAR(CH,M); INC(OUTX);
```

```
          END; (*IF*)
        END; (*FOR*)
      |ITCNEWL:  PUTCHAR(NL,M); OUTX:= 1; (*RESET FOLDING*)
      |ITCSTRTL: PUTCHAR(STRTL,M); OUTX:= 1; (*RESET FOLDING*)
       ELSE EXIT;
            END; (*CASE*)
    END; (*LOOP*)
  END; (*IF NOT PURGE*)
  RESETBUF(M);
  IF INTEGER(BH) = BHEOMYRT THEN NOTICE:= TEXTTURN
  ELSE   NOTICE:= NORMALTEXT;
  END; (*IF*)
  FREEBUF(L);
  MSGINQ(USERQ,NORMAL,M,NOTICE);
END TEXTRECEIVED;

PROCEDURE SENDSET;
VAR L,SAVE: BUFHEAD; P: PARTYPES; I,NOFPARS: INTEGER;
BEGIN
  IF SETSENT THEN RETURN END; (*IF*)
  INITBUF(L); I:= 0; NOFPARS:= 0;
  PUTCHAR(CHAR(BHNEGO),L);
  PUTCHAR(CHAR(ITCSET),L);
  REMEMBER(L,SAVE); PUTCHAR(OC,L);
  SENTRANDOM:= RANDOM(); PUTCHAR(CHAR(SENTRANDOM),L);
  FOR P:= CLASS TO XSIZE DO
    IF ACTPAR[P] # PARTOSET[P] THEN
      PUTCHAR(CHAR(I),L);
      PUTCHAR(CHAR(PARTOSET[P]),L);
      INC(NOFPARS);
    END; (*IF*)
    INC(I);
  END; (*FOR*)
  IF NOFPARS = 0 THEN FREEBUF(L)
  ELSE
    I:= NOFPARS*2 + 1;  (*ONE PLACE FOR RANDOM NUMBER*)
    PUTCHAR(CHAR(I),SAVE);
    SETSENT:= TRUE;
    RESETBUF(L);
    ENQ(TSQ,NORMAL,L);
  END; (*IF*)
  FORGET(SAVE);
END SENDSET;

PROCEDURE NEGORECEIVED(VAR LETTER: BUFHEAD);
VAR ITC,CH: CHAR; P,ITL,I: INTEGER;
    PT: PARTYPES; AGR: BOOLEAN; M: BUFHEAD;
    RECOMMENDED: BITSET; RCD: PARINDEX;
BEGIN
  INITBUF(M); PUTCHAR(CHAR(BHNEGO),M);
  LOOP
    GETCHAR(ITC,LETTER);
    IF ITC = FINI THEN EXIT END; (*IF*)
```

```
GETCHAR(CH,LETTER); ITL:= INTEGER(CH);
    CASE INTEGER(ITC) OF
 ITCREQUEST:
   PUTCHAR(CHAR(ITCINDICATE),M);  (*OUTGOING ITEM CODE*)
   PUTCHAR(CHAR(ITL),M); (*OUTGOING ITEM LENGTH*)
   FOR P:= 0 TO ITL-1 BY 2 DO
     GETCHAR(CH,LETTER); PUTCHAR(CH,M);
      (*PARAMETER INDEX*)
     I:= INTEGER(CH); CH:= CHAR(LIMITEPINDEXEI]]);
     PUTCHAR(CH,M); (*APPROPRIATE PARAMETER*)
     GETCHAR(CH,LETTER); (*IGNORING*)
   END; (*FOR*)
|ITCINDICATE:
   FOR P:= 0 TO ITL-1 BY 2 DO
     GETCHAR(CH,LETTER); I:= INTEGER(CH);
     GETCHAR(CH,LETTER);
     PARTNEREPINDEXEI]]:= INTEGER(CH);
   END; (*FOR*)
   SDNOT(PARTNERSLIMITS);
|ITCSET: RECOMMENDED:= {};
  GETCHAR(CH,LETTER);
   (*RANDOM NUMBER FOR CONTENTION CHECK*)
  IF SETSENT & (CH <= CHAR(SENTRANDOM)) THEN (*IGNORE*)
  ELSE
   NEWPAR:= ACTPAR;
   FOR P:= 0 TO ITL-2 BY 2 DO
     GETCHAR(CH,LETTER);
     RCD:= INTEGER(CH); (*PARAMETER INDEX*)
     PT:= PINDEXERCD];
     INCL(RECOMMENDED,RCD);
     GETCHAR(CH,LETTER);
     NEWPAREPT]:= INTEGER(CH);
   END; (*FOR*)
   PT:= CLASS; RCD:= 0; AGR:= TRUE;
   LOOP
    IF RCD IN RECOMMENDED THEN
       CASE PT OF
    CLASS,OVERPRINT,XSIZE:
        IF NEWPAREPT] > LIMITEPT] THEN AGR:= FALSE END;
    |AUXDEV:
       IF BITSET(NEWPAREPT]) - BITSET(LIMITEPT]) # {}
       THEN AGR:= FALSE
       END; (*IF*)
    |MODE:
      IF (ACTPAREPT] # CODEFREE)
          & (NEWPAREPT] # CODEFREE)
          & (ACTPAREPT] = NEWPAREPT]) THEN
         AGR:= FALSE
      ELSIF NEWPAREPT] = CODEMYTURN
      THEN NEWPAREPT]:= CODEYOURTURN
      ELSIF NEWPAREPT] = CODEYOURTURN
      THEN NEWPAREPT]:= CODEMYTURN
      END; (*IF*)
```

```
            END; (*CASE*);
          END; (*IF RECOMMENDED*)
          IF NOT AGR OR (PT = XSIZE) THEN EXIT END; (*IF*)
          INC(PT); INC(RCD);
        END; (*LOOP*)
        IF AGR THEN PUTCHAR(CHAR(ITCAGREE),M); PUTCHAR(OC,M);
          ACTPAR:=NEWPAR;
           (*ACCEPTS THE RECOMMENDED PARAMETERS*)
          SDNOT(AGREESENT);
        ELSE PUTCHAR(CHAR(ITCDISAGREE),M); PUTCHAR(OC,M);
          PUTCHAR(CHAR(ITCREQUEST),M); (*REQUEST*)
          I:= 0; FOR PT:= CLASS TO XSIZE DO INC(I,2) END;
          PUTCHAR(CHAR(I),M); (*ITEM LENGHT*) I:= 0;
          FOR PT:= CLASS TO XSIZE DO
            PUTCHAR(CHAR(I),M); PUTCHAR(OC,M); INC(I);
          END; (*FOR*)
          NEWPAR:= ACTPAR;
          SDNOT(DISAGREESENT);
        END; (*IF*)
        SETSENT:= FALSE;
      END; (*BIG IF*)
    |ITCAGREE: IF SETSENT THEN USERNOT(AGREEREC) END;
    |ITCDISAGREE: IF SETSENT THEN USERNOT(DISAGREEREC) END;
     ELSE EXIT; (*ERROR: IGNORE*)
        END; (*CASE ITC*)
  END; (*BIG LOOP*)
  FREEBUF(LETTER);
  IF BUFLENGTH(M) = 1
  THEN FREEBUF(M)  (*NO INFORMATION IN BUF*)
  ELSE RESETBUF(M); ENQ(TSQ,NORMAL,M);
  END; (*IF*)
END NEGORECEIVED;

BEGIN SETLIMITS; SETDEFAULT;
  SETSENT:= FALSE; PURGSTATE:= FALSE;
  STATE:= DATA; INX:= 1; OUTX:= 1;
  SETPINDEX;

END VT.
```

A modell összes változóját itt tárgyalom, azokat is, amelyeket a definiciós modulban deklaráltam. A virtuális terminál aktuális paramétereit az ACTPAR nevü tömb tartalmazza. A partnertől SET parancsban vett uj javaslatokat NEWPAR, a felhasználótól a parancs értelmezőn át vett javaslatokat PARTOSET tartalmazza. A VT paraméter határértékei a LIMIT nevü tömbben, a partner (INDICATE által vett) határértékei PARTNER-ben vannak. STATE és PURGSTATE a szinkron megszakitás kezelésének állapotváltozói, SENTRANDOM a SET ütközés elkerülésére utoljára kiküldött véletlenszám.

A VT modul a modellnek messze legnagyobb modulja, és talán célszerü lett volna több kisebb modulra felosztani. Külön modulba lehetett volna tenni például a megszakitás kezelését. Mivel ezek az egységek mind hivatkoznak néhány közös változóra, a VT protokollgép egységeit az egyszerüség kedvéért külön modulok helyett, külön eljárások formájában valósitottam meg. Az eljárások viszonylag rövidek.

A felhasználótól a FROMUSER eljárás veszi el az üzeneteket. Az eljárás a begépelt üzenetből a VTP leirásnak megfelelő tételeket állit elő, amelyeket blokkokba fog összse. Ha egy blokk hossza eléri a TS interface által definiált maximális blokk hosszát (MAXBLOKCKL), akkor a blokkot kiküldi BHTEXT blokkfejjel, ami azt jelzi, hogy az üzenetnek még nincs vége. A blokkfejet egyébként a felhasználó által beadott zárókarakter szerint állitja. Az eljárásból könnyen kiolvasható, hogy csak akkor teszi be a szöveget, ha XSIZE megengedi, és nincs PURGSTATE.

A felhasználónak a TOUSER eljárás adja át az üzeneteket. Ha felfüggesztett állapot van egy szinkron megszakitás feldolgozása közben, akkor csak sürgős üzenetre vár, ha nincs felfüggesztett állapot, akkor bármelyik sorból elfogad kimenő üzenetet. A felfüggesztett állapotból biztosan ki tud lépni, mert a várt megszakitás bejövetele sürgős üzenet generálását jelenti. Az üzenet kisorolása után (MSGOUTQ) a sürgőseket (amelyek mindig szerviz jellegüek) átadja az MSGTOUSER eljárásnak. A normál sorban is jön két szerviz jellegü üzenet, ezek azt jelzik, hogy a kiküldött SET-re a partner milyen választ küldött. Ezek azért kerülnek a normál sorba, mert nem szabad megelőzniük a normál adatáramot. Ennek megfelelően a paramétercserére vonatkozó állapotátmenet is itt kerül végrehajtásra!

A hálózat felől bejövő blokkokat a FROMTS eljárás veszi át. Ha a blokk hossza 1, akkor megszakitásnak tekinti, és átadja az ITFROMTS eljárásnak. Ellenkező esetben a blokkfejtől függően adja át NEGORECEIVED, CONTROLRECEIVED vagy TEXTRECEIVED valamelyikének. Az ITRFOMTS eljárás szinkron megszakitás esetén végrehajtja a megfelelő állapotátmeneteket, aszinkron esetben egyszerüen értesiti a felhasználót. CONTROLRECEIVED a jelenleg egyedüli definiált MARK vezérlőjel hatására a megfelelő állapotátmeneteket valósitja meg. NEGORECEIVED a bejövő negotiation tételeket dolgozza fel. TEXTRECEIVED feltördeli a bejövő tétel struktúrát a felhasználói interface-en definiált üzenetformátummá (ld. USERIF).

A hálózat felé kimenő blokkokat TOTS adja át. Ha felfüggesztett állapot van, akkor csak a sürgős üzeneteket engedi át. Miután az állapot megszünése után nem biztos,

hogy jön sürgős üzenet, ezért az állapot megszünését összekapcsoljuk egy pszeudo megszakitás küldésével, ami kimozdithatja TOTS-t a várakozó állapotból. Ez a megoldás nem tul szép, azért van rá szükség, mert az ütemezést összekötöttem a sorkezeléssel, tehát csak sorolási müveleten keresztül lehet SIGNAL-t küldeni. Egy másik lehetséges megoldás lett volna, bevezetni egy külön SIGNAL-t a VT modulban a felfüggesztett állapot megszünésének jelzésére. Ekkor viszont a VT modult WAIT és SEND közvetlen kiadásával kellett volna terhelni, ami még kevésbé szép. Ennél a megoldásnál lényegesen jobbat csak egy intelligensebb processz ütemező segitségével lehetne adni. (A MODULA-2 nyelv lehetővé teszi, hogy ilyet készitsünk.)

4.6 AZ INTERFACE-EK

A virtuális terminál modell nemcsak a protokollt megvalósitó modult (VT) tartalmazza, hanem a virtuális terminál réteg interface-eit is. Ez igen lényeges pont, mert a VTP definiciók ezt a kérdést teljes egészében az implementációkra hagyják. Ez elvileg helyes is, egy tényleges hálózatban azonban különös jelentősége van annak, hogy a felhasználók felé mutatott interface a lehetőségek határain belül azonos legyen. Ez utóbbinál is különösen a manuális felhasználó (terminál operátor) felé mutatott interface jelentősége nagy, mert joggal elvárható, hogy egy hálózati operátor a hálózat bármelyik terminálját többé-kevésbé azonosan kezelhesse. Fizikailag különböző terminálok esetén ez nyilván csak részben lehetséges, de nem szabad a kezelésének függ-

nie attól, hogy a terminál fizikailag hova kapcsolódik! Ezért a felhasználói interface-nek az egész hálózatra kiterjedő egyértelmü definiálása a hálózat vonzereje szempontjából döntő jelentőségü.

A modellben a felhasználói interface-t igyekeztünk ugy megfogalmazni, hogy az független legyen attól, hogy a felhasználó program, vagy manuális kezelő-e. A VT a felhasználóval üzeneteken (message-eken) keresztül kommunikál. Az üzenet tetszőleges grafikus karakterek sorozata. Az üzenettel kisérő információként jár az üzenet minősitő MSGKIND , amelynek értéke lehet PARITAL (részleges, folytatás jön), WITHEOM (befejezett , WITHEOMYRT befejezett, és egyszersmind átadja az adási jogot a partnernek) és KILLENDLINE (hibás, törlendő). Az üzenetben vezérlő szerepe van néhány nem grafikus karakternek: NL (uj sor elejére), STRTL az adott sor elejére és BREAK (parancs sor kezdetét jelzi). Ezeknek a kódja nyilván függhet az implementációtól, ezt a USERIF modul exportálja:

```
DEFINITION MODULE USERIF;

FROM BUFFER IMPORT BUFHEAD;


EXPORT QUALIFIED MSGIN,MSGOUT,MSGKIND,NL,STRTL,BREAK,
        BACKSPACE,KILLINE;

CONST NL = 16C; STRTL = 23C; BREAK = 33C; (*CNTRL B*)
      BACKSPACE = 10C; KILLINE = 13C; (*CNTRL K*)

TYPE MSGKIND = (PARTIAL,WITHEOM,WITHEOMYRT,KILLEDLINE);

PROCEDURE MSGIN(VAR M: BUFHEAD; VAR MSGK: MSGKIND);
(*READS A MESSAGE FROM THE USER*)

PROCEDURE MSGOUT(VAR M: BUFHEAD);
(*GIVES A MESSSAGE TO THE USER*)

END USERIF.
```

Az implementáció manuális felhasználót szolgál ki. Tartalmaz néhány kellemes, lokális, a partner számára láthatatlan szolgáltatást. Törekszik arra, hogy többé-kevésbé független legyen a fizikai terminálról:

```
IMPLEMENTATION MODULE USERIF;

FROM VT IMPORT ACTPAR,PARTYPES,CODEYOURTURN;
FROM VTQ IMPORT QKIND,QDIRECTION,EXHAUSTEDQ,FEDUPQ;
FROM PROCESSSCHEDULER IMPORT SIGNAL,INITSIGNAL,AWAITED,
     WAIT,SEND;
FROM RESOURCE IMPORT SEMAPHORE,INITSEM,REQUEST,RELEASE;
FROM TTYS IMPORT CHIN,CHOUT,TELETYPES;
FROM BUFFER IMPORT BUFHEAD,INITBUF,RESETBUF,FREEBUF,STRMAXB,
     MAXB,GETCHAR,PUTCHAR,PUTTEXT,BUFLENGTH,FIN1;

CONST  MAXPHYS = 80; CR = 15C; LF = 12C;
       ESC = 33C; BELL = 7C;
       EOM =CR; EOMYRT = LF;
       RESET = 22C; (*CNTRL R*)
       LINESEP = 14C; (*CNTRL L*)
       ECHOSWITCH = 5C; (*CNTRL E*)
       HOLDSWITCH = 10C; (*CNTRL H*)
       FOLDSWITCH = 6C; (*CNTRL F*)
       DISCARDSWITCH = 4C; (*CNTRL D*)
       MAXMSGSIZE = 256;

VAR    XPHYS: [1..MAXPHYS+1]; (*LINE INDEX*)
       PRESDEV: SEMAPHORE;     (*PRESENTATION DEVICE*)
       ECHO,HOLD,FOLD,DISCARD: BOOLEAN;
       RESUME: SIGNAL; (*FOR HOLDING*)
       TTY: TELETYPES; (*ACTUAL DEVICE*)

PROCEDURE NEWL(ANYWAY: BOOLEAN);
(*PRINTS NEW LINE IF ANYWAY IS TRUE, OR ECHO IS TRUE*)
BEGIN
  IF ANYWAY OR ECHO THEN
```

```
    CHOUT(CR,TTY); CHOUT(LF,TTY); CHOUT(0C,TTY);
  END; (*IF*)
  XPHYS:= 1;
END NEWL;

PROCEDURE STARTL(ANYWAY: BOOLEAN);
BEGIN
  IF ANYWAY OR ECHO THEN
    CHOUT(CR,TTY); CHOUT(0C,TTY);
  END; (*IF*)
  XPHYS:= 1;
END STARTL;

PROCEDURE GRAPHICAL(CH: CHAR): BOOLEAN;
BEGIN RETURN CH >= 40C;
END GRAPHICAL;

PROCEDURE INCLP(ANYWAY: BOOLEAN);
(*INCREMENTS LINE POINTERS AND PRINTS NEW LINE IF FOLDING IS
 REQUIRED*)
BEGIN
  IF XPHYS < MAXPHYS THEN INC(XPHYS)
  ELSIF FOLD THEN NEWL(ANYWAY)
  ELSE XPHYS:= 1;
  END; (*IF*)
END INCLP;

PROCEDURE DOBACK(ANYWAY: BOOLEAN);
BEGIN
  IF ANYWAY OR ECHO THEN
    IF XPHYS > 1 THEN
      CHOUT(BACKSPACE,TTY); CHOUT(' ',TTY);
      CHOUT(BACKSPACE,TTY);
      DEC(XPHYS)
    ELSE CHOUT(BELL,TTY);
    END; (*IF*)
  END; (*IF*)
END DOBACK;

PROCEDURE PRINTOUT(CH: CHAR; ANYWAY: BOOLEAN);
(*PRINTS OUT A CHARACTER IF ANYWAY IS TRUE, OR IF ECHO IS
 TRUE.
  IT INCREMENTS XPHYS FOR GRAPHICAL SYMBOLS*)
BEGIN
  IF ANYWAY OR ECHO THEN CHOUT(CH,TTY) END; (*IF*)
  IF GRAPHICAL(CH) THEN INCLP(ANYWAY) END; (*IF*)
END PRINTOUT;

PROCEDURE DEFAULT;
BEGIN
  ECHO:= TRUE; FOLD:= TRUE; HOLD:= FALSE; DISCARD:= FALSE;
END DEFAULT;
```

```
PROCEDURE MSGIN(VAR M: BUFHEAD; VAR MSGK: MSGKIND);
VAR CH: CHAR;
BEGIN
  LOOP CHIN(CH,TTY);
    (*LOOKS FOR THE FIRST NON-FUNCTIONAL CHARACTER*)
       CASE CH OF
    BREAK: EXIT;  (*COMMANDS MAY BE READ ALWAYS*)
   |ECHOSWITCH: ECHO:= NOT ECHO;
   |RESET: DEFAULT;
   |DISCARDSWITCH: DISCARD:= NOT DISCARD;
   |FOLDSWITCH: FOLD:= NOT FOLD;
   |HOLDSWITCH: HOLD:= NOT HOLD;
      IF NOT HOLD THEN
        WHILE AWAITED(RESUME) DO SEND(RESUME) END; (*WHILE*)
      END; (*IF*)
    ELSE (*CASE*)
      IF (ACTPAR[MODE] # CODEYOURTURN)
         & NOT FEDUPQ(USERQ,NORMAL) THEN
         (*THIS STATEMENT ENSURES,
           THAT COMMANDS MAY BE READ ALWAYS*)
       EXIT; (*STARTS TO PROCESS NON-FUNCTIONAL CHAR*)
      END; (*IF*)
       END; (*CASE*)
  END; (*LOOP*)
  REQUEST(PRESDEV);
  INITBUF(M);
  IF CH = BREAK THEN
    PUTCHAR(CH,M);
    IF XPHYS # 1 THEN NEWL(TRUE) END; (*IF*)
    PRINTOUT('>',TRUE); PRINTOUT(' ',TRUE);
    CHIN(CH,TTY);
  END; (*IF*)
  LOOP
       CASE CH OF
    KILLINE: PUTCHAR(CH,M); MSGK:= KILLEDLINE;
             IF XPHYS # 1 THEN NEWL(FALSE) END; (*IF*)
             EXIT;
 (*|BACKSPACE: DOBACK(FALSE); STEPBACKWARD(M,1);*)
   |EOM     : PUTCHAR(NL,M);
             (*INSERTS AN ADDITIONAL NEW LINE*)
             (*PUTCHAR(CH,M);*) MSGK:= WITHEOM;
             NEWL(FALSE); EXIT;
   |EOMYRT  : (*PUTCHAR(CH,M);*) MSGK:= WITHEOMYRT; EXIT;
   |NL      : PUTCHAR(CH,M); NEWL(FALSE);
   |STRTL   : PUTCHAR(CH,M); STARTL(FALSE);
   |LINESEP: NEWL(FALSE);
    ELSE         PUTCHAR(CH,M); PRINTOUT(CH,FALSE);
       END; (*CASE*)
    IF BUFLENGTH(M) = MAXMSGSIZE THEN
        MSGK:= PARTIAL; EXIT; (*PARTIAL MESSAGE*)
    END; (*IF*)
    CHIN(CH,TTY);  (*READ NEXT CHAR*)
  END; (*LOOP*)
```

```
  RESETBUF(M);  (*PREPARES IT FOR PROCESSING*)
  RELEASE(PRESDEV);
END MSGIN;

PROCEDURE MSGOUT(VAR M: BUFHEAD);
VAR CH: CHAR;
BEGIN
  REQUEST(PRESDEV);
  RESETBUF(M);
  LOOP
  IF DISCARD THEN EXIT END; (*IF*)
  IF HOLD THEN
    RELEASE(PRESDEV);
     (*FREES IT FOR THE TIME IT IS SUSPENDED*)
    WAIT(RESUME);
    REQUEST(PRESDEV);  (*OCCUPIES IT AGAIN*)
  END; (*IF*)
    GETCHAR(CH,M);
       CASE CH OF
    FINI       : PRINTOUT(BELL,TRUE);
                 EXIT;
(*  !EOM       : PRINTOUT(BELL,TRUE); EXIT;
    !EOMYRT    : PRINTOUT(":",TRUE); PRINTOUT(BELL,TRUE);
                 IF XPHYS # 1 THEN NEWL(TRUE) END; (*IF*)
                 EXIT;  *)
    !NL        : NEWL(TRUE);
    !STRTL     : STARTL(TRUE);
     ELSE        PRINTOUT(CH,TRUE);
       END; (*CASE*)
  END; (*LOOP*)
  FREEBUF(M);
  RELEASE(PRESDEV);
END MSGOUT;

BEGIN INITSEM(PRESDEV,1); INITSIGNAL(RESUME); XPHYS:=1;
  TTY:= TTO; DEFAULT;
END USERIF.
```

Az XPHYIS nevü változó a soron belüli poziciót tartalmazza. A PRESDEV nevü szemafor a kiirómü lefoglalására szolgál egy üzenet kiirásának vagy beolvasásának tartamára. Az ECHO, HOLD, FOLD és DISCARD logikai változók helyi funkciókat vezérelnek. ECHO igaz, ha a legépelt karaktereket vissza kell irni, hamis, ha nem. Ha HOLD igaz, akkor a kiirandó üzenetek várnak (a RESUME SIGNAL-on), ha hamissá válik, akkor folytatódik a kiirás. Ha FOLD igaz, akkor a kiirómü fizikai végének elérésekor soremelést kell csinálni, ha hamis, akkor nem. Ha DISCARD igaz, akkor a kiirandó üzeneteket el kell dobni, amig ujra hamissá nem válik. Az ECHO és FOLD kapcsolók a fizikai terminálтól való függetlenséget szolgálják (az automatikusan echózó és sort emelő terminálokon ezeket nyilván hamis értékkel kell ellátni). Az MSGIN eljárásban látható, hogy a kiirómüvet csak az első karakter beolvasása után foglalja le (REQUEST-tel). Ez lehetővé teszi, hogy a helyi vezérlő funkciókat kiirás közben is lehessen igényelni. Ez annál is indokoltabb, mert ezek többsége a kiirást vezérli.

A hálózati interface-ről tett feltételezéseket már ismertettük. Ezek megfogalmazása MODULA-2-ben:

```
DEFINITION MODULE TSIF;

FROM BUFFER IMPORT BUFHEAD;
EXPORT QUALIFIED LETTERIN,LETTEROUT,ITOUT,MAXBLOCKL;

CONST MAXBLOCKL = 255;

PROCEDURE LETTERIN(VAR L: BUFHEAD);
(*READS A LETTER FROM THE TS*)

PROCEDURE LETTEROUT(VAR L: BUFHEAD);
(*GIVES A LETTER TO THE TS*)

PROCEDURE ITOUT(VAR IT: BUFHEAD);
(*GIVES AN INTERRUPT TO THE TS*)

END TSIF.
```

Lényeges pont, hogy a MAXBLOCKL nevü konstans a TS interface tulajdona, a VT ezt innen köteles importálni. A TS interface implementációját a modellben egy kiiró/beolvasó helyettesiti, amely lehetővé teszi, hogy a hálózat belsejében áthaladó információkat megjelenitsük, és tetszőleges (helyes vagy hibás) bemeneteket állitsunk elő. A modell igy szemléltető és tesztelési eszköz. A TSIF modul implementációs része bármikor lecserélhető egy igazi átviteli modulra.

KÖVETKEZTETÉSEK

A párhuzamos rendszerek és különösen az operációs rendszerek programozását ma még mindig általában Assembly szintü nyelven végzik. A dolgozat bemutatta, hogy ma már nagy mennyiségben léteznek olyan jól definiált magasszintü nyelvek, amelyekkel az ilyen jellegü feladatok is megoldhatók. A 3. fejezetben példát láthattunk egy processz ütemező és a fizikai input/output programozására MODULA-2-ben. A 4. fejezet arra mutatott példát, hogy egy magasszintü nyelv a párhuzamosságot és a dekompoziciót nemcsak támogatja, szinte kényszeriti is. A Virtuális Terminál Modell struktúrája valószinüleg nem lett volna ilyen, ha nem MODULA-2-ben, hanem mondjuk Assembler-ben készült volna. Hogy pontosabbak legyünk, a modell el sem készült volna Assembler nyelven, mert egyszerüen nem lett volna rá idő. Igy viszont kevesebb, mint 3 hónap alatt egyedül elkészitettem a modellt, az összes segédfunkcióval együtt.

A párhuzamos rendszerek tervezésében az átlagosnál jóval óvatosabbnak kell lennünk. Ennek az az oka, hogy ezek néha reprodukálhatatlan hibákat tartalmaznak, amelyeknek felderitése szinte reménytelen. Egy kolléga szellemes hasonlata szerint ez a munka hasonlit a detektivéhez, itt a hulla, de sehol semmi nyom, és az esetet megismételni nem lehet (és nem is tanácsos). Ebből a helyzetből egyetlen kézenfekvő kiut van: olyan programokat kell irni, amelyekben nincs ilyen hiba. Ez utópisztikusnak tünik, és talán nem is oldható meg teljesen. Hogy erre mégis egyáltalán esélyünk legyen, nem nélkülözhetjük a magasszintü nyelvek által nyujtott kényelmet és biztonságot.

A helyes programok irása morális kérdés is, kétféle értelemben is. Morális kérdés egyrészt, mert a hibás programok a legváltozatosabb károkat okozhatják. C.A.R. Hoare a Turing dij átvétele alkalmából irt cikkében (CACM 1980 október) néhány ijesztő példát hoz erre. (Néhány évvel ezelőtt elveszett az ürben egy Venus rakéta, mint később kiderült annak köszönhetően, hogy a vezérlő FORTRAN programban egy hibás azonositó keletkezett az automatikus deklaráció jóvoltából.) De morális kérdés a helyes programozás azért is, mert a rossz és a jó, a csunya és a szép tevésének kérdése mindig az. E.W. Dijkstra 1978-ban Zürichben tartott előadásán azt mondta erről, hogy mindenki, akinek egészséges keze van, felelős azért, ha csunyán ir. Mindenki, akinek egészséges értelme van, felelős azért, ha rossz programokat ir.

IRODALOMJEGYZÉK

[Ada80] ADA reference manual
U.S. DoD July 1980

[And79] Andler, S.: "Synchronization primitives and the verification of concurrent programs" In Operating Systems: Theory and Practice P. Lanciaux(ed.) North-Holland 1979

[Bar79] Bárdos A.: "A programbizonyitás alapjai"
KSH NSZÁMOK Budapest 1979

[BoJ78] Bochmann, G., Joachim, T.: "Development and structure of an X.25 implementation"
Universite de Montreal, April 1978

[Bos75] Böszörményi L., Braun P., Horniák G., Vid Ö.: "Több terminált kezelő rendszer kialakitása a VEIKI R40 számitógépén"
VEIKI-SZK-145, Budapest 1975, november

[Bos79a] Böszörményi L.: "A VEIKI R40 számitógépéhez csatlakozó R10 alapu front-end"
Előadás kézirat 1979 május

[Bos79b] Böszörményi L.: "Az R10 alapu kommunikációs processzor rendszerterve"
VEIKI-SZK-53.99-055, Budapest, 1979 május

[Bos79c] Böszörményi L.: "Néhány müködő X.25 implementáció irodalmának feldolgozása"
VEIKI-50.99-001 (OMFB megb.) 1979 december

[Bos81] Böszörményi, L.: "MODULA-2 used in the implementation of a Virtual Terminal MOdel"
MTA-SzTAKI Working Paper Budapest June 1981

[BrH73] Brinch Hansen,P.: "Operating System Principles"
Prentice Hall, Englewood Cliffs, N.J., 1973

[BrH77] Brinch Hansen, P.: "The architecture of concurrent programs"
Prentice Hall, Englewood Cliffs, N.J., 1977

[BrH78] Brinch Hansen, P.: "Distributed Processes: A concurrent programming concept"
Comm. ACM Vol.21. N11, p.934-941 Nov. 1978

[BrH81a] Brinch Hansen, P.: "EDISON - a multiprocessor language"
SOFTWARE Practice and Experience
Vol.11 p. 325-361 April 1981

[BrH81b] Brinch Hansen, P.: "The design of EDISON"
SOFTWARE Practice and Experience
Vol. 11 p.363-396 April 1981

[BrH81c] Brinch Hansen, P.: "EDISON programs"
SOFTWARE Practice and Experience
Vol. 11 p.397-414 April 1981

[CCI78] CCITT Recommendations, Geneve 1978

[Dah78] Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R.:
"Strukturált programozás"
Müszaki könyvkiadó, Budapest 1978

[Dij68a] Dijkstra, E.W.: "Cooperating sequential processes"
In Programming Languages, F.Genuys(ed.),
Academic Press, New York, 1968

[Dij68b] Dijkstra, E.W.: "The structure of the THE multiprogramming system"
Comm. ACM 11,5 May 1968 , 341-346

[Dij68c] Dijkstra, E.W.: "Go to statement considered harmful"
Comm. ACM Vol.11,N.3,March 1968

[Dij71] Dijkstra, E.W.: "Hierarchical ordering of sequential processes"
Acta Informatica 1, p.115, 1971

[Dij75] Dijkstra, E.W.: "Guarded commands, nondeterminacy and formal derivation of programs"
Comm. ACM Vol. 18, N8, p.453-457 Aug. 1975

[DuS77a] Dünki, A., Schicker, P.:
"The Virtual Terminal Definition"
EIN/ZHR/77 Zürich, September 1977

[DuS77b] Dünki, A., Schicker, P.: "Page Virtual Terminal formal implementation"
EIN/ZHR/77/024 Zürich 1977

[Erc79a] Ercsényi A.: "Az R10 alapu hálózati processzor rendszertechnikája"
Előadás kézirat 1979 május

[Erc79b] Ercsényi A.: "Az R10 alapu hálózati kommunikációs processzor"
Kézirat 1979 május

[FaM77] Farber, J., Mockapertis, P.V.: "Experience with the Distributed Computer System DCS
UCI Technical Report 116 1977

[Far78] Farber, J.: "A ring network"
Datamation p.44-46, February, 1978

[Gei79] Geissmann, L.: "Modulkonzept und separate Compilation in der Programmiersprache MODULA-2"
In Microcomputing, W.Remmele, H.Schecher Eds.
98-114, Stuttgart 1979

[Hab72] Habermann, A.N.: "Synchronization of communicating processes"
Comm. ACM Vol. 12.N.10,p.171-176,March 1972

[Her77] Hertweck, F.R., Raubold, E., Vogt, F.:
"X.25 based process-process communication -
- concepts and facilities - "
PIX/HLP/TEK Stuttgart December 1977

[Hoa69] Hoare, C.A.R.: "An axiomatic basis for computer programming",
Comm. ACM, 12 p. 576-581, 1969

[Hoa72] Hoare, C.A.R.: "Towards a theory of parallel programming"
In Operatins Systems Techniques,
Academic Press, New York, N.Y., 1972

[Hoa73] Hoare, C.A.R.: "Monitor: an Operating System structuring concept"
STAN-CS-73-401, Stanford, 1973

[Hoa78] Hoare, C.A.R,: "Communicating Sequential Processes"
Comm. ACM Vol.21, N8,p.666-677 Aug. 1978

[HoK77] Hoare, C.A.R.: McKeag, R.M.: "Structure of an Operating System"
Second Draft. The Queen's Univ. Belfast Oct. 1977

[Hop78] Hoppe, J.: "Betriebssystem-Typen und ihre Grundlagen"
Diss. ETH Nr. 6143, Zürich, 1978

[Hop8ü] Hoppe, J.: "A simple nucleus written in MODULA-2"
ETH-IFI Berichte 35 Zürich March 1980

[Ich79a] Ichbiah et al.: "Preliminary ADA reference manual"
SIGPLAN Notices, 14(6) Part A 1979

[Ich79b] Ichbiah et al.: "Rationale for the design of the ADA programming language"
SIGPLAN Notices, 14 6 Part B 1979

[Inw78] "Proposal for a Standard
Virtual Terminal Protocol"
INWG PROTOCOL February, 1978

[ISO79] ISO: Open System Interconnection
ISO/TC97/SC16

[JeW74] Jensen, K., Wirth, N.: "PASCAL User Manual and Report"
Lecture Notes in Computer Science,
Springer, Berlin, 1974

[Joa77] Joachim, T.: "Implantation du protocole standard X.25 a partir d'un modele de formalisation et de mécanismes abstraits de programmation"
Document de travail 103, Univ. de Montreal 1977

[Lal76] Lalive d'Epinay, T.: "The Virtual Computer System"
Report III-11-1 of TC8 of Purdue Eur, Wshp 1976

[Lam81] Lampson, B.W., McDaniel, K.A., Ornstein, S.M.: "An instruction fetch unit for a high-performance personal computer"

[LaP81] Lampson, B.W., Pier, K.A.: "A processor for a high-performance personal computer"
In. Proc. 7th Symp. on Comp. Architecture 1980

[Le78] Le, K.V.: "The module: a tool for structured programming"
ETH-Diss. Nr.6153,1978

[Loc67] Lőcs Gy.: "Algol/60 programozási nyelv"
Müszaki Könyvkiadó Budapest 1967

[MeB76] Metcalfe, R.M., Roggs, D.J.: "ETHERNET: Distributed Packet Switching for local computer networks"
Comm. ACM, Vol. 19.N.§, July 1976

[Mit79] Mitchell, J.G., Maybury, W., Sweet, R.: "MESA language manual"
Xerox Report CSL-79-3 April 1979

[Nag78] Nägeli, H.H.: "Der Stammcompiler. Ein Beitrag zum Übertragungsproblem"
Diss. ETH Nr. 6095 Zürich 1978

[Nag79] Nägeli, H.H.: "Programmieren mit PORTAL"
LGZ Landys & Gyr Zug 1979

[OwG76] Owicki, S., Gries, D.: "Verifying properties of parallel programs"
Comm. ACM. Vol. 19, N.5, p. 279-285, May 1976

[Sha79] Shaw, A.C.: "Software specification languages based on regular expressions"
ETH-IFI Berichte 31, Zürich, 1979

[WeD78] West, A., Davison, A.: "CNET A Cheap network for distributed computing"
Queen Mary College TR 120 London March 1978

[Wel79] Welsh, J., Lister, A., Salzman, E.J.:
"A comparison of two notations for process communication"
Proc. of the Symp. on Language Design and Programming Methodology, Sydney Sept. 1979

[WeL81] Welsh, J., Lister, A.: "A comparative study of task communication in ADA"
SOFTWARE Practice and Experience
Vol. 11 p. 257-290 1981

[Wir71] Wirth, N.: "Program development by Stepwise Refinement"
Comm. ACM, p.221-227. April 1971

[Wir77a] Wirth, N.: "MODULA: A language for modular multiprogramming"
SOFTWARE - Practice and Experience
Vol. 7, p.2-35,1977

[Wir77b] Wirth, N.: "The use of MODULA"
SOFTWARE - Practice and Experience
Vol.7, p.37-65, 1977

[Wir77c] Wirth, N.: "Design and implementation of MODULA"
SOFTWARE - Practice and Experience
Vol. 7, p.67-84,1977

[Wir77d] Wirth, N.: "Toward a discipline of real-time programming"
Comm. ACM. Vol.20,p. 577-583, Aug. 1977

[Wir78] Wirth, N.: "Systematsches Programmieren"
B.G. Teubner Stuttgart 1978

[Wir81] Wirth, N.: "The personal computer Lilith"
ETH IFI Berichte 40 Zürich April 1981

[Zim73] Zimmerman: Virtual Terminal Protocol /VTP/
propsed specifications
Cyclades, Dec. 1973.

[Wir80] Wirth, N.: MODULA-2

ETH-IFI Berichte Nr. 36. 1980 Zürich

KÖSZÖNETNYILVÁNITÁS

Köszönetet mondok kollégáimnak, akik megértésükkel és jó tanácsaikkal egyáltalán lehetővé tették, hogy ez a tanulmány elkészüljön. Külön köszönetet mondok Ercsényi Andrásnak, akivel az elmult években közösen alakitottuk ki azt a szemléletet, amelynek egyik eredménye ez a dolgozat.

Köszönöm dr. Lehel Csabának a sokoldalu segitséget.

Köszönetet mondok a Zürichi Müegyetem Informatika Tanszék munkatársainak, akik sokat segitettek, egyebek között rendelkezésemre bocsátották a MODULA-2 fordító különböző változatait.

Köszönöm a lelkiismeretes átolvasást és értékes megjegyzéseket dr. Gehér István, Szabó Miklós és Baján Péter kollégáimnak.

Köszönöm dr. Harangozó Józsefnek és dr. Kondorosi Károlynak az értékes tanácsokat.

A TANULMÁNYSOROZATBAN 1980-BAN JELENTEK MEG:

101/1980 Gerencsér László - Hangos Katalin:
Diszkrét lineáris sztochasztikus rendszerek
önhangoló szabályozása

102/1980 Pásztorné Varga Katalin: Rekurziv eljárás

103/1980 Gerencsér Piroska - Szép Endre - Zilahy Ferenc
Marton Zsolt: Robotmegfogók adaptivitása I.

104/1980 Knuth Előd - Radó Péter - Tóth Árpád:
A SDLA előzetes ismertetése

105/1980 E. Knuth - P. Radó - Á. Tóth:
Preliminary description of SDLA

106/1980 Prékopa András: Sztochasztikus programozási
modellek és alkalmazásuk

107/1980 Kelle Péter: Megbizhatósági készletmodellek
és alkalmazásuk

108/1980 Almásy Gedeon: Mérlegegyenletek és mérési hibák

109/1980 Békéssy A. - Demetrovics J. - Gyepesi Gy.:
Relációs adatbázis logikai szintü vizsgálata
funkcionális függőségek szempontjából

110/1980 Gaál A. - Soltész J. - Ruda M. - Ratkó I.:
Tanulmányok a statisztikai adatfeldolgozásról

111/1980 Benedikt Szvetlána: Nem ismételhető döntéshozatal
analizise kockázattal járó esetekben

112/1980 Verebély Pál: Többprocesszoros, osztott intel-
ligenciáju grafikus rendszerek tervezési és meg-
valósitási kérdései

113/1980 V. Visegrádi Téli Iskola

114/1980 Demetrovics János: Relációs adatmodell logikai és strukturális vizsgálata

115/1980 Gergely József: Program package for sparse matrices

1981-BEN JELENTEK MEG:

116/1981 Siegler András: Egy 6 szabadságfoku antropomorf manipulátor kinematikája és számitógépes vezérlése

117/181 Knuth Előd - Radó Péter: Principles of Computer Aided System Description

118/1981 Demetrovics János - Gyepesi György: Általános függések és lekérdezéssel kapcsolatos algoritmusok relációs adatmodellekben

119/1981 Sztanó Tamás: REAL-TIME programrendszerek eseményvezérelt szervezése

120/1981 Szentgyörgyi Zsuzsa: A számitástechnika müszaki fejlődése és társadalmi hatásai

121/1981 Vicsek Tamásné (Strehó Mária): Vizsgálatok a kezdeti érték problémák numerikus megoldásával kapcsalatban

122/1981 Andó Györgyi-Lipcsey Zsolt: Sztochasztikus Ljapunov módszerek és alkalmazásaik

123/1981 Márkusz Zsuzsanna: Intelligens interaktiv rendszerek elvi problémái

124/1981 Márkusz Zsuzsanna: Logikai alapu programozási módszerek és alkalmazásaik számitógéppel segitett épitészeti tervezési feladatok megoldásához

125/1981 Fabók Julianna: Software implementációs nyelvek

126/1981 Várszegi Sándor: Multimikroszámitógép-rendszerek

127/1981 Lipcsey Zsolt: N-személyes minőségi differenciál-játékok késleltetéssel és késleltetés nélkül