



**Miskolci Egyetem ... és Informatikai Kara**

**Végh János**  
**Bevezetés a számítógépes**  
**rendszerekbe –**  
**programozóknak**  
**Egy gyors körkép**

Copyright © 2008-2015  
(J.Vegh@uni-miskolc.hu)  
V0.11@2015.02.16

Ez a segédlet a *Bevezetés a számítógépes rendszerekbe* tárgy tanulásához igyekszik segítséget nyújtani. Nagyrészt az irodalomjegyzékben felsorolt Bryant-O'Hallaron könyv részeinek fordítása, itt-ott kiegészítve és/vagy lerövidítve, néha már jó tankönyvek illeszkedő anyagaival kombinálva. A képzés elején, még a számítógépekkel való ismerkedés fázisában kerül sorra, amikor előbukkannak a különféle addig ismeretlen fogalmak, és megpróbál segíteni eligazodni azok között. Alapvetően a számítógépeket egyfajta rendszerként tekinti és olyan absztrakciókat vezet be, amelyek megkönnyítik a kezdeti megértést.

Ez az anyag még erőteljesen fejlesztés alatt van, akár hibákat, ismétléseket, következetlenségeket is tartalmazhat. Ha ilyet talál, jelezze a fenti címen. Az eredményes tanuláshoz szükség van az irodalomjegyzékben hivatkozott forrásokra, és az órai jegyzetekre, ottani magyarázatokra is.



---

# Tartalomjegyzék

<b>Tartalomjegyzék</b>	<b>i</b>
<b>1 Egy gyors körkép</b>	<b>1</b>
1.1 Információ . . . . .	11
1.2 Fordítás . . . . .	14
1.3 A fordítás részletei . . . . .	18
1.4 Tárolt utasítások . . . . .	22
1.4.1 Hardveres felépítés . . . . .	24

1.4.2	A <b>hello</b> program futtatása . . . . .	31
1.5	A gyorsítótár . . . . .	35
1.6	A tárolóeszközök hierarchiája . . . . .	39
1.7	Az operációs rendszer . . . . .	41
1.7.1	Folyamatok . . . . .	44
1.7.2	Virtuális memória . . . . .	49
1.8	Kommunikáció a hálózaton át . . . . .	54
1.9	Fontos egyébek . . . . .	58
1.9.1	Konkurrens és párhuzamos . . . . .	59
1.9.2	Számítógépes absztrakciók . . . . .	68
1.10	Összefoglalás . . . . .	71

<b>Tárgymutató</b>	<b>73</b>
--------------------	-----------

<b>Táblázatok jegyzéke</b>	<b>76</b>
----------------------------	-----------

<b>Ábrák jegyzéke</b>	<b>77</b>
-----------------------	-----------

<b>Bibliography</b>	<b>82</b>
---------------------	-----------

1

---

# Egy gyors körkép

A számítógépes rendszerek világával ismerkedve, nagyon sok új és ismeretlen fogalommal találkozunk. Rendkívül nehezen tudjuk elválasztani a még fontos részletet az esetleg érdekes, de az adott pillanatban nem nélkülözhetlentől. Ez a segédlet azoknak készült, akik most kezdenek ismerkedni a területtel, és szeretnék megtanulni, hogyan lehet a nagyrészt állandó elvek felhasználásával

eligazodni a számítógép programozás nagyon gyorsan változó és rendkívül szerteágazó világában.

Célkitűzésünk, hogy a hallgatók *számítógép programozó*ként hatékonyabban tudják használni a számítógépes rendszereket, függetlenül attól, hogy rendszer-, adatbázis- vagy WEB-programozóként dolgoznak, netán fordítóprogramot írnak, hálózati alkalmazást vagy beágyazott rendszert készítenek. Magát a túrát [1] alapján, az ott megadott útvonalon tesszük, de jelentősen csökkentve a tárgyalás részletességét. A könyv szerzői az anyagot mindazoknak ajánlják, akik meg akarják érteni (kezdetben inkább csak felszínesen), mi megy végbe egy számítógépes rendszerben a színelak mögött. Későbbi tanulmányaikban az itt éppen csak érintett témákat részletesen tanulmányozni fogják.

Látszólag hardver témákat érintünk, de nem konstruktóri, hanem programozói szempontból: hogyan készíthetünk hatékonyabb, gyorsabb programokat a jól megismert tulajdon-

ságú hardverekre. Alapvetően Linux és C (vagy C++) ismereteket használunk, és gyakorlatilag nem tételezünk fel előzetes hardver vagy gépi/assembly kódolási ismereteket sem.

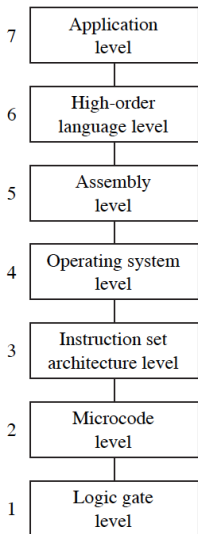
Programozni természetesen csak programozás közben lehet megtanulni. Akinek még nem stabil és készség szintű a tudása, azoknak nagyon érdemes beható ismeretséget kötni a [4] könyvecskével. Számítógép mellett érdemes olvasni, és azonnal kipróbálni a programozási példákat.

Mint oly sok más területen, a számítógépes világban is nagyon jól használható egy absztrakt, a bonyolult rendszert többé-kevésbé jól szintekre bontó megközelítés<sup>1</sup>. A segédletben ilyen szemlélet alapján, de a jóval kisebb terjedelemnek megfelelően kevésbé részletesen tekintjük át, és főként nem belebonyolódva a részletekbe. Az egyes szintek megismerésére nem mindig a feltüntetett sorrendben

---

<sup>1</sup>Főként [8] alapján

kerítünk sort, hanem ahogyan az céljaink és a már megszerzett ismereteink alapján lehetséges.



1.1. ábra. Egy tipikus számítógérendszer rétegmodellje



A bevezető jellegű anyagban nem követelünk meg előismeretet, de támaszkodunk a korábban megismertekre, és azokat csak ismétlés jelleggel megemlítjük. Bemelegítésként megismerkedünk a "kis ember számítógépe" modellel[2, 3], amit érdemes figyelmesen tanulmányozni, mivel nagyon szemléletesen vezeti be a számítógéppel kapcsolatos fogalmakat, bár konkrét technikai megvalósítást nem tartalmaz.

A második fokozatban egy félkész elemekből sajátkezűleg elkészítendő számítógépet ismertetünk [5, 6, 7], ami már konkrét technikai megvalósítást ismertet. Valóban megépíthető és működőképes, de mi csak szimulációs szinten fogjuk használni. Ez már szinte minden főbb olyan elemet tartalmaz, amit a mai processzorokban is alkalmaznak, de mindenképp csak a minimumot. Ennek következtében nagyon könnyű átlátni és használni egyszerű feladatok megértésére, megtanulhatjuk használni az elemi gépi utasításokat, de komoly feladatokat csak nagy nehézségek árán lehet

vele megoldani. Nagy hátránya, hogy nincs magas szintű támogatása, így nem lehet vele a számítógép magas szintű nyelvein készített programok alacsony szintű következményeit megtanulni. Nagyon kiváló gyakorlati segéd-eszközök állnak rendelkezésre, gyakran fogjuk használni a [5] oldalról letölthető szimulátort. Az eredményes tanuláshoz sok önálló foglalkozásra is szükség van.

Harmadik fokozatként megismerkedünk napjaink asztali és hordozható számítógépekben használt leggyakoribb processzorának, az Intel x86 családjának tulajdonságaival és programozásával. A rendelkezésre álló rövid idő alatt nem tudjuk teljes részletességgel tárgyalni az utasításkészletét, felépítését, stb, de a felületes megismerkedés alapján is sok hasznos ismeretet tudunk szerezni, és a tanulást a következő félév gyakorlataival fogjuk kiteljesíteni. Ehhez eleinte a DOSbox szimulátor programot fogjuk használni (hogy kiküszöböljük az operációs rendszer által okozott jelenségeket), majd áttérünk a valódi

(Linux) rendszer alatt futó programok használatára. A programíráskor egyértelműen a C nyelvet használjuk (a Java használatában gyakorlottaknak nagyon ajánlott a mutatók, a dinamikus tárterület foglalás, stb. használatának felelevenítése, ami nincs a Java alatt).

Hogy a jelen segédlet még elviselhető terjedelmű legyen, a gyakorló feladatok (és azok kapcsolódó útmutatásai) külön segédletbe kerültek. Ezt a két segédletet egymást kiegészítve és támogatva kell használni. Ismét hangsúlyozzuk, hogy a megértés kulcsa a végrehajtás és a gyakorlás. A gyakorlókönyv a legtöbb esetben tartalmazza a megoldást. Azt csak saját megoldásunk ellenőrzésére érdemes használni, nem helyettesíti az önerőt.

Egy **számítógépes rendszer** hardver és szoftver elemekből áll, amelyek együttműködve felhasználói programozásokat futtatnak. A rendszer tényleges megvalósítása idővel változik, de az alapvető elvek nem. Valamennyi számítógép hasonló elemekből áll és a szoftver elemek is hasonló feladatokat látnak el. Azt

fogjuk megismerni, hogyan lehet mindezeket hatékonyan felhasználni, és ezek a komponensek hogyan befolyásolják programjaink helyességét és hatékonyságát. Azért érdemes mindezt megtanulnia, hogy szakmáját értő programozóvá ("power programmer") váljon, aki érti a használt számítógépes rendszer működését és annak hatását a készített programra.

Valamennyi, a C nyelvet tanító kurzuson szerepel az 1.1 program. Bár maga a program rendkívül egyszerű, egy számítógépes rendszer valamennyi részének összehangoltan kell működnie a sikeres végrehajtásához. Bizonyos értelemben, a kurzus célja annak bemutatása, mi történik pontosan, amikor ez a 'hello' program lefut. ☺

Azzal kezdjük, hogy nyomon követjük a 'hello' programot, kezdve azzal, hogy a programozó megírja, egészen addig, hogy fut a rendszeren, kinyomtatja ezt az egyszerű üzenetet és befejeződik. Közben pedig bevezetjük a kulcsfontosságú fogalmakat és a

## Programlista 1.1: A "Hello Világ" program forráskódja

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
}
```

terminológiát, valamint a használt fontosabb komponenseket. Később majd részletesen is megismerkedünk velük.

### Megtanuljuk

- hogyan kerülhetjük el a számítógépes számábrázolásból fakadó furcsa numerikus hibákat
- milyen trükkökkel optimalizáljuk programunkat a modern processzorok és memóriák használatával
- hogyan valósítja meg a fordítóprogram az eljárás hívásokat és ez hogyan teszi sebezhetővé a hálózati szoftvereket
- hogyan kerülheti el a program összecha-

toláskor keletkező csúnya hibákat

- hogyan kell saját parancs értelmezőt, memórafoglalást vagy WEB kiszolgálót írni
- megismerjük a sokmagos processzorok használatakor mind fontosabb konkurrens programozást

**1.1. Információ = bitek + értelmezés**

A mi **hello** programunk **forrás fájl**ként (**source program**, forrás kód) kezdi pályafutását, amit a programozó egy szerkesztővel hoz létre és a **hello.c** szöveg fájlban ment el. A forrás kód egyszerűen bitek sorozata, amelyek egyenként 0 vagy 1 értéket vehetnek fel, és amelyeket 8 bites darabokba (bájtok) szervezünk. Minden egyes bájtt a program egy szöveg karakterét ábrázolja.

1.1. táblázat. A **hello.c** program (lásd 1.1 programlista) ábrázolása ASCII szöveggént

#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	<sp>	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	<sp>	<sp>	<sp>	<sp>	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	112	114	105	110	116	102	40	34	104	101	108	
l	o	,	<sp>	w	o	r	l	d	\n	"	)	:	\	n	}
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	125

A legtöbb modern rendszer ASCII szabványú karakterekkel ábrázolja a szöveget, amely szabvány az egyes karaktereket egyedi, bájtt méretű egész értékekkel ábrázolja. A `ctext-hello.c` programot egy fájlban bájtok soroza-

taként tároljuk, lásd 1.1 táblázat. Az egyes bájtok egész értékek, amelyek egy bizonyos karakternek felelnek meg. Például, az első bájt 35, ami a '#' karakternek felel meg. A második bájt értéke a 105 egész szám, ami az 'i' betű megfelelője, és így tovább. Vegyük észre, hogy az egyes szöveg sorokat a "láthatatlan" 'új sor@új sor' (newline, '\n') karakter határolja, amelyet a 10 érték ábrázol. A 'hello.c'-hez hasonló fájlokat, amelyek kizárólag ASCII karaktereket tartalmaznak, **szövegfájl**nak (text file) nevezzük, minden más fájlt pedig **bináris fájl**nak (binary file).

A 'hello.c' reprezentációja egy nagyon fontos elképzelést mutat be: egy rendszerben minden információt – beleértve a mágneslemez állományokat, a memóriában tárolt programokat és felhasználói adatokat, a hálózaton átvitt adatokat – egy kupac bitként ábrázolunk. Ami a különböző adat objektumokat megkülönbözteti, az az értelmezés, amelyet hozzájuk fűzünk. Különböző összefüggésben ugyanaz a bájt sorozat jelenthet egy egész szá-



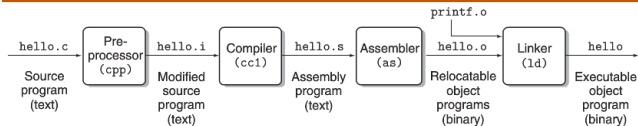
mot, egy lebegőpontos számot, egy karakter sztringet, vagy gépi utasítást.

Programozóként meg kell értenünk a számok számítógépes ábrázolását, mivel az nem ugyanaz, mint az egész vagy valós szám. Ezek mindegyike egy véges közelítés, amelyik teljesen váratlan módon is viselkedhet. Ezt majd a következő fejezetben tárgyaljuk.

## 1.2. Programot programmal fordítunk

A 'hello' program tehát magas szintű C nyelven megírt programként kezdi életét, mivel az emberek ilyen formában tudják megérteni és kezelni. Ahhoz azonban, hogy a 'hello.c' programot futtassuk egy rendszerben, az egyes C utasításokat más programokkal le kell fordítanunk gépi utasítások sorozatává. Ezeket az utasításokat aztán végrehajtható fájlba fűzzük össze és bináris fájlként tároljuk a mágneslemezen. Egy Unix rendszeren a forráskódot objekt fájlra egy fordítóprogram felhasználásával alakítjuk át:

```
unix> gcc -o hello hello.c
```



1.2. ábra. A számítógépes fordításhoz használt rendszer

Ennek az utasításnak a hatására a GCC fordítóprogram beolvassa a **hello.c** forrás fájl tartalmát és azt a **hello** végrehajtható objekt kóddá fordítja. Ez a folyamat négy fázisban megy végbe, lásd 1.2 ábra.

- **Előfeldolgozás** (preprocessing phase) Az előfeldolgozó (**cpp**) az ún. direktívák (a # karakterrel kezdődő sorok) által meghatározott módon módosítja az eredeti C programot. Például az **#include** <stdio.h> utasítássor hatására az előfeldolgozó beolvassa a **stdio.h** rendszer fejzet fájlt, és annak tartalmát közvetlenül ennek a sornak a helyére helyettesíti. Ennek eredményeként egy másik C program keletkezik, tipikusan **.i** kiterjesztéssel.
- **Magas szintű fordítás** (compilation phase) A fordítóprogram (**cc1**) a **hello.i** szövegfájlt a **hello.s** szövegfájlba fordítja le, ami már egy ún. assembly nyelvű programot fog tartalmazni. Az **assembly nyelvű program** egyes utasításai szten-

derd szöveges formátumként pontosan leírnak egy alacsony szintű gépi utasítást. Az assembly nyelv azért hasznos, mert a különféle magas szintű nyelvek fordítóprogramjai számára közösen használható kimeneti nyelvet biztosít.

- **Gépi fordítás** (assembly phase) Ezután az assembler fordító (**as**) a **hello.s** tartalmát gépi utasításokká fordítja, és azokat az **athelyezhető objekt program@áthelyezhető objekt program**ként ismert formába helyezi el, a **hello.o** fájlba. Ez a **hello.o** olyan bináris fájl, amelynek bájtjai a gépi utasítások kódját és nem pedig karaktereket tartalmaznak.
- **Csatolás** (linking phase) Vegyük észre, hogy a **hello** program meghívja a **printf** függvényt, ami a minden C fordítóprogramhoz tartozó **sztenderd C könyvtár** része. A **printf** függvény egy különálló, előre lefordított **printf.o** objekt fájlban található meg, amit valahogyan össze kell kapcsolnunk saját **hello.o** programunkkal.

A csatoló (linker, ld) végzi el a csatolás feladatát. Ennek eredménye a **hello** fájl, amelyik egy **végrehajtható objekt program** (végrehajtható állomány), azaz betölthető a memóriába és azt a rendszer végre tudja hajtani.

### 1.3. Érdemes megérteni a fordítás részleteit

A `hello.c` programhoz hasonló "bonyolultság" esetén bízhatunk benne, hogy a fordítóprogram helyes és hatékony programot készít. Van azonban pár általános indok, miért érdemes megérteni a fordító rendszer működését:

- **A program működés optimalizálása**  
A modern fordítóprogramok bonyolult eszközök, amelyek általában jó kódot készítenek. Ahhoz, hogy hatékony kódot írassunk, meg kell értenünk a fordítóprogram belső működését. Viszont ahhoz, hogy jó kódolási döntéseket hozunk C programunkban, legalább a gépi kódolás alapjait meg kell ismerni, valamint tudnunk kell, a C fordító hogyan fordítja le a C utasításokat gépi kóddá. Például, egy **switch** utasítás mindig hatékonyabb, mint **if-else** utasítások sorozata? Mennyi többlet tevékenységet okoz egy függvényhívás? Egy **while** ciklus

hatékonyabb, mint egy **for** ciklus? A mutatóval való hivatkozás hatékonyabb, mint egy tömb indexelése? Miért fut egy ciklus sokkal gyorsabban, ha az összeget egy helyi változóban gyűjtjük, és nem egy hivatkozásként átadott változóban? Miért fut gyorsabban egy függvény, ha egyszerűen átrendezzük a zárójeleket egy aritmetikai kifejezésben? Hogyan befolyásolja a program futás gyorsaságát számítógépünk memóriájának szerkezete?

- **Csatolási hibák megértése** Tapasztalatunk szerint a legzavarbaejtőbb hibák a csatoló program működésével kapcsolatosak, különösen ha nagyobbacska szoftver rendszert építünk. Például, mit jelent, ha a csatoló hibajelentése szerint nem lehet feloldani egy hivatkozást? Mi a különbség egy statikus és egy globális változó között? Mi történik, ha különböző C fájlokban két globális változót ugyanazzal a névvel definiálunk? Mi a

különbség egy statikus és egy dinamikus könyvtár között? Miért számít az, milyen sorrendben adjuk meg a könyvtárakat a parancssorban? És a legfurcsább: bizonyos csatolási hibák miért csak futási időben nyilvánulnak meg?

- **Biztonsági lyukak elkerülése** Sok éven át a puffer túlcsordulás sebezhetőség számított a hálózatok és az Internet szerverek fő biztonsági problémájának. Ennek a legfőbb oka, hogy csak kevés programozó értette meg, hogy a megbízhatatlan forrásból származó adatok mennyiségét és formáját korlátozni kell. A biztonságos programozáshoz vezető út első lépése, hogy meg kell érteni annak következményeit, hogy a program veremtarolójában adat és vezérlő információt tárolunk. Megismerjük a veremtaroló működési elvét és a puffer túlcsordulás okozta sebezhetőséget, assembly szinten. Megtanuljuk, hogy a programozó, a fordítóprogram és az operációs



*A processzor tárolt utasításokkal dolgozik* 21  
rendszer hogyan tudja csökkenteni a  
támadás lehetőségét.

## **1.4. A processzor tárolt utasításokkal dolgozik**

Eddig tehát **hello.c** forrásprogramunkat a fordító rendszer lefordította a **hello** nevű végrehajtható objekt fájlá, amit a mágneslemezen tárolunk. Hogy futtatni tudjuk programunkat egy Unix rendszeren, leírjuk annak nevét egy parancsértelmező felhasználói programban (shell):

```
unix> ./hello  
hello, world  
unix>
```

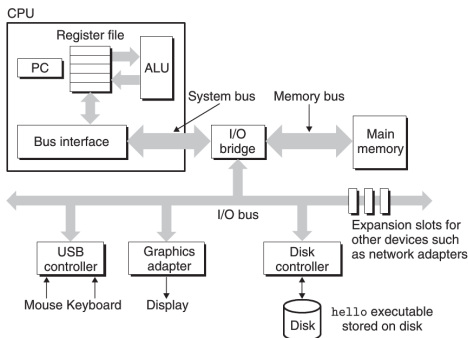
A parancsértelmező kinyomtat egy ún. prompt jelzést (annak jeléül, hogy készen áll parancs végrehajtására), megvárja, amíg leírjuk az utasítássort és végrehajtja az utasítást. Ha az első sor nem értelmezhető beépített utasításként, a parancsértelmező azt tételezi fel, hogy az egy végrehajtható fájl neve, amit be kell tölteni a memóriába és le kell futtatni. Ezt megteszi, majd megvárja, amíg az befejeződik. A **hello** program kinyomtatja az üzenetet és

*A processzor tárolt utasításokkal dolgozik* 23  
befejeződik. Ezután a parancsértelmező ismét jelzi a futáskészségét egy prompt kiírásával, és várja a következő utasítást.

### 1.4.1. A rendszer hardveres felépítése

Hogy megértsük, mi is történik, amikor futtatjuk a **hello** programot, meg kell értenünk, hogy milyen hardver felépítésű egy tipikus rendszer, lásd 1.3 ábra. Ezen a képen egy Intel Pentium alapú rendszer modellje látható, de valamennyi rendszer hasonlónak látszik és hasonlóan működik. A részleteket majd később fogjuk megérteni.

- A számítógép **sín rendszere** (vagy más néven **busz rendszere**) olyan vezetékekből áll, amelyek az elektromos jeleket szállítják az egyes komponensek között. A sínrendszereket tipikusan úgy tervezik, hogy a bájtok többszörösének megfelelő rögzített méretű darabokat (ezeket nevezzük szavaknak) szállítanak. A szóban található bájtok száma (a **szó mérete**) fontos paraméter, ami eltér a rendszerek között. A legtöbb mai számítógép 4 bájtos (32 bites) vagy 8 bájtos (64 bites) szóhosszal rendelkezik. Az egyszerűség ked-



1.3. ábra. Egy tipikus rendszer hardveres felépítése. CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program counter, USB: Universal Serial Bus.

©[1] 2013

véért a továbbiakban 4 bájtos hosszúságot tételezünk fel, valamint hogy a busz egyidejűleg csak egy szót tud átvinni.

- A rendszer **be/kiviteli (I/O) eszközökön** át kapcsolódik a külvilághoz. A példában szereplő rendszernek négy I/O eszköze van: billentyűzet (keyboard) és

egér (mouse) a felhasználói adatbevitelhez, kijelző (display) az adatmegjelenítéshez, és mágneslemez egység (disk drive) az adatok hosszú távú tárolására. Kezdetben a **hello** program a mágneslemezen van. Az egyes I/O eszközök egy **vezérlőn** (controller) vagy **adapteren** át kapcsolódnak az I/O sínrendszerhez. A kettő közötti különbség lényegében csak a tokozás. A vezérlők olyan áramköri elemek, amelyek vagy magában az eszközben vagy a rendszer fő nyomtatott áramköri lapján (anyakártya avagy motherboard) találhatóak. Az adapter pedig olyan kártya, ami az anyakártya csatlakozójához kapcsolódik. Mindkettő célja, hogy adatot szállítson az I/O sínrendszer és az I/O eszköz között.

- A **fő memória** olyan átmeneti tároló, amelyik mind a programot, mind az általa kezelt adatokat tárolja, amíg a programot a processzor végrehajtja. Fizikailag a fő memória véletlen hozzáfé-

résű memória áramkörökből (dynamic random access memory, DRAM) épül fel. Logikailag a memória egy lineáris bájt tömbnek tekintendő, ahol minden bájt-nak saját egyedi címe van (a tömb index), mely nullától kezdődik. A programot alkotó egyes gépi utasítások általában eltérő számú bájtból állnak. A C program változóinak megfelelő adat elemek mérete azok típusának megfelelően változik. Például, egy Linuxot futtató IA32 számítógépen a **short** típusú adat két bájtot igényel, a **int**, **float** és **long** típusú adatok négy bájtot, a **double** típusúak pedig nyolc bájtot.

- A központi feldolgozó egység (avagy **central processing unit**, CPU) az a feldolgozó "motor" amelyik értelmezi (vagy végrehajtja) a fő memóriában tárolt utasításokat. Ebben központi szerepet játszik egy szó-méretű tároló (avagy regiszter), a program számláló (program counter, PC). A PC bármely pillanatban a főme-

móriában valamely gépi kódú utasításra mutat (annak a címét tartalmazza).

Attól a pillanattól kezdve, hogy a rendszernek tápfeszültséget adunk, amíg azt ki nem kapcsoljuk, a processzor folyamatosan végrehajtja a programszámláló által kijelölt utasítást és frissíti a programszámlálót, hogy az a következő utasításra mutasson. A processzor nagyon egyszerű utasítás végrehajtási modell szerint dolgozik, amit az [utasítás készlet szerkezete \(instruction set architecture\)](#) határoz meg. Ebben a modellben az utasítások szigorú sorrend szerint hajtódnak végre és egy utasítás végrehajtása egy lépés sorozat elvégzését jelenti. A processzor beolvassa az utasítást a programszámláló (PC) által kijelölt memória címről, értelmezi az utasítást, elvégzi az utasítás által előírt egyszerű műveletet, majd frissíti a PC értékét, hogy az a következő utasításra mutasson, ami vagy a memóriának az utasítás helyét követő



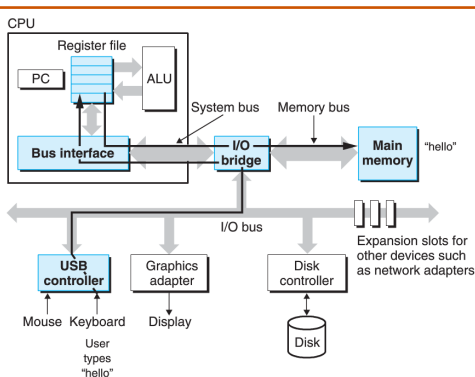
A processzor tárolt utasításokkal dolgozik 29 sorszámú helye, vagy egészen más érték. Csak pár ilyen egyszerű művelet van, amelyek a fő memóriával, a **regiszter tömbbel** (register file) és az **aritmetikai/logikai egységgel** (arithmetic/logic unit, ALU) foglalkoznak. A regiszter tömb egy olyan kisméretű tároló, amely szó méretű regiszterekből áll és mind-egyiknek saját neve van. Az ALU számítja ki az új adatok és címek értékét. Pár példa, milyen egyszerű műveleteket hajthat végre a CPU egy utasítás hatására:

- **Betöltés (Load)** egy bájtot vagy szót a fő memóriából egy regiszterbe másol, a regiszter előző tartalmát felülírva
- **Eltárolás (Store)** Egy bájtot vagy szót másol egy regiszterből a fő memóriába, a memória előző tartalmát felülírva
- **Művelet (Operate)** két regiszter tartalmát az ALU-ba másolja, a két szóval aritmetikai műveletet végez, majd az eredményt egy regiszterbe menti, a regiszter előző tartalmát felülírva
- **Ugrás (Jump)** Magából az utasításból

*A processzor tárolt utasításokkal dolgozik* 30  
következtet egy egy szó hosszúságú értéket, azt a szót a programszámlálóba másolja, felülírva a PC előző értékét.

Azt mondjuk ugyan, hogy a processzor egyszerűen az utasításkészletének a tényleges megvalósítása, de valójában a modern processzorok összetett mechanizmusokat használnak a program végrehajtás felgyorsítására. Ennek megfelelően megkülönböztetjük a processzor utasításkészlet architektúráját, amely az egyes mikroutasítások hatását írja le, a processzor mikroarchitektúrájától, amely a processzor tényleges megvalósítása.

## 1.4.2. A hello program futtatása

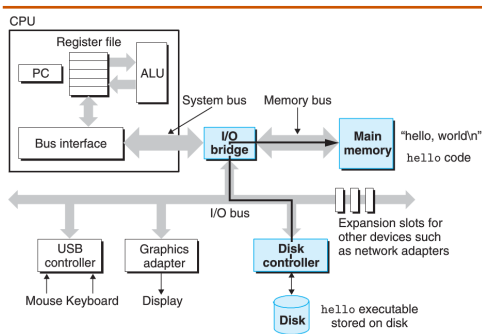


1.4. ábra. A hello utasítás beolvasása a billentyűzetről.

©[I] 2013

A számítógépes rendszerek szerkezetének és működésének eddig megismert részleteivel már elkezdhetjük megérteni, mi történik, amikor futtatjuk példa programunkat. Néhány (a későbbiekben megismerendő) részlettől eltekintve, már képesek vagyunk egyfajta "ma-

A processzor tárolt utasításokkal dolgozik 32 dártaavlati" képet adni. Kezdetben a parancs-értelmező (shell) hajtja végre saját utasításait, arra várva, hogy begépeljünk egy utasítást. Amint begépeljük a **"/hello"** karaktereket a billentyűzeten, a parancsértelmező beolvassa azokat egy regiszterbe, és eltárolja a memóriába, lásd 1.4 ábra.



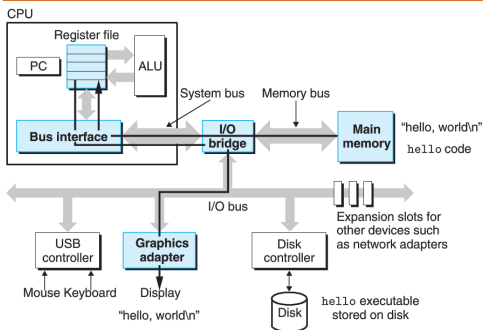
1.5. ábra. A végrehajtható fájl betöltése a mágneslemezről a fő memóriába.

©[I] 2013

Amikor lenyomjuk az "enter" gombot a billentyűzeten, a parancsértelmező tudomásul

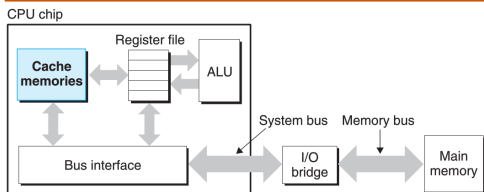
veszi, hogy befejeztük az utasítás begépelését. Ezután a parancsértelmező betölti a végrehajtható **hello** fájlt, egy olyan utasítássorozat végrehajtásával, amelyik a **hello** objekt fájl kódját és adatait bemásolja a mágneslemezről a fő memóriába. Az adatok között van az a “hello, world\n” karakterfüzér (string), amit esetleg kinyomtatunk. Ebben az esetben a [közvetlen memóriaelérés \(direct memory access, DMA\)](#) nevű technika segítségével az adatok a mágneslemezről közvetlenül a fő memóriába kerülnek, a processzoron való áthaladás nélkül, lásd [1.5](#) ábra.

Amikor a **hello** program kódja és adatai betöltődtek a memóriába, a processzor elkezdi a **hello** program gépi kódú utasításainak végrehajtását. Ezek az utasítások a “hello, world\n” szöveg bájtjait a memóriából a regiszter tömbbe másolják, onnét pedig a kijelzőre, aminek következtében azok láthatóvá válnak a képernyőn, lásd [1.6](#) ábra.



1.6. ábra. A kimenő szöveg kiírása a memóriából a képernyőre.

## 1.5. A gyorsítótár is számít



1.7. ábra. Gyorsítótárak.

©[I] 2013

Fontos tanulság ebből az egyszerű példából, hogy a rendszer sok időt tölt azzal, hogy információt mozgasson egyik helyről a másikra. A **hello** programban levő gépi utasításokat eredetileg a mágneslemezen tároltuk. Amikor a programot betöltjük, ezeket bemásoljuk a fő memóriába. Amikor a processzor futtatja a programot, ezeket az utasításokat a fő memóriából a processzorba másolja. Hasonlóképpen, a “hello,world\n” adatfüzér, amit eredetileg a mágneslemezen tároltunk, bemásolódik a fő memóriába, majd a fő

memóriából a kijelző eszközre. A programozó szempontjából, eme másolások jelentős része csak olyan többlet tevékenység (overhead), ami lelassítja a program "valódi munkáját". Emiatt a rendszer tervezők fő célja, hogy ezeket a másolási műveleteket a lehető leggyorsabban végre lehessen hajtani.

A fizikai törvények miatt, a nagyobb tárolóeszközök lassúbbak, mint a kisebbek. A gyorsabb eszközök pedig többre kerülnek, mint a megfelelő lassúbb változatok. Például, egy tipikus mágneslemez tároló kapacitása 1000-szer nagyobb lehet, mint a fő memóriáé, de a processzor akár 10,000,000-szor lassabban tud egy szót a mágneslemezzel elővenni, mint a memóriából.

Hasonlóképpen, egy tipikus regiszter tömb csak pár száz bájtnyi információt tárol, szemben a fő memória néhány milliárdnyi bájtjával. A processzor azonban csaknem százszor gyorsabban tudja a regiszter tömbből olvasni az adatokat, mint a memóriából. Még ennél is rosszabb, hogy a félvezető technológia

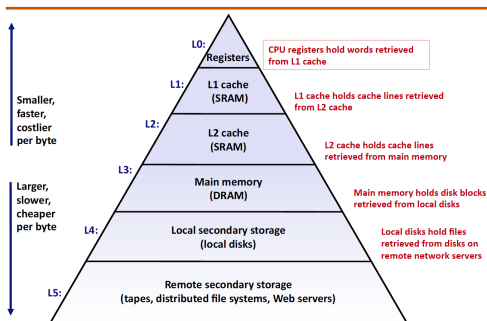


fejlődésével ez a szakadék a processzor és a memória között tovább mélyül. Egyszerűbb és olcsóbb a processzorokat gyorsítani, mint a memóriákat.

A szakadék mélységének csökkentésére a rendszer tervezők kisméretű gyors tároló eszközöket, **gyorsítótárat** (**cache memories** vagy egyszerűen **cache**) helyeznek el a processzorban, amelyek olyan információk átmeneti tárolására használhatók, amely információkra a processzornak várhatóan szüksége lesz a közeli jövőben. Az 1.7 ábra egy ilyen, gyorsítótárral ellátott rendszert mutat. A processzor chipben levő *L1 gyorsítótár* pár tízezer bájtot tartalmaz és csaknem ugyanolyan gyorsan lehet elérni, mint a regisztertömböt. A nagy *L2 gyorsítótár* pár százezer ... pár millió bájtot tartalmaz és egy speciális busszal kapcsolódik a processzorhoz. Az *L2 gyorsítótár* ötször lassabban érhető el, mint az *L1 gyorsítótár*, de ez még mindig 5-10-szerese a fő memória elérésének. Az *L1* és *L2 gyorsítótárat* a static random access memory (SRAM) techno-

lógiaival készítik. Az újabb rendszerek három szintű gyorsítótárat használnak: L1, L2, és L3. A gyorsítótár mögött az az ötlet áll, hogy a rendszer jó hasznát látja mind a nagyon nagy memóriának, és az ún. lokálitás (azaz, hogy a program jól lokalizálható helyről veszi a következő adatokat és kódot) kihasználásával a nagyon gyors memóriának is. Ha a gyorsítótárat úgy tudjuk beállítani, hogy az tartalmazza a várhatóan gyakran használt adatokat és kódokat, a legtöbb memória műveletet a gyorsítótár felhasználásával tudjuk elvégezni. Az egyik legfontosabb következtetés, hogy *az alkalmazói programok írói, ha jól tudják használni a gyorsítótárat, nagyságrenddel gyorsabb programot tudnak készíteni.*

## 1.6. A tárolóeszközök hierarchiája



1.8. ábra. Példa a memória hierarchiára.

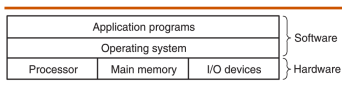
©[1] 2013

Az az ötlet, hogy egy kisebb, de gyors tárolóegységet (azaz gyorsítótárat) tegyünk a processzor és a nagy eszköz (pl. fő memória) közé, általánosanak érvényű. Valójában minden számítógépes rendszerben a tárolóeszközök olyan hierarchiába szervezettek, lásd 1.8 ábra. Ahogyan a hierarchia tetejétől az alja felé haladunk, az eszközök egyre lassúbbá, nagyobbá és fajlagosan (bájtónként) olcsóbbá

válnak. A hierarchia tetején a regiszter tömb található, 0. szint (L0) néven. Ezt az 1-3 szinten található L1-L3 gyorsítótárak követik. A fő memória van a négyes szinten, és így tovább.

A memória hierachia lényege, hogy egy szint gyorsítótárként szolgál a következő alacsonyabb szint számára. Azaz, a regiszter tömb az L1 gyorsítótár gyorsítótára. Az L1 és az L2 gyorsítótár az L2 és L3 gyorsítótár számára. Az L3 gyorsítótár gyorsítja a fő memória működését, ami viszont a mágneslemez gyorsítótára. A hálózatba kapcsolt számítógépek elosztott fájlrendszere számára a helyi mágneslemez a hálózat többi számítógépe mágneslemezének a gyorsítótára.

## 1.7. A hardvert kezelő operációs rendszer



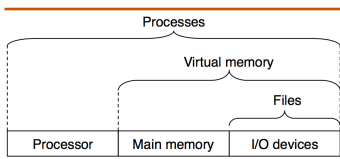
1.9. ábra. A számítógépes rendszer rétegszerkezete.

©[1] 2013

Térjünk vissza a **hello** példánkhoz. Amikor a parancsértelmező betöltötte a **hello** programot, és a **hello** program kinyomtatta az üzenetét, a program a billentyűzet, a kijelző, a mágneslemez vagy a fő memória egyikével sem lépett közvetlenül kapcsolatba. Ehelyett az **operációs rendszer** szolgáltatásaira hagyatkoztak. Az operációs rendszert olyan szoftver rétegnek tekinthetjük, amelyik az alkalmazói program és a hardver között helyezkedik el, lásd 1.9 ábra. A felhasználói program kizárólag az operációs rendszeren keresztül férhet hozzá a hardverhez.

Az operációs rendszernek kettős célja van:

- a hardvert megvédeni egy elszabadult alkalmazás kártevésétől
- az alkalmazások számára egyszerű és egységes mechanizmust kínálni a bonyolult és típusonként erősen eltérően kezelendő alacsony szintű eszközök kezelésére



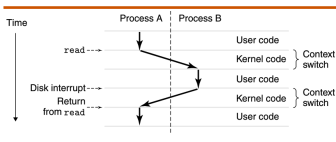
1.10. ábra. Az operációs rendszer által biztosított absztrakciók

©[1] 2013

Az operációs rendszer ezt a két célt az 1.10 ábra szerinti absztrakciók alkalmazásával éri el: bevezeti a folyamat, a virtuális memória és a fájl fogalmát. Amint azt az ábra sugallja, a fájlok az I/O eszközök, a virtuális memória a fő memória és a mágneslemez I/O eszközök, a

folyamat pedig a processzor, a fő memória és az I/O eszközök absztrakciója. Vegyük sorra ezeket.

### 1.7.1. Folyamatok



#### 1.11. ábra. A folyamatok környezetváltása

©[1] 2013

Amikor egy program, például a **hello**, fut egy modern rendszerben, az operációs rendszer igyekszik azt a látszatot kelteni, hogy az operációs rendszer számára ez az egyetlen folyamat létezik. Úgy tűnik, a program kizárólagosan használja a processzort, a fő memóriát és az I/O eszközöket. Úgy tűnik, hogy a processzor a program utasításait hajtja végre, egyiket a másik után, megszakítás nélkül. Továbbá, hogy a program kódja és adatai az egyetlen objektum a rendszer memóriájában. Ezt az illúziót a folyamat (process) szolgáltatja, ami a számítógép tudomány egyik legfontosabb és legsikeresebb fogalma.



A **folyamat** (process) egy futó program absztrakciója az operációs rendszerben. Egyazon rendszerben több folyamat is futhat konkurrens módon, és mindegyik folyamat látszólag kizárólagosan használja a hardvert. A konkurrens végrehajtás alatt azt értjük, hogy az egyik folyamat utasításai közé beiktatódnak egy másik folyamat utasításai. A legtöbb rendszerben több folyamat van, mint olyan CPU, amelyen futhatnak. A hagyományos rendszerek egyidejűleg csak egyetlen programot futtathatnak, az újabb többmagos processzoros rendszerek viszont több programot képesek egyidejűleg futtani. Ilyen esetekben a CPU több folyamatot futtat konkurrens módon, a processzort a folyamatok között kapcsolgatva. Ezt az operációs rendszer végzi, a környezet átkapcsolás (context switching) mechanizmus használatával. A továbbiakban az egyszerűség kedvéért egyprocesszoros, egyetlen CPUt tartalmazó rendszereket vizsgálunk.

Az operációs rendszer nyomon követi azt

az állapot információt, ami ahhoz szükséges, hogy a folyamat futni tudjon. Ez az állapot, amit környezetként (context) is ismernek, olyan információt tartalmaz, mint a PC aktuális értéke, a regiszter tömb, és a fő memória tartalma. Bármely időpillanatban, egy egyprocesszoros rendszer csak egyetlen folyamat kódját képes végrehajtani. Amikor az operációs rendszer úgy dönt, hogy a vezérlést az aktuális folyamatból egy másikba viszi át, egy környezet váltást (context switch) végez, amelynek során elmenti a jelenlegi folyamat környezetét, visszaállítja az új folyamat környezetét és a vezérlést az új folyamatnak adja át. Az új folyamat ilyen módon pontosan ott folytatja, ahol előzőleg abbahagyta. Az elképzelést a **hello** példaprogram esetén az [1.11](#) ábra mutatja. Példánkban két konkurrens folyamat szerepel: a parancsértelmező (shell) folyamat és a **hello** folyamat. Kezdetben a parancsértelmező folyamat egyedül fut, és arra vár, hogy a parancssorból adat bemenetet kapjon. Amikor a **hello** program

futtatását kérjük, a parancsértelmező egy rendszerhívásként ismert speciális függvény meghívásával hajtja azt végre, amely rendszerhívás a vezérlést az operációs rendszernek adja át. Az operációs rendszer elmenti a parancsértelmező környezetét, létrehozza a **hello** folyamatot és annak környezetét, majd átadja a vezérlést az új **hello** folyamatnak. Miután **hello** befejeződik, az operációs rendszer visszaállítja a parancsértelmező környezetét és visszaadja annak a vezérlést, miután az várja a következő utasítássort.

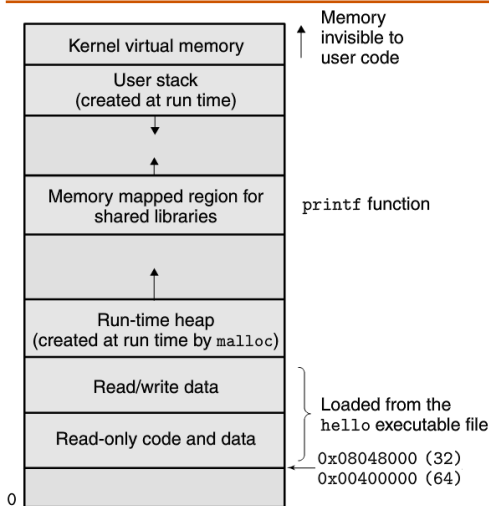
A folyamat absztrakció megvalósítás szoros együttműködést igényel az alacsony szintű hardver és az operációs rendszer szoftver között.

## **Szálak**

Bár rendszeresen úgy gondolunk a folyamatra, hogy annak egyetlen vezérlési folyama van, a modern rendszerekben egy folyamat több végrehajtási egységből (ún. **szálból**, **thread**)

állhat, amely szálak mindegyike a folyamat környezetét használva fut és ugyanazokat a globális adatokat és kódot használja. A szál alapú programozás növekvő jelentőségű, mivel a konkurens feldolgozásra nagyon sok helyen szükség van, és sokkal könnyebb közöttük adatokat megosztani, mint folyamatok között, továbbá jóval hatékonyabb megvalósításuk is. A többszálú megvalósítás jó lehetőséget kínál arra is, hogy gyorsítsuk programunk futását, ha több processzor áll rendelkezésünkre.

## 1.7.2. Virtuális memória



1.12. ábra. Egy folyamat virtuális címtere

©[I] 2013

A **virtuális memória** olyan absztrakció, amely minden folyamatot azzal az illúzióval ruház fel, hogy az illető folyamat a fő memóriát kizárólagosan használja. Ennek megfelelően

mindegyik folyamat egyformán látja a memóriát, amit saját virtuális címtérként kezel, pl. Linux esetén az 1.12 ábrán mutatott módon. Linux esetén a címtér legfelső része az operációs rendszer számára van fenntartva, ahol az operációs rendszer azon kód és adat részeit tárolja, amiket valamennyi folyamat közösen használ. A címtér alsó részében található a felhasználói folyamat által definiált kód- és adat részek. Megjegyezzük, hogy az ábrán a címek alulról felfelé növekszenek.

Az egyes folyamatok által látott virtuális címtér több jól-definiált területből áll, amelyek meghatározott célt szolgálnak:

- **Program kód és adat** A kód valamennyi folyamat számára ugyanazon a rögzített címen kezdődik, ezt követik azok a memória helyek, amelyek globális C változóknak felelnek meg. A kód és adatterületek a végrehajtható objekt fájl (a **hello**) alapján kapnak kezdőértéket.
- **Heap** A kód- és adat területeket közvetlenül követi a futási időben használ-

ható dinamikus memória tartománya, a **heap**. A kód- és adat területektől eltérően, amelyeknek mérete a folyamat futásának elkezdésekor rögzítődik, a heap memória dinamikusan kiterjed és összehúzódik, a C sztenderd könyvtár olyan rutinjainak hatására, mint a **malloc** és a **free**.

- **Megosztott könyvtárak** A címtér közepe táján található egy olyan terület, amely olyan megosztott könyvtárakat tartalmaz, mint a sztenderd C könyvtár vagy a **math** könyvtár. A megosztott könyvtár nagyon hatékony, de nem egyszerű koncepció.
- **Verem memória** (Stack) A verem memória tetején található a felhasználói verem memória, amit a fordítóprogram függvényhívások megvalósítására használ. A heap memóriához hasonlóan, a felhasználói verem memória is dinamikusan kiterjed és összehúzódik a program végrehajtása során. Nevezetesen, minden

függvény híváskor kiterjed és minden visszatéréskor összehúzódik.

- **Kernel virtuális memória** A kernel az operációs rendszer azon része, amelyik mindig a memóriában található. A memória címtér felső része a kernel számára van fenntartva. A felhasználói programok számára nem engedélyezett ezt a memóriaterületet közvetlenül írni és olvasni, vagy onnét függvényt hívni.

A virtuális memória működéséhez a hardver és az operációs rendszer szoftver bonyolult kölcsönhatására van szükség, amelybe beletartozik a processzor által előállított címek mindegyikének hardveres "lefordítása" is. Az alapötlet, hogy a folyamat virtuális memóriáját mágneslemezen tároljuk, majd a fő memóriát használjuk a mágneslemez gyorsítótáraként.

## **Fájl**

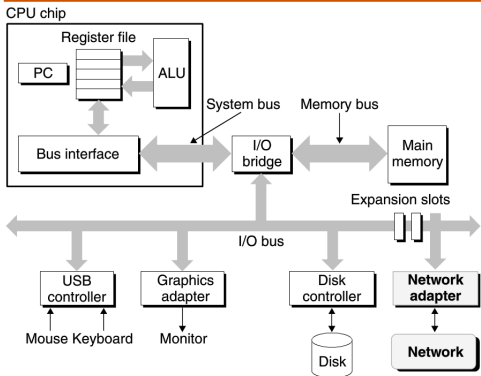
A fájl (file) egy bájtsorozat, sem több, sem



*A rendszerek hálózaton kommunikálnak egymással* 53  
Kevésbé. Valamennyi I/O eszközt, beleértve mágneslemezt, billentyűzetet, kijelzőt, sőt még a hálózatot is, fájlként modellezünk. A rendszerben valamennyi kivitel és bevétel fájl írásával és olvasásával valósul meg, rendszerhívások egy kis csoportját használva (Unix I/O).

A fájl eme egyszerű és elegáns fogalma nagyon hatékony is, mivel lehetővé teszi az alkalmazások számára, hogy a rendszerben előforduló különféle I/O eszközöket egyformán tekinthessük. Például, egy alkalmazás fejlesztő programozó, aki egy mágneslemez fájl tartalmát manipulálja, teljesen figyelmen kívül hagyhatja az éppen használt mágneslemez technológiát. Ezen kívül, ugyanaz a program különböző rendszereken, eltérő mágneslemez technológiák esetén is futni fog.

# 1.13. A rendszerek hálózaton kommunikálnak egymással



1.13. ábra. A hálózat egy másfajta I/O eszköz

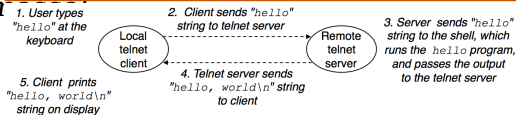
©[1] 2013

Eddig a számítógépes rendszereket egyszerűen hardver és szoftver elemek egymástól elszigetelt összességének tekintettük. A gyakorlatban azonban a modern rendszerek egymással hálózaton keresztül össze vannak kapcsolva. Az egyedi rendszerek szempont-

A rendszerek hálózaton kommunikálnak egymással. A hálózat csupán egy másfajta I/O eszköznek tekintendő, lásd 1.13 ábra. Amikor az egyik rendszer egy bájtsorozatot másol a fő memóriából a hálózati adapterre, az adatfolyam az egyik gépről a másikra kerül, mondjuk, egy helyi mágneslemez helyett. Hasonló módon, a rendszer tud elolvasni olyan adatokat, amelyeket egy másik számítógép küldött, és azokat az adatokat a fő memóriába tudja másolni.

Az Internet-szerű globális hálózatok fejlődésével az adatmásolás egyik számítógépről egy másikra a számítógépes rendszerek egyik fő alkalmazásává vált. Például, az elektromos levél vagy azonnali üzenet küldése, a World Wide Web, FTP és telnet mind a hálózaton át való másolás képességén alapulnak.

Visszatérve **hello** példánkhoz, a szokásos **telnet** alkalmazást használva, a **hello** programot egy távoli számítógépen is futtathatjuk. Tételezzük fel, hogy a **telnet** klienst futtatjuk helyi számítógépünkön, hogy a megfelelő **telnet** kiszolgálót elérjük a távoli számítógépen.



1.14. ábra. A `hello` futtatása hálózaton át egy távoli számítógépen a **telnet** használatával

©[1] 2013

Miután bejelentkeztünk a távoli számítógépen és egy parancsértelmezőt futtatunk, a távoli parancsértelmező arra vár, hogy egy bemenő parancsot kapjon. Ettől a ponttól kezdve, a **hello** program távoli futtatása a 1.14 ábrán mutatott alapvető öt lépésből áll.

Miután leírtuk a "hello" szöveget a **telnet** kliensnek és megnyomtuk az "enter" gombot, a kliens elküldi a sztringet a **telnet** kiszolgálónak. Miután a telnet kiszolgáló megkapja a hálózaton át a sztringet, átadja azt a távoli parancs értelmező programnak. Ezután a távoli parancs értelmező futtatja a **hello** programot, és annak kimenő üzenetét átadja a telnet kiszolgálónak. Végül a telnet kiszolgáló

a kimeneti sztringet a hálózaton keresztül eljuttatja a telnet kliensnek, amelyik kiírja azt a helyi képernyőn.

Ez a fajta üzenet csere a kiszolgáló és az ügyfél között jellemző a hálózati alkalmazásokra.

## 1.9. Fontos egyebek

Ezzel végére is értünk bevezető körutazásunknak. Amit az itt tárgyaltakból feltétlenül le kell vonnunk, az az, hogy a számítógépes rendszer több mint csupán hardver. Az szorosán egymáshoz kötődő hardver és rendszer szoftver összessége, amelyeknek együtt kell működniük annak a végső célnak az elérésében, hogy alkalmazói programokat futtassanak. A későbbiekben megtárgyaljuk ennek a hardver és szoftver részleteit, és bemutatjuk, hogy ezen részletek ismeretében hogyan írhatunk olyan programokat, amelyek gyorsabbak, megbízhatóbbak és biztonságosabbak. A fejezet lezárásaként még bemutatunk pár olyan fogalmat, amelyek a számítógépes rendszerek szinte minden vonatkozásában jelen vannak.

### 1.9.1. Konkurens és párhuzamos végrehajtás

A digitális számítógépek történetében két igény szolgált állandó hajtóerőként a tökéletesítésben: hogy mind többet hajtsanak végre és hogy mind gyorsabban. Mindkét említett tényező javul, amikor a processzor több dolgot hajt végre egyidejűleg. A következőkben a **konkurens végrehajtás** kifejezést használjuk, amikor egy rendszer többféle, egyidejű aktivitást végez, és **párhuzamos végrehajtásról** beszélünk, amikor a konkurens végrehajtást a rendszer gyorsabb futtatására használjuk. A párhuzamosságot egy rendszerben többféle szinten is használhatjuk egy számítógép rendszerben. Az alábbiakban három szintű párhuzamosságra világítunk rá, a rendszer hierarchia legmagasabb szintjétől a legalacsonyabbig.

#### **Szál-szintű konkurens végrehajtás**

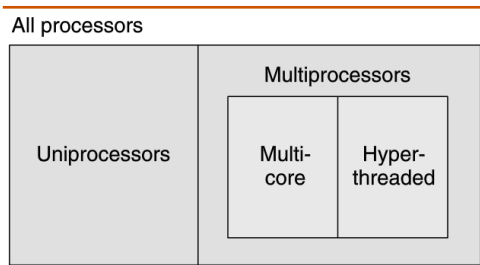
A folyamat absztrakcióra építve, képesek

vagyunk megérteni azokat a rendszereket, amelyekben több program hajtódik végre egyidejűleg, ami konkurrens végrehajtást eredményez. A szálak használatával egyetlen folyamaton belül több vezérlési folyam is lehetséges. A konkurrens végrehajtás a számítógépekben az 1960-as évek, az időosztás feltalálása óta létezik. Hagyományosan, a konkurrens végrehajtást csak szimulálták, egyetlen számítógépet kapcsolgatva a futó folyamatok között. Ez a fajta konkurrens végrehajtás lehetővé teszi, hogy egyidejűleg több felhasználó legyen kapcsolatban a rendszerrel, de akár egyetlen felhasználó is többféle feladatot futtathat. Egészen mostanáig a tényleges számítás egyetlen processzor végezte, és azt kapcsolgatni is kellett a több feladat között. Ezt a konfigurációt nevezik egyprocesszoros (uniprocessor) rendszernek.

Amikor olyan rendszert hozunk létre, hogy több processzor van egyetlen operációs rendszer kernel irányítása alatt, azt többprocesszoros (multiprocessor) rendszernek hív-



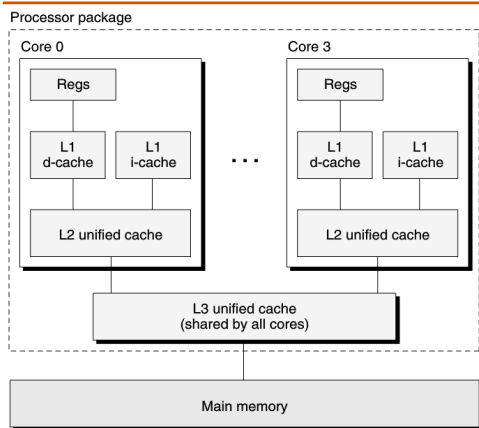
jük. Ilyen rendszerek a nagyobb skálájú számítások céljára az 1980-as évek óta léteznek, de széleskörűen csak a többmagos processzorok és az ún hyper-threading megjelenésével terjedtek el. Az 1.15 ábra mutatja a processzortípusok elnevezési rendszerét.



1.15. ábra. A különböző processzor típusok kategorizálása. A multiprocesszorok a többmagos processzorok és a hyper-threading megjelenésével váltak dominánssá.

©[I] 2013

A többmagos processzorok több CPU-t (ezeket hívjuk "mag"-nak) tartalmaznak egy áramköri tokba integrálva.



1.16. ábra. Az Intel Core I7 szerkezete. Négy processzor van egyetlen áramköri tokba integrálva.

©[I] 2013

Az Intel Core I7 processzor, lásd 1.16 ábra, négy CPUt tartalmaz, amelyeknek mindegyike saját L1 és L2 gyorsítótárral rendelkezik, de a magasabb szintű gyorsítótárak már közösek, éppúgy, mint a fő memória interfésze. Az ipari szakértők szerint akár processzorok százait

lehet egyetlen áramkörü tokba sűríteni.

A hyperthreading, amit időnként egyidejű többszálásításnak is hívnak, olyan technika, amely lehetővé teszi, hogy egyetlen CPU több vezérlési szálát futtasson egyidejűleg. Ezekben a CPU hardver egy része – például programszámláló és regiszter tömb – több példányban, a hardver többi része – például a lebegőpontos aritmetika – egy példányban van jelen. Amíg egy hagyományos processzor kb. 20,000 órajelet igényel az egyik szálról egy másikra való átálláshoz, a hyperthreaded processzor órajelenként eldöntheti, melyik szálát futtatja. Ez a módszer lehetővé teszi, hogy jobban kihasználja számítási erőforrásait. Például, ha az egyik szálnak várakozni kell arra, hogy egy adat a gyorsítótárba töltődjön, a CPU közben egy másik szálát futtathat. Például az Intel Core I7 processzorban mindegyik mag két szálát futtathat párhuzamosan, azaz egy négymagos rendszer összesen nyolcat.

A multiprocessing kétféle módon javíthatja a rendszer teljesítményét. Először is, csök-

kenti a konkurrens működés szimulálásának szükségletét, amikor több feladatot kell végrehajtani. Mint említettük, még a személyi számítógépek felhasználói is több feladatot végeznek egyidejűleg. Másodsor, akár egyetlen alkalmazást is gyorsabbá tehet, feltéve, hogy a program több szálat használ, amelyek valóban párhuzamosan futhatnak. Ilyen módon, bár a konkurrens működés elveit fél évszázada használják és kutatják, a többmagos és multithreading rendszerek nagyon megnövelték annak igényét, hogy olyan alkalmazói programokat írjunk, amelyek valóban kihasználják a hardver által kínált szál-szintű párhuzamosságot.

## **Utasítás szintű párhuzamos végrehajtás**

Egy sokkal mélyebb absztrakciós szinten, a modern processzorok képesek egyidejűleg több utasítást végrehajtani, ami tulajdonságot [utasítás-szintű párhuzamosság \(instruction-level parallelism\)](#) néven ismerünk. A korai

mikroprocesszorok (például az Intel 8086 is) több (tipikusan 3-10) órajel ciklus alatt hajtottak végre egy utasítást. Az újabb processzorok 2-4 utasítást végeznek el egy órajel ciklus alatt. Egy adott utasítás sokkal több időt követel, a kezdettől a végéig, akár 20 vagy még több órajel ciklust, a processzor azonban okos trükkökkel akár 100 utasítást is végrehajthat egyidejűleg. Tanulmányainkban már találkozhattunk a csővezetékezés (pipelining) módszerével, ahol is a szükséges műveleteket lépésekre bontják és a processzor hardvert állomások sorozataként szervezik meg, ahol mindegyik állomáson ezen lépések egyikét végzik el. Az állomások párhuzamosan működhetnek, miközben a különböző utasítások különböző részein dolgoznak. Látni fogjuk, hogy egy viszonylag egyszerű hardveres felépítéssel közel egy utasítás per órajel végrehajtási sebességet lehet fenntartani.

Azokat a processzorokat, amelyek egy utasítás per órajelnél nagyobb végrehajtási sebességet tudnak biztosítani, [szuperskaláris](#)

processzornak nevezzük. A legtöbb modern processzor támogatja a szuperskaláris működési módot.

## **Egy utasítás, több adat párhuzamosság**

A legalacsonyabb szinten, sok modern processzor rendelkezik olyan speciális hardverrel, amelyik lehetővé teszi, hogy egyetlen utasítás több, párhuzamosan elvégzett műveletet okozzon, ami módot egy utasítás, több adat párhuzamosság (single-instruction, multiple-data, or "SIMD" parallelism) ismerünk. Például, a jelenlegi Intel és AMD processzoroknak vannak olyan utasításai, amelyek négy pár egyszeres pontosságú lebegőpontos számot (C `float` adattípus) képesek összeadni párhuzamosan. Ezeket a SIMD utasításokat azzal a szándékkal hozták létre, hogy felgyorsítsák a kép-, hang- vagy video feldolgozással foglalkozó alkalmazásokat. Bár bizonyos fordítóprogramok megpróbálják a SIMD párhuzamosságot kivonni a C prog-

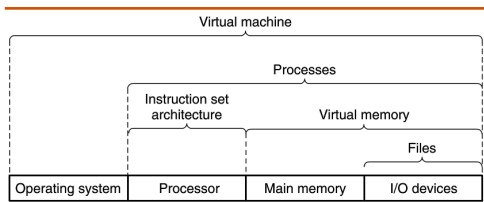
ramokból, biztosabb módszer olyan speciális vektoros adattípusokat használni a programírásakor, amelyeket a fordítóprogram (pl. **gcc**) támogat.

### 1.9.2. Számítógépes absztrakciók

A számítógép tudományban az absztrakciók használata az egyik legfontosabb követelmény. Például, a helyes programozási gyakorlatnak az egyik vonatkozása a rendelkezésre álló funkcionalitáshoz egy olyan egyszerű programozási interfész (application-program interface, API) megfogalmazása, amelyik lehetővé teszi, hogy a programozók a kód belső működésének ismerete nélkül használhassák magát a kódot. A különböző programnyelvek különböző formákat és támogatási szinteket biztosítanak az absztrakció számára, például a Java osztály deklarációi vagy a C függvény prototípusai. A számítógépes rendszerekben használt néhány absztrakciót már megismertünk, lásd 1.17 ábra. A processzor oldaláról, az utasítás készlet architektúra a tényleges processzor hardver egy absztrakciója. Ezzel az absztrakcióval, egy gépi kódú program látszólag úgy viselkedik, mintha egy olyan processzoron hajtódna végre, amelyik egyide-



júleg csak egyetlen utasítást hajt végre. A tényleges hardver ennél sokkal bonyolultabb, több utasítást hajt végre egyidejűleg, de olyan módon, amelyik az egyszerű, soros végrehajtási modellel konzisztens. A végrehajtási modell megtartásával, különböző processzor megvalósítások is végrehajthatják ugyanazt a gépi kódot, miközben költségekben és működési hatékonyságban jelentősen eltérhetnek.



1.17. ábra. A számítógépes rendszerek által kínált néhány absztrakció. A számítógép rendszerek használatakor nagyon fontos szempont a különböző szinteken absztrakt ábrázolásokat biztosítani, amivel elrejthetjük a tényleges megvalósítás bonyolultságát.

Az operációs rendszer oldaláról három absztrakciót vezettünk be: a **fájlt** mint az I/O , a **virtuális memóriát** mint a program memória és a **folyamatot** mint a futó program absztrakcióját. Most egy újat adunk ezekhez az absztrakciókhoz: a **virtuális számítógép** fogalmát, ami az egész számítógép (az operációs rendszert, a processzort és a programokat magában foglaló) absztrakciója. A virtuális számítógép fogalmát az IBM az 1960-as években vezette be, de csak mostanában vált széles körűen elterjedtté, mint egy olyan módszer, amelyikkel lehetővé válik többféle operációs rendszerre (úgy mint Microsoft Windows, MacOS és Linux) tervezett programok vagy ugyanazon operációs rendszer többféle változatának futtatása.

## 1.10. Összefoglalás

Egy számítógépes rendszer hardver és szoftver komponensekből áll, amelyek együttműködve futtatják a felhasználói alkalmazásokat. A számítógép belsejében az információt bit csoportok ábrázolják, amelyeket különbözőképpen értelmezhetünk, összefüggéstől függően. A programokat más programok fordítják különféle formátumokra, amelyek ASCII szöveggént kezdik életüket és a fordító és csatoló programok fordítják bináris végrehajtható fájlkká.

A processzorok elolvassák és értelmezik a végrehajtható utasításokat, amelyek a fő memóriában tárolódnak. Mivel a számítógépek idejük nagy részét azzal töltik, hogy adatot másolgatnak a memória, az I/O eszközök és a CPU regiszterei között, a tárolóeszközöket olyan hierarchiába szervezik, amelynek tetején a CPU regiszterek állnak, azokat a gyorsítótárak különféle szintjei követik, majd a DRAM fő memória és a mágneslemez követ-

kezik. Az ebben a hierarchiában magasabban fekvő eszközök gyorsabbak és fajlagosan (egy bitre számítva) költségesebbek, mint a hierarchiában mélyebben fekvők. A hierarchiában magasabban fekvő eszközök gyorsítótárként szolgálnak a hierarchia alacsonyabb szintjén található eszközök számára. A programozók optimalizálni tudják C programjuk működését, ha megértik és felhasználják ezt a memória hierarchiát.

Az operációs rendszer kernel közvetítői szerepet játszik az alkalmazás és a hardver között. Három alapvető absztrakciót használ

- A fájlok az I/O eszközök absztrakciója
- A virtuális memória a fő memória és a mágneslemez absztrakciója
- A folyamatok a processzor, a fő memória és az I/O eszközök absztrakciója

Végezetül, a hálózat nyújt módot arra, hogy a számítógépek egymással kommunikáljanak. Az egyes rendszerek szempontjából, a hálózat csupán egyfajta I/O eszköz.



---

# Tárgymutató

adapter, 4

application-

program in-  
terface, API,

11

arithmetic/logic

unit, 5

aritmetikai/logikai

egység, 5

assembly nyelvű  
program, 3

áthelyezhető objekt  
program, 3

be/kiviteli (I/O)  
eszköz, 4

bináris fájl, 2

binary file, 2

busz rendszer, 4

cache, 6

cache memories, 6

central processing  
unit, 4

direct memory ac-  
cess, DMA,  
5

egy utasítás, több  
adat párhuzamosság,  
10

fő memória, 4

folyamat, 7

forrás fájl, 2

gyorsítótár, 6

heap, 8

instruction set ar-  
chitecture,  
4

instruction-  
level paral-  
lelism, 10

központi feldolgozó  
egység, 4

közvetlen memória-  
elérés, 5

konkurrens végre-  
hajtás, 9

newline, 2

operációs rendszer,  
7

párhuzamos végre-  
hajtás, 9

process, 7

programozási in-  
terfész, 11

register file, 5

regiszter tömb, 5

sín rendszer, 4  
single-instruction,  
multiple-  
data, or  
“SIMD” pa-  
rallelism,  
10

source program, 2

szövegfájl, 2

szál, 7

számítógépes rend-  
szer, 2

szó méret, 4

sztenderd C könyv-  
tár, 3

szuperskaláris pro-  
cesszor, 10

text file, 2

thread, 7

új sor, 2

utasítás készlet, 4

utasítás-szintű  
párhuzas-  
mosság, 10

végrehajtható  
objekt prog-  
ram, 3

vezérlő, 4

virtuális címtér, 8

virtuális memória,  
8



---

# Táblázatok jegyzéke

1.1. A <b>hello.c</b> program (lásd 1.1 programlista) ábrázolása ASCII szöveggént . . . . .	11
---	----





---

# Ábrák jegyzéke

1.1. Egy tipikus számítógéprendszer rétegmodellje . . . . .	4
1.2. A számítógépes fordításhoz használt rendszer . . . . .	14

1.3. Egy tipikus rendszer hardveres felépítése. CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program counter, USB: Universal Serial Bus. . . . .	25
1.4. A hello utasítás beolvasása a billentyűzetről. . . . .	31
1.5. A végrehajtható fájl betöltése a mágneslemezről a fő memóriába. . . . .	32
1.6. A kimenő szöveg kiírása a memóriából a képernyőre. . . . .	34
1.7. Gyorsítótárak. . . . .	35
1.8. Példa a memória hierarchiára. . . . .	39
1.9. A számítógépes rendszer rétegszerkezete. . . . .	41
1.10Az operációs rendszer által biztosított absztrakciók . . . . .	42
1.11A folyamatok környezetváltása . . . . .	44
1.12Egy folyamat virtuális címtere . . . . .	49
1.13A hálózat egy másfajta I/O eszköz . . . . .	54
1.14A hello futtatása hálózaton át egy távoli számítógépen a <b>telnet</b> használatával . . . . .	56

- 1.15A különböző processzor típusok kategorizálása. A multiprocesszorok a többmagos processzorok és a hyper-threading megjelenésével váltak dominánssá. . . . . 61
- 1.16Az Intel Core I7 szerkezete. Négy processzor van egyetlen áramkörtokba integrálva. . . . . 62
- 1.17A számítógépes rendszerek által kínált néhány absztrakció. A számítógép rendszerek használatakor nagyon fontos szempont a különböző szinteken absztrakt ábrázolásokat biztosítani, amivel elrejtjük a tényleges megvalósítás bonyolultságát. . . . . 69



---

# Programlisták

1.1	A "Hello Világ" program forrás- kódja . . . . .	9
-----	--	---





---

# Bibliography

- [1] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Pearson, 2014. ISBN: 978-1-292-02584-1.
- [2] Irv Englander. *The Architecture of COMPUTER HARDWARE, SYSTEMS SOFTWARE AND NETWORKING An Information*

*Technology Approach*. Fourth. John Wiley & Sons, Inc., 2010. ISBN: 978-0-470-40028-9.

- [3] Irv Englander. *The Architecture of COMPUTER HARDWARE, SYSTEMS SOFTWARE AND NETWORKING An Information Technology Approach*. <http://www.wiley.com/go/global/englander>. 2010.
- [4] Neil Matthew and Richard Stones. *Beginning Linux Programming*. <http://longfiles.com/fzi3sbsh0lhu/LinuxProgram4th147627.pdf.html>. Wrox Press Ltd, 2008. ISBN: 978-0-470-14762-7.
- [5] Clive "Max" Maxfield. *DIY Calculator*. <http://diycalculator.com/>. 2003.
- [6] Clive "Max" Maxfield. *How Computers Do Math*. John Wiley & Sons, Inc., 2005. ISBN: 0471732788.
- [7] Clive "Max" Maxfield and Alvin Brown. *The Official DIY Calculator Data Book*.

John Wiley & Sons, Inc., 2005. ISBN: 0471732788.

- [8] Stanley J. Warford. *Computer Systems*. Jones and Bartlett, 2010. ISBN: 0-7637-7144-9.