5. Csővezetékek – gyakorlat feladatai

Tartalom

1. Csővezeték apa fiú között

(Sablon: fel1.c) Oldjuk meg a következő feladatot: egy folyamat megnyit egy fájlt (a fájl lehet szöveg vagy akár bináris is, pl. egy kép) és átküldi egy csővezetéken a fiának. A fiú folyamat kimenti egy más nevű fájlba. Kilépés előtt a fiú kiírja, hogy hány byte-ot tartalmazott a fájl.

Apa folyamat megvárja a fiát. A két fájlnevet a parancssoron adjuk meg, és ellenőrizni kell, hogy ne legyenek azonosak. Miután a fiú kilépett, az apa a `cmp` parancsot egy `exec` függvénnyel indítva ellenőrzi, hogy a két fájl tartalma azonos, ha a `cmp` hiba nélkül fut le, az apa kiírja: „OK”.
Futtatás:

```
$ ./fel1 jet.png jet1.png
fiú: 3554 byte
apa: OK
$
```

2. Az fdopen() függvény

(Sablon: `fel2.c`)Az `fdopen()` függvény átalakít egy alsó szintű fájl azonosítót (`int fd`) egy `FILE *` típusú mutatóvá, így a standard könyvtár függvényeivel is lehet a csővezetékeket használni.

```
FILE * fdopen(int fd, const char *mode);
```

A `mode` paraméterben meg kell adni, hogy írni vagy olvasni akarunk az állományból, tehát ennek értéke `"r"` vagy `"w"` . Használatát lásd az `fdopen.c` programban. A függvény man lapja: **`man 3 fdopen`** .

A `fdopen.c` programot alaposan át kell nézni mielőtt nekifogunk a feladatnak. Ebben található példa a cső író, illetve olvasó végének átalakítására.

Feladat: a `fdopen()` függvényt használva írjuk át az 1. feladatot úgy, hogy az apa a csővezetékbe az `fputs()` függvénnyel írjon, a fiú pedig a csővezetékből pedig az `fgets()` függvénnyel olvasson.

Az átírásokat **szöveg soronként** végezzük, így csak szöveges fájlokat vihetünk át a csővezetéken!

3. Szűrő

(`charfilt.c`) Egy apa folyamat a szöveget olvas a standard bemenetről soronként, és továbbítja a fiának egy csővezetéken keresztül. Írjunk egy olyan fiú folyamatot, amely egy függvénnyel megszűri az apa folyamat által a kimenetre írt szöveget, és úgy írja be egy fájlba. A fiú folyamat olvashat pufferelve, nem kell néznie azt, hogy sorok jönnek. A szöveg angol nyelvű szöveg, ne használjunk magyar ékezetes karaktereket.
a. Az első változatban a függvény legyen egy rögzített függvény, amely minden kisbetűt nagybetűvé alakít.
b. Utána módosítsuk úgy, hogy az apa folyamat a fiú indításakor döntse el, hogy milyen szűrővel futtatja, és azt állítsa be a fiú indítása előtt mutató formájában (a fiú örökli a mutatót és a függvény kódot is). A program indításakor az apa folyamat kapja meg egy opcióban, hogy milyen szűrőt ad át a fiának.
A szűrő prototípusa legyen:

```
char filter ( char c);
```

A mutató pedig amit beállít:

```
char (*fptr)(char) ; // a fptr függvényre mutató mutató
```

Írjunk legalább 3 szűrő függvényt amelyek az alábbi műveleteket végzik:

- `-l` opció: kisbetűt nagybetűvé alakít,
- `-p` opció: kiveszi a punktuációs karaktereket a szövegből és helyettük szóközt ír (pl.: `. ! ; ` , az `ispunct()` függvénnyel kapjuk meg őket),
- `-d` opció: csak a számjegyeket hagyja meg a szövegben, minden más helyre szóközt ír.

Program indítás:

```
./charfilt -l ki.txt
```

Az apa a standard bemenetről olvas soronként, és a fiú a `ki.txt` fájlba írja a szűrt szöveget. Vagy a standard bemenetre küldünk egy teszt fájlt:

```
./charfilt -l ki.txt < be.txt
```

A tesztekhez szükséges függvényeket itt találjuk meg: `man 3 ispunct`, `man 3 toupper`, `man 3 isdigit`.

4. Csővezeték két fiú folyamat közt

(megoldva mint `start2.c`) Írjunk egy programot amelyik 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
```

Feladat: (`start22.c`) Valósítsuk meg úgy is a programot, hogy a 2 programnak parancssori paraméterei is legyenek, pl.:

```
cat a.txt | sort -nr
```

megvalósításához indítsuk az alábbi módon:

```
./a.out "cat a.txt"  "sort -nr"
```

5. Szinkronizálás csővezetékkel (1)

(Sablon: `fel5.c`) A csővezeték felhasználható egyszerű szinkronizálásra két folyamat között. Pl. ha egy apa folyamat várakoztatni akarja a fiait addig, amíg el nem végez egy feladatot, ezt megoldhatja csővezetékkel.

Ez megvalósítható karakter átírás nélkül is, egyszerűen fiúknak a `read()` függvényben való **blokkolása** révén, amennyiben nem írunk a csőbe. Egy folyamat, amelyik olvas a csővezetékből blokkolva van addig, amíg nem tud annyi karaktert olvasni amennyit kért. A blokkolás véget ér akkor is, ha lezárjuk a cső írható végét.

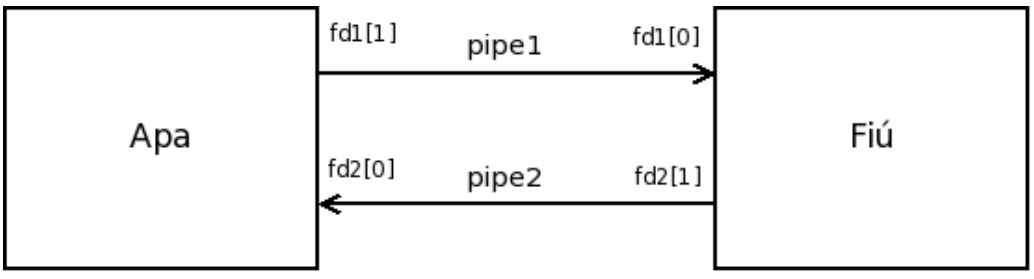
Valósítsuk meg az alábbit:

- Egy apa folyamat indítson 2 fiú folyamatot, és várakoztassa mindkettőt addig amíg befejez egy feladatot: kiír egy szövegsort egy fájlba és lezárja a fájlt, pl. azt, hogy „Helló!”.
- A fiúk várakozását oldjuk meg úgy, hogy egy, az apával közös csővezetékből akarnak mindketten olvasni egy byte-ot.
- Ha az apa befejezte a fájl létrehozását, engedje mindkét fiút futni úgy, hogy bezárja a cső írható végét.
- A fiúk olvassák végig az állományt és írják ki a terminálra (az állományban teszteléskor csak egy szövegsor legyen).
- Az apa folyamatban egy `sleep()` függvénnyel várakoztatva az apát a fájl létrejötte után jól fog látszani, hogy a fiúk csak akkor indulnak, ha az apa lezárta a csővezetéket.

6. Szinkronizálás csővezetékkel (2)

(R. Stevens feladat) Valósítsunk meg egy olyan szinkronizációs kommunikációt apa és fiú között, ahol az apa tud egy várakozási művelettel a fiúra várni, a fiú pedig egy adott ponton közölheti az apával, hogy ne várakozzon tovább. Utána mindketten párhuzamosan futnak. Ugyanezt fordítva is működtethetjük (fiú vár az apára). Ehhez kétirányú kommunikációra van szükség, amit két csővezetékkel valósítunk meg.

Az alábbi ábra szerint van szükségünk a két csővezetékre:



Ha a fiú várakoztatja az apát az alábbi műveletekre van szükség. Ezeket mint C függvényeket kódolhatjuk, és külön állományban fordíthatjuk le egy fejléc állománnyal:

```
void init_wait();
```

ezt a függvényt az apa hívja meg a `fork()` hívás előtt, és csak annyit csinál, hogy megnyitja a két csővezetéket. Ha külön állományban kódoljuk a függvényeket, akkor a két `pipe()` azonosítót tartalmazó változó is ebbe az állományba kerülhet és globálisak kell legyenek.

```
void wait_child();
```

ezt az apa hívja meg, és megpróbál egy karaktert olvasni az `fd2` csővezetékből. Addig, amíg a fiú nem ír a csőbe egy karaktert, az apa blokkolva lesz, tehát vár. Megállapítunk egy konvenciót, hogy a fiúnak a `'p'` karaktert kell küldeni a várakozás beszüntetésére. Tehát ez a függvény egy karaktert olvas:
– ha megjött a karakter és nem 'p' hibával kilép
– ha 'p' akkor visszatér a függvény

```
void tell_parent();
```

ez a függvény értesíti a fiú az apát arról, hogy már nem kell várakoznia. Ezért beír egy 'p' karaktert az `fd2` csővezetékbe.

A két folyamat így fog viselkedni:

Apa	***Fiú***
`init_wait();` fd1, fd2 létrejön	
`fork()`	fiú létrejön, örökli fd1, fd2-t
`wait_child();`	feladatát végzi. ennek a végét kell az apának várnia.
vár	
itt olvassa be a 'p' karaktert	`tell_parent();`
tovább fut fiútól függetlenül	tovább fut apától függetlenül

Ugyanígy megírjuk a `wait_parent()` és `tell_child()` függvényeket az `fd1`-et használva.

Külön állományban kompiláljuk és így általánosan lehet majd használni más feladatoknál is.

A megoldott feladat két C fájlból áll: a szinkronizáló függvényeket és az alábbi feladatot tartalmazó kód.

Teszteléshez írjuk meg az alábbi kódot:

– apa elindul, létrehozza a két csővezetéket az `init_wait()`-el, majd fork, és vár a fiú értesítésére a `wait_child()`-ot meghívva;
– a fiú indulás után létrehoz egy állományt, amibe két valós számot ír, pl. 2.5 és 3.6 , lezárja, utána értesíti az apát: `tell_parent()`, hogy a fájl kész, majd vár az apa értesítésére: `wait_parent()`;
– a két valós számot a fiú a terminálról olvassa és tetszőleges kódolással írja a fájlba (szövegesen

vagy binárisan);
– az apának ezt meg kell várnia: `wait_child()`, és csak akkor indulhat tovább, ha a fiú lezárta az állományt, kiolvassa a két valós számot, összeadja őket, és a fájl csonkolása után beírja a fájlba, majd jelez a fiának: `tell_child()`, hogy ki lehet olvasni az eredményt; megvárja a fiát és kilép;
– a fiú tovább indul az apa jelzésére, kiolvassa az eredményt és kiírja a standard kimenetre, utána kilép.

Az `init_wait()` után nem zárjuk le a cső kihasználatlan végét. Ilyenkor csővezetéknél lehetséges, hogy két folyamat írjon bele. Mi itt természetesen csak az egyikből írunk. Ha az egyik folyamat lezárja a cső írható végét, az olvasható végén a `read()` nem jelez fájl véget, ahogy egyébként tenné amikor csak egy folyamat ír a csővezetékbe.

A szinkronizáció végén nem zárjuk le a csővezetékeket: ezt majd az apa `exit()` hívása után történik meg a kernelben.

7. Két fiú

Valósítsuk meg az alábbi feladatot:

a. Egy apa folyamat két fiút hoz létre,
- az egyiknek végig kell olvasni egy szöveges állományt soronként, és át kell küldenie a másik fiúnak,
- átküldés előtt minden sort N karakter hosszúra vág: a fennmaradó karaktereket eldobja,
- a második fiú átveszi az N karakteres sorokat és kiírja egy fájlba.

Az N értékét adjuk meg parancssoron.

Figyeljünk oda, hogy minden *nem használt fájl azonosítót zárjunk le!* A csővezetéket az apának kell létrehozni, de ő nem fogja használni: így mielőtt megvárja a fiait mindkét végét le kell zárnia, egyébként a fiai azt fogják hinni, hogy valaki még használja a csövet. Az apa folyamat megvárja fiait. Parancs indítás:

```
./a.out a.txt b.txt 5
```

Az `a.txt`-t átalakítja 5 karakteres sorú `b.txt`-vé.

b. Egészítsük ki a feladatot a következővel: a második fiú rendezze az átvett szöveget a `sort`-al: tehát ne állományba írjon, hanem hozzon létre egy fiú folyamatot (az apa szempontjából unoka), amelyben a `sort` fut, és annak írja át a szöveget, és ez írjon a fájlba.

8. A popen() függvény

Tanulmányozzuk a `popen.c` példaprogramot és a `man 3 popen` kézikönyv oldalt.

A `popen()` függvény egy parancs teljes a parancssorát kapja meg (a példában: `grep 'alma'`), és a következőket végzi:
1. megnyit egy csővezetéket;
2. utána elindít egy fiú folyamatot, és abban `exec`-el a megadott parancssort;
3. a `popen()` függvény utolsó paramétere "r" vagy "w": ennek megfelelően az elindított parancs ír vagy olvas a csővezetékből átirányítással (a példában "w", akkor az elindított `egrep` olvasni fog, mert a `popen()`-t hívó folyamat fog írni!);

4. a hívó folyamatnak a `popen()` a cső másik végét adja vissza mint 2. szintű `FILE *` : ha a paraméter "w" volt mint a példában, akkor írni lehet bele 2. szintű függvényekkel;
5. a hívó folyamat így írhat (mint a példában) vagy olvashat a csőből;
6. ha befejezte `pclose()`-al kell lezárja a `FILE *` -t;
7. a `pclose()` hívás alatt lefut a `wait()`, így a hívó folyamat megvárja a fiát;
8. a `pclose()` visszaadja a `wait()` status értékét.

A programot így indítjuk:

```
./popen a.txt
```

9. Számoló fiú

(`szamolj.c`) Egy apa folyamat olvasson valós számokat egy szöveges fájlból és küldje át a fiának bináris formátumban egy csővezetéken keresztül A fiú folyamat írja ki: a legnagyobb és legkisebb számot, valamint a számok összegét egy fájlba, egymás után a hármat külön sorokba, az alábbi minta szerint. Ha vége van a fájlnak, az apa folyamat bezárja azt, ennek hatására a fiú kilép, az apa megvárja a fiát.

`be.txt` tartalma:

```
2.7
1.8
3.2
0.23
-2.8
5.2
6.3
```

Program indítás:

```
./szamolj be.txt ki.txt
```

`ki.txt` kimeneti fájl:

```
Összeg: 15.00
Ln: 6.3
Lk: -2.8
```

Mivel binárisan kell átküldeni a számokat, a fájlból `scanf()` függvénnyel olvasunk egy `double` típusú változóba soronként, utána ez a változót minden sor után átírjuk a csővezetéken bináris formátumban, úgy ahogy a memóriában van, a következő módon. Egy valós szám byte-okban való hossza a memóriában `sizeof(double)`, ennyit írunk és olvasunk egy alkalommal:

```
double x;   //ebben van a szám
int pfd[2]; // a csővezeték azonosítói
//írás:
if (write (pfd[1], (void*) &x, sizeof(double))<0) {
        syserr("write");
}
```

olvasás a másik oldalon:

```
int n;
double x;
while ((n=read(pfd[0], (void*) &x, sizeof(double))>0)){
```

```
         //... feldolgozás
        //itt egy olvasáskor az n-nek sizeof(double) értéknek kell lennie
        //ezt lehet ellenőrizni !!!
}
if (n<0){
    syserr("read");
}
```

10. A sort használata fiú folyamatban

(`sortchild.c`) Írjuk meg a következő apa-fiú folyamatokat:

a) Apa folyamat létrehoz egy csővezetéket és egy fiú folyamatot.

b) Az apa folyamat egy szöveges fájlt olvas, az olvasást a `read()` függvénnyel végzi és 16 byte-os puffert használ (nem soronként olvas, ömlesztve átküldi a fájlt).

c) Átküldi a szöveget a fiának.

d) A fiú folyamatban a `sort` programot futtatjuk, ez a csőből olvas és egy kimeneti fájlba írja a számok szerint rendezett sorokat. A kimeneti fájlba való írást átirányítással kell megoldani átirányítással, és nem a `sort -o` opcióját használva.

e) Fájl végére apa lezárja a csövet, ennek hatására a sort kilép.

Bemenetnek használjuk a `lab4_exec/sablonok/szam.txt` fájlt. Tesztelés:

`./sortchild szam.txt ki.txt`

11. Két irányú kommunikáció

Oldjunk meg két irányú kommunikációt apa és fiú folyamat közt csővezetékkel. Az apa folyamat létrehoz két csővezetéket, egyiken majd a fiának ír, a másikon a fiától olvas. Utána elindít egy fiú folyamatot és a következőket végzi:

Egy bementi fájlból valós számokat olvas, egy olvasáskor két számot olvas be, a fájl a következőképpen néz ki:

```
1.1 2.2
3.3 4.4
5.5 6.6
7.7 8.8
9.9 10.1
```

A sorokból az első számot az apa fogja feldolgozni, a másodikat a fiú, tehát a második számot átküldi a fiának. Utána „feldolgozza” az első számot, ez legyen 2-vel való osztás. A fiú folyamat beolvassa az egyik csővezetéken a valós számot, megszorozza 2-vel, és a másik csövön át visszaküldi. Az apa folyamat átveszi a fiútól a feldolgozott számot, összeadja a saját eredményével, és kiírja a standard kimenetre. Tehát ketten a következő műveleteket végzik egy sorral, amennyiben a két szám *x1* és *x2*: $x1 / 2 + x2 \times 2$. Ezt addig ismétlik, amíg vége van a fájlnak, a fájl minden sorában két számnak kell lennie.

Ha vége van a fájlnak, az apa folyamat lezárja a fiú felé írható csővezetéket, ennek észlelésekor (fájl vége a fiú oldalán), a fiú is lezárja az apa felé írható csővezetéket és kilép. Apa megvárja a fiát és kilép. Mivel a számokat egyenként írják át, a fiú a következő szám olvasásánál szinkronizálódik az apával (tehát csak akkor tudja beolvasna, ha az apa átírta), ugyanígy az apa is szinkronizálódik a fiú írásaira (nem tud előre futni még egy sort olvasni a fájlból, mert meg kell várnia a 2. szám

feldolgozott eredményét a fiútól).

A számokat írjuk át binárisan egyenként, tehát egy *double* típusú változó címéről, *sizeof(double)* méretet használva íráskor és olvasáskor is.

A kimeneten ezt kell látnunk:

```
4.95
10.45
15.95
21.45
25.15
```

Teszteléskor tegyünk a fiúba a feldolgozás mellé egy 500 ms-os várakozást az `usleep()` függvénnyel, így is kell működnie a programnak.

12. Csővezeték használata adatok átírására standard könyvtár függvényekkel

(`fdopenrw.c`) Egy apa folyamat írjon át a fiának számokat, amelyeket egy szöveges fájl soraiból olvas. Egy írással két valós számot írjon át a `be.txt`-ből (egy sor tartalmát, de a két számot külön vegye két valós típusú változóból):

```
1.1 2.3
-4.2 1.2
-3 2
-1.1 3.3
2.3 3.3
0.2 0.6
```

A fiú folyamat írja ki minden kapott szám párosból a nagyobbik számot a kimenetre. Az átírást végezzük átalakítva a csövet `FILE *` -re, majd az `fprintf()` és `fscanf()` függvényekket használva.

Pl. ha

```
int pfd [2];
float x1,x2;
```

pfd a csővezeték azonosítói, és x1, x2 a két szám, az apa folyamat feladata így oldható meg:

```
 //apa feladat
   if (close (pfd [0])<0) {
        syserr("close");
    }
    FILE *fr, *fw;
    if ((fr=fopen("be.txt","r")) == NULL) syserr("fopen1");
    //írható pipe vég átalakítása
    if ((fw=fdopen(pfd[1],"w")) == NULL) syserr("fdopen1");

    while (fscanf(fr,"%f %f", &x1, &x2) == 2){
        if (fprintf(fw,"%f %f\n", x1, x2) < 0) syserr("fprintf");
    }
    if(ferror(fr) || ferror(fw)){
       syserr("fájl hiba");
    }
    fclose (fw); fclose (fr);
```

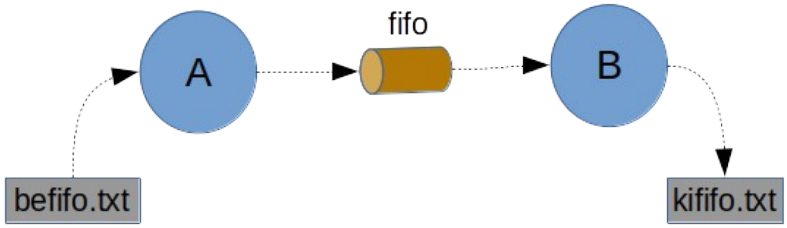
```
    wait(NULL);
// ... stb.
```

13. Fifo használata külön futó folyamatok közt

Tanulmányozzuk a névvel rendelkező csővezetékek (fofo) működését az 5_pipe.pdf labor útmutató 5.5 pontban bemutatott példa alapján (további magyarázatok a forráskódokban).

Oldjuk meg az alábbi feladatot:

(`fifoA.c`, `fifoB.c`) Olvassuk be egy A külön futó folyamatba a `befifo.txt` fájlból a számokat, írjuk át ezeket egy fifo szerkezeten egyenként a külön futó B folyamatnak. B írja két egymás utáni szám összegét két tizedes pontossággal a `kififo.txt` fájl egymás utáni soraiba.



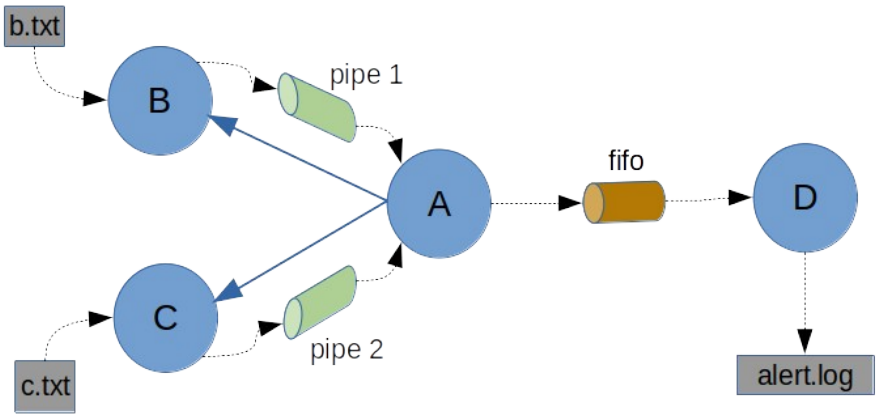
A `befifo.txt` fájl tartalma:

```
1.11 2.31
-4.2 1.2
-3 2.234
-1.1 3.3
2.3 3.33
0.2 0.6
```

A `kififo.txt` első sorának tartalma 3.42, a második soré: -3.00, stb. Az összeadást a B folyamatban kell elvégezni. A fifo-t meg lehet nyitni első szintű (`open()`) vagy standard könyvtár (`fopen()`) fájlkezeléssel is.

14. Hőmérős feladat

(`homerok.c`, `alert.c`) Egy apa folyamat (A), két fiú folyamattal (B és C) olvastatja le két hőmérő értékét. Mindkét fiú egy-egy csővezetéken át küldi fel a beolvasott számokat az apának. Az átküldést megoldhatjuk binárisan (mint a 9. feladatban) vagy `fprintf()`/`fscanf()`-et használva, mint a 12. feladatban.
B és C két érték beolvasása közt 400 ms-ot várnak (`usleep()`-el). Egy ciklusban az apa egy számot olvas B-től, majd egy számot C-től. A számok valós számok.



Az apa kiszámolja a két érték különbségét, és ha ennek abszolút értéke nagyobb mint 2, akkor jelzi azt egy fifo szerkezeten keresztül egy 4., külön futó folyamatnak (D), átküldve a két értéket. A 4. kiszámolja a különbséget, és egy fájlba írja egy milliszekundum szintű időbélyeget írva eléjük.

B által olvasott fájl:

```
11.2
10.3
10.4
12.6
11.8
10.12
```

C által olvasott fájl:

```
11.5
12.4
10.9
12.3
13.9
10.12
```

D kimenete hasonló kell legyen az alábbihoz:

```
2021-03-19 22:49:13.410 -2.1
2021-03-19 22:49:14.220 -2.1
```

Ha befejeződik a beolvasás (véget ér a két fájl, fiúk lezárják a csővezetéket), az apa folyamat megvárja a két fiát és kilép, majd lezárja a fifo-t és kilép. A fifo lezárása után D is kilép.

Azért, hogy elkerüljük a fifo megnyitásánál adódó blokkolódási problémát, a feladat első része (A,B,C) megoldható függetlenül a másodiktól (fifo és D) úgy, hogy az A kimenetét a standard kimenetre írjuk. Ha ez megvan, beírhatjuk A kódjába a fifo megnyitását, elindítjuk a D folyamatot egy külön terminál ablakban és elkezdhetjük tesztelni a fifo-ba írást.

C programból az időt a `time.c` kód felhasználásával írhatjuk ki (letölthető a Moodle-ról a labor szakaszából), amelyből függvényt készíthetünk (https://moodle.ms.sapientia.ro/pluginfile.php/20153/mod_resource/content/5/time.c).

15. Atomikus írás csővezetékbe

Mi történik, ha két folyamat ír egy csővezetékbe, mindkettő a `write()` függvényt használja és nincsenek szinkronizálva? Megtörténhet, hogy az egyik `write()` félig fut le és a cső olvasható végén a két folyamat egyedi `write()` írásai összekeverednek?

Atomikus műveletnek nevezzük azokat a kernel függvény végrehajtásokat,a melyek ha meghívunk, akkor nem lesznek megszakítva ha elkezdtek futni, hanem teljes egészben lefutnak. A csővezetékek esetében a `write()` atomikusan fut le ha az írt puffer hossza maximum 4096 byte a Linux esetében a jelenlegi Ubuntu rendszereken. Ez a

```
$ less /usr/include/linux/limits.h
```

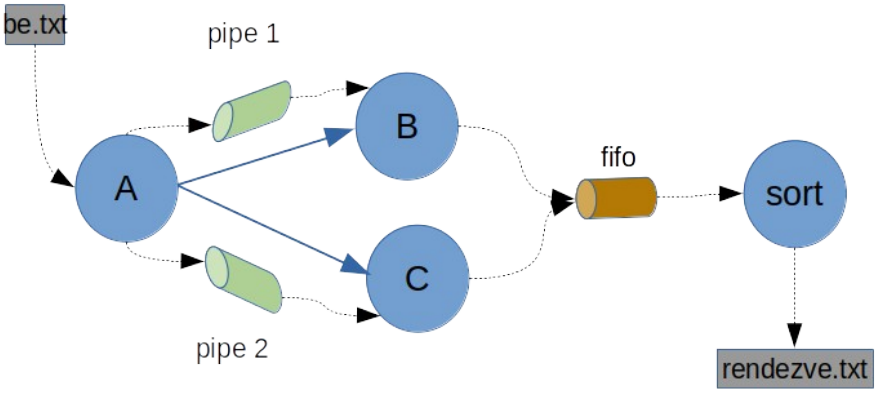
paranccsal olvasható ki a fejléc fájl:

```
#define PIPE_BUF        4096    /* # bytes in atomic write to a pipe */
```

sorából.

Ezek a paraméterek valamilyen standardtól függhetnek (pl. POSIX), vagy egyszerűen egy adott kernel beállításaitól.

Oldjuk meg az alábbi feladatot (`split.c`, `sorter.c`):



A `be.txt` tartalma pl.:

```
11. 99.
33. 22.
56. 88.
66. 10.
77. 88.
```

vagy a `be_hosszu.txt` fájl.

A folyamat beolvas egy sort, az egyik számot **B**-nek, a másikat **C**-nek küldi át. B és C atomikusan írják át a számokat a `sort`-nak, mindig szövegsorrá alakítva át azokat. Elvileg ezek nem lesznek hosszabbak, mint 4096 byte, így nem keveredhetnek az írások a B és C folyamatból. A sort szempontjából viszont nincs jelentősége annak, hogy milyen sorrendben érkeznek.

Az **A** folyamat két szám átküldése után várjon 50 milliszekundumot.