



# Bevezető

Azaz ki, miért, kinek, miről, mennyit, hogyan...

---

## ☰ Kedves sorstársak!

A nevem Buzogány László. Én állítottam össze ezt a dokumentumot valamikor 2002 decembere és 2003 januárja között, a szesszió előtti nagy készülődésben. Akkor harmadéves egyetemista voltam.

Miért? Egy 10-es vizsgajegyért az Operációs rendszerek (2) tantárgyból. Ez volt a fő ok, de természetesen egyéb más hasznom is származott belőle. Egyrészt miközben leírtam e tömérdek mennyiségű információt, egy-két apróság rám is ragadt, másrészt használható dokumentumot nyertem a laborfeladatok megoldásához, harmadrészt nem a szesszióban kellett végigtanulnom ezt az anyagrészt, egyszóval jó befektetésnek bizonyult.

Azért ne higgyétek, hogy olcsóbban megúsztam mint Ti! Ugyanis a dokumentum összeállítása, begépelése, javítása körülbelül három hetet és közel 125 órát vett igénybe, ami szerintem tökéletesen elegendő, ahhoz, hogy valaki 10-esre fel tudjon készülni a vizsgára. S még így sem bánom.

Minden hasznomtól eltekintve, kívánom, hogy a dokumentum mindenkinek hasznára váljon, és segítse a vizsgára való felkészülésben és a laborfeladatok megoldásában. Mindezek mellett, ha valakinek bármilyen megjegyzése, javítani valója van, azt nyugodtan eszközölje az oldalakon, vagy írja meg nekem e-mailben. A lényeg, hogy e forrás az évek során helyes, teljes és használható segédeszközzé váljék.

Nem tagadom, hogy a dokumentum egyes részei nagyon hasonlítanak a Iosif Ignat és Adrian Kacso *UNIX – Gestionarea proceselor* című könyvében ("a kék könyvben") látottakhoz, hiszen ez a könyv szolgált a dokumentum alapjául. Ez alatt természetesen nem teljes fordítás értendő. Igyekeztem a lényegét és a használhatót megkeresni, magamban átrágni, s ahol szükséges volt, egyéb forrásból utánaolvasni az információknak. Éppen ezért a dokumentumban előfordulhatnak a fordításból és a begépelésből származó hibák. Ezekért elnézést kérek, de felelősséget nem vállalok értük!

A forrás összesen 17 darab htm formátumú Microsoft FrontPage-dzsel szerkesztett lapból tevődik össze. A kezdőlap index.htm. Innen indulnak az általában egy-két mélységű oldalak. Az oldalak közti navigálást a lap tetején található gombok segítik, de a szerkezet amúgy sem bonyolult. A linkeket (hivatkozásokat) mindenütt aláhúzás jelzi. A címeket sárga, az alcímeket, a kifejezéseket és az utasításokat fehér, míg a kódot halványsárga betűkkel írtam. A kódrészletek ezenkívül Courier New betűtípussal vannak kiemelve.

Minden oldal felső részén rövid tartalomjegyzék található a lap tartalmáról. A szöveg előtti kis ikonok jelzik a leírás helyét az aktuális pozícióhoz viszonyítva. Külön logóval láttam el a leíró részek címeit is.

A dokumentum utolsó két fejezete a Függelék és az Index. A Függelékben két összefoglaló táblázat található, míg az Index a dokumentumban szereplő kifejezések és utasítások gyors megkeresését teszi lehetővé.

Sok sikert a böngészésben és a vizsgán!

## ➡ Mi a folyamat?

---

Copyright (C) Buzogány László, 2002



[About](#)



# Mi a folyamat?

A fájlrendszertől a folyamatokig...

---

## 📁 Fájlrendszer

- 📁 [Általános jellemzők](#)
- 📁 [Névkonvenciók](#)
- 📁 [Könyvtárak \(katalógusok\)](#)
- 📁 [Általános könyvtárnevek](#)
- 📁 [Fájlkezelő utasítások](#)

## 📁 Folyamatok

- 📁 [A folyamat fogalma](#)
  - 📁 [Folyamatkezelő utasítások](#)
  - 📁 [Folyamatkezelés a Korn shellben \(ksh\)](#)
- 

## 📁 Bevezető

### 📁 Fájlrendszer

#### 📁 Általános jellemzők

A UNIX fájlrendszere a DOS-szal összehasonlítva sokkal egységesebb: felfogása szerint minden fájl. Ez azt jelenti, hogy a felhasználó fájljai, a rendszer könyvtárkatalógusai és a rendszerhez csatlakoztatott hardver eszközök egységesen kezelendők és egy könyvtárfára vannak "felakasztgatva".

A fájl – Unix felfogás szerint – egy strukturálatlan bájt-sorozat. Nincsen semmilyen rá vonatkozó megkötés, nincs fájlvég-jel stb. Minden fájlnak van tulajdonosa, nyilván van tartva, hogy a tulajdonos mely csoport tagja (csoportmunkához). A fájlokhoz olvasási, írási, végrehajtási jogok kötődnek, amelyek külön beállíthatók a tulajdonos, a csoportja és mindenki más számára (lásd chmod). Ezenkívül a rendszer nyilvántartja a fájl hosszát, az utolsó módosítás időpontját valamint, hogy a fájlrendszer hány pontjáról hivatkozunk (könyvtárbejegyzéssel) erre a fájlra (linkszám).

#### 📁 Névkonvenciók

Unixban a fájlnevek (manapság, általában) 255 karakter hosszúak lehetnek, tetszőleges számú elválasztó karaktert tartalmazhatnak (.), sőt érvényes név a ponttal kezdődő név is. Ezeket a rendszer

némileg rejtettként értelmezi; például alapértelmezésben nem listázza és nem is törli őket (lásd `ls` parancs). Az ilyen fájlok gyakran különféle programok inicializációs adatait tartalmazzák.

A Unix minden esetben (paraméterekben is) megkülönbözteti a kis és nagybetűket, így például nem azonos `file`, `File` és `FILE`.

A futtatható állományokra nincs névbeli megkötés, akkor futtathatunk valamit, ha van rá futtatási jogunk (lásd `chmod`). A futtatás a név beírásával történik, ekkor a shell megkísérli bináris fájlként értelmezni és végrehajtani, ha ez nem megy, akkor shell scriptként értelmezi (a shell scriptek a DOS batch fájljaihoz állnak a legközelebb.)

## ☰ Könyvtárak (katalógusok)

A UNIX hierarchikus felépítésű, ami azt jelenti, hogy a fájlokat könyvtárakban tárolja. Egy könyvtárból alkönyvtárak nyílhatnak, amelyek ugyancsak tartalmazhatnak további fájlokat és alkönyvtárakat. A gyökérkönyvtárnak nincs neve és szülő könyvtára. A `/` jel jelenti a gyökeret (root), alkönyvtárai pedig az `usr`, `home` stb. Ezek a `/usr`, `/home` stb. hivatkozással érhetők el. Ezt a hivatkozást elérési útvonalnak (path name) hívják. Ez független attól az alkönyvtártól, amelyikben éppen tartózkodunk. A `/usr` alkönyvtárnak további alkönyvtárai vannak, például `bin`, `etc` stb. Ezek elérési útvonala `/usr/bin`, `/usr/etc` és így tovább.

## ☰ Általános könyvtárnevek

|                             |   |
|-----------------------------|---|
| <code>/</code>              | a főkönyvtár (root), a fájlrendszer kiindulópontja                                    |
| <code>~/</code>             | a felhasználó saját könyvtára   |
| <code>./</code>             | az aktuális munkakönyvtár   |
| <code>../</code>            | az aktuális fölötti könyvtár  |
| <code>/dev</code>           | a különféle eszközök könyvtára  |
| <code>/mnt</code>           | az ideiglenesen „felakasztott” eszközök becsatolási pontja (lásd <code>mount</code> ) |
| <code>/usr</code>           | felhasználói programok, forrásszövegek, könyvtárak                                    |
| <code>/usr/bin</code>       | végrehajtható programok (főként segédprogramok)                                       |
| <code>/usr/lib</code>       | utasításkönyvtárak programozóknak   |
| <code>/usr/src</code>       | forrásnyelvű szövegek   |
| <code>/usr/local</code>     | helyileg telepített programok és tartozékaik  |
| <code>/usr/local/bin</code> | helyileg telepített futtathatók   |

## ➡ Fájlkezelő utasítások

Azon utasítások gyűjteménye, amelyek a fájlrendszerben való eligazodást, navigálást, könyvtárak, fájlok létrehozását, törlését és kezelését végzik.

## ☰ Folyamatok

## ☰ A folyamat fogalma

A UNIX-ban megkülönböztetjük a program, a folyamat (process) és a feladat (job) fogalmát. A program a háttértárolón várakozó, végrehajtható fájl, a folyamat a program egy memóriában futó példánya, a job pedig egy végrehajtáshoz sorba állított utasítás (pl. nyomtatás). A felhasználó bejelentkezésekor elindul egy shell egy folyamatazonosítóval (PID, process identifier). Ha egy programot elindítunk, az leszarmazott folyamatként (child process) indul el. Az ilyen folyamatok is egyenrangúak a szülővel, bizonyos paramétereket örökölnek tőlük és maguk is újabb folyamatokat indítanak.

A munka során a kiadott parancsok a DOS-hoz hasonlóan, szekvenciálisan kerülnek végrehajtásra. A DOS-szal ellentétben itt viszont mód van ún. **háttér**folyamatok elindítására, ahol a parancs kiadása után a promptot azonnal visszakapjuk és az előzőleg kiadott parancs végrehajtása a háttérben folyik tovább.

A háttérfolyamat egy speciális fajtája a démon. Ezt a rendszer automatikusan indítja el és valamilyen felügyeleti szerepet lát el. Például gondoskodik a terminálvonalak figyeléséről, a nyomtatási kérelmek besorolásáról és végrehajtásáról.

## ☰ Folyamatkezelő utasítások

ps – futó folyamatok kiírása

ALAKJA:

ps [opciók]

Paraméter nélkül indítva egy ilyen listát kapunk:

```
$ ps
PID TTY STAT TIME COMMAND
49  p2  S    0:01 -ksh
208 p2  R    0:00 ps
$
```

A lista elemei (sorban):

- a folyamatazonosító (PID),
- mely terminálról adtuk ki,
- milyen állapotban van (S: sleeping, alszik; R: running, fut; Z: zombie),
- mennyi processzoridőt vett eddig igénybe,
- milyen utasításhoz tartozik.

A ps parancs kiadása után minden futó folyamatról tájékoztatást kapunk. A ps -f paraméterezéssel a szülő folyamatok PID-jét is kiírja a program (PPID).

kill – folyamat kiirtása

(CÉLSZERŰ) ALAKJA:

kill -9 processzazonosító

A processzorazonosítót megtudhatjuk például egy ps utasításból.

## ☰ Folyamatkezelés a Korn shellben (ksh)

A Korn shellben egy futtatható fájl nevét beírva, előtérben indítjuk el azt. Háttérben indítható egy fájl a neve mögé tett & jellel, például:

```
$ sleep 10&  
[1] 245  
$
```

Ez azt jelzi számunkra, hogy a shell 245-ös PID-del saját számozása szerint [1]-es jobbként indította el a folyamatot. Ha 10 másodperc múlva begépelünk egy parancssort, kapunk egy plusz üzenetet is:

```
$  
[1]+ Done sleep 10  
$
```

Az üzenet a folyamat befejezéséről tájékoztat bennünket.

Ha egy folyamatot előtérből háttérbe szeretnénk helyezni, előbb állítsuk le Ctrl-Z-vel azt (a program a memóriában marad!). Ekkor írjuk be:

```
$ bg
```

Ennek hatására a folyamat háttérben fog futni, mígnem az fg parancssal előtérbe nem hozzuk.

## ☞ Folyamatok közti kommunikáció

---

Copyright (C) Buzogány László, 2002



[About](#)



# Folyamatok közti kommunikáció

Azaz hogyan tud két folyamat egymás közt információt cserélni...

---

## ☰ Folyamatok jellemzői

- ☰ Apa-fiú kapcsolat
- ☰ Folyamatazonosító (PID)
- ☰ Memóriakiosztás
- ☰ Folyamatok attribútumai
- ☰ Folyamatok állapotai

## ☰ Folyamatok közti kommunikáció

- ☰ Jelzések (signals)
  - ☰ Állományok
  - ☰ Csővezeték (pipe)
  - ☰ Névvel ellátott csővezeték (FIFO)
  - ☰ Üzenetsorok (message queues)
  - ☰ Szemaforok (semaphores)
  - ☰ Osztott memória (shared memory)
- 

## ☰ Mi a folyamat?

### ☰ Folyamatok jellemzői

#### ☰ Apa-fiú kapcsolat

Amint már láttuk, folyamatnak nevezzünk egy futó programot. Unixban minden folyamatot – kivéve a 0 és az 1 folyamatokat, amelyek a rendszer indulásakor töltődnek be – egy másik folyamat indít el a fork rendszerfüggvény meghívásával. Az újonnan elindított folyamatot fiúnak (gyereknek), azt a folyamatot pedig amelyik őt elindította szülőnek (apának) nevezzük. (Előfordulhat, hogy a szülőfolyamat hamarabb befejeződik mint a gyerek, ekkor a gyerek kap egy új szülőt: az init (1) folyamatot.)

Egy multitasking rendszerben ugyanazt a programot egyidőben akár több folyamat is futtathatja, és bármelyik folyamat végrehajthat egy másik programot.

## ☰ Folyamatazonosító (PID)

A Unixban minden folyamatnak egyedi azonosítója van. Ezt a pozitív egész számot nevezzük **folyamatazonosítónak (PID)**. A számot a rendszer automatikusan adja minden új folyamat létrehozásakor. Egy folyamat lekérdezheti a saját PID-jét a **getpid** rendszerhívás segítségével. Mivel minden folyamat PID-je egyedi, nem lehet azt megváltoztatni, de a szám újra felhasználható, amint a folyamat befejeződött.

A 0 PID-del rendelkező folyamat mindig a **swapper**, az 1 azonosítójú pedig az **init** folyamat. Ezeket a rendszer induláskor automatikusan betölti, s mindvégig a memóriában maradnak.

## ☰ Memóriakiosztás

Amikor a rendszer egy folyamatot elindít létrehoz számára bizonyos adatszerkezeteket:

- kódszegmens (amennyiben a folyamatnak még nem volt futó példánya létrehoz egy új kódszegmentet, ha már volt, a létezőt fogja használni),
- adatszegmens (tartalmazza a statikus és dinamikus változókat is; lefele nő),
- veremszegmens (felfele nő).

Az adat- és veremszegmens folyamathoz kapcsolt, azaz annyi példányban jön létre, ahány futó példánya van a folyamatnak.

## ☰ Folyamatok attribútumai

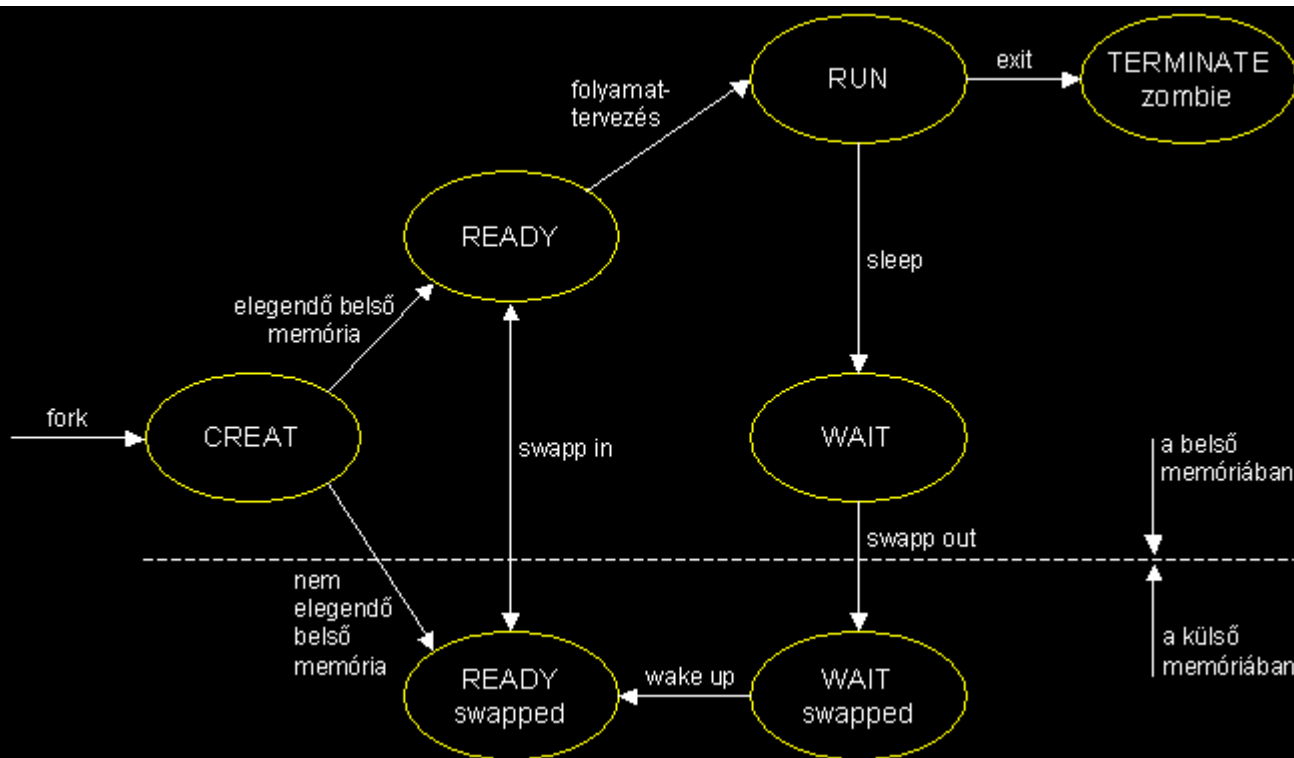
Egy folyamat több attribútummal is rendelkezik, amelyeket a megfelelő függvényekkel le is kérdezhetünk; ezek közül a legfontosabbak:

- folyamatazonosító (PID vagy ID),
- szülőfolyamat folyamatazonosítója (azé a folyamaté, amely a fork függvényhívást kiadta),
- a terminál, amelyről a folyamatot elindítottuk,
- a felhasználó azonosítója,
- a felhasználó csoportjának azonosítója,
- S jogosultság (a folyamat elindítása során megkaphatja a tulajdonosa összes jogait),
- állomány maximális mérete (milyen nagy állományt tud létrehozni),
- adatszegmens maximális mérete,
- nice érték (0 és 39 közötti szám, amelyet a folyamat prioritásának a kiszámításánál használ).

## ☰ Folyamatok állapotai

Egy folyamatnak a létrehozásától számítva több különböző állapota is lehet, attól függően, hogy éppen fut-e, várakozik-e valamilyen erőforrásra, van-e elegendő memória stb. Az állapotok közti átmeneteket az alábbi ábra szemlélteti:





Egy folyamat a fork függvényhívás után CREAT állapotba kerül. Ekkor a szülő szegmensei gyakorlatilag megduplázódnak. Ha van elegendő memória a folyamat számára átkerül READY (végrehajtható) állapotba, különben a háttértárolóra kerül és READY swapped állapotban várakozik. Amennyiben a folyamat elegendő prioritással rendelkezik READY állapotból átkerülhet RUN (futó) állapotba. Ebből az állapotból csak akkor kerül ki, ha valamilyen erőforrásra kell várakoznia (WAIT). Ha ezután fogyni kezd a memória szintén a háttértárolóra kerül WAIT swapped állapotba. A folyamat feladata befejeztével TERMINATE (zombie) állapotba kerül, ami azt jelenti, hogy a folyamattáblában még megmarad amíg a szülője fut.

### ☐ Folyamatok közti kommunikáció

Unix alatt egy felhasználó akár több folyamatot is elindíthat – lényegében minden futó program külön folyamatként fut. Egyszerre azonban csak egy folyamattal tudunk kommunikálni (az egy darab billentyűzet miatt). A háttérben futó folyamatok is kommunikálhatnak egymással.

A folyamatok közti kommunikáció (IPC, Inter-process communication) többféleképpen is megvalósulhat:

- jelzések segítségével,
- állományokon keresztül,
- csővezetékekkel (pipe),
- névvel ellátott csővezetékeken keresztül (FIFO),
- szemaforok alkalmazásával (semaphores),
- üzenetsorokkal (message queues),
- osztott memória segítségével (shared memory),
- socket,
- stream.

Az utolsó két módszert főként olyankor használjuk, amikor a két folyamat nem ugyanazon a gazdagépen fut.

## ☰ Jelzések (signals)

A jelzéseket főként más folyamatok figyelmeztetésére, riasztására, vagy kivételek kezelésére használjuk. Ezen üzenetek nagyon kevés információt hordoznak, mivel véges számú jelből állhatnak. A jelzés hatására a folyamat megszakad és értelmezi az adott jelet. Alkalmazásuk nem túl hatékony, főleg folyamatok befejezésére használjuk.

## ☰ Állományok

Az állományok segítségével a folyamatok viszonylag könnyen tudnak adatokat cserélni. Például az egyik folyamat ír egy fájlba, a másik folyamat pedig olvassa azt, visszafele pedig épp fordítva – természetesen egy másik állományt használva. Enne a módszernek két nagy hátránya van:

- ha a két folyamat egyidőben dolgozik, nem tudnak megfelelően összehangolódni (például az egyik folyamat a fájl végét hibásan érzékeli, s a kommunikáció megszakad),
- ha a kommunikáció hosszú időn keresztül tart, az állomány mérete jelentősen megnőhet.

## ☰ Csővezeték (pipe)

A csővezetékek (pipe-ok) már megoldják a fájlknál fellépő szinkronizációs problémákat. A pipe a rendszer által létrehozott (név nélküli) speciális állomány. Maximális hossza 10 blokk (5120 byte). Feladata: két folyamat közötti kommunikáció megvalósítása (az egyik folyamat ír a standard kimenetre, a másik pedig olvas a standard bemenetről). A pipe FILO szerkezetű. Két mutatója van: egyik az író folyamaté, a másik az olvasóé. Így az írás/olvasás cirkulárisan történik (ha a pipe megtelik az írónak kell várnia, ha kiürül az olvasónak).

Pipe-okat a parancssorban is használhatunk. Például:

```
$ ls | more
```

A pipe-on keresztül történő kommunikációnak is vannak hátrányai:

- a kommunikációban résztvevő folyamatok kötelezően apa-fiú kapcsolatban kell legyenek, vagy kell legyen egy közös ősök,
- a régebbi verziók nem engedik meg az elemi szinten történő írást és olvasást, több író/olvasó folyamat esetén,
- lassú működés.

## ☰ Névvel ellátott csővezeték (FIFO)

A névvel ellátott csővezetékek nagyon hasonlítanak a hagyományos pipe-okhoz. Egy FIFO állománynak azonban van neve és minden olyan folyamat hozzáférhet, amelynek joga van hozzá. A FIFO a második problémát is megoldotta, hiszen itt már értelmezettek az atomi műveletek is. Egyetlen hátrány maradt, az alacsony sebesség.

## ☰ Üzenetsorok (message queues)

Az üzenetek olyan kis adatcsomagok, amelyeket üzenetként küldünk el. Itt természetesen meg kell mondanunk a címzett (folyamat) azonosítóját. Az üzenetek többfélék lehetnek. Bármely folyamat – amelynek van joga – kaphat üzenetet a hálózaton keresztül.

A folyamat választhat a bejövő üzenetek közül: lehet mindig az első, az első egy bizonyos típusúból vagy az első egy típuscsoportból.

## ≡ Szemaforok (semaphores)

A szemafor egy olyan jel (például egy változó), amely megmutatja, hogy egy folyamat végrehajthat-e egy bizonyos utasítást, vagy várakoznia kell.

## ≡ Osztott memória (shared memory)

Az osztott memória a leggyorsabb kommunikációt eredményezi. Alapötlete: ugyanazt a memóriarészt használja az összes összeköttetésben levő folyamat. Minél gyorsabban beírja az egyik folyamat az információt a memóriába, annál gyorsabban tudja kiolvasni azt egy másik. Ebben az esetben is felhasználhatunk egy szemafort vagy küldhetünk üzenetet, azért, hogy a folyamatok írását/olvasását összehangoljuk.

## ➡ ANSI C

---

Copyright (C) Buzogány László, 2002



[About](#)



# ANSI C

A folyamatok kezeléséhez szükséges ANSI C utasítások összefoglalója...

---

## ⚡ [C. Hogyan?](#)

⚡ [gcc](#)

## ⚡ [Környezet, szövegszerkesztő](#)

⚡ [vi](#)

## ⚡ [Segítség](#)

⚡ [man](#)

## ⚡ [A parancssor paraméterei és a környezeti változók](#)

⚡ [argc](#) [argv](#) [envp](#)

⚡ [env](#)

⚡ [environ](#)

⚡ [getenv](#)

## ⚡ [Hibakezelés](#)

⚡ [errno](#)

⚡ [strerror](#)

⚡ [perror](#)

⚡ [err.c](#) – [err\\_sys](#) [err\\_ret](#) [err\\_quit](#) [err\\_msg](#) [err\\_dump](#)

## ⚡ [Memóriakezelés](#)

⚡ [malloc](#)

⚡ [calloc](#)

⚡ [realloc](#)

⚡ [free](#)

## ⚡ [Aktuális könyvtár váltása \(\[chdir\]\(#\)\)](#)

## ⚡ [Gyökérkönyvtár megváltoztatása \(\[chroot\]\(#\)\)](#)

---

## 📁 Folyamatok közti kommunikáció

### ☰ C. Hogyan?

A Unix alatt használható ANSI C nagyrészt megegyezik a Borland C-vel.

Különbségek, megjegyzések:

- nincs conio.h
- nincs nyomkövetés (a tesztelésre a legjobb módszer, ha több printf-et szúrunk be a programunkba)
- az ANSI C fájlok kiterjesztése kötelezően .c
- a C++ fájlok kiterjesztése kötelezően .C
- a printf csak a sorvége jel (\n) után ír a képernyőre

A fordítás a gcc nevű fordítóval történik.

```
$ gcc program.c
```

A parancs paraméter nélkül használva mindig létrehoz egy a.out nevű állományt. Ez lesz a kimenet: bináris, futtatható kód. Ha mi szeretnénk nevet adni a kimenetnek, akkor használjuk a -o opciót.

```
$ gcc program.c -o futtathato
```

Mind a két esetben a kimeneti fájl már rendelkezik a futtatható (x) attribútummal.

Az így kapott programot futtathatjuk, ha egyszerűen beírjuk a nevét (egyes Unix verziókban meg kell adnunk az útvonalat még akkor is, ha a fájl az aktuális könyvtárban található):

```
$ ./futtathato
```

Ha több állományból álló programot (projektet) írunk, ezeket a make parancssal kapcsoljuk össze.

### ☰ Környezet, szövegszerkesztő

Amint láttuk gcc-nek nincs saját kezelőfelülete, szövegszerkesztője. Ezért a programunkat bármilyen szerkesztőben begépelhetjük. A különböző Unix verziók rengeteg ilyen (szebbnél szebb) alkalmazást tartalmaznak.

Egy biztos: a vi szövegszerkesztő mindenik verzióban megtalálható.

Használati útmutató:

- ne ijedjünk meg, nem olyan vészes, mint amilyennek elsőre látszik
- elindítása: vi vagy vi program.c
- gépelés elkezdése: INSERT billentyűvel (ezzel egyébként válthatunk beszúrási/felülírási mód között)
- parancs üzemmódba váltás (minden parancs kiadása előtt kötelező): ESC
- kilépés: :q
- állomány mentése (ha van már neve): :w
- állomány mentése (ha még nincs neve): :w név
- mentés, kilépés: :wq
- kilépés mentés nélkül: :q!

Jól használható szövegszerkesztő még a joe, emacs stb.

## ☰ Segítség

Akárcsak a shell parancsok esetén a man program C utasításokról is nagyon sok információt tartalmaz.

Például:

```
$ man printf
```

## ☰ A parancssor paraméterei és a környezeti változók

Amikor egy folyamat elindít egy programot, átadja neki a parancssor paramétereit és a környezeti változókat. Ahhoz, hogy programunk ezeket a változókat használni tudja a főprogramot a következőképpen kell deklarálnunk:

```
main(int argc, char *argv[], char *envp[])
```

A parancssor paramétereit az argv, a környezeti változókat pedig az envp által mutatott táblázat (vektor) tartalmazza. Az argv elemeinek számát az argc változó tartalmazza.

A paraméterek és a környezeti változók kiírása történhet például így:

```
for (k=0; k<argc; ++k)
{
    printf("a %d. parameter: %s\n", k+1, argv[k]);
}

for (k=0; ; ++k)
{
    if (envp[k][0])
    {
        printf("a %d. környezeti változó: %s\n", k+1, envp[k]);
    }
    else break;
}
```

A környezeti változók néhány karaktersorból állnak. Formájuk:

**NÉV = érték**

A leggyakoribb ilyen változók:

- HOME – azt a könyvtárat jelöli, amelybe egy cd utasítás hatására ugrik
- PATH – azon könyvtárak listája ahol a shell a kiadott parancsokat keresi
- MAIL – a leveleket tartalmazó fájl neve
- TERM – a terminál típusa
- SHELL – a bináris shell állomány elérési útvonala
- LOGNAME – a név amelyen a felhasználót a rendszer nyilvántartja
- PS1 – a shell prompt jele (alapért. \$)
- PS2 – a prompt további sorainak jele (alapért. >)

A Unix shellből az env parancsot kiadva megjeleníthetők a környezeti változók értékei.

A környezeti változók értékének programból történő meghatározásában nem az `envp` az egyetlen lehetőség. Használhatjuk az `environ` külső változót is, amely éppen a rendszer által tárolt táblázatra mutat.

```
extern char **environ;

main()
{
    int k;
    for (k=0; ; ++k)
        if (environ[k][0])
        {
            printf("a %d. környezeti változó: %s\n", k+1, environ[k]);
        }
        else break;
}
```

Az `envp` és az `environ` változók ugyanazt az eredményt adják.

Egy bizonyos környezeti változó értékét lekérdezhetjük a `getenv` változóval:

```
#include <stdlib.h>
char *getenv(const char *nev);
```

A függvény egy pointert ad vissza, amely a `nev` környezeti változóhoz hozzárendelt értékre mutat. Értéke `NULL`, ha nem létezik ilyen nevű változó.

```
char *terminal;
if ((terminal = getenv("TERM")) == NULL)
{
    printf("a TERM környezeti változó nincs definiálva\n");
    exit(1);
}
printf("a terminal: %s\n", terminal);
```

A fenti példa lekérdezi a terminál értékét, amennyiben a változónak nincs értéke hibaüzenetet ad.

## ☰ Hibakezelés

Unixban egy függvényhívás következtében fellépő hiba esetén a legtöbb függvény -1 értéket térít vissza. Ilyen esetekben a hiba kódját az `errno` globális változó tartalmazza. Ez azonban nem egy elfogadott szabály, hiszen egyes függvények `NULL` értéket adnak vissza.

Az `errno.h` deklarációs fájlban van definiálva az `errno` változó, valamint azon szimbolikus konstansok értékei, amelyeket ez a változó felvehet. Az `errno` nem veheti fel a 0 értéket, és a szimbolikus konstansok között sincs 0 értékű.

```
extern int errno;
```

Amennyiben egy függvény nem hibával zárul, az előző `errno` érték megmarad!

A standard C két függvényt definiál a hibaüzenetek kiírására (szöveggel történő meghatározására). Az egyik az `strerror`, amelynek alakja:

```
#include <string.h>
char *strerror(int nrerr);
```

A függvény egy pointert térít vissza a hiba szövegéhez. Az nrerr rendszerint az errno változó szokott lenni, de megadhatunk tetszőleges hibakódot is.

A másik hibaértelmező függvény a perror; ennek szintaxisa:

```
#include <stdio.h>
void perror(const char *msg);
```

A függvény a standard hibacsatornára kiírja az msg szöveget, egy : (kettőspontot), majd az errno változó által meghatározott hibaüzenetet.

Az egyes hibakódokhoz tartozó [hibaüzenetek listája](#) megtalálható a Függelékben.

Az alábbiakban definiálunk néhány olyan függvényt, amelyek leegyszerűsítik a hibakezelést. A továbbiakban a példaprogramokban is ezeket a függvényeket fogjuk felhasználni. A függvények működését a következő táblázat foglalja össze:

| Függvények | strerror(errno) | Befejeződés |
|------------|-----------------|-------------|
| err_sys    | igen            | exit(1);    |
| err_ret    | igen            | return;     |
| err_quit   | nem             | exit(1);    |
| err_msg    | nem             | return;     |
| err_dump   | igen            | abort();    |

A függvények definíciója az [err.c](#) állományban található.

(A fenti függvények alapötlete és a mellékelt kód Iosif Ignat, Adrian Kacso *UNIX – Gestionarea proceselor* című könyvéből származik)

☰ **Memóriakezelés**

☰ **malloc**

Lefoglal egy megadott méretű (meret) memóriaterületet. A lefoglalt memóriazónát nem inicializálja semmilyen értékkel. Alakja:

```
#include <stdlib.h>
void *malloc(size_t meret);
```

A függvény 0-tól különböző értéket ad, ha a lefoglalás sikeres volt, és NULL-t, ha sikertelen (ekkor a hiba kódját az errno tartalmazza). Az első esetben a visszatérített érték éppen a lefoglalt memóriaterület kezdetére mutató pointer lesz.

Példa:

```
void *ptr;
if ((ptr = malloc(10000)) == NULL)
```



```
{
    perror("malloc");
    exit(1);
}
```

## === calloc

Lefoglal egy `elemszam*meret` nagyságú folytonos memóriaterületet – magyarul `elemszam` darab, egyenként `meret` nagyságú területet –, majd ezt NULL értékekkel tölti fel. Alakja:

```
#include <stdlib.h>
void *calloc(size_t elemszam, size_t meret);
```

Sikeres lefoglalás esetén 0-tól különböző értéket térít vissza (pointer a lefoglalt memóriazónára), különben a NULL értéket.

Példa:

```
struct
{
    int s1;
    long s2;
    char s3[10];
} s;

void *ptr;

if ((ptr = calloc(10, sizeof(s))) == NULL)
{
    perror("calloc");
    exit(1);
}
```

## === realloc

Átméretez egy `malloc`, `calloc` vagy `realloc` függvénnnyel lefoglalt memóriaterületet `uj_meret` nagyságúra. Alakja:

```
#include <stdlib.h>
void *realloc(void *ptr, size_t uj_meret);
```

Sikeres lefoglalás esetén egy nem nulla értéket ad vissza (amely lehet a régi mutató, de lehet egy új érték is), különben a NULL-t.

Példa:

```
void *ptr;

if ((ptr = malloc(10000)) == NULL)
{
    perror("malloc");
    exit(1);
}
```

```

if ((ptr = realloc(ptr, 20000)) == NULL)
{
    perror("realloc");
    exit(2);
}

```

## ≡ free

Felszabadít egy a malloc, a calloc vagy a realloc függvénnyel lefoglalt memóriaterületet. Alakja:

```

#include <stdlib.h>
void free(void *ptr);

```

Amennyiben a megadott pointer nem létezik a függvény furcsán viselkedhet!

Példa:

```

void *ptr;

if ((ptr = malloc(10000)) == NULL)
{
    perror("malloc");
    exit(1);
}

free(ptr);

```

## ≡ Aktuális könyvtár váltása (chdir)

Minden folyamathoz tartozik egy ún. aktuális könyvtár. Ez a könyvtár határozza meg az összes relatív keresési útvonalat, amelyet a folyamat használ. A folyamathoz tartozó aktuális könyvtárat megváltoztathatjuk a chdir függvény segítségével. Alakja:

```

#include <unistd.h>
int chdir(const char *path);

```

A függvény 0-t ad vissza, ha a váltás sikerült, és -1-et hiba esetén. A path változóban az új útvonalat kell megadni (ehhez legalább végrehajtási (x) jogunk kell legyen).

A rendszerbe való belépéskor minden felhasználó a saját gazdakönyvtárából indul. A gazdakönyvtár a felhasználót jellemzi, míg az aktuális könyvtár egy folyamat jellemzője. Vigyázat, a kettőt ne tévesszük össze!

Példa:

```

#include "<u>hdr.h</u>"

int main(void)
{
    if (chdir("/tmp")<0)
        err_sys("chdir hiba");
    printf("a konyvtarvaltas sikerult\n");
    exit(0);
}

```

## ☰ Gyökérkönyvtár megváltoztatása (chroot)

A gyökérkönyvtár megváltoztatására való a chroot függvény, melynek alakja:

```
#include <unistd.h>
int chroot(const char *path);
```

A függvény 0-t ad vissza, ha a váltás sikerült, és -1-et hiba esetén. A path változóban az új útvonalat kell megadni.

Ezt a függvény azonban csak a superuser hajthatja végre!

## ➡ Folyamatokkal végzett műveletek

---

Copyright (C) Buzogány László, 2002



[About](#)



# Folyamatokkal végzett műveletek

Folyamatok létrehozása, végrehajtása, befejezése...

---

## Bevezető

Folyamat létrehozása ([fork](#))

Folyamat befejezése ([exit](#))

Várakozás egy folyamatra ([wait](#), [waitpid](#))

Program meghívása, futtatás ([exec](#))

Folyamat attribútumainak lekérdezése ([getpid](#), [getppid](#), [getuid](#), [getgid](#), [geteuid](#), [getegid](#), [setuid](#), [setgid](#))

Folyamatcsoportok ([getpgrp](#), [setpgid](#))

Parancs végrehajtása ([system](#))

Folyamat prioritásának változtatása ([nice](#))

---

## ANSI C

### Bevezető

A Unixban folyamatkezelés alatt a következőket értjük:

- folyamat létrehozása,
- programok meghívása,
- folyamat várakoztatása addig, amíg a gyerekfolyamat befejeződik,
- folyamat befejezése.

Amint már említettük, minden folyamatnak egyedi azonosítója (PID-je) van: egy pozitív egész szám.

Egy Unix rendszerben léteznek speciális folyamatok, amelyek a rendszer indulásakor töltődnek be:

- A 0 azonosítójú folyamat mindig az ütemező, neve **swapper**. Ez a folyamat része a rendszer magjának és rendszerfolyamatnak számít.
- Az 1 azonosítójú folyamat az **init** (/etc/init, újabb verziókban /sbin/init), amelyet a rendszer maga hív meg a rendszer betöltése után. Ez a folyamat folyamatosan beolvassa a rendszertől függő inicializációs fájlokat és stabilizálja a rendszert. Az **init** (a **swapper**-rel ellentétben) felhasználói folyamat, de futtatáskor mégis superuser jogokkal rendelkezik. Az **init** folyamat nem fejeződik be soha!

- A Unix egyes, virtuális memóriát is használó implementációiban a 2 azonosítóval rendelkező folyamat a pagedaemon. Ez a rendszerfolyamat a lapozásért felel.

## Folyamat létrehozása (fork)

Egy folyamatot a fork rendszerfüggvénnyel hozhatunk létre. Azt a folyamatot, amely a fork-ot hívta szülőnek, az újonnan létrehozott folyamatot pedig gyereknek (fiúnak) nevezzük.

A fork függvény szintaxisa:

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

A visszaadott pid\_t típusú érték:

0 – a gyerekben,  
a gyerek PID-je – a szülőben,  
-1 – hiba esetén.

Hiba esetén az errno változó a hiba kódját fogja tartalmazni.

Egy folyamat több gyereket is létrehozhat. Mivel nincs egyetlen olyan függvény sem, amellyel meg lehetne egy gyerek PID-jét változtatni, a fork függvény a szülőnek visszaadja a (létrehozott) gyerek folyamatazonosítóját. A gyerekfolyamatban ez az érték 0, hiszen egy gyereknek csak egy szülője lehet.

Egy gyerek a szülője azonosítóját a getppid függvénnyel kérdezheti le. A saját PID lekérdezésére a getpid függvény szolgál.

```
#include <sys/types.h>
#include <unistd.h>

pid_t getppid(void);
pid_t getpid(void);
```

**Figyelem!** A gyerekfolyamat és a szülőfolyamat kódját ugyanabban az állományban kell megírni! A fork hívás után a szülő- és a gyerekfolyamat két különböző folyamatként, egyidőben fog futni! A végrehajtást mindkettő a fork utáni első utasítással fogja folytatni.

**Tehát:** a két folyamat (szülő és gyerek) ugyanazon kódszegmenst használják, az adat- és veremszegmensük viszont különböző!

Példa:

```
#include <stdio.h>
#include <sys/types.h>

int main(void)
{
    pid_t pid;
    pid = fork();
    printf("%d folyamat, pid = %d\n", getpid(), pid);
}
```

A fenti program végrehajtásának eredménye:

```
$ a.out
226 folyamat, pid = 207
207 folyamat, pid = 0
```

Mivel a fork hívás után két azonos kódszegmensű folyamat jött létre, a printf függvényt mind a szülő, mind pedig a gyerek végrehajtja. A szülő egyrészt kiírja a saját azonosítóját (226), másrészt a gyerek PID-jét (207). A gyerek szintén előbb a saját PID-jét (207), majd a 0 értéket jeleníti meg (hiszen a pid változó értéke a gyerekekben 0).

Azért, hogy a két kódrészt (gyerek-szülő) jól elkülönítsük, a fork-ot a következőképpen szoktuk használni:

```
pid = fork();
if (pid == 0)
{
    /* gyerek folyamat */
}
else
{
    /* szülő folyamat */
}
```

Az alábbi példában a hibát is kezeljük:

```
switch (fork())
{
    case -1:
        perror("fork");
        exit(1);
    case 0:
        /* gyerek folyamat */
    default:
        /* szülő folyamat */
}
```

Észrevehetjük, hogy a szülő- és a gyerekfolyamatok különböznek, külön azonosítóval rendelkeznek, mégis nagyon sok közös attribútumuk van. A fork előtt deklarált változókat mind a szülő, mind a gyerek láthatja, de értékük a későbbi módosítások során csak az aktuális folyamatban változik.

```
#include <sys/types.h>

int gvar = 4;

int main(void)
{
    pid_t pid;
    int var = 7;
    printf("a fork előtt\n");
    if ((pid = fork()) == -1)
        err_sys("fork hiba");
```

```

else
    if (pid == 0)
    {
        gvar++;
        var += 2;
    }
    else
        sleep(2);
printf("folyamat(pid)=%d, gvar=%d, var=%d\n", getpid(), gvar, var);
exit(0);
}

```

A fent definiált két folyamat közül az egyiket a rendszer hamarabb fogja végrehajtani de, hogy melyiket, azt nem tudhatjuk előre (még a sleep(2) utasítás sem garantálhatja, hogy a gyerek folyamat fog hamarabb lefutni). Ezért futtatáskor akár két különböző eredményt is kaphatunk:

```

$ a.out
a fork előtt
folyamat(pid)=184, gvar=5, var=9
folyamat(pid)=183, gvar=4, var=7

$ a.out
a fork előtt
folyamat(pid)=187, gvar=4, var=7
folyamat(pid)=188, gvar=5, var=9

```

Látható, hogy a két futtatás eredményeként különböző folyamatazonosítókat osztott ki a rendszer.

## Folyamat befejezése (exit)

Egy folyamat szabályszerű, önmaga által történő, azonnali leállítása az `exit`, illetve az `_exit` függvényekkel történik. (Az előbbi ANSI C utasítás, míg a második csak a POSIX-ben található meg, és tartalmazza a Unix sajátosságokat.)

```

#include <stdlib.h>
void exit(int exit_code);

#include <unistd.h>
void _exit(int exit_code);

```

Az `exit_code` a kilépési kód, amelyet a hívó program használ ellenőrzésre. Egy folyamat befejezésekor az összes gyerekfolyamat átöröklődik az `init` (1) folyamathoz (tehát új szülőt kap). A rendszer ezáltal biztosítja, hogy minden folyamatnak legyen szülője.

Ha egy program nem tartalmazza az `exit` utasítást, a rendszer automatikusan végrehajtja az `exit`-et a `main` függvény befejezése után.

Rendellenes működés esetén a folyamat befejezésére használjuk az `abort` függvényt. Szintaxisa:

```

#include <stdlib.h>
void abort(void);

```

Amennyiben egy gyerekfolyamat hamarabb befejeződik, mint a szülő a rendszer bizonyos információkat

még megőrzi vele kapcsolatban (PID, befejeződési állapot, elhasznált processzoridő). Ezen információk a `wait` és a `waitpid` függvények segítségével érhetők el. Ezek a függvények felfüggesztik a folyamat működését, ameddig a gyerekfolyamat be nem fejeződik.

A Unix felfogása szerint, azt a folyamatot, amely befejeződött, de a szülő nem adott ki `wait` parancsot, **zombie** folyamatnak nevezzük. Ebben az állapotban a folyamatnak nincs semmilyen lefoglalt memóriaterülete, csak egy bemenete a folyamatáblában. A rendszer felhasználhatja a lefoglalt memóriazónákat, illetve bezárhatja az általa megnyitott fájlokat. A zombie folyamatokat a `ps` parancs segítségével követhetjük nyomon.

Példa:

```
#include "hdr.h"

int main(void)
{
    pid_t pid;
    if ((pid = fork()) == -1)
        err_sys("fork hiba");
    else
        if (pid == 0)
            exit(0);
    sleep(3);
    system("ps");
    exit(0);
}
```

A fenti program által kiírt eredmény:

```
PID TTY STAT TIME COMMAND
54 v01 S 0:00 -bash
90 v01 S 0:00 a.out
91 v01 Z 0:00 (a.out) <zombie>
92 v01 R 0:00 ps
```

Egy folyamat, amelynek szülője az `init`, sosem kerülhet zombie állapotba, mivel az `init` mindig meghív egy `wait` függvényt, ellenőrizve ezzel a befejezett folyamat állapotát.

### Várakozás egy folyamatra (`wait`, `waitpid`)

Ha egy folyamat befejeződik, a rendszer egy `SIGCLD` üzenettel értesíti a szülőfolyamatot. Ezek után a szülő figyelmen kívül hagyja az illető folyamatot.

Előfordulhat, hogy a szülőnek meg kell várnia a gyerekfolyamat lefutását, s csak azután tud valamilyen feladatot megoldani. Erre (azaz például várakozásra) használjuk a `wait` és a `waitpid` függvényeket. Egy folyamat, amely meghívja a `wait` vagy a `waitpid` függvényeket:

- várakozhat (ha minden gyereke fut),
- érzékelheti, hogyha egy gyerek befejeződött,
- visszatéríthet hibát (ha nincs gyereke).

Szintaxisuk:



```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int opt);
```

A `status` egy mutató egy olyan táblázatra, amely egy gyerekfolyamat befejeződési állapotát tartalmazza. Ez a 16 bites információ a következőket tartalmazza:

- ha a gyerekfolyamat `exit`-tel ért véget, az állapot (`status`) változó a következőket tartalmazza:



ahol az `exit_code` az `exit` függvény által megadott érték.

- ha a gyerekfolyamat egy jelzés hatására ért véget, a `status` változó:



ahol az `x` értéke 1, ha a jel üres memóriaterületet eredményezett, és 0 különben. A `jel_id` annak a jelnek az azonosítója, amely a gyerekfolyamat befejezését eredményezte.

- ha a gyerekfolyamatot leállították, a `status` változó:



ahol a `jel_id` annak a jelnek az azonosítója, amely a folyamatot leállította

Különbségek a `wait` és a `waitpid` között:

- a `wait` felfüggeszti a hívó folyamatot, amíg a gyerek befejeződik, ezzel szemben a `waitpid` egy külön opciót kínál fel (`opt`), melynek használatával a felfüggesztés elkerülhető,
- a `waitpid` nem mindig az első fiú befejezéséig vár, hanem a `pid` változóban megadott azonosítójú gyerek befejezéséig,
- a `waitpid` az `opt` argumentum segítségével engedélyezi a programok vezérlését.

A `wait` függvény visszatérési értéke azon gyerekfolyamat azonosítója, amely éppen befejeződött. Mivel tehát a függvény a folyamat PID-jét téríti vissza, mindig pontosan tudjuk, mely gyerek fejeződött be éppen. Egy bizonyos gyerek befejeztét megvárhatjuk például így:

```
while (wait(allapot) != pid);
```

ahol a `pid` azon gyerekfolyamat azonosítója, amelyre várakozunk.

A `waitpid` függvényhívásnál megadható `pid` változó lehet:

- `pid = -1` – bármely gyerekre várakozhat; ekvivalens a `wait`-tel,
- `pid > 0` – a `pid` azonosítójú folyamatra várakozik,
- `pid = 0` – bármely olyan folyamatra várakozik, amelynek a csoportazonosítója megegyezik a hívó programéval,
- `pid < -1` – bármely olyan folyamatra várakozik, amelynek a csoportazonosítója megegyezik a hívó programéval abszolút értékben.

A `waitpid -1` értéket térít vissza, ha nem létezik a `pid`-ben megadott azonosítójú folyamat, vagy nem gyereke a hívó folyamatnak.

## Program meghívása, futtatás (exec)

Az `exec` parancs az aktív folyamat kódját egy másikkal helyettesíti (azaz átadja a vezérlést egy másik programnak). A program ezáltal egy teljesen új kódszegmenst, és egy ennek megfelelő adatszegmenst kap. Az új programnak egy futtatható állománynak kell lennie.

Az `exec` utasítás különböző végződésekkel rendelkezhet. A különbségeket a következő táblázat szemlélteti:

| Függvények          | Paraméter | Keresési<br>útvonal | Környezet |
|---------------------|-----------|---------------------|-----------|
| <code>execl</code>  | lista     | <code>./</code>     | marad     |
| <code>execv</code>  | tömb      | <code>./</code>     | marad     |
| <code>execlp</code> | lista     | <code>PATH</code>   | marad     |
| <code>execvp</code> | tömb      | <code>PATH</code>   | marad     |
| <code>execle</code> | lista     | <code>./</code>     | változik  |
| <code>execve</code> | tömb      | <code>./</code>     | változik  |

A fenti végzések jelentése a következő:

l – a paraméterátadás listán keresztül történik,  
v – a paraméterátadás tömbön keresztül történik,  
e – változik a környezeti változó,  
p – figyelembe veszi a `PATH` változót.

A függvények szintaxisa:

```
#include <unistd.h>

int execl(const char *path,          /* elérési út */
          const char *arg0,          /* programnév */
          const char *arg1,          /* paraméterek */
          ...
          const char *argn,
          NULL);                    /* a paraméterek befejeztét jelző NULL */

int execv(const char *path, char *argv[]);

int execlp(const char *filename,      /* a futtatható állomány neve */
          const char *arg0,
          const char *arg1
          ...
          const char *argn,
          NULL);
```

```
int execvp(const char *filename, char *argv[]);

int execl(const char *path,
          const char *arg0,
          const char *arg1,
          ...
          const char *argn,
          NULL,
          char *envp[]);      /* környezeti változók */

int execve(const char *path, char *argv[], char *envp[]);
```

Az egyes változók jelentései:

- **path**: mutató egy karaktersorhoz, amely a futtatható állomány keresési útvonalát jelöli,
- **arg0**: mutató a futtatható állomány nevéhez,
- **arg1, arg2, ..., argn**: mutatók, amelyek a programnak átadott paramétereket jelölik,
- **argv**: mutató a paramétervektorhoz (a 0-dik paraméter az állomány neve),
- **filename**: mutató a futtatható állomány nevéhez; ha a név nem kezdődik a gyökérrel (és nincs megadva a teljes útvonal), akkor az állományt a PATH változó által definiált könyvtárakban keresi a rendszer,
- **envp**: mutató az új környezeti változókhoz, amelyek a vektorban egyenként **változó=érték** alakban jelennek meg.

Amikor a paramétereket listaként adjuk meg, az utolsó paramétert követően beírunk egy NULL paramétert is. Ez jelzi a felsorolás végét.

Példák:

```
execl("/bin/time", "time", "ps", NULL);

...
char *argv[3];
argv[0] = "time";
argv[1] = "ps";
argv[2] = (char *)0;
execv("/bin/time", argv);

execlp("prog", "prog", "p1", "p2", NULL);

...
char *argv[4];
argv[0] = "prog";
argv[1] = "p1";
argv[2] = "p2";
argv[3] = (char *)0;
execvp("prog", argv);

...
char *envp[2];
envp[0] = "term=vt100";
```

```
envp[1] = (char *)0;
execl("prog", "prog", "p1", "p2", 0, envp);
```

**Feladat:** Írjunk programot, amely billentyűzetről bekér Unix parancsokat és végrehajtja őket, pontosan úgy, ahogy a shell!

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

int main(void)
{
    int MAXLINE = 1024;
    char buf[MAXLINE];
    pid_t pid;
    int status;

    printf("> ");
    while (fgets(buf, MAXLINE, stdin) != NULL)
    {
        buf[strlen(buf)-1] = 0;
        if ((pid = fork()) < 0)
        {
            printf("fork hiba\n");
            exit(1);
        }
        else
            if (pid == 0)
            {
                execlp(buf, buf, NULL);
                printf("nem lehet vegrehajtani: %s\n", buf);
                exit(127);
            }
        if ((pid = waitpid(pid, &status, 0)) < 0)
        {
            printf("waitpid hiba\n");
            exit(1);
        }
        printf("> ");
    }
    printf("befejeztem\n");
    exit(0);
}
```

A végrehajtandó parancsra várva a program megjeleníti a promptot (>). A programból a Ctrl-\ megszakitási jelzéssel tudunk kilépni.

A program futtatásának eredménye lehet például a következő:

```
$ a.out
```

```
> date
Wed Dec 25 09:56:37 EST 2002
> pwd
/home/lacka
> who
lacka    vc/1    Dec 25 07:16
> Ctrl-\
$
```

## Folyamat attribútumainak lekérdezése

A Unix minden folyamathoz legkevesebb 6 azonosítót rendel. Ezeket céljuk szerint a következőképpen lehet csoportosítani:

- a. amelyek a valódi felhasználót azonosítják (azt a személyt aki a felhasználónevével belépett):
  - a valódi felhasználó ID-je,
  - a valódi csoport ID-je,
- b. amelyek meghatározzák a hozzáférési jogát egy állományhoz:
  - effektív felhasználó ID-je,
  - effektív csoport ID-je,
  - a csoport kiegészítő azonosítói (egyes verziókban),
- c. amelyeket elmentett az exec függvény:
  - elmentett felhasználó ID,
  - elmentett csoport ID.

Amint már láttuk egy folyamat azonosítójának lekérdezésére a getpid függvény szolgál, míg a szülő azonosítóját a getppid függvény szolgáltatja.

Egy folyamat valódi felhasználójának azonosítóját a getuid, a valódi csoport azonosítóját pedig a getgid függvénnyel lehet lekérdezni. Az azonosítók effektív változatait a geteuid és a getegid függvények adják. Ezek szintaxisa:

```
#include <sys/types.h>
#include <unistd.h>

uid_t  getuid(void);
gid_t  getgid(void);
uid_t  geteuid(void);
gid_t  getegid(void);
```

A valódi felhasználó és csoport azonosítójának módosítására is van lehetőségünk a setuid és a setgid függvények által. Itt meg kell adnunk az új azonosítókat.

```
#include <sys/types.h>
#include <unistd.h>

int  setuid(uid_t uid);
int  setgid(gid_t gid);
```

Mind a két függvény 0-t térít vissza, ha a módosítás sikerült, és -1-et hiba esetén.

## Folyamatcsoportok

Unixban minden folyamat egy csoport része. Egy folyamatcsoport több folyamatból álló gyűjtemény, amely egyedi azonosítóval rendelkezik (ennek típusa megegyezik a folyamatazonosítók típusával).

A `getpgrp` függvény visszatéríti annak a csoportnak az azonosítóját, amelyhez a hívást végző folyamat tartozik. Ennek szintaxisa a következő:

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpgrp(void);
```

Minden folyamatcsoportnak van egy ún. **csoportvezetője**. Ezt könnyen meghatározhatjuk, mivel ennek a folyamatnak az azonosítója éppen megegyezik a csoport ID-jével.

Egy folyamat csatlakozhat egy már létező csoporthoz, vagy létrehozhat egy új csoportot a `setpgid` függvény használatával. Ennek alakja:

```
#include <sys/types.h>
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

A függvény 0-t ad vissza, ha a csoport létrehozása sikerült, és -1-et, ha nem. Továbbá a `pid` azonosítójú folyamatot hozzárendeli az újonnan létrehozott `pgid` azonosítójú csoporthoz.

Egy folyamat a csoportja számára adhatja a saját ID-ját, vagy valamelyik gyereke azonosítóját. Ellenben nem változtathat meg egy csoport ID-t valamely gyereke számára, ha a gyerek már meghívott egy `exec` függvényt.

Ha a `pid` értéke 0, a csoport a hívó program ID-ját fogja használni, ha viszont a `pgid` 0, akkor a `pid` által meghatározott azonosítót.

## Parancs végrehajtása (system)

A `system` függvény segítségével egy C programban is végrehajthatunk bármilyen Unix parancsot.

```
#include <stdlib.h>

int system(const char *cmdstr);
```

A parancsot a `cmdstr` karaktersorban kell megadnunk, a visszaadott érték pedig -1, ha a végrehajtás nem sikerült, illetve a visszatérési állapot minden más esetben.

**Figyelem!** A `system` igen hasonlít az `exec`-hez, de ő előbb egy `fork` hívással létrehoz egy új gyerekfolyamatot, amelyben az `exec` segítségével végrehajtja az adott parancsot. Éppen ezért a `system` nem adja át a vezérlést, hanem a gyerek befejeztével a soron következő utasítással folytatja az aktuális folyamat végrehajtását.

Példa:

```
...
system( "ps" );
...
```

## Folyamat prioritásának változtatása (nice)

Egy folyamat prioritásának megváltoztatására szolgál a nice függvény. Alakja:

```
#include <unistd.h>
int nice(int inc);
```

A függvény hozzáadja az inc értéket a hívó folyamat nice értékéhez. (Egy folyamatnak minél nagyobb a nice értéke, annál kisebb a prioritása.) Csak a superuser adhat meg negatív értékeket, s növelheti ezáltal egy folyamat futási sebességét.

A visszatérített érték 0, ha a művelet sikeres volt, és -1 különben.

Egy folyamat prioritását a rendszer a következő képlettel határozza meg:

$$\text{folyamat\_prioritás} = \text{alap\_prioritás} + \text{elhasznált\_processzoridő} / \text{konstans} + \text{nice\_érték}$$

A nice érték minden esetben egy -20 és 19 közötti egész szám.

## Jelzések (signals)

---

Copyright (C) Buzogány László, 2002



[About](#)



# Jelzések (signals)

Azaz a folyamatok informálása előre definiált jelek segítségével...

---

## Definíció

## Jelzések kezelése

signal

kill, raise

alarm, pause

setjmp, longjmp

sigaction

---

## Folyamatokkal végzett műveletek

## Definíció

A jelzés olyan szoftveres megszakítás, amelyet egy folyamat kap valamilyen esemény bekövetkeztekor. A jelzéseket főként kivételek fellépésekor, valamilyen riasztáskor, váratlan befejezéskor, vagy ritkábban folyamatok közti kommunikációkor használják. Ezek jelentik a legegyszerűbb, legrégebbi, de legmerevebb kommunikációt a folyamatok között. A jelzéseket felfoghatjuk akár figyelmeztetésként is egy folyamat felé.

Minden jelzésnek van egy neve, amely kötelezően a SIG előtaggal kezdődik. Ezek a nevek szimbolikus konstansok formájában vannak definiálva, és szigorúan pozitív egész számokat jelölnek.

Egy jelzés által hordozott információ a folyamat számára minimális, hiszen csupán a típust (magát a számot) foglalja magába. Minden jelzésnek van egy forrása és valamilyen okból keletkezik.

A jelzések okait a következőképpen csoportosíthatjuk:

a. futtatás során fellépő hibák:

- a folyamat megengedett határain kívül eső címzés (ekkor egy SIGSEGV jel keletkezik),
- írási próbálkozás egy csak olvasható memóriazónára (például a kódszegmensbe),
- magasabb privilégium szinttel rendelkező utasítások végrehajtása,
- hardverhiba detektálása esetén,

b. szoftverhiba egy rendszerfüggvény hívásakor:

- nem létező rendszerfüggvény,



- egy pipe állományba való írás, anélkül, hogy egy másik folyamat olvasná azt (SIGPIPE),
- nem megfelelő paraméter használata egy függvényhívásban,
- valamilyen szükséges erőforrás hiánya egy függvény végrehajtása során (például nincs elegendő külső memória egy állomány betöltésére),

c. kommunikáció két folyamat vagy folyamatcsoportok között úgy, hogy a folyamat egy jelet kap a kill függvényen keresztül,

d. háttérben futó folyamat befejezése a kill parancs használatával,

e. egy folyamat erőteljes befejezése a rendszer által egy SIGKILL jel segítségével (például egy shutdown parancs esetén),

f. egy folyamat az idő függvényében egy SIGALRM jelet küld magának,

g. a felhasználó a billentyűzetről megszakít egy terminálon futó folyamatot,

- egy folyamat feladása (Ctrl-\\),
- megszakítás generálása (Ctrl-C/Break vagy DELETE),
- a terminálhoz való újrakapcsolódás,

h. egy folyamat befejezése, amint éppen az exit függvényt hajtja végre, vagy amint a folyamat meghívja a SIGCHLD jelet a signal rendszerfüggvény segítségével,

i. egy folyamat megjelölése.

A jelzés típusa értelemszerűen tükrözi a jelzés okát is. A Függelékben bemutatjuk a UNIX SVR2 által definiált 19 jelzéstípust. A Unix SVR4 és 4.3+BSD verziókban már 31 típusú jelzés szerepel. Ezek részletes bemutatását lásd a bibliográfiában.

Egy folyamat, amely valamilyen jelet kapott a következőképpen viselkedhet:

- Figyelman kívül hagyja a jelet és folytatja az aktivitását. A jelzések közül csak a SIGKILL nem halasztható el (a rendszernek jogában áll befejezni bármelyik folyamatot).
- A jel kezelését a rendszer magjára bízta. Ebben az esetben, kivétel a SIGCLD és SIGPWR (és újabban a SIGINFO, SIGURG és SIGWINCH), amelyeket figyelmen kívül hagy, az összes többi jelzés a folyamat befejezéséhez megy – ez az implicit működés.
- Egy saját eljárással automatikusan lekezeli a jelet, ahogy az megjelenik. Az eljárás befejeztével a program ott folytatódik, ahol abbahagyta.

A fork rendszerfüggvény hívására a létrejövő gyerekfolyamat a szülőtől öröklí a jelzésekhez kapcsolódó eseményeket is.

Az exec függvény a rendszer magjára hagyja a jelzések kezelését, még akkor is, ha ezeket végül a folyamat hajtja végre. Csak az előrelátott, eredeti tevékenységeket tartja meg, figyelmen kívül hagyva egyes jelzéseket a folyamat felé. Ez azért van, mert az exec hívására az aktuális folyamat kódszegmense elvesztődik, s ezáltal a jelzések kezelését végző eljárások is megsemmisülnek.

Függetlenül attól, hogy miként reagál egy program egy bizonyos jelre, a tevékenység befejeztével – ha a jelet nem hagyjuk figyelmen kívül – a rendszer újrainicializálja az értékeket, felkészülve ezáltal egy későbbi jel fogadására. Kivételt képeznek ez alól a SIGILL és a SIGTRAP jelek, amelyeket nagyon gyakran előfordulnak, ezért ezek többszöri aktualizálása igencsak lassítaná a kezelésüket.

Ha egy függvény végrehajtása során a rendszer egy jelet érzékel, a hívás hibával fog befejeződni. Sajátos esetben, ha a rendszer egy lassú periférián próbál ki/bemeneti műveletet végezni és egy jelzést kap, a függvény -1 értéket (azaz hibát) fog visszatéríteni. Ebben az esetben az `errno` változó az `EINTR` értéket fogja tartalmazni. Ilyenkor az `errno` változó tesztelése után, a program újra próbálkozhat a művelet elvégzésével.

A folyamathoz érkezett jelzések a rendszer nem tárolja egy várakozási sorban. Éppen ezért, ha a folyamat egy jelet figyelmen kívül hagyott, az mindörökké elveszett. Egyetlen kivétel a `SIGCLD` jel, amelyet a gyerek küld a szülőnek, tudatva, hogy pályafutását befejezte. Ezt a jelet a szülő egy `wait` függvényhívás során érzékelheti. Ez azért fontos, mert egy gyerek már azelőtt befejeződhet, hogy a szülő kiadná a `wait` parancsot. Ha viszont a jelet nem őriznénk meg, a szülő leblokálna miközben hiába várná a gyerek befejeződését. Mivel a bejövő jelzéseket a folyamatok nem tudják megőrizni ez a mechanizmus nem túl hatékony, és sok hibával járhat.

**Összefoglalás:** Egy jelet elküldhetünk bármely pillanatban, időszakosan egy másik folyamattól, de általában a kerneltől indulnak valamilyen különleges esemény hatására. A jelek nem tartalmaznak információt csak amit a típusuk által hordoznak. A jelzések másik hátránya, hogy a folyamatok nem tudják azonosítani a beérkezett jelek forrását.

## ☰ Jelzések kezelése

### ☰ signal

A `signal` függvény szerepe biztosítani a jelzések programból történő kezelését.

Alakja:

```
#include <signal.h>
```

```
int (*signal (jelzes, fuggveny))();
int jelzes;
int (*fuggveny)();
```

vagy

```
#include <signal.h>
```

```
void (*signal (int jelzes, void (*fuggveny)(int)))(int);
```

Az első változat a régebben használatos stílus (amikor még nem volt értelmezve a `void` típus). A második változatban az argumentumok típusa is fel van tüntetve: egy egész szám és egy mutató egy függvényhez.

A `jelzes` paraméter a jel száma vagy az ennek megfelelő szimbólum, amelynek a kezelését szeretnénk testreszabni.

A `fuggveny` argumentum leírja, hogyan kezelje a folyamat az adott jelet. A következő értékeket veheti fel:

- `SIG_DFL` – A jelzések kezelését a rendszer magja végzi. Ez a jelzések implicit kezelése.
- `SIG_IGN` – Figyelmen kívül hagyja az adott jelet (kivétel a `SIGKILL` és a `SIGSTOP`).
- mutató egy függvényhez, amely lekezeli a jelet – Ebben az esetben egy jel érkezésekor az adott függvény meghívódik. Ez a függvény a kezelő rutin nevét viseli. Egyetlen argumentuma van, ami

éppen a kezelni kívánt jel számát jelenti. Miután egy jelzést ily módon lekezelünk, ugyanannak a jelnek a következő érzékelése már implicit módon történik. Tehát, ha ugyanezt a függvényt még egyszer szeretnénk használni megfelelő intézkedéseket kell tennünk.

A függvény visszatérési értéke az előzőleg definiált függvény erre a jelre. Hiba esetén a SIG\_ERR konstanst kapjuk, amelynek definíciója:

```
#define SIG_ERR (void (*)( ))-1;
```

A signal függvény bemutatására tekintsük a következő példát:

```
#include <signal.h>
#include "hdr.h"

static void sig_usr1(int);      /* generat cu kill -USR1 <pid> */
static void sig_intr(int);     /* generat la Ctrl-C si rearmat */
static void sig_quit(int);     /* generat cu Ctrl-\ si resetat */
static void sig_alarm(int);     /* generat dupa scurgerea timpului t din alarm(t) */

int main(void)
{
    if (signal(SIGALRM, sig_alarm) == SIG_ERR)
        err_sys("hiba: signal(SIGALRM, ...)");
    if (signal(SIGUSR1, sig_usr1) == SIG_ERR)
        err_sys("hiba: signal(SIGUSR1, ...)");
    if (signal(SIGINT, sig_intr) == SIG_ERR)
        err_sys("hiba: signal(SIGINT, ...)");
    if (signal(SIGQUIT, sig_quit) == SIG_ERR)
        err_sys("hiba: signal(SIGQUIT, ...)");
    for ever pause();
}

static void sig_alarm(int sig)
{
    printf("SIGALRM jelet vettem...\n");
    return;
}

static void sig_quit(int sig)
{
    printf("SIGQUIT jelet vettem...\n");
    if (signal(SIGQUIT, SIG_DFL) == SIG_ERR)
        err_sys("nem lehet visszaallitani ezt a jelet...");
    return;
}

static void sig_intr(int sig)
{
    printf("SIGINT jelet vettem...\n");
    if (signal(SIGINT, sig_intr) == SIG_ERR)
        err_sys("nem lehet ujra betolteni...");
    return;
}
```

```

}

static void sig_usr1(int sig)
{
    printf("SIGUSR1 jelet vettem...\n");
    alarm(1);
    printf("a riasztas 1 masodperc mulva elindul!\n");
    return;
}

```

Az `alarm` függvény a hívásától számított 1 másodperc leteltével egy `SIGALRM` jelet generál.

Ha a programot háttérben futtatjuk, a következőképpen viselkedik:

```

$ a.out&
[1] 324                                a shell kiírja a folyamat azonosítóját

$ kill -USR1 324                       a folyamatnak egy SIGUSR1 jelet küld
SIGUSR1 jelet vettem...               generálja a jelet 1 másodperc múlva
a riasztas 1 masodperc mulva elindul!
SIGALRM jelet vettem...               a SIGALRM jelet generálta

$ kill 324                             egy SIGTERM jelet küld

```

Ha azonban a programot nem a háttérben futtatjuk:

```

$ a.out
                                leütünk egy Ctrl-C billentyűt

SIGINT jelet vettem...
SIGINT jelet vettem...
                                leütünk egy Ctrl-\ billentyűt

SIGQUIT jelet vettem...
                                még egyszer leütünk egy Ctrl-\ billentyűt

$

```

A `SIGINT`-et kezelő rutin újra és újra betölti saját magát minden alkalommal amikor végrehajtódik. Erre azért van szükség, mert alapértelmezésben ez az eljárás csak az első jel érkezéséig él. Ezzel szemben a `SIGQUIT`-hoz tartozó eljárást nem töltjük be csak egyszer, ezért másodszor már az eredeti rutin szerint jár el, és bezárja a programot.

## kill, raise

A `kill` függvény egy jelet küld egy folyamatnak vagy egy folyamatcsoportnak. A `raise` megengedi a folyamatnak, hogy saját magának is küldjön jelet. Az első függvény a POSIX.1 verzióban van definiálva, míg a második az ANSI C-ben.

Szintaxisuk:

```

#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int jel);
int raise(int jel);

```

Mindkét folyamat 0 értéket térít vissza, ha a művelet sikeres volt, és -1-et hiba esetén. A pid változó azt a folyamatot, vagy folyamatcsoportot jelöli, aminek a jel jelet szeretnénk küldeni.

A pid változó értékei a következők lehetnek:

- pid > 0 – a jelet a pid azonosítójú folyamatnak küldi,
- pid = 0 – a jelet minden olyan folyamatnak elküldi az aktuális csoporton belül, amelyekhez van joga (kivétel a swapper (0), az init (1) és a pagedaemon (2) folyamatok); ez a dolog nagyon gyakori, amikor a kill 0 utasítással töröljük a háttérben futó folyamatokat, anélkül, hogy az azonosítóikat megadnánk,
- pid < 0,
- pid ≠ -1 – a jelet minden olyan folyamatnak elküldi, amelyeknek az ID-je megegyezik a pid változó abszolút értékével (és természetesen van ehhez joga az adott folyamatnak),
- pid = -1 – a POSIX.1 ezt a lehetőséget nem specifikálta; az SVR4 és 4.3+BSD ezt az értéket a broadcast jeleknél használja; ezek nem küldődnek el a fent már említett folyamatokhoz; ha a küldő a superuser, akkor a jelet minden folyamat megkapja; ezt a SIGTERM jel küldésekor szokták használni, azért, hogy a rendszert felfüggeszték.

A folyamatok közti jelküldésnél a szabály az, hogy a küldőnek legyen joga a jelet elküldeni (a valódi vagy effektív felhasználó ID-ja legyen egyenlő a valódi vagy effektív vevő uid-jével). A superuser bármelyik folyamathoz küldhet jeleket.

Kivételt képez a fenti szabály alól a SIGCONT jel, amelyet bármely folyamatnak el lehet küldeni, amely ugyanahhoz a géphez tartozik.

A POSIX.1 szabvány definiálja a 0 jelet is, amely a nul jelzésnek felel meg. A kill függvény 0 paraméterrel nem küldi el a megadott jelet, hanem csak ellenőrzi, hogy létezik-e a pid azonosítójú folyamat. Ha a folyamat nem létezik, a függvény -1-et térít vissza és az errno értéke ESRCH lesz.

A raise függvény implementációja a kill függvény segítségével:

```
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>

int raise(int jel)
{
    return (kill(getpid(), jel));
}
```

## ☰ alarm, pause

Az alarm függvény lehetővé teszi egy időzítő beállítását. A megadott idő (mp másodperc) elteltével egy SIGALRM jelet ad ki. Ha a folyamat a jelet figyelmen kívül hagyja, vagy nem érzékeli, a függvény alapbeállítás szerint befejezi a folyamatot. Alakja:

```
#include <unistd.h>
unsigned int alarm(unsigned int mp);
```

A visszatérített érték vagy 0, vagy az előző SIGALRM jelzés kiadása óta eltelt másodpercek száma.

Minden folyamathoz egyetlen időmérő (óra) tartozik, ezért a függvénynek egy újabb meghívása esetén az előző (mp) érték felülíródik. Ha az mp értéke 0, az előzőleg kiadott SIGALRM kérések törölődnek.

Mivel a SIGALRM implicit a folyamat befejezéséhez vezet, több folyamat, amely az alarm függvényt használja érzékeli ezt a jelet. Mielőtt a folyamat befejeződne lehetőségünk nyílik különféle utómunkák (például törlések) végrehajtására.

A `pause` függvény felfüggeszti (várakozási állapotba helyezi) a hívó folyamatot a legelső jel érkezéséig.

```
#include <unistd.h>
int pause(void);
```

Ha a bejövő jelet a folyamat nem kezeli le vagy figyelmen kívül hagyja, akkor ez a művelet a folyamat befejezéséhez vezet. A program `pause` függvényből csak a kapott jel lekezelése után jön ki. Ezért a függvény a minden esetben a -1 értéket téríti vissza, míg az `errno` változó értéke `EINTR` lesz.

A `pause` függvényt legtöbbször az alarm-mal együttesen szoktuk használni.

Az alarm függvényt nagyon gyakran alkalmazzuk olyan esetekben, amikor egy bizonyos műveletet valamilyen időintervallumon belül szeretnénk elvégezni. Amennyiben a műveletet a megadott idő alatt nem sikerült elvégezni a végrehajtást felfüggesztjük.

Az alábbi példa a standard bemenetről olvas és a standard kimenetre ír:

```
#include <setjmp.h>
#include <signal.h>
#include "hdr.h"

static void sig_alarm();

int main(void)
{
    int n;
    char line[MAXLINE];
    if (signal(SIGALRM, sig_alarm) == SIG_ERR)
        err_sys("hiba: signal(SIGALRM, ...)");
    alarm(10);
    if ((n = read(0, line, MAXLINE)) < 0)
        err_sys("read hiba");
    alarm(0);
    write(1, line, n);
    exit(0);
}

static void sig_alarm(int sig)
{
    return;
}
```

A kódnak két hátránya is van:

- ha a rendszer az alarm és read utasítások végrehajtása között a megengedettnél (10 másodperc) többet késik, a read hívás mindörökre leáll,

- ha a rendszerfüggvények túlsúlyban vannak, a read hívás nem szakad meg a SIGALRM jelzés kezelésekor; ebben az esetben az alarm függvénynek nincs értelme.

Azért, hogy ezt a kellemetlenséget elkerüljük, segítségünkre vannak a `setjmp` és a `longjmp` függvények.

## ≡ setjmp, longjmp

A jelzésekkel összeköttetésben, egy programban gyakran szükségünk van (nem helyi) ugrások végrehajtására. Erre két függvényünk van. A `setjmp` segítségével rögzíthetünk egy ugrási pontot a processzor állapotának a lementésével, míg a `longjmp` függvény elvégzi az ugrást egy, a paraméterén keresztül megadott pontra.

```
#include <setjmp.h>

int setjmp(jmp_buf jmpenv);
void longjmp(jmp_buf jmpenv, int val);
```

A `setjmp` függvényből két esetben térhetünk vissza:

- az ugrási pont sikeres megállapítása esetén; a függvény 0 értéket térít vissza,
- egy nem helyi ugrás elvégzése esetén a függvény visszatérési értéke az a `val` érték lesz, amellyel a `longjmp` függvényt hívtuk.

Most lássuk az előbbi példa javított változatát (`setjmp` és `longjmp` függvények használatával):

```
#include <setjmp.h>
#include <signal.h>
#include "<u>hdr.h</u>"

static void sig_alarm();
static jmp_buf env_alarm;

int main(void)
{
    int n;
    char line[MAXLINE];
    if (signal(SIGALRM, sig_alarm) == SIG_ERR)
        err_sys("hiba: signal(SIGALRM, ...)");
    if (setjmp(env_alarm) != 0)
        err_quit("lejart az olvasasra szant ido");
    alarm(10);
    if ((n = read(0, line, MAXLINE)) < 0)
        err_sys("read hiba");
    alarm(0);
    write(1, line, n);
    exit(0);
}

static void sig_alarm(int sig)
{
    longjmp(env_alarm, 1);
}
```

## sigaction

A `sigaction` függvény lehetővé teszi a jelzésekhez rendelt tevékenységek vizsgálatát és/vagy módosítását. A régebbi Unix verziókban a `sigaction` helyettesítette a `signal` függvényt. Szintaxisa:

```
#include <signal.h>
int sigaction(int jel,
              const struct sigaction *act,
              struct sigaction *vact);
```

ahol

```
struct sigaction
{
    void (*sa_handler)();
    sigset_t sa_mask;
    int sa_flags;
}
```

A függvény 0-t térít, ha a művelet sikeres volt, és -1-et különben. A `jel` argumentum annak a jelnek a sorszáma, amelyet vizsgálni és/vagy módosítani szeretnénk. Ha az `act` paraméter nem NULL a jelhez rendelt tevékenység módosulni fog. Ha a `vact` paraméter nem NULL a rendszer visszaadja a jelhez tartozó előbbi tevékenységet.

A `sigaction`-ról bővebben lásd a bibliográfiában.

## [Csővezeték \(pipe\)](#)

---

Copyright (C) Buzogány László, 2002



[About](#)








# Csővezeték (pipe)

Folyamatok közti kommunikáció név nélküli állományok segítségével...

---

## Definíció

### Műveletek pipe állományokkal

-  Létrehozás (`pipe`)
-  Bezárás (`close`)
-  Olvasás és írás (`read`, `write`)

## Példa

A fájlleíró megduplázása (`dup`, `dup2`)

Kétirányú kommunikáció

Egyszerűen pipe (`popen`, `pclose`, `fgets`, `fputs`)

---

## Jelzések (signals)

### Definíció

A kommunikációs csatornák működését már láthattuk a Unix shell parancsok végrehajtásakor. Például, ha egy rendezett listát szeretnénk megjeleníteni a hálózatra belépett felhasználókról a következőképpen járunk el:

```
$ who | sort | pr
```

A fenti példában három folyamatot két csatornával kötöttünk össze. Az adatok áramlása balról jobbra történik.

Unix alatt azonban pipe-okat programból is létrehozhatunk, amely nagy flexibilitást eredményez, és megoldja a folyamatok közötti körkörös kommunikációt is.

A **csővezeték (pipe)** a rendszer által létrehozott (név nélküli) speciális állomány. Maximális hossza 10 blokk (azaz 5120 byte).

**Feladata:** két folyamat közötti kommunikáció megvalósítása úgy, hogy az egyik folyamat ír a standard kimenetre, a másik pedig olvas a standard bemenetről.

A pipe FIFO szerkezetű (elsőnek be, elsőnek ki). Két mutatója van: egyik az író folyamaté, a másik az

olvasóé. Így az írás/olvasás cirkulárisan történik. Ha a pipe megtelik az írónak kell várnia, ha kiürül az olvasónak. A kiolvasott adatok törlődnek a listából.

A FIFO algoritmus szabályainak betartása érdekében, egy pipe állományt csak az őt létrehozó folyamat és annak leszármazottai használhatnak. Általában a pipe-ot egy olyan folyamat hozza létre, amely később meghív egy fork függvényt, és ezáltal a pipe mind a szülő, mind a gyerek számára láthatóvá válik. A fork hívás után az állománytábla bemenetei közösek lesznek.

Tehát levonhatjuk a következtetést: pipe fájlokat csak apa-fiú kapcsolatban levő vagy közös őssel rendelkező folyamatok használhatnak.

A pipe fájlok szintjén a következő műveleteket kell (lehet) elvégezni:

1. az állomány létrehozása a pipe függvény segítségével,
2. olvasás/írás a read/write függvények segítségével,
3. az O\_NDELAY mutató beállítása az fcntl függvénnyel, amely a következőket eredményezi: ha a read vagy write függvények nem fejeződnek be, az a folyamat, amely ezeket meghívta nem fog leállni, viszont a read/write függvények befejeződnek és 0 eredményt szolgáltatnak,
4. az állomány bezárása a close függvény segítségével,
5. az állomány (pontosabban az i-node bemenet) törlése; ezt a rendszer automatikusan végzi, amint az i-node-ra történő hivatkozások száma 0 lesz.

## Műveletek pipe állományokkal



### Létrehozás (pipe)

Egy csővezeték létrehozása a pipe függvény segítségével történik, amelynek alakja:

```
#include <unistd.h>
int pipe(int pfd[2]);
```

A függvény 0-t térít vissza, ha a létrehozás sikerült, és -1-et, ha nem.

A pfd egy két elemű táblázat, ahol a pfd[0]-ból olvasunk, és a pfd[1]-be írunk.

A pipe függvényhívás tehát egy olyan kommunikációs csatornát hoz létre, amelynek az egyik végén beírjuk az adatokat, a másik végén pedig kiolvassuk. A pfd[0]-ba való írás során (write) az adatok a pipe fájlba kerülnek, míg a pfd[1]-ből olvasva (read) törlődnek onnan.

Hiba esetén az errno változó a hiba kódját fogja tartalmazni.



### Bezárás (close)

A csővezeték (azaz a pipe fájl) bezárására szolgál a close függvény. Szintaxisa:

```
#include <unistd.h>
int close(int pfd);
```

A függvény 0-t térít vissza, ha a bezárás sikerült, és -1-et különben.

Látható, hogy a pfd argumentum egy egész szám, tehát csak az állomány egyik végét zárja be. A nem szükséges pipe végeket ajánlatos minél előbb bezárni! Például, ha egy programban a szülő információkat küld a gyereknek pipe-on keresztül, akkor nem szükséges a pfd[0] a szülőben (mivel ő nem olvas), sem a pfd[1] a gyerekben (mivel ő nem ír).

## Olvasás és írás (read, write)

Mivel a csővezeték is egyfajta fájl, a pipe olvasása és írása az állományokéhoz hasonlóan történik.

```
#include <unistd.h>
```

```
ssize_t read(int pfd, void *buf, size_t count);
```

```
ssize_t write(int pfd, const void *buf, size_t count);
```

vagy

```
#include <stdio.h>
```

```
int fscanf(FILE *stream, const char *format, ...);
```

```
int fprintf(FILE *stream, const char *format, ...);
```

A második változatot főként standard fájlok esetén használjuk. Pipe fájlok kezelésére a read és write függvényeket ajánljuk. Itt paraméterként meg kell adnunk a pipe fájl egyik végének azonosítóját (pfd), a buf paraméterben a beírandó vagy kiolvasandó információt, míg a count változóba ennek méretét.

A függvények visszatérített értéke a pipe-ból sikeresen kiolvasott (beírt) bájtok száma.

## Példa

Készítsünk egy apa és egy fiú folyamatot, majd küldjünk el egy szöveget az apától a fiúnak pipe vezeték segítségével. A gyerek írja ki, hogy hány bájtot olvasott, és jelenítse meg az üzenet szövegét.

```
#include "hdr.h"
```

```
int main(void)
```

```
{
```

```
    int pfd[2], n; /* pipe változó, olvasott bájtok száma */
```

```
    pid_t pid;
```

```
    char buf[MAXLINE]; /* ebbe fogja kiolvasni a gyerek az inf. */
```

```
    char test[] = "pipe teszt\n"; /* ezt küldi a gyerek */
```

```
    if (pipe(pfd) < 0) /* pipe létrehozása */
```

```
        err_sys("pipe hiba");
```

```
    if ((pid = fork()) < 0) /* gyerekfolyamat létrehozása */
```

```
        err_sys("fork hiba");
```

```
    else
```

```
        if (pid == 0) /* gyerek */
```

```
        {
```

```
            close(pfd[1]); /* fölösleges vezeték bezárása */
```

```
            n = read(pfd[0], buf, MAXLINE); /* információ kiolvasása */
```

```
            printf("beolvastam %d bajtot: \n", n);
```

```

    fflush(stdout);
    write(1, buf, n);
}
else                                /* szülő */
{
    close(pfd[0]);                   /* fölösleges vezeték bezárása */
    write(pfd[1], test, sizeof(test)); /* információ beírása a pipe-ba */
}
exit(0);
}

```

## A fájlleíró megduplázása (dup, dup2)

Sok program az adatokat a standard bemenetről olvassa, amelynek fájlleírója 0, vagy az adatokat a standard kimenetre írja, amelyhez az 1 fájlleíró tartozik. Tehát ahhoz, hogy két programot a parancssorban össze tudjunk kapcsolni (cd1 | cd2), a pipe fájlhoz ezen két fájlleírót (tehát a standard kimenetet és bemenetet) kell kapcsolnunk. A két fájlleíró egyszerű megnyitása a pipe hívás előtt egyáltalán nem garantálja, hogy pontosan ezen fájlleíró értékeket fogják tartalmazni. Ezért, a kívánt fájlleírók megnyitása után szükséges a pipe fájlleírók megduplázása a dup és a dup2 függvények segítségével.

Ez azt jelenti, hogyha a dup meghívása előtt már megnyitottuk a 0 fájlleírót, ez a hívás biztosan 0-val fog visszatérni. A függvények szintaxisa:

```

#include <unistd.h>

int dup(int pfd);
int dup2(int pfd, int npfd);

```

A függvény az új fájlleírót téríti vissza, ha a művelet sikerült, illetve -1-et hiba esetén.

A dup sajátossága, hogy a nem használatosak közül mindig a legkisebb számú fájlleírót téríti vissza (azaz hozzárendeli a pfd változóhoz). Ezzel ellentétben a dup2 az npfd változó által megadott új fájlleírót rendeli a pfd-hez.

A pipe-okhoz rendelt fájlleírók megduplázását megértése érdekében nézzük a következő példát, amely a parancssorban megadott fájlt a képernyőre írja. A program az állomány tartalmát oldalról-oldalra jeleníti meg. Azért, hogy az egész állomány tartalmát ne kelljen egy időszakos állományba írni, s azután a system függvénnyel kiíratni, e program kimenetelét össze kell kapcsolnunk a lapok formázását végző programmal. Ennek érdekében előbb létrehozunk egy pipe állományt, majd a fork függvény segítségével egy gyereket. Ezután a gyerekfolyamat standard kimenetét hozzárendeljük az olvasó fájlleíró vezetékehez, végül pedig meghívjuk az exec függvényt, hogy az oldalformázásokat elvégezze.

```

#include <sys/wait.h>
#include "hdr.h"
#define DEF_PAGER "/usr/bin/more"

void main(int argc, char *argv[])
{
    int pfd[2], n;
    pid_t pid;
    char buf[MAXLINE], *pager, *arg;

```

```

FILE *fp;

if (argc != 2)
    err_quit("hasznalat: a.out <path>");

if (pipe(pfd) < 0)
    err_sys("pipe hiba");

if ((fp = fopen(argv[1], "r")) == NULL)
    err_sys("fopen hiba %s", argv[1]);

switch (pid = fork())
{
    case -1:
        err_sys("fork hiba");
    case 0:
        /* gyerek olvas */
        close(pfd[1]);
        if (pfd[0] != 0)
        {
            if (dup2(pfd[0], 0) != 0)
                err_sys("dup2 hiba");
            close(pfd[0]);
        }
        if ((pager = getenv("PAGER")) == NULL)
            pager = DEF_PAGER;
        if ((arg = strrchr(pager, '/')) != NULL)
            arg++;
        else
            arg = pager;
        if (execl(pager, arg, NULL) < 0)
            err_sys("execl hiba: %s", pager);
    default:
        /* szülő ír */
        close(pfd[0]);
        while (fgets(buf, MAXLINE, fp) != NULL)
        {
            n = strlen(buf);
            if (write(pfd[1], buf, n) != n)
                err_sys("write hiba");
        }
        if (ferror(fp))
            err_sys("ferror hiba");
        close(pfd[1]);
        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid hiba");
        exit(0);
        /* azért nincs több kommentár, hogy */
        /* mindenki alaposan gondolja át ;-) */
}
}

```

## Kétirányú kommunikáció

Az előbbi példákban a kommunikáció csak egyirányú volt, azaz mindig a szülőtől haladt az információ a gyerekhez. Egy apa-fiú kapcsolatban akkor akkor beszélhetünk kétirányú kommunikációról, ha a gyerek is tud adatot küldeni a szülő felé. Ezt, egyszerűen két pipe fájlal fogjuk megvalósítani.

Nevezzük a két pipe-ot például pfd1 és pfd2-nek. Oldjuk meg a következő feladatot: a szülő elküld a gyereknek egy n egész számot, mire a gyerek 0-t küld vissza, ha a szám páros, és 1-et, ha páratlan.

A pipe mechanizmust a következő ábra szemlélteti:



A program pedig a következő:

```
#include "hdr.h"

int main(void)
{
    int n, k, pfd1[2], pfd2[2];
    printf("kerem a szamot:\n");           /* az n bekérése */
    scanf("%d", &n);
    pipe(pfd1);                            /* a két pipe fájl létrehozása */
    pipe(pfd2);
    if (fork())                            /* itt be lehet tenni még hibaellenőrzést is */
    {                                       /* szülő */
        close(pfd1[0]);                  /* fölösleges pipe-ok bezárása */
        close(pfd2[1]);
        write(pfd1[1], &n, sizeof(int)); /* az n elküldése a pfd1 pipe-on keresztül */
        close(pfd1[1]);
        read(pfd2[0], &k, sizeof(int));  /* az eredmény kiolvasása a pfd2-n keresztül */
        if (k == 0)
            printf("a szam paros\n");    /* az eredmény kiírása */
        else
            printf("a szam paratlan\n");
        exit(0);
    }
    else                                  /* gyerek */
    {
        close(pfd1[1]);
        close(pfd2[0]);
        read(pfd1[0], &n, sizeof(int));  /* az n kiolvasása a pfd1 pipe-ból */
        close(pfd1[0]);
    }
}
```

```

    k = n%2;                                /* k=1, ha az n páratlan, k=0, ha n páros */
    write(pfd2[1], &k, sizeof(int));        /* az eredmény elküldése a pfd2 pipe-on */
    close(pfd2[1]);
    exit(0);
}
}

```

## Egyszerűen pipe (popen, pclose)

A popen és a pclose függvények bizonyos esetekben megkönnyítik a pipe-ok használatát. A popen a következőket teszi:

- létrehoz egy pipe-ot (pipe),
- létrehoz egy gyerekfolyamatot (fork),
- a kommunikáció irányától függően bezárja a nem szükséges csatornákat (close),
- a gyerekfolyamatban végrehajtja a megadott parancsot (exec),
- megvárja, ameddig a parancs befejeződik (wait).

A két függvény alakja:

```

#include <stdio.h>

FILE *popen(const char *cmd, const char *type);
int pclose(FILE *fp);

```

A popen egy állományra mutató pointert ad vissza, ha a művelet sikeres volt, és NULL értéket egyébként. A pclose függvény visszaadja a cmd parancs végrehajtásának befejeződési állapotát, vagy a -1 értéket hiba esetén.

A popen függvény tehát létrehoz egy gyerekfolyamatot (fork), majd az exec függvény segítségével végrehajtja a cmd parancsot. Ha a type argumentum értéke "r", az állományhoz rendelt mutató a cmd parancs kimenete lesz (tehát a fájlból olvashatunk). Ha az argumentum értéke "w", a mutató a cmd parancs bemenetéhez lesz hozzárendelve (így írni tudunk a fájlba).

A pclose lezárja az állománymutatókat, és megvárja a gyerekfolyamat által végrehajtott parancs befejezését, majd visszatéríti a parancs kilépési kódját.

Feladat: Listázzuk egy állomány tartalmát a more parancs segítségével!

```

#include "hdr.h"

int main(int argc, char *argv[])          /* argumentumok átadása */
{
    if (argc != 2)                         /* argumentumok számának ellenőrzése */
        err_quit("hasznalat: a.out <fajlnev>");

    char buf[MAXLINE];
    int err;

    FILE *megadott;                        /* az arg.-ban megadott, bemeneti fájl */
    FILE *fp;                              /* ez lesz a more parancs bemenete */

    megadott = fopen(argv[1], "r");        /* a megadott fájl megnyitása olvasásra */
    fp = popen("more", "w");              /* a more parancs futtatása gyerekként */
}

```

```

while (fgets(buf, MAXLINE, megadott)          /* a fájl átküldése a more bemenetére */
    fputs(buf, fp);

err = pclose(pf);                             /* állománymutatók bezárása */
fclose(megadott);

if (err == -1)                                /* hiba kezelése */
    err_sys("more hiba");
}

```

Amint a fenti példában is láthatjuk külön függvények vannak a popen által megnyitott fájlok írására és olvasására.

Ha a popen-t olvasásra ("r") nyitottuk meg, akkor az fgets függvényt kell használnunk. Alakja:

```

#include <stdio.h>
char *fgets(char *s, int size, FILE *stream);

```

A függvény size bájtot próbál kiolvasni a stream fájlból. Az eredményt (sikeresen kiolvasott információt) az s stringben kapjuk vissza. A függvény visszatérített értéke szintén s, ha az olvasás sikeres volt, különben NULL.

Ha a popen-t írásra ("w") nyitottuk meg, akkor az fputs függvényt használjuk:

```

#include <stdio.h>
int fputs(const char *s, FILE *stream);

```

A függvény a stream fájlba írja az s változóban megadott stringet. A visszatérített érték egy nemnegatív szám, ha a művelet sikeres volt, különben EOF.

**Feladat:** A következőkben olyan C programot írunk, amely két folyamatot hoz létre: egy szervert és egy klienst. A kliensfolyamat egy kérést intéz a szerverhez, mire a szerver a popen parancs segítségével megkeresi a választ, majd egy pipe fájlon visszaküldi az eredményt a klienshez. Ebben az esetben a kliens küldjön a szervernek egy host nevet, a szerver pedig ellenőrizze, hogy az illető host létezik-e vagy sem, és küldjön vissza a szervernek egy megfelelő üzenetet.

```

#include "hdr.h"

int main(int argc, char *argv[])
{
    int pipe1[2], pipe2[2];                    /* két pipe a kétirányú kommunikációhoz */
    int pid;
    char all[3];

    if (argc != 2)                            /* ha a parancssorban nincs paraméter */
        err_quit("hasznalat: hc <hostnev>\n");

    pipe(pipe1); // client --> server          /* pipe-ok létrehozása */
    pipe(pipe2); // server --> client

    switch (pid = fork())                      /* gyerekfolyamat létrehozása */
    {
        case -1 :                             /* hiba a létrehozáskor */
            err_quit("fork hiba");

```



```

case 0 :                                /* gyerek - szerver */
{
    FILE *ps;                            /* a popen ebből fog olvasni */
    char shellcmd[200];                  /* kiadandó shell parancs */
    char c;                             /* az eredmény karakterenként itt lesz */
    int i;

    close(pipe1[1]);                     /* pipe1 --> - olvasás a pipe1-ből */
    close(pipe2[0]);                     /* --> pipe2 - írás a pipe2-be */

    strcpy(shellcmd, "ping -c 3 ");      /* a shell parancs eleje */
    i = strlen(shellcmd)-1;

    do
    {
        read(pipe1[0], &c, sizeof(char)); /* olvassa a host nevet ... */
        if (i < 149) i++;
        shellcmd[i] = c;                  /* ... hozzátoldja a shell parancshoz */
    } while (c != 0);

    ps = popen(shellcmd, "r");            /* shell parancs vegrehajtása popen-nel */
    do
    {
        c = fgetc(ps);                    /* eredmény kiolvasása a popen-ből */
        write(pipe2[1], &c, sizeof(char)); /* eredmény küldése a kliensnek */
    } while (c != EOF);

    pclose(ps);                           /* shell parancs vége */

    exit(0);                              /* folyamat vége */
}

default :                                /* szülő - kliens */
{
    char c;

    close(pipe1[0]);                      /* --> pipe1 - írás a pipe1-be */
    close(pipe2[1]);                      /* pipe2 --> - olvasás a pipe2-ből */

    write(pipe1[1], argv[1], strlen(argv[1])+1);
    wait();                               /* a host név elküldve; várakozás */

    do
    {
        read(pipe2[0], &c, sizeof(char)); /* válasz kiolvasása a pipe2-ből */
        if (c != EOF)
            printf("%c", c);              /* válasz kiírása */
    } while (c != EOF);

    waitpid(pid, all, 0);

    exit(0);                              /* kliens vége */
}

```

```
}  
  
}
```

Névvel ellátott csővezeték (FIFO)

Copyright (C) Buzogány László, 2002



About



# Névvel ellátott csővezeték (FIFO)

Azaz olyan állomány, amelyhez bármelyik folyamat hozzáférhet amelynek van rá megfelelő joga...

---

## Definíció

### Műveletek FIFO állományokkal

Létrehozás (mknod, mkfifo)

Megnyitás (open)

Olvasás és írás (read, write)

Bezárás (close)

Törlés (unlink)

### Folyamatok közti kommunikáció FIFO állományok segítségével

#### Példák

1.példa (apa-fiú kapcsolat)

2. példa (kliens-szerver rendszer)

---

## Csővezeték (pipe)

### Definíció

A névvel ellátott csővezeték (FIFO állomány) a közönséges fájl és a pipe kombinációja. A FIFO állománynak van egy szimbolikus neve, és egy könyvtára, ahová létrehozzuk. Mindemellett megőrzi a pipe fájlok összes jellemzőit.

A FIFO állományok előnye a pipe-okkal szemben az, hogy a rajtuk keresztül kommunikáló folyamatok nem kell továbbítsák a fájlleírókat. Éppen ezért a FIFO állományokat nem csak az a folyamat (vagy annak leszármazottja) használhatja, amely azt létrehozta, hanem más, külső folyamatok is. Tehát bármely folyamat megnyithat írásra, illetve olvasásra egy FIFO fájlt, csak a szimbolikus nevet kell ismernie.

Egy FIFO állomány megtekinthető az `ls -l` Unix parancs segítségével. A hozzáférési jogok első mezője ez esetben egy `p` betű.

### Műveletek FIFO állományokkal

Létrehozás (mknod, mkfifo)

A FIFO állományok létrehozására két parancs is létezik: az `mknod` és az `mkfifo`. A második függvény tulajdonképpen az első hívja, s a különbség csak a paraméterek számában van. Alakjuk:

```
#include <sys/types.h>
#include <sys/stat.h>

int mknod(char *pathname, int mode, 0);
int mkfifo(const char *pathname, mode_t mode);
```

Sikeres létrehozás esetén mindkét függvény 0-t térít vissza, különben -1 értéket. A `pathname` argumentum a FIFO állomány elérési útvonala. A `mode` egész típusú változó az `S_FIFO` állomány típusát és a hozzáférési jogokat (r, w, x) jelöli a tulajdonos, a csoport és a többi felhasználó számára. Ezt a paramétert a következő alakban kell megadni:

**S\_IFIFO | jogok**

Például ahhoz, hogy egy 'read' és 'write' jogokkal rendelkező FIFO állományt létrehozzunk a tulajdonosnak, a csoportjának és a többiek számára, használhatjuk a következő függvényt:

```
int makepipe(char *path)
{
    return (mknod(path, S_IFIFO | 0666, 0));
}
```

## Megnyitás (open)

Egy FIFO állomány megnyitása írásra vagy olvasásra a hagyományos fájlkezelőhöz hasonlóan történik az `open` függvény használatával, azaz:

```
#include <sys/types.h>
#include <sys/stat.h>

int open(const char *pathname, int flags);
```

A függvény visszatérített értéke az új fájlleíró, vagy -1 hiba esetén. A `pathname` argumentum a FIFO állomány elérési útvonala, a `flags` változóban pedig a hozzáférési jogokat kell megadni, amely éppen a következő konstansok egyike:

- `O_RDONLY`, csak olvasható,
- `O_WRONLY`, csak írható,
- `O_RDWR`, olvasható és írható.

Például:

```
...
int fd;
fd = open("FIFO_1", O_WRONLY);
...
```

## Olvasás és írás (read, write)

A FIFO állományok olvasása pontosan úgy történik, mint a hagyományos fájlkezelő. A `read` és a `write` függvények szintaxisa megtalálható [itt](#).

## Bezárás (close)

A FIFO állományok bezárása is hasonló a hagyományos fájlokéhoz, illetve a pipe-oknál látottakhoz. A `close` parancs szintaxisának megtekintéséhez kattints [ide](#).

Megjegyzés: Ha egy FIFO állományt bezárunk, akkor az őt olvasó folyamat az illető állományt befejezettnek látja.

## Törlés (unlink)

A pipe fájlokkal ellentétben a FIFO állományok törlése az `unlink` függvény segítségével történik. Az állományt akkor nyilváníthatjuk töröltnek, ha a rá vonatkozó hivatkozások száma zero.

```
#include <unistd.h>
int unlink(const char *pathname);
```

A függvény tehát törli a `pathname` elérési útvonalon (lehet csak állománynév is) keresztül megadott FIFO állományt. A visszatérített érték 0, ha a törlés sikerült, és -1 különben.

## Folyamatok közti kommunikáció FIFO állományok segítségével

A folyamatok közti kommunikáció a FIFO állományok segítségével a következő lépések szerint történik:

- egy folyamat a szimbolikus név alapján létrehozza a FIFO állományt az `mknod` vagy `mkfifo` függvények segítségével,
- egy folyamat, amely információt szeretne közölni egy másikkal megnyitja a FIFO állományt az `open` függvénnyel, és a `write` segítségével beírja az adatokat,
- egy folyamat, amely az adatokat szeretné kiolvasni, megnyitja a FIFO állományt olvasásra az `open` függvénnyel, majd a `read` segítségével kiolvassa a kívánt információkat,
- egy folyamat a szimbolikus név alapján törli a FIFO állományt az `unlink` függvénnyel.

## Példák

### 1. példa (apa-fiú kapcsolat)

A folyamatok közti kommunikáció kétirányú is lehet. Ekkor két FIFO állományt kell létrehoznunk. A következő példa e kétirányú információcserét valósítja meg egy apa-fiú kapcsolatban.

```
#include "hdr.h"

int main(void)
{
    int gyerek;
    ...
    mknod("PIPE_FIFO_1", 0666 | S_IFIFO, 0);          /* FIFO fájl létrehozása */
    mknod("PIPE_FIFO_2", 0666 | S_IFIFO, 0);
    ...
    if ((gyerek = fork()) < 0)                          /* gyerekfolyamat létrehozása */
        err_sys("fork hiba");
```

```

if (gyerek)
    apa();                                /* P1 folyamat */
else
    fiu();                                /* P2 folyamat */
exit(0);
}

void apa(void)                            /* P1 folyamat */
{
    int fd1, fd2;
    ...
    if ((fd1 = open("PIPE_FIFO_1", O_WRONLY)) < 0) /* FIFO fájl megnyitása írásra */
        err_sys("hiba a PIPE_FIFO_1 nyitása során");
    if ((fd2 = open("PIPE_FIFO_2", O_RDONLY)) < 0) /* másik FIFO nyitása olvasásra */
        err_sys("hiba a PIPE_FIFO_2 nyitása során");
    ...
    write(fd1, ...);                      /* írás a PIPE_FIFO_1-be */
    ...
    read(fd2, ...);                       /* olvasás a PIPE_FIFO_2-ből */
    ...
    close(fd1);                           /* FIFO fájlok bezárása */
    close(fd2);
    exit(0);
}

void fiu(void)                            /* P2 folyamat */
{
    int fd1, fd2;
    ...
    fd1 = open("PIPE_FIFO_1", O_RDONLY);    /* FIFO fájl megnyitása olvasásra */
    fd2 = open("PIPE_FIFO_2", O_WRONLY);    /* FIFO fájl megnyitása írásra */
    ...
    read(fd1, ...);                       /* olvasás a PIPE_FIFO_1-ből */
    ...
    write(fd2, ...);                      /* írás a PIPE_FIFO_2-be */
    ...
    close(fd1);                           /* FIFO fájlok bezárása */
    close(fd2);
    exit(0);
}

```

## 2. példa (kliens-szerver rendszer)

A következőkben egy olyan FIFO állományt hozunk létre, amely nem csupán apa-fiú kapcsolatoknál működik, hanem tetszőleges két folyamat között. A feladat egy olyan kliens-szerver rendszer (két különálló állományból áll!) létrehozása, amelyben a kliens küld egy számot a szervernek, mire a szerver válaszként visszaküldi a szám négyzetét.

Megjegyzések:

- a szerver létrehoz egy szerverfifot, amelyre az összes kliens csatlakozni fog,
- minden kliensnek külön FIFO-ja van, amelyet minden kliens magának készít el; ezért amikor a kliens a szervernek elküldi a kérést, valahogyan jeleznie kell, hogy milyen nevű FIFO-n keresztül szeretné a választ megkapni; a legegyszerűbb, ha a kliens FIFO-jának nevében szerepel a kliens folyamatazonosítója is, így a név egyértelmű,
- a kliens előbb megnyitja a saját FIFO-ját olvasásra, s csak azután küldi el az üzenetet a szerver felé,
- a szerver FIFO-ja sosem záródik be,
- a kliens FIFO-jának a szerver oldalát a szerver a válaszadás után bezárja; ha újabb kérés érkezik, újból megnyitja,
- ha a kliens befejezte működését be kell zárnia a saját FIFO-ját.

Mivel a FIFO-n küldött adatok típusa megegyezik a szerverben és a kliensben, a könnyebb kezelhetőség érdekében ajánlatos egy közös adatszerkezetet létrehozni, és ezt egy külön `hdr` állományban tárolni. Esetünkben ez a következő lesz:

### struktura.h

```
typedef struct elem
{
    int szam;
    int pid;
} azon;
```

A kliens-szerver rendszert megvalósító folyamatok a következők lesznek:

### szerver.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include "struktura.h"                                /* a fent megadott fejléc */

int main(void)
{
    int fd, fd1;                                     /* szerver- és kliensfifo */
    char s[15];                                     /* kliensfifo neve; pl. fifo_143 */
    azon t;                                          /* küldendő "csomag" */

    mkfifo("szerverfifo", S_IFIFO|0666);           /* a szerver létrehozza a saját fifo-ját */
    fd = open("szerverfifo", O_RDONLY);             /* megnyitja olvasásra; jöhetnek a számok */

    do                                              /* addig megy, míg 0-t nem küld egy kliens */
    {
        read(fd, &t, sizeof(t));                   /* szám kiolvasása */
        t.szam = t.szam * t.szam;
        sprintf(s, "fifo_%d", t.pid);              /* a pid segítségével meghat. a kliensfifo nevét */
        fd1 = open(s, O_WRONLY);                   /* kliensfifo megnyitása írásra */
        write(fd1, &t, sizeof(t));
        close(fd1);                                /* adatok elküldve, kliensfifo vége */
    }
```

```

    } while (!t.szam);
    close(fd);
    unlink("szerverfifo");
    exit(0);
}
/* szerverfifo vége */
/* törli a szerverfifot, hiszen ő hozta létre */

```

## kliens.c

```

#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include "struktura.h"
/* a fenti fejlécállomány */

int main(int argc, char *argv[])
/* a számot a parancssorban adjuk meg */
{
    int fd, fd1;
    /* kliens- és szerverfifo */
    char s[15];
    azon t;

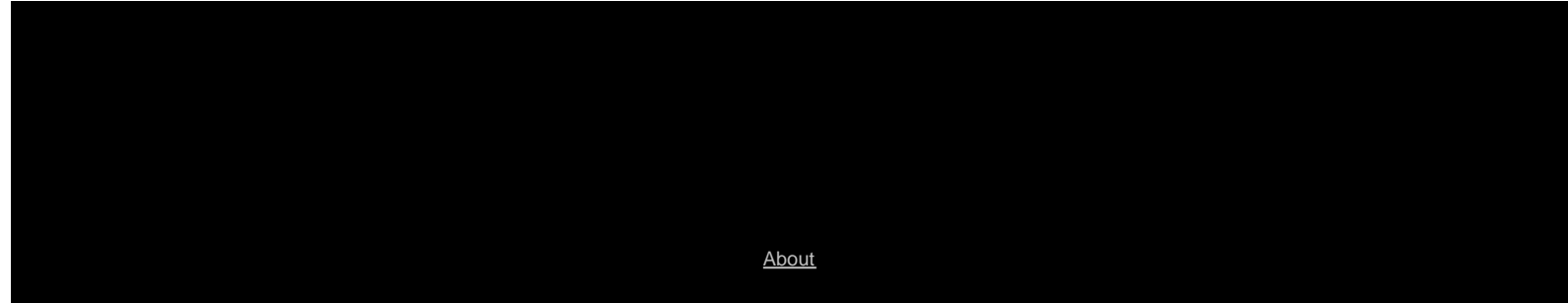
    if (argc != 2)
        /* nincs megadva argumentum, hiba */
    {
        printf("hasznalat: kliens <szam>\n");
        exit(1);
    }

    sprintf(s, "fifo_%d", getpid());
    /* meghat. a fifonevet a pid segítségével */
    mkfifo(s, S_IFIFO|0666);
    /* létrehoz egy kliensfifot */
    fd = open("szerverfifo", O_WRONLY);
    t.pid = getpid();
    /* a küldendő adatok */
    t.szam = atoi(argv[1]);
    /* string átalakítása számmá */
    write(fd, &t, sizeof(t));
    /* küldi a szervernek */
    fd1 = open(s, O_RDONLY);
    read(fd1, &t, sizeof(t));
    /* a válasz */
    close(fd1);
    unlink(fd1);
    /* kliensfifo törlése */
    printf("a negyzete: %d\n", t.szam);
    exit(0);
}

```

## Üzenetsorok (message queues)





[About](#)



# Üzenetsorok (message queues)

Folyamatok közti kommunikáció üzenetek (kis adatcsomagok) küldözgetésével...

## Definíció

### Műveletek üzenetsorokkal

↳ Létrehozás (`msgget`)

↳ Az üzenetsor adatainak lekérdezése, módosítása és törlése (`msgctl`)

↳ Üzenet küldése (`msgsnd`)

↳ Üzenet fogadása (`msgrcv`)

## Példa

### Névvel ellátott csővezeték (FIFO)

#### Definíció

A folyamatok közti kommunikáció egyik leghatékonyabb módja az üzenetsorok használata. Az üzenetsorokat a rendszer magja felügyeli. Egy ilyen sorban az adatok (vagyis az üzenetek) cirkulárisan közlekednek, a folyamatok közti szinkronizáció pedig a előállító/fogyasztó elv alapján valósul meg. Magyarul: ha az üzenetsor megtelt, az előállító leáll, ameddig egy fogyasztó ki nem olvas egy üzenetet, illetve ha az üzenetsor kiürült, a fogyasztónak kell várnia az első (neki címzett) üzenet érkezéséig.

Tehát az üzenetsor egy, a rendszer által tárol láncolt lista, amelynek jól meghatározott azonosítója van. Valamely folyamat egy üzenetsort egy kulcs segítségével azonosít. (Vigyázat! Nem tévesztendő össze az azonosító és a kulcs. A kulcsot mi választjuk, az azonosítót pedig a rendszer minden üzenetsorhoz automatikusan rendeli hozzá.)

Az a folyamat, amely üzenetet szeretne küldeni, előbb az adott kulcs alapján az `msgget` függvénnyel lekéri az üzenetsor azonosítóját, majd az `msgsnd` függvény segítségével elküldi az üzenetet. Az üzenet a sor végére kerül.

Az a folyamat, amely üzenetet szeretne fogadni, ugyanúgy lekérdezi a kulcs és az `msgget` függvény segítségével az üzenetsor azonosítóját, majd az `msgrcv` függvénnyel kiolvassa az üzenetet. Az üzenetsorból való olvasás nem feltétlenül a FIFO módszer szerint történik, ugyanis az `mtype` mező értékétől függően az üzeneteket tetszőleges sorrendben is kiolvashatjuk.

Egy üzenet szerkezete az `msg.h` állományban a következőképpen van definiálva:

```
struct msgbuf
```

```
{
    long mtype;
    char mtext[1];
};
```

Az `mtype` mező egy hosszú egész számot tartalmaz, amely az üzenet típusát jelképezi. Ezután következik a tulajdonképpeni üzenet (`mtext`), amely jelen esetben egyetlen karakterből áll. Amennyiben ennél hosszabb üzenetet szeretnénk egyszerre küldeni, mi is definiálhatunk hasonló szerkezetű struktúrákat. Például:

```
struct msg
{
    long tip;
    char uzenet[256];
};
```

Amint látjuk, az üzenetek szerkezete nem rögzített, de kötelezően tartalmazniuk kell a típust és magát az üzenetet. Ez utóbbi hossza mindig az alkalmazástól függ.

Minden üzenetsorhoz a rendszer hozzárendel egy `msqid_ds` típusú struktúrát, amelynek szerkezete:

```
struct msqid_ds
{
    struct ipc_perm msg_perm; /* definiálja a jogokat és a tulajdonost */
    struct msg *msg_first; /* mutató az első üzenetre */
    struct msg *msg_last; /* mutató az utolsó üzenetre */
    ulong msg_cbytes; /* sor bájtjainak száma */
    ulong msg_qnum; /* üzenetek száma a sorban */
    ulong msg_qbytes; /* maximális bájtok száma sorban */
    pid_t msg_lspid; /* utolsó msgsnd() pid-je */
    pid_t msg_lrpid; /* utolsó msgrcv() pid-je */
    time_t msg_stime; /* utolsó msgsnd() ideje */
    time_t msg_rtime; /* utolsó msgrcv() ideje */
    time_t msg_ctime; /* utolsó struktúramódosítás ideje */
};
```

ahol az `ipc_perm` struktúra a következőképpen definiált:

```
struct ipc_perm
{
    uid_t uid; /* effektív felhasználó ID-je */
    gid_t gid; /* effektív csoport ID-je */
    uid_t cuid; /* effektív létrehozó felhasználó ID-je */
    gid_t cgid; /* effektív létrehozó csoport ID-je */
    mode_t mode; /* hozzáférési jogok */
    ulong seq; /* număr de secvență utilizare slot */
    key_t key; /* kulcs */
};
```

Egy folyamat csak akkor férhet hozzá egy üzenetsorhoz ha:

- a folyamat a superuser-é,

- a felhasználó ID-je (uid) megegyezik az `msg_perm.cuid`-vel vagy az `msg_perm.uid`-vel és az `msg_perm.mode` tartalmazza a kívánt jogokat,
- a felhasználó csoportazonosítója (gid) megegyezik az `msg_perm.cgid` vagy `msg_perm.gid` értékek egyikével és az `msg_perm.mode` tartalmazza a kívánt jogokat,
- a felhasználó beleesik a "többi felhasználó" kategóriába és az `msg_perm.mode` tartalmazza a megfelelő jogokat.

## ☰ Műveletek üzenetsorokkal

### ☰ Létrehozás (msgget)

Az `msgget` rendszerfüggvény engedélyezi a folyamatnak, hogy létrehozzon egy üzenetsort felhasználva egy bizonyos kulcsot. Alakja:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int flg);
```

A függvény visszatérési értéke az újonnan létrehozott vagy egy régi üzenetsor azonosítója, ha a művelet sikerült, különben -1. A `key` változóban az üzenetsorhoz rendelt kulcsot kell megadni, míg az `flg`-ben a létrehozási tevékenységet és a hozzáférési jogokat. A kulcsot az üzenetsorhoz kapcsolódó összes folyamatnak ismernie kell.

Egy új üzenetsor létrehozása esetén az `flg` mezőt a következő formában kell megadni:

`IPC_CREAT | IPC_EXCL | hozzáférési_jogok`

Példaként hozzunk létre egy 2003 kulccsal rendelkező üzenetsort, és adjunk hozzá írási és olvasási jogot minden felhasználónak.

```
#define KEY 2003

int msgid;
msgid = msgget((key_t) KEY, IPC_CREAT | 0666);
```

Ha az üzenetsor már létezik, de meg szeretnénk határozni az azonosítóját (ID-ját), akkor az `flg` mezőbe 0-t írunk – ez esetben a függvény nem fog létrehozni új sort.

```
msgid = msgget((key_t) KEY, 0);
```

Létrehozáskor a társított adatstruktúra (`msg_perm`) mezői a következő információkkal töltődnek fel:

- `msg_perm.cuid`, `msg_perm.uid` – az `msgget` függvényt meghívó folyamathoz hozzárendelt felhasználó ID-ja,
- `msg_perm.cgid`, `msg_perm.gid` – az `msgget` függvényt meghívó folyamathoz hozzárendelt felhasználócsoport ID-ja,
- `msg_perm.mode` – az `msgget` függvény hívásakor megadott `flg` argumentum, amely a hozzáférési jogokat tartalmazza,
- `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, `msg_rtime` – értéke 0,
- `msg_ctime` – az aktuális időt tartalmazza,

- `msg_qbytes` – azt a legnagyobb megengedett értéket tartalmazza, amely a rendszer létrehozásakor volt rögzítve.

Az `msgget` függvény hívásakor visszaadott azonosítót az üzenetsorral dolgozó függvények használják.

### ☰ Az üzenetsor adatainak lekérdezése, módosítása és törlése (`msgctl`)

Az `msgctl` függvény az üzenetsorok szintjén az információk lekérdezésére, módosítására és törlésére használható. Szintaxisa:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int msgid, int cmd, struct msqid_ds *buf);
```

A függvény visszatérési értéke 0, ha a művelet sikeres volt, ellenkező esetben -1. Az `msgid` paraméter az `msgget` függvény által meghatározott üzenetsor azonosítója.

A `cmd` argumentum a kívánt műveletet határozza meg és a következő értékeket veheti fel:

- `IPC_STAT` – az üzenetsorhoz rendelt struktúra tartalma a `buf` változóba kerül,
- `IPC_SET` – az üzenetsorhoz rendelt struktúrát frissíti a `buf` által megadott struktúrával,
- `IPC_RMID` – az üzenetsorhoz rendelt struktúrát törli; a művelet azonnali és minden folyamat, amely ezt a sort használja az `errno=EIDRM` üzenetet kapja; ez esetben a `buf` argumentumnak a `NULL` értéket kell adni.

Példa:

```
#define KEY 2003

msgid = msgget((key_t) KEY, 0);
msgctl(msgid, IPC_RMID, NULL);
```

### ☰ Üzenet küldése (`msgsnd`)

Az `msgsnd` függvény egy már létrehozott üzenetsorba ír egy adott üzenetet. Szintaxisa:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msgid, const void *addr, size_t nbytes, int flg);
```

A függvény visszatérési értéke 0, ha a művelet sikeres volt, ellenkező esetben -1. Az `msgid` az `msgget` által meghatározott üzenetsor azonosítója. Az `addr` pointer típusú argumentum az üzenetre mutat. Ez lehet `void` vagy `msgbuf` típusú. Az előbbi esetben az üzenet szerkezete tetszőleges lehet. Az `nbytes` megadja az üzenet tartalmának a hosszát (nem az egész üzenetét!). Az `flg` paraméterben megadhatjuk, hogy a rendszer hogyan viselkedjen, ha az üzenetet nem lehet beírni a sorba. Például, ha a sor megtelt és az `IPC_NOWAIT` opciót választottuk, akkor a folyamat nem áll le, hanem visszatér a függvényből, és az `errno` az `EAGAIN` hibaüzenetet fogja tartalmazni.

A következő példa egy üzenetnek a sorba való beírását mutatja be:

```
#define KEY 2003
struct msgbuf uzenet;
char *uzen = "probauzenet";

msgid = msgget((key_t) KEY, 0);
uzenet.mtype = 100;
strcpy(uzenet.mtext, uzen);
msgsnd(msgid, &uzenet, strlen(uzen), IPC_NOWAIT);
```

## ☰ Üzenet fogadása (msgrcv)

Az `msgrcv` függvény feladata kiolvasni egy üzenetet az üzenetsorból. A paraméterek segítségével megadhatjuk, hogy milyen típusú üzeneteket fogadjon. Alakja:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv(int msgid, void *addr, size_t nbytes, long msgtype, int flg);
```

A függvény sikeres olvasás esetén visszaadja a kiolvasott bájtok számát, különben a -1 értéket. Az `msgid` az `msgget` által meghatározott üzenetsor azonosítója. Az `addr` pointer a kapott üzenetre mutat.

Az `nbytes` a lekérdezett üzenet maximális hossza bájtokban. Az üzenet valós mérete különbözhet az `nbytes`-ban megadottól. Ha az üzenet hossza kisebb a megadott maximális méretnél, akkor minden OK. Ha viszont átlépi az `nbytes` határt, akkor két eset lehetséges:

- Ha az `MSG_NOERROR` be van állítva, az üzenet megcsonkul, de nem kapunk hibaüzenetet.
- Ha az `MSG_NOERROR` nincs beállítva, a függvény nem olvassa ki az üzenetet, a függvény hibát ad, és az `errno` változó értéke `E2BIG` lesz.

Az `msgtype` paraméter határozza meg a kiolvasandó üzenet típusát. 3 eset lehetséges:

- `msgtype = 0` – a sor első üzenetét olvassa ki,
- `msgtype > 0` – a sor első `msgtype` típusú üzenetét olvassa ki,
- `msgtype < 0` – a sor legkisebb, de az `abs(msgtype)`-nál nagyobb vagy azzal egyenlő értékű üzenetét olvassa ki.

Az `flg` argumentummal megadhatjuk, hogy a rendszer hogyan viselkedjen, ha a függvény meghívásának feltételei nem teljesülnek. Az előbb bemutatottuk az `MSG_NOERROR` használatát. Következzen az `IPC_NOWAIT`-é. Szintén két eset lehetséges:

- Ha az `IPC_NOWAIT` nincs beállítva, a folyamat nem várja meg az óhajtott üzenetnek a sorban való elhelyezését. A függvény hibát ad, és az `errno` értéke `ENMSG` lesz.
- Ha az `IPC_NOWAIT` nincs beállítva, a folyamat addig várakozik, ameddig kerül egy megfelelő üzenet, ameddig a sor megsemmisül (`errno=EIDRM`), vagy egy jelzés érkezik (`errno=EINTR`).

A következő programrészlet példa egy üzenet kiolvasására:

```
#define KEY 2003
struct msgbuf uzenet;
```

```
msgid = msgget((key_t) KEY, 0);
msgrcv(msgid, &uzenet, 25, long(100), IPC_NOWAIT | MSG_NOERROR);
```

Ha az üzenetsorba bekerülő adatok típusai megegyeznek, akkor az üzenetsorok tulajdonképpen FIFO állományként működnek.

## ☰ Példa

Hozzunk létre egy kliens-szerver rendszert! A kliensfolyamat a parancssorból beolvas egy tevékenységekódot (pl. kliens 2), majd ezt egy üzenetsoron keresztül elküldi a szervernek. A visszakapott választ a képernyőre írja. A szerverfolyamat kiolvassa az üzenetsorból a tevékenységekódot, elvégzi a tevékenységet, majd jelentést küld a kliensnek.

Az egyszerűség kedvéért a közös adatokat (így az üzenet struktúráját is) egy külön fejléc állományban tároljuk, amelyet mind a kliens, mind a szerver el tud majd érni.

### mes.h

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define KEY 13                                /* az üzenetsor kulcsa */

#define READ 1                                /* tevékenységekódot elnevezései */
#define WRITE 2

struct                                          /* az üzenet struktúrája */
{
    long mtip;                                /* üzenet típusa */
    int pid;                                  /* küldő folyamat azonosítója */
    int cod_op;                                /* kliens üzenete; tevékenységekódot */
    char mtext[13];                           /* szerver üzenete; tev. elnevezése */
} MESSAGE;
```

A kliens első lépésben a parancssorból beolvas egy tevékenységekódot. Egyetlen üzenetsort használunk, amelyen keresztül minden kliens elküldheti a kéréseit a szerver felé, és amelyen a válasz is érkezik. Azért, hogy egy kliens megkapja a neki címzett választ, csak azon üzeneteket fogja kiolvasni a sorból, amelyek típusa éppen megegyezik a saját pid-jével. Éppen ezért a kéréssel egyidőben minden kliensnek el kell küldenie az azonosítóját is.

### kliens.c

```
#include "mes.h"                                /* a közös hdr állomány */
#include "<u>hdr.h</u>"

int msgid;                                    /* üzenetsor azonosítója */

void main(int argc, char **argv)
{
    pid_t pid;                                /* folyamatazonosító */
    MESSAGE mesp;                              /* üzenet */

    if (argc != 2)                            /* hibás argumentumok */
```

```

    err_quit("hasznalat: c <cop_op>");

    if ((msgid = msgget((key_t) KEY, 0666)) < 0)
        err_sys("msgget hiba");                                /* üzenetsor ID-jának lekérdezése */

    mesp.mtip = 1;                                              /* a kérés típusa mindig 1 */
    mesp.cod_op = atoi(argv[1]);                                /* tevékenység kódja parancssorból */
    pid = mesp.pid = getpid();
    printf("a %d kliens elküldte a kerest\n", mesp.pid);

    if (msgsnd(msgid, (struct msghdr *)&mesp, sizeof(mesp)-sizeof(long), 0) < 0)
        err_quit("msgsnd hiba");                                /* tev.kód elküldése a szervernek */

    if (msgrcv(msgid, (struct msghdr *)&mesp, sizeof(mesp)-sizeof(long), pid, 0) < 0)
        err_sys("msgrcv hiba");                                /* tev.név fogadása a szervertől */

    mesp.mtext[12] = '\0';                                     /* eredmény előkészítése, kiíratás */

    printf("a %d szerver a %d kliensnek %s kerest küldött\n", mesp.pid, pid, mesp.mtext);
}

```

A szerverfolyamat kap a kientől egy 1-es típusú üzenetet, majd egy olyan választ készít, amelynek típusa megegyezik a kérést küldő kient pid-jével. Az alábbi példában a szerver által felkínált szolgáltatások a következők: READ és WRITE. Ezek végrehajtását a szerver csak szimulálja. A választ az mtext mező fogja tartalmazni, amely megmutatja, hogy a szervernek sikerült-e azonosítani a tevékenységet vagy sem.

#### szerver.c

```

#include "mes.h"                                                /* a közös hdr állomány */
#include "hdr.h"

int msgid;                                                      /* üzenetsor azonosítója */

void do_it(MESSAGE *mesp)                                       /* tev.kód -> tev.elnevezés */
{
    switch (mesp->cod_op)                                        /* a kliens által küldött kód */
    {
        case READ:                                             /* READ = 1 tev. elnevezése */
            strcpy(mesp->mtext, "READ\n");
            break;
        case WRITE:                                             /* WRITE = 2 tev. elnevezése */
            strcpy(mesp->mtext, "WRITE\n");
            break;
        default:                                                /* ismeretlen tev.kód */
            strcpy(mesp->mtext, "Ismeretlen\n");
            break;
    }
}

void main(void)
{
    MESSAGE mesp;                                                /* üzenet */
}

```



```

if ((msgid = msgget((key_t) KEY, IPC_CREAT)) < 0)
    err_sys("msgget hiba");                                /* üzenetsor létrehozása */

for ever                                                    /* végtelenciklus */
{
    if (msgrcv(msgid, (struct msgbuf *)&mesp, sizeof(mesp)-sizeof(long), 1L, 0) < 0)
        err_sys("msgrcv hiba");                            /* tev.kód kiolvasása a sorból */

    do_it(&mesp);                                           /* átalakítás */

    mesp.mtip = mesp.pid;                                   /* kliens kódja = üzenet típusa */
    mesp.pid = getpid();

    if (msgsnd(msgid, (struct msgbuf *)&mesp, sizeof(mesp)-sizeof(long), 0) < 0)
        err_sys("msgsnd hiba");                            /* tev. elnevezés küldése */
}
}

```

A fenti példa egy tesztesetre a következőképpen működik:

```

$ szerver &
$ kliens 1 & kliens 4

a 995 kliens elküldte a kerest
a 768 szerver a 995 kliensnek READ kerest küldött

a 999 kliens elküldte a kerest
a 768 szerver a 999 kliensnek ismételten kerest küldött

```



## Szemaforok (semaphores)

Copyright (C) Buzogány László, 2002



[About](#)



# Szemaforok (semaphores)

Olyan jelek, amelyek megmutatják, hogy egy folyamat végrehajthat-e egy programrészt vagy sem...

## Definíció

## Műveletek szemaforokkal

↳ Létrehozás (`semget`)

↳ Szemafor adatainak lekérdezése, módosítása és törlése (`semctl`)

↳ Szemafor értékének növelése és csökkentése (`semop`)

## Példák

## Üzenetsorok (message queues)

## Definíció

A processzor maximális kihasználása érdekében az operációs rendszerek jelentős része – így a Unix is – engedélyezi a folyamatoknak a párhuzamos (egyidőben történő) futást. Ez a megoldás igen hatékony, de mi történik akkor, ha két folyamat egyidőben próbálja elérni ugyanazt az erőforrást? Ilyen "érzékeny" erőforrások például a nyomtató vagy a memória, amelyet egyszerre csak egy folyamat használhat, különben a műveletnek előre nem látható következményei lehetnek.

Kritikus szakasznak nevezzük tehát egy olyan módosítást a rendszeren, amelyet nem szabad megszakítani. A kritikus szakasz erőforráshoz kötött (például a nyomtatóhoz).

A szemaforok lehetővé teszik a felhasználók számára a folyamatok szinkronizálását. Általában a szemafor egy egész változó, amelyhez hozzárendelünk egy  $e_0(v)$  kezdeti értéket, egy  $e(v)$  aktuális értéket és egy  $e_0(v)$  várakozási sort.

Az alábbiakban bemutatjuk a legismertebb szinkronizációs P és V eljárásokat.

```
P(v)
{
    e(v) = e(v) - 1;
    if (e(v) < 0)
    {
        folyamat_allapota = WAIT;
        f(v) sorba <- folyamat;
    }
}
```

```

V(v)
{
    e(v) = e(v) + 1;
    if (e(v) <= 0)
    {
        folyamat <- f(v) sorból;
        kivlasztott_folyamat_allapota = READY;
    }
}

```

A szemafor aktuális állapotát a következő képlet adja:

$$e(v) = e_0(v) + n \cdot V(v) - n \cdot P(v)$$

ahol:

$n \cdot V(v)$  – a  $v$  szemaforon végrehajtott  $V$  eljárások száma,  
 $n \cdot P(v)$  – a  $v$  szemaforon végrehajtott  $P$  eljárások száma.

Tehát egy kritikus szakasz megvalósítása a következőképpen történik: a szemafor kezdeti értéke 1 lesz, a kritikus szakaszt pedig a  $P$  és  $V$  eljárások határolják körül.

```

P(v)
kritikus szakasz
V(v)
folyamat többi része

```

Megjegyezzük, hogy a  $P$  és  $V$  eljárás és áttatában a szinkronizáló eljárások feloszthatatlanok (atomi műveletek).

A UNIX System V verzióban a szemafor fogalmát általánosították. Ezért egy szemafor esetében egyidőben akár több műveletet is megadhatunk (a  $P$  és  $V$  eljárások más-más időben hívódnak meg a folyamaton belül), és az  $e(v)$  értékének növelése vagy csökkentése nem feltétlenül 1-el kell történjen. Minden a rendszer által végrehajtott művelet feloszthatatlan. Tehát ha a rendszer nem tudja elvégezni a folyamat által kért összes műveletet, nem végzi el egyiket sem, és a folyamat várakozási állapotba kerül egészen az összes művelet végrehajtásáig.

Egy folyamat létrehozhat egy egész szemaforköteget. A kötegnek van egy bemenete a szemafortáblában, amely a szemaforköteg fejlécét tartalmazza. A folyamathoz rendelt táblázat pontosan annyi elemből áll, amennyi szemafor tartozik a köteghez. Minden elem megőrzi az illető szemaforhoz rendelt értéket.

A szemaforok kezelése nagyon hasonlít az osztott memória és az üzenetsorok kezeléséhez. Ezért egy folyamatnak csak akkor van joga hozzáférni egy szemaforhoz, ha ismeri a hozzárendelt kulcsot. Belsőleg a szemaforköteget egy egész számmal azonosítjuk, ezért a folyamat bármelyik szemaforhoz hozzáférhet

Tehát minden szemaforkötegnek van egy azonosítója (amely egy pozitív egész szám) és egy `semid_ds` típusú adatszerkezete.

```

struct semid_ds
{

```

```

struct ipc_perm sem_perm; /* definiálja a jogokat és a tulajdonost */
struct sem *sem_base;     /* pointer az első szemaforra a kötegből */
int sem_nsens;            /* a kötegben található szemaforok száma */
time_t sem_otime;        /* utolsó semop() művelet ideje */
time_t sem_ctime;        /* utolsó struktúramódosítás ideje */
};

```

Egy folyamat csak akkor férhet hozzá egy szemaforköteghoz ha:

- a folyamat a superuser-é,
- a felhasználó ID-je (uid) megegyezik az sem\_perm.cuid-vel vagy az sem\_perm.uid-vel és az sem\_perm.mode tartalmazza a kívánt jogokat,
- a felhasználó csoportazonosítója (gid) megegyezik az sem\_perm.cgid vagy sem\_perm.gid értékek egyikével és az sem\_perm.mode tartalmazza a kívánt jogokat,
- a felhasználó beleesik a "többi felhasználó" kategóriába és az sem\_perm.mode tartalmazza a megfelelő jogokat.

Egy szemaforhoz rendelt adatszerkezet a következő:

```

struct sem
{
    ushort semval; /* a szemafor értéke (semval>=0) */
    pid_t sempid; /* utolsó, műveletet végző folyamat ID-ja */
    ushort semncnt; /* azon foly. száma, amelyek a semval növekedését várják */
    ushort semzcnt; /* azon foly. száma, amelyek a semval=0-t várják */
};

```

## ☰ Műveletek szemaforokkal

### ☰ Létrehozás (semget)

A **semget** rendszerfüggvény lehetővé teszi egy szemaforköteg ID-jának a meghatározását. Ha a szemaforköteg előzőleg nem létezett, a függvény létrehozza azt. Függetlenül attól, hogy a köteg létezett-e vagy sem a folyamatnak ismernie kell a szemaforhoz rendelt kulcsot. A függvény alakja:

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nrsem, int flg);

```

A függvény visszatérési értéke az újonnan létrehozott vagy egy régi szemafor azonosítója, illetve -1 hiba esetén. A **key** változóban a szemaforhoz rendelt kulcsot kell megadni, míg az **flg**-ben a létrehozási tevékenységet és a hozzáférési jogokat. A kulcsot a szemafor használó összes folyamatnak ismernie kell. Az **nrsem** argumentum a kötegben található szemaforok számát jelenti. Ha egy új köteget hozunk létre meg kell adnunk a változó értékét, ha viszont egy már létező kötegre hivatkozunk az **nrsem** értéke 0 lesz.

Egy új szemaforköteg létrehozása esetén az **flg** mezőt a következő formában kell megadni:

## IPC\_CREAT | IPC\_EXCL | hozzáférési\_jogok

Ha az IPC\_EXCL nincs beállítva, akkor ha már létezik egy előzőleg létrehozott köteg a megadott kulcsra, a program nem jelez hibát, hanem visszaadja a létező köteg ID-ját.

Példa egy szemaforköteg létrehozására:

```
#define KEY 2003

int semid;

semid = semget((key_t) KEY, 5, IPC_CREAT | 0666);
```

Ha csak a szemaforköteg azonosítójára vagyunk kíváncsiak, a semget hívásakor adjuk meg a kulcsot, a többi paraméternek pedig 0 értéket.

```
semid = semget((key_t) KEY, 0, 0);
```

Létrehozáskor a társított adatstruktúra (sem\_perm) mezői a következő információkkal töltődnek fel:

- sem\_perm.cuid, sem\_perm.uid – a semget függvényt meghívó folyamathoz hozzárendelt felhasználó ID-ja,
- sem\_perm.cgid, sem\_perm.ugid – a semget függvényt meghívó folyamathoz hozzárendelt felhasználócsoporthoz hozzárendelt csoport ID-ja,
- sem\_perm.mode – a semget függvény hívásakor megadott flg argumentum, amely a hozzáférési jogokat tartalmazza,
- sem\_nsems – a semget függvény hívásakor megadott szemaforok száma,
- sem\_ctime – az aktuális időt tartalmazza,
- sem\_otime – értéke 0.

Előfordulhat, hogy egy szemaforköteghez nincs hozzárendelt kulcs. Ebben az esetben a köteg létrehozásakor a semget függvény key paramétereként az IPC\_PRIVATE szimbolikus konstanst kell megadni.

## ☰ Szemafor adatainak lekérdezése, módosítása és törlése (semctl)

A semctl függvény a szemaforkötegek szintjén az információk lekérdezésére, módosítására és törlésére használható. Szintaxisa:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int nrsem, int cmd, union semun arg);
```

A függvény visszatérített értéke az összes GET parancs kimeneti értéke, kivételt képez a GETALL parancs. Minden más esetben a függvény értéke 0.

A semid paraméter a semget függvény által meghatározott szemaforköteg azonosítója, míg az nrsem annak a szemafornak a száma, amelyen végre szeretnénk hajtani a cmd műveletet.

A cmd argumentum tehát a kívánt műveletet határozza meg és a következő értékeket veheti fel:

- GETVAL – a függvény visszatéríti a semval értékét az nrsem által meghatározott szemaforban,
- SETVAL – az nrsem által meghatározott szemafor semval paraméterének értéke arg.val lesz,

- GETPID – a függvény visszatéríti a sempid értékét az nrsem által meghatározott szemaforban,
- GETNCNT – a függvény visszatéríti a semncnt értékét az nrsem által meghatározott szemaforban,
- GETZCNT – a függvény visszatéríti a semzcnt értékét az nrsem által meghatározott szemaforban,
- GETALL – minden szemafor semval értékét elhelyezi az arg.array tömbben,
- SETALL – minden szemafor semval értékét feltölti az arg.array tömbben megadott értékekkel,
- IPC\_STAT – a semid\_ds struktúra elemeit lementi az arg.buf tömbbe,
- IPC\_SET – a sem\_perm.uid, a sem\_perm.gid és a sem\_perm.mode mezők értékeit frissíti az arg.buf tömbben megadott értékekkel,
- IPC\_RMID – a szemaforköteg törlése; a törlés azonnali, tehát minden ezt a szemafor használó folyamat egy EIDRM üzenetet kap, és hibával leáll.

Az arg variáns típusú változó, amely a cmd parancs által felhasznált argumentumokat tartalmazza:

```
union semun
{
    int val;                /* a SETVAL-hoz */
    struct semid_ds *buf;   /* az IPC_STAT-hoz és az IPC_SET-hez */
    ushort *array;         /* a GETALL-hoz és a SETALL-hoz */
}
```

### ☰ Szemafor értékének növelése és csökkentése (semop)

A semop függvény feladata egy szemaforköteg egy elemének növelése és csökkentése. Szintaxisa:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf op[], size_t nrop);
```

A függvény visszatérési értéke 0, ha a művelet sikeres volt, ellenkező esetben -1. A semid a semget által meghatározott szemaforköteg azonosítója.

Az op egy mutató nrop darab sembuf típusú struktúrához, ahol:

```
struct sembuf
{
    ushort sem_num; /* a kötegben levő szemaforok száma */
    short sem_op;   /* végrehajtandó művelet */
    short sem_flg;  /* végrehajtási körülmények */
}
```

Tehát minden szemafor esetén a végrehajtandó műveletet a sem\_op mező jelzi. Ha:

- sem\_op < 0

a. Ha semvan >= |sem\_op|, akkor semval = semval - |sem\_op|. Ez biztosítja, hogy a szemafor által visszatérített érték >= 0.

b. Ha (semval < |sem\_op|) & (sem\_flg & IPC\_NOWAIT) = true, akkor a függvény egy hibakódot térít vissza.

c. Ha semval < |sem\_op| és az IPC\_NOWAIT nincs megadva, akkor a folyamat leáll, amíg a

következők közül valamely feltétel nem teljesül:

- `semval >= |sem_op|` (egy másik folyamat felszabadít pár erőforrást),
- a szemafor törlődik a rendszerből; a függvény ebben az esetben -1 értéket térít vissza és `errno=ERMID`,
- ha egy folyamat egy jel lekezelése után visszatér, a `semcnt` értéke a szemaforban csökken, és a függvény -1-et ad vissza (`errno=EINTR`).

- `sem_op > 0`

Ekkor `semval = semval + sem_op`.

- `sem_op = 0`

a. Ha a `semval = 0`, akkor a függvény azonnal befejeződik.

b. Ha a `semval ≠ 0` és az `IPC_NOWAIT` be van állítva, akkor a függvény -1-et ad vissza és az `errno=EAGAIN`.

c. Ha a `semval ≠ 0` és az `IPC_NOWAIT` nincs beállítva, akkor a folyamat leáll, ameddig a `semval` 0 nem lesz, vagy a szemaforköteg megsemmisül, vagy a folyamat egy jelet kap.

## ≡ Példák

A következőkben megnézzük, hogyan lehet implementálni a P és V eljárásokat:

`pv.c`

```
#include "hdr.h"

static void semcall(int semid, int op)
{
    struct sembuf pbuf;

    pbuf.sem_num = 0;
    pbuf.sem_op = op;
    pbuf.sem_flg = 0;

    if (semop(semid, &pbuf, 1) < 0)
        err_sys("semop hiba");
}

void P(int semid)
{
    semcall(semid, -1);
}

void V(int semid)
{
    semcall(semid, 1);
}
```

Azért, hogy a P és a V eljárások működését jobban megértsük tekintsük a következő példát.

Írjunk folyamatot, amely három gyereket hoz létre. Mindenik gyerek hozzá szeretne férni egy közös

erőforráshoz. A kritikus szakasz 10 másodpercig tart. (A valóságban a kritikus szakasznak sokkal rövidebbnek kell lennie.) A szemafor biztosítja minden folyamat egyéni hozzáférését a kritikus szakaszban. A tesztprogram a következő:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include "pv.c"                /* a fent definiált P és V eljárások */

#define SEMPERM 0600           /* hozzáférési jogok */

void rut_sem(int semid);
int initsem(key_t semkey);

void main(void)
{
    key_t semkey = 0x200;      /* kulcs */
    int semid, i;
    semid = initsem(semkey);    /* szemafor létrehozása */

    for (i=0; i<3; i++)        /* 3 darab gyerekfolyamat létrehozása */
        if (fork() == 0)
            rut_sem(semid);     /* erőforrási kérelmek a gyerekfolyamatoktól */
}

void rut_sem(int semid)        /* szemaforok megvalósítása */
{
    pid_t pid;
    pid = getpid();            /* gyerekfolyamat PID-je */

    P(semid);                  /* belépés a kritikus szakaszba */
    printf("a %d folyamat kritikus szakaszban van\n", pid);

    sleep(random()%5);         /* kritikus szakasz; várakozás */

    printf("a %d folyamat elhagyja a kritikus szakaszt\n", pid);
    V(semid);                  /* kilépés a kritikus szakaszból */

    exit(0);                   /* gyerekfolyamat vége */
}

int initsem(key_t semkey)
{
    int semid;                 /* szemafor létrehozása */

    semid = semget(semkey, 1, SEMPERM | IPC_CREAT);

    if (semctl(semid, 0, SETVAL, 1) < 0)
        err_sys("semctl hiba"); /* szemaforok száma = 1 */

    return semid;              /* szemafor ID-jának visszatérítése */
}
```

A következő tesztesetek is bizonyítják, hogy egy bizonyos időpillanatban csak egyetlen folyamat lehet a



kritikus szakaszban.

```
$ a.out
```

```
a 790 folyamat kritikus szakaszban van
```

```
a 790 folyamat elhagyja a kritikus szakaszt
```

```
a 791 folyamat kritikus szakaszban van
```

```
a 791 folyamat elhagyja a kritikus szakaszt
```

```
a 792 folyamat kritikus szakaszban van
```

```
a 792 folyamat elhagyja a kritikus szakaszt
```

➡ [Osztott memória \(shared memory\)](#)

---

Copyright (C) Buzogány László, 2002



[About](#)



# Osztott memória (shared memory)

Alkalmazása esetén ugyanazt a memóriarészt használja az összes összeköttetésben levő folyamat...

## Definíció

## Műveletek az osztott memóriával

- ↳ Létrehozás (shmget)
- ↳ Az osztott memória adatainak lekérdezése, módosítása és törlése (shmctl)
- ↳ Memóriarész hozzárendelése (shmat)
- ↳ Memóriarész hozzárendelésének megszüntetése (shmdt)

## Példa

## Szemaforok (semaphores)

## Definíció

Az osztott vagy közös memória segítségével megoldható, hogy két vagy több folyamat ugyanazt a memóriarészt használja. Az osztott memóriazónák általi kommunikáció elvei:

- Egy folyamat létrehoz egy közös memóriazónát. A folyamat azonosítója bekerül a memóriazónához rendelt struktúrába.
- A létrehozó folyamat hozzárendel az osztott memóriához egy numerikus kulcsot, amelyet minden ezt a memóriarészt használni kívánó folyamatnak ismernie kell. Ezt a memóriazónát az `shmid` változó azonosítja.
- A létrehozó folyamat leszögezi a többi folyamat hozzáférési jogait az illető zónához. Azért, hogy egy folyamat (beleértve a létrehozó folyamatot is) írni és olvasni tudjon a közös memóriarészből, hozzá kell rendelnie egy virtuális címterületet.

Ez a kommunikáció a leggyorsabb, hiszen az adatokat nem kell mozgatni a kliens és a szerver között.

A folyamatoknak a közös memóriarészhez való hozzáférését nem a rendszer felügyeli, a konfliktusok kezelése a felhasználó folyamatok feladata. Ha a szerver adatokat helyez el a közös memóriarészben, akkor a kliensnek várakoznia kell egészen a művelet befejezéséig, s csak akkor férhet hozzá az illető adatokhoz. A hozzáférések összehangolására gyakran használunk szemaforokat.

A rendszer minden egyes közös memóriarész esetén a következő adatokat tárolja:

```
struct shmid_ds
```

```

{
    struct ipc_perm shm_perm; /* definiálja a jogokat és a tulajdonost */
    struct anon_map *shm_amp; /* pointer a rendszerben */
    int shm_segz; /* szegmens mérete bájtokban */
    pid_t shm_cpid; /* létrehozó folyamat pid-je */
    pid_t shm_lpid; /* utolsó shmop() pid-je */
    ulong shm_nattach; /* eddig kapcsolódott folyamatok száma */
    ulong shm_cnattach; /* csak az shminfo használja */
    time_t shm_atime; /* utolsó beírás ideje */
    time_t shm_dtime; /* utolsó kiolvasás ideje */
    time_t shm_ctime; /* utolsó struktúramódosítás ideje */
}

```

Egy folyamat csak akkor férhet hozzá a közös memóriarészhez ha:

- a folyamat a superuser-é,
- a felhasználó ID-je (uid) megegyezik az shm\_perm.cuid-vel vagy az shm\_perm.uid-vel és az shm\_perm.mode tartalmazza a kívánt jogokat,
- a felhasználó csoportazonosítója (gid) megegyezik az shm\_perm.cgid vagy shm\_perm.gid értékek egyikével és az shm\_perm.mode tartalmazza a kívánt jogokat,
- a felhasználó beleesik a "többi felhasználó" kategóriába és az shm\_perm.mode tartalmazza a megfelelő jogokat.

## === Műveletek az osztott memóriával

### === Létrehozás (shmget)

Az shmget rendszerfüggvény engedélyezi a közös memória azonosítójának lekérdezését felhasználva egy bizonyos kulcsot. Alakja:

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int flg);

```

A függvény visszatérési értéke a key kulcshoz rendelt memóriazóna azonosítója, illetve -1 hiba esetén. A size változóban a közös memória méretét kell megadni, míg az flg-ben a létrehozási tevékenységet és a hozzáférési jogokat. A kulcsot a közös memóriát használó összes folyamatnak kell ismernie.

Egy új memóriaterület létrehozása esetén az flg mezőt a következő formában kell megadni:

IPC\_CREAT | IPC\_EXCL | hozzáférési\_jogok

Ha az IPC\_CREAT opció nincs beállítva, és már létezik egy előzőleg létrehozott memóriazóna, a függvény ennek az azonosítóját téríti vissza.

Példaként hozzunk létre egy 2003 kulccsal rendelkező 200 bájt méretű memóriarészt.

```
#define KEY 2003
```

```
int shmid;
shmid = shmget((key_t) KEY, 200, IPC_CREAT | 0666);
```

Ha egy már létező osztott memóriarész azonosítóját (ID-ját) szeretnénk meghatározni, akkor a size és az flg mezőbe 0-t írunk – ez esetben a függvény nem fog létrehozni új sort. Például az előzőleg létrehozott memóriazóna esetén a következőképpen járunk el:

```
shmid = shmget((key_t) KEY, 0, 0);
```

Létrehozáskor a társított adatstruktúra (shm\_perm) mezői a következő információkkal töltődnek fel:

- shm\_perm.cuid, shm\_perm.uid – az shmget függvényt meghívó folyamathoz hozzárendelt felhasználó ID-ja,
- shm\_perm.cgid, shm\_perm.uid – az shmget függvényt meghívó folyamathoz hozzárendelt felhasználócsoporthoz hozzárendelt ID-ja,
- shm\_perm.mode – az shmget függvény hívásakor megadott flg argumentum, amely a hozzáférési jogokat tartalmazza,
- shm\_qnum, shm\_lspid, shm\_lrpid, shm\_stime, shm\_rtime – értéke 0,
- shm\_ctime – az aktuális időt tartalmazza,
- shm\_segz – az shmget függvény hívásakor megadott size argumentum.

Előfordulhat, hogy egy osztott memóriazónának nincs hozzárendelt kulcsa. Ebben az esetben a key paraméternek az IPC\_PRIVATE értéket adjuk.

## ☰ Az osztott memória adatainak lekérdezése, módosítása és törlése (shmctl)

Az shmctl függvény az osztott memóriarész információinak lekérdezésére, módosítására és törlésére használható. Szintaxisa:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

A függvény visszatérési értéke 0, ha a művelet sikeres volt, ellenkező esetben -1. Az shmid paraméter az shmget függvény által meghatározott osztott memóriarész azonosítója.

A cmd argumentum a kívánt műveletet határozza meg és a következő értékeket veheti fel:

- IPC\_STAT – az osztott memóriához rendelt struktúra tartalma a buf változóba kerül,
- IPC\_SET – az osztott memóriához rendelt struktúrát frissíti a buf által megadott struktúrával,
- IPC\_RMID – az osztott memóriarész elméletileg törlődik; a tulajdonképpeni törlésre csak akkor kerül sor, amikor az utolsó folyamat is, amely ezt a zónát használja, megszakítja a kapcsolatát ezzel a memóriarésszel; függetlenül attól, hogy ez a rész éppen használat alatt van-e vagy sem az ID törlődik, s ezáltal ez a memóriarész többet nem osztható ki egyetlen folyamat számára sem; ebben az esetben a buf argumentumnak a NULL értéket kell adni,
- SHM\_LOCK – megtiltja a hozzáférést a közös memóriarészhez,
- SHM\_UNLOCK – engedélyezi a hozzáférést a közös memóriarészhez.

Példa:

```
#define KEY 2003
```

```
shmid = shmget((key_t) KEY, 0, 0);
shmctl(shmid, IPC_RMID, NULL);
```

### Memóriarész hozzárendelése (shmat)

Az `shmat` függvény feladata egy folyamat címterületéhez hozzárendelni egy osztott memóriazónát. A hozzárendelés után a folyamat írhat, illetve olvashat erről a memóriarésről. A függvény szintaxisa:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void *shmat(int shmid, void *addr, int flg);
```

A függvény visszatérített értéke egy pointer a közös memóriazónára, ha a művelet sikeres volt, ellenkező esetben -1. Az `shmid` paraméter az `shmget` által meghatározott osztott memória azonosítója.

A `addr` pointer típusú változó a közös memóriarész hozzáférési címe a hívó folyamat adatszegmensében. Ezért ha:

- ha az `addr`  $\neq$  NULL, a következő esetek fordulnak elő:
  - a. ha az SHM\_RND opció be van állítva, a hozzárendelés az `addr` címhez történik,
  - b. ha az SHM\_RND nincs beállítva, a hozzárendelés az  $(addr - (addr \bmod SHMLBA))$  címhez történik,
- ha az `addr` = NULL, a memóriarész a rendszer által kiválasztott első szabad címhez történik (ajánlott).

Az `flg` paraméter meghatározza a hozzárendelt memória megadási módját (SHM\_RND) és a közös részhez való hozzáférést, tehát hogy írásvédett (SHM\_RDONLY) vagy sem.

A következő példa bemutatja, hogyan lehet írni egy közös memóriazónára:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

#define KEY 2003

void main(void)
{
    int shmid;
    char *p;
    ...
    shmid = shmget((key_t) KEY, 0, 0);
    p = shmat(shmid, NULL, 0);
    strcpy(p, "proba");
    ...
    exit(0);
}
```

## ☰ Memóriarész hozzárendelésének megszüntetése (shmdt)

Az shmdt függvény feladata a hívó folyamat címterületéhez hozzárendelt osztott memóriazóna felszabadása. Megjegyezzük, hogy a memóriarészhez hozzárendelt struktúra és az ID nem törlődik a rendszerből, míg egy folyamat (általában a szerver) az shmctl függvényhívással (IPC\_RMID) azt végérvényesen nem törli. Alakja:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt(void *addr);
```

A következő programrészlet bemutatja, hogyan lehet kiolvasni adatokat egy olyan közös memóriazónából, ahová előzőleg egy másik folyamat írt. A végén a memóriaterületet felszabadítjuk és töröljük.

```
#include <stdio.h>
#include <sys/type.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define KEY 2003

void main(void)
{
    int shmid;
    char *p;
    ...
    shmid = shmget((key_t) KEY, 0, 0);
    p = shmat(shmid, NULL, 0);
    printf("a kozos memoria tartalma: %s\n", p);
    shmdt(p);
    shmctl(shmid, IPC_RMID, NULL);
    exit(0);
}
```

## ☰ Példa

Készítsünk olyan programot, amely létrehozza, olvassa, írja és törli az osztott memóriát! A műveletet a parancssoron keresztül fogjuk megadni. Amennyiben egy művelet kiadásakor a közös memória nem létezik a program automatikusan hozza létre azt!

A folyamat tehát a következő műveleteket tudja elvégezni:

- írás a memóriazónába: **shmtool w "text"**
- a memóriazónán található szöveg kiolvasása: **shmtool r**
- a hozzáférési jogok módosítása (mode): **shmtool m (mode)**
- memóriarész törlése: **shmtool r**

A forráskód tartalmazza a főprogramot és a műveleteket elvégző segédeljárásokat. (Az ftok utasítással a folyamat számára egyedi kulcsot hozunk létre.)

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SEGSIZE 100                                /* a tárolandó szöveg max. mérete */

void writeshm(int shmid, char *segptr, char *text);
void readshm(int shmid, char *segptr);
void removeshm(int shmid);                        /* függvények deklarálása */
void changemode(int shmid, char *mode);
void usage(void);

int main(int argc, char *argv[])                  /* parancssorból a paraméterek */
{
    key_t key;                                    /* kulcs */
    int shmid;                                    /* osztott memória ID-ja */
    char *segptr;                                 /* osztott memória címe */

    if (argc == 1)                                /* hiányos paraméterlista */
        usage();

    key = ftok(".", 'S');                          /* egyedi kulcs létrehozása */

                                                    /* megnyitás, szükség esetén létrehozás */
    /*

    if ((shmid = shmget(key, SEGSIZE, IPC_CREAT|IPC_EXCL|0666)) == -1)
    {
        printf("az osztott memóriazona letezik - megnyitom\n");

        if ((shmid = shmget(key, SEGSIZE, 0)) == -1)
        {
            perror("shmget hiba");
            exit(1);
        }
    }
    else
        printf("letrehozok egy új osztott memóriazonat\n");

                                                    /* memóriacím hozzárendelése */

    if ((segptr = shmat(shmid, 0, 0)) == (void *) -1)
    {
        perror("shmat hiba");
        exit(1);
    }

    switch (tolower(argv[1][0]))                    /* a param. alapján elvégzi a műv. */
    {
        case 'w':                                    /* megadott szöveg írása */
            writeshm(shmid, segptr, argv[2]);
            break;

```

```

    case 'r':                                /* osztott memóriazóna kiolvasása */
        readshm(shmid, segptr);
        break;
    case 'd':                                /* törlés */
        removeshm(shmid);
        break;
    case 'm':                                /* jogok módosítása */
        changemode(shmid, argv[2]);
        break;
    default:                                 /* hibás opció */
        usage();
}
}

void writeshm(int shmid, char *segptr, char *text)
{
    strcpy(segptr, text);                   /* szöveg beírása a memóriába */
    printf("kész...\n");
}

void readshm(int shmid, char *segptr)
{
    printf("segptr: %s\n", segptr);         /* mem. tartalmának kiolvasása */
}

void removeshm(int shmid)
{
    shmctl(shmid, IPC_RMID, 0);              /* osztott memória törlése */
    printf("torolve\n");
}

void changemode(int shmid, char *mode)
{
    struct shmid_ds myshmds;
    shmctl(shmid, IPC_STAT, &myshmds);     /* aktuális jogok lekérdezése */
    printf("a regi jogok: %o\n", myshmds.shm_perm.mode);
    sscanf(mode, "%o", &myshmds.shm_perm.mode);
    shmctl(shmid, IPC_SET, &myshmds);      /* új jogok beállítása */
    printf("az uj jogok: %o\n", myshmds.shm_perm.mode);
}

void usage(void)                            /* használati útmutató */
{
    printf("shmtool - osztott memoria menedzselo rendszer\n\n");
    printf("HASZNALAT: shmtool (w)rite <szoveg>\n");
    printf("                    (r)ead\n");
    printf("                    (d)eleete\n");
    printf("                    (m)ode change <oktalis_mod>\n");
    exit(1);
}

```



A fenti példa tesztelésekor a következő eredményre jutottunk:

```
$ shmtool w teszt
letrehozok egy uj osztott memoriazonat
kész...

$ shmtool r
az osztott memoriazona letezik - megnyitom
segptr: teszt

$ shmtool w szasz
letrehozok egy uj osztott memoriazonat
kész...

$ shmtool r
az osztott memoriazona letezik - megnyitom
segptr: szasz

$ shmtool d
az osztott memoriazona letezik - megnyitom
torolve

$ shmtool m 660
letrehozok egy uj osztott memoriazonat
a regi jogok: 666
az uj jogok: 660

$
```

---

Copyright (C) Buzogány László, 2002



[About](#)



# Függelék

Különböző összefoglaló táblázatok, rövid utasításleírások és egyéb hasznos segítség...

## ☰ Hibakódok

| Szimbólum | Hibaüzenet<br>(Jelentés)  |
|-----------|---|
| EPERM     | <b>NOT OWNER</b><br>(Fájlmegnyitási próbálkozás egy olyan felhasználónak, aki nem az állomány tulajdonosa)  |
| ENOENT    | <b>NO SUCH FILE OR DIRECTORY</b><br>(A megadott nevű fájl vagy könyvtár nem létezik)  |
| ESRCH     | <b>NO SUCH PROCESS</b><br>(A megadott azonosítójú folyamat nem létezik)   |
| EINTR     | <b>INTERRUPTED SYSTEM CALL</b><br>(A függvény végrehajtása során egy jelzés érkezett)   |
| EIDRM     | <b>REMOVE IDENTIFIER</b><br>(Nem megengedett művelet az adott hálózaton)  |
| EIO       | <b>I/O ERROR</b><br>(Ki/bemeneti hiba az előzetes kérés)  |
| ENXIO     | <b>NO SUCH DEVICE OR ADDRESS</b><br>(Ki/bemeneti hiba: a megadott periféria nincs telepítve)  |
| E2BIG     | <b>ARG LIST TO LONG</b><br>(Az exec függvény argumentumainak listája túl nagy)  |
| ENOEXEC   | <b>EXEC FORMAT ERROR</b><br>(Az exec-ben megadott állomány hibás, például nem futtatható)   |
| EBADF     | <b>BAD FILE NUMBER</b><br>(Olvasási próbálkozás egy csak írásra megnyitott fájlból vagy fordítva (írás egy olvasásra megnyitott fájlba))            |
| ECHILD    | <b>NO CHILDREN</b><br>(Egy folyamat meghívta a wait függvényt anélkül, hogy a folyamatnak lett volna gyereke)                                       |
| EAGAIN    | <b>NO MORE PROCESSES</b><br>(A fork függvény nem futtatható, mivel a folyamattáblában nincs szabad bemenet, vagy a folyamatnak túl sok gyereke van) |
| ENOMEM    | <b>NOT ENOUGH SPACE</b><br>(Nincs elegendő memória)   |

|         |  |
|---------|--|
| EACCES  | <b>PERMISSION DENIED</b><br>(Nem megengedett módon történő hozzáférési próbálkozás egy állományhoz)  |
| EFAULT  | <b>BAD ADDRESS</b><br>(Fizikai hiba egy rendszerfüggvény hívása során)   |
| ENOTBLK | <b>BLOCK DEVICE REQUIRED</b><br>(Nincs megadva blokk típusú periféria, tehát egy ilyen megadása szükséges)   |
| EBUSY   | <b>MOUNT DEVICE BUSY</b><br>(Próbálkozás egy olyan periféria hozzáadására, amely már illesztve van, vagy egy már aktív periféria eltávolítása)   |
| EEXIST  | <b>FILE EXISTS</b><br>(Nem megfelelő hivatkozás a fájlra)  |
| EXDEV   | <b>CROSS-DEVICE LINK</b><br>(Próbálkozás két, különböző perifériákon levő fájl összekapcsolására (a link függvényhívással))  |
| ENODEV  | <b>NO SUCH DEVICE</b><br>(A megadott periféria nem létezik)  |
| ENOTDIR | <b>NOT A DIRECTORY</b><br>(Hivatkozás egy olyan állományra, amely nem könyvtár, bár erre szükség lenne)  |
| EISDIR  | <b>IS A DIRECTORY</b><br>(Írási kísérlet egy könyvtárba)   |
| EINVAL  | <b>INVALID ARGUMENT</b><br>(Hibás paraméter)   |
| ENFILE  | <b>FILE TABLE OVERFLOW</b><br>(Az állománytáblázatnak nincs szabad bemenete, tehát egy újabb fájl már nem nyitható meg)  |
| EMFILE  | <b>TO MANY OPEN FILES</b><br>(Egy folyamatnak maximum korlátos számú (20), egyidőben megnyitott állománya lehet, ezért nem nyithat meg újabb állományt és nem duplázhat meg egy fájlleíró) |
| ENOTTY  | <b>NOT A TYPEWRITER</b>  |
| ETXTBSY | <b>TEXT FILE BUSY</b><br>(Futtatási kísérlet egy olvasásra/írásra nyitott fájlból, vagy írási/olvasási kísérlet egy futó fájlba)   |
| EFBIG   | <b>FILE TOO LARGE</b><br>(Túl nagy az állomány)  |
| ENOSPC  | <b>NO SPACE LEFT ON DEVICE</b><br>(Egy állomány írása során elfogyott a fizikai memória)   |
| ESPIPE  | <b>ILLEGAL SEEK</b><br>(Elhelyezési kísérlet az lseek függvénnyel egy pipe fájlra)   |
| EROFS   | <b>READ ONLY FILE SYSTEM</b>   |

|        |  |
|--------|--|
|        | (Írási kísérlet egy olvasásra megnyitott fájlba vagy könyvtárba)   |
| EMLINK | <b>TOO MANY LINKS</b><br>(Túl sok link egy fájlra (több mint 1000))  |
| EPIPE  | <b>BROKEN PIPE</b><br>(Olyan pipe-ba való írás, amelyet egy folyamat sem olvas; ez a hiba csak akkor jelentkezik, ha nem vesszük figyelembe a megfelelő jelzést) |
| EDOM   | <b>MATH ARGUMENT</b><br>(Egy paraméter értéke a függvény által megengedett tartományon kívül esik)   |
| ERANGE | <b>MATH RESULT NON REPRESENTABLE</b><br>(Egy eredmény értéke nem ábrázolható a gép precizitásával)   |
| ENOMSG | <b>NO MESSAGE OF DESIRED TYPE</b><br>(Próbálkozás egy olyan üzenet fogadására, amelynek a típusa nem létezik a hálózaton)  |

(A fenti táblázat losif Ignat, Adrian Kacso *UNIX – Gestionarea proceselor* című könyvéből származik)

☰ Jelzéstípusok

| Szimbólum (sorszám)<br>(Megnevezés)              | Jelentés   |
|--|--|
| <b>SIGHUP (1)</b><br>(Hangup)                    | A terminál kikapcsolása (a terminálhoz való kapcsolódás) |
| <b>SIGINT (2)</b><br>(Interrupt)                 | A terminálon való megszakítás (Ctrl-/Break vagy DELETE)  |
| <b>SIGQUIT (3)</b><br>(Quit)                     | Egy folyamat feladása a Ctrl-\ billentyűkombinációval    |
| <b>SIGILL (4)</b><br>(Illegal instruction)       | Illegális utasítás                                       |
| <b>SIGTRAP (5)</b><br>(Trace trap)               |  |
| <b>SIGIOT (6)</b><br>(I/O Trap instruction)      | IOT utasítások végrehajtása                              |
| <b>SIGEMT (7)</b><br>(Emulator trap instruction) | EMT utasítások végrehajtása                              |
| <b>SIGFPE (8)</b><br>(Floating point exception)  | Tizedes vessző kivétel (felső túllépés)                  |
| <b>SIGKILL (9)</b><br>(Kill)                     | Egy folyamat erőteljes befejezése                        |

|   |   |
|---|---|
| <b>SIGBUS (10)</b><br>(Bus error)                             | Fővonal hiba  |
| <b>SIGSEGV (11)</b><br>(Segmentation violation)               | Szegmens túllépés hibás hivatkozás következtében                  |
| <b>SIGSYS (12)</b><br>(Bad argument to sysmem call)           | Hibás argumentum egy rendszerfüggvény hívásakor                   |
| <b>SIGPIPE (13)</b><br>(Write on pipe not opened for reading) | Írás egy olyan pipe állományba, amelyet egy folyamat sem olvas    |
| <b>SIGALRM (14)</b><br>(Alarm clock)                          | Figyelmeztető óra   |
| <b>SIGTERM (15)</b><br>(Software termination)                 | Softver befejezési jelzés (törli az időszakos állományokat)       |
| <b>SIGUSR1 (16)</b><br>(User defined signal 1)                | A felhasználó által definiált szabad jelzés                       |
| <b>SIGUSR2 (17)</b><br>(User defined signal 2)                | A felhasználó által definiált szabad jelzés                       |
| <b>SIGCLD (18)</b><br>(Death of child)                        | A szülő által kapott jel, amikor a gyerek befejeződött            |
| <b>SIGPWR (19)</b><br>(Power-fail restart)                    | Implementációfüggő: a feszültség csökkentése érdekében generálják |

(A fenti táblázat Iosif Ignat, Adrian Kacso *UNIX – Gestionarea proceselor* című könyvéből származik)

A **signal** függvényben egy jelet megadhatunk a sorszámaival, illetve a számára fenntartott szimbólummal (lásd a fenti táblázatban). Ezek a szimbolikus konstansok az `/usr/include/signal.h` állományban vannak definiálva.

Copyright (C) Buzogány László, 2002



[About](#)



# Index

A dokumentumban szereplő kulcsszavak, kifejezések, utasítások jegyzéke...

## Kulcsszavak, kifejezések

|                                   |  |                                  |
|-----------------------------------|--|----------------------------------|
| <a href="#">ANSI C</a>            | <a href="#">jelzések</a>                   | <a href="#">program</a>          |
| <a href="#">apa-fiú kapcsolat</a> | <a href="#">jelzéstípusok</a>              | <a href="#">programozás</a>      |
| <a href="#">attribútum</a>        | <a href="#">job</a>                        | <a href="#">PS1</a>              |
| <a href="#">állapot</a>           | <a href="#">katalógus</a>                  | <a href="#">PS2</a>              |
| <a href="#">azonosító</a>         | <a href="#">kiölés</a>                     | <a href="#">READY</a>            |
| <a href="#">C</a>                 | <a href="#">kliens-szerver rendszer</a>    | <a href="#">root</a>             |
| <a href="#">CREAT</a>             | <a href="#">könyvtár</a>                   | <a href="#">RUN</a>              |
| <a href="#">csoportok</a>         | <a href="#">Korn shell</a>                 | <a href="#">segítség</a>         |
| <a href="#">csővezeték</a>        | <a href="#">környezeti változók</a>        | <a href="#">semaphores</a>       |
| <a href="#">démon</a>             | <a href="#">kulcs</a>                      | <a href="#">shared memory</a>    |
| <a href="#">elérési út</a>        | <a href="#">LOGNAME</a>                    | <a href="#">SHELL</a>            |
| <a href="#">environment</a>       | <a href="#">MAIL</a>                       | <a href="#">signals</a>          |
| <a href="#">fájlrendszer</a>      | <a href="#">meghívás</a>                   | <a href="#">swapper</a>          |
| <a href="#">FIFO</a>              | <a href="#">memóriakezelés</a>             | <a href="#">szemafor</a>         |
| <a href="#">folyamat</a>          | <a href="#">message queues</a>             | <a href="#">szövegszerkesztő</a> |
| <a href="#">folyamatazonosító</a> | <a href="#">message</a>                    | <a href="#">szülő</a>            |
| <a href="#">folyamatok közti</a>  | <a href="#">névvel ellátott csővezeték</a> | <a href="#">TERM</a>             |
| <a href="#">kommunikáció</a>      | <a href="#">nice</a>                       | <a href="#">TERMINATE</a>        |
| <a href="#">függelék</a>          | <a href="#">osztott memória</a>            | <a href="#">utasítások</a>       |
| <a href="#">futtatás</a>          | <a href="#">pagedaemon</a>                 | <a href="#">üzenet</a>           |
| <a href="#">gyerek</a>            | <a href="#">parancssor paraméterei</a>     | <a href="#">üzenetsor</a>        |
| <a href="#">háttérfolyamat</a>    | <a href="#">PATH</a>                       | <a href="#">végrehajtás</a>      |
| <a href="#">hibakezelés</a>       | <a href="#">path name</a>                  | <a href="#">WAIT</a>             |
| <a href="#">hibakód</a>           | <a href="#">PID</a>                        | <a href="#">zombie</a>           |
| <a href="#">hibaüzenet</a>        | <a href="#">pipe</a>                       |                                  |
| <a href="#">HOME</a>              | <a href="#">prioritás</a>                  |                                  |
| <a href="#">init</a>              |  |                                  |

## Utasítások, parancsok

|                        |                          |                         |                        |                         |
|------------------------|--------------------------|-------------------------|------------------------|-------------------------|
| <a href="#">_exit</a>  | <a href="#">err_quit</a> | <a href="#">geteuid</a> | <a href="#">mknod</a>  | <a href="#">semctl</a>  |
| <a href="#">alarm</a>  | <a href="#">err_ret</a>  | <a href="#">getgid</a>  | <a href="#">msgctl</a> | <a href="#">semget</a>  |
| <a href="#">argc</a>   | <a href="#">err_sys</a>  | <a href="#">getpgrp</a> | <a href="#">msgget</a> | <a href="#">semop</a>   |
| <a href="#">argv</a>   | <a href="#">errno</a>    | <a href="#">getpid</a>  | <a href="#">msgrcv</a> | <a href="#">setgid</a>  |
| <a href="#">bg</a>     | <a href="#">exec</a>     | <a href="#">getppid</a> | <a href="#">msgsnd</a> | <a href="#">setjmp</a>  |
| <a href="#">calloc</a> | <a href="#">execl</a>    | <a href="#">getuid</a>  | <a href="#">mv</a>     | <a href="#">setpgid</a> |
| <a href="#">cat</a>    | <a href="#">execle</a>   | <a href="#">getgid</a>  | <a href="#">nice</a>   | <a href="#">setuid</a>  |

|                                 |                                |                                |                                |                                  |
|---------------------------------|--------------------------------|--------------------------------|--------------------------------|----------------------------------|
| <a href="#"><u>cd</u></a>       | <a href="#"><u>execlp</u></a>  | <a href="#"><u>getpgrp</u></a> | <a href="#"><u>open</u></a>    | <a href="#"><u>shmat</u></a>     |
| <a href="#"><u>chdir</u></a>    | <a href="#"><u>execv</u></a>   | <a href="#"><u>getpid</u></a>  | <a href="#"><u>P</u></a>       | <a href="#"><u>shmctl</u></a>    |
| <a href="#"><u>chmod</u></a>    | <a href="#"><u>execve</u></a>  | <a href="#"><u>getppid</u></a> | <a href="#"><u>pause</u></a>   | <a href="#"><u>shmdt</u></a>     |
| <a href="#"><u>chroot</u></a>   | <a href="#"><u>execvp</u></a>  | <a href="#"><u>getuid</u></a>  | <a href="#"><u>pclose</u></a>  | <a href="#"><u>shmget</u></a>    |
| <a href="#"><u>close</u></a>    | <a href="#"><u>exit</u></a>    | <a href="#"><u>hdr.h</u></a>   | <a href="#"><u>perror</u></a>  | <a href="#"><u>sigaction</u></a> |
| <a href="#"><u>cp</u></a>       | <a href="#"><u>fg</u></a>      | <a href="#"><u>joe</u></a>     | <a href="#"><u>pipe</u></a>    | <a href="#"><u>signal</u></a>    |
| <a href="#"><u>dup</u></a>      | <a href="#"><u>fgets</u></a>   | <a href="#"><u>kill</u></a>    | <a href="#"><u>popen</u></a>   | <a href="#"><u>sleep</u></a>     |
| <a href="#"><u>dup2</u></a>     | <a href="#"><u>fork</u></a>    | <a href="#"><u>ln</u></a>      | <a href="#"><u>printf</u></a>  | <a href="#"><u>strerror</u></a>  |
| <a href="#"><u>emacs</u></a>    | <a href="#"><u>fprintf</u></a> | <a href="#"><u>longjmp</u></a> | <a href="#"><u>ps</u></a>      | <a href="#"><u>system</u></a>    |
| <a href="#"><u>env</u></a>      | <a href="#"><u>fputs</u></a>   | <a href="#"><u>ls</u></a>      | <a href="#"><u>pwd</u></a>     | <a href="#"><u>unlink</u></a>    |
| <a href="#"><u>environ</u></a>  | <a href="#"><u>free</u></a>    | <a href="#"><u>make</u></a>    | <a href="#"><u>raise</u></a>   | <a href="#"><u>V</u></a>         |
| <a href="#"><u>envp</u></a>     | <a href="#"><u>fscanf</u></a>  | <a href="#"><u>malloc</u></a>  | <a href="#"><u>read</u></a>    | <a href="#"><u>vi</u></a>        |
| <a href="#"><u>err.c</u></a>    | <a href="#"><u>gcc</u></a>     | <a href="#"><u>man</u></a>     | <a href="#"><u>realloc</u></a> | <a href="#"><u>wait</u></a>      |
| <a href="#"><u>err_dump</u></a> | <a href="#"><u>getegid</u></a> | <a href="#"><u>mkdir</u></a>   | <a href="#"><u>rm</u></a>      | <a href="#"><u>waitpid</u></a>   |
| <a href="#"><u>err_msg</u></a>  | <a href="#"><u>getenv</u></a>  | <a href="#"><u>mkfifo</u></a>  | <a href="#"><u>rmdir</u></a>   | <a href="#"><u>write</u></a>     |

Copyright (C) Buzogány László, 2002



[About](#)