4. gyakorlat: Az exec függvények

Feladatok

1. Az exec függvények

(Sablon: `feladat1.c`) Adott egy `szam.txt` szöveges állomány, amelyben számok vannak (minden sorban egy egész szám). Hajtsuk végre a `sort -rn szam.txt` programhívást:

- `execl`
- `execlp`
- `execv`
- `execvp`

hívások segítségével.

2. Parancsértelmező

Módosítsuk az `execvp1.c` példát úgy, hogy egy minimális parancsértelmező programot kapjunk, amelyben az apa folyamat:

– kiír egy készenléti jelet: `ok>`
– beolvas egy parancssort, például: `ls -l /` vagy `cat -A a.txt`
– előállítja a parancssorhoz szükséges sztringeket vagy az `argv` tömböt
– elindít egy fiút amely `exec()`-el végrehajtja a parancsot – válasszunk megfelelő `exec()` függvényt
– vár amíg a fiú végez
– ismét készenléti jelet ír ki

Addig végzi ezt, amíg beütjük az `exit` parancssort.

3. Fájl azonosító duplikálása

Hajtsuk végre C programból a:

```
tail -2 a.txt > b.txt
```

shell parancsot.

a. a `system()` függvény segítségével,
b. az `execlp()` függvény és a `dup2()` függvény segítségével.

A `dup2()` leírását lásd alább (részletesen a `man 2 dup` leírásban), vagy az 5_pipe.pdf labor anyag 3. pontjában.

Egy már létező állományleírót duplikálni lehet az alábbi függvénnyel. A művelet során egy második azonosító jön létre, amely ugyanazt az állományt „kapcsolja” a folyamathoz mint az első, de más azonosító számmal:

```
    #include <unistd.h>

     int dup2 ( int regi_fd, int uj_fd);

Vissza: az új állományleíró értékét adja, -1 ha hiba van.
```

Mivel a standard input állományleírója az 0-s, amelyet szimbólummal is megadhatunk, mint: `STDIN_FILENO` (ennek a fejlécekben definiált értéke általában 0), az alábbi hívás:

```
int fd, fd1;
// fd-t megnyitjuk olvasásra
if ( (fd = open ("a.txt", O_RDONLY)) == -1 ){
      syserr("open");
}
//
fd1 = dup2 (fd, STDIN_FILENO);
```

Az `fd1`-ben az `STDIN_FILENO`, azaz 0 értéket ad vissza, ugyanakkor az addig csak `fd`-vel azonosított állományhoz az `STDIN_FILENO` azonosítót rendeli. Így amennyiben a továbbiakban a `STDIN_FILENO`-ról olvasunk, akkor az `a.txt`-ből fogunk olvasni.

A b. megoldásához a `b.txt` állományt meg kell nyitni az `execlp()` hívás előtt, és a kapott azonosítót duplikálni kell úgy, hogy a duplikált azonosító száma a standard kimenet legyen. Így az elindított `tail -2 a.txt` parancs a standard kimenet helyett a `b.txt`-be fog írni, mert örökli annak megnyitott azonosítóját, azzal az azonosító számmal ami számára a standard kimenet.
A programban a következő beállítás jön létre a fájl (`a.txt`) megnyitása:

```
int fd;
if ((fd=open("b.txt",O_WRONLY | O_CREAT | O_TRUNC , 0644))<0){
      syserr("open");
}
```

után, ha az `a.txt` megnyitott fájlra mutató azonosító neve `fd`:

`STDOUT_FILENO  --> Terminál`
`fd              --> b.txt`

a `dup2()` hívása:

```
if (dup2(fd,STDOUT_FILENO)<0)
    syserr("dup2");
```

után a beállítás a következő lesz:

```
STDOUT_FILENO  --> b.txt
fd             --> b.txt
```

A `dup2()` az átállítás előtt lezárja a fájlt (terminál) amire az `STDOUT_FILENO` mutat.

Az `fd` azonosítót akár le is lehet zárni az `exec()` előtt. Mivel a `tail` a standard kimenetre ír, amit kiír a `b.txt` fájlba fog kerülni. Elvileg a `dup2()` után az `fd`-t és `STDOUT_FILENO-t` alternatív lehet használni.

c. Mi történik, ha a működő b. megoldásban közvetlenül az `execlp()` előtt lezárjuk a `b.txt` duplikált azonosítóját (a standard kimenetet), szimulálva ezzel az azonosító `close-on-exec` módosító opciójának beállítását (lásd `dup3()` vagy `fcntl()` leírásokban)? Próbáljuk ki.

d. Valósítsuk meg az alábbi parancssor lefutását C programból, `fork()`és `execlp()` hívásokat alkalmazva:

```
tail -2 < a.txt > c.txt
```

4. exec-el indított program helyes lefutásának ellenőrzése

Írjunk egy C programot amelyik az alábbit végzi:

– végrehajtja az `ls a.txt` parancsot
– ha ez sikeres, tehát `a.txt` létezik, végrehajtja a `cat a.txt` parancsot.

5. A system függvény használata

Írjunk egy parancsértelmezőt amelyik `system()` függvényt használva hajt végre összetett shell parancsokat (pl. olyant is amiben csővezeték van: `ls | egrep '*.txt'`). Tanulmányozzuk a `system()` függvényt (`man 3 system`), és a parancssor lefutása után írjunk ki mindent amit megtudunk a lefutott parancssorról! A `status` érték kezelését lásd a 3. labor segédletében. Írjunk egy külön fájlban található, máshol is felhasználható függvényt:

```
void printstatus (int status, pid_t pid);
```

amelyik mindent kiír a `wait()` vagy `system()` által visszaadott kódról!

6. system függvény írása

Írjunk egy, a `system()` függvényhez hasonló saját függvényt. Ez megoldható, ha felhasználjuk a `bash -c` opcióját:

```
bash -c  "parancs"
```

indítással bármilyen `bash` által végrehajtható parancssort el tudunk indítani, pl:

```
bash -c  "cat a.txt | sort | uniq | head -1"
```

7. wait és dup2 feladat

Egy C programmal (`getwords.c`) oldjuk meg az alábbi két parancsnak megfelelő futtatást (generálnak egy random listát 11 karakteres angol szavakból):

```
egrep  "^.{9}[^'].$" /usr/share/dict/words > ki.txt
shuf < ki.txt
```

A program egy fiú folyamatot indít, abban fut az első sor. Után várja meg a fiúfolyamat kilépését `wait()`-el, ellenőrizze, hogy hiba nélkül futott le a fiú és a `ki.txt` fájl nem üres (ha mindkét feltétel teljesül) csak akkor indítsa el a `shuf` parancsot saját kódja helyett. Az üres fájl ellenőrzést megtehetjük az `lseek()` függvénnyel.

Ha működik, oldjuk meg, hogy 11 karakteres szavak helyett bármilyen hosszat keressen:
`./getwords 8`
adjon vissza 8 karakteres szavakat. Ha nincs olyan hosszú szó mint amilyent keresünk (`ki.txt` üres), akkor a standard hibakimenetre írja ki: *Not found*, a standard kimenetre ne írjon csak egy új sort.

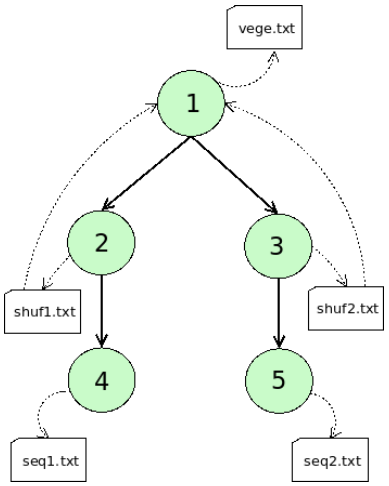
8. sort és tail

Írjunk egy C programot ami az alábbi szekvenciát valósítja meg:

1. Egy apa a folyamat elindítja egy fiú folyamatban a `sort` parancsot, ennek a `sablonok` könyvtárban található `szam.txt` fájlt kell elrendeznie numerikusan fordított sorrendben, a kimenet egy fájlba kell kerüljön, ennek neve `ki.txt`. Használjuk a sort `-o` opcióját a kimeneti fájl megadására.
2. Az apa folyamat megvárja a `sort`-ot, és ha az kilépett, ellenőrzi, hogy a sort hiba nélkül futott (`WIFEXITED(status) && WEXITSTATUS(status)==0`).
3. Ha létrejött, akkor kicseréli a saját kódját a `head` parancs kódjával úgy, hogy a terminálra íródjon a legnagyobb szám.

9. seq-shuf feladat

Írjunk egy programot ami megvalósítja az alábbi feladatot (`seqshuf.c`):



1. folyamatnak két fia van, 2. és 3.; 2. és 3. folyamatnak egy-egy fia van (4. és 5.).

A programot így indítjuk:

```
seqshuf N
```

ahol N egy egész szám. pl. 6:

```
$ ./seqshuf 6
```

Amennyiben N=6, a folyamatok az alábbiakat végzik:

- 4. folyamat kódjában a `seq` parancs fut, az alábbi alakban (annyi számot generál amennyi a program paramétere): `seq 1 6 > seq1.txt`
- 5. folyamatban ugyanaz a parancs fut, a kimeneti fájl `seq2.txt`.
- 2. és 3. folyamat megvárják fiaikat, majd a `shuf` parancsot futtatják, 2. folyamat: `shuf seq1.txt > shuf1.txt` alakban, 3. folyamat `shuf seq2.txt > shuf2.txt` alakban.
- 1. folyamat megvárja fiait, majd a `sort` parancsot futtatja: `sort shuf1.txt shuf2.txt -o vege.txt`

A megoldás jobb, ha észrevesszük, hogy 2.-4. és 3.-5. ugyanazt a feladatot végzik, és így írjuk meg a kódot.