# Csővezetékek (pipe)

## Tartalom

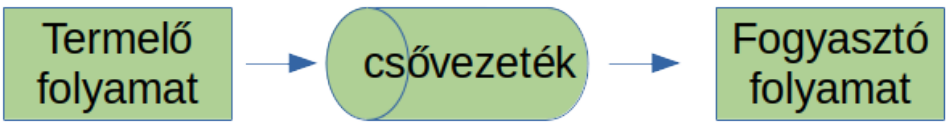


## *1. Egyszerű adatcsere folyamatok között csővezetékeken keresztül*

A továbbiakban két fontos témakörrel foglalkozunk, amelyek meghatározóak akkor, amikor egy operációs rendszer alatt több folyamat fut. Az első a folyamatok közti adatcsere, a másik a folyamatok közti szinkronizáció. Először az adatcsere legegyszerűbb formáit tekintjük át, majd a következő hetekben rátérünk a bonyolultabbakra, illetve a szinkronizáció különböző formáira.
Amikor shell parancsként leírjuk:

```
$ cat szoveg.txt | grep -i 'alma'
```

az első program kimenete egy kommunikációs csatornán jut el a másik program bemenetéhez. Ezt a csatornát csővezetéknek (*pipe*) nevezzük. A mechanizmus feltételezi egy termelő és egy fogyasztó folyamat jelenlétét, illetve egy operációs rendszer által megvalósított kommunikációs csatornát, amelyet a kernel hoz létre. A csatornához legalább két folyamat csatlakozik, az egyik csak ír, ezt termelő folyamatnak nevezzük (*producer*), a másik csak olvas, ezt fogyasztó folyamatnak nevezzük (*consumer*).




A csővezeték létrehozásakor a kernel gyakorlatilag egy puffert hoz létre, amelyben két mutató van, az egyik írásra a másik olvasásra. Ez a puffer a memóriában van, mérete 4-64 kilobyte operációs rendszertől függően, a modern Linux rendszerekben leggyakrabban 64 kilobyte.
Mivel a Unix alapú rendszerekben minden be és kimeneti műveleteket fájlok kezelésére vezet vissza a rendszer, a csővezetékek is fájlként lesznek kezelve. A folyamatok a csővezetéket állományleírókon keresztül érik el, így az ismert `read()` és `write()` függvényeket használhatják adatátvitelre. A csővezeték mérete véges, ezért amennyiben valamelyik folyamat gyorsabban dolgozik, például az első gyorsabban ír mint az olvasó folyamat, akkor a kernel blokkolja, mindaddig míg ürül hely, és ismét írásra kerül a sor. Ugyanez a helyzet az olvasásnál: ha nincs mit olvasnia a kernel blokkolja a folyamatot.

A blokkolás által a csővezetékek esetében a kernel oldja meg a szinkronizációt.

A Unix csővezeték egyirányú adatátvitelt tesznek lehetővé, azt mondjuk, hogy *half-duplex* kommunikációt valósítanak meg, ellentétben azokkal a módszerekkel ahol a kommunikáció *full-duplex*.
A csővezetékbe elméletileg egyszerre több folyamat is írhat és több olvashat, ilyenkor ezeknek szinkronizálniuk kell egymást. Ezen a laboron egyszerű esetekkel foglalkozunk, amikor egy

folyamat ír és egy folyamat olvas.

## 2. A pipe() függvény

Csővezetéket a folyamatok a `pipe()` függvénnyel hozhatnak létre:

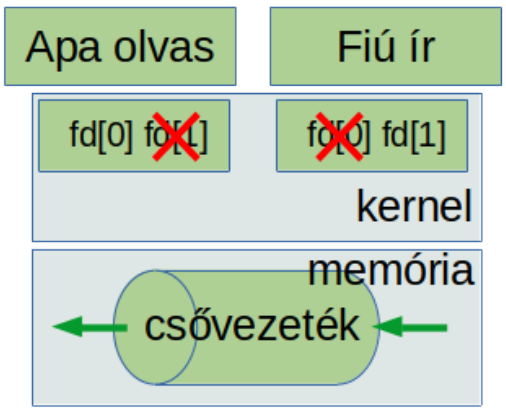
```
#include <unistd.h>
int pipe(int fd [2]);
```

Vissza: -1 hiba esetén, 0 siker esetén. Az `fd` tömbben két állományleírót térít vissza, az elsőt, `fd[0]` olvasásra, a másodikat, `fd[1]` -et a csővezetékbe való írásra lehet használni.

A `pipe()` függvény két állományleírót hoz létre, az másodikon keresztül írni lehet a csővezetékbe, az első pedig annak olvasására szolgál. Ugyanazokat a függvényeket (`read()`, `write()`) használjuk mint egy igazi állománynál. Az írás és olvasás viszont szekvenciális, tehát nem lehet az `lseek()` függvényt használni pozicionálásra. Két fontos dolgot kell megjegyezni:

1. Az adatok egy irányba közlekednek a csővezetéken.
2. Az állományt nem az `open()` függvénnyel nyitjuk hanem a `pipe()`-al, így azonnal állományleírót kapunk, ami csak apa-fiú kapcsolatban továbbítható más folyamatnak (a fiúnak). Tehát az egyszerű csővezetékeket csak közös őssel rendelkező folyamatok használhatják, mivel ezeknél a megnyitott állományleírók öröklődnek.

Ha például egy apa folyamat megnyit egy csővezetéket, az alábbi esetben felhasználhatja az első állományleírót olvasásra, a fiú pedig a második leírót írásra. A következő struktúra jön létre:




Mivel csak egy irányban lehet majd közvetíteni, ezért a másik irányt mindkét folyamatnak le kell zárnia.
A nem használt azonosító lezárásához a `close()` függvényt kell használni.
Példaprogram: **`pipe.c`** , tesztelés:

```
$ gcc -Wall pipe.c -o pipe
$ ./pipe
az apa kiolvasta amit fiú folyamat írt: Helló!
$
```

A csővezetékek gyakorlati megvalósítása különbözhet rendszerenként, így pl. más méretű lehet a

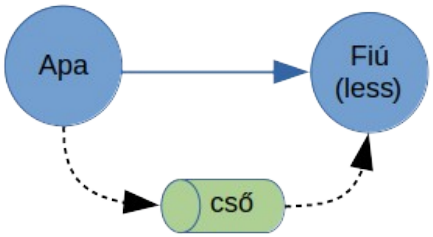
memóriában allokált közös puffer mérete.

> Linuxon a konkrét megvalósítást a **`man 7 pipe`** kézikönyv oldalon találjuk meg.

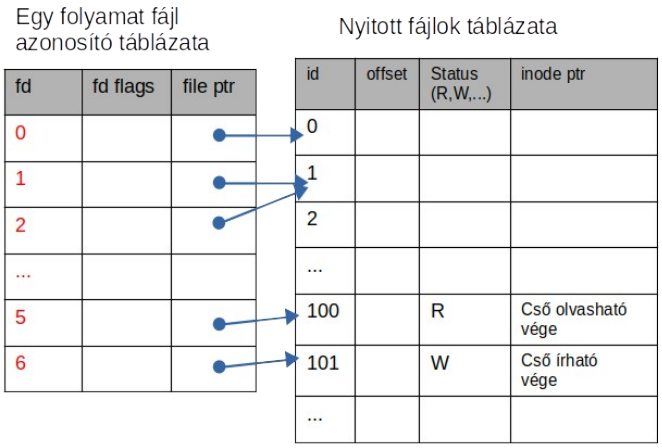
Ubuntu 20 esetében az alapértelmezett puffer méret 65536 byte (65kB), de megnövelhető programból 1048576 byte-ra (1048kB).

## 3. Csővezetékek átirányítása

A gyakorlatban nagyon sok olyan eset van, amikor valamilyen feladatra egy már jól bevált, az operációs rendszer alatt létező programot akarunk felhasználni egy általunk írt programból. A klasszikus példa erre, ha egy általunk írt C programból listázni akarunk, és mivel nem szeretnénk ha egy hosszú lista elfutna a terminál kimenetén, a listát a `less` parancs használatával szeretnénk felfogni majd lapozni. Elindítjuk egy `fork()` és `exec()` segítségével a `less` programot, a saját programunkból egy csővezetékbe írunk, a fiúnak pedig a cső másik végét kell olvasnia. A `less` egy szöveget olvasható formában kiíró program, lapozni tudjuk a szöveget, q-val lehet kilépni belőle.




Az apa az általa létrehozott csővezetékbe ír. A cső másik végének leíróját megkapja a fiú, amelyben a `less` fog futni (`exec`-el indítva). Ez viszont csak akkor működik, ha a `less` a csővezeték azonosítóját úgy kapja meg *mint a standard input* azonosítóját. Hiszen a `less` alapértelmezésben szűrőként működik, tehát standard bemenetet olvas és standard kimenetre ír.
A kernel egy folyamat esetében a folyamat leíró blokkban a következő módon tartja nyilván a megnyitott fájlokat:



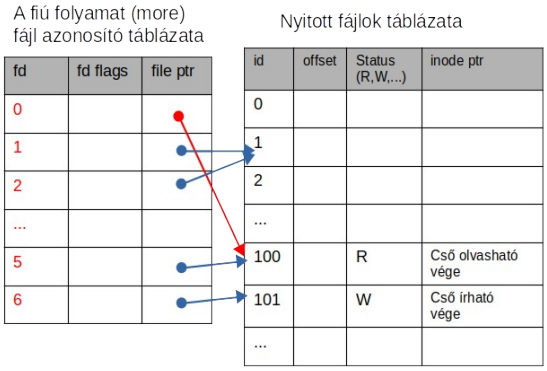

Az fiú által kapott azonosítónak viszont nem annyi az azonosító száma mint a standard bemenetnek,

meg kell tehát oldani ennek az azonosítónak hozzárendelését a standard inputhoz. A standard bemenet állományleírójának értéke általában 0, míg a `pipe()` függvény egy más számot ad vissza állomány leíróként, pl. 5-öt.

Példaprogram: **`useless.c.`**
Mivel a standard input állományleírója az 0-s, amelyet név szerint is megadhatunk, mint: `STDIN_FILENO` (ennek a fejlécekben definiált értéke általában 0), az alábbi hívás (amennyiben int pfd[2]; a csővezeték azonosítói):

```
// hozzárendelem a pipe olvasható véget az stdin-hez
if (dup2(pfd[0], STDIN_FILENO) < 0) {
            syserr("dup2");
}
```

Az `fd1`-ben az `STDIN_FILENO`, azaz 0 értéket ad vissza, ugyanakkor az addig csak `fd`-vel azonosított állományhoz az `STDIN_FILENO` azonosítót rendeli. A fájl azonosító táblázat a következő lesz a fiúban, ahol a `less` fut:




Hibalehetőség: vigyázni kell arra az esetre, ha a `dup2()` esetében már a hívás előtt egyenlő a két paraméterként megadott leíró. Ilyenkor `dup2()` visszatéríti ugyanazt a leírót, anélkül hogy előtte lezárná. Ez akkor becsapós, ha közvetlenül a `dup2()` után lezárjuk a régi leírót, azt gondolván, hogy újat kaptunk mert ilyenkor lezárjuk azt az állományt amivel tovább kellene dolgozni.

Az **`useless.c`** példa az említett `less` program meghívásának megvalósítása.
Így futtatjuk:

```
$ gcc -Wall useless.c -o useless
$ ./useless be_hosszu.txt
...
```

## 4. Névvel rendelkező csővezetékek (named pipes) vagy fifo-k

A `pipe()` függvénnyel létrehozott csővezeték csak apa-fiú folyamatok közt használható. A visszaadott fájl azonosító nem küldhető át egy külön futó, nem fiú folyamatnak.

Két külön futó folyamat között is lehet csővezetéket létrehozni, de ehhez meg kell oldani, hogy egy közösen elérhető névvel érjék el a folyamatok a folyamatok a csővezetéket.

Ahhoz, hogy két folyamat elérjen egy közös kommunikációs lehetőséget, az operációs rendszer egy fájlrendszerben levő nevet használ fel, mint speciális fájl. Ezt nem csak a

fifo, hanem más kommunikációs megoldásban is használja.

A csővezeték (fifo) esetében, ez egy csővezeték típusú fájl (a `ls -l` kimenetében a fájl típus **p**).

A **`fifo`** könyvtárban levő példák a névvel rendelkező csővezeték használatát mutatják be. Ez a szerkezet pontosan úgy működik mint egy csővezeték, de van egy neve a fájlrendszerben, így nem csak apa-fiú kapcsolatú folyamat használhatja, hanem két független folyamat is.

A fájlrendszerben található nevet `open()` vagy `fopen()` függvényekkel is meg lehet nyitni (tehát lehet használni kernel szintű vagy a standard könyvtár függvényeivel is).

A kommunikáció akkor indul el, ha egy folyamat olvasható (read), egy másik pedig írható (write) módban nyitotta meg a fájlt. Csak miután mindkét nyitás lefut, jön létre egy ugyanolyan csővezeték, mint a `pipe()` függvény által létrehozott, és a két folyamat ezen keresztül küldhet át adatokat, pontosan úgy, mint a `pipe()` függvény által visszaadott csővezeték esetében.

Amíg nem hívódik meg mindkét fájl nyitás, a folyamatok blokkolódnak az `open()` hívásaikban.

Fifo-t létrehozhatunk a shellből is a `mkfifo` paranccsal, vagy C programból a `mkfifo()` függvénnyel.

Shellből:

```
$ mkfifo fifo1
$ ls -l
összesen 0
prw-rw-r-- 1 lszabo lszabo 0 Mar  3 23:14 fifo1
$ cat fifo1 &          #a cat a háttérben olvas a fifoból
[1] 4036
$ echo abc >> fifo1    #az echo a fifoba ír
  abc                  # a cat kiolvassa a fifo-t

[1]+  Kész                    cat fifo1    #cat kilép
$
```

A **`fiforead.c`** és **`fifowrite.c`** programok a névvel rendelkező csővezeték C programból való használatát mutatják be. Magyarázatok a C kódban. Használatuk:

```
$ gcc -Wall fiforead.c -o fiforead
$ gcc -Wall fifowrite.c -o fifowrite
$ ./fiforead fifo1 &      #háttérben olvas
[1] 4094
$ ls
fifo1  fiforead  fiforead.c  fifowrite  fifowrite.c  myinclude.h
$ ./fifowrite fifo1
teszt szöveg
teszt szöveg
(Ctrl-D)
[1]+  Kész                    ./fiforead fifo1
$
```

A próbát egyszerűbb, ha két terminálon végezzük, egyiken a `fiforead` a másikon a `fifowrite` fusson.

## 5. Példaprogramok

További példaprogramok:

1. Az **fdopen.c** az `fdopen()` függvény használatát mutatja be, ez a függvény átalakít egy alsó szintű fájl azonosítót egy `FILE *` típusú mutatóvá, így magas szintű (standard könyvtár) függvényekkel is lehet csővezetékbe írni.

2. A **dup.c** program, amely a `sort` parancsot indítja el, egy rendezési feladatot megoldani. A fiú folyamatban a `sort` parancsot futtatjuk, ez elrendezi azt a szöveget amit az apa folyamat ír ki.

Így futtatjuk:

```
$ gcc -Wall dup.c -o dup
$ ./dup
aaa
abc
klm
xyz
$
```

3. A **start2.c** , az alábbi, shell alatti parancsvégrehajtást valósítja meg:

```
$ cat | sort
```

azaz két parancsot indít el, és az első kimenetét a második olvassa. A két parancsnevet adjuk meg az általunk írt program parancssorán, tehát így indítjuk programunkat:

```
$./start2 cat sort
xyz
abc
klm
(Ctrl-D)
abc
klm
xyz
$
```

A standard bemenetre írt szöveget végül a `sort` rendezi el, a bemenetet zárjuk Ctrl-D-vel

4. **popen.c** a `popen()` függvény használatát mutatja be, magyarázat a fájl elején és a labor 8. feladatának leírásában.