tanulmányok

168/1985

MTA Számitástechnikai és Automatizálási Kutató Intézet Budapest

MAGYAR TUDOMÁNYOS AKADÉMIA
SZÁMITÁSTECHNIKAI ÉS AUTOMATIZÁLÁSI KUTATÓ INTÉZETE

L I L I P U T H

Irták

Böszörményi László
Kovács László
Martos Balázs
Szabó Miklós

Tanulmányok 168/1985

A kiadásért felelős:

Dr.Vámos Tibor

Főosztályvezető:

Dr. Bakonyi Péter

ISBN 963 311 189 7
ISSN 0324-2951

"A dilettánsok, ha a tőlük telhetőt megtették,
mentségükre szokták mondani,
hogy a munka még nincsen készen."

(Goethe: Wilhelm Meister vándorévei)

TARTALOMJEGYZÉK

BEVEZETÉS

Ez a tanulmány összefoglalja azokat az elképzeléseket, amelyeket e pillanatban a LILIPUTH gyűjtőnévvel jelölünk. A tanulmány célja, hogy kiindulópontúl szolgáljon egy esetlegesen beinduló kutatási és fejlesztési témához. A kutatás célja kettős: egyrészt egészen általánosan kívánunk foglalkozni tervezési kérdésekkel, másrészt keressük azokat a fogalmakat, amelyek lehetővé teszik, hogy egy számitógéprendszert minél egységesebben írhassunk le. A fejlesztés célja, hogy létrehozzunk egy olyan hardware-, mikrosoftware- és software rendszert (a LILIPUTH gépet), amely a kutatási téma bázisául szolgálhat a továbbiakban. A LILIPUTH gép első változata erősen támaszkodik a LILITH gép [Wirt81] tanulmányozása során szerzett gazdag tapasztalatokra (de nem másolata a LILITH-nek). Elképzelésünk szerint a kutatás és a fejlesztés kettősségét hosszú távon is fenn kell tartani, tehát a fejlesztés tapasztalatait általánosítani kell és így bevonni a kutatásba, a kutatás eredményeit pedig a gyakorlatban is használható rendszerek készítésére kell felhasználni.

A tanulmány 1.fejezete vázlatosan áttekintést ad a tervezett kutatásról és fejlesztésről; a 2.fejezet irodalmi áttekintés a LILIPUTH géphez hasonló rendszerekről; a továbbiak részletesen ismertetik egyrészt a LILITH rendszert, másrészt a LILIPUTH rendszerről eddig megalkotott elképzeléseket. Ez utóbbiak elég vegyesek, egyes kérdések már egyértelműen eldöntöttnek tekinthetők, mások vitatottak, számos kérdés pedig még meg sem jelenik ebben a tanulmányban.

1. A LILIPUTH KUTATÁS ÉS FEJLESZTÉS VÁZLATOS ÁTTEKINTÉSE

1.1 A LILIPUTH KUTATÁS

A jelenleg LILIPUTH névvel jelzett kutatást sokkal szélesebb körűnek szánjuk, mint a LILIPUTH gép megépítéséhez kapcsolódó vizsgálódások körét.

1.1.1 Módszertani kutatás

A cél legáltalánosabban megfogalmazva: bonyolult összefüggéseket tartalmazó rendszerek tervezési módszereinek a kutatása. Bonyolult rendszerek esetén elterjedten javasolt módszer a felülről lefelé való (top-down) tervezés és az alulról felfelé való (bottom-up) kivitelezés. Egy másik megközelítés a lépésenkénti finomítás módszerét ajánlja. Akárhogy is, ezek a módszerek megegyeznek abban, hogy a tervezés és a kivitelezés egy pontján a rendszer többi részének részleteitől eltekintünk. A kivitelezés esetében ez természetes is, éppen az a terv célja és értelme, hogy ezt megtehessük. A tervezés esetében azonban a részletektől való eltekintés követelménye gyakran paradox. Ahhoz ugyanis, hogy valamitől eltekintsünk, tudnunk kell, hogy mi az, amitől el akarunk tekinteni. Új rendszer esetében ez nem triviális, a rendszerbe bevont ismert komponensek részleteire viszont úgysem terjed ki a tervezés. Csaknem biztosra vehető, hogy azok a példák, amelyek a lépésenkénti finomítás módszereit szemléltetik, nem eleve az adott módszer szerint keletkeztek, hanem utólag nyerték el szép formájukat. (Őszintébb szerzők – pl. Per Brinch-Hansen – ezt legalábbis részben be is vallják. Az is kizártnak tűnik, hogy egy professzor kiálljon a katedrára szemléltetni a lépésenkénti finomítás módszerét úgy, hogy a megoldást ne tudná előre.) Ez azt jelenti, hogy a lépésenkénti finomítás módszere kiválóan alkalmas lehet didaktikai célokra, de ellentmondásokat tartalmaz, mint tervezési módszer. Teljesen hasonló a helyzet a "top-down"-nál. Az az eszményi állapot, hogy minden megvalósítási részlettől eltekintve elképzeljük, hogy mit szeretnénk, sohasem áll elő, hallgatólagosan mindig jelen van egy csomó előfeltételezés, és nem is lehet másképp. Az a körülmény, hogy az előfeltételek egy része nem tudatosul, egyáltalán nem előny, hanem hátrány. Egy sikeres tervezésben ezért éppoly fontos a rendelkezésre álló adottságok (beleértve saját intellektuális adottságainkat is) számbavétele, mint a célkitűzések minél szabadabb megfogalmazása.

Akárhogy is, egy biztos: a sikeres tervezésnek kell hogy legyen egy olyan pontja, amikor van legalább egy ember, aki az EGÉSZ rendszert EGYBEN átlátja. (Hogy ez pontosan mit jelent, azon nem érdemes vitatkozni, ezt úgyis mindenki tudja, aki tervez.)

Az egész rendszer átlátásához vezető út lépései mindenesetre elég homályosak, és éppen ezt szeretnénk kutatás tárgyává tenni. Nem reméljük azt, hogy egyszerű képletekhez jutunk, mert a feltétlenül szükséges intuitív lépéseket biztosan semmiféle módszer sem helyettesítheti. Inkább éppen a leegyszerűsítésektől szeretnénk megszabadulni (például az olyan elképzelésektől, amelyek szerint a gondolkodás "szekvenciális" stb.). Másrészt a tervezés intuitív mivoltát szeretnénk megkülönböztetni az "ad-hoc" tervezéstől, és különösen nem kívánatosnak tartjuk a kivitelezés megkezdését, mielőtt az egész rendszert magunk előtt láttuk volna.

Egy új rendszer tervezésének alapvető paradoxona tehát ez: a rendszer egésze a részek együttműködéséből áll össze, de ezeket a részeket csak az egészből kiindulva lehet meghatározni. Ezért a tervezésnek, induljon akár fölülről, akár alulról, ezt a paradoxont előbb-utóbb, bizonyosan sok kis lépés és tévút megtétele után, "egyetlen fogással" kell feloldania.

A tervezés módszertanának tanulmányozása a fentiek értelmében csak a gyakorlatban történhet, vagyis úgy, hogy tervezünk valamit, és igyekszünk közben megfigyelni, illetve utólag értékelni ezt a folyamatot. Ebből a szempontból a tervezés témája teljesen másodlagos, csak az a fontos, hogy ne legyen eleve leegyszerűsített iskolapélda.

1.1.2 Architektúra kutatás

Az utóbbi években jelentős szemléletváltozási folyamatok indultak el a számítástechnikában. Korábban magától értetődőnek tekintett, esetleg észre sem vett alapok váltak kérdésessé. Az olyan fogalmak, mint hardware, és software, operációs rendszer, fordító, editor, stb., elvesztették egyértelmű jelentésüket, meginogtak a közöttük levő határok. Ez a megváltozott helyzet nagyobb szabadságot és több gondot jelent a tervezőknek. Fölmerül az igény olyan alapvető fogalmak megalkotására, amelyek képesek a számítástechnikai rendszer egészét átfogni, egyetlen, egységes fogalmi rendszeren belül. Miközben a számítástechnikai eszközök, szolgáltatások száma egyre nő, palettája egyre színesebb, tudományos erőfeszítések történnek az alapvető fogalmak számának csökkentésére, ahol ezeknek a fogalmaknak természetesen megfelelő horderejűeknek kell lenniük. Talán nem teljesen erőltetett az a hasonlat, ha azt mondjuk, hogy a Neumann-i világ a Newton-ihoz hasonló változásokat él át (amennyiben korábban teljesen lezártnak hitt fogalmakról derül ki, hogy tartalmuk erősen tágulhat, mások elvesztik tartalmukat stb.).

A megnövekedett szabadság következményeképpen új architektúrák jelennek meg, amelyek megkisérlik fokozottan figyelembe venni a rendszerek belső összefüggéseit. Jellegzetes példa erre a LILITH gép, amelynek különleges tulajdonsága, hogy a gép architektúráját és nyelvét (a MODULA-2-t [Wirt82]) szoros összefüggésben tervezték; a gép utasításkészlete, regiszterei stb. messzemenően figyelembe veszik a MODULA-2 fordító igényeit. A LILITH tehát szoros összefüggést teremt két hagyományosan csak periférikusan érintkező terület között. A gyakorlati eredmény, hogy mind a gépi architektúra, mint a fordító egyszerűbb és hatékonyabb a hagyományos eljárásokhoz képest. A LILIPUTH gép, amely konkrét megvalósításában (hardware és software tekintetében egyaránt) sok tekintetben eltér majd a LILITH-től, első lépésben szintén a gépi architek-

túra és a nyelvi környezet közötti kapcsolatot valósítja meg, a LILITH-hez erősen hasonló módon.

Könnyű látni azt is, hogy elvileg nem lehetetlen hasonló kapcsolatokat létesíteni további olyan rendszerkomponensek között, amelyek hagyományosan többé-kevésbé függetlennek számítanak. Csak példaként említjük, hogy igen gyümölcsöző lehetne az imént említett kapcsolatrendszerbe bevonni az operációs rendszert, sőt talán éppen ez utóbbiból kellene kiindulni. Hasonló, de egyelőre viszonylag elszigetelt mozgások észlelhetők az "editorok" területén [Burk81-a, b], amelyek szintén szerves kapcsolatba hozhatók lennének az eddig említett komponensekkel. Utolsó példaként említjük az adatbázis kezelést, amely az operációs rendszer mellett szintén aspirálhat a fogalomrendszer kulcsszerepére. Ezzel kapcsolatban felmerülhet egy program adatainak kiterjesztése térben és időben: térben, amennyeiben az adatok a központi táron kívül is elhelyezkedhetnek, és időben, amennyiben az adatok "túlélhetik" az őket létrehozó program futását. Egy ilyan koncepció bevonása a programnyelvbe szintén izgalmas kérdéseket vet fel: erre vonatkozóan is történtek már lépések (MODULA/R [Zehn83]). Az egységesítésbe bevont kategóriák persze maguk is átértékelődnek egy ilyen szemléletváltozási folyamat során: a közöttük lévő határok eltünhetnek, módosulhatnak, új kategóriák jelenhetnek meg stb.

Nyilvánvaló, hogy egy ilyen kapcsolatrendszer felépítése nagyon igényes feladat. Egyrészt azért, mert e hagyományosan elkülönülő területek áttekintése már majdnem olyan nehéz, mintha egy-egy másik szakmába kellene behatolnunk, másrészt azért, mert az egységesítési törekvés nem lehet pusztán elméleti. Semmi értelme sincs ugyanis egy olyan rendszernek, amely egységes elvek szerint épül fel, de nem kivitelezhető hatékonyan. Csak olyan ponton szabad egységesíteni és kapcsolatokat létrehozmi, ahol ez a kivitelezésben is minőségjavulást eredményezhet (különben csak áttoltuk a lovat a Karpfenstein utcába).

1.2 A LILIPUTH FEJLESZTÉS

Az előző két pontban leírt kutatási célkitűzések bázisa a LILIPUTH gép, amely mind hardware, mind software szempontból a LILITH gép [Wirt81] tapasztalataira támaszkodik. A LILITH, a már emlitett elméleti érdekessége mellett, igen jó gyakorlati tulajdonságokkal is rendelkezik, és a róla rendelkezésre álló ismereteket a LILIPUTH fejlesztést megelőző kutatásnak is tekinthetjük. A LILIPUTH ugyan mind hardware, mind software tekintetben, már az első lépésben is jelentősen eltér a LILITH-től (se hardware megoldásokat, se programokat nem veszünk át egy az egyben); de először nem tervezünk olyan nagy eltéréseket tőle, hogy ne bízhatnánk abban, hogy a teljes rendszer "összeáll". (Az őszinteség kedvéért fontosnak tartjuk leszögezni, hogy a LILIPUTH gép fejlesztésének egyik fő motivuma éppen az, hogy módunk volt a szokásosnál mélyebben megismerni és megkedvelni a LILITH-et. Ha ezek az ismeretek nem állnának a rendelkezésünkre, akkor valószinűleg egyáltalán nem kezdtünk volna bele ilyen jellegű munkába, vagy legalábbis nem pont így. Hogy ez a körülmény előnye vagy hátránya-e az egész LILIPUTH témának, ezt nem tudjuk eldönteni.)

A LILIPUTH fejlesztés egyik legfontosabb célkitűzése, hogy a "pincétől a padlásig",

a hardware, a mikrosoftware és a software teljes egészében a kezünkben legyen. Ez azt jelenti, hogy az elemek legnagyobb részét magunk fejlesztjük ki, és az esetlegesen átvett elemeket is ugyanolyan jól kell ismernünk, mint a saját fejlesztéseket. A LILITH-ről sok software áll rendelkezésre forrás szinten, ezeket elsősorban tanulási célból kívánjuk elolvasni, és nem kritikátlanul átvenni.

A LILIPUTH gép elkészítéséhez kapcsolódóan számos előkészítő jellegű munkát végeztünk eddig, amelyeket különböző helyeken ismertettünk már [Bösz81, 83, Szab84]. Ezek közül kiemelkedő fontosságú a LILITH interpreter, amely az R10 gépcsaládon (R10, R10M, R11) és Z80-on fut, és lehetővé teszi a software munkák egy részének azonnali elvégzését. A LILITH interpreter C nyelven írt változata UNIX alatt is rendelkezésre áll. A LILITH interpretereken fut a MODULA fordító, és elkészült a MODULA-ban írt szimbólikus debugger [Ercs84].

A LILIPUTH gép részletesebb ismertetése előtt leírjuk az alkalmazói szempontból legfontosabb jellemzőket, majd megkíséreljük megrajzolni a várható alkalmazások körét.

1.3 A LILIPUTH GÉP FELÉPITÉSE ALKALMAZÓI SZEMPONTBÓL

A LILIPUTH EGYFELHASZNÁLÓS, ALAPVETŐEN INTERAKTIV, JÓ GRAFIKAI KÉPESSÉGEKKEL RENDELKEZŐ, KULÖNÖSEN JÓL PROGRAMOZHATÓ, NAGYTELJESITMÉNYŰ SZEMÉLYI SZÁMITÓGÉP. Több LILIPUTH hálózatba kapcsolható, de mindegyik önállóan is képes működni (pl. rendelkezik háttértárral).

A LILIPUTH egyik legjellegzetesebb vonása, hogy architektúrája különösen jól illeszkedik a PASCAL-szerű nyelvekről való fordítás követelményeihez, ezen belül is a MODULA-2 (a továbbiakban MODULA) nyelv [Wirt82] fordításához. Ezért a LILIPUTH-on a legalacsonyabb szintű nyelv a MODULA (assembler nincs), a rendszerprogramozási feladatok is megfelelő hatékonysággal végezhetők el MODULA-ban. A MODULA-val kb. azonos szintű egyéb nyelveket nem támogatjuk. Célszerű viszont egyes alkalmazói körök érdekében új, MODULA alapú nyelveket létrehozni (amilyen pl. a MODULA/R nyelv [Zehn83], amely adatbázis kezelési feladatok irányában terjeszti ki a MODULA-t).

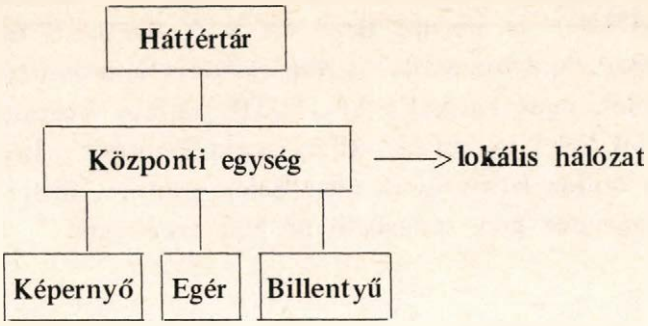
1.3.1 Hardware felépítés

A LILIPUTH gép felépítését az 1.1 ábra szemlélteti.

A központi egység hajtja végre a MODULA fordító által generált kódot. Ezenkívül képes néhány speciális utasítás végrehajtására is, amelyeket a fordító nem generál, de MODULA programból kiadhatók, és a rendszerprogramozói munkát támogatják (pl. pixel manipulációs műveletek).

A LILIPUTH központi tárának méretére a címzés oldaláról gyakorlatilag nincs korlátozás (32 címbit), tényleges kiépítését 1-4 MByte-ra tervezzük.

A LILIPUTH háttértárának a központi tárnál legalább 1 nagyságrenddel nagyobb (10 és 100 MByte között), gyors elérésű tárnak kell lennie, lehetőleg kis fizikai méretben és alacsony zajszinttel. Ezeket a követelményeket csak a Winchester diszkek teljesítik.



1.1 ábra

A LILIPUTH felépítése felhasználói szempontból

Előnyös lenne cserélhető diszk alkalmazása, mert ez lehetővé teszi, hogy több felhasználó dolgozzék ugyanazon a gépen különböző időben, egymástól teljesen függetlenül. Rögzített diszk esetén software-es úton kell hasonló függetlenséget biztosítani. Ebben az esetben a diszk kapacitására vonatkozó igény és a mentés jelentősége megnő, ez utóbbit megfelelő minőségű és kapacitású floppy-val biztosíthatjuk. Hálózatba kötött gépek esetén egy központi tároló is alkalmas lehet mentésre.

A LILIPUTH bemeneti eszközei a billentyűzet és az egér (mouse). A billentyűzetnek feltétlenül tartalmaznia kell a magyar ABC betűit, mégpedig úgy, hogy gyakorlott gépírók is használhassák, tehát a standard irógép billentyűzet köré épüljenek azok a kiegészítések, amelyek lehetővé teszik például a teljes magyar betűkészlet alkalmazását stb. Fontos követelmény, hogy csakis egyféle billentyűzetet alkalmazzunk. Az egyes programoknak természetesen joguk lehet arra, hogy az egyes billentyűknek különleges jelentést adhassanak. Az egér pozicionáló berendezés, kb. akkora mint egy szappantartó. Ha egy sima felületen (pl. az asztalon) mozgatjuk, akkor ezt megfelelő interface és software segítségével a cursor mozgatásává transzformálhatjuk. Az egéren elhelyezkedő nyomógombokhoz tetszőleges jelentéseket rendelhetünk. Az egér kiválóan alkalmas például menük által vezérelt lekérdezésekre.

A LILIPUTH alapvető megjelenítő egysége a raszter képernyő, amely nagyszámú fénypont megjelenítésére alkalmas (800*1024 A4-es állított formátum). Ez a fajta képernyő számos olyan alkalmazás előtt nyitja meg a lehetőséget, amely a hagyományos alfanumerikus képernyőkön nem áll rendelkezésre. Ilyen például egy olyan szövegszerkesztő, amely lehetővé teszi változtatható betűtípusok, ábrák, matematikai szimbólumok stb. megjelenítését. (Egy ilyen képernyő nagy terhet is ró a gépre; tárolni kell a pixel-térképet, biztosítani kell a megfelelő frekvenciájú frissítést, és nagy adatmennyiségek mozgatását bitcímről bitcímre.)

Az eddig ismertetett egységek (esetleg a floppy kivételével) kötelező tartozékai a LILIPUTH-nak (annak az elvnek megfelelően, hogy a LILIPUTH-nak önállóan is működőképesnek kell lennie). A gép opcionálisan rendelkezik lokális hálózati csatlakozással is. A

lokális hálózat alapvető szolgáltatása a file-tárolás és a nyomtatás. Erre a célra használhatunk LILIPUTH-ot is, de más rendszert is. A file-tároló esetén szükség van igen nagy méretű diszkekre, a nyomtatáshoz pedig raszteres laserprinterre. A raszteres laserprinter sarkalatos kérdés, mert enélkül a LILIPUTH grafikus lehetőségei nem sokat érnek. Kisebb minőségi igényű helyi nyomtatás céljára nagyfelbontású mátrixprintert is alkalmazhatunk. Ez különösen fontos lehet egyedi installációk számára, illetve arra az időszakra, amig megfelelő laserprinter nem szerezhető be Magyarországon.

1.3.2 Software felépítés

A LILIPUTH alapgéphez tartozónak tekintünk egy bizonyos alap software-t. Ide tartozik mindenekelőtt a MODULA fordító. Az alap operációs rendszer első számú követelménye ezért, hogy képes legyen a fordító futtatására. További követelmény, hogy kezelje a LILIPUTH alapvető erőforrásait, de úgy, hogy könnyű legyen új erőforrások felvétele, és a meglévők kezelésének módosítása (úgy is mondhatjuk, hogy az operációs rendszer legyen "nyílt").

Az alapvető erőforrások a perifériák, a központi tár és maga a processzor. Az erőforrások közvetlen fizikai kezelése általában magától értetődő. Az operációs rendszernek azonban olyan magasabb szintű logikai műveleteket is nyújtania kell, amelyek az elképzelhető alkalmazások széles körét kielégítik. A processzor kezelése megfelelő processz ütemezést igényel, a központi tár kezeléséhez megfelelő tárfoglalási algoritmusra van szükség. Ez a két feladat nem független egymástól (erről később részletesen szólunk). A perifériák magasszintű kezelése mindenekelőtt egy megfelelő file rendszer kialakítását jelenti. A file rendszer műveleteinek elég általánosnak kell lenniük ahhoz, hogy különböző készülékekre is implementálhatók legyenek (például a saját lemezre és a hálózaton keresztül elérhető távoli lemezekre is). További logikai kezelést kell biztosítani az interaktív készülékekre, mindenekelőtt egy egységes ablak (window) és menűkezelést, amelyre az összes magasabb szintű program épülhet. Az interaktív jellegű műveleteknél különösen fontos az egységes kezelés az egész rendszerre vonatkozóan.

A fenti irányelveken túlmenően az operációs rendszerre vonatkozóan nagy a szabadság, ez éppen a kutatás egyik fő témája lesz a jövőben.

A fejlesztés első fázisában a LILIPUTH operációs rendszerére a fentieken túlmenően további előírások is érvényesek. Az operációs rendszer eszközt ad arra, hogy (tetszőleges számú modulból álló) "programokat" hozzunk létre, amelyek a központi tárba tölthetők és egyetlen "szuper-eljárás"-ként elindíthatók. Ez a mechanizmus overlay struktúra megvalósítására is alkalmas. Ez a programtöltési technika stack-szerűen osztja a központi tárat a betöltött programoknak. Az operációs rendszer egy igényes processz ütemezővel támogatja a processzor kiosztását a processzek között. Ennél magasabb szinten nem kezel párhuzamosságot, nem támogatja például standard módon programok egyidejű futtatását, noha ilyen programok készítését nem zárja ki. A file rendszer iránti fő követelmény, hogy eszközt adjon a file-ok nyilvántartására (könyvtáros szolgáltatásokkal) és az egyedi file-ok tartalmának elérésére, mindkettőt minél egyszerűbb és hatékonyabb módon.

A fejlesztés további szakaszaiban az operációs rendszer tetszőlegesen módosulhat a

kutatás állásának megfelelően, de azt reméljük, hogy sokféle később felmerülő igény kielégíthető lesz úgy, hogy újabb operációs rendszer szintű szolgáltatások épülnek a meglévőkre.

1.4 A LILIPUTH GÉP VÁRHATÓ ALKALMAZÁSAI

Előljáróban hadd irjuk le azt a közhelyet, hogy egy programozott számítógép alapvetően univerzális gép, ez a lényege. Ezért, noha nagyon fontos, hogy elképzeljük a várható LILIPUTH alkalmazásokat, azért azt is reméljük, hogy olyan alkalmazások is születnek majd a gépre, amelyekről e pillanatban sejtelmünk sincs.

A LILIPUTH mindenesetre olyan gépkategóriát képvisel, olyan különleges sajátosságokkal, amely pillanatnyilag nem létezik a magyar piacon, és várhatólag egy ideig nem is fog. Ez a kategória ma a legfejlettebb technológia világába tartozik, ahol persze egyáltalán nem mellékes a kidolgozás minősége. Ezért nagyon fontos, hogy a fejlesztés első szakaszát, ahol az alapvető funkciókat megvalósítjuk, kövesse egy második, ahol a kidolgozás minősége kerülhet előtérbe.

Jelenleg a következő fő tevékenységeket kívánjuk támogatni a LILIPUTH-tal:

- programozói,
- szövegfeldolgozói,
- adatfeldolgozói,
- mérnöki, irodai munkahely.

Ez a felsorolás meglehetősen eklektikus, ezért kicsit részletesebben kifejtjük az egyes pontokat.

1.4.1 Programozói tevékenység

A LILIPUTH kitüntetett tulajdonsága a jó programozhatóság; az egységes, hatékony MODULA környezet révén. A LILIPUTH különösen alkalmas RENDSZERPROGRAMOZÓI munkára, lehetőségünk van operációs rendszerek, készülékvezérlők, processzütemezők stb. gyors kifejlesztésére. Ez nagyon jelentős lehet oktatási szempontból is. Programozóként ezért elsősorban rendszerprogramozókra számítunk, kutatóintézetek, egyetemek stb. részéről. Ez nem jelenti azt, hogy egyéb feladatok elvégzésére nem ajánljuk a LILIPUTH-ot, csak azt, hogy ezen a területen különösen előnyös tulajdonságokat mutat (egyéb programozói tevékenységekre még utalunk a további pontokban).

1.4.2 Szövegfeldolgozói tevékenység

A LILIPUTH-hoz tervezett pont-raszteres képernyő és az egér alkalmassá teszi az általánosnál jobb szövegszerkesztésre. Ezen a területen Magyarországon óriási hiány van, an-

nál is inkább, mert a meglévő rendszerek általában nem támogatják a magyar ABC-t. Itt többről van szó, mint a magyar írásjelek támogatásáról: a LILIPUTH magasszintű szövegszerkesztést kíván lehetővé tenni, változtatható betűtípussal, on-line tördeléssel, ábrák, különleges jelek rajzolásával. A szövegszerkesztésen túlmenően egy további lépést kell tenni egy általános szövegfeldolgozó rendszer felé, amely lehetővé teszi, hogy a szövegeket egy adatbázisban tároljuk és különböző műveleteket végezzünk rajtuk (pl. keresések). Ez az igény már átvezet a következő tevékenységi körbe, az adatfeldolgozáshoz.

1.4.3 Adatfeldolgozási tevékenység

A LILIPUTH gép elég nagy teljesítményű ahhoz, hogy nagyobb adatmennyiség tárolását és visszakeresését is képes legyen elvégezni. Erre vonatkozóan kedvező tapasztalatok állnak rendelkezésre a LILITH-en, a LIDAS [Zehn83] relációs adatbázis kapcsán. Az adatbázis felhasználása nagyon sok további alkalmazás felé nyitja meg a lehetőséget. Az egyik, talán legkézenfekvőbb, hogy egy interaktív program segítségével egyszerű, közvetlen lekérdezéseket tegyen lehetővé nem számítástechnikai szakemberek számára is. Erre a raszter képernyő és az egér ergonómiai szempontból is különösen alkalmas. Még jelentősebb talán azonban a közvetett felhasználások köre. Ezt támogatja a MODULA/R nyelv [Zehn83], amely a MODULA-2 kiterjesztése a relációs adatbázis kezelés irányában (hasonlóan a PASCAL/R-hez). A programozó definiálhat különböző relációkat és hozzájuk tartozó műveleteket. Létrehozhat olyan adatokat, amelyek túlélik a program futását. A MODULA/R fordító a MODULA fordítóra épül, és így a MODULA/R programok futtatása is élvezi a gépi architektúra által nyújtott előnyöket. Az adatbázis közvetett felhasználására épülhet egy szövegfeldolgozó rendszer, amely a magasfokú szövegszerkesztésen (tehát a formátum manipulálásán) túlmenően, a szövegek tartalmára vonatkozó műveleteket is segíti.

1.4.4 Mérnöki, irodai munkahely

A LILIPUTH rendszer alapvető sajátságai, a jó programozhatóság, a fejlett interaktivitás, a szöveg- és adatfeldolgozás támogatása számos alkalmazás előtt nyitja meg az utat. E pillanatban jól látszik két olyan irány, amely Magyarországon különösen rászorul a fejlődésre: a mérnöki és az irodai munkahely. Ilyen (esetleg kulcsrakész) rendszerek készítése e pillanatban csak távlati elképzelés, noha a LILIPUTH már az előző három pontban felsorolt képességekkel is alkalmas a mérnöki és az irodai munka támogatására, valószínűleg jobban, mint bármelyik e pillanatban Magyarországon kapható rendszer.

2. IRODALMI ÁTTEKINTÉS A LILIPUTH-HOZ HASONLÓ GÉPEKRŐL

Ez a fejezet kötetlen stílusban, a teljesség igénye nélkül, összefoglalja néhány olyan nagyteljesítményű személyi számítógép paramétereit, amelyek ugyanahhoz az irányzathoz tartoznak, mint a LILIPUTH gép. Ezzel az a célunk, hogy éreztessük ennek az egész irányzatnak a jellegét, és a LILITH, illetve a LILIPUTH helyét ezen belül.

A mikroelektronika robbanásszerű fejlődése – amely napjainkban is folytatódik – átalakítja a számítástechnikát. Megdőlni látszik a Gross törvény, amely szerint a számítógépek ára teljesítményük négyzetével arányos. Naponta jelennek meg a korábbiaknál nagyobb teljesítményű rendszerek – alacsonyabb árakon. Ami egy évtizeddel ezelőtt egy géptermet megtöltött, ma befér egy irodába. Egyre "okosabb" és nagyobb teljesítményű chipek kaphatók, amelyek "fejből" tudnak olyan dolgokat, amelyek korábban tekintélyes részét képezték a HW/SW-nek. A software egy része beépült a hardware-be, és ez a tendencia még folytatódik. A MAINFRAME és MEGAMINI kategóriák mellett megjelent a PERSONAL COMPUTER, amely ma igen széles ár és teljesítmény kategóriát képvisel.

A LILIPUTH gép olyan kategóriát céloz meg, amellyel a szocialista országokban (tudtunkkal) még nem foglalkoznak, számunkra nem beszerezhető (többszörösen embargós), ára kb. 10.000 US$ körül mozog. Ilyen nagyteljesítményű, egyszemélyi számítógépek kutatása az USA-ban a hetvenes évek közepétől jelentős, a piacon 1980-tól kaphatók. A kutatásban úttörő szerepet töltött be a XEROX cég PALO ALTO-i kutatóközpontja a maga idejében olyan újszerű rendszertechnikai elvek alkalmazásával, mint például a WINDOW-grafika, a magasszintű, géparchitektúra által támogatott, korszerű programnyelv [Alto79, Inga76, Mitc79] stb..(A "professzionális", "interaktív" stb. jelzőket szándékosan kerüljük, mivel ezeket Magyarországon (és másutt is) már elhasználták 8 bites, CP/M-es rendszerekre.) Nem soroljuk ebbe a kategóriába az APPLE LISA és MACINTOSH gépeit sem, mivel jellemzőik alulmaradnak az alábbiakban részletezett követelményeknek, viszont tárgyaljuk a XEROX DORADO, és a SYMBOLICS 3600 gépeket, amelyekről ma Magyarországon csak álmodozni lehet.

2.1 KÖVETELMÉNYEK

A nagyteljesítményű számítógépekkel szembeni elvárásokat a CARNEGIE-MELLON Egyetem Számítógéptudományi Intézete foglalta össze egy tanulmányában [SPICE79]. A tanulmányt a nagy számítógépgyártók ösztönzésére szánták, hogy kifejlesszenek egy rendszert, amely eleget tesz a tanulmányban vázolt követelményeknek, kereskedelmi forgalmazású és ára elfogadható (10.000 US$ körül). Ez a – fiktív – gép a SPICE (Scientific

Personal Integrated Computing Environment). A tanulmány a nyolcvanas évek közepére teszi a SPICE gép megszületésének várható dátumát. A tanulmány közöl egy csokrot azon alkalmazásokból is, amikre a SPICE-t kívánják felhasználni.

2.1.1 Hardware specifikáció

- CPU
  - 1.000.000 makroutasítás/sec (1MIPS);
  - írható mikrotár, minimum 16K mikroutasítás;
  - virtuális címzés, 2↑30...2↑32 címtartomány javasolt.

- MEMÓRIA
  - 1 Mbyte, minimum.

- HÁTTÉRTÁR
  - 100 Mbyte, lapozáshoz is legyen megfelelő.

- KÉPERNYŐ
  - Színes, bit-map, 1000x1000x4 képelem, jóminőségű billentyűzettel és mouse-zal. A színes képernyő opció, az alap a fekete-fehér változat, minimum 600x800x1 képelemmel, 60 Hz frissítéssel (esetleg gradációval). A bit-map (frame-buffer) legyen direkt címezhető.

- EGYÉB
  - nagysebességű lokális hálózati csatoló (10 Mbit/sec), standard protokoll);
  - lehetőség képbeviteli eszköz (TV kamera) csatlakoztatására;
  - hang I/O lehetőség, 8 bites felbontással, 8...20 000 minta/sec;
  - külső busz, kiegészítő eszközök csatlakoztatására. (RS232, IEEE-488, soros és/vagy párhuzamos).

A szerzők lényegesnek tartják a felhasználói mikroprogramozhatóságot. Igy lehetőség nyílik más virtuális gépeket emuláció útján kialakítani, pl. LISP vagy PASCAL gépeket. Hatékonyan lehet a gépen megvalósítani számos mikroprocesszor utasításkészletét. A programok és az operációs rendszer (pl. KERNEL) használhatnak mikrorutinokat a hatékonyság növelése érdekében. Lehetőség van a különböző programnyelvekhez más-más, az egyes nyelv fordítójához közel álló utasításkészlet definiálására. A szerzők úgy gondolják, hogy a c:= a+b "tipikus" PASCAL utasítás 2 utasításban végrehajtható, melynek ideje 2 mikrosec vagy kisebb.

A címtartomány legyen nagy. A tanulmány szerint a számítógépek címtartománya két évenként egy bittel nő. Jelenleg a 16..18 bites címtartományt már az alkalmazások kinőtték. A 24 bites tartomány is csak pillanatnyi megoldás, tekintettel a jövendő alkalmazásokra (pl. hang- és képfeldolgozás stb.).

2.1.2 Software specifikáció

A SPICE három fő alkalmazási területet szándékozik lefedni:

- tudományos (műszaki) programok;
- dokumentáció készítése;
- kommunikáció.

A SPICE magasszintű nyelven programozható. Egy esélyes jelölt az ADA. A programozási környezet a választott nyelv köré épül, beleértve a nyelv-orientált szövegszerkesztőt, fordítót, linkert, szimbólikus debuggert és programkönyvtárat. Nagyon fontos, hogy az egész software (beleértve az operációs rendszert is) azonos elvek szerint épüljön fel (pl. hibajelzések, modul-specifikációs és dokumentációs szabványok stb.).

Kommunikációs protokollként a SPICE valószinűleg a XEROX PUP-ot és az ARPA INTERNET/TCP-t fogja támogatni. Ezzel a választással csatlakozási lehetőség van az ARPA-ra, és az ARPA-n keresztül a többi említésre érdemes hálózatra.

Hálózati szolgáltatásként FILE-SERVER és PRINTER-SERVER funkciókat nyújt a SPICE. A PRINTER-SERVER dokumentum minőséget (xerografikus printer) és nyomdai minőséget (fényszedőgép) is szolgáltat. A FILE-SERVER tárolja az osztott (mindenkinek, vagy adott hozzáférési joggal rendelkező csoport számára) dokumentációkat, programokat, információkat stb..

A tanulmány szerint a szerzők több hasonló rendszer tanulmányozása után a legtöbb vonatkozásban tisztán látnak, de még nem tudják, hogy milyen jellegű programok és emberek fogják ezt használni, és mekkora teljesítmény szükséges.

2.1.3 Néhány tervezett felhasználás

- Dokumentáció készítés. On-line szövegszerkesztés, dokumentációs és ábrakönyvtár használattal. A rendszer ellenőrizhet néhány egyszerű nyelvtani szabályt is. Az ernyőn rögtön a tördelt, ábrákkal ellátott szöveg látható, vagy tovább módosítható. A kész dokumentáció azonnal közzétehető (FILE-SERVER), és/vagy kinyomtatható.
- Programfejlesztés. Programnyelv orientált editor, mely ismeri a programnyelv szintaxisát. Segíti a kulcsszavak gépelését, jelzi a zárójelezés vagy a struktúrák kiegyenlítetlenségét, tabulálást végez a struktúra mélysége szerint. Hiba esetén a hibás struktúrát az ernyőn valamilyen fajta kiemeléssel (pl. szinnel) jelzi. A programozónak algoritmus, program, szubrutin és specifikációs könyvtárak állnak rendelkezésére. Amint egy modul elkészül, továbbítható a "project management" rendszerhez, amelyik koordinálja a verziókat, specifikációkat, és naprakész információkat szolgáltat a projekt állásáról.
- Programbelövés. Ahogy a fejlesztő lépésenként hajtja végre a programját, a forrásprogram aktuális utasításrészlete kiemelve jelenik meg az ernyőn. Az aktuális eljárás-stack, a hívási paraméterek és különböző változók megfelelő WINDOW-kban vizsgálhatók. Amint kiderült a hiba oka, azonnal javítható a forrásprogramban. Ha a hiba miatt a modul interface-eit módosítani kell, a rendszer tájékoztatást nyújt a javítás által inkonzisztenssé

vált modulokról.

• VLSI tervezés. A tervező egy új VLSI chipen dolgozik. A felépítést a színes grafikus képernyő, a MOUSE és a MENÜ rendszer segítségével megtervezi, az adott konstrukció monitor programjának felügyelete alatt. Ez a felügyelő program egészében futhat a tervező saját gépén, vagy részei futhatnak a hálózat más, éppen nem foglalt gépein. A tervezőnek az adatbázisokban rendelkezésére állnak a különböző katalóguselemek, amelyeket le tud hívni, és be tud építeni konstrukciójába. Eközben a felügyelőprogram az adatbázisokból lehívott elemek katalógusadataival finomítja a konstrukció paramétereit. A munka befejeztével a tervező még egy utolsó szimulációs vizsgálatot végez, majd a tervet elküldi az automatikus maszk készítő rendszerhez. Nagyon hasonló forgatókönyv írható pl. híd, épület, repülőgép vagy hasonlók tervezésére.

• Kommunikációs szolgáltatások. Egy nagy projektben rendszerint többen vesznek részt. A munkatársak különböző épületekben dolgoznak a személyükre lebontott feladatokon, a saját gépükön. Időről-időre konzultációkat kell tartani a felmerült problémákról. Bármelyik munkatárs kezdeményezheti egy vagy több társ hívását a hálózaton keresztül. A hívottak elfogadhatják vagy elutasíthatják a hívást. A telekonferencia az akusztikus I/O eszközök segítségével zajlik, a hálózaton keresztüli továbbítás kompresszáltan, digitális úton történik (VOICE SWITCHING). A munkatárs a beszélgetés során a számára fontos részleteket megörökítheti a saját "memo" file-jában. A konferencia végeztével akár az egész szöveg elküldhető egy, az adott témához tartozó levelezési lista alapján az érintettek részére.

Olyan szolgáltatásra is igény van, hogy a munkatárs (pl. a professzor) megadhassa azon témák vagy személyek listáját, akik egy fontos, sürgős munka közben is "zavarhatják" (pl. a titkárnő, ha elkészült a kávé). Az összes többi hívás tárolódik, és később visszakereshető.

Mindez persze megvalósítható egy titkárnővel is, de a jó titkárnő ritka, drága és rendszerint csak a hivatali időben található meg.

• Telefonhívások kezelése. A munkatárs távol van munkahelyétől, de fontos hívást vár. Ilyenkor a SPICE gép fogadja a bejövő hívást, közli a hívóval, hogy a hívott távol van, és kéri, adja meg a nevét és a tartózkodási helyét. Egy beszédfelismerő program analizálja a választ, és a hívót felveszi a visszahívandók listájára.

• Információ szolgáltatás. Új munkatárs érkezett az intézetbe. Amikor először jelentkezik be a rendszerbe, a rendszer felkéri az újonnan érkezettet életrajzi adatok szolgáltatására, beleértve, hogy az egyik, TV kamerával felszerelt SPICE-on keresztül arcképet is rögzítik. A következő napon a rendszer bemutatja az új munkatársat a többieknek, és arcképét is kirajzolja az ernyőre. Ettől a naptól kezdve bárki érdeklődhet pl. John Smith után, az adatbázisból lehívható az arcképe és önéletrajzi adatai.

2.1.4 Következtetések

Úgy tűnik, a pályázat kiírói leginkább a hardware követelményeket tisztázták, kevésbé részletezettek elképzeléseik a rendszer software oldaláról. Ez részben érthető, mivel tanulmányuk a számítógépgyártók részére készült. Elképzeléseik a gép felhasználhatóságá-

ról grandiózusak. Érdemes megjegyezni, hogy a fejlesztés idejét 5 évre becsülik, mialatt több mint 100, magasan kvalifikált ember-évnyi munkát biztosítanak.

Lényeges követelmények:

- 1 MIPS, min. 1 MBYTE, virtuális címzés;
- felhasználói mikroprogramozhatóság;
- nagyfelbontású raszter képernyő;
- nagy (és saját) háttértár (tehát ne legyen "vízfejű" gép, mint például sok minifloppy-s rendszer);
- nagysebességű hálózati csatlakozás;
- grafikus kimenet (PRINTER-SERVER – laserprinter);
- magasszintű nyelv és fejlett programozási környezet;
- egységes szemléletű és nyílt (bővíthető) rendszer;
- fejlett kommunikációs szolgáltatások.

2.2 NÉHÁNY NAGYTELJESITMÉNYŰ SZEMÁLYI SZÁMITÓGÉP

Ebben az alfejezetben az

- ALTO (XEROX – USA);
- DORADO (XEROX – USA);
- PERO (ICL – ANGLIA);
- DOMAIN (APOLLO COMPUTER – USA);
- SUN–2 (SUN MICROSYSTEM – USA);
- SYMBOLICS 3600 (SYMBOLICS – USA)

nagyteljesítményű személyi számitógépek vázlatos leírása található. A leírások mélysége erősen változó, a rendelkezésre álló dokumentációk függvényében, és elsősorban technikai jellegű.

2.2.1 ALTO

A XEROX cég Palo Alto-i kutatóközpontja (PARC) úttörő munkát végzett többek közt a nagyteljesítményű személyi számítógépek (ALTO, DORADO), a nagysebességű lokális hálózatok (ETHERNET), a magasszintű nyelvek (MESA, SMALLTALK), a WINDOW-grafika, az iroda-automatizálás terén.

A legenda szerint az ALTO fejlesztés úgy kezdődött, hogy néhány kutató a PARC-ban azzal a kéréssel fordult főnökükhöz (aki egyébként pszichológus), hogy vegyenek egy DEC SYSTEM 10-et, mire a lakonikus válasz az volt, hogy "a XEROX nem veszi, hanem gyártja a számítógépet".

Az első ALTO 1973-ban készült el. Az ALTO mikroprogramozott gép, ahol a perifériák vezérlését is részben mikroprogram végzi [ALTO78]. Virtuális cím- és tárkezelése nincs. Az ALTO MESA nyelven programozható [Mitc79], amely valószinűleg nap-

jaink egyik legjobb programozási nyelve. A MESA és néhány korszerű, magasszintű nyelv összevetéséről [Bösz81]-ben olvashatunk.

Az ALTO mikrogép ciklusideje 170 mikrosec. 2K PROM és 3K RAM mikrotára van, megszakítási rendszere 15 csatornás, egyszintű, vektoros. Memóriája 256K szóig bővíthető, hibajavítással rendelkezik. A display állított, 8.5x11" méretű, 606x808 képpontot tartalmaz, 60Hz-es frissítéssel (interlace-es). Ethernet csatolója 3Mbps sebességű. Az ALTO-hoz különféle diszkeket csatoltak, ezek kapacitása 5Mb körüli.

Az ALTO tulajdonképpen nem tesz eleget a SPICE követelményeknek, de vegyük figyelembe korai megjelenését, hogy mennyi újdonságot hozott, valamint, hogy a későbbiekben mindenki az ALTO-t másolta több-kevesebb módosítással.

2.2.2 DORADO

A DORADO gépet 1976-77-ben tervezték az ALTO kiváltására [Dora81]. 1976-ra nyílvánvaló lett, hogy szükség van egy sokkal nagyobb teljesítményű berendezésre a továbblépéshez. (Ekkorra már több száz, egymással összekötött ALTO működött a PARC-ban). Az alkalmazások nemcsak az ALTO sebességét, de háttértár és főleg memória kapacitását nőtték ki. A prototípus 78-ra, az újratervezett DORADO 79-re készült el. A DORADO szintén mikroprogramozott, és a perifériákat is mikroprogram kezeli.

A DORADO külön utasítás előkészítő egységgel (IFU), CACHE tárral és multiport memóriával rendelkezik. A gép igen gyors, mikrociklus ideje 60nsec. A DORADO 16 mikrotaskkal rendelkezik, melyek 5, a hardware által ütemezett prioritási szinten futnak. A legalacsonyabb prioritású szintű mikrotask az utasításkészlet emulátor. A gépnek van hardware EXPRESSION stack-je is. CACHE tára 8..32Kbyte, memóriája 512K..16Mbyte között bővíthető. Virtuális címképzéssel rendelkezik. (Gondoljunk bele: EZ IS PERSONAL COMPUTER!)

A tár 64K dinamikus cipekből épül fel. 1 hibát javítanak. A gép két I/O busszal rendelkezik. A lassú busz 256 Mbit/s, a gyors 530 Mbit/s sebességű. A gyors buszra csak a display csatlakozik, amelyről annyit tudunk, hogy 1.7 mikrosec ciklusidővel 256 bitet olvas.

Az egész gép kb. 3000 MSI integráltságú, nagyobbrészt ECL áramkörökből áll. A teljes processzor mérete 0.14 köbméter, teljesítményfelvétele (beleértve a 80Mbyte-os cserélhető diszket is) kb. 2.5KW.

A DORADO-n négyféle emulátort használnak:

- MESA [Mitc79, John82, McDa82];
- SMALLTALK [Inga76];
- BCPL;
- INTERLISP.

A DORADO architektúrája a MESA nyelvhez illeszkedik. Jellemző, hogy egy tipikus SMALLTALK utasítás végrehajtása 30-40 ciklus, ezzel szemben vannak olyan MESA utasítások, amelyek végrehajtása 1 mikrociklust igényel, és a függvényeljárás hívás,

valamint a blokk-transfer kivételével egyik sem igényel 6 ciklusnál többet.

Manapság a PARC-ban több mint ezer ALTO és DORADO gép működik ETHERNET-en [Ethe80] összekötve, ahol csupán NAME-SERVER-ből 6 van! A nagyteljesítményű gépek gyors hálózaton történő kommunikációjával megteremtődött az alap az elosztott rendszerek (DISTRIBUTED SYSTEM) létrehozására. A PARC-ban az ehhez javasolt teljesen új primitiv a távoli eljárás hivás (REMOTE PROCEDURE CALL – RPC). Az RPC interface mechanizmusa talán egy külső modulból történő eljárás hívásához hasonlitható, azzal a különbséggel, hogy a "külső modul" egy másik gépben található. Igen érdekes riport olvasható az RPC megvalósításáról és problémáiról [Nels81]-ben.

2.2.3 PERQ

A PERQ [PERQ82] születése 1979-re tehető. A gép 1Mips sebességű, mikroprogramozott, irható mikrotárral. A mikrotár 4K, 48 bit széles. A CPU az AMD2910 sequencer-en alapul, 20 bites. Hardware stackje 16 szintű, emellett még egy kétportos, 256x20 bites általános celú regiszterkészletet is tartalmaz.

Memóriája multiportos, 0.5..4Mbyte között bővithető. A képernyő 1024x768 képpontot tartalmaz, mérete 210x275 mm, állitott, 50 Hz-es frissitéssel (nincs interlace). A memória hozzáférés a képernyő és a CPU között igy kb. 50-50%-ban oszlik meg. Érdekes, hogy a konstruktőrök ebbe beletörődtek, ezt talán az magyarázza, hogy a processzor még igy is "elég gyors", valamint az utasitás lehivó 8 byte-os "pipeline" regiszterrel rendelkezik.

A PERQ mikrokódja a PASCAL Q-code-ot emulálja. A Q-code tulajdonképpen a PASCAL P-code kibővitése néhány plusz művelettel, mint például a képernyőt kezelő raszter műveletek. A gépen a PERQ-OS és PNX operációs rendszerek futnak. A PNX a UNIX helyi változata, a PNX támogatásához a PERQ-ben a C nyelvet támogató emulátor fut.

A PERQ 24Mbyte Winchester diszkkel, valamint 10Mbit/s sebességű ETHERNET típusú lokális hálózati csatolóval van ellátva. Ezenkívül van MOUSE-a, fényceruzája, floppy diszkje, valamint RS232C és GPIB (IEEE-488) interface-e.

Az egész gép fogyasztása mindössze 740 W.

2.2.4 DOMAIN

A DOMAIN (Distributed Operating Multi-Access Interactive Network) rendszert az Apollo cég forgalmazza. A DOMAIN processzora 68010-es memory management egységgel. Reális tára 0.5 .. 3.5 MByte közötti kapacitással rendelkezik, virtuális memóriája 16 MByte-ig terjed. 33/66 Mbyte Winchester diszke és 12 Mbit/s sebességű, RING típusú lokális hálózati csatolója van. Képernyőből állitott (portrait) és fektetett (landscape) típust egyaránt szállitanak, melyek felbontása 1024x800 (60Hz). A képernyő bitmap me-

memóriája le van választva a főtárról, amelyet külön egység (RasterOp) kezel (állítólag egy 10MHz-es 68010). A DOMAIN munkahelyek nem szükségképpen rendelkeznek saját diszkkel, hálózaton keresztüli lapozás (DEMAND PAGING) biztosított (FILE-SERVER). A lapméret 1024 byte. Egyéb perifériák csatolása MULTIBUS-on keresztül lehetséges. Egy DOMAIN munkahely ára $ 10.000.

A DOMAIN rendszer-software sok érdekességet tartalmaz. Operációs rendszere (AEGIS) egyidőben megengedi több taszk futtatását. Minden egyes objektumnak egyedi (96 bit) azonosítója van, amelyből 62 bit specifikus azonosító, 32 bit relatív cím. Rendelkezik "SHELL" és levelesláda (MAIL BOX) szolgáltatással, támogat interprocessz kommunikációt stb.

A DOMAIN software rendszerét – bizonyos utalások alapján ezt gondoljuk – PASCAL-ban implementálták. Ezenkívül még C és (természetesen a kiirthatatlan) FORTRAN áll rendelkezésre.

2.2.5 SUN-2

A SUN sokban hasonlít a DOMAIN rendszerre. Processzora 68010, van memory management egysége, reális tára 4 Mbyte-ig bővíthető, paritásbittel ellátott. Különböző 34..380 Mbyte-os Winchester és SMD diszkek csatolhatók a géphez MULTIBUS-on keresztül. BACKUP célra 20 MByte kazettás és 45 MByte szalagos egység is kapható. Képernyője 1152x900, 70 Hz, nem interlace-es, mérete 19", fektetett. A képernyőnek külön multiport bitmap tára van, VLSI RasterOp támogatással. Opcióként 640x480x8-as interlace-es, 60Hz-es színes display, és lebegőpontos processzor is vásárolható. Lokális hálózata ETHERNET (10 Mbit/s) típusú. Ezenkívül van két RS-423 soros vonala (az UUCP net számára).

A SUN-on UNIX 4.2 (Berkeley) fut.

2.2.6 SYMBOLICS 3600 – a LISP-gép

A 3600-as a "state-of-the-art" az egyfelhasználós számítógépek között [SYMB83]. Ennek a gépnek az ősét az M.I.T. Mesterséges Intelligencia Laboratóriuma tervezte (1974). A software fejlesztés 1975-től folyik. Az első generációs LISP-gép (a CONS) 1976-ra készült el. 1978-ra elkészült a gép továbbfejlesztett változata, a CADR. Ez volt a LISP-gépek második generációja. 1980-ban már a Symbolics cég a legutóbbi technológiákat alkalmazva újratervezte a gépet, amely alapvetően még a M.I.T. CADR-on alapult. Ez – az LM-2 – 1981-re készült el (harmadik generáció). Az 1979-1982 között új fejlesztések egy sokkal nagyobb teljesítményű, árban kedvező gépet eredményeztek. Ez a LISP-gépek negyedik generációja, a 3600-as, amely új hardware, de software kompatibilis az LM-2-vel.

A 3600-as egész software-je – hozzávetőlegesen félmillió forrássor – a LISP nyelv

3600-as "nyelvjárásában" – ZETALISP-ben készült. A 3600 a MAINFRAME-ek teljesítményét kínálja a felhasználónak. A processzora 36 bites, amiből 32 bit adat, 4 bit pedig az un. "TAG" mező, a futásközbeni típusellenőrzésre. (Valójában egy szó 44 bites, amiből 7 bit a hibajavítás (ECC), 1 bit tartalék). Rendelkezik virtuális tárkezeléssel (DEMAND PAGING típusú). Virtuális címtartománya 256 Mszó (= 1 GByte-28 bit), fizikai címtartománya 16 Mszó (24 bit). A gépbe fizikailag minimum 1, maximum 30 Mbyte tár helyezhető el. A memória 64 Kbites dinamikus chipekből (200 nsec) áll. Véletlenszerű címzésnél egy memória művelet 600, szekvenciális elérésnél 200 nsec.

A CPU ciklusideje 180..250 nsec (változó). Mikrotára 8K, CACHE tárába 2 K utasítás fér. A gépnek két hardware stackje van, melyeknek "felső" része szintén a CACHE tárban helyezkedik el. A LISP programok dinamikusan allokálnak tárat. Amikor a tár feldarabolódása egy bizonyos határt elér, akkor azt tömöríteni kell. Erre a feladatra a 3600-ban hardware-rel támogatott GARBAGE-COLLECTOR mechanizmust alkalmaztak.

A CPU-n kívül még két processzora van (CONSOLE processzor, Front-End Processzor), amelyek a CPU-val párhuzamosan dolgoznak. A FEP csatolja a laserprintert, a soros vonalakat, valamint ide csatlakozik a MULTIBUS vezérlő. A MULTIBUS-ra további perifériák csatolhatók, mint például mágnesszalag stb.

A FEP segítségével történik a gép boot-olása és tesztelése is. A 3600 diszkjén a FEP részére külön terület van biztosítva, ahová a hibajelentéseket, karbantartási és nyomkövetési információkat teszi le. A FEP processzora 68000, amelynek külön 128K RAM és 64K ROM tára van. A FEP boot-olása egy "NanoFEP"-nek nevezett, INTEL 8749-es mikroprocesszorral történik (hiába, hierarchikus rendszertervezés).

A 3600 konzola a képernyőn, a mouse-on és a klaviatúrán felül még tartalmaz akusztikus perifériát is. A konzol soros buszon keresztül csatlakozik a 3600-hez. 60 m-re eltávolítható a géptől.

Noha a konzol tartalmaz egy 68000-es processzort is, ez csak a billentyűzetet és a mouse-t kezeli. A képernyő és a hang kimenet kezelését (mikroprogram támogatással) maga a 3600 végzi.

A fekete-fehér képernyő 1150x900 képelemet tartalmaz, 60 Hz frissitésű, nem interlace-es. Opcionálisan csatolható színes display (pluszként), amely 1024x1024 8 bites képelemet tartalmaz.

A 3600 két 16 bites hangcsatornával rendelkezik (sztereo), a mintavételi frekvencia 50 KHz. A DA konvertert (12 bit), valamint a Manchester decodert és a hozzá tartozó áramköröket a konzolban helyezték el. A 3600-nak a hang I/O standard tartozéka.

A 3600 nem "vízfejű" gép, háttértár kapacitása arányos egyéb jellemzőivel. A géphez alapkiépítésben egy 169 MByte-os Winchester diszk tartozik. Opcióként rendelhető 474 MByte-os Winchester, amely befér az előző helyére. A diszk kontroller négy ilyen egységet képes illeszteni, a processzorba két diszk kontroller kártya helyezhető be (2x4x474 MByte = 3.8 GByte).

Ezen kívül van 300 Mbyte kapacitású cserélhető diszk (SMD) is. A konzolra csatlakozik a MOUSE, a klaviatúra, valamint a hang I/O.

A 3600-at 10 Mbit/s sebességű ETHERNET tipusú lokális csatolóval látták el.

Backup célra MULTIBUS-on keresztül csatolt kazettás és mágnesszalagos egységek használhatók. Laserprintere, melyet szintén a Symbolics cég gyárt, 8.5x11 inches papírra dolgozik, felbontása 240 pont/inch.

A 3600-as Shottky TTL, ECL 10 és 100 K sorozatú alkatrészekre épül. Érdekes, hogy a hátlapon kívül nincs egyéb belső kábelezés. A tápegységek kapcsolóüzeműek, a CPU teljesitményfelvétele 2000 W.

A képernyő, diszk, ETHERNET csatoló és az akusztikus I/O kezelését mikrotaszkok támogatják. A FEP és a konzol processzor software-je LIL-ben (Lisp-like Implementation Language) készült. A gépen a Zetalisp mellett még az Interlisp és FORTRAN fut.

3. A MODULA NYELV

A MODULA nyelvről részletes ismertetés található [Wirt82]-ben és [Szab84]-ban, egyes jellegzetességeinek ismertetése [Bösz81]-ben. Itt csak legfőbb jellegzetességeit ismertetjük, elsősorban azért, mert a nyelv bizonyos konstrukcióinak ismerete nélkül a LILITH és a LILIPUTH hardware és gépi architektúra lényege nem jól érthető. A MODULA sok tekintetben a PASCAL [Jens78] utódjának tekinthető, ugyanúgy jellemző rá a struktúrált programozás elveit szem előtt tartó adat- és vezérlési-struktúrák gazdag választéka. A típus koncepció is hasonló a PASCAL-éhoz, de a típusazonosságra vonatkozóan a MODULA kiküszöbölte a PASCAL-ban meglévő inkonzisztenciát. Több egyéb hiba is megszünt a PASCAL-hoz képest. Az egyetlen lényegesebb hiányzó elem a MODULA-ban a PASCAL-hoz képest a FILE típus és a rajta végezhető műveletek. A MODULA nyelv maga nem tartalmaz input/output műveleteket, viszont eszközt ad arra, hogy ilyeneket MODULA-ban létrehozzunk. Ez megfelel a nyelv egyik alapkoncepciójának, ami szerint a programnyelvnek nem az a dolga, hogy megoldja a problémát, hanem, hogy ESZKÖZT adjon a megoldáshoz.

A PASCAL-hoz képest alapvetően a következő új elemekkel bővült a nyelv: a MODUL, a KÜLÖN FORDÍTÁS, a KORUTIN és az EXPLICIT GÉPFÜGGŐSÉG.

A modul lényege, hogy a program egyik részét szintaktikusan elválasztja a többitől, belső objektumait "elrejti" a külvilág elöl. A modul határokon csak az explicit export/import segítségével lehet átjutni. A modul globális változói egészen addig megtartják értéküket, amig a modult tartalmazó eljárás aktív. A 0.szinten elhelyezkedő modul globális változói ezért a program teljes futása alatt megőrzik értéküket. A következő egyszerű kis példa csak izelítőül szolgál a MODULA nyelvhez. A "Stack" nevű modul implementál egy stack elven működő tárat, amelynek csak méretét és a rajta végezhető két műveletet (Push, Pop) exportálja. A stack felhasználói (a példában a "StackUser" nevű modul) kizárólag az exportált műveleteken keresztül fordulhatnak a stack-tárhoz. E példán is jól látható, hogy a modul milyen fontos eszköze a biztonságos programozásnak: egy egyszer jól megírt modult nem rongálhatnak meg egy hibás partner-modul mellékhatásai.

A modul fordítási egység is lehet. Ebben az esetben két részre bomlik, az exportált konstansokat, tipusokat, változókat és az exportált eljárások fejeit tartalmazó DEFINÍCIÓS és az exportált eljárások kifejtését tartalmazó IMPLEMENTÁCIÓS modulra. A külön fordított modulok között a MODULA fordító teljes szintaktikai ellenőrzést végez, ugyanúgy, mintha együtt fordultak volna. A külön fordítás ilyenfajta kezelése kiemelkedően hatékony eszközt ad nagyobb programok részenként való elkészítéséhez. A definiciós modult a programozók specifikációs eszközként is kiválóan használhatják egy-

más között.

```
MODULE Main;

  MODULE Stack;

    EXPORT Push, Pop, StackLength;

    CONST StackLength = 16;

    TYPE StackRange = [0..StackLength-1];
         PointRange = [0..StackLength];

    VAR StackPointer: PointRange;
        StackStore: ARRAY StackRange OF CARDINAL;

    PROCEDURE Push(elem: CARDINAL);
    BEGIN
      StackStore[StackPointer]:= elem;
      INC(StackPointer);
    END Push;

    PROCEDURE Pop(): CARDINAL;
    BEGIN
      DEC(StackPointer);
      RETURN StackStore[StackPointer];
    END Pop;

    BEGIN  StackPointer:= 0;
  END Stack;

  MODULE StackUser;

    IMPORT Push, Pop, StackLength;

    VAR i, elem: CARDINAL;

    BEGIN
      FOR i:= 1 TO StackLength DO
        Push(i)
      END; (*FOR*)
      FOR i:= 1 TO StackLength DO
        elem:= Pop();
      END; (*FOR*)

  END StackUser;

END Main.
```

A következő példa az előbbi modul felbontását mutatja külön fordításhoz.

```
DEFINITION MODULE Stack;

  EXPORT QUALIFIED Push, Pop, StackLength;

  CONST StackLength = 16;

  PROCEDURE Push(elem: CARDINAL);

  PROCEDURE Pop(): CARDINAL;

END Stack.


IMPLEMENTATION MODULE Stack;

  TYPE StackRange = [0..StackLength-1];
       PointRange = [0..StackLength];

  VAR StackPointer: PointRange;
      StackStore: ARRAY StackRange OF CARDINAL;

  PROCEDURE Push(elem: CARDINAL);
  BEGIN
    StackStore[StackPointer]:= elem;
    INC(StackPointer);
  END Push;

  PROCEDURE Pop(): CARDINAL;
  BEGIN
    DEC(StackPointer);
    RETURN StackStore[StackPointer];
  END Pop;

  BEGIN  StackPointer:= 0;
END Stack.


MODULE StackUser;

FROM Stack IMPORT Push, Pop, StackLength;

  VAR i, elem: CARDINAL;

  BEGIN
    FOR i:= 1 TO StackLength DO
      Push(i)
    END; (*FOR*)
    FOR i:= 1 TO StackLength DO
      elem:= Pop();
    END; (*FOR*)

END StackUser.
```

A modulhoz megadható egy prioritás, ami azt jelenti, hogy a modul összes eljárása azon a (hardware) prioritási szinten működik. Ez a lehetőség a megszakítások megfelelő kezelését szolgálja (ld. később).

A MODULA másik újszerű tulajdonsága a korutinok létrehozásának lehetősége. A korutin szekvenciális utasítás sorozat, amely képes más koutinokkal "kvázi-paralel" működni. A kvázi-paralel működés azt jelenti, hogy egyszerre több korutin lehet futáskész állapotban (ezért paralel), de ezek közül csak egyetlen egy fut valójában (ezért kvázi).

(A hétköznapi szóhasználatban gyakori a "processz" elnevezés a korutin helyett; az irodalomban általában a processz a teljes paralel, a korutin a kvázi-paralel működés elfogadott megjelölése [Bösz81].) A korutinok létrehozása a (SYSTEM modulból importált – ld. később) NEWPROCESS nevű eljárással történik, tetszőleges (globális, paraméter nélküli) eljárás elindítható korutinként (akár többször is). A korutin saját adatterülettel rendelkezik (stack + heap). A korutinok egymás között a (szintén a SYSTEM-ből importált) TRANSFER utasítás segítségével kommunikálhatnak. Ez lényegében egy programozott kontextus cserét hajt végre. A következő kis példa bemutatja korutinok létrehozását és kommunikációját. A példában 3 korutin látható, "A" és "B", valamint az indítást végző "main". "A" és "B" felváltva hívogatja egymást. Hangsúlyozzuk, hogy ez a példa csak a nyelvi eszközöket szemlélteti, egy valódi rendszerben ezeket egy megfelelő ütemező készítésére kell felhasználni [Bösz81], nem közvetlenül, mint itt.

```
MODULE Coroutines;

FROM SYSTEM IMPORT NEWPROCESS, PROCESS, TRANSFER, SIZE, ADR;
FROM Terminal IMPORT WriteString, WriteLn;
FROM DecimalIO IMPORT WriteDec;

VAR
  MessageA, MessageB: CARDINAL;
  WorkSpaceA, WorkSpaceB: ARRAY [0..99] OF CARDINAL;
  main, procA, procB: PROCESS;
  ch: CHAR;

PROCEDURE A; (*coroutine*)
BEGIN
  LOOP
    WriteString('Corutine-A:  message from B = ');
    WriteDec(MessageB); WriteLn;
    INC(MessageA);
    TRANSFER(procA, procB);
  END; (*LOOP*)
END A;

PROCEDURE B; (*coroutine*)
BEGIN
  LOOP
    WriteString('Corutine-B:  message from A = ');
    WriteDec(MessageA); WriteLn;
    DEC(MessageB);
    TRANSFER(procB, procA);
  END; (*LOOP*)
END B;

BEGIN (*main coroutine*)
  MessageA:= 0; MessageB:= 65535; (*Initialization*)
  NEWPROCESS(A, ADR(WorkSpaceA), SIZE(WorkSpaceA), procA);
  NEWPROCESS(B, ADR(WorkSpaceB), SIZE(WorkSpaceB), procB);
  TRANSFER(main, procA);
END Coroutines.
```

A MODULA közelítése a párhuzamossághoz azért nagyon érdekes, mert szemben a legtöbb hasonló nyelvben követett megoldással [Bösz81], semmilyen eleve beépített processz- vagy korutin-ütemezőt nem tartalmaz, viszont ESZKÖZT ad annak elkészítésére. Ezért a MODULA rendszerben a szokásosnál sokkal könnyebben készíthetünk ütemezőt.

A MODULA nyelv következő jelentősebb újdonsága a gépfüggőség nyelvi kifejezhetősége. Ennek legfőbb eszköze egy olyan speciális, pszeudo-modul (SYSTEM), amely gépfüggő típusokat és eljárásokat exportál. Ez lehetővé teszi, hogy a fordító implementálói ebbe az egyetlen modulba koncentrálják a gépfüggő sajátságokat, amelyek így szabályozott módon állnak a (rendszer)programozó rendelkezésére. A MODULA ezen kívül eszközt ad arra, hogy olyan gépi utasítás(sorozat)okat is kiadjunk, amelye(ke)t a fordító nem generál. Lehetőség van arra, hogy egy eljárás törzse a szokásos BEGIN helyett a CODE kulcsszóval kezdődjék; az utána következő konstans(ok) változatlanul kerül(nek) be a kódba. Ez az eszköz veszélyes (de bizonyos esetekben nélkülözhetetlen) kiskapu a rendszerprogramozó számára. Az alábbi eljárás a fordító által nem generált "DDT" utasítás kiadását példázza (a paramétereket az eljáráshívás mechanizmusa megfelelő módon adja át):

```
PROCEDURE DisplayDot(m: Mode; bmd: ADDRESS;
                     x,y: CARDINAL);
CODE 342B
END DisplayDot;
```

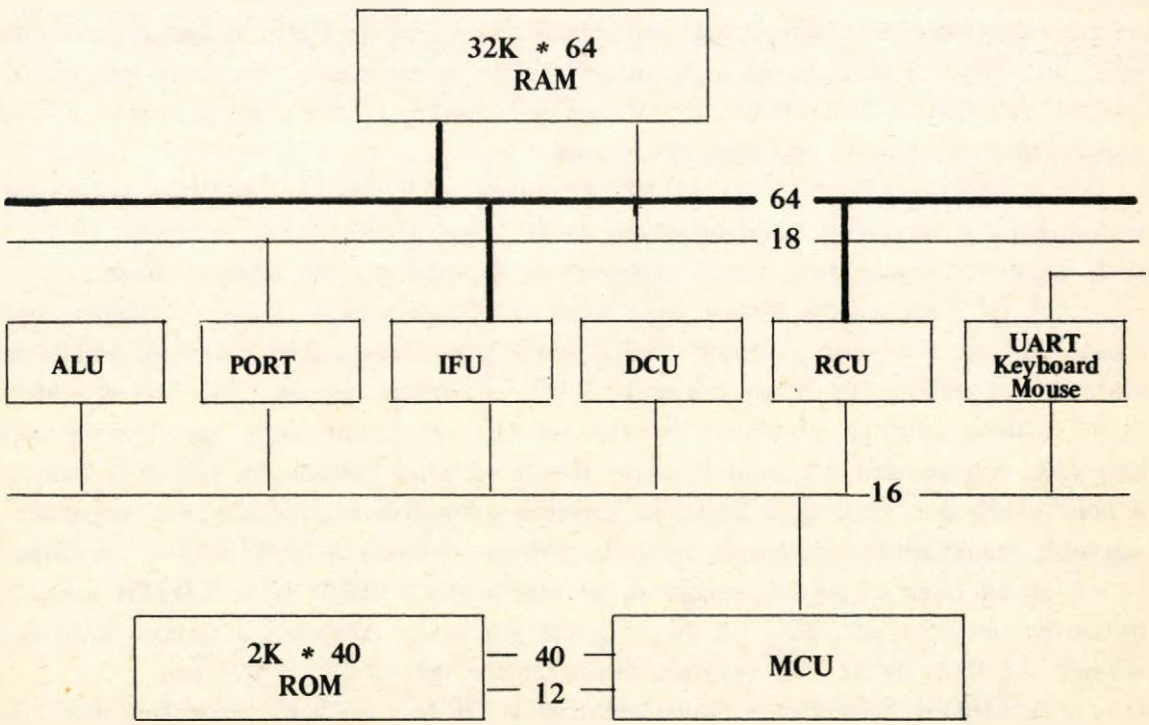
Az eddig ismertetett tulajdonságokból is látszik, hogy a MODULA eszközt ad az input/output legalacsonyabb szintű kezelésére is. Ezért nem tartalmaz a nyelv semmilyen beépített file kezelést (mint pl. a PASCAL), ehelyett az egyes implementációk könyvtári modulok segítségével nyújtanak ilyen szolgáltatásokat, izlés szerint, és a MODULA fordító teljes szintaktikai ellenőrzése alatt. A különfordítás ismertetett elve általában nagyon kedvez könyvtári szolgáltatások létrehozásának, amelyek egy érett implementációban (pl. a LILITH-en) nagyszámban rendelkezésre is állnak (pl. formázott I/O, matematikai könyvtár stb.).

4. A LILITH RENDSZER RÉSZLETES ÁTTEKINTÉSE

A LILITH rendszer egyes elemeit részletesen tárgyalja [Geis83, Jaco82, Knud82, Wirt81, 82], valamint [Bösz83, Szab84]. Az itt következőkben megkisérlünk átfogó képet nyújtani a teljes rendszerről, különös tekintettel azokra a részekre, amelyek a LILIPUTH szempontjából fontosak.

4.1 A LILITH HARDWARE

A LILITH hardware részletes ismertetése [Wirt81]-ben található, most csak főbb elemeit tekintjük át. A tárgyalás csak architekturális kérdésekre terjed ki. A LILITH hardware felépítését a 4.1 ábra szemlélteti.



4.1 ábra
A LILITH hardware felépítése

A hardware fő elemei egy 16 bites busz körül helyezkednek el.

A LILITH egyik alapvető tulajdonsága a mikroprogramozható központi egység, amely lehetővé teszi, hogy az utasítás- és regiszterkészletet szabadon válasszuk meg. Ez lehetővé teszi a MODULA fordító és a raszter képernyő kiszolgálását alacsony szinten is.

A mikroutasítások végrehajtását a mikrovezérlő (MCU) vezérli. A 40 bites mikroutasítások a 12 biten címezhető ROM-ban helyezkednek el, a mikroprogramtár a lehetséges 4K-ból 2-t használ fel. 1K-t a kód interpreter és a boot töltő, 1K-t a pixel manipulációs és a lebegőpontos műveletek foglalnak el.

Az alkalmazott aritmetikai egység (ALU) az AM2901 típusú bitszeletelt processzorra épül (150 nsec). A különböző bázisregiszterek az AM2901 belső regisztereiben helyezkednek el. Aritmetikai regiszterek nincsenek, ezek szerepét az un. "expression stack" látja el, amit külön hardware valósít meg (16*16 RAM). A LILITH architektúra itt közvetlenül kihasználja annak ismeretét, hogy a MODULA fordító az aritmetikai és logikai kifejezéseket egyaránt postfix formátumra hozza, és ezért a stack szervezés igen egyszerű kódgenerálást tesz lehetővé. Külön érdekesség, hogy a logikai műveletek kezelése nem különbözik az aritmetikai műveletek kezelésétől, a feltételes ugró utasítások is az expression stack tetejének értéke szerint (és nem valamiféle feltétel kódok szerint) ugranak. Lényeges tulajdonság, hogy az expression stack túlcsordulását NEM kell hardware-ben ellenőrizni, mert a fordító nem generál olyan kódot, amely 16-nál több elemet tenne a stack-re. A LILITH architektúra számos egyéb olyan ellenőrzés alól is mentesíthette a hardware-t, ahol a MODULA fordító már fordítási szinten ki tudja szűrni a hibát. Az ALU részét képezi egy Barrel shifter, amely lehetővé teszi tetszőleges rotáció megvalósítását, vagy maszk generálását egy lépésben. Ezt a mikroprogram felhasználja a halmaz-, a lebegőpontos- és mindenekelőtt a bittérképre vonatkozó parancsok megvalósításakor. A LILITH-nek a software felé mutatott architektúráját (nevezzük gépi architektúrának) külön tárgyaljuk részletesen.

A LILITH központi tára 16 Kbites dinamikus RAM-okból épül fel. Az elrendezés olyan, hogy a tár 16/64 biten olvasható és 16 biten írható. A tár címzésére 18 bit áll rendelkezésre. (A címzésről a gépi architektúra kapcsán szólunk részletesebben.)

A 64 bites olvasás elsősorban a képernyő frissités miatt fontos, 16 bites olvasás esetén ugyanis a képernyő vezérlő (RCU) a tár hozzáférések felét vinné el, ami természetesen elfogadhatatlan. Maga a képernyő két változatban létezik (592*768-as fektetett és 928*704-es állított), mindkettő interlace-es. (Ez azt jelenti, hogy egy frissítési ciklusban csak minden második pontsor kerül frissítésre, tehát feleannyira terheli a tárat, mint a nem interlace-es képernyő. Hátránya persze a gyöngébb képminőség, ami egyrészt nagyobb utánvilágítású képernyőt igényel, másrészt fokozza a fehér pontok vibrálását.)

A 64 bites olvasást használja az utasítás kiolvasó (IFU) is. A LILITH utasításai byte-folyamot alkotnak, és a 64 bites olvasás jelentősen csökkenti a tárhoz fordulások számát. Az RCU és az IFU együttes tárhozzáférési igénye 10% körül van.

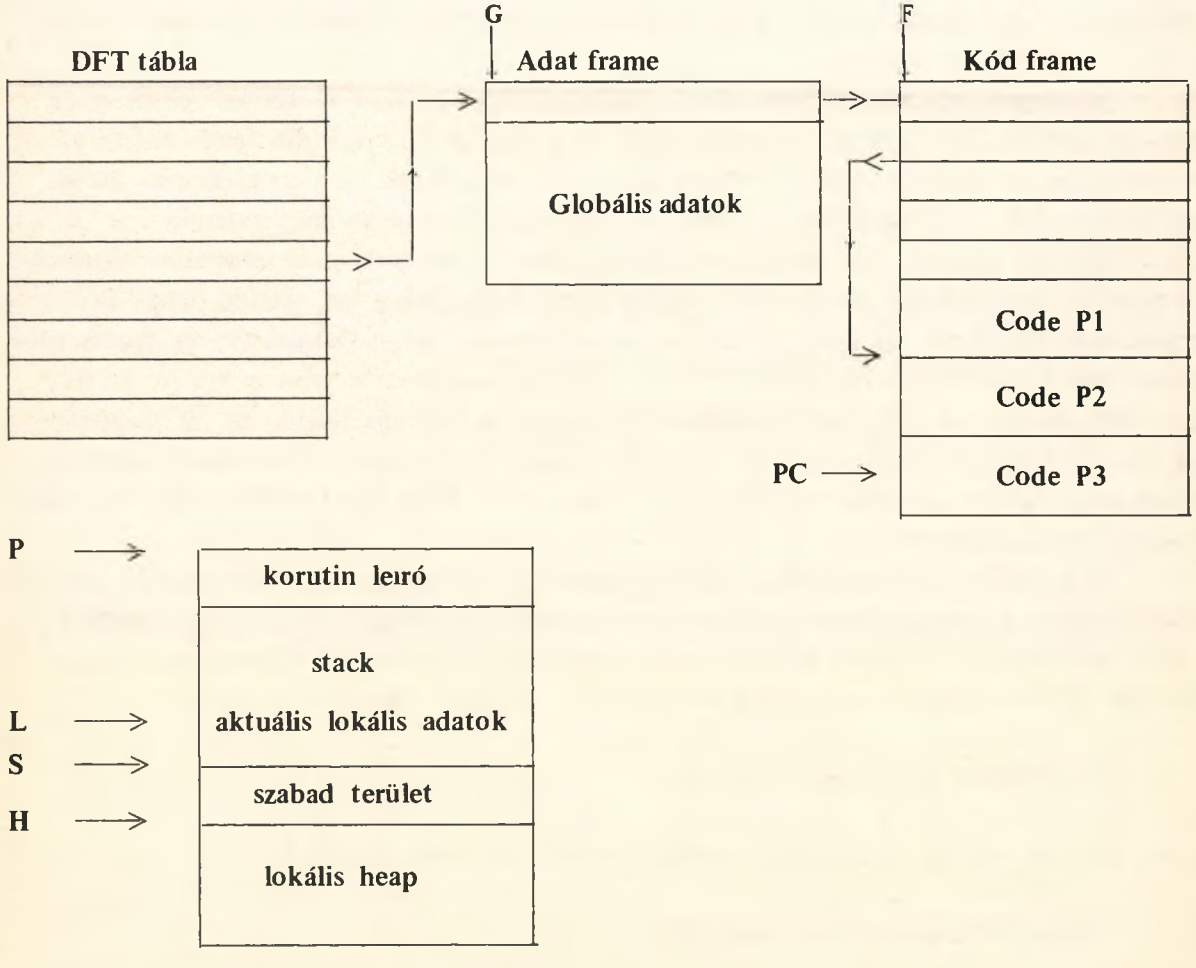
A LILITH háttértára a Honeywell-Bull D-120-as cserélhető (cartridge) diszk, amely a diszk csatolón (DCU) keresztül kapcsolódik a buszra. A diszk vezérlését mikroprogram támogatja speciális (a fordító által nem generált) utasítások által. A diszk kapacitása 10 MByte, maximális átviteli sebessége 720 KB/s, a tényleges átvitel szekvenciális file-ok esetén 60 KB/s.

A billentyűzet (keyboard) interface-ének kezelését is támogatja a mikroprogram. A mouse interface kezelését software végzi, amely az interface-ről bejövő 10 bites számlálókat 16 bites számlálókká képezi le.

A LILITH elektronikája 10 kártyán helyezkedik el: az ALU, az MCU, a mikroprogramtár, az IFU és a periféria illesztők 1-1 kártyán, a központi tár 4 kártyán. További 1 kártyát foglal el a "monitor", amely belövési és tesztelési feladatok ellátására alkalmas.

4.2 A LILITH GÉPI ARCHITEKTURA

A LILITH gép architektúra részletes ismertetése [Wirt81]-ben és [Jaco82]-ben található, most csak főbb elemeit tekintjük át. A LILITH gépi architektúra felépítését a 4.2 ábra szemlélteti.



4.2 ábra

A LILITH gépi architektúra

4.2.1 Regiszterek és nevezetes tárterületek

A külön fordított modulok (ld. a MODULA nyelvről szóló részt) 0.szintű, globális adatai a modulokból képzett program (ld. a LILITH alap software-ről szóló részt is) teljes futási ideje alatt megtartják értéküket. Ezek az adatok az un. "adat frame"-ekben helyezkednek el, az éppen a tárban tartózkodó adat frame-ek címeit tartalmazza a tár nevezetes elmén elhelyezkedő un. DFT tábla (data frame table). (Minden külön fordított modulhoz tartozik adat frame, még akkor is, ha a programozó nem deklarált globális adatokat.) Az adat frame 0.szava a modulhoz tartozó un. "kód frame"-re mutat (1-el jobbra tolva, duplaszó-cím formájában). A kód frame egy ugrótáblázattal kezdődik, amelynek indexe az eljárások sorszáma, tartalma az eljárás relatív címe (a kód frame kezdetéhez képest, byte-ban). Az ugrótábla mögött helyezkedik el a modulhoz tartozó eljárások kódja.

Az éppen aktuális (az éppen futó eljárást tartalmazó modulhoz tartozó) adat frame címe a "G" regiszterben, az aktuális kód frame címe (duplaszó-címként) az "F" regiszterben helyezkedik el. Az éppen végrehajtás előtt álló utasítás (byte relativ) címét a "PC" regiszter tartalmazza.

Az éppen végrehajtás alatt álló korutinhoz tartozó "korutin frame" címét a "P" regiszter tartalmazza. A korutin frame elején helyezkedik el a korutin leíró, amely a korutin állapotvektorát tárolja (ide teszi le a bázis regiszterek és a megszakítási maszk regiszter értékét a TRANSFER utasítás és a SYSTEM modulban implementált NEWPROCESS eljárás – ld. még [Wirt82] és [Szab84]. A korutin az aktivizált eljárások számára a korutin frame-ben a leíró mögött foglal tárat, stack elv szerint, tehát az egymásután aktivizált eljárások lokális adatai egymásután helyezkednek el. Az éppen futó eljárás lokális adatterületének címét az "L" regiszter tartalmazza, ami a lokális adatok gyorsabb elérését szolgálja. A korutinhoz tartozó stack aktuális tetejét az "S" regiszter tárolja, felső határát pedig "H". Az S és H közötti terület szabad. H mögött helyezkedik el a korutin dinamikus adatterülete (a heap). A stack tehát alulról fölfelé, a heap fölülről lefelé növekszik.

A központi tár nevezetes címén helyezkednek el az interrupt vektorok. A LILITH mikroprogram a megszakításokat korutin transzferré konvertálja. Az interrupt vektorok ennek megfelelően 2 PROCESS típusú változóból állnak (driver és interrupted). Megszakításkor a mikroprogram a megszakítási szintnek megfelelő interrupt vektorra

TRANSFER (interrupted, driver)

alakú utasítást ad ki: a megfelelő készülék vezérlő program (driver)

TRANSFER (driver, interrupted)

formájú utasítással térhet vissza a megszakítás előtti pontra.

A megszakítások helyes lekezeléséhez a modulok prioritását is felhasználjuk: egy adott készülék vezérlője a készülék megszakítási szintjével azonos prioritási szinten fut. Fontos szabály, hogy egy adott szintről nem szabad alacsonyabb szinten lévő eljárást

meghívni (a 0.szint kivételével ezt a mikroprogram ellenőrzi is, a 0.szintre vonatkozóan ez teljesen a programozó felelőssége). Ebből következik, hogy egy megszakítási szintről csak korutin transzfer által lehet alacsonyabb szintre jutni. Ez biztosítja az alacsonyabb szintek "zavartalanságát" a megszakításoktól, tehát egy adott szinten futó program futása egy időre felfüggeszthetődhet ugyan, de biztosan ott fog folytatódni, ahol megszakadt.

4.2.2 Az "expression stack"

A hardware-ről szóló részben már említettük, hogy a LILITH-ben az aritmetikai regiszterek szerepét az expression stack tölti be. A stack elv alkalmazása közismerten egyszerűsíti a kód generálást postfix formátumra hozott kifejezésekhez, de általában rontja a végrehajtás hatékonyságát. A hardware-ben megépített expression stack kiküszöböli ezt a hátrányt, és így az egyszerű kódgenerálást hatékony végrehajtás kiséri. Az expression stack legfelső elemét az ALU belső regisztere ("T" regiszter) tárolja, ami lehetővé teszi, hogy egy aritmetikai művelet két operandusának kiolvasása és az eredmény tárolása egyetlen mikrociklusban lejátszódhassék. Mint már a hardware leírásánál említettük, az expression stack túlcsordulását futás közben nem kell ellenőrízni, mert ezt a MODULA fordító meg tudja tenni. E ponton látható, hogy hogyan hathat egyszerűsítőleg a nyelvi környezet ismerete a gépi architektúrára.

4.2.3 Címzés

A LILITH a következő címzési módokkal rendelkezik:

- stack: ES[top] – az operandus az expression stack-en van;
- közvetlen: n – az operandus a kódban van, az utasítás mögött;
- lokális: M[L+n] – az operandus címe L-hez relatív;
- globális: M[G+n] – az operandus címe G-hez relatív;
- indirekt: M[ES[top]+n] – az operandus címe az expression stack tetején tárolt címhez relatív:
- külső: M[DFT[m]+n] – az operandus címe az "m" sorszámú modul globális adat frame-jéhez relatív.

A LILITH központi tár 16 bites szavakból épül fel. Indirekt címzés során futás közben abszolút címek keletkeznek. (Pl. ha a MODULA programban POINTER típusú érték keletkezik, vagy VAR típusú paraméter átadás történik.) Az abszolút címek alkalmazásának előnye, hogy gyors elérést tesz lehetővé, hátránya viszont, hogy a címzett adatok nem mozdíthatók el a tárban, valamint, hogy a maximális cím értéket a gépi szó mérete határozza meg. A LILITH esetében a 16 bites szó 64 K egység címzését teszi lehetővé, ahol ez az egység a szó (az utasítások általában szavakon operálnak). A LILITH ezért 64 Kszó adat címzésére alkalmas. A gépbe fizikailag ennél több tár is elhelyezhető. Ez a kiegészítő tár részint a kód tárolására használható (a kód szabadon el-

tolható, abszolút címeket nem tartalmaz), részint speciális utasításokkal érhető el (amelyek két szóban tárolt címet képesek tárolni, egy bázis és egy eltolás összegeként). Ezeket a speciális utasításokat a fordító nem generálja, elérésük a MODULA programból "CODE" eljáráson keresztül történhet. Ez a kényelmetlen hozzáférés azt vonja maga után, hogy a kiegészítő tár (a kódon kívül) csak speciális adatok tárolására alkalmas (pl. font-ok, bit térképek stb.).

4.3 MODULA FORDITÓ A LILITH-EN

A MODULA fordító részletes leírása [Geis83]-ban és [Jaco82]-ben, utóbbi elsősorban a kódgenerálással foglalkozik. A MODULA fordító eredetileg PDP/11-re készült, és kereszt-fordítási technikával került át a LILITH-re. Ez az "előtörténet" érezhető helyenként a fordító felépítésében és működésében. A fordító több menetes, az egyes menetek köztes file-okon és a központi tárban tárolt táblázatokon keresztül kommunikálnak. A fordító a következőképpen épül fel:

- Fordító vezérlő. Ez a rész szervezi a többi menet töltését és futását (erről ld. a MEDOS-ról szóló részt is). Itt helyezkednek el a központi tárban tárolt táblázatok címei is.
- Inicializáló. Az output file-ok előkészítését végzi. Ez a rész függ a befogadó operációs rendszertől, szerencsés esetben (pl. a LILITH interpreteren) el is hagyható.
- Pass1 (1.menet). Az 1.menet alapvető feladata a lexikális és szintaktikai analizis elvégzése. Az 1.menet az azonosítókat számmá alakítja (spelling index), ami megjavítja a tárolás és feldolgozás feltételeit. Inicializálja az azonosító táblákat és a szimbólum táblákat (ld. később is), bevezeti a MODULA kulcsszavakat, a standard (eleve deklarált) objektumokat és a SYSTEM pszeudo-modul objektumait. Végigolvassa és ellenőrzi a forrás szöveget, kiértékeli a konstansokat. Szükség esetén megkeresi a megfelelő definiciós modulok fordításakor generált szimbólum file-okat (ld. alább is), és egybeolvasztja az aktuális fordítási egységgel (továbbítja a 2.menet felé a köztes file-on).
- Pass2 (2.menet). A 2.menet alapvető feladata a deklarációk analízise. Bevezeti a deklarált objektumokat a szimbólum táblába, ellenőrzi az érvényességi tartományok betartását. Megfelelő hívást generál a lokális modulok inicializálásához (a modulok inicializáló része név nélküli eljárásnak tekinthető). Kiszámítja a deklarált változók és rekord mezők relatív címeit. Számokat rendel az eljárásokhoz. Kiértékeli a deklarációkban előforduló konstans kifejezéseket. Ellenőrzi az exportok és az előrehivatkozások teljességét, valamint implementációs modul fordítása esetén a megfelelő definiciós modul elemek (eljárás fejek és rejtett típusok) helyes és teljes kiegészítését. A 2.menet gene-

rálja az un. referencia file-t, amely a szimbólikus debugger működését támogatja.

- Pass3 (3.menet). A 3.menet alapvető feladata az utasítások analízise. Elvégzi a típus kompatibilitásokra vonatkozó ellenőrzéseket. Kiértékeli a kifejezésekben előforduló konstansokat. A WITH utasításokhoz rejtett változókat generál. A standard tár foglalási és felszabadítási NEW és DISPOSE eljárásokat helyettesíti a felhasználó által definiált eljárások hívására.

- Pass4 (4.menet). A 4.menet alapvető feladata a kód generálása. A 4.menet kódot generál a kifejezésekhez, utasításokhoz, eljáráshívásokhoz. Szükség esetén megfelelő információkat készít elő a köztes file-on a listázó számára.

- Szimbólum file generáló. Ez a "menet" csak definiciós modul fordítása után kerül végrehajtásra, a 2.menet után. Feladata, hogy egy file-ba írja mindazokat az információkat, amelyek szükségesek ahhoz, hogy a definicióra hivatkozó modulok (a saját implementációs modulja és az importáló modulok) fordításakor el lehessen végezni a hivatkozásokra vonatkozó teljes szintaktikus ellenérzést.

- Listázó. Szükség (kérés vagy hiba) esetén lista file-t készít, az esetleges szintaktikus hibák, illetve hibátlan program esetén a beültetések feltüntetésével.

4.4 A LILITH OPERÁCIÓS RENDSZERE, A MEDOS

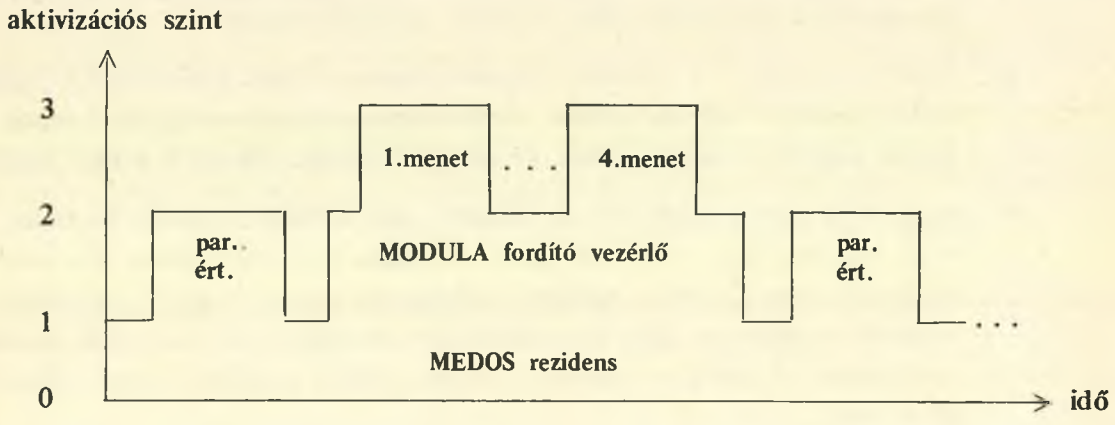
A LILITH operációs rendszerének a MEDOS-2-nek (a továbbiakban MEDOS) részletes leírása [Knud82]-ben, alkalmazói ismertetése [Lili82]-ben található. Itt most megkisérlünk áttekintést adni a MEDOS-ban követett fő elvekről.

A MEDOS alapvető szolgáltatása, hogy lehetővé tegye "programok" futtatását. A program a MEDOS értelmében lehet egyetlen modul, vagy egy modul és az általa importált modulok összessége. A MEDOS alapvető szolgáltatást nyújtó része tárrezidens program, amely a kezdeti töltés (boot) hatására kerül a tárba. A rezidens rész fő szolgáltatásai: a parancs értelmező automatikus betöltése, program szerkesztés és töltés, program indítás, processz kezelés, dinamikus tárkezelés és file-ok kezelése a diszken. A további részletesebb ismertetésben a MEDOS legújabb verziójára támaszkodunk.

4.4.1 A MEDOS felhasználói interface-e

A kezdeti töltés (boot) után a rendszer automatikusan betölti a parancs értelmezőt (amely tehát nem része a tárrezidens rendszernek), amely parancsként az indítandó program nevét várja. A begépelt programnév után a rendszer betölti a megfelelő prog-

ramot, majd ha az lefutott, akkor ismét automatikusan betölti a parancs értelmezőt. A betöltött programnak természetesen módja van további programokat hívni, ehhez a MEDOS támogatást ad (ld. a következő pont), de ezt a szóban forgó programnak kell szerveznie. A 4.3.ábra szemlélteti a MODULA fordító futását:



4.3 ábra

Programok végrehajtása a MEDOS-ban

A képernyő előtt ülő felhasználó begépeli a fordító nevét (a rendszer már gépelés közben keresni kezd, és ha a név egyértelművé válik, akkor automatikusan kiegészíti); ennek hatására betöltődik a fordító vezérlője, amely megszervezi az egyes menetek töltését.

Nem rezidens segédprogramok formájában a MEDOS számos további szolgáltatást nyújt (könyvtár kezelő, különböző másolók stb.), részint a parancs értelmezőn keresztül, részint programból hívhatóan.

4.4.2 Programok szerkesztése és töltése

A programok szerkesztését és töltését a "Program" nevű modul végzi. A szerkesztés és töltés összekapcsolásának előnye, hogy egy több modulból álló program esetén az egyik modul megváltoztatása után nem kell újraszerkeszteni, hiszen a töltés automatikus szerkesztést von maga után. A LILITH különleges architektúrája és kapcsolata a MODULA fordítóval ezt az eljárást elfogadható sebességűvé teszi. (A LILITH interpreteren [Bösz83], ahol természetesen a sebesség viszonyok nem olyan jók, mint a LILITH-en, a "belőtt" rendszerprogramok számára külön szerkesztési menetet iktattunk be [Szab84], és ilyenkor az előre leszerkesztett programokat igen gyorsan tudjuk betölteni.) Maga a

szerkesztési funkció nagyon egyszerű; fel kell oldani a lefordított modulokban előforduló hivatkozásokat a külső, külön fordított modulokra. A szerkesztő összegyűjti a vezérmodulként kijelölt modul által importált modulokat; ha ezek közül egyesek már a tárban tartózkodnak, akkor azokat nem tölti be újra. A már említett DFT tábla tartalmazza az egyes modulok globális adatterületének címét, egy másik, hasonló táblázat (loadedModules) az egyes programokhoz tartozó legmagasabb modul sorszámot. A globális adatterület első szava a kód frame-re mutat, amelyet egy leíró előz meg, amely egyebek között tartalmazza a modul nevét. A szerkesztő-töltő a kód frame számára lehetőleg a memória felső részében foglal tárterületet (a "Frames" modulon keresztül), a globális adatok számára csak az alsó 64 Kszóban szabad allokálni ("Heap").

A "Program" modul fő szolgáltatását, a (szerkesztéssel egybekötött) töltést kétféleképpen lehet igénybe venni: az "Include" és a "Call" eljáráson keresztül. Mindkettőnek bemeneti paramétere a vezérmodul neve. Az "Include" eljárás hívása a betöltött programot "egybeolvasztja" a hívóval, vezérmoduljának inicializáló részét közönséges formális eljáráshívással indítja el. A "Call" által betöltött program "szuper eljárás"-ként indul el (ld. a következő pontot).

Az érdekesség kedvéért megmutatjuk a MEDOS régebbi verziójához tartozó (a jelenleg a LILITH interpreteren futó) rezidens modulok (kicsit leegyszerűsített) szerkesztési listáját:

```
System          0H    data at     0H    code at    8CH
BasicLoop       1H    data at   0EEH    code at   1D0H
Program         2H    data at   27DH    code at   300H
Terminal        3H    data at   810H    code at   820H
Monitor         4H    data at   861H    code at   902H
CompFile        5H    data at   9EEH    code at   9FCH
Stack at    0A48H
```

A "Program" funkciója lényegében megegyezik a fent leírtakkal ("Include" funkció nélkül), a "Monitor" fő funkciója a szuper-eljárás indítás és korutin kezelés (az új verzióban ezeket a "Programs" és "Processes" modul végzi). A "System" modul a legfontosabb rendszerváltozókat definiálja, "BasicLoop" a parancs értelmezési és program töltési ciklust hajtja végre, a "Terminal" modul a billentyűzet és a képernyő egyszerű kezelését végzi. A "CompFile" modul a file rendszernek a MODULA fordító (compiler) számára releváns részét tartalmazza. (A figyelmes olvasónak talán feltünt, hogy ez utóbbi modul milyen rövid; ennek oka az, hogy az interpreteren hatékonysági okokból a file rendszer érdemi részét Assembler-ben implementáltuk.)

4.4.3 "Szuper-eljárások"

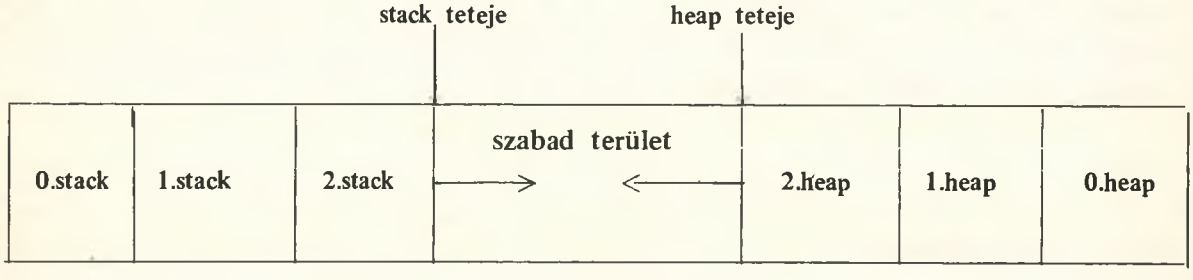
A programok indítását "szuper-eljárás"-ként a "Programs" nevű modul végzi. A kezelése alatt álló programokat egy-egy leíró jellemzi (4.4 ábra). Egy új program indításakor egy új leírót hoz létre a leírók stack-jén (programStack).

mainCoroutine	CARDINAL
state	CARDINAL
initProcs	CARDINAL
termProcs	CARDINAL
sharedProgram	Level
heapBottom	ADDRESS
heapTop	ADDRESS
heapLimit	ADDRESS
allocationCoroutine	PROCESS
freeList	↑FreeElement
heapKind	HeapKind
user	UserId

4.4 ábra

Program leíró

A programok indítása analóg egy közönséges eljárás indításával, a program így felfogható, mint egy szuper-eljárás. A Programs modul ennek megfelelően az adatok tárolására rendelkezésére álló tárterületet (az alsó 64 Kszót) úgy szervezi, hogy az felfogható két egymás felé növekvő stack-ként (4.5 ábra); az egyik, a magasabb címek felé növekvő, a fél-dinamikus adatok tárolására szolgáló eljárás-stack, a másik, az alacsonyabb címek felé növekvő, a dinamikus adatok tárolására szolgáló heap (a heap elnevezés nem precíz; a dinamikus adatok tárolása nem feltétlenül heap jellegű).



4.5 ábra

Programok aktivizálása

Ha a modul új szuper-eljárást (programot) indít, akkor befagyasztja az alatta lévő szintű szuper-eljárás stack és heap határait, és a fennmaradó területet az éppen induló szuper-eljárás rendelkezésére bocsátja, amely alulról felfelé foglal helyet fél-dinamikus, stack-szerűen elért adatainak (eljárás hiváskor az eljárás lokális adatai), fölülről lefelé pedig dinamikus adatainak (NEW-val foglalt, POINTER-en keresztül elért adatok). A heap felépítése annyiban eltérhet az ábrán láthatótól, hogy több egymást követő szintnek módja van közös (shared) heap-et használni. Amikor egy program lefutott, akkor a hozzá tartozó tárterület felszabadul, és ismét az alatta lévő szint rendelkezésére áll (teljesen hasonlóan ahhoz, ahogy egy eljárás lefutása után a lokális adatai számára elfoglalt terület felszabadul a következő eljárás számára). Érdekes programtechnikai megoldás, hogy a Programs modul a szuper-eljárást a korutinok indítására szolgáló TRANSFER utasítással indítja el, amelynek így használhatja regiszter mentő és visszaállító tulajdonságát. A szuper-eljárásból való visszatérést ennek megfelelően szintén egy TRANSFER utasítás végzi, amelyet a szuper-eljárást reprezentáló korutin (mainCoroutine) lefutásakor fellépő TRAP hatására ad ki a trap kezelő. Ehhez a megfelelő kontextus előkészítését a NEWPROCESS eljárás végzi el a korutin indításakor.

4.4.4 Processzek és korutinok

Mint már említettük, a MODULA eszközt ad korutinok létrehozására (NEWPROCESS) és a közöttük levő vezérlésátadásra (TRANSFER). A LILITH architektúra ezen felül a megszakításokat leképezi korutin transzferre. Ezzel rendelkezésre állnak a primitívek tetszőleges párhuzamossági stratégia megvalósítására. A korutinokat jellemző kontextust, a korutin leírót a 4.5 ábra szemlélteti.

gReg
lReg
PC
mask
sReg
hReg
errCode
trapMask

4.5 ábra
Korutin leíró

A MEDOS (legújabb verziója) a "Processes" nevű modul által lehetőséget ad processzek létrehozására és ütemezésére. (A MEDOS a processz elnevezést a korutintól való megkülönböztetésként használja, elvi szempontból nem teljesen korrekt módon,

hiszen a MEDOS processzek működésének jellege közelebb áll a korutin definiciójához, mint a processzéhez.)

A MEDOS megengedi, hogy minden program szint létrehozzon párhuzamos processzeket. Megjegyezzük, hogy új program szint indítását viszont csak a fő processzként induló rezidens rendszerből engedi meg. A processzeket egy-egy leíró jellemzi (4.7 ábra).

next	↑ProcessDesc
coroutine	PROCESS
state	ProcessState
queue	↑ProcessDesc
basicPrio	BITSET
schedulingPrio	BITSET
signalAddr	ADDRESS
device	CARDINAL
nextTO	↑ProcessDesc
waitMs	CARDINAL
user	UserId
coroutineAddr	CARDINAL
coroutineSize	CARDINAL

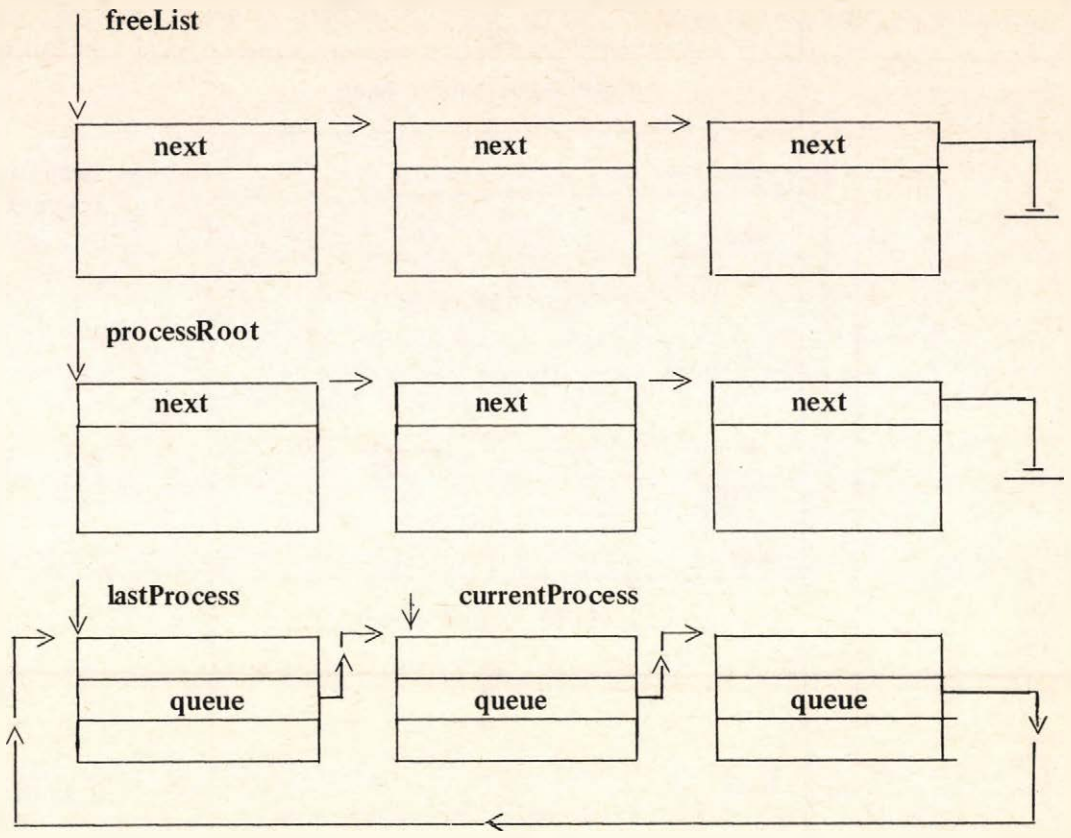
4.7 ábra

Processz leíró

A szabad leírók a freeList, a használatban lévők pedig a processRoot nevű POINTER-re fűződnek föl. A futáskész processzek leírói (a queue nevű mezőn keresztül) gyűrűbe fűződnek (4.8 ábra).

A nem futáskész processzek várakozhatnak egy jelre (SIGNAL), egy bizonyos idő leteltére, vagy mindkettőre (time-out-os várakozás). A készülék vezérlő (driver) processzek leírói teljesen hasonlóak a közönséges processzekéhez. A készülék vezérlő processzek vagy futnak, vagy egy megszakítás bejövetelére várakoznak. A megszakítás bejövetelekor a megfelelő vezérlő processz a gyűrű elejére kerül (currentProcess rámutat), és azonnal futni kezd.

A Processes modul eszközt ad közönséges és készülék vezérlő processzek létrehozására. A processz ilyenkor önálló tárterületet kap stack+heap céljára. Ha egy program szint befejezi működését, akkor az összes hozzá tartozó processz megszűnik, és a megfelelő tárterület automatikusan felszabadul.



4.8 ábra

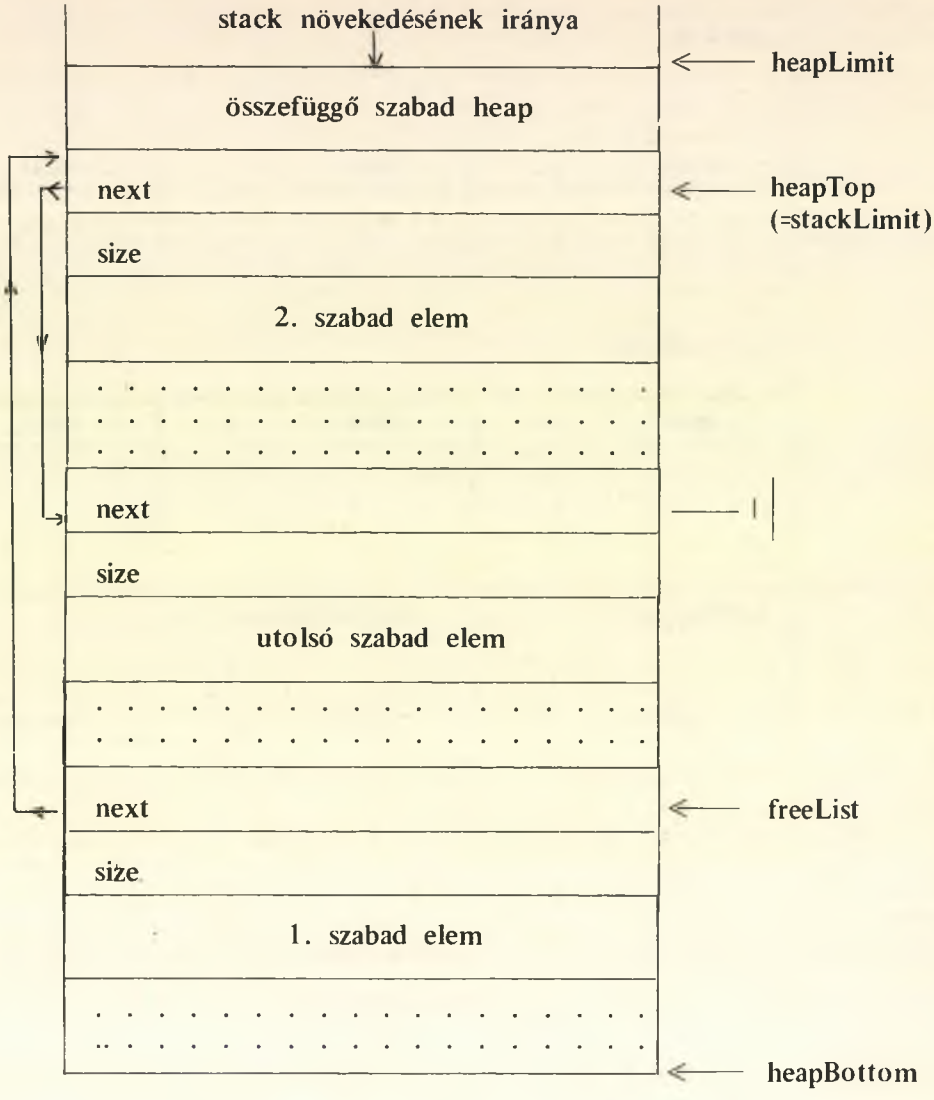
Processzek sorai

4.4.5 Dinamikus tárkezelés

Már említettük, hogy a LILITH címzési sajátságai miatt (16 bites abszolút címek keletkezése futás közben) a központi tár logikai szempontból két részre oszlik: a 16 biten címezhető alsó részre (64 Kszó) és a 18 bites cimbusz teljes tartományát kihasználó felső részre (256 Kszóig). MODULA nyelvű programok adataiként csak az alsó rész érhető el "tisztességesen". A felső rész csak speciális, a fordító által nem generált utasítások (LXFW, SXFW stb.) által érhető el, amelyek számára 2 szóban adható meg a cím, egy 4-gyel osztható és 2-vel eltolva ábrázolt "frame" cim, és egy frame-en belüli relatív cim formájában. A felső rész ezért csak különleges "adatok" tárolására szolgálhat (ilyen maga a kód, font-ok, bit térképek és tár rezidens file-ok).

A dinamikus tárkezelést a fentieknek megfelelően két különböző, bár hasonló algoritmus szolgálja: az alsó részből allokálható dinamikus területeket a "Heap" nevű modul kezeli, a speciális címzésű tárat a "Frame"nevű modul.

A heap felépítése a 4.9 ábrán látható.



4.9 ábra

A heap felépítése

A heap egy adott program szint (shared level) birtokában van, jellemző adatait a program leíró tartalmazza (4.4 ábra). Tárfoglalási kérelem esetén az algoritmus először a korábban használt, de már felszabadított elemek listáján (freeList) keres olyan elemet, amelynek mérete nem kisebb a szükségesnél. Az első ilyen elemet lefoglalja. Ha a szabad elemek listáján nincs megfelelő méretű tár elem, akkor a heap tetejéről kisérli meg a tér-foglalást. Ha a heap "fix" típusú, akkor csak a megengedett határig (heapLimit), ha "dynamic" típusú, akkor addig, amig csak az ellenkező irányból növekvő stack-kel össze nem ütközik. Tár felszabadításkor megkisérli "egybeolvasztani" a visszaadott elemet az előtte és az utána lévőkkel (ha éppen fizikailag is határosak), ha ez nem sikerül, akkor felfűzi a szabad elemek listájára.

A frame-ek címzése eltolt POINTER-ekkel történik, az eltolás 2 bit, tehát a leg-

kisebb frame hossza 4 szó. Ez éppen a frame leíró hossza is (4.10 ábra), az ábrán látható két szélső, 1 (frame-) hosszúságú frame az egész frame területet határolja be.

size	1
owner	0
type	0
version	1
size	
owner	
type	free
alignmentsize	
. . .	
size	
owner	
type	fixed
alignmentsize	
. . . . .	
size	
owner	
type	free
alignmentsize	
. . .	
size	1
owner	0
type	255
version	1

4.10 ábra

A frame-ek felépítése

A tár foglalás és visszaadás algoritmusa nagyon hasonló a heap algoritmusához. A legfontosabb eltérés, hogy a Frame modul, ha nem talál megfelelő frame-et, akkor még megpróbál allokálni a heap-ből (fordítva ez természetesen nem lehetséges). A Frame modul különleges szolgáltatása, hogy lehetőséget ad arra, hogy egy frame méretét menetközben lekérdezzük, és esetleg megnöveljük (ez persze nem biztos, hogy sikerül).

4.4.6 A file rendszer

A MEDOS file fogalma meglehetősen elvont. A file rendszer interface-ét a FileSystem nevű modul adja, amely definiálja a File típust és a rajta végezhető műveleteket. Ez utóbbiak nagy része a szokásos primitívekből áll: file létrehozás, menyitás, megnevezés és átnevezés, írás, olvasás, pozicionálás, törlés stb. Ami a MEDOS file rendszert különösen érdekessé teszi, az az a tulajdonsága, hogy teljesen készülékfüggetlen. A FileSystem modul tulajdonképpen egy keretet ad, amelyhez tetszőleges készülékhez tartozó implementáció jelentkezhet be dinamikusan, futás közben. Ennek megfelelően a FileSystem a fent felsorolt primitíveken túlmenően eszközt ad arra is, hogy egy új készülékhez tartozó implementáció bejelentkezzék a file rendszerbe, egyszersmind átadja azt a két eljárását, amelyek a file nyilvántartási (DirectoryCommand), illetve a file-on belüli (FileCommand) műveleteket végzik. Ez a megoldás programozástechnikai szempontból is nagyon érdekes, szép példa az eljárás típusú változó használatára, és az objekt orientált programozás jeleit is magán viseli (nem ingyen, ami a hatékonyságot illeti). A FileSystem a továbbiakban a készüléknév alapján kapcsol ki a megfelelő készülékhez tartozó implementációra. A legfontosabb ilyen készülék természetesen a diszk, továbbiak lehetnek pl. a lokális hálózat (remote files), a központi tár (memória rezidens ideiglenes file-ok) stb.

A file-ok megnevezésére a file nevek szolgálnak, amelyek a MODULA nyelvben használatos minősített azonosítóhoz hasonló szintaktikával rendelkeznek [Szab84]. Az első minősítő tag kötelezően a készüléknév (pl. DK.SYS.LIB.Filel.OBJ). A file név többi részét a file rendszer nem kezeli, csak továbbítja a file rendszer implementációjához. Ezzel lehetőséget ad arra, hogy az implementáció modul egy- vagy többszintű könyvtárrendszert hozzon létre. Az ideiglenes file-ok névtelenek. A file rendszeren belül egyedi belső azonosítók különböztetik meg a file-okat. A file rendszer egészében erősen törekszik a NYITOTTSÁGRA; a legtöbb kérdésben szabad kezet hagy a file-okat implementáló modulok készítőinek.

A file rendszer felhasználói egyszerűbb esetekben a File típuson (4.11 ábra) csak a file rendszer által exportált eljárásokat hajtják végre.

"Elvetemültebb" programozók élhetnek azzal a lehetőséggel, hogy a file típus mezőin keresztül közvetlenül elérjék a file rendszer által kezelt puffereket. Ez a lehetőség sajnos komoly hibaforrás, ezért az egész koncepció erősen vitatható. Annál is inkább, mert a pufferelés megvalósítását kötelezően írja elő a file rendszer implementációjának, ami a nyitottság elvével sem áll igazán összhangban.

A file-okat implementáló készülékek közül kiemelt jelentőségű a diszk, amelyet a

DiskSystem nevű modul kezel. A diszken kétféle katalógus helyezkedik el, az egyik a file-okat írja le (4.12 ábra), a másik a neveket tartalmazza.

puffer cím
puffer elem címe
elem cím páratlan jelző
puffer szabad hely címe
szabad cím páratlan jelző
file állapot jelző
file vége jelző
átviteli eredmény jelző
parancs
. . .
készülék név
. . .

4.11 ábra

A file típus fontosabb mezői

fenntartott hely
lokális azonosító
lokális verziószám
generáció (apa, fiú)
hossz
módosítás (időpont)
hivatkozás (időpont)
védelem
. . .
lap nyilvántartás

4.12 ábra

File leíró a diszken

A két katalógus között a kapcsolatot az azonos index teremti meg. A diszken a katalógusok kivételével valamennyi file-t lapokra bontva tárolja a rendszer (1 lap = = 8 szektor = 2 KByte). Ennek az az előnye, hogy sohase kell tömöríteni. A lapokat elérés-folytonosan tárolja a rendszer, az alkalmazott "interleaving" faktornak (12 szektor)

megfelelően. A diszk lapjairól nyilvántartás van a központi tárban, ami jó és gyors allokációt enged meg. Valahányszor megnyitjuk a DiskSystem-et, mindig ellenőrzi a lapok és a lokális file azonosítók nyilvántartását, és inkonzisztencia esetén megfelelő intézkedést tesz (például törli a "megmaradt" ideiglenes file-okat).

Az adatforgalmat a diszk rendszer saját tulajdonú és kezelésű puffereken keresztül bonyolítja. A puffereket statikusan lefoglalja, a pufferek száma egyenlő az egyszerre megnyitható file-ok maximális számával (16). Az eljárás előnye, hogy megfelelő stratégiával biztosítható, hogy minden file-hoz legalább 1 puffer mindig rendelkezésre álljon. A pufferek mérete 1 szektor (256 byte) befogadására alkalmas. A diszk rendszer statisztikát vezet a file-ok elérési módjáról, és szekvenciális elérés esetén előreolvasással gyorsítja az átvitelt (ez átlagosan 50% gyorsulást hoz). (A puffer kezelés és a biztonsági követelmények részletes ismertetése megtalálható [Knud82]-ben.)

4.5 A MAGNET LOKÁLIS HÁLÓZAT

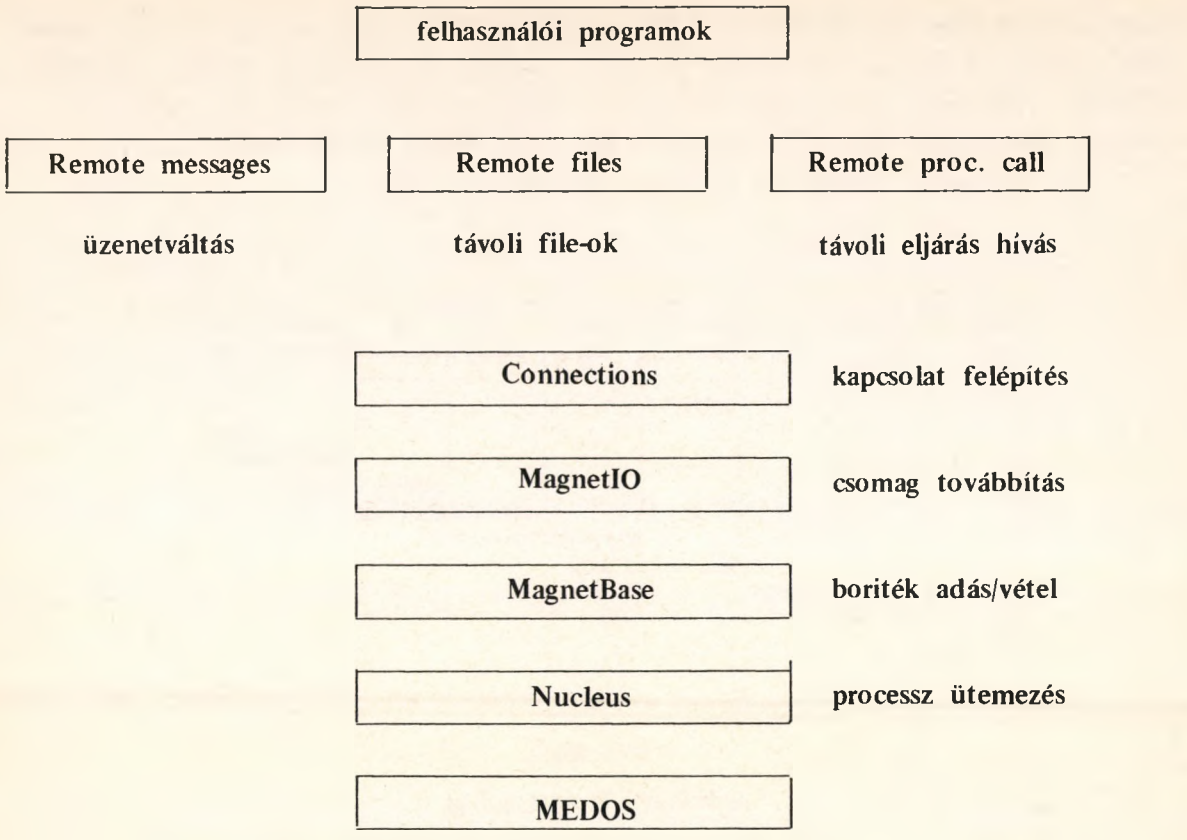
MAGNET a LILITH-ekből felépíthető lokális hálózati elemek gyűjtőneve [Hopp84, Wagn83]. A hálózat homogén, és ez sok mindent drasztikusan leegyszerűsit. A gyakorlatban nemcsak a hardware, hanem az egyes LILITH-eken futó alap software (operációs rendszer és hálózati software) is azonos, csak az applikációk különböznek.

A MAGNET fizikai alapját egy ETHERNET típusú hálózat alkotja, amely 3 Mbit/sec átviteli sebességet biztosít (Stanford University kisérleti interface). Érdemes megjegyezni, hogy [Hopp83] adatai szerint a Zürich-i ETH-n üzemben lévő hálózatban a kábel terhelése általában 5% alatt van, tehát mondhatjuk, hogy átlagos irodai alkalmazásokban 1 Mbit/sec fölött bármilyen sebesség jó, a nagy sebességek (10 Mbit/sec) viszont esetenként túlzott terheket róhatnak a software-re. Ezért a hálózat sebességének megválasztásakor célszerű előre felmérni a várható alkalmazásokat, mert nem biztos, hogy a leggyorsabb a legjobb (sat cito si sat bene). A MAGNET interface kezel egy 16 csomag befogadására elegendő puffert, amely a központi tárban helyezkedik el. Ez lehetővé teszi, hogy a gyors egymásutánban érkező csomagok se vesszenek el a software lassúsága miatt.

A MAGNET software elvi felépítése a 4.13 ábrán látható.

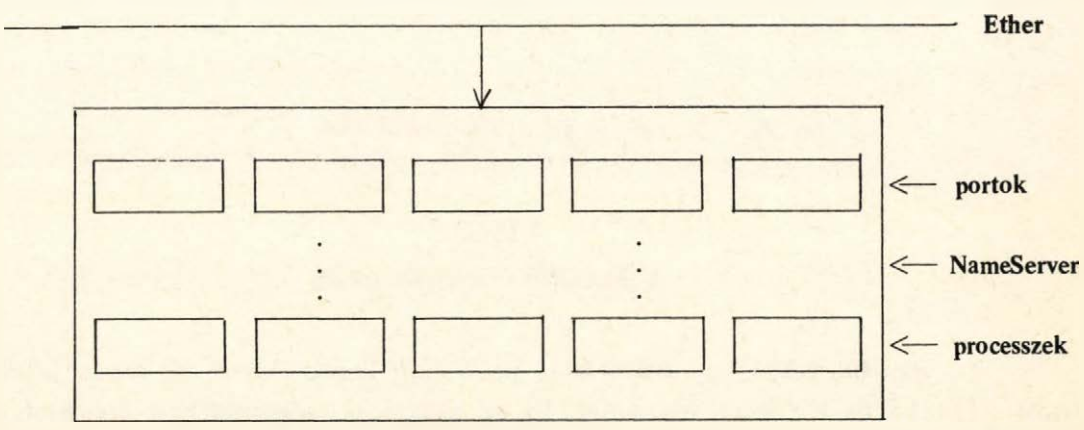
Az ábrán a MEDOS fölötti "Nucleus" nevű egység arra utal, hogy a MAGNET saját nucleus-szal (processz ütmezővel) működik (ennek részben történeti okai vannak – a MAGNET irásakor a MEDOS egyáltalán nem támogatta a párhuzamosságot). A MAGNET útmezője üzenet orientált, szolgáltatásai viszonylag szerények, amennyiben egy processz egyszerre csak egy port-ra (üzenetre) tud várni, kívánság szerint time-out-tal. Ez ugyan tartalmazza a szükséges minimumot, de komolyabb alkalmazásoknál (pl. a MAGNET esetében is) eléggé kényelmetlen.

A hálózat magasabb szintjeinek ismertetése előtt tekintsük át a címzési mechanizmust (4.14 ábra).



4.13 ábra

A MAGNET software felépítése



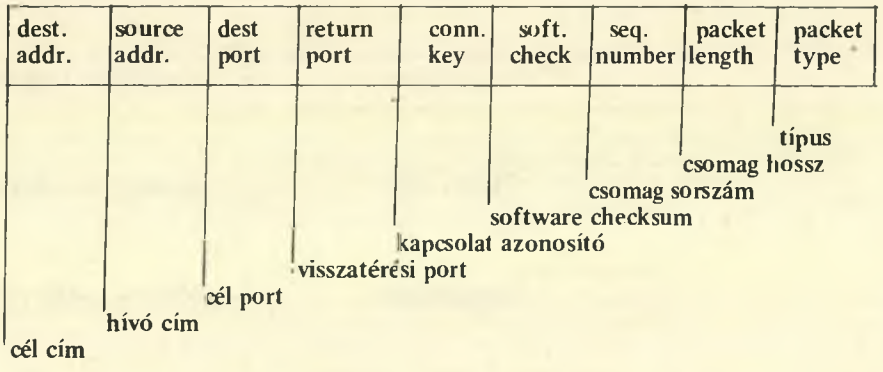
4.14 ábra

A MAGNET címzése

Az egyes állomásoknak (az egyes LILITH-eknek) a hardware által ismert címe van (0..255). Valamennyi állomás ismeri a broadcast címet, és elvileg egy állomás több

címet is felismerhet. Az állomáson belüli címzés eszköze a port, amelyek között vannak előre definiált és vannak privát célú portok. A portok és processzek közötti kapcsolat létesítését, valamint a processzek azonosítását (helyüktől függetlenül, név szerint) az elosztott NameServer végzi (ld. a Connections nevű modul ismertetését).

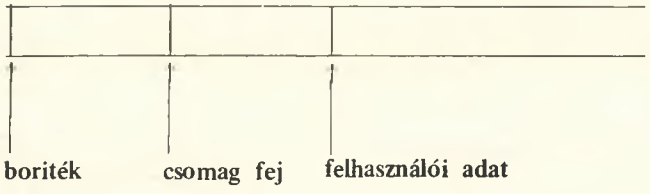
Az információ továbbítás egysége a csomag (4.15 ábra)



4.15 ábra

A MAGNET csomag feje

A csomag az egyes állomásokon belül "boriték"-ban helyezkedik el, formátuma megegyezik a Nucleus "üzenet"-ének formátumával (4.16 ábra)



4.16 ábra

A MAGNET boriték cseréj

Az üres borítékok egy megfelelő "pool"-ban helyezkednek el; innen lehet borítékot igényelni és ide lehet visszaadni. Ez az eljárás jó tárkihasználást eredményez, de egyrészt korlátozza a borítékok számát, másrészt dead-lock veszélyt rejt magában.

A MagnetBase modul tartalmazza a boriték kezelés eljárásait, valamint egy vevő és egy adó processzt. Az adó egy nevezetes portról továbbítja a borítékokba foglalt csomagokat a kábel felé, a vevő a csomagban lévő információk (destination port) alapján irányítja el a bejövő csomagot.

A csomagok átvitelére szolgáló eljárásokat a MagnetIO modul exportálja. Ezek

teszik lehetővé a fölötte lévő szint számára csomagok adását, vételét és cseréjét (adás + válasz).

A Connections nevű modul lehetővé teszi, hogy két állomás processzei között (tartós) kapcsolat épülhessen ki. A kapcsolat csak névvel rendelkező SZOLGÁLTATÁS felé épülhet ki (ezt a korlátozást feloldja, hogy lehetőség van ideiglenes szolgáltatás létrehozására – éppen egy kapcsolat idejére). A szolgáltatások nevük közlésével "jelentkeznek be" a Connections modul által implementált name server-nek. A name server broadcast üzenettel ellenőrzi, hogy a név egyedi-e a hálózatban, és ha igen, akkor feljegyzi egy memória-rezidens táblázatba. A név ezután bárhonnan hívható. A hivás szintén broadcast üzenettel történik. A kapcsolatokon csomagok forgalmazhatók. A kapcsolat állapota bármikor lekérdezhető, a megfelelő szervíz csomagokat a modul automatikusan forgalmazza. A Connections modul felett a csomagok feje már rejtett, csak a felhasználói adat hozzáférhető. A MAGNET alsó szintjei a csomagokat másolás nélkül, címek átadásával továbbítják egy állomáson belül.

A Connections modul fölött 3 modul helyezkedik el: a RemoteMessages, RemoteFiles és RemoteProcedureCall nevű modulok.

A RemoteMessages modul két irányban terjeszti ki az alatta lévő szint szolgáltatásait. Egyrészt megengedi tetszőleges hosszúságú üzenetek átvitelét, másrészt több üzenet szintű kapcsolat multiplexálását egyetlen csomag szintű kapcsolatra. Ez utóbbi pontosabban azt jelenti, hogy megengedi 1-1 és 1-n jellegű kapcsolatok felépítését. Utóbbiakat a server-ek használják. A server-ek a beérkező csomagok alapján (connection key) kiválasztják a megfelelő kapcsolatot és aktivizálják a megfelelő magasabb szintű eljárást (formális eljárás hívással – amelynek aktuális értékét inicializáláskor tárolják el). Az 1-1 jellegű kapcsolatok általában úgy épülnek fel, hogy a kapcsolatba lépni kívánó partnerek ideiglenesen szerver-ként installálják magukat, majd ha megtalálták egymást, a nevek eltünnek a hálózatból. Az egymásra találás időtartamára time-out adható meg (ez végtelen is lehet).

A RemoteFiles modul lehetővé teszi, hogy az egyik állomás a saját file rendszerén keresztül elérje a másik állomáson tárolt file-okat. Mint már említettük, a LILITH file rendszere eleve készülék független, és megengedi, hogy egy új készülék futás közben kapcsolódjék be a file rendszer szolgáltatói közé. Ilyen szolgáltatóként kapcsolódik be a RemoteFiles, amely a hálózatot mint egy készüléket mutatja. Erre épül a MagnetFiles nevű program, amely eleve installál egy (vagy több) diszk nevet a partnerei számára. Ezután tetszőleges file műveletek hajthatók végre, kezdve az egyszerű file másolástól egészen a byte szintű elérésig.

A RemoteProcedureCall nevű hálózati elérés azt teszi lehetővé, hogy egy másik állomáson lévő eljárást hajtsunk végre. Ennek a módszernek részletes elvi tárgyalása található [Nels81]-ben. A LILITH-en a távoli eljárás hívások nem tartalmazhatnak cím jellegű paramétereket (POINTER, VAR) és globális változókat sem. A hívás és a fogadás helyén standard műveleteket kell végrehajtani, a felhasználónak csak a távoli eljárás érdemi részét kell megírnia. A standard műveletek generálását elő-fordító végzi.

A MAGNET fölött jelenleg 3 fő szolgáltatás (server) működik: nyomtató állomás, file tároló és postaláda. Továbbiak vannak készülőben: dátum és idő lekérdezés, távoli töltés, távoli program végrehajtás stb.

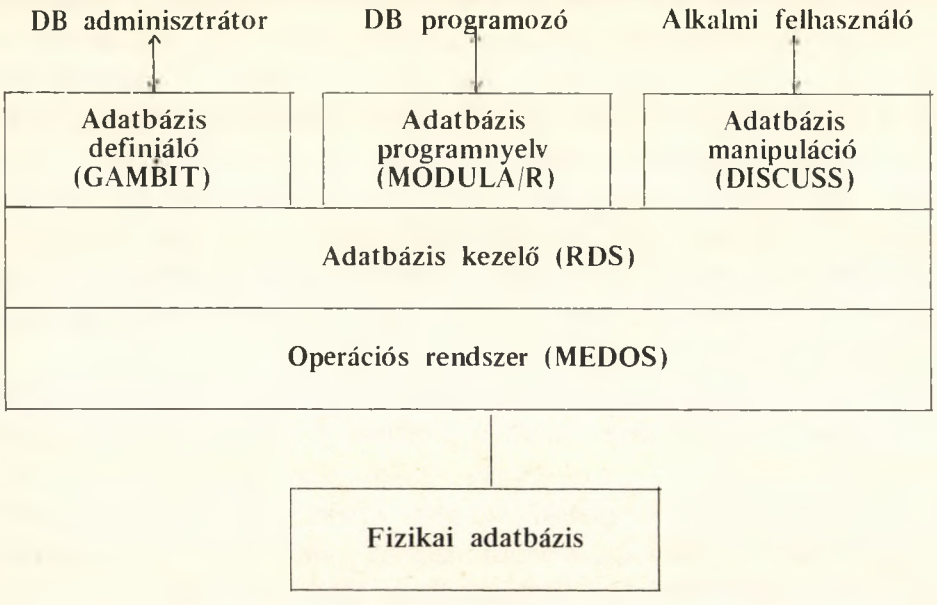
4.6 ADATBÁZIS KEZELÉS – LIDAS

A LIDAS (Lilith Database System) a Lilith számítógépre készült egyfelhasználós személyi adatbázis kezelő rendszer. Részletes leírása [Zehn83]-ban található, itt megkísérlünk áttekintést adni róla.

A LIDAS rendszer alapvetően oktatási és software architektúra kutatási célzattal jött létre. A készítők a software mérnökök, titkárnők, egyetemi oktatók és hallgatók számára egy olyan, a mindennapos használatra készült adatbázis kezelő rendszert hoztak létre, amelyik jól illeszkedik a LILITH személyi számítógép jellegéhez (pl. az adatbázis mérete a néhány 100 KByte-tól néhány MByte-ig terjed) és kihasználja a LILITH magasszínvonalú interaktív lehetőségeit (menű, mouse, grafika). A teljes LIDAS rendszert MODULA-ban írták. A LIDAS kutatás egyik fontos célja volt annak a kutatása is, hogy milyen felhasználói interface-eket nyújtson egy személyi számítógép környezetben működő adatbázis kezelő rendszer.

A LIDAS rendszer magában foglalja a rendszer felhasználói interface-eit képező programrendszereket is (4.17 ábra):

- interaktív grafikus adat(struktúra) definiáló rendszer (GAMBIT),
- adatbázis programozási nyelv és környezete (MODULA/R),
- interaktív adatbázis lekérdező és módosító rendszer (DISCUSS).



4.17 ábra

A LIDAS felhasználói interface-e

A következőkben a rendszer interface-eitől elindulva vázlatosan bemutatjuk a rendszer architektúráját. A bemutatás sorrendjét az indokolja, hogy a LIDAS-ban nem alkalmaztak originálisan új adatbázis kezelési módszert, helyette inkább a magasszintű felhasználói interface-ek kidolgozására koncentráltak.

4.6.1 Adatbázis programozási nyelv (MODULA/R)

A MODULA/R adatbázis programozási nyelv a MODULA-2 nyelvet és a relációs adatbázis modell elemeit egyesíti. A MODULA/R előzménye a PASCAL nyelvből kifejlesztett PASCAL/R nyelv volt. A MODULA/R fontosabb konstrukcióit példákon keresztül mutatjuk be.

A MODULA/R a reláció fogalmát mint új adattípust (RELATION típus) vezeti be. A RELATION típus azonosan struktúrált elemek halmaza, felfoghatjuk MODULA RECORD-ok halmazának:

```
TYPE RelType = RELATION OF RECORD f1:F1Type; ... fn:FnType END;
```

A reláció halmaz tulajdonsága miatt egy reláció nem tartalmazhat két olyan elemet, amelyeknek a megfelelő mezői páronként azonosak.

A MODULA/R kulcsnak (KEY) nevezi a reláció azon mezőinek rendezett halmazát, amelyek alapján az elemek olyan módon állíthatók sorba, hogy nem lesz két egyforma kulccsal ellátott elem. A következő példa egy kisméretű könyvtár adatbázisának a deklarációit tartalmazza:

```
TYPE
   ItemNr     = CARDINAL;
   TitleType  = ARRAY [0..50] OF CHAR;
   NameType   = ARRAY [0..20] OF CHAR;
   KindType   = (book, conf, rep, per);
   LendStat   = (lent, claimed);
   DateType   = RECORD
                  day: [1..31];
                  month: [1..12];
                  year: [1200..2200];
                END;

   ItemRecType = RECORD
                   inr   : Itemnr;
                   title : TitleType;
                   kind  : KindType;
                 END;
   ItemRelType = RELATION inr OF ItemRecType;

   PersRecType = RECORD
                   name : NameType;
                   ...
                 END;
   PersRelType = RELATION name OF PersRecType;

   LendRecType = RECORD
                   name : NameType;
                   inr  : Itemnr;
                   stat : LendStat;
                   date : DateType;
                 END;
   LendRelType = RELATION name, inr OF LendRecType;

VAR
   items    : ItemRelType;
   persons  : PersRelType;
   lendings : LendRelType;
```

A relációk definiálásánál a RELATION alapszó után a reláció kulcsai helyezkednek el.

A MODULA/R lehetővé teszi az elsőrendű predikátum kalkulusban megfogalmazott Boolean és un. relációs kifejezések használatát. Az egzisztenciális és univerzális kvantorokat a nyelv a SOME és az EACH alapszavakkal jelöli, a relációk egyes mezőire történő hivatkozásnál pedig a MODULA-2 "." minősítést használja.

Példa: A könyvtár kölcsönzött-e könyvet?

```
SOME i IN items ((i.kind=book) AND
  SOME le IN lendings ((le.inr=i.inr) AND (le.stat=lent)))
```

A relációs kifejezés elsőrendű predikátumnak megfelelő reláció elemeket választ ki.

```
{[le.name, i.title] OF EACH le IN lendings, EACH inr IN items :
    (le.inr=i.inr) AND (le.stat=lent)}
```

A MODULA/R az értékadás (:=) operátora mellett bevezeti a relációs beiktatás (:+ insert), a törlés (:- delete) és a csere (:& replace) operátorokat. A {...} reláció konstruktor alakitja át a rekord-kifejezést relációs kifejezéssé (felfogható a MODULA SET konstruktor kiterjesztésének).

Példa: Müller visszavonja az előjegyzéseit:

```
lendings :- {EACH le IN lendings :
                le.name='Müller') AND (le.stat=claimed)}
```

A MODULA/R az összetett adatbázis operációkat tranzakcióknak (TRANSACTION) nevezi. A rendszer a tranzakciókat atomi műveletnek tekinti és biztosítja azt, hogy ha a tranzakció valamilyen (hiba) oknál fogva megszakad, akkor is megmaradjon az adatbázis konzisztenciája. Példaként tekintsük a könyv kölcsönzés tranzakcióját:

```
TRANSACTION Borrow(name:NameType; inr:Itemnr);
  VAR lendrec:LendRecType;
BEGIN
  IF NOT SOME p IN persons (p.name=name) THEN
    WriteLn('Unknown person cannot borrow');
  ELSIF NOT SOME i IN items (i.inr =inr ) THEN
    WriteLn('Unknown item cannot be borrowed');
  ELSIF SOME le IN lendings ((le.inr=inr) AND (le.name#name)) THEN
    WriteLn('Item lent or claimed');
  ELSE
    lendrec.inr := inr;
    lendrec.name:=name;
    lendrec.stat:=lent;
    Write('Return date?');
    ReadDate(lendrec.date);
    IF SOME le IN lendings (le.inr=inr) THEN
      lendings:& {lendrec};
    ELSE
      lendings:+ {lendrec};
    END;
  END;
END Borrow;
```

A LIDAS rendszer MODULA/R interface-e lehetőséget teremt arra, hogy a relációkat és a rajtuk értelmezett műveleteket mint nyelvi objektumokat használjuk programjainkban (pl. eljárás- vagy modul-lokális reláció illetve eljárás reláció paramétere formában). Az ilyen módon használt relációk perzisztenciája (élettartama) kontexus függő, a MODULA-ban megszokott szabályok szerint. (Például egy eljárás-lokális reláció az eljárás befejeződésével megszünik. A modul relációinak élettartama a modul egyéb globális változóinak élettartamával egyezik meg, tehát a modult tartalmazó eljárás lefutása után az ilyen relációk megszünnek. A relációk perzisztenciájának megnövelésére, tehát arra, hogy a relációk túlélhessék az őket definiáló programot, a MODULA/R nyelv az adatbázist definiáló modul (DATABASE DEFINITION MODULE) konstrukciót vezette be. Az ilyen modulban deklarált relációk az adatbázis permanens részeivé válnak, igy perzisztenciájuk az adatbáziséval egyezik meg. A fentiek illusztrálására a könyvtár (Library) adatbázist definiáló modult mutatjuk be:

```
DATABASE DEFINITION MODULE Library;

  EXPORT Items, persons, lendings, Borrow, ...;

  TYPE (* type definitions as above *)

  VAR
    Items    : ItemRelType;
    persons  : PersRelType;
    lendings : LendRelType;

  TRANSACTION Borrow(name:NameType; inr:Itemnr);
END Library.
```

4.6.2 Interaktiv grafikus adat definiáló rendszer (GAMBIT)

A GAMBIT rendszer olyan software eszköz, amelynek segitségével a felhasználó interaktiv módon a fokozatos finomitás (stepwise refinement) módszerével meghatározhatja a létrehozni kivánt adatbázis adatstruktúráit és azok kapcsolatait.

Az adatbázis felépitésének főbb lépései a következők:

- Az adatbázis entitás halmazait (relációk) és a köztük lévő kapcsolatokat a felhasználó a képernyőn ábrák (dobozok és az azokat összekötő vonalak) segitségével ábrázolja;
- A korábban grafikusan specifikált kapcsolatokat a relációs adat modell terminusaival irja le, tehát meg kell határoznia azokat az attribútumokat, amelyek egynél több relációban jelennek meg. Az attribútumok egymáshoz való viszonyainak megjelenítése hasonló a relációk és az azok kapcsolatait reprezentáló ábrákhoz;
- A harmadik lépésben az attribútumok pontos definiálása történik;
- Az utolsó lépés az adatbázis szemantikus konzisztencia kritériumainak meghatározását és az adatbázison értelmezett műveletek (tranzakciók) definiálását tartalmazza.

Bizonyos konzisztencia feltételek már az adatstruktúrák meghatározásából kiadód-

nak, mig a modell-független feltételeket a MODULA/R nyelvi eszközeivel (pl. relációkon értelmezett predikátumokkal) a felhasználónak kell megfogalmaznia. Mindezek eredményeképpen a GAMBIT rendszer egy olyan (MODULA/R nyelvű) adat modul halmazt (DATA MODULES) hoz létre, amely az adatstruktúra definiciók mellett a tranzakciók meghatározását is magában foglalja. Az így előállt program képezi majd az adatbázis kezelő rendszer és a felhasználói programok közötti interface-t.

4.6.3 Interaktív adatbázis lekérdező és módosító rendszer (DISCUSS)

A DISCUSS (Database Interface Specified for Casual User of a Small System) rendszer segítségével az adatbázis kezelésben járatlanabb felhasználó is képessé válik az adatbázis kezelő rendszer szolgáltatásainak rendszeres, munkaeszköz szerű igénybevételére.

A DISCUSS rendszer három komponensből áll:

- felhasználói lekérdező nyelv (HIQUEL);
- adatbázis editor;
- űrlap kezelő rendszer.

A HIQUEL (Hierarchical Interactiv Query Language) lekérdező nyelv fontosabb tulajdonságai a következők:

1. Dialógus szervezés

A rendszer a parancsok és nevek (entitások és azok attributumai) meghatározására a menü vezérelt (menü driven) dialógus technikát alkalmazza. Ez azt jelenti, hogy a választások a képernyőn felsorolt parancsok közül cursor mozgatással (a mouse segítségével) történnek. A billentyűzet használatát minimumra korlátozták, így a különféle elgépelésből származó hibák nagy része kiküszöbölődik. A lekérdezés interaktív módon előre definiált táblázatok segítségével lépésről lépésre történik.

2. Hierarchikus szemlélet felhasználói szinten

A rendszer támogatja hierarchikus struktúrák (kérdés és válasz struktúrák) definiálását és közvetlen használatát. Ez nagyon megkönnyíti a hierarchikus jellegű lekérdezések megfogalmazását.

3. Entitás-reláció szemlélet logikai szinten

A rendszer az entitás-reláció megközelítést használja magasszintű adatmodell gyanánt.

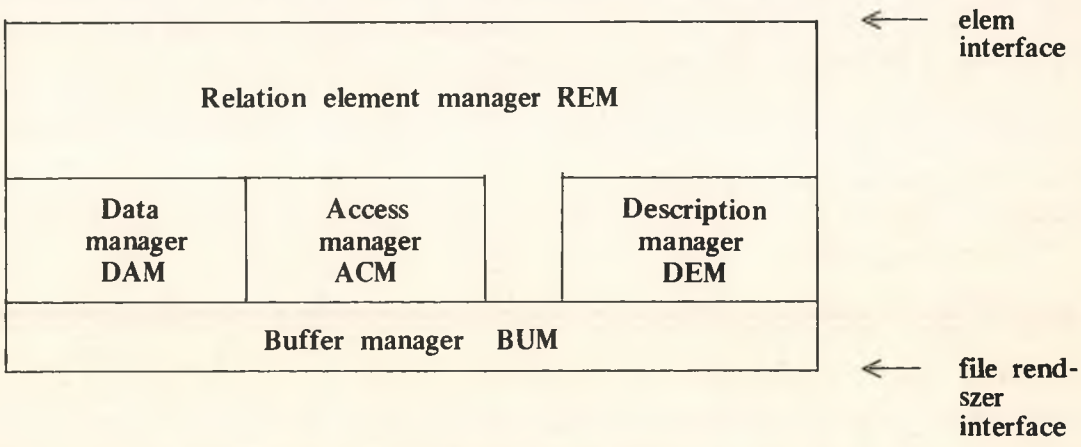
4. Egymásra épülő nyelvi rétegek

A lekérdező nyelv (HIQUEL) használatának megkönnyítésére a tervezők a nyelvet egymásra épülő rétegekre bontották, lehetővé téve azt, hogy a viszonylag kevés ismerettel

rendelkező felhasználók is már hamar igénybe tudják venni.

4.6.4 Az adatbázis kezelő rendszer implementációja (RDS)

A LIDAS adatbázis kezelő rendszer központi része az RDS (Relation Data System) – 4.18 ábra.



4.18 ábra

Az RDS felépítése

Az RDS a MEDOS file rendszerére épülve a felette lévő programszintek számára egy un. adat elem interface-t (element interface) nyújt. Ezen az interface-en keresztül a programok egy reláció elemet (praktikusan egy rekordot) érhetnek el.

Az RDS BUM (Buffer Manager) modulja az adatbázis file-jainak kezelését végzi (nyitás, zárás, törlés, létrehozás, adatbeírás és -törlés). A BUM rögzített hosszúságú lapokból álló puffereket kezel, kiosztja a lapokat, szervezi a központi- és a háttértár közötti lapcseréket.

A DAM (Data Manager) egy reláció elem egy adott lapon való allokálását végzi. Az elemek azonosítása a lap sorszámával és a lapon belüli relatív címmel történik. A DAM modul műveleteket szolgáltat a rekordok írására, olvasására, új rekordok létrehozására és törlésére.

Az ACM (Access Manager) komponens a rekordok elérését lehetővé tevő struktúrák (access path) felépítését és kezelését végzi.

A REM (Relation Element Manager) hozza létre a korábban már említett adatelem interface-t. Az interface műveletei a reláció elemek létrehozására, törlésére, helyettesítésére, lekérdezésére szolgálnak. A modul biztosítja a legalapvetőbb konzisztencia feltételek teljesítését is (azonosító kulcsok egyértelműsége).

A DEM (Description Manager) az adatbázis struktúrájának leírására szolgáló rend-

szer táblázatok karbantartását végzi.

4.6.4.1 A relációs séma belső ábrázolása

A felhasználó adatbázisára vonatkozó információkat a meta adatbázis tárolja. Leírja az adatbázisban lévő adatok struktúráját (relációkat, attributumokat, adattípusokat) és az adatbázis konzisztencia feltételeit. Az adatbázis kezelő rendszer a meta adatbázist felhasználva tölti fel inicializáláskor a rendszer leíró tábláit, amelyek majd a felhasználói programok futása közben kerülnek alkalmazásra. A leíró táblákat a DEM modul a BUM által kezelt lapokon tárolja, tehát a meta adatbázis adatainak tárolása gyakorlatilag nem különbözik a felhasználó adatainak tárolásától.

4.6.4.2 Az adatok belső ábrázolása

A felhasználó és a meta adatbázis adatait a LIDAS rendszer 3 file-ban tárolja. Az adatbázis file (database file) az állandó relációk adatait, a hozzáférést segítő adatokat (access path), a leíró táblákat és egyéb belső állapot információkat tartalmaz. A nem permanens relációkat ideiglenes file foglalja magában, amely kezdetben a program indításakor üres. A tárolástól eltekintve azonban az átmeneti relációk nem különböznek a permanens relációktól. Az adatbázis helyreállíthatóságának (recovery) céljából a futás közben változtatott adatok régi értékeit a rendszer egy file-ban (undo) tárolja.

A file-ok lapok sorozatából épülnek fel. A sorozatok hosszúsága (a file mérete) futás közben dinamikusan változhat. (A BUM modul az általánosított óra algoritmus "Generalized Clock Algorithm" stratégiát alkalmazza a központi tárban és a diszken lévő lapok cseréjére.)

A lapok egy reláció rekordjait (elemeit) hordozzák. Mivel egy adott reláció elemeinek hosszúsága rögzített (a reláció definiálásakor határozódik meg), ezért rekord szinten az üres puffer kezelés viszonylag egyszerű. Az üres illetve a használatban lévő rekord területek kezelése egyszeresen illetve duplán láncolt listák segítségével történik.

Az ACM komponens az adat rekordok elérését gyorsító struktúrát B* - faként implementálja.

Az adatbázis konzisztenciájának védelmére szolgál a rendszer tranzakció mechanizmusa. A tranzakció olyan elemi adatbázis műveletekből álló sorozat, amely az adatbázist konzisztens állapotból konzisztens állapotba juttatja. Mivel a tranzakciók inkonzisztens állapotokon keresztül valósulnak meg, ezért, ha egy tranzakció valamilyen hiba miatt megszakad, akkor a rendszernek vissza kell tudni állítania az abortált tranzakciót megelőző állapotot. Ilyenkor a rendszer az undo file-ba mentett régi lapokat visszamásolja a megváltoztatottak helyére. Mivel a LIDAS egy diszk egységet használ, így katasztrófális diszk hiba esetén nincs automatikus helyreállítás, ez ellen a felhasználónak kell védekeznie (pl. időszakonkénti adatbázis mentéssel).

Az RDS komponens szolgáltatásait a QEM (Query Evaluation Manager) modul veszi igénybe. A QEM modul a programozó MODULA/R nyelven, relációs kifejezés vagy

predikátumok formájában megfogalmazott kérdéseit reláció elemeken végzett műveletek sorozatává transzformálja.

4.7 A LILITH SZÖVEGFELDOLGOZÓ RENDSZERE, AZ ANDRA

A LILITH szövegfeldolgozó rendszere, az ANDRA, kihasználva a LILITH különleges erőforrásait, nagyságrendekkel magasabb színvonalú szolgáltatásokat nyújt a hagyományos szövegszerkesztőknél. A rendszer részletes leírása [Gutk84]-ben található, itt megkisérlünk áttekintést adni róla.

A nagyfelbontású pont-raszteres képernyő az iróasztal lapját idézi. Egyszerre több "papírlap", vagyis window (ablak) látható rajta, amelyek egymást részben át is fedhetik. A legfelül lévő (aktív) ablakok mindegyikén editálhatunk, és az ablakok között is mozgathatunk szöveget. Igy, ha különböző ablakokban különböző dokumentumokhoz tartozó szövegrészek láthatók, akkor a dokumentumok között cserélhetünk információkat.

Az ANDRA, ahol csak lehet, menü jelleggel kéri a parancsokat, ami azt jelenti, hogy a tényleges szövegbevíteltől eltekintve szinte kizárólag a mouse használatára van szükség.

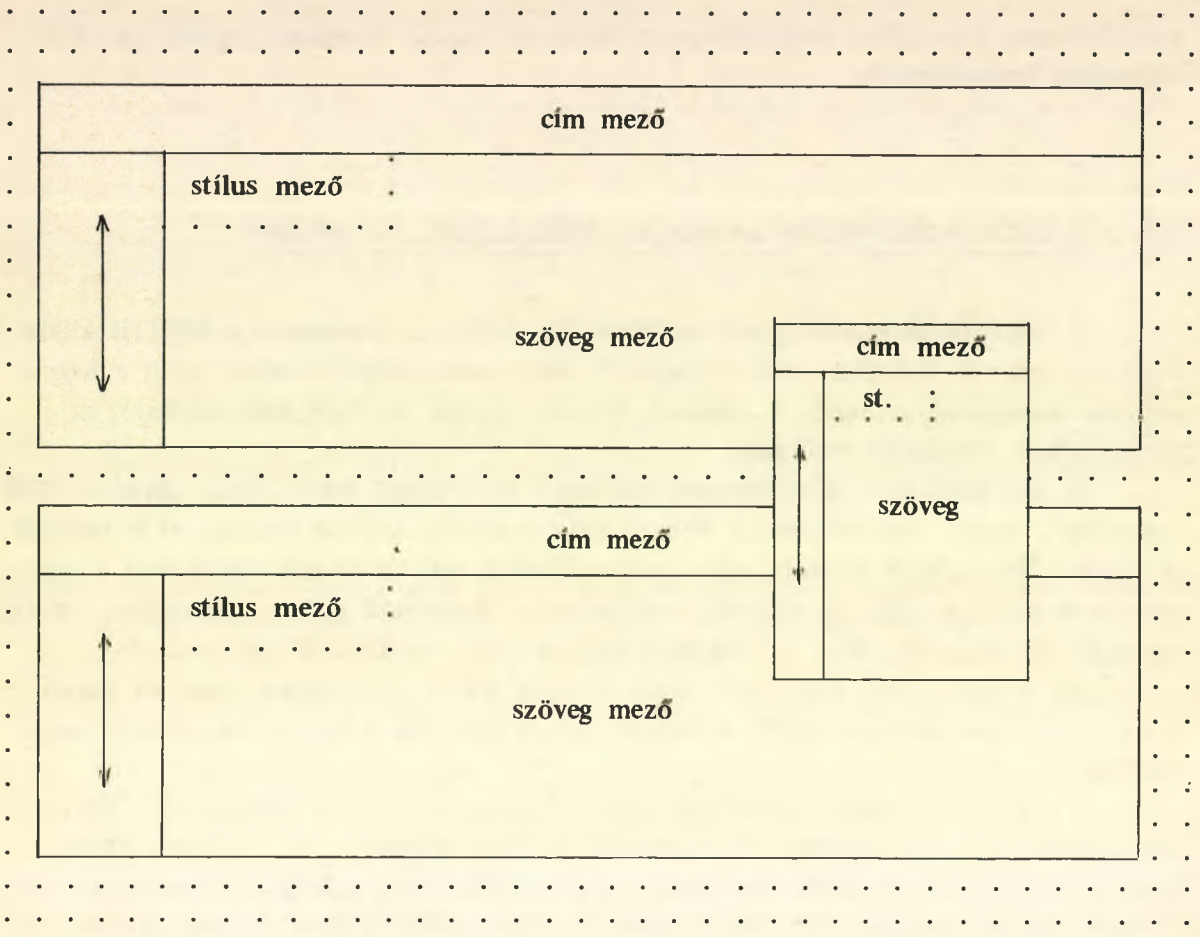
Az ANDRA gazdag eszköztárat kínál a szöveg megjelenési formájának ("stílusának") vezérlésére: betűtípusok megválasztása, tördelési előírások stb. Az ilyen jellegű parancsokat azonnal (on-line) végrehajtja, ezért a felhasználó mindig a dokumentum stílusának megfelelő képet látja a képernyőn. Bizonyos gyűjtő jellegű információk (pl. lapszámozás) csak nyomtatáskor jelennek meg.

4.7.1 Az ANDRA felhasználói interface-e

Mint említettük, az ANDRA képernyőn egy vagy több ablak látható, amelyek tartalma a felhasználó által behívott dokumentum részlet. Egy ablak 4 területre oszlik: a léptető-, a cím-, a szöveg- és a stílusmezőre (4.19 ábra). Ezek közül az első három diszjunk terület, a stílus mező fedi a szövegmező bal felső 20%-át. Az egyes mezők különböző célokra szolgálnak: a mouse középső (K) gombját benyomva más-más menü jelenik meg.

A léptető (scroll) mező (az ábrán kettős nyíl jelzi) az ablak bal oldalán elhelyezkedő keskeny csík, amely az ablak tartalmának léptetésére szolgál. A mouse bal oldali nyomógombja (B) fölfelé, a jobb oldali gomb (J) lefelé léptet, a cursor helyzetétől indulva. (K) hatására a léptető mezőben a dokumentumon belüli globális léptetés történik, a cursor függőleges helyzetének arányában. Ha tehát a cursor kb. középen áll, amikor (K)-t megnyomjuk, akkor kb. a dokumentum középső része jelenik meg az ablakban.

A szöveg mező tartalmazza a dokumentumnak az ablakon keresztül éppen látható részét. Itt kétféle cursor típus létezik, az egyik a beszúrás helyét jelző "caret", a másik a mouse mozgását jelző cursor. Ha gépelni kezdünk a billentyűzeten, akkor a



4.19 ábra

Ablakok az ANDRA képernyőn

caret helyétől kezdve az új szöveg megjelenik. Ha a caret előtt már volt szöveg, akkor azt tolja maga előtt (az esetleges bekezdés határ és sorkiegyenlítési szabály betartásával). Ha a mouse-on megnyomjuk (K)-t, akkor megjelenik a menü, amely a kiadható parancsokat tartalmazza. (A menü által "eltakart" szöveg természetesen nem vész el, és a menü törlése – (K) fölengedése – után az eredeti kép visszaáll.) Az a parancs, amelyre a cursor éppen mutat, inverz ábrázolásban jelenik meg. Érdekesség, hogy a parancsok paramétereit – vagyis azt a szövegrészt, amelyre a parancs vonatkozik – a menü behívása ELŐTT kell kijelölni. A kijelölés a mouse és (J) segítségével történik. A kiadható parancsok: copy, delete, save stb. Az ANDRA a gyakorlott (!) felhasználók számára eszközt ad a gyakori parancsok közvetlen kiadására, (B) és (J) kombinált használatával, menü nélkül.

A stílus mezőben a (K) hatására megjelenő menü a szöveg formátumára, "stílusára" vonatkozó parancsok jelennek meg. Az ANDRA "stílus" (maga is ANDRA dokumentum) a formátumra vonatkozó parancsok gyűjteménye. Ez a koncepció lehetőséget ad arra, hogy alapvető formattáló műveletekből tetszőleges stílusokat definiáljunk, egy

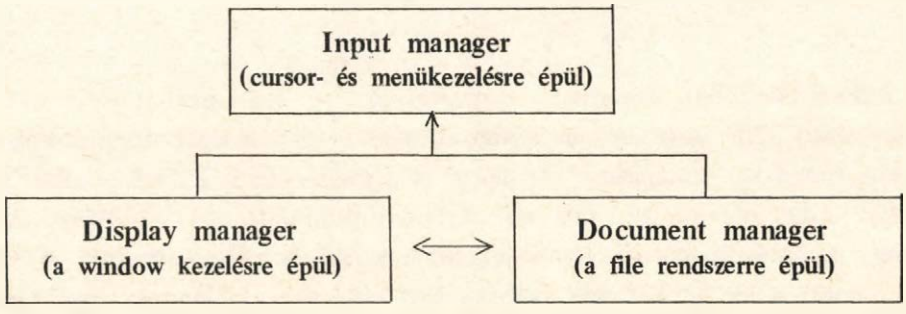
egyszerű parancsnyelv segítségével (erről ld. később is). A stílus menü (valójában menü-sorozat) lehetővé teszi, hogy egy kijelölt szövegrészhez megfelelő stíluselemeket rendeljünk (pl. cím, bekezdés stb., ahol ezek értelmezése az aktuális stílus file-ban található).

A cím mezőben ismét új parancsok adhatók ki. Itt lehet megváltoztatni egy dokumentum teljes stílusát (egy másik stílus file betöltésével). A cím mezőben parancsokat adhatunk az ablakokra vonatkozóan. Felhasíthatunk egy meglévő ablakot két ablakra és eggyé olvaszthatunk két ablakot. Ha egy dokumentum több ablakon keresztül látható, akkor a képernyőn látható sorrend megfelel a dokumentumon belüli sorrendnek, tehát pl. a képernyő tetején lévő ablakban lévő szövegrész meg kell hogy előzze a lejjebb elhelyezkedő ablakok tartalmát.

A képernyő alján található a dialógus ablak, amely az operációs rendszerrel való kapcsolattartást teszi lehetővé (pl. egy új dokumentumot tartalmazó file nevének begépelése stb.).

4.7.2 Az ANDRA rendszer felépítése

Az ANDRA rendszer két alrendszerre épül, az on-line editálást, formattálást támogató Andra-ra és a nyomtatást (laserprinterre) végző AndraPrint-re. Az Andra alrendszer felépítése a 4.20 ábrán látható.



4.20 ábra

Az ANDRA rendszer felépítése

Az input manager ciklikusan olvassa és értelmezi a billentyűzetről és a mouse-ról bejövő adatokat. Hibás parancs esetén hibajelzést küld a dialógus ablakba. Az input manager egy speciális modulja tárolja a felhasználó teljes input tevékenységét, ami lehetővé teszi, hogy ha az editálás valamilyen hiba miatt megszakad, akkor újra játsszuk az egész párbeszédet. (Ez a lehetőség az Andra belövésének is fontos segédeszköze volt.)

A dokumentum kezelésének két alapkérdése a dokumentumok tartalmának és a formattáló információknak a tárolása. Az Andra a dokumentumokat fa struktúrában tá-

rolja, a formattáló információkat az un. stílus formájában.

A formattálás vezérlésére szükség van bizonyos alacsony szintű primitívekre, mint pl. a betűtípus adatai, a margók helyzete stb. Ha ezeket a primitíveket közvetlenül használjuk a formátum vezérlésére, akkor ez nemcsak sok fölösleges munkát ró a felhasználóra, de szinte biztosan azt eredményezi, hogy minden dokumentum különböző formátumú lesz. Az ANDRA stílus koncepciója lehetővé teszi, hogy a felhasználó tetszőleges, magasabb szintű egységet definiáljon (mint cím, alcím, bekezdés, lábjegyzet stb.), az alacsony szintű primitívek segítségével. Egy érett implementációban mindenki számára elérhető néhány közkeletű stílus (pl. cikk, tanulmány, üzleti levél, stb.), amelyek lehetővé teszik, hogy egy adott intézet hasonló jellegű dokumentumai azonos külső stílussal rendelkezzenek. Egyedi célokra (pl. szerelmes levél) természetesen bármikor új stílust készíthetünk. Egy stílus definiálását szemlélteti a következő példa:

```
1 lap          :  FontGroup = TIMES, FontSize = 12,
                  FontSlope = Upright, FontWeight = Normal,
                  LeftMargin = 15, PageWidth = 155,
                  TopMargin = 20, PageLength = 250,
                  VertSpace = 2.5, Lead = 0.1;
5 főcím        :  FontSize = 20, Mode = Centered,
                  VertSpace = 7.6, Lead = 1.6;
6 alcím        :  FontSlope = Italic, Mode = LeftAdjusted,
                  VertSpace = 3.8;
7 cím          :  FontSize = 16, Mode = LeftAdjusted,
                  VertSpace = 5.7, Lead = 1.3;
10 bekezdés:      Mode = Adjusted;
11 táblázat:      Mode = LeftAdjusted,
                  TabStops = 15 / 40 / 70 / 115 / 140;
20 kiemelés:      FontSlope = Italic;
```

A felhasználó által megadható elemeket (a ":" -tól balra) a jobb oldalon látható rögzített jelentésű kulcsszavakból és számokból álló egyenlőségek formájában adhatjuk meg. A nem specifikált formátum elemekre alapértelmezések lépnek életbe, amelyek lehetővé teszik stílus nélküli (de nem stílustalan) dokumentumok editálását. A példából látható, hogy a betű formátum egyes jellemzői külön is változtathatók. A stílus mezőben megjelenő menüben az itt definiált felhasználói kifejezések jelennek meg, (azok első hat betűje), tehát a konkrét példában magyarul.

Az Andra fa struktúrában ábrázolja a dokumentumokat. A dokumentumokban megjelenő elemek (fejezet, bekezdés, sor, szó, betű) eleve kínálkoznak a fa jellegű ábrázolásra. Az Andra dokumentumokban a formattálás egysége a stílus elem. Egy csomóponthoz tetszőleges számú elem kapcsolódhat. A stílus elemeket kétállású kapcsolók sorozataként ábrázolhatjuk, amelyeket minden csomópontra beállíthatunk, illetve törölhetünk. Az alapvető formattálás ezért csomópontok hozzáadásával, illetve törlésével történhetne, de ez a szint nem látható a felhasználó számára. Ha a felhasználó töröl egy szövegrészt, amely éppen egy csomópontnak felel meg (ez nem olyan valószinűtlen, már csak azért sem, mert a szöveg kiválasztása erre "rásegíti" a felhasználót, és lehetőleg mindig stílus elem egységekben invertálja a képet), akkor törlődik a megfelelő csomópont. Ha olyan szövegrészt választ ki, amely nem esik egybe egy meglévő csomóponttal sem, akkor új csomópont keletkezik. A csomópontok öröklik egymástól a nem specifikált attribútumokat, hasonlóan a blokk struktúrát tartalmazó program nyelvek érvényes-

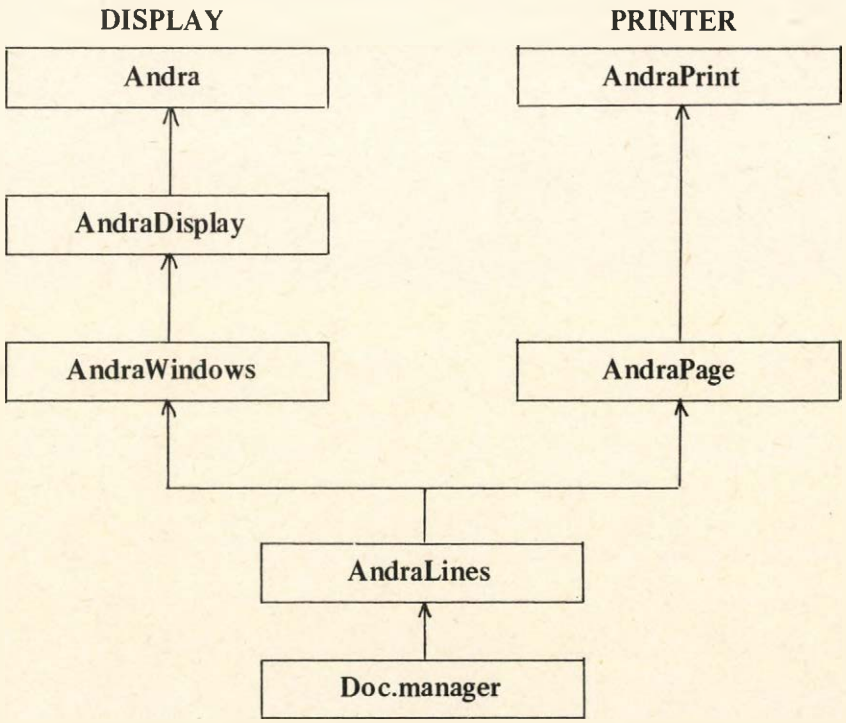
ségi szabályaihoz. A csomópont MODULA definiciója:

```
NodeDescriptor = RECORD
  up, left, right: POINTER TO NodeDescriptor;
  styleElements:   LONGSET;
  CASE type: NodeType OF
    TextNode: text: POINTER TO PieceDescriptor;
    SubTree:  down: POINTER TO NodeDescriptor;
  END; (*CASE*)
END; (*RECORD*)

PieceDescriptor = RECORD
  next,prev: POINTER TO PieceDescriptor;
  file: File;
  position: LONGCARDINAL;
  length: CARDINAL;
END; (*RECORD*)
```

Utóbbi definició (PieceDescriptor) a tényleges szöveg ábrázolását is mutatja. Eszerint a központi tárban csak egy leíró lista található és nem maguk a szöveg karakterek. Minden, a file-ban folyamatosan elhelyezkedő szövegdarabnak megfelel egy lista elem. A szöveg hosszára így semmi korlátozás sincs. A módszer hátránya, hogy a szekvenciális hozzáférés viszonylag lassúvá válhat, ha a szöveg nagyon feldarabolódik. A dokumentum lezárásakor az Andra rendszer folyamatos területre másolja az egy csomóponthoz tartozó szövegdarabokat, akár különböző file-okból is.

A dokumentumok megjelenítését a képernyőn a display manager, a nyomtatón az AndraPrint végzi. Ezek célszerűen ugyanarra a belső ábrázolásra támaszkodnak (4.21 ábra).



4.21 ábra A megjelenítést végző modulok kapcsolata

Az AndraDisplay modul kezeli a képernyőt, és az ablakokat, mint egészeket. Az ablakok belsején az AndraWindows modul végez műveleteket. Az AndraPage modul a nyomtató lapokra törlést, lapszámozást végzi. A sor orientált műveleteket végző AndraLines-t mindkét megjelenítő ág importálja.

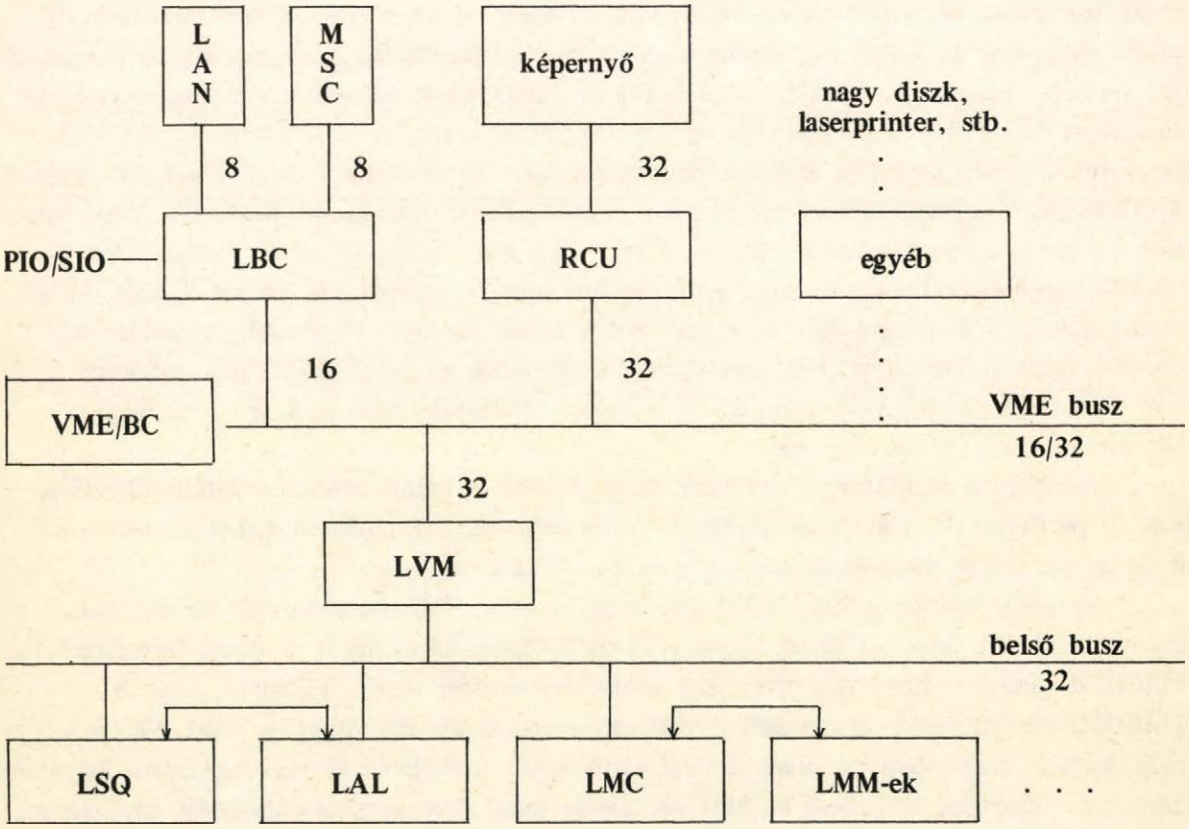
Az ANDRA jelenlegi verziója csak szöveges információk kezelésére képes. Az ábrákat külön kell elkészíteni (a DRAW nevű program segítségével). Az ANDRA viszonylag könnyen kiterjeszthető lenne grafikus elemek befogadására is.

5. A LILIPUTH RENDSZER RÉSZLETES ÁTTEKINTÉSE

5.1 A LILIPUTH HARDWARE

A LILIPUTH hardware architektúrális szempontból erősen hasonlít a LILITH-re. Az alapvető eltérések abban állnak, hogy egyrészt számos erőforrás kapacitása megnőtt (központi tár, pont-raszteres képernyő stb.), másrészt az architektúra igyekszik figyelembe venni a COSY szempontjait.

A LILIPUTH hardware elképzelései most vannak kialakulóban, végleges formája nem utolsósorban a beszerzési lehetőségeknek is függvénye. Ezért a következőkben egyes kérdésekre több lehetséges választ is adunk. Hasonló okból egyes kérdéseket fel sem teszünk. A LILIPUTH hardware felépítését az 5.1 ábra szemlélteti.



5.1 ábra A LILIPUTH hardware felépítése

A hardware fő elemei a 32 bites VME busz körül helyezkednek el. A VME busz vezérlő (BC) és az ábrán a VME busz felett elhelyezkedő elemek várhatólag készen vehetők át a COSY rendszerből. Ezek részletes ismertetése a "Supermicro" hardware-ről szóló ismertetésben és a rendszeresen megjelenő COSY kiadványokban találhatók, itt csak fő funkcióikat ismertetjük.

Az LBC (local board controller) lehetővé teszi két párhuzamos és soros vonal (PIO/SIO), a lokális hálózati csatoló (LAN) és egy háttértároló csatoló (MSC – mass storage controller) illesztését. A LILIPUTH-ban a PIO és a SIO elsősorban segédeszközök illesztését szolgálhatja, elvileg nem lehetetlen a billentyűzet és a mouse ilyen illesztése, de e pillanatban célszerűbbnek tűnik, ha ezeket külön illesztjük a belső buszra. A lokális hálózati illeszkedés tekintetében teljes mértékben a COSY-ra támaszkodunk, a LILIPUTH rendszerbe tetszőleges hálózati csatlakozás beilleszthető. Az MSC-t az alapvetően csak mentési célokra előirányzott floppy illesztésére használjuk, a diszk illesztésére a LILIPUTH igényeihez ez a megoldás elfogadhatatlanul lassú (a Z80 buszon keresztül).

Az RCU illeszti a pont-raszteres képernyőt. A LILIPUTH (az intézeti irányt is követve) 1024x1024-es nem interlace-es képernyőt alkalmaz, ami lényegesen jobb a LILITH képernyőjénél. Ennek a minőségi előrelépésnek súlyos ára van, itt ugyanis a LILITH-nél szükséges kb. 15 MHz-es frissítés helyett kb. 70 MHz-re van szükség. Ez azt jelenti, hogy ha az illesztés módja olyan lenne, mint a LILITH-nél, akkor az RCU elvinné a tárhozzáféréseknek mintegy 50%-át, ami nem elfogadható. A COSY-ban e pillanatban javasolt RCU megoldás szintén nem kielégítő. Itt ugyanis a képernyő vezérlését alapvetően egy NEC 7220 GDP chip végzi, amely képes ugyan bizonyos grafikai feladatok önálló ellátására, de külső hozzáférés számára már gyakorlatilag nem marad idő (és az is Z80 buszony keresztül). Mivel a LILIPUTH (a LILITH-hez hasonlóan) mikroprogrammal támogatott MODULA programokkal akarja a grafikát vezérelni (ami gyors és flexibilis egyszerre), a fenti megoldás teljesen elfogadhatatlan. Az általunk (a COSY felé is) javasolt megoldás ezért egyrészt abban áll, hogy a pixel térképet tartalmazó puffert a VME busz felöl 32 biten lehessen olvasni és írni, a képernyő felé pedig 32 biten olvasni (akár a NEC-chip segítségével is), másrészt pedig abban, hogy a puffert két részre osszuk, és a frissítés felváltva forduljon hol az egyik, hol a másik részhez. Ez utóbbi megoldás egy variációja lenne, hogy a puffert megkettőzzük, és amig az egyik blokkot a software a VME buszon keresztül módosítja, addig a másik blokkból történik a frissítés, majd a software jelzésére váltás történik.

A VME busz lehetőséget ad arra, hogy később minden nehézség nélkül illeszthessünk új perifériákat, például az esetleges server állomásként működő LILIPUTH-okhoz laserprintert, illetve különösen nagy kapacitású diszkeket.

Az LVM egység a LILIPUTH belső busza és a VME busz között teremt kapcsolatot. A belső busz 32 bites. Ugyancsak az LVM-re kapcsolódik a diszk interface. A LILIPUTH alapvető háttértára nagy kapacitású Winchester diszk. Tekintve, hogy a LILIPUTH alapvetően 1 felhasználós rendszer, ezért fontosabb, hogy a diszk elérése gyors legyen, mint, hogy a diszk átvitel alatt egyéb műveletek is végrehajthatók legyenek. Ezért nem illesztjük a diszket az MSC-re, amely lassú, bár szuverén illesztést ad, hanem a megfelelő (valószinűleg SASI) csatolóval ellátott diszket közvetlenül a belső buszra illesztjük. A diszk kezelését a LILITH-hez hasonlóan mikroprogram is támogatja. A cél,

hogy a diszk fizikai sebességéből minél kevesebbet veszítsünk el az illesztés által.

Az LSQ egység tartalmazza az MCU-t, a mikrotárat, és ide kapcsolódik a fejlesztő rendszer is. A mikroutasítások végrehajtását vezérlő MCU az AM2910-es sequencer köré épül, a mikroutasítások hossza előreláthatólag 48 bit. A mikroprogramtár maximális mérete (a 2910-nek megfelelően) 4K. A LILIPUTH-on lehetővé kívánjuk tenni a felhasználói mikroprogramozást is. Erre több út is kínálkozik. Az egyik, hogy a mikroprogramtárat ROM helyett RAM-ban helyezzük el. Ennek hátránya a nagyobb tokszám, és az, hogy meg kell oldani a töltést diszkről vagy EPROM-ból. A következő lehetőség, hogy a mikrotár egyik fele ROM, a másik fele RAM. Ennek hátránya, hogy csak a mikrotár fele szabad. A harmadik megoldás, hogy a mikrotár címtartomány egy része a makrotárat címzi. Ennek hátránya, hogy lassú, viszont kíváló belövési eszköz egy későbbi beégetéshez.

A LAL egység tartalmazza az ALU-t, az expression stack-et és a Barrel shiftert. Az aritmetikai egység a LILIPUTH-ban is bitszeletelt mikroprocesszorra épül. Az AM2901 mellett fölmerül a 2903, a 29116 és különösen a 2901-quadro, ami egy tokba épített 4 db 2901-et jelent. A 16 bites LILITH-től való újabb eltérés, hogy a LILIPUTH aritmetikája 32 bites (tehát 8 db 2901, vagy 2 db 2901-quadro szükséges hozzá). Ennek megfelelően az expression stack-et és a Barrel shiftert is 32 bitre kell kiépíteni. Ezek funkciója egyébként nem változik. Megjegyezzük, hogy a memória címregisztert – amelyet hardware-ben kell kiépíteni – nem tervezzük 32 bitesre, jelenleg úgy tűnik, hogy 24 bites kiépítés (16 Megység címzésére alkalmas) minden gyakorlati igényt kielégít.

A memória vezérlését végzi az LMC, az egyes memória blokkokat az ábrán az LMM jelképezi. A LILIPUTH tára előreláthatólag 64 Kbites dinamikus RAM-okból épül fel. Szervezése olyan, hogy 32 biten olvasható és írható. A LILITH 64 biten tudja olvasni a tárat az RCU és az IFU számára, amihez képest a 32 bites olvasás csökkenést jelent. Ezt ellensúlyozza viszont az RCU fent ismertetett eltérő szervezése, és az egységes kezelésből fakadó előnyök. (A gépi szó méretének megnövekedéséből fakadó előnyökről külön szólunk a gépi architektúra c.pontban.) Az LMC a memória vezérlése mellett képes ellátni az utasítás kiolvasást (IFU), valamint a billentyűzet és a mouse illesztését is.

Felül kell vizsgálni a LILITH megszakítási stratégiáját (ahol, ha csak lehet, kerülik a megszakítást). Nyílván megszakítással működik az óra, célszerűen a floppy és a hálózat is. A többi periféria esetében ezt egyenként kell mérlegelni. Különösen érdekes kérdés a pixel műveletek megszakíthatósága. Ha ezt megengedjük, akkor ez a látható kép összetöredezéséhez vezethet (ami nyilván elfogadhatatlan), ha egyáltalán nem engedjük meg, akkor nagyon sokáig marad megszakíthatatlan a gép.

Az 5.1 ábrának a VME busztól fölfelé eső részeinek kártya kiosztása a COSY rendszerből ismert, hosszú távon kb. 10 kártyáig mehet el. Az alsó rész egységei a terv szerint egy-egy kártyát jelentenek. 1 LMM kártyára előreláthatólag 1/2 MByte tár fér el, az LMM kártyák száma elvileg tetszőlegesen növelhető (gyakorlatilag több tényező is befolyásolja a megengedhető maximális tárat).

5.2 A LILIPUTH GÉPI ARCHITEKTURA

A LILIPUTH gépi architektúrája lényegében követi a LILITH-ét (tehát hasonló a regiszterek szerepe, megmarad egy expression stack és hasonlóan a címzési módok). Az eltérések fő oka a hardware-ben bevezetett változásokra vezethető vissza, de megfontolás tárgyává teszünk néhány javítást is.

5.2.1 Címzés

A legfontosabb eltérések abból fakadnak, hogy a LILITH 16 bites, a LILIPUTH 32 bites. Ez azt jelenti, hogy a futás közben keletkező abszolút címek tárolása nem jelent többé nehézséget, a "tisztességesen" címezhető tár terület mérete gyakorlatilag korlátlan. Ezért megszűnik a központi tár kényszerű felosztása alsó (64 K) és felső részre, az egész tár egységesen kezelhető. Ez bizonyos utasításokat (LXFW, SXFW, MOVF) szükségtelenné tesz. A LILITH-en két szavas címmel paraméterezett utasítások (BBLT stb.) paraméterezése módosítható úgy, hogy azonnal bit címet vesznek át (egyetlen szóban elfér).

Fölmerül az a lehetőség is, hogy a LILITH-ben preferált szócímzés mellett kisebb egységek (byte, félszó) címzését is támogassuk. (A támogatás történhet hardware segítséggel , vagy tisztán mikroprogrammal.) Ennek egyik előnye lenne, hogy feloldana olyan korlátozásokat, amelyek a byte cím előállításának hiányából fakadnak (pl. string típusú tömb eleme nem adható meg VAR típusú paraméter aktuális paramétereként). A másik előny az lenne, hogy optimalizálni lehetne korlátozott értéktartományú változók ábrázolását (jelenleg pl. a BOOLEAN típusú változók is egy szót foglalnak le). A kisebb egységek címzésének bevezetése mindenesetre elég mélyreható módosítás az utasításkészletben.

Megváltozik az F regiszter szerepe, amely a LILITH-ben a kód frame eltolt címét tartalmazza, és amelyhez az utasítás kiolvasó minden lépésben hozzáadja a byte-relatív PC-t. A LILIPUTH-on módunk van arra, hogy a PC abszolút címet tartalmazzon, ezért az F regiszternek csak eljáráshíváskor van szerepe, amikor a kód frame elején lévő, az eljárások belépési pontjait a kód frame kezdetéhez képest byte-relatívan tartalmazó ugrótábla kiértékelése történik. Ez a cím a G[0]-ban mindenképpen megtalálható, ezért szükség esetén az F regiszter elhagyható. Elvileg az eljárások belépési pontjait is tárolhatnánk abszolút cím formájában, de ebben az esetben a kód eltolása futás közben körülményessé válik, ami ugyan az esetek nagy részében nem hátrány, de bizonyos esetekben súlyos megkötés lehet. (Erről még szólunk a következő pontban.)

5.2.2 Javaslatok a tárfeldarabolás kiküszöbölésére

A tár feldarabolása a dinamikusan keletkező és megszűnő, változó méretű tárterületek által jön létre. A LILITH-en (ld. a LILITH gépi architektúra c. fejezetet) a következő fő tárterület fajtákat különböztetjük meg:

- Adat frame. A (külön fordított) modulok globális adatait tartalmazza. Töl-

téskor keletkezik, futás közben tartalma változik, mérete nem.

- Kód frame. A (külön fordított) modulok utasításait tartalmazza. Töltéskor keletkezik, futás közben se tartalma, se mérete nem változik.
- Korutin frame. A korutinok féldinamikus (stack) és dinamikus (heap) adatait tartalmazza. Keletkezhet töltéskor is és futás közben is. A korutin frame mérete futás közben nem változik, de ezen belül a stack és a heap tartalma és mérete egyaránt változik. Eközben a stack mindig összefüggő marad, a heap tetszőlegesen feldarabolódhat.

E felsorolásból látszik, hogy egy adott program (egy töltési egység) futása során csak a korutin frame okozhat tárfeldarabolódást, több program esetén viszont az összes tár fajta. A korutinok által okozott tárdarabolódás kétféle:

- A korutinok dinamikus keletkezése és megszünése által korutin frame szinten.
- A korutinokon belül, a heap-ből foglalt és felszabadított változó hosszúságú dinamikus adatok álta heap szinten.

Az előző fejezetben láttuk, hogy milyen megoldásokat alkalmaz a MEDOS a fenti problémák enyhítésére. A program szinten a szuper-eljárás stack-szerű szervezése maradéktalan megoldást jelent. A korutin szintű feldarabolódás elleni védelmet a MEDOS visszavezeti az előzőre, amennyiben az egy programhoz (pontosabban az egy közös (shared) szinthez) tartozó korutinok frame-jeit a szuper-eljárás stack-jéhez tartozónak tekinti. Ez az eljárás tehát megakadályozza a különböző program szintek által foglalt korutin frame-ek keveredését, de egy szinten belül nem nyújt semmiféle védelmet. A heap szintű feldarabolódást a MEDOS egyáltalán nem kezeli (vagyis a felhasználóra bizza), még segédletet se sokat ad. Ez a stratégia a LILITH architektúrája mellett bizonyára optimálisnak tekinthető, és gyakorlatilag is sikeres. Mégis célszerűnek tűnik megvizsgálni, hogy nem lehetne-e a LILIPUTH architektúrát úgy kialakítani, hogy a fentinél igényesebb stratégiákat is támogasson. Most csak megemlitjük a lehetséges megoldásokat, megvalósításukat nyitva hagyjuk.

A tár feldarabolódásának kiküszöbölésére alapvetően két fő utat látunk:

1) A tárfoglalás korlátozása kötött méretű egységekre. Ennek a megoldásnak szélsőségesen megvalósított változata az lenne, hogy az egész tárat lapokból felépítve képzeljük el úgy, hogy a lapok egymáshoz láncolását mikroprogram (esetleg hardware is) támogatja. Ebben az esetben a feldarabolódás egyszer s mindenkorra megszűnik. Hátrány, hogy a tár elérése lasabbá válik, valamint, hogy az eljárás meglehetősen bonyolult. Kevésbé radikális verziók megvalósíthatók a software-ben (ld. később), de ezek már valószinűleg nem befolyásolják az architektúrát.

2) A másik megoldás, hogy kiküszöböljük a futás közben keletkező abszolút címeket. Ezt a megoldást virtuális címzésnek is nevezhetjük. Egy ilyen megoldás lehetővé teszi, hogy megfelelő feltételek fennállása esetén (pl. nincs folyamatban lévő I/O) tár tö-

mörítést hajtsunk végre. Ez ugyan elég lassú megoldás, de nem terheli a gépet állandóan (mint a lapozás), csak bizonyos pontokon.

A kód frame-re vonatkozóan a virtuális címzés követelménye eleve megoldott, már láttuk, hogy a kód tetszőlegesen eltolható. Az adatokra vonatkozóan (akár az adat, akár a korutin frame-ben helyezkednek el) azonban meg kell oldani, hogy a futás közben keletkező címek valamilyen vonatkoztatási alaphoz képesti relatív értelmezést kapjanak. Lehet egy vagy több ilyen vonatkoztatási alap.

A legkézenfekvőbb megoldás, ha a korutin frame kezdetét (P regiszter) tekintjük vonatkoztatási alapnak [Jaco82]. A központi tár nevezetes helyén (pl. a DFT tábla mögött) tároljuk a korutinok címeit (nevezzük CFT táblának). A virtuális cím két részből áll: a korutin indexből és a korutin frame-en belüli címből. A virtuális cím ábrázolása történhet duplaszóban, de kihasználhatjuk azt a tényt, hogy a tár címregisztere amúgy is 24 bites, ezért a virtuális címet egyetlen szóban is tárolhatjuk:

8 24

A 0 index érték célszerűen a virtuális címképzés kikapcsolását jelzi, a tényleges korutin index az 1...255 tartományba esik. Ez azt jelenti, hogy max. 255 korutin létezhet egyszerre a tárban, ami nem tűnik súlyos korlátozásnak. A virtuális cím képzésének meggyorsítása érdekében célszerű a P regisztert "kettéhasítani": a PI és PA regiszterekre, ahol PA tartalmazza az aktuális korutin frame címét, PI pedig az indexét. Ezután bevezetünk egy új utasítást, amely fizikai címből képez virtuális címet (VIRT).

A VIRT utasítás működése a fent bevezettet két új regiszter (PI és PA) segítségével: az expression stack tetején lévő (fizikai) címből levonja PA értékét (modulo 2↑24 – a "↑" szimbólum jelentse a hatványozást), és a bal szélső byte-ba beírja PI értékét (+PI*2↑24):

```
stack_top:= (stack_top - PA) MOD 2↑24 + PI*2↑24
```

Elvileg bevezethetnénk egy inverz utasítást is, amely virtuális címből állít elő fizikai címet, de ennél sokkal egyszerűbb, ha megváltoztatjuk azoknak az utasításoknak az értelmezését, amelyek az expression stack tetejét címnek tekintik (LSW, SSW stb.). Ezek az utasítások ellenőrzik a stack tetején lévő cím bal szélső byte-ját (DIV 2↑24); ha az 0,, akkor működésük megegyezik az eddigivel. Ha ez az érték nem 0, akkor a szó alsó 24 bitjét (MOD 2↑24) hozzáadják az index által jelölt CFT elemhez:

```
IF (stack_top DIV 2↑24) # 0 THEN
  stack_top:= (stack_top MOD 2↑24) + CFT[stack_top DIV 2↑24]
END
```

A virtuális címzés bevezetése megváltoztatja a TRANSFER utasítás paramétereinek értelmezését, amelyek korutin frame cím helyett ezután korutin frame indexet jelentenek.

Példaként tekintsük az alábbi mintaprogramot:

```
 1 0004  MODULE CodeTest;
 2 0004
 3 0004  VAR Global: INTEGER;
 4 0004
 5 0004  PROCEDURE P(par1: INTEGER; VAR par2: INTEGER);
 6 0004  BEGIN
 7 0008    par1:= par2;
 8 0008    par2:= Global;
 9 000C  END P;
10 0010
11 0010  BEGIN (*modul initialization*)
12 001B    P(Global,Global)
13 001E  END CodeTest.
```

Ebben a programban a LILITH-en a P eljárás hívásának (12.sor) a következő utasítás sorozat felel meg:

```
LGW3
LGA 3
CL1
```

Az LGW3 utasítás az expression stack-re tölti az adat frame 3.szavának (Global) tartalmát, az LGA 3 utasítás pedig a címét. Ez megfelel annak, hogy a hívott eljárás (P) első paramétere érték szerinti, a második pedig név szerinti (VAR). A CL1 utasítás meghívja az eljárást. Az LGA 3 utasítás hatására tehát abszolút cím keletkezik: G regiszter tartalma + 3 értékkel. Ha virtuális címzést alkalmazunk, akkor az abszolút cím helyett virtuális címet kell előállítani. Ez az újonnan bevezetett VIRT utasítás segítségével könnyen megtehető:

```
LGW3    (változatlan)
LGA 3   [G] + 3
VIRT    (([stack top] - [PA]) MOD 2↑24) + (PI * 2↑24)
CL1     (változatlan)
```

A P eljárásban a par2:= Global (8.sor) utasításnak a következő utasítás sorozat felel meg:

```
LLW5
LGW3
SSWO
```

Az LLW5 utasítás az expression stack-re tölti az L regiszterhez képest 5.szó tartalmát, ahová az eljárásba való belépéskor par2 tartalmát mentette le (erről tudjuk, hogy cím – VAR paraméter). LGW3 ismert módon Global tartalmát tölti, az SSWO utasítás pedig az expression stack tetejét (Global) az alatta lévő címre tárolja. Virtuális címzés esetén az SSWO utasítás módosított értelmezése biztosítja, hogy ugyanez az utasítás sorozat is helyesen működjék.

Ha az itt szemléltetett virtuális címzést alkalmazzuk, akkor a megfelelő feltételek esetén tártömörítést hajthatunk végre úgy, hogy a DFT és CFT táblákat megfelelően módosítjuk.

A LILITH-en és a LILITH interpreteren készített statisztikák alapján [Jaco82, Bösz83] számos olyan utasítás megszüntethető (LSW7-LSW15, SSW7-SSW15 stb.), amelyek eleve optimalizálási célt szolgálnak, de nem hozták meg a kívánt eredményt. Így sok szabad operáció kódot nyerhetünk.

5.3 A LILIPUTH ALAPSOFTWARE

A LILIPUTH alapsoftware első változatánál fontos cél, hogy minél hamarabb működő rendszerhez jussunk. Ezért az első lépésben erősen támaszkodunk a LILITH software-re. Feltétlenül változtatni kell bizonyos helyeken az architektúrák különbözősége miatt, és tervezünk bizonyos javításokat.

5.3.1 A fordító

Az eredeti LILITH fordítót először is módosítani kell a legújabban megjelent apró nyelv módosítások miatt [Wir84]. (Mint már említettük, egyelőre a leghatározottabban tartózkodunk a MODULA NYELV bármiféle módosításától, amig csak ki nem kristályosodik egy új MODULA-dialektus, vagy akár egy új nyelv.)

Mindenképpen módosítani kell a fordítót a LILIPUTH új architektúrája miatt. Az első számú ilyen módosítás az áttérés a 16 bites szavakról a 32 bites szavakra. Ez önmagában nem nagy változtatás, de számos vonzata van. A legnagyobb nehézség az, hogy bizonyára az egész fordítót át kell nézni, hogy megtaláljuk azokat a helyeket, ahol a fordító a 16 bites szóhosszt implicite kihasználja (ez ugyan csak helytelen programozói stílus esetén fordulhat elő, de elvileg nem zárhatjuk ki). Felmerülnek olyan kérdések is, mint az újonnan bevezetett LONGCARD, LONGINT és LONGREAL standard típusok értelmezése. Módosítani kell a konstansok kiértékelését. (Ez sajnos az összes menetre kiterjed.)

További módosítást jelent, ha bevezetjük a szónál kisebb egységek (byte, félszó) címzését. Elemi adatok esetén a szónál kisebb tárolási egységnek nincs jelentősége. Struktúrált adatok (rekordok és tömbök) esetében azonban a tömörített ábrázolás igen komoly nyereséget jelenthet. A tömöritett ábrázolás hátránya általában az elérés lassúsága; de ha a LILIPUTH architektúra támogatja a byte-ok és félszavak elérését, akkor a tömörítés már bizonyára kifizetődő. A fordító ilyen irányú módosítása elég mélyreható, de nem túl bonyolult.

A virtuális címzés esetleges bevezetése újabb módosítást igényel. Az előző részben bemutatott javaslat egyik fő előnye, hogy a fordító módosítása viszonylag egyszerű. Ennek szemléltetésére tekintsük a fordító 4.menetének azt az eljárását, amely egy objektum címét az expression stack-re tölti (ld. a következő oldalon).

A globalMod, externalMod és localMod esetben, ha feltételezzük, hogy az eredeti utasítások (LGA, LEA, LLA) jelentése nem változik, akkor az új (már említett) VIRT utasítás segítségével egyszerűen Emit(VIRT)-tel kell kiegészíteni az utasítás kibocsátást.

Az absolutMod (aminek egy MODULA programban a

```
VAR VarAtAbsAddr [40H]: CARDINAL;
```

formában leírt abszolút cím felel meg) kezelése triviális, hiszen, ha a felső byte értéke 0, akkor a virtuális címzés kikapcsolódik, tehát 24 bitnél nem nagyobb szám esetén abszolút

cím keletkezik.

A többi módot itt nem érinti a változás, mert abban a pillanatban, amit ez az eljárás tükröz, a bázis címnek már helyesnek kell lennie (akár virtuális, akár nem).

Az itt bemutatott módosítás ugyan nem kimeritő, de látható, hogy viszonylag egyszerű technikáról van szó.

```
PROCEDURE LoadAddr(VAR fat: Attribut);
  (* Generate code to load the address of
     fat onto the expression stack *)
  VAR laddr: CARDINAL;
BEGIN
  WITH fat DO
    CASE mode OF

      localMod, globalMod, externalMod:
        IF addr<=255 THEN
          CASE mode OF
            globalMod:
              Emit(LGA); Emit(addr);
              INC(loadCount)

          | externalMod:
              Emit(LEA);
              MarkVarAddr(fat);
              INC(loadCount)

          | localMod:
              Emit(LLA); Emit(addr); INC(loadCount)
          END
        ELSE
          laddr := addr-255; addr := 255;
          LoadAddr(fat); UAddToTop(laddr)
        END

    | absolutMod:
        EmitLI(addr)

    | stringTemplateMod:
        (* imported strings are copied into the actual module*)
        IF addr=0 THEN EmitPacked(LGW, 2);
        ELSIF addr<=255 THEN Emit(LSTA); Emit(addr);
        ELSE EmitPacked(LGW, 2); UAddToTop(addr)
        END;
        INC(loadCount);

    | addrLoadedMod:
        UAddToTop(addr)

    | indexMod:
        Emit(UADD); DEC(loadCount)

    | doubleIndexMod:
        Emit(COPT); Emit(UADD); Emit(UADD); DEC(loadCount)

    ELSE (*illegalMod, constantMod, procedureMod, stringConstMod,
           byteIndexMod, loadedMod, doubleConstMod*)
        CompilerError
    END;
    mode := addrLoadedMod; addr := 0
  END;
  IF loadCount>16 THEN Error(204) END
END LoadAddr;
```

A LILITH-en jelenleg futó fordító felépítése tükrözi valamelyest kialakulásának történetét. Az első változat a PDP/11-en 7 menetes volt, amire eljutott a LILITH-re 3 menetet "lefogyott". Valójában ez még mindig sok, különösen a LILIPUTH-ra, ahol bőségesen áll rendelkezésre tár. Ezért a fejlesztés egy későbbi szakaszában célszerű lenne a fordítót teljesen újraírni, 1 vagy 2 menetes változatban. Az 1 menet ellen (nyilvánvaló előnyei mellett) az szól, hogy a post-hivatkozások kezelésére korlátozásokat kell tenni.

5.3.2 A LILIPUTH operációs rendszer

Az operációs rendszer bizonyos részei szükségképpen szorosan kötődnek a gépi architektúrához, ezért ezeket teljesen újra kell írni. A MEDOS-tól számos egyéb eltérést is tervezünk, részben hatékonysági okokból, részben mert a MEDOS helyenként elavult elveket valósít meg.

5.3.2.1 Felhasználói interface és programok indítása

A MEDOS felhasználói interface-ének funkcionális része, tehát, hogy a <parancs értelmezés – program végrehajtás> végtelen ciklust hajtja végre, első lépésben elfogadható. Ha később felmerül az az igény, hogy egyszerre több aktivítást indíthassunk, akkor természetesen ez is módosul. A felhasználói interface tényleges megvalósítása azonban semmiképpen sem elfogadható. Ez ugyanis szintén magán viseli előtörténetének nyomait, és a PDP/11 operációs rendszereiből örökölt elemek ismerhetők fel rajta. Tekintve a LILIPUTH kiemelkedő interaktív adottságait, nem tartjuk kívánatosnak az irógépszerű, sok felhasználói gépelést igénylő interface-t. Ezért a LILIPUTH operációs rendszerében a menüzést és a window-zást a legalacsonyabb szinten is megjelenő lehetőségek közé kívánjuk sorolni. A menüzés igényének felmerülése felhívja a figyelmet a LILITH rendszer egyik szépséghibájára. Ez abban áll, hogy a MODULA fordító által generált "object" file-okon [Lili82] nem látszik, hogy az önálló futásra képes modulból keletkezett, vagy csak importálni lehet. Ez a koncepció tulajdonképpen hiányzik magából a nyelvből. Mindenesetre, ha egy rendszer ki akarja teríteni egy menün az indítható programokat, akkor az ilyen megkülönböztetés hiánya a menüt nagyon telezsúfolhatja. Ez ellen esetleg elfogadható védelmet nyújt, ha lehetővé tesszük a felhasználó számára, hogy a csak importra szánt object-jeit alkönyvtárakban tárolja.

A programok töltésére, legalábbis első lépésben, átvesszük a MEDOS szuper-eljárás koncepcióját. Kérdés viszont, hogy helyes-e a szerkesztést kizárólag töltéskor végezni. A LILITH interpreteren [Bösz83] igen jó tapasztalatokat szereztünk a külön szerkesztéssel és gyors töltéssel. Ezért az egyértelműen standard alapszolgáltatásokat nyújtó programok esetére célszerű fenntartani a külön szerkesztés lehetőségét. Ehhez viszont a gyors töltőt ki kell egészíteni azzal a képességgel, hogy szerkesztetlen program (object) indításakor elő-

ször a szerkesztő töltőt töltse be (az interpreteren ezt jelenleg IDOS szinten oldottuk meg).

5.3.2.2 Párhuzamosság és tárkezelés

Mint már ismertettük, a MEDOS legújabb verziója nyújt bizonyos támogatást a (kvázi-)parallel működések támogatásához. Ezt a támogatást azonban elégtelennek tartjuk. A párhuzamosság kérdésében elég széles körű tapasztalatok állnak mögöttünk, és erről többször írtunk is már [Bösz78, 81]. A párhuzamosság két alapvető kérdése a szinkronizáció és a tárfoglalás. Ez utóbbit ugyan általában nem sorolják ebben a témakörbe, de a párhuzamosság jelenléte fokozott követelményeket állít a tárkezelés elé is.

A szinkronizáció kérdésében a MEDOS alapvető fogyatékossága, hogy a Processes modul által definiált SIGNAL típuson végezhető műveletek túl primitívek. A döntő hiányosság, hogy egy processz egy időben csak egyetlen SIGNAL-ra várakozhat. (Ennek bizonyára részben szintén történeti okai vannak. Az eredeti MODULA nyelvben [Wirt77] és sok hasonló nyelvben, mint a Concurrent Pascal, Mesa stb. [Bösz81] a szinkronizációs műveletek kiadása ugyanis a fordító ellenőrzése alatt történik, így a WAIT utasítás csak monitorban [Bösz81], vagy annak ekvivalenseiben adható ki. Mivel egy processz egyszerre csak egy monitorban tartózkodhat, így logikus, hogy egyszerre csak egy SIGNAL-ra várakozzék. A MODULA-2 nyelvből tudatosan kihagyták a szinkronizációt, így a fordító nem képes a fent leírthoz hasonló ellenőrzéseket végezni. Ezt a hátrányt ellensúlyozza a megnövekedett szabadság, amit éppen úgy használhatunk ki, hogy a WAIT utasítást a processz bárhol kiadhatja, és akárhány SIGNAL-ra.) Ennek a kérdésnek a teljesen általános megoldása, hogy egy process SIGNAL-oknak (vagy azok valamilyen ekvivalensének) tetszőleges logikai kombinációjára várhat. Ennek megvalósítását túlzottnak tartjuk. Kiemelt jelentőségű az az eset, amikor egy processz több SIGNAL közül egyre (OR) vagy az összesre (AND) vár. Ezek közül is fontosabb az első eset, mert az AND kapcsolat (amely (amely amúgy is ritkább) kiváltható ciklusban kiadott OR jellegű várakozással, míg az igen gyakori OR csak ciklusban kiadott time-out-os várakozásokkal váltható ki, ami csúnya és rossz hatásfokú megoldás. Ezért a MEDOS ütemező minimális bővítése az OR jellegű várakozás megvalósítása. Ennek definiciója a következő lehet.

```
WaitOr (Signals: ARRAY OF SIGNAL;
        Supress: WORD; VAR Result: WORD);
```

A Signals nevű tömb tartalmazza azokat a SIGNAL-okat, amelyekre a kiadó processz várni kíván. A Supress nevű paraméter lehetővé teszi, hogy a hívó e SIGNAL-ok közül egyeseket kitiltson a várakozásból (ez a SIGNAL-ok tárolását könnyítheti meg). A Result nevű paraméter annak a SIGNAL-nak a sorszámát tartalmazza, amely aktivizálta a processzt.

A következő modul a WaitOr hívására ad példát.

```
MODULE User;

IMPORT SIGNAL, WaitOr;

TYPE
  Sigs = (sig1, sig2, sig3);
  SigSet = SET OF Sigs;

VAR
  S: ARRAY Sigs OF SIGNAL;
  Sup: SigSet; Res: Sigs;

BEGIN
  WaitOr( S, 0, Res);      (*1. hivás*)
  CASE Res OF
   sig1: (*...*)   |
   sig2: (*...*)   |
   sig3: (*...*)
  END;

  Sup:= SigSet{sig2};
  WaitOr( S, Sup, Res);    (*2. hivás*)
  CASE Res OF
   sig1: (*...*)   |
   sig3: (*...*)
  END;

  Sup:= SigSet{};          (*WaitAnd szimulációja*)
  REPEAT
    WaitOr( S, Sup, Res);
    INCL(Sup, Res);
  UNTIL Sup = SigSet{sig1..sig3};

END User;
```

Az első hivás vár az összes lehetséges SIGNAL (sig1, sig2, sig3) közül az elsőre, a második sig2-t figyelmen kívül kívánja hagyni. A mintaprogram bemutatja az összes SIGNAL-ra való várást is.

A dinamikus tárkezelés vonatkozásában eleve egyszerűsödik a helyzet a LILITH-hez képest az egységes címzési lehetőség miatt. Ezért a korábban ismertetett "Frames" nevű modul teljes egészében fölöslegessé válik. A "Heap" nevű modul stratégiája jelenleg a lehető legegyszerűbb, és egy kevés párhuzamosságot tartalmazó rendszerben kielégítő. További szolgáltatásként bevezethető egy olyan modul, amely software-es lapozást tesz lehetővé, tehát rögzített méretű cellákat képes összefűzni, és az interface-en összefüggő területként mutatni. Ilyen algoritmust már ismertettünk [Bösz79]-ben. Célszerű lehet a cellakezelés meggyorsítása olyan mikroutasítás bevezetésével, amely segíti a cellahatárok közötti váltást.

A "Heap" által nyújtott általános, és a rögzített cellamérettel dolgozó algoritmus együttesét helyettesítheti esetleg egy olyan algoritmus, amely a kettő közötti kompromisszumot valósít meg, például mindig a kért tármennyiséghez legközelebb eső 2 hatvány méretű tárat foglal. Ez csökkenti a feldarabolódás valószínűségét, és egyszerűen megvalósítható. Hátránya, hogy azért a feldarabolódást nem zárja ki.

A javasolt virtuális címzés megvalósítása újabb lehetőségeket nyit a tárkezelés előtt, hiszen ekkor a különböző frame-ek (adat, kód és korutin) bizonyos feltételek esetén eltolhatók futás közben is. Ezt a lehetőséget az operációs rendszer többféleképpen is kihasználhatja. Ennek legegyszerűbb módja, ha csak arra használjuk ki, hogy ha egy korutin megszűnik, akkor tártömörítést hajtunk végre. További lépés, ha lehetővé tesszük, hogy ha egy processznek betelik a dinamikus területe (stack + heap), akkor valamilyen

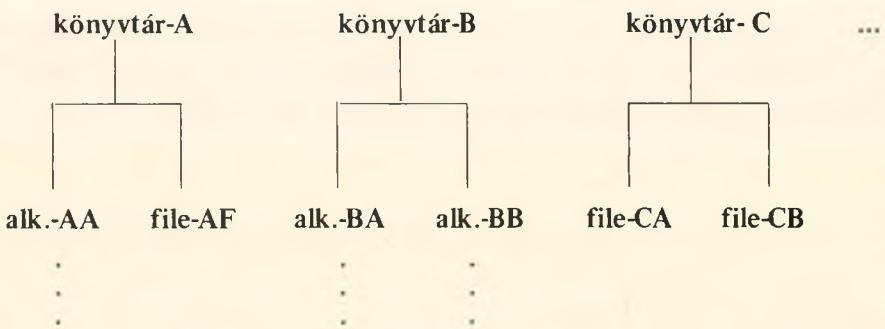
"mentő" akció kerül végrehajtásra. Ebben az esetben nem mindegy, hogy a stack telik-e be, vagy a heap. A stack betelése esetén mindenképpen tömörítést kell végezni, de a heap esetére elképzelhető olyan megoldás is, hogy minden processznek van egy kisegítő területe egy közös rendszerterületen belül, ahonnan szintén lehet dinamikus tárat foglalni (NEW-val). A virtuális címzés által elvileg arra is lehetőség nyilik, hogy egy processzt ideiglenesen háttértárolón tároljunk, és később visszatöltsünk. Ez azt jelenti, hogy a LILIPUTH architektúra nem zárja ki tetszőlegesen igényes, több felhasználós (time sharing) rendszer hatékony implementációját, mégha ez egyelőre nem is cél.

5.3.2.3 File rendszer

A MEDOS file- és diszk-rendszere néhány nagyon szellemes megoldása ellenére jelentős hátrányokkal rendelkezik. Egyrészt nem kielégítő az egyszintű katalógus rendszer, amely az összetett file neveket csak tárolja, de nem kezeli. Ezenkívül elvi okokból is lassú a file-ok kiolvasása. Ezért a LILIPUTH-on a MEDOS-étól erősen eltérő file rendszert alakítunk ki, a következő elvek szerint:

1) A teljes file rendszer felületet nem kívánjuk készülék függetlenné tenni. A teljes készülék függetlenség ugyanis egyrészt lassítja a hívás mechanizmusát (formális eljárásokon keresztül), másrészt bár szép, de nem igazán szükséges. Ennél fontosabb, hogy meghatározzuk a műveleteknek azt a halmazát, amely minden készülékre értelmes és szükséges. A továbbiakban ezért a file rendszert lényegében azonosítjuk a diszk rendszerrel.

2) A file rendszer lehetővé teszi könyvtárak és alkönyvtárak létrehozását. A könyvtárak mellérendelt viszonyban vannak egymással, az alkönyvtárak egy könyvtáron belül hierarchikus rendszert alkotnak:



A könyvtárak elsődleges célja, hogy a közös diszken osztozó felhasználókat elkülönítsék, az alkönyvtárak pedig ezen belüli struktúrálást tesznek lehetővé. A file-ok, alkönyvtárak és könyvtárak azonosítására programból egy egyszerű, kódolt mechanizmus áll rendelkezésre, a file rendszerből exportált típusok segítségével, a következő jelleggel:

```
TYPE
  SysLibs   = (opsys, compiler, linker, debugger);
  UserLibs  = [1..300];
  SubLibs   = [1..600];
  Files     = [1..10000];
  FileId    = RECORD
                LibNo : UserLibs;
                SubNo : SubLibs;
                FileNo: Files;
              END;
  SysFileId= RECORD
                LibNo : SysLibs;
                SubNo : SubLibs;
                FileNo: Files;
              END;
```

A képernyő előtt ülő felhasználó felé történő azonosítással a file rendszer nem foglalkozik. Ha a felhasználói interface elég következetesen tudja megvalósítani a menüzést, akkor ez általában implicite történik. Explicit azonosításra javasolható például a MEDOS-ban is alkalmazott minősített azonosító szerű megoldás.

3) Minden file-nak, alkönyvtárnak és könyvtárnak meghatározott típusa van. Ez lehetővé teszi, hogy a rendszer ennek alapján ellenőrízze, hogy egy adott művelet kiadható-e vagy sem. A megfelelő műveletek kiadhatók nemcsak egy file-ra, hanem nagyobb egységre is (pl. lehetséges egy egész alkönyvtár "fordítása", aminek értelmezése, hogy az alkönyvtárban szereplő összes file-t le kell fordítani).

4) A file rendszer az egyes file-ok elérésére gazdag választékot ad, exportált eljárások formájában. Ezek lehetővé tesznek szekvenciális és direkt elérést. Az elérés egysége tetszőleges hosszúságú blokk. Maga a file rendszer nem végez pufferelést, ezt a felhasználóra hárítja. Az operációs rendszer viszont nyújt egy modult, amely a tipikus elérési egységek kezelését (byte, szó, szektor) és pufferelését elvégzi. Ez lehetővé teszi a felhasználó számára, hogy válasszon az egyszerű, de lassabb, vagy a saját maga által kezelt, de gyors pufferelési technikák között.

A file rendszert ennél részletesebben nem specifikáljuk addig, amig nem ismerjük annak a háttértárnak az adatait, amelyre implementáljuk. A file rendszer jósága ugyanis elsősorban nem elméleti, hanem gyakorlati kérdéseken múlik.

ZÁRSZÓ

Tanulmányunkban felvázoltuk egy meglehetősen ambiciózus kutatás és fejlesztés terveit. Nagyvonalúan mellőztük a gyakorlati feltételeket, nem szóltunk pénzről, határidőkről és egyéb fontos kérdésekről. Ez nem azt jelenti, hogy ezeket nem tartjuk lényegesnek, inkább azt, hogy ezekben nem érezzük magunkat teljesen illetékesnek. Ehelyett most megkíséreljük néhány mondatban összefoglalni elképzeléseink leglényegesebb vonásait, gyakorlati szempontból is.

A LILIPUTH téma kutatás és fejlesztés összefonódása. Ha bármelyik oldal hosszútávon kiszorítja a másikat, akkor ezt az egész téma megsínyli (a látszólag győztes is). Ezt kevésbé finoman úgy is megfogalmazhatjuk, hogy ha a fejlesztés nehézségei teljesen háttérbe szorítják a kutatást (mondjuk anyagi kényszer vagy lustaság hatására), akkor a fejlesztés is gyönge minőségű lesz.

Ezen túlmenően a fejlesztés megfelelő minőségének elengedhetetlen feltétele, hogy a prototípus elkészülte után beindítsunk egy második, MINŐSÉGI szakaszt, amely szükség esetén kétszer olyan sokáig is tarthat, mint az első. A prototípus garantáltan NEM lesz termék szintű, az első verzió esetleges áruba bocsátása (bármilyen kényszerek hatására is) dilettantizmus lenne, és merénylet a műszaki tisztesség ellen.

Maga a LILIPUTH gép kifejezetten exkluzív kategóriába tartozik. Ezért tömeges eladására nem lehet számítani (legalábbis ma úgy tűnik). Nem tudjuk megitélni, hogy Magyarország abban a helyzetben van-e, hogy elviselje ilyen exkluzív gépek kifejlesztését. Az is kérdés azonban, hogy abban a helyzetben van-e, hogy NE foglalkozzék ilyen témákkal. Mindenesetre abban bizonyosak vagyunk, hogy ha szükség van ilyen jellegű témákra, akkor arra a Tudományos Akadémia kutatóintézetei a legelhívatottabbak.

A téma műszaki érdekességénél fogva sok örömet szerez a benne résztvevőknek. Ez nem elhanyagolható szempont. Mégis, semmiképp sem lehet elég ok arra, hogy mindenáron erőltessük.

Minden észrevételt köszönettel fogadunk!

KÖSZÖNETNYILVÁNITÁSOK

Köszönetet mondunk a Zürichi Műszaki Egyetem Informatika Intézet munkatársainak, különösen is Prof. N.Wirth-nek, W.Winiger-nek és L.Geissmann-nak a sokoldalú segítségért.

Köszönetet mondunk Ercsényi Andrásnak, aki minden szinten alkotó módon részt vett e tanulmány létrejöttében. Köszönjük Pollák Tibornak a mikrofejlesztő rendszer fejlesztésében végzett kiváló munkáját.

Köszönjük Bakonyi Péternek, Csaba Lászlónak és Verebély Pálnak a téma felkarolását és a sokoldalú segítséget.

IRODALOMJEGYZÉK

[Alto79] ALTO: A Personal Computer System Hardware Manual
XEROX Corporation 1978, 1979.

[Bösz79] Böszörményi, L.: Az R10 alapú kommunikációs processzor rendszerterve
VEIKI–SzK-53. 99-055, Budapest, 1979.

[Bösz81] Böszörményi, L.: Multi-task rendszerek fejlesztése magasszintű nyelveken.
MTA SZTAKI Tanulmányok, 128/1981.

[Bösz82] Böszörményi, L.: MODULA-2 used for the implementation of a Virtual Terminal Model
CL & CL, Vol.XV. 1982.

[Bösz83] Böszörményi, L. - Ercsényi, A. - Szabó, M.: A MODULA-2 compiler transported via an interpreter
Computer and Automation Institute (CAI) of the Hungarian Academy of Sciences, Budapest, Working Paper, 1983.

[Bur81-a] Burkhart, H.: Konzepte zur Systematisierung der Benutzerschnittstelle in interaktiven Systemen und ihre Anwendung auf den Entwurf von Editoren
Institut für Informatik - 43, ETH, Zürich, 1981.

[Bur81-b] Burkhart, H. - Nievergelt, J.: Structure-oriented editors
Institut für Informatik - 38, ETH, Zürich, 1981.

[DOMA] Apollo DOMAIN Architecture
Apollo Computer Inc.

[Dora81] The DORADO: A High-Performance Personal Computer. Three Papers.
XEROX Corporation, CSL-81-1, January 1981.

[Ercs84] Ercsényi, A.: Szimbólikus debugger a MODULA-LILITH rendszerhez
Working Paper, előkészületben

[Ethe80] The ETHERNET Local Network. Three Reports.
XEROX Corporation, CSL-80-2, February 1980.

[Geis83] Geissmann, L.: Separate Compilation in MODULA-2 and the structure of the MODULA-2 compiler on the personal computer LILITH
Diss. ETH No. 7286, 1983.

[Gutk83] Gutknecht, J.: System programming in MODULA-2: Mouse and Bitmap Display
Institut für Informatik - 56, ETH, Zürich, 1983.

[Gutk84] Gutknecht, J. - Winiger, W.: ANDRA: The document preparation system for the personal computer LILITH
SOFTWARE Practice & Experience, Vol.14. 73-100. 1984.

[Hopp83] Hoppe, J.: MAGNET: A Local Network for LILITH computers
Institut für Informatik - 57, ETH, Zürich, 1983.

[Inga76] Ingalss, D.: The Smalltalk-76 Programming System Design and Implementation
Conference Record of the Fifth Annual Symposium on Principles of Programming Languages 1976.

[Jaco82] Jacobi, Ch.: Code generation and the LILITH architecture
Diss. ETH, No. 7195, 1982.

[Jens78] Jensen, K. - Wirth, N.: PASCAL User Manual and Report
Springer-Verlag Berlin Heidelberg New York, 1978

[John82] Johnson, K.R. - Wick, D.J.: XEROX OPD: An Overview of the MESA Processzor Architecture, 1982. ACM 0-89791-66-4 82/03/0020

[Knud82] Knudsen, S.E.: MEDOS-2: A MODULA-2 oriented operating system for the personal computer LILITH
Diss. ETH, No. 7346, 1983.

[Lili82] Collective of the IFI-ETH: The LILITH handbook
ETH IFI 1982

[McCo83] McCormack, J. - Gleaves, R.: MODULA-2. A Worthy Successor to PASCAL
Byte, p. 385-395, April, 1983.

[McDa82] McDaniel, G.: An Analysis of MESA Instruction Frequencies
1982 ACM 0-89791-066-4 82/03/0167

[Mitc79] Mitchell, J.G. - Maybury, W. - Sweet, R.: MESA Programming Manual
XEROX Report, CSL-79-3, April 1979.

[Nels81] Nelson, B.J.: Remote Procedure Call
XEROX PARC, CSL-81-9 1981

[PERQ82] PERQ: Technical Overview
ICL, P1431, 1982

[SPICE79] CMU SPICE Committee
Proposal for a Joint Effort in Personal Scientific Computing
Technical Report, CARNEGIE-MELLON University, Department of Computer Science, August 1979.

[Summ82] Summer, T.R. - Gleaves, E.R.: MODULA-2 A Solution of PASCAL Problem
Volation System, 1982.

[Sun83] The Sun-2 Product Tamily: A technical Overview
Sun Microsystem, 1983.

[SYMB83] SYMBOLICS 3600 Technical Summary
Symbolics, Inc., February 1983.

[Szab84] Szabó, M.: MODULA: Nyelv, fordító, környezet
MTA SZTAKI Tanulmányok (megjelenés előtt).

[Wagn83] Wagner, B. - Hoppe, J.: Using the LAN MAGNET; Maintenance and testing of LILITH
Institut für Informatik - 58, ETH, Zürich, 1983.

[Wirt77] Wirth, N.: MODULA: A language for modular multiprogramming
SOFTWARE Practice & Experience, Vol.7. pp. 2-35. 1977.

[Wirt81] Wirth, N.: The Personal Computer LILITH
Institut für Informatik - 40, ETH, Zürich, 1981.

[Wirt82] Wirth, N.: Programming in MODULA-2
Springer-Verlag Berlin Heidelberg New York, 1982.

[Wirt84] Wirth, N. Schemes for multiprogramming and their implementation in MODULA-2
Revisions and amendments to MODULA-2
Institut für Informatik - 59, ETH, Zürich, 1984.

[Zehn83] Zehnder, C.A.: Database techniques for professional workstations
Institut für Informatik - 55, ETH, Zürich, 1983.

1984-BEN JELENTEK MEG:

155/1984 Deák, Hoffer, Mayer, Németh, Potecz, Prékopa, Straziczky: Termikus erőműveken alapuló villamosenergiarendszerek rövidtávu, optimális, erőművi menetrendjének meghatározása hálózati feltételek figyelembevételével.

156/1984 Radó Péter: Relációs adatbáziskezelő rendszerek összehasonlitó vizsgálata

157/1984 Ho Ngoc Luat: A geometriai programozás fejlődései és megoldási módszerei

158/1984 PROCEEDINGS of the 3rd International Meeting of Young Computer Scientists,
Edited by: J. Demetrovics and J. Kelemen

159/1984 Bertók Péter: A system for monitoring the machining operation in automatic manufacturing systems

160/1984 Ratkó István: Válogatott számitástechnikai és matematikai módszerek orvosi alkalmazása

161/1984 Hannák László: Többértékü logikák szerkezetéről.

162/1984 Kocsis J. - Fetyiszov V.: Rugalas autamatizált rendszerek: megbizhatóság és irányitási problémák

163/1984 Kalavszky Dezső: Meleghengermüvi villamos hurokemelő hajtás vizsgálata

164/1984 Knuth Előd: Specifikációs adatbázis modellek

165/1984 Petrőczy Judit: Publikációk 1983

1985-BEN EDDIG MEGJELENTEK:

166/1985 Radó Péter: Információs rendszerek számitógépes tervezése

167/1985 Studies in Applied Stochastic Programming I. Szerkesztette: Prékopa András

COOPRINT 85-248