Az exec() függvények

Tartalom

1. Az exec() függvények

A fiú folyamatok indítása után az új folyamat kódja kicserélhető a memóriában egy más kóddal, így valósul meg egyik folyamatnak a másik által való indítása.
Új kódot egy folyamatból az `exec()` függvénycsalád valamelyik tagjával indítunk. Ha egy folyamat meghívja az `exec()` függvények valamelyikét, akkor a folyamat memóriaképét teljesen felülírja az `exec()` által elindított új program. Mivel elágazást a Unixban a `fork()` függvény hozhat létre , amikor az egyik folyamat új folyamatot indít, az első lépés egy `fork()` hívás (hacsak nem a saját kódját akarja kicserélni egy folyamat, ami szintén lehet használati eset).

A `fork()` után az `exec()` család egyik függvénye kerül meghívásra, ezek paraméterként az elindítandó program nevét, illetve argumentumait és környezeti változóit kapják. A programindítás során a következők történnek:

– a futó folyamat kódját helyettesíti az új program,
– a memóriába megmaradnak a futó folyamat kernel által nyilvántartott adatai, a program kód és adatrészeit törli a kernel és helyettük betölti az új program kódját és adatait a lemezről,
– az új program a régitől örökli a folyamatazonosítót (a fiúét),
– a felhasználó azonosítóját amely alatt fut (*user id*),
– a terminált amelyikhez csatlakozik (kontrolláló terminál),
– a munka könyvtárat,
– a már kapott, de még nem kiszolgált jelzéseket,
és sok más tulajdonságot.

Amennyiben az indított program nem bináris program, hanem egy szöveges állomány amelyet valamilyen értelmező programnak kell végrehajtani (például egy `bash` vagy egy `awk` szkript) akkor a kernel a parancsértelmezőt (`bash` vagy más shell) vagy a kért programot (pl. `awk`) tölti be, és annak adja át argumentumként a szkriptet.

A programindító folyamat által már megnyitott állományok kezelése hasonló a `fork()`-hoz, tehát örökli őket az új program és ez az implicit viselkedési mód. Minden állományhoz tartozik egy `close-on-exec` nevű állapotjelző, amelyet az `fcntl()` függvénnyel lehet állítani (lásd `man 2 fcntl`). Ha ezt beállítjuk, akkor az `exec()` során az állományazonosítók lezárulnak (példa Kerrisk *Process creation* fejezetében).

Sikeres hívás esetén az `exec()` függvények nem térnek vissza, az új program kódja fog futni helyettük.

Az `exec()` függvények az alábbiak:

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

int execl( const char *pathname, const char *arg0, ..., (char*)NULL);


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

int execle( const char *pathname, const char *arg0, ..., (char*)NULL,
            char *const envp[]);

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

int execlp( const char *filename, const char *arg0, ..., (char*)NULL);


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

Visszatérített érték: –1 ha hiba van, egyébként nem térnek vissza.

A változó argumentumlistákban csak a pontok szerepelnek a lista végén, és jelzik, hogy az argumentumlisták egy 0 értékű mutatóval végződnek ott, ahol mutatólista van.

A függvények megkapják az indítandó program nevét. Ez kétféleképpen lehetséges: vagy egy teljes könyvtár útvonalat kapnak (*pathname, pl:* `/usr/bin/sort`), ilyenkor megkeresik az állományrendszerben az illető futtatható állományt, vagy pedig egy állomány nevet kapnak (*filename, pl.:* `sort`), és ilyenkor amennyiben az állománynév nem kezdődik `/` karakterrel, megkeresik az állományt azokban a könyvtárakban, amelyek fel vannak sorolva a `PATH` környezeti változóban, például:

```
PATH=/bin:/usr/bin:/usr/local/bin:$HOME
```

Ezen kívül a paraméterek közt szerepel az induló program argumentumlistája. Az `argv[]` tartalmát vagy tömb (vektor, ez pont olyan mint a `main()` függvény által kapott `char * argv[]`) vagy pedig mutatólistában lehet átadni (ilyenkor megadjuk minden argumentum sztring mutatóját). Amennyiben mutatólistát használunk, a lista utolsó eleme egy `NULL` értékű mutató.

Hasonlóan, bizonyos `exec()` függvények megkaphatják egy tömbben a környezeti változókat. Egyébként az új folyamat örökli őket.

A függvény neveket az alábbi szavak kezdőbetűiből lehet megjegyezni, ezek előfordulnak a függvény nevekben:

l = list (az argv tartalma lista formájában jön),

v = vektor (az argv paraméter vektor, a C-ben tömb formában van),
e = environment – környezeti változók listája is jön,
p = keresd a programot a PATH által meghatározott úton, ha / -el kezdődik akkor az állományrendszerben.

Az említett függvények közül csak az `execve()` rendszerhívás. Ennek mindkét listát tartalmazó argumentuma vektor. A többi függvény standard C könyvtári függvény, amelyek a kapott argumentumokat vektorrá alakítják és utána meghívják az `execve()`-t.

2. Példaprogramok exec() függvényekkel

2.1. Az execl() függvény

```
int execl( const char * pathname, const char *arg0, ..., (char*)0);
```

A függvényhívás paramétereiben egyszerűen felsoroljuk a main függvény `char * argv[]` argumentumait egy listában. Amint említettük, siker esetén az `exec()` nem tér vissza. Figyeljük meg, hogy a listában a hívott program neve kétszer szerepel: egyszer mint a kódot tartalmazó állomány neve, egyszer pedig mint a `main` első paramétere.
Példa: `execl.c`

```c
#include "myinclude.h"

int main(int argc, char* argv[]) {
    pid_t pid;   // folyamat azonosító
    int status;  // kilépési állapot tárolására

    pid = fork();  // új folyamat
    if (pid < 0) {
        syserr("fork");
    }

    if (pid == 0) {
        // fiú folyamat, új programot indítunk
        //  az ls -l parancsot indítjuk
        // a /bin/ls azért szerepel kétszer, mert az első sztring
        // az indítandó programkód helye, a második a parancssor első argumentuma
        execlp("ls", "ls", "-l", (char*)NULL);
        // ide már nem tér vissza
        // ha mégis, akkor hiba
        syserr("execl");
    } else {
        // apa folyamat megvárja a fiút
        wait(&status);
    }
    exit(EXIT_SUCCESS);
}
```

2.2. Az execv() függvény

Ennek a függvénynek egy vektorban kell megadni a `main()` argumentumát. Egy olyan példát mutatunk be, ahol egy parancssorban kapjuk meg a program nevét és argumentumait, és mielőtt meghívnánk az `execv()`-t, elő kell állítani a `char * argv[]` vektort.

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

Argumentumlista vektort az alábbiak szerint deklarálunk C-ben:

```
char * argumentumok[] = {"ls", "-l", "/", NULL};
```

A valóságban azonban a sztringeket többnyire egy karaktersorban kapjuk, valahogy így:
`"/bin/ls -l /"`
és ezt át kell alakítani vektorrá, amint az alábbi függvény teszi. Figyeljük meg, hogy a program nevét teljes útvonallal kell megadni, mert azt az állományrendszerből fogja egyenesen kiolvasni, nem a `PATH` változó útvonalaiból. A függvényt és használatát az `execvp.c` példaprogramban találjuk meg:

```
// a parancssort feldolgozó és execvp()-vel elindító függvény
// bemenet: parancssor sztring, pl:
//    "ls -l /home"
// kimenet: hibakód, ha nem hajtódik végre a parancs
// az execvp() nem tér vissza, ha sikerült
int parancssor(const char *parancs) {
    const char *elvalaszto = " ";
    int argc = 0;
    char **argv = NULL; /*ebben lesz az argv*/
    char *masolat = NULL;
    char *s;

    /* mivel a strtok() tönkreteszi a sztringet amin dolgozik,
     * egy másolatot kell készíteni róla */
    masolat = (char *)malloc(strlen(parancs) + 1);
    if (masolat == NULL)
        return -1;
    strcpy(masolat, parancs);
    argc = 0;
    s = strtok(masolat, elvalaszto);
    while (s != NULL) {
        /* van még egy paraméter, megnövelem az argv[]-t */
        argv = (char **)realloc(argv, (size_t)(argc + 1) * (sizeof(*argv)));
        /* helyet foglalok a következő stringnek */
        argv[argc] = (char *)malloc((size_t)strlen(s) + 1);

        strcpy(argv[argc], s);
        argc++;
        s = strtok(NULL, elvalaszto);
        /* a második hívásnál NULL az első paraméter */
    }
    /* amikor befejezzük, az utolsó mutatónál vagyunk, még be kell írni egyet */
    argv = (char **)realloc(argv, (size_t)(argc + 1) * (sizeof(*argv)));
    argv[argc] = NULL;
    /* itt megvan az argv */
    execvp(argv[0], argv);
    /* ide akkor jutunk ha nem sikerült */
    /* felszabditjuk a memóriat */
    while (--argc > 0)
        free(argv[argc]);
    free(masolat);
    return EXIT_FAILURE;
}
```

A függvényben használjuk a `strtok()` függvényt:

```
#include <string.h>
char *strtok(char *str, const char *delim);
```

Vissza: karakter mutató, ha vannak még darabok (*token*), `NULL` ha elfogytak.

A függvény feldarabolja több egymás utáni híváson keresztül az `str` sztringet. Az első hívásnál megadjuk az `str` sztringet, a következőknél pedig `NULL` mutatót adunk helyette. A `delim` sztring azokat a karaktereket tartalmazza, amelyeket elválasztónak tekintünk. Amennyiben az `strtok()` `NULL`-al tér vissza, a sztring darabjai elfogytak. Az `strtok()` felülírja az `str` sztringet, ezért mindig egy másolatot kell neki megadni paraméterként.

A `strtok()` helyett lehet még használni ugyanerre a feladatra a `strtok_r()` vagy a `strsep()` függvényt.

2.3. Az execvp() függvény

Az alábbi példa illusztrálja az `execvp()` használatát. A függvénynek az `argv` argumentumot vektorban kell megadni, és elegendő a program nevét megadni, mert az állományt a `PATH` szerinti könyvtárakban fogja keresni.

Példa: `spawn.c`

```
/*
 * fork és exec együtt = spawn
 */

#include "myinclude.h"

// ez a függvény elvégzi a fork() és execvp() hívást
// a fiú folyamatban ha sikerül nem tér vissza
// a fiú folyamatban ha NEM sikerül hibakóddal tér vissza
// az apa folyamatban visszatér a fiú PID-jével
int spawn(char* prognev, char** argumentumok) {
    pid_t fiu_pid;
    fiu_pid = fork();  // elindítja a fiú folyamatot
    if (fiu_pid != 0) {
        // az apa itt befejezi
        return fiu_pid;
    } else {
        // ez a fiú folyamat
        // az execvp() betölti az új kódot a fiú memóriájába
        execvp(prognev, argumentumok);
        // ide nem szabad eljutni, csak ha hiba történt, és execvp()
        // nem volt végrehajtható
        syserr("execvp");
    }
}

int main() {
    pid_t pid;
    int status;

    // az argumentumok sztring listával vannak megadva, tehát az
```

```
    // argv tömb elő van készítve: nem kell rajta semmit alakítani
    char* argumentumok[] = {"ls", "-l", "/home/", (char*)NULL};

    // a spawn függvény elindítja a fiút és az apa ág visszatér ugyanide
    // tehát itt az apa fut tovább a spawn után
    // a fiú folyamat vagy fut, vagy hibával kilép
    pid = spawn("ls", argumentumok);

    wait(&status);  // megvárja a fiút

    if (pid > 0) {
        printf("\nVége, a fiu PID=%d volt.\n", pid);
    } else {
        printf("fiú indítása nem sikerült\n");
    }

    exit(EXIT_SUCCESS);
}
```

3. A system() függvény

```
#include <stdlib.h>

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

Vissza: -1 hiba esetén, egyébként a végrehajtott parancsot futtató fiú folyamat kilépési kódja.

A `system()` függvény egy teljes parancssort kap argumentumként, például:

```
"ls -l /"
```

és elindít egy új shellt mintha az alábbi parancsot adnánk ki a parancssoron:

```
bash -c "ls -l /"
```

A `system()` nem rendszerfüggvény, hanem a standard könyvtári függvény, és a következő hívást hajtja végre:

```
execl("/bin/sh", "sh", "-c", "ls -l /", (char *) NULL);
```

ahol `sh` a standard shell, és `"ls -l /"` a végrehajtott parancssor.
Példahívás:

A `system()` függvény a `fork()`, `waitpid()` és `exec()` függvényeket alkalmazza a parancssor végrehajtására, tehát megvárja az elindított fiú folyamatot.

4. A dup() függvények: fájlok átirányítása

Hajtsuk végre C programból a:

```
$ tail -1 a.txt > b.txt
```

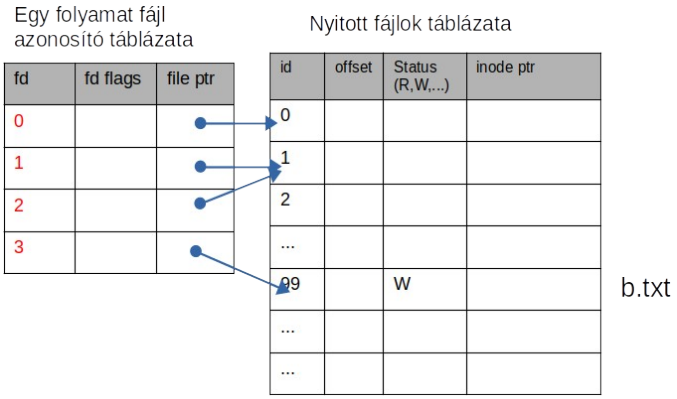
shell parancsot.

Alapértelmezetten a `tail` a standard kimenetre ír, de ez átirányítható egy egy más írható fájlba.

A kernel egy nagy táblázatban tartja nyílván a rendszer szinten megnyitott fájlokat, ennek neve nyitott fájlok táblázata. A shell szintjén a `lsof` paranccsal listázhatunk erről egy kimenetet, pl. az alábbi parancs az `sdiak` felhasználó nyitott fájljait listázza:

```
$ lsof -u sdiak
```

Ha a `tail` indítása előtt megnyitjuk írásra `b.txt` fájlt, a folyamat leíró blokkban a kernelben a fájl azonosítók táblázata a következő lesz:



Egy már létező állományleírót duplikálni lehet, ez azt jelenti, hogy létrehozok egy új azonosítót, amelyik ugyanazt a fájlt azonosítja. Ezt egy `dup()` nevű függvény valósítja meg, az új azonosító pedig a legkisebb fel nem használt azonosító szám lesz.

Ennek e műveletnek egy másik változata (ezt az alábbi `dup2()` függvény végzi), amikor megadom, hogy milyen érték legyen a második azonosító, és ez akár egy használatban levő, más fájlra mutató azonosító is lehet.

A művelet lefutása után a második azonosító ugyanarra a fájlra fog mutatni mint az első. Amennyiben a `uj_fd` már mutat egy megnyitott fájlra, azt a kernel le fogja zárni a duplikálás előtt.

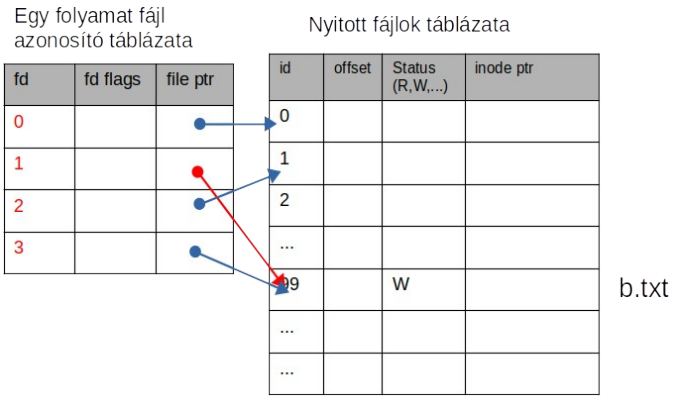
```
#include <unistd.h>

int dup2 ( int regi_fd, int uj_fd);
```

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

A 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.
Példa: `dup.c`

```
int fd;
if ((fd=open("b.txt",O_WRONLY | O_CREAT | O_TRUNC , 0644)) == -1){
        syserr("open");
}
if (dup2(fd,STDOUT_FILENO) == -1){
    syserr("dup2");
}
```

A `STDOUT_FILENO` a standard kimenet fájl azonosítója. A kernelben a következő módosulás történik a fájl azonosítók táblázatában:



Így az `exec()`-el elindított `tail -1 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.
Hasonló módon, a standard bemenet átirányítását is megoldhatjuk, pl. az alábbi parancssor futtatásához:

```
$ tail -1 < a.txt > b.txt
```

A következőt kell tenni az `exec()` előtt:

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

5. *Példaprogramok*

A példaprogramok a **`lab4_exec`** könyvtárban találhatóak:

`execl.c`	Az `execl()` használata.
`execvp.c, execvp1.c`	Az `execvp()` használata.
`spawn.c`	`fork()` és `execvp()` végrehajtó függvény
`parancssor.c`	Parancssort vektorrá alakító függvény.
`system.c`	A `system()` függvény használata.
`dup.c`	A `dup2()` függvény használata.