Dr. Varga Imre

Rendszerközeli programozás

Oktatási segédlet mérnökinformatikus hallgatók részére

Debreceni Egyetem

Informatikai Kar

2019

Legutóbbi frissítés: 2019.02.03.

Tartalomjegyzék

A tananyag elkészítését az **EFOP-3.4.3-16-2016-00021** számú projekt támogatta. A projekt az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósult meg.

A program futási környezete

Ebben a fejezetben egy összegzés található arról, hogy a C nyelven írt program és a futási környezete hogyan cserélhet információt. Ennek a következő lehetőségei kerülnek említésre:

- parancssori argumentum
- program visszatérési érték
- input/output/error felületek átirányítása
- környezeti változók
- operációs rendszer detektálás
- program/parancs futtatás programon belül
- időkezelés
- egyéb lehetőségek

Parancssori argumentum

Egy program indításakor adhatunk át neki információt a parancsértelmezőben. Ezek a parancssori argumentumok, amelyek a főprogram (`main`), mint függvény paramétereiként jelennek meg a forráskódban. A főprogram általános definíciója, mint ismert, a következő alakú:

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

Itt `argc` a parancssori argumentumok aktuális száma, `argv` pedig a parancssori argumentumként megadott sztringek tömbje. Az `argc` minimális értéke 1, mivel az első parancssori argumentum nem más, mint a futtatott program neve. Az `argc` értéke nem más, mint a futtatáshoz kiadott parancsban szereplő whitespace karakterrel elválasztott karaktersorozatok száma, míg az `argv` tömb elemei `char*` mutatók az említett elválasztott karaktersorozatok kezdetére. Az `argv` tömb `argc` számú elemet tartalmaz. Nézzük az alábbi példát!

```
./a.out alma 567 –f 8.9
```

A parancssori argumentumok száma (`argc`) itt 5. Az első parancssori argumentum (`argv[0]`) a `"./a.out"` sztring, a második parancssori argumentum (`argv[1]`) a `"alma"` sztring, a harmadik parancssori argumentum (`argv[2]`) a `"567"` sztring, és így tovább. Fontos hangsúlyozni, hogy az egyes argumentumok mindig sztringek. Például a `567` a fenti parancsban nem egy egész szám, hanem a `"567"` karaktersorozatként jelenik meg a programban. Így például az `argv[2][1]` jelenti a `'6'` karaktert. Gyakran előforduló hiba az, hogy kezdő programozó a második argumentum meglétét tévesen az `argv[1]==NULL` kifejezéssel vizsgálja, a helyes `argc>=2` kifejezés helyett.

Program visszatérési érték

C nyelvű programban a főprogram egy függvény, azaz visszatérési értékkel rendelkezik. Ennek típusa `int`. A főprogram által visszaadott érték a hívóhoz, azaz legtöbb esetben az operációs rendszer parancsértelmezőjéhez kerül. Linux rendszerben ez a visszaadott érték (vagyis inkább ennek legkisebb helyiértékű bájtja) egy speciális környezeti változóba kerül, melynek neve: `?` (kérdőjel). Emiatt a legutóbb lefutott program/parancs által visszaadott értéket parancssorban az `"echo $?"` paranccsal jeleníthetjük meg. Általánosan elfogadott gyakorlat, hogy ha egy program a tervek szerint, sikeresen ér véget 0 értéket ad vissza, míg ha valamilyen oknál fogva működése sikertelen, akkor egy nullától eltérő egész számmal (mint egyfajta hibakód) tér vissza.

Az input/output/error felületek átirányítása

A `printf` könyvtári függvénnyel az alapértelmezett kimenetre (általában képernyőre) írhatunk, míg a `scanf` függvénnyel az alapértelmezett bemenetről (általában billentyűzetről) olvashatunk be. Van azonban C programokban 3 előre definiált azonosító, amelyek egy-egy `FILE*` típusú mindig nyitott állományokra mutat. Ezek az `stdin` (alapértelmezett bemenet), az `stdout` (alapértelmezett kimenet) valamint az `stderr` (alapértelmezett hiba felület, általában szintén a

képernyő). Így tehát `printf(…)` helyett bármikor használhatjuk az `fprintf(stdout, …)` kifejezést. „Kulturált” programozó a program hibaüzeneteit az `fprintf(stderr, …)` hívással íratja ki. Ezek a felületek az operációs rendszerekben I/O átirányítással természetesen bármikor (akár külön-külön, akár együtt) átirányíthatóak (más fájlokba, egymásba, nyomtatóra, stb.). Azt is könnyen elérhetjük egy csővezeték (pipe) segítségével, hogy az egyik program outputja közvetlenül egy másik program inputjára kerüljön, azaz így kommunikálhassanak.

Példák:

`./a.out > out.txt`	A program kimenete az out.txt fájlba kerül (felülírva a régi tartalmat).
`./a.out >> out.txt`	A program outputja az out.txt fájlba végére kerül (hozzáfűzés).
`./a.out 2> error.txt`	A program hiba kimenete (nem output) az error.txt fájlba kerül.
`./a.out 1>x.txt 2>&1`	A kimenet a fájlba, hiba oda, ahol a kimenet van (ugyanabba a fájlba).
`./a.out &> x.txt`	Mind a kimenet, mind a hiba az x.txt fájlba kerül.
`./a.out < in.txt`	Az a.out program a szükséges adatokat az in.txt fájlból olvassa be.
`./prog1 | ./prog2`	A prog1 által előállított kimenetet a prog2 olvassa be.

Környezeti változók

A környezeti változók az operációs rendszer által, a programjainkon kívül tárolt szöveges információkat tartalmaznak a rendszerről. Ezekhez a programjainkon belül hozzáférhetünk, módosíthatjuk őket. A `getenv` nevű `stdlib.h` headerben definiált függvény környezeti változó értékének lekérdezésére szolgál. Egyetlen paramétere egy sztring, amely a környezeti változó nevét tartalmazza. A függvény egy `char*` mutatóval tér vissza, amely által mutatott címen a környezeti változó értéke (ami mindig sztring) található. A `putenv` nevű alprogrammal pedig egy környezeti változó értékét módosíthatjuk/állíthatjuk be. Egyelten paramétere egy sztring, amelyben először a módosítani kívánt környezeti változó neve szerepel, majd egy egyenlőség karakter (`'='`), végül az új érték. Például: `putenv("PATH=/home/hallgato");`. Fontos megjegyezni, hogy a programunkkal létrehozott változások lokálisak, azaz csak a programon belül érhetőek el.

Az egyes operációs rendszerekben különböző környezeti változók lehetnek, de általában elérhetőek a következő egyszerű és gyakran használt szimbólumok:

`PWD`	az aktuális munkakönyvtár elérési útja
`HOME`	az aktuális felhasználó alapértelmezett könyvtára
`PATH`	könyvtárak listája, ahol a parancsértelmező keresi a kiadott parancsokat
`LOGNAME`	az aktuális felhasználó bejelentkezési neve
`HOSTNAME`	a számítógép neve

Operációs rendszer detektálása

A program számára mindig elérhetőek olyan előre definiált azonosítók, amelyek az aktuális operációs rendszerről árulkodnak. Ilyen például Linux/Unix alatt a `linux`, `_linux`, `__linux__`, `__unix__`, Windows alatt a `_WIN32`, `_WINDOWS`, `__WINDOWS__`, DOS alatt az `MSDOS`, `_MSDOS`, `__MSDOS__`, MacOS alatt a `__MACH__`, `__APPLE__`.

Fordítás során fordítási direktívák segítségével definiálhatunk operációs rendszerfüggő nevesített konstanst, amellyel futásidőben rendszerspecifikus végrehajtást érhetünk el.

Például:

```
#if defined(__linux__)
  #define OS 1
```

```
#else
  #define OS 0
#endif
if(OS==1)
  /* Things to do in Linux system. */
else
  /* Things to do in other system. */
```

Programfuttatás programon belül

Időnként előfordul az a helyzet, hogy egy programban el szeretnénk végezni egy műveletet, amire már létezik program, viszont az újraimplementálás túl sok idő vesz igénybe, vagy túl bonyolult és egyébként sem praktikus. (Ha már kész van, miért dolgozzak?) Ilyenkor a programunk végrehajtásának egy adott pontján fel kell függesztenünk a végrehajtást, megkérni az operációs rendszert, hogy futtassa le a kívánt programot/parancsot és aztán (ha kész) folytatnunk kell az eredeti programvégrehajtást. Erre a célra szolgál az `stdlib.h` header `system` függvénye. Ennek egyetlen paramétere egy sztring, amit megkap a parancsértelmező és megpróbál lefuttatni (ezalatt az eredeti folyamatunk „várakozó" állapotban van). Például a `system("ls *.bmp > pic.txt");` utasítás hatására (Linux rendszerben) az aktuális könyvtárban lévő bitmap képfájlok nevei bekerülnek egy „pic.txt" szöveges állományba. (Ezt a funkciót C nyelvű kóddal megírni kicsit összetettebb lenne.)

Időkezelés

A `time.h` header definiál egy speciális `time_t` típust az idő kezelésére, a rendszeridő lekérdezésére pedig a `time` függvényt használhatjuk. A `time` függvény egyetlen paramétere egy `time_t` típusú memóriacím, ide kerül a lekérdezett idő értéke pszeudo-címszerinti paraméterátadással. Az idő értékét visszatérési értékkel is megadja a függvény. Az idő adatokat 32 biten ábrázolja a rendszer az 1970. január 01. 00:00:00 óta eltelt másodpercek számaként. (A 2016.05.07 11:47:04 dátum/idő érték `1462621624`-ként tárolódik.) Így csak 1970 és 2038 közötti dátumokat tárolhatunk.

A `ctime` függvény segítségével ezt ember számára olvasható sztringgé alakíthatjuk az időt. A függvény egyetlen paramétere egy `time_t` típusú érték címe, a visszatérési érték pedig az eredmény karaktersorozatra mutató pointer. (A fenti példa dátumértéket "`Sat May  7 11:47:04 2016`" formában jeleníti meg.)

Van egy további lehetőség, amihez egy `struct tm` pointerre van szükség, amelynek értéket a `localtime` függvény adhat. Ha a függvény paramétere egy `time_t` típusú változó címe, akkor az abban tárolt időpont segítségével feltölti az idő struktúra mezőit. A rekord mezői az alábbiak:

```
struct tm {
   int tm_sec;         /* seconds,  range 0 to 59          */
   int tm_min;         /* minutes, range 0 to 59           */
   int tm_hour;        /* hours, range 0 to 23             */
   int tm_mday;        /* day of the month, range 1 to 31  */
   int tm_mon;         /* month, range 0 to 11             */
   int tm_year;        /* The number of years since 1900   */
   int tm_wday;        /* day of the week, range 0 to 6    */
   int tm_yday;        /* day in the year, range 0 to 365  */
   int tm_isdst;       /* daylight saving time             */
};
```

További hasznos időkezelő függvények: `gmtime`, `strftime`, `strptime`.

A program futtatása felfüggeszthető egész számú másodpercekre a `sleep` alprogram segítségével. A paraméterként megadott számú másodpercig a folyamat alvó állapotba kerül.

Egyéb lehetőségek

A számítógéprendszer egyéb paramétereiről, korlátozó tényezőiről, a jelen lévő további folyamatokról információt nyerhetünk ki a programjaink segítségével, ha kihasználjuk a `limits.h` header által biztosított lehetőségeket illetve hozzáférünk a `/proc` könyvtár speciális tartalmához (pl. `/proc/cpuinfo`, `/proc/meminfo`, stb.).

Irodalom:

- Neil Matthew, Richard Stones: *Beginning Linux Programming*, Third Edition (2004), 135-138, 142-145, 146-153. oldal
- Brian W. Kernighan, Dennis M. Ritchie: *A C programozási nyelv*, Műszaki kiadó (2008)

Véletlenszámok

A PC-k nem tudnak igazi véletlen számokat előállítani, mivel determinisztikus eszközök. Amivel ebben a szakaszban foglalkozni fogunk, az az ún. pszeudo-véletlen számok generálása. A továbbiakban mindig erről lesz szó, bár gyakran elhagyjuk a 'pszeudo' előtagot (lustaságból).

A véletlen számok előállítása mindig azt jelenti, hogy a számítógép egy kezdőértékből (mag) kiindulva előállít egy újabb számot, majd ez alapján generálja a következőt, és így tovább. Igazából mindig ugyanannak a nagyméretű, ciklikus számsorozatnak az elemeit generálja egy matematikai műveletsorozat (kongruens moduló generátor) segítségével, azonban ez egy olyan számsorozat, amelynek az egymás utáni elemei között az ember nem vesz észre összefüggést, azaz véletlennek tekinti.

C nyelvű programban az `stdlih.h` header `rand` nevű paraméter nélküli függvényét tudjuk erre a célra használni. A függvény minden egyes hívása során egy (az aktuális mag értéktől függő) egész számot állít elő `0` és `RAND_MAX` közötti zárt intervallumban. `RAND_MAX` szintén az `stdlib.h`-ban van definiálva megadja, hogy az adott rendszerben mi a legnagyobb előállítható véletlenszám. Fontos megjegyezni, hogy Linux rendszerben ez általában 2 147 483 647, míg Windows alatt sokszor csak 32767, ami jelentős különbség. (Például egymillió véletlenszám előállítása során Windows alatt durván 30-szor ismétlődik meg ugyanaz a számsorozat.) Nulla és `RAND_MAX` között mindegyik egész szám egyenlő valószínűséggel áll elő.

Minden hívás során a mag érték változik, de minden programindítás során ugyanarról a kezdőértékről indul. Emiatt a program mindig ugyanazt a „véletlen" számsorozatot állítaná elő. Ez elkerülhető, ha manuálisan megváltoztatjuk a program elején a mag kezdőértékét. Erre szolgál az `srand` eljárás, amely a paraméterként kapott egész számot állítja be kiinduló mag értéknek. Az `srand` eljárást elég egyszer meghívni a program elején ezzel inicializálva a generátor magját, ezt annyi `rand` hívás követhet, ahány véletlenszámra szükségünk van. Ha az `srand` paramétereként egy konstanst adunk meg, az ugyan megváltoztatja az alapértelmezett mag kezdőértéket, de mindig ugyanarra az értékre, azaz a programunk még mindig ugyanazt a számsorozatot álltja elő minden futtatásnál. Hogy ezt végleg elkerüljük érdemes olyan értékkel inicializálni a generátort, ami minden futás esetén más értéket vehet fel. Gyakran használjuk erre a célra az aktuális rendszeridőt vagy a program aktuális PID-jét.

Eddig csak egész típusú egyenletes eloszlású véletlenszámokat állítottunk elő [`0 ; RAND_MAX`] zárt intervallumban. Ha bármi mást szeretnénk, akkor azt kell valamilyen matematikai módszerrel áttranszformálni. Nézzünk néhány alapesetet (a teljesség igénye nélkül):

Egész számok egyenletes eloszlással [A ; B] zárt intervallumban:
```
int x = rand()%(B-A+1)+A;
```

Valós számok egyenletes eloszlással [A ; B] zárt intervallumban:
```
double y = (B-A)*(double)rand()/RAND_MAX+A;
```

P valószínűséggel (0%≤P≤100%) `X` érték, 1-P valószínűséggel `Y` érték:
```
float Q = ((double)rand()/RAND_MAX <= P) ? X : Y;
```

Több forrásból álló programok

Header állományok

Egy C nyelvű forráskódban számos alprogram, nevesített konstans, típus vagy egyéb programozói eszköz lehet definiálva a `main` függvényen kívül. Néha ezeket átláthatóság vagy újrafelhasználás céljából érdemes lehet kiszervezni egy külön állományba. Ezt egyszerűn megtehetjük. A C forráskód egyes részeit egyszerűen átmásolhatjuk egy „`.h`” kiterjesztésű ún. header állományba. Ezután már csak inkludálni kell azt az eredeti forráskódban. Ha például a header állomány neve `mydef.h`, akkor az eredeti forráskód elejére az alább utasítást kell írni:

```
#include ”mydef.h”
```

Ez egy előfordítónak szóló utasítás, ami arra kéri a compilert, hogy fordítás előtt másolja be az adott utasítás helyére a header fájl tartalmát és csak egy az egyesített forráskód forduljon le. Ebben az konkrét esetben a header fájlnak és a „`.c`” kiterjesztésű fájlnak ugyanabban a könyvtárban kell lennie (bár az inkludálásnál megadható általában elérési út is). A header fájl tartalmilag/formailag megegyezik egy sima C forrásfájllal, azzal a kivétellel, hogy nem tartalmaz `main` függvényt.

A fenti eset annyiban tér el például a szokásos `#include<stdio.h>` utasítástól, hogy utóbbi esetben a header állomány a fordító beépített állománya, ami egy a fordító által jól ismert könyvtárban van eltárolva.

A header állományok használatakor vigyázni kell arra nehogy azonos néven több dolgot is definiáljunk. Másik tipikus hiba, hogy egy headerben inkludálunk egy másikat, amit egy másik headerben vagy forráskódban is alkalmaztunk. Az ilyen helyzetek elkerülésére a feltételes fordítás lehetősége szolgál.

```
#ifndef printf
  #include<stdio.h>
#endif
```

Ha a fordító számára a `printf` azonosító nem definiált, akkor inkludálni kell az standard input/output headert, különben nem (mert már korábban volt inkludálva).

Tárgykódok összeszerkesztése

Egy másik lehetőség arra, hogy több forrásfájlból álló programot készítsünk az, ha a linkelés során kapcsoljuk össze a különböző forráskódokból származó tárgykódokat. A C fordító (pl. `gcc`) alap esetben egy „`.c`” kiterjesztésű forrásállományból (mint a nyelv fordítási egységéből) egy „`.o`” kiterjesztésű tárgykódú (object code) állományt állít elő. Ez önmagában még nem futtatható Ezután a kapcsolatszerkesztő program (linker, pl. `ld`) ezeket összeszerkeszti egyetlen futtatható állományba. A `gcc` külön kérés nélkül egyébként ezt is megcsinálja. Tehát ha van egy programunk, amelyik két forráskódból áll össze (`prog1.c`, `prog2.c`) és ebből egy `prog` nevű futtatható állomány szeretnénk generálni, akkor a `gcc prog1 prog2 –o prog` parancsot kell csak kiadnunk (így a tárgykódok és a linkelés megtörténik ugyan, de rejtve marad).

Irodalom:

- Brian W. Kernighan, Dennis M. Ritchie: *A C programozási nyelv*, Műszaki kiadó (2008)

Mutató-orientált lehetőségek

A mutatók

A mutató vagy pointer a C nyelvben egy olyan változó, amely egy másik változó memóriabeli címét tárolja. A pointer is egy változó tehát, amelynek a címkomponense által meghatározott helyen szereplő értéke nem más, mint egy másik memóriaterület címét reprezentáló bitsorozat. Egy speciális mutató érték a NULL, ami azt fejezi ki, hogy a pointer nem tartalmaz címet, nem mutat sehova. Egy mutató mindig egy adott típusú memóriaterület címére hivatkozik, amely típus a deklaráció során dől el.

```
int *p; // p egy egész típusú érték memóriaterületére mutat
float *x = NULL; // x képes tárolni egy float változó címét,
                    de egyelőre nem hivatkozik semmire
```

Egy változó címét a & karakterrel megjelenített, egyoperandusú címképző operátor segítségével kaphatjuk meg. Egy így előálló érték (memóriacím) értékül adható egy pointernek.

```
int A = 123;
int *P = &A; // a P pointer az A egész változóra mutat
```

Egy mutató által hivatkozott memóriacímen lévő értéket az egyoperandusú * operátor segítségével ismerhetjük meg, amelynek operandusa természetesen egy pointer.

```
int A = 321, B;
int *P = &A; // a P pointer az A egész változóra mutat
B = *P; // B megkapja a P által mutatott területen lévő értéket,
          azaz A és B értéke meg fog egyezni
```

A mutató értékek alacsony szinten előjel nélküli egészekként foghatóak fel. 32 bites rendszerekben a cím mindig 4 bájtos, míg 64 bites rendszerekben a pointerek mérete 8 bájt. A NULL speciális mutató érték a csupa nulla bitet tartalmazó bitsorozattal van implementálva. Ezek miatt a címekkel, a mutatók értékeivel aritmetikai műveletek is végrehajthatóak. (Jelentősége a tömbök kapcsán lesz.)

```
int A;
int *P = &A; // P az A-ra mutat
P++; // P mutasson az A változó melletti
       (azt követő) memóriaterületre
```

Mivel a mutatók is változók, ők is rendelkeznek címkomponenssel, azaz rájuk is mutathat pointer, ahogy az alábbi példában láthatjuk:

```
int A=987, B;
int *P = &A; // A P az A egészre mutat
int **P_2 == &P; // A P_2 a P egészekre mutató pointerre mutat
B = **P_2; // B értéke 987 lesz:
             a P_2 által mutatott memóriaterületen lévő cím (P)
             által mutatott területen lévő adat (A)
```

A tömbök és a mutatók kapcsolata

A tömbök a C nyelvben szorosan kapcsolódnak a mutatókhoz. Minden tömb neve egy nevesített konstans pointer érték, amely nem más, mint a tömb első elemének a címe. Emiatt az alábbi `if` utasításban szereplő kifejezés mindig igaz lesz:

```
int Tomb[10];
if (Tomb == &Tomb[0]) ;
```

A C nyelven emiatt a tömbindexelésre használt `[]` felfogható egyfajta címoperátornak. A `Tomb[5]` jelentése az, hogy a `Tomb` nevű tömb kezdőcíméhez képest 5 egységgel hátrébb megtalálható memóriaterületen szereplő érték. Ebből kifolyólag a Tomb[0] jelentése: a kezdőcímhez képest 0 egységgel hátrébb lévő érték, vagyis a tömb első eleme. Ez az oka annak, hogy a C a tömbök indexelését 0-val kezdi. Ebből kifolyólag az alábbi kódrészletben a tömb 3. elemére kétféleképpen is hivatkozhatunk: index operátor alkalmazásával illetve címaritmetikai kifejezés segítségével.

```
int A, B, C[10]={10,9,8,7,6,5,4,3,2,1};
A = C[2]; // a tömb 3. eleme (A==8)
B = *(C+2); // a tömb 3. eleme (B==8)
```

A változók (tömbök) memóriaterületének lefoglalása nem mindig a deklaráció felírási sorrendjében történik (fordító és operációs rendszer függő). Az `float A, B;` deklaráció hatására az A változó után közvetlenül nem biztos, hogy a B lesz. Viszont egy tömb elemei mindig szigorúan egymás után helyezkednek el a memóriában.

A pointerek és a tömbök kapcsolata természetesen a sztringek esetén is fontos. Azonban fel kell hívni a figyelmet arra, hogy például egy (karakter) tömb deklarációjakor lefoglalódik a megfelelő méretű memóriaterület, míg egy (karakter) mutató esetén csak a cím tárolására van lehetőség.

```
char S1[10]; // Hely 10 karakter számára
char *S2; // Hely egy cím számára
```

A többdimenziós tömböket a C a „tömbök tömbje” logika szerint kezeli, azaz végül is minden NxM-es mátrixot N*M elemet tartalmazó vektorra képez le sorfolytonosan. Egy `int T[10][20]` kétdimenziós tömb estén például a `T[4][3]` elemre hivatkozhatunk az alábbi kifejezéssel is: `*(T+4*20+3)`.

Pszeudocím-szerinti paraméterátadás

A pointerek használata a paraméterátadásnál is fontos lehetőséget biztosít a programozónak. A C nyelvben csak érték szerinti paraméterátadás használható, viszont pointert érték szerint átadni gyakorlatilag egy cím szerinti paraméterátadásra hasonlít. Ebben az esetben a paraméter nem csak input információt jelenthet az alprogram számára, hanem outputot is. Ezzel egy alprogram könnyen szolgáltathat a hívónak egynél több értéket is, mint az alábbi példában (ahol a `z` értéke 8 lesz):

```
void csere(int *a, int *b){ // változóértékek felcserélése
   int c;
   c=*a;
   *a=*b;
   *b=c;
   }
...
int x=4, y=32, z;
```

```
csere(&x,&y);
```

z = x/y;

A pszeudocím-szerinti paraméterátadás segítségével könnyen kezelhető például a tetszőleges méretű tömbök paraméterkénti átadása. Mivel C nyelvben nem ismerhetjük a paraméterként kapott tömb méretét, így azt egy külön paraméterben (érték szerinti) át kell adni.

```
double osszeg(double *T, int N){ // Kezdőcímtől N elem összege
   int i;
   double sum=0.0;
   for(i=0;i<N;i++)
      sum+=T[i];
   return sum;
   }
...
double total1,total2;
double A[8]={1.2, 2.3, 3.4, 4.5, 5.6, 6.7, 7.8, 8.9};
total1 = osszeg(A,8); // osszeg A[0]-tól A[7]-ig
total2 = osszeg(&A[2],3); // osszeg A[2]-tól A[4]-ig
```

Fontos megemlíteni, hogy ilyenkor a tömb értékei nem adódnak át, csak a kezdőcím. Azaz nincs lokális másolat a tömb elemeiről, közvetlenül a hívó memóriaterületén dolgozunk.

Irodalom:

- Brian W. Kernighan, Dennis M. Ritchie: A C programozási nyelv, Műszaki kiadó (2008) 107-128. oldal

Dinamikus memóriakezelés

Egy tömb memóriaterülete (az alapértelmezett dinamikus élettartam kezelésnek megfelelően) akkor foglalódik le, amikor aktiválódik az őt tartalmazó programegység és automatikusan akkor szabadul fel, amikor elhagyjuk ezt a programegységet. Közben (futásidőben) a tömb méretét nem tudjuk befolyásolni. A tömb mérete az ANSI szabvány szerint csak konstans kifejezéssel adható meg (bár egyes implementációk engedik a változóval történő megadást is). Mint az összes lokális változó, a tömbök is a veremszegmensben kerülnek tárolásra, azonban a verem mérete korlátos, tehát nem tudunk akármekkora tömböt használni (még akkor sem, ha a RAM kapacitása ezt engedné). A tömbök segítségével a folytonos reprezentációjú adatszerkezetek könnyen implementálhatóak.

Mindezek a tények erős korlátot jelentenek a programozó számára, ezért a C lehetőséget biztosít programozó által vezérelt élettartam kezelésű memóriaterületek használatára könyvtári függvények segítségével (`stdlib.h` esetleg `malloc.h` inkludálásával). A foglalások nem a veremben, hanem a heap-ben történnek, így a foglalásoknak csak fizikai korlátai vannak.

Az egyik legfontosabb alap függvény a `malloc`. Ez egy darab egész jellegű paramétert vár és egy ennyi bájt méretű egybefüggő memóriaterületet próbál meg lefoglalni a RAM-ban. Visszatérési értéke a lefoglalt memóriaterület kezdőcíme (vagy hiba esetén `NULL` érték). Ez a visszaadott érték `void*` típusú, amikor ezt eltároljuk egy pointer változóban érdemes cast-olást alkalmazni. Ha mondjuk 100 db int típusú érték tárolásához szükséges méretű helyre van szükségünk és nem vagyunk biztosak abban, hogy az adott típus, az adott rendszerben hány bájton ábrázolódik, akkor használhatjuk a `sizeof` operátort, azaz ezt írhatjuk:

```
int* p=(int*)malloc(100*sizeof(int));
```

Felhívnám a figyelmez arra, hogy a `sizeof`-ban szereplő `int` nem határozza meg a pointer típusát. Az így lefoglalt terület inicializálatlan, azaz memóriaszemetet tartalmaz. Mivel a tömbindexelés C-ben pointeraritmetikai operátort jelent, a lefoglalt területet ugyanúgy használhatjuk, mint egy hagyományos tömböt: `p[7]=-93;`

Amennyiben inicializált memóriaterületre van szükségünk, használhatjuk a `calloc` függvényt, amely a lefoglalt területen ’0’ biteket helyez el. (Ez felfogható fixpontos 0-nak vagy lebegőpontos 0.0-nak, vagy akár üres-sztring sorozatnak is.) Paraméterként a terület méretét máshogy kell megadni: először az ott tárolandó adatelemek számát majd ezek méretét bájtban. Siker esetén a visszatérési érték itt is egy `void*` memóriacím. Tehát a korábbihoz hasonló eset inicializálással így néz ki:

```
p=(int*)calloc(100,sizeof(int));
```

Előfordul az, hogy lefoglalunk egy adott méretű területet és aztán később (futásidőben) rájövünk arra, hogy a terület túl kicsi vagy esetleg túl nagy. Lehetőségünk van a lefoglalt terület átméretezésére, újrafoglalással. Erre való a `realloc` függvény. Első paramétere a már korábban lefoglalt terület címe, második pedig az új méret (bájtban). Ha a lefoglalt terület mérete nagyobb, mint a második aktuális paraméter értéke, akkor a terület végén lévő „felesleges” részt felszabadítja a rendszer (így későbbi foglalások számára az elérhető lesz). Ha növelni szeretnénk a lefoglalt terület méretét és közvetlenül utána van is még szabad hely, akkor abból kiegészítve megnöveli a foglalást. Amikor viszont növelés esetén közvetlenül a lefoglalt terület után nincs elegendő méretű szabad terület, akkor a rendszer a RAM-ban máshol keres megfelelő méretű területet, lefoglalja azt és átmásolja a korábbi memóriaterületen eddig tárolt adatokat az új, kibővített terület elejére. A lefoglalt terület tartalmazza tehát a korábbi adatoknak legalább az elejét, viszont bővítés során a terület inicializálatlan. Sikeres végrehajtás során a függvény az újrafoglalt terület címével tér vissza (`void*`). Ha mondjuk a korábbi példában bemutatott területről használat közben kiderül, hogy dupla annyi adatot szeretnénk ott tárolni akkor használhatjuk az alábbi hívást:

```
p=(int*)realloc(p,2*100*sizeof(int));
```

Figyelni kell arra, hogy ha bővítés során nincs elegendő hely és emiatt „költöztetni” kell az adatokat, az nagy mennyiségű adatmozgatást jelent a háttérben, így lassítja a programot.

A lefoglalt területek nem szabadulnak fel automatikusan, ha például elhagyjuk a foglalás programegységét, a terület akkor is foglalt állapotú lesz, csak esetleg nem tudunk rá hivatkozni, mert a címét tartalmazó (lokális) pointer szabadul fel. Így előbb utóbb kifogyhatunk a memóriából, ezért a dinamikusan (kézzel) foglalt memóriaterületeket (ha már nem kellenek) fel kell szabadítani! Erre szolgál a `free` eljárás. Egyetlen paramétere a felszabadítani kívánt memóriaterület címe: `free(p);`

Egy (inicializálatlan) memóriaterületet egyetlen lépés segítségével feltölthetjük bizonyos értékekkel. Erre használhatjuk a `string.h` header `memset` eljárása. Ennek első paramétere a beállítandó terület kezdőcíme. Második paraméterében megadott érték legkisebb helyiértékű bájtjának sorozatával tölti fel a rendszer a területet. A harmadik paraméter adja meg, hogy az előbbi értéket hányszor kell másolni a kitöltés során. Ezek alapján a

```
p=(int*)calloc(100,sizeof(int));
```

utasítás viselkedését tekintve egyenértékű ezzel a két utasítással (csak lassabb):

```
p=(int*)malloc(100*sizeof(int));
memset(p,0,100*sizeof(int));
```

Az adatszerkezetek szétszórt ábrázolása elképzelhetetlen dinamikus memóriahasználat nélkül. Ilyenkor a tárolási egységünk összetett típusú (rekord), általában tartalmaz egy adatrészt és pointert/pointereket más, azonos szerkezetű, futásidőben foglalt adategységekre. Például az alábbi kód egy három elemű tökéletesen kiegyensúlyozott, bináris fa létrehozását mutatja (a levél elemek adatrésze 0):

```
typdef struct elem{
  int adat;
  struct elem *bal, *jobb;
  } FaElem;
FaElem *gyoker = (FaElem*)calloc(1,sizeof(FaElem));
gyoker.adat=24;
gyoker.bal  = (FaElem*)calloc(1,sizeof(FaElem));
gyoker.jobb = (FaElem*)calloc(1,sizeof(FaElem));
```

Irodalom:

- Brian W. Kernighan, Dennis M. Ritchie: A C programozási nyelv, Műszaki kiadó (2008)

Bitműveletek

Az egész jellegű konstansok megadására C nyelven több lehetőség is van. Az a numerikus literál, amely csak számjegyeket és esetleg előjelet tartalmaz és nem ’0’ számjeggyel kezdődik, az 10-es számrendszerben értelmezendő. Ez a jól megszokott alap formája az egész konstansoknak. Ha egy egész konstans ’0’ számjeggyel kezdődik és azt további számjegyek követik (kivéve a ’8’-at és a ’9’-et), akkor az érték 8-as számrendszerben van megadva. Az az egész konstans, amely „0x” karakterekkel kezdődnek azok 16-os számrendszerben vannak megadva, azaz számjegyeket és ’A’-’F’ karaktereket is tartalmazhat. (A printf függvény formátumsztringjében a „%x” karaktersorozat lehetővé teszi az egész értékek hexadecimális formátumban történő kiíratását.) Ez a három különböző számrendszerben való megadás része az ANSI C-nek. Ezeken kívül egyes fordítók lehetővé teszik a bináris konstans megadást. Ebben az esetben a literál (esetleges előjel után) a „0b” előtaggal kezdődik és csak ’0’ vagy ’1’ számjegyeket tartalmazhat. Az alábbi példában mind a 4 változónak az értéke ugyanaz a bitsorozat:

```
int a, b, c d;
a = 14;
b = 016;
c = 0xE;
d = 0b1110;
```

Az C programozási nyelv 6 operátora segíti a bitekkel való műveleteket. Ezek közös jellemzője, hogy fix pontos számábrázolású (`int`, `char` és ezek módosulatai) operandusokon hajthatóak végre és az eredmény egyes bitjei az operandusok adott bitjeitől függenek. Vegyük sorra ezeket a bitenkénti operátorokat.

- `&` (bitenkénti ÉS) operátor: az operandusainak i. bitjei között logikai ÉS műveletet hajt végre és ennek eredménye lesz a végeredmény i. bitje. Egy eredménybit akkor és csak akkor lesz „1” ha mindkét operandusban az adott pozícióban „1” bit volt.
- `|` (bitenkénti VAGY) operátor: az operandusainak i. bitjei között logikai VAGY műveletet hajt végre és ennek eredménye lesz a végeredmény i. bitje. Egy eredménybit akkor és csak akkor lesz „0” ha mindkét operandusban az adott pozícióban „0” bit volt.
- `^` (bitenkénti XOR) operátor: az operandusainak i. bitjei között logikai KIZÁRÓ VAGY (XOR) műveletet hajt végre és ennek eredménye lesz a végeredmény i. bitje. Egy eredménybit akkor és csak akkor lesz „0” ha mindkét operandusban az adott pozícióban azonos bitek voltak.
- `~` (bitenkénti tagadás, egyes komplemens) operátor: egyetlen operandusának bitjeit negálja (ellenétesre állítja).
- `<<` (bitléptetés balra, SHIFT LEFT) operátor: a bal oldali operandusában szereplő biteket a jobb oldali operandus segítségével megadott számú pozícióval balra tolja (megőrizve a bitek sorrendjét). A jobb oldalon „megüresedő” bitpozíciókba ’0’ bitek kerülnek. Az N pozícióval balra tolás egyenértékű 2^N értékével való szorzással.
- `>>` (bitléptetés jobbra, SHIFT RIGHT) operátor: a bal oldali operandusában szereplő biteket a jobb oldali operandus segítségével megadott számú pozícióval jobbra tolja (megőrizve a bitek sorrendjét). A bal oldalon „megüresedő” bitpozíciók sorsa az ábrázolási módtól függ. Ha a bal oldali operandus előjelnélküli (unsigned), akkor balról ’0’ bitek kerülnek be az eredménybe, azaz logikai SHIFT műveletről beszélünk. Ha az első operandus előjeles, akkor az előjelbit (MSB) másolatai kerülnek be a megüresedő helyekre, azaz aritmetikai vagy előjeltartó SHIFT műveletről beszélünk.

  Az alábbi példában az összes változó értéke 48 lesz az értékadások után:

```
int a, b, c, d, e, f;
a = 48 & 51;
```

```
b = 32 | 16;
c = 33 ^ 17;
d = ~~48;
e = 96>>1;
f = 3<<4;
```

Nincs külön operátor arra, hogy egy adott érték egyik bitjére hivatkozzunk, kizárólag azt módosítsuk, vagy egyszerűen csak megtudjuk, hogy mi a bit értéke. Ilyen esetekben használjuk az ún. maszkolást, amikor egy olyan bitsorozatot/értéket adunk meg (ez a maszk), amely az általunk érdekesnek tartott pozíció(k)ban ’1’-es bitet tartalmaz az összes többiben ’0’-t. A vizsgált érték és a maszk között bitenkénti ÉS műveletet végrehajtva a számunkra lényegtelen bitek kitörlődnek. Ha például csak egy bitre vagyunk kíváncsiak egy több bájtos egész értékben, akkor a 2 megfelelő hatványa, mint egyetlen ’1’-es bitet tartalmazó érték megfelel a maszk szerepére. Ha a vizsgált értéket és ezt a maszkot összeÉSeljük és az eredmény 0*, akkor a vizsgált bit is ’0’, különben a vizsgált bit ’1’. (*Felhívnám a figyelmet arra, hogy a maszkolás (ÉS művelet) eredménye mindig egy több bites érték nem ’0’ vagy ’1’ bit, viszont ez a több bites érték minden pozícióban tartalmazhat ’0’ bitet, ami definíció szerint a 0 egész szám.) Példa:

```
// Mi a jobbrol 4. pozicioban levo bit erteke az x valtozoban?
int x = 4325265;
int maszk=0x00000008; // binaris …00001000
if(x&maszk==0) printf(”0 bit”);
else printf(”1 bit”); //Az ES eredmenyeben nem minden bit 0.
```

Érdemes megjegyezni néhány hasznos trükköt, amelyek bitenkénti operátort tartalmaznak:

- ’0’ bitekkel ÉSelve bármit az eredmény ’0’ bitek sorozat (bit törlés)
- ’1’ bitekkel VAGYova bármit az eredmény ’1’ bitek sorozata (bit beállítás)
- Egy érték saját magával XORolása csupa ’0’ bitet eredményez
- Csupa ’1’ bitet tartalmazó értékkel történő ÉSelés eredménye és másik operandusa megegyezik.
- Csupa ’0’ bitet tartalmazó értékkel történő VAGYolás eredménye és másik operandusa megegyezik.
- Csupa ’0’ bitet tartalmazó értékkel történő XORolás eredménye és másik operandusa megegyezik.
- Csupa ’1’ bitet tartalmazó értékkel történő XORolás egyenértékű a tagadással
- Bármilyen értékre kétszer végrehajtva a negálást (bitenkénti tagadást) az eredmény nem változik.
- Balra tolás 1 bittel egyenértékű a 2-vel szorzással (túlcsordulás lehet)
- Jobbra tolás 1 bittel egész osztás 2-vel (negatív számokra is)
- N db bittel egymás után mindkét irányba (balra majd jobbra, vagy jobbra, majd balra) eltolás során a kiinduló érték nem biztos, hogy azonos az eredménnyel.

Irodalom:

- Brian W. Kernighan, Dennis M. Ritchie: A C programozási nyelv, Műszaki kiadó (2008)

Alacsony szintű fájlkezelés

A C programozási nyelv (könyvtári függvények segítségével) kétféle lehetőséget biztosít I/O (pl. fájlkezelés) megvalósítására: formázott- és alacsonyszintű input-output. Előbbi bizonyára jól ismert a korábbi tanulmányokból. Lényege, hogy a beolvasandó adatokat egy formátum-sztring segítségével megadott alakú karaktersorozatból adott (különböző típusú) változó-tartalmakká alakítjuk. Írásnál ennek a fordítottja történik. Vegyük az alábbi függvényhívást példaként:

```
fprintf(stdout, ”50%%+%03d%c%6.2f.”, 1, ’=’, 1.5);
```

Az alapértelmezett kimeneti felületen megjelenő szöveg: `50%+001=  1.50`. Itt kezdetben a kiírandó értékek egy egész szám (előjeles, 4 bájtos, fix pontos számábrázolás), egy karakter (1 bájtos, ASCII kód fix pontos számábrázolással) és egy valós szám (8 bájtos, dupla pontosságú lebegőpontos számábrázolás) valamint további karakterek sorozata egy sztringben (karaktertömb). Ezt alakítja át a függvény egy egyszerű (ember számára olvasható információt tartalmazó) karaktersorozattá, amit majd outputként olvashatunk. Az ilyen formázott I/O alkalmazásánál főként az alábbi `stdio.h` függvényeket használhatjuk: `printf`, `scanf`, `fopen`, `fprintf`, `fscanf`, `fseek`, `fclose`, `fgets`, `fputs`, `fflush`, stb.

Ezzel szemben az alacsonyszintű I/O során nincs formátum-sztring, egyszerűen egy adott memóriacímtől kezdve adott számú bájt kerül kiírásra (beolvasásra) pontosan olyan bitsorozatként, ahogyan azt a RAM-ban is megtalálhatjuk. Így valósíthatunk meg például a bináris fájlkezelést.

A fájl megnyitása az `fcntl.h` header-ben található `open` rendszerhívással történik. Az `open` függvény első paramétere egy sztring, amely a megnyitandó állomány nevét (elérési útját) tartalmazza. Második paramétere azt határozza meg, hogy milyen módon történjen a megnyitás. A paraméter egy egész szám, amelynek egyes bitjei különböző jelentésekkel bírnak. Az egyszerűbb használat kedvéért az egyes „funkciókhoz” nevesített konstansok lettek definiálva az `fcntl.h` header állományban. A nevekhez olyan értékek tartoznak, amelyek csak az adott bitpozícióban tartalmaznak ’1’ bitet mindenhol máshol ’0’ bitérték van jelen. Az alábbi nevesített konstansokat használhatjuk:

`O_CREAT`	Nem létező fájl esetén a létrehozásról gondoskodik.
`O_RDONLY`	Kizárólag a fájl olvasását engedélyezi. Megnyitás után a „fájl kurzor” az állomány elejére pozícionálódik.
`O_WRONLY`	Kizárólag a fájl írását engedélyezi. Megnyitás után a „fájl kurzor” az állomány elejére pozícionálódik.
`O_RDWR`	A fájl írása és olvasása is lehetséges. Megnyitás után a „fájl kurzor” az állomány elejére pozícionálódik.
`O_APPEND`	Hozzáfűzést tesz lehetővé, azaz írhatjuk az állományt és megnyitás után a „fájl kurzor” az állomány végére pozícionálódik.
`O_TRUNC`	A megnyitandó fájl tartalmát nyitáskor törli (csonkolja) függetlenül az esetleges korábbi tartalomtól, azaz biztosak lehetünk benne, hogy üres állománnyal kezdünk dolgozni.
`O_BINARY`	Lehetővé teszi a bináris fájlok kezelését. Csak Windows rendszerben szükséges (mivel Linux alatt nincs különbség az egyszerű karakteres és a bináris fájlok között).

Ezen értékeket bitenkénti VAGY operátorral összekapcsolva meghatározhatjuk azt az egy darab egész értéket, amely a kívánt „funkció kombinációt” elérhetővé teszi.

Amennyiben az `open` rendszerhívás második paraméterében az O_CREAT maszk által meghatározott bit is be van állítva, akkor (és csak akkor) a függvénynek egy harmadik paramétere is van, amely a segítségével a létrehozandó fájl jogosultságrendszerét állítja be. Itt is kettő hatványoknak megfelelő nevesített konstansok használhatóak. Ezek a `sys/stat.h` header-ben vannak definiálva. Az olvasási/írási/futtatási jogosultságok tulajdonos/csoport/egyéb felhasználók számára történő

beállítására használhatjuk a `S_IRUSR`, `S_IWUSR`, `S_IXUSR`, `S_IRGRP`, `S_IWGRP`, `S_IXGRP`, `S_IROTH`, `S_IWOTH`, `S_IXOTH` értékeket. (Lásd még az inode kezelés fejezetben!)

Az open függvény egész értékkel tér vissza. Megnyitási hiba esetén `-1` értéket kapunk. Sikeres nyitás esetén egy 2-nél nagyobb egész számmal tér vissza a függvény, ezt a számot használhatjuk később fájlleíróként, azaz megnyitás után ezzel hivatkozhatunk az adott állományra. A `0` érék mindig az alapértelmezett bemeneti felületet (stdin), az `1` az alapértelmezett kimeneti felületet (stdout), míg a `2` érték az alapértelmezett hiba felületet (stderr) jelenti. Ezeket nem szükséges/lehet `open` rendszerhívással megnyitni, sem lezárni, mindig elérhetőek.

Az alacsony szinten megnyitott állományból történő olvasás a `read` rendszerhívás segítségével történik (`unistd.h`). Ennek három paramétere van. Első az olvasandó fájl leírója (az `open` által visszaadott érték). A második egy memóriacím (általában egy buffer tömb kezdőcíme), ahová a beolvasott bájtok kerülnek. Ez természetesen lehet akár statikus tömb, akár programozó által lefoglalt memória, stb. A harmadik paraméter egy (nem negatív) egész szám, amely azt határozza meg, hány bájt kerüljön beolvasásra. (Természetesen ez a szám ne legyen nagyobb a használandó memóriaterület méreténél!) Maximum ennyi bájtnyi fájlból beolvasott adattal felülíródik a használt memóriaterület eleje. Az `read` rendszerhívás egy egész értékkel tér vissza, ami nem más, mint a ténylegesen/sikeresen beolvasott bájtok száma, azaz nem nagyobb érték, mint az aktuális `read` hívás harmadik paramétere. A beolvasás végére ennyi bájttal kerül hátrébb a „fájl kurzor”. A visszaadott 0 érték jelentheti azt is, hogy elértük az állomány végét, azaz nincs több beolvasandó bájt.

A fájlok írása a `write` rendszerhívással történik (`unistd.h`). Ennek is három paramétere van. Első az írandó fájl leírója. Második a kiírandó bájtokat tartalmazó összefüggő memóriaterület kezdőcímet. A harmadik az írandó bájtok maximum száma. A függvény egy nem negatív egész számmal tér vissza, a sikeresen kiírt bájtok számával. (Ez természetesen nem lehet nagyobb érték, mint a `write` harmadik paramétere.) A „fájl kurzor” ennyivel hátrébb kerül az írás végére. A korábbi fájltartalom(rész) az írás során felülíródik.

A „fájl kurzort”, azaz a következő írási/olvasási művelet végrehajtási helyét a `write`/`read` rendszerhívástól függetlenül is pozícionálhatjuk az `lseek` függvény segítségével. Első paramétere egy egész típusú fájlleíró, amivel az adott fájlra hivatkozunk. Második paramétere egy bájtokban mért relatív eltolás/távolság a harmadik paraméterként megadott bázishoz képest. A bázis lehet `SEEK_SET` (fájl eleje), `SEEK_CUR` (aktuális fájl pozíció) illetve `SEEK_END` (fájl vége). Így tehát ha a 4-es fájlleíróval rendelkező állomány „kurzor” pozícióját az aktuálishoz képest 10 karakterrel/bájttal előrébb szeretnénk állítani a `lseek(4,-10,SEEK_CUR);` hívással kell élnünk.

Ha már nem akarunk dolgozni egy megnyitott állománnyal, akkor bezárhatjuk a `close` rendszerhívás segítségével. Ennek egyetlen paramétere a bezárandó fájl leírója.

A háttértárolókon a fájlok hardveres írása nem bájtonként, hanem blokkonként/szektoronként történik. A blokkok mérete kB nagyságrendű, így egy bájt (char) kiírása és 1000 bájt kiírása is ugyanannyi időbe kerül. Emiatt jól meg kell gondolnia a programozónak, hogy hányszor és mekkora méretű adathalmaz kiírását hajtja végre. Adott mennyiségű adat háttértárra történő írása akár 1000-szer több időbe is kerülhet, mint optimális esetben. Alacsony szintű fájlkezelés esetén ugyanis nincs rendszer által biztosított automatikus pufferelés, a kiírás azonnal történik. (Szemben a formázott fájlkezeléssel, ahol az írandó tartalmat a rendszer egy pufferbe helyezi, és csak akkor íródik ki ténylegesen a fájlba, ha a puffer megtelt. Ezt változtatja meg az `fflush` függvény.) Így alacsonyszintű fájlkezelés esetén a program hatékony/gyors működését (szükség esetén a pufferelést) a programozó feladata biztosítani.

Fontos még egyszer hangsúlyozni, hogy az alacsony szintű írás/olvasás során egy adott memóriakép egy az egyben történő mozgatása valósul meg. Vegyük például az alábbi programrészletet!

```
int abc=130;
```

```
write(1,&abc,sizeof(abc));
```

Az `abc` nevű, egész típusú változó értéke 130. Előjeles fix pontos számábrázolásnak és az Intel rendszerek fordított bájtsorrendű (littleendianness) működésének köszönhetően az `abc` változó memóriaterületén így a következő bitek találhatóak:

```
00000010 00000001 00000000 00000000.
```

A kiírás során ez a 4 bájt (ebben a sorrendben) kerül kiíratásra, azaz a következő ASCII kódú karakterek jelennek meg a standard output-on: 2, 1, 0, 0. (Ugyanekkor formázott I/O esetén a 49, 51, és 48 ASCII kódoknak megfelelő '1', '3' és '0' karakterek jelentek volna meg.) Ezért hívjuk ezt bináris fájlkezelésnek.

Nézzünk egy másik példát! Adott egy 100 darab valós számot tartalmazó tömb (`float asd[100];`). (Az egyszerűség kedvéért tegyük fel, hogy minden tömbelemnek `-314.1593` értéket adtunk.) Ezt kiírhatjuk formázottan egy fájlba az alábbi programrészlet segítségével:

```
for(i=0;i<100;i++)
  fprintf(fd1, "%f\t", asd[i]);
```

Ekkor a fájlba ember számára könnyen olvasható karakterek sorozata kerül, azaz

```
-314.159302     -314.159302     -314.159302     -314.159302 …
```

Vagyis a lebegő pontos számok karaktersorozattá konvertálódnak, szeparátorként tabulátor karaktert kell alkalmaznunk és a fájl mérete 1200 bájt lesz. Ugyanennek a tömbnek a kiíratása alacsony szinten így néz ki (ciklusszervezés nem szükséges):

```
write(fd2, asd, 100*sizeof(float));
```

Ekkor az adatok ember számára nagyon nehezen értelmezhetően, binárisan, a konkrét memóriaképnek megfelelően, konverzió és szeparátorok nélkül íródnak ki. (Az ember számára olvashatóság egy adatfájlban sokszor nem követelmény, főleg ha annak feldolgozása is egy program feladata.) A fájl mérete mindössze 400 bájt. Látható, hogy nagyon nem mindegy, hogy mikor melyik technikát alkalmazzuk. Az adott feladat követelményei, körülményi alapján a programozónak kell tudnia, hogy abban az esetben melyik technika hatékonyabb.

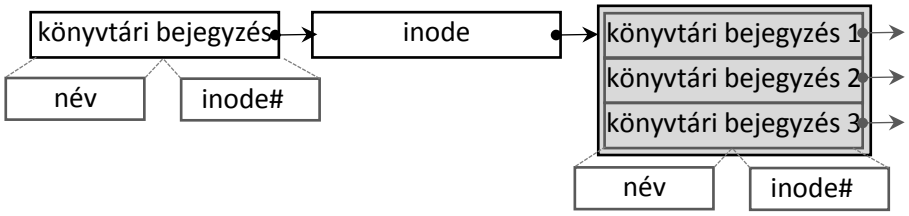
Irodalom

- Neil Matthew, Richard Stones: *Beginning Linux Programming*, Third Edition (2004), 96-107. oldal
- Brian W. Kernighan, Dennis M. Ritchie: *A C programozási nyelv*, Műszaki kiadó (2008) 187-192. oldal

Könyvtár és inode kezelése

Könyvtár kezelés

A Linux fájlrendszerekben egy objektum (fájl, könyvtár, link) tárolása, nyilvántartása egy három részből álló egységben történik, ugyanis az objektum neve, attribútumai és tartalma külön helyen tárolódik. Egy szöveges fájl tartalma például egyszerű bájtsorozatként a háttértár szektorainak egy csoportjában tárolódik. A fájl attribútumai a háttértár egy másik részén egy ún. inode-ban található. A név egy külön könyvtári bejegyzésben található, egy az objektumhoz tartozó inode azonosítóval együtt. Mivel Linux rendszerben majdnem minden fájl, így a könyvtárak is fájlként jelennek meg. A könyvtár egy olyan fájl, amelynek a tatalma speciális, ugyanis további fájlok könyvtári bejegyzés rekordjainak egy csoportját tartalmazza. Emiatt a könyvtárak kezelésének filozófiája hasonló a fájlkezeléshez.

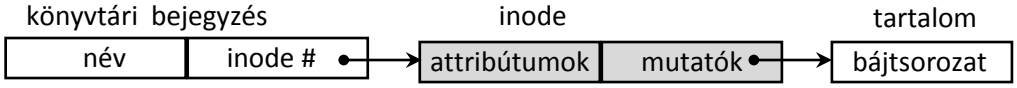


Egy C nyelvű programban a könyvtárakra `DIR*` típusú mutató segítségével hivatkozhatunk. Első lépésben meg kell nyitni a könyvtárat a `dirent.h` header állományban definiált `opendir` függvény segítségével. Ennek egyetlen paramétere van, egy sztring, amely meghatározza a megnyitandó könyvtár nevét (és elérési útját). A megnyitás módját nem kell/lehet meghatározni, mivel csak olvasni tudjuk a könyvtárak bejegyzés rekordjait. A függvény egy `DIR*` típusú mutató értékkel tér vissza, amellyel a későbbiekben az adott könyvtárra hivatkozhatunk. Mivel egy könyvtár-állomány tartalma nem más, mint könyvtári bejegyzés rekordok sorozata, ezeket tudjuk egyesével beolvasni a `readdir` függvény segítségével. Ennek egyetlen paramétere egy `DIR*` típusú érték, amely azonosítja az (korábban megnyitott) olvasandó könyvtárat. Egy könyvtári bejegyzés rekord a `dirent` nevű (`dirent.h`-ban definiált) struktúrába tárolható el. A `readdir` függvény egy ilyen `dirent*` típusú mutatóval tér vissza, amely a beolvasott rekord adatait tartalmazó memóriaterületre mutat. Amennyiben a függvény `NULL` értékkel tér vissza, akkor nincs több beolvasandó bejegyzés a könyvtárban. Egy `dirent` rekordnak a két legfontosabb mezője a `d_name` és a `d_ino`. Előbbi az adott/megnyitott könyvtárban lévő objektum (fájl, másik könyvtár, link) nevét, utóbbi az adott névhez tartozó inode azonosítóját tartalmazza. A műveletek elvégzése után a megnyitott könyvtárat be kell zárni a `closedir` eljárás segítségével történhet, melynek egyetlen paramétere a bezárandó könyvtárhoz tartozó `DIR*` mutató értéke.

Egy C programban az adott folyamat aktuális munkakönyvtárának futásidőben történő megváltoztatására az `unistd.h` header állomány `chdir` függvénye szolgál, melynek egyetlen paramétere az új munkakönyvtár elérési útját és nevét tartalmazó sztring.

Az inode kezelése

Ahogy már korábban is meg lett említve Linux fájlrendszerekben egy objektum (fájl, könyvtár, link) tárolása, nyilvántartása egy három részből álló egységben történik, ugyanis az objektum neve, attribútumai és tartalma külön helyen tárolódik. Például egy szöveges fájl tartalma egyszerű bájtsorozatként a háttértár szektorainak egy csoportjában tárolódik. A fájl attribútumai a háttértár egy másik részén egy ún. inode-ban található. Ez az információs csomópont tartalmazza például a méretet, a tulajdonost, a jogosultságokat, időbélyegeket, stb. valamint azt, hogy mely szektorokban található a tartalom, viszont nem tartalmazza az objektum nevét. A név egy külön könyvtári bejegyzésben található, egy az objektumhoz tartozó inode azonosítóval együtt.



Az inode-nak jól meghatározott szerkezete van, amelynek kezeléséhez a `sys/stat.h` header állományban egy rekord van definiálva az alábbi módon:

```
struct stat {
  dev_t     st_dev;         /* fájlt tartalmazó eszköz ID */
  ino_t     st_ino;         /* inode szám */
  mode_t    st_mode;        /* fájl típus és mód */
  nlink_t   st_nlink;       /* hard linkek száma */
  uid_t     st_uid;         /* tulajdonos felhasználó ID */
  gid_t     st_gid;         /* tulajdonos csoport ID */
  dev_t     st_rdev;        /* eszköz ID (ha specialis fájl) */
  off_t     st_size;        /* teljes méret bájtokban */
  blksize_t st_blksize;     /* blokk méret */
  blkcnt_t  st_blocks;      /* lefoglalt 512B blokkok száma */
  struct timespec st_atim; /* utolsó hozzáférés ideje */
  struct timespec st_mtim; /* utolsó módosítás ideje */
  struct timespec st_ctim; /* utolsó állapotváltás ideje */
  };
```

Az `st_mode` mező egyes bitjei külön-külön jelentéssel bírnak (egy-egy állapot/jellemző meglétét vagy hiányát jelenti). A `sys/stat.h` headerben léteznek olyan nevesített konstansok, amelyek egy olyan maszk értéket takarnak, amelyek '`1`' bitet tartalmaznak a kérdéses bitpoziciókban és minden más bitpozícióban pedig '`0`' bitet. Ezek közül a legfontosabbak az alábbi listában láthatóak.

`S_IFREG`	szabályos fájl-e
`S_IFDIR`	könyvtár-e
`S_IFLNK`	szimbolikus link-e
`S_IFSOCK`	socket-e
`S_IFBLK`	blokkos eszköz-e
`S_IFCHR`	karakteres eszköz-e
`S_IRUSR`	a tulajdonosnak van-e olvasási joga
`S_IWUSR`	a tulajdonosnak van-e írási joga
`S_IXUSR`	a tulajdonosnak van-e futtatási joga
`S_IRWXU`	a tulajdonosnak megvan-e minden joga
`S_IRGRP`	a csoportnak van-e olvasási joga
`S_IWGRP`	a csoportnak van-e írási joga
`S_IXGRP`	a csoportnak van-e futtatási joga
`S_IROTH`	a többieknek van-e olvasási joga
`S_IWOTH`	a többieknek van-e írási joga
`S_IXOTH`	a többieknek van-e futtatási joga

A `stat` függvény segítségével lehet a C nyelvű programban lekérdezni egy fájl (könyvtár, link) állapotát. A függvény két paraméterrel rendelkezik az első egy sztring, amely a lekérdezendő fájl nevét (elérési útját) tartalmazza. A második paramétere `struct stat` típusú változó címe, hívás után ez a változó fogja tartalmazni a fájl adatait. A függvény hiba (pl. nem létező fájl) esetén `-1` értéket ad vissza.

Irodalom:

- Neil Matthew, Richard Stones: *Beginning Linux Programming,* Third Edition (2004), 120-126. oldal
- Neil Matthew, Richard Stones: *Beginning Linux Programming,* Third Edition (2004), 104-106. oldal

Socket programozás

A hálózati (IP) kommunikáció alkalmazását teszi lehetővé C nyelvű programokban. A technológia sikere elsajátításához szükség van alapvető hálózati ismeretekre, amelyek nem képezik tárgyát a jegyzetnek. (Frissítsd fel az ismereteidet az IP, TCP és UDP protokollal kapcsolatban!) Ahhoz, hogy a hálózaton kommunikáló programot írjunk (az alapokon túl) meg kell ismernünk néhány speciális rekord típust, számos újabb rendszerhívást, konvertáló- és információs függvényeket.

Mindenek előtt létre kell hoznunk az programunkban egy socket-et (szoftveres csatlakozót), amelyen keresztül a programunk a hálózathoz tud csatlakozni. A hálózati kommunikáció lehet kapcsolat-orientált (lásd. TCP) vagy kapcsolat nélküli (lásd. UDP). Előbbi megbízható, de lassú adatátvitelt valósít meg bonyolultabb technológiával, utóbbi ugyan nem megbízható, de a gyors és egyszerű adatátvitelre törekszik. Ezeknek megfelelően két különböző socket létezik. Az egyik az adatfolyam jellegű kommunikációt biztosító `SOCK_STREAM` típus (TCP), a másik az egyszerű adatcsomag továbbításon alapuló `SOCK_DGRAM` típus (UDP).

Alapvető struktúrák

Az IPv4 címek tárolására a `netinet/in.h` header állomány `in_addr` szerkezetét használhatjuk, ami legalább egy mezőt tartalmaz az alábbi definíció értelmében:

```
struct in_addr {
  unint32_t s_addr;
  };
```

Ez a mező egy előjel nélküli 32 bites egészként tárolja az IP címet big-endian, azaz hálózati bájtsorrendben. Tehát ha a 127.0.0.1 IP címet szeretnénk eltárolni, akkor írhatjuk ezt:

```
struct in_addr IP;
IP.s_addr=16777343; // 0x0100003f
```

Egy IP cím önmagában még nem elegendő a kommunikációhoz más adatokra is szükségünk van, ezért használjuk az alábbi szerkezetű `sockaddr_in` struktúrát:

```
struct sockaddr_in {
  short int sin_family;
  unsigned short int sin_port;
  struct in_addr sin_addr;
  unsigned char sin_zero[8];
  };
```

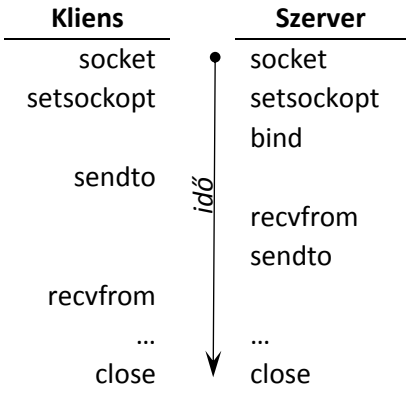
Az első mező a címcsaládot adja meg, amelynek értéke az Interneten kommunikáló programjaink esetén az `AF_INET` nevesített konstanssal adható meg. A második mezőben adhatjuk meg a szállítási réteg protokollok 16 bites portját big-endian, azaz hálózati bájtsorrendben. A harmadik mezőben az imént ismertetett módon egy IP címet adhatunk meg. A negyedik mező csak egy egyszerű kitöltő, ami arra szolgál, hogy a szerkezet azonos méretű legyen a `sys/socket.h` header állományban definiált `sockaddr` struktúrával, amit bizonyos később említendő függvények használnak, így a kompatibilitás biztosítható. Ennek a rekordnak a szerkezete a következő:

```
struct sockaddr {
  unsigned short sa_family;
  char sa_data[14];
  };
```

Itt az első mező ugyanaz, mint a `sockaddr_in` struktúrában. A második mező pedig a protokollcím tárolására szolgáló bájtsorozat.

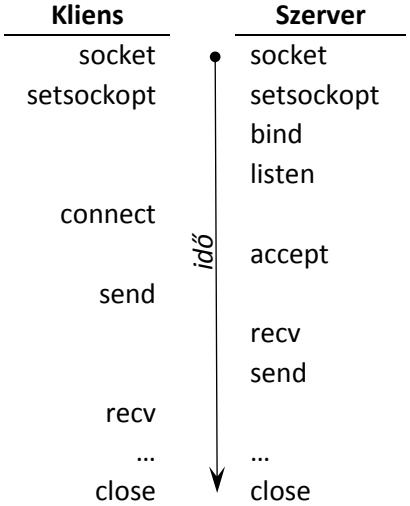
A socket programozás rendszerhívásai

A hálózati kommunikációhoz különböző rendszerhívásokat kell alkalmaznunk megfelelő pillanatban. Mielőtt megnéznénk, hogyan kell használni ezeket a rendszerhívásokat, tekintsük át, mikor kell alkalmaznunk őket! Először nézzük az egyszerűbb kapcsolat nélküli kommunikáció (lásd UDP) lépéseinek időrendjét!



Minkét félnél először létre kell tehát hozni egy socketet, majd ha szükséges rájuk vonatkozó opciókat lehet beállítani. A szerveren össze kell rendelnünk a címeket a sockettel, hogy tudja a rendszer, hogy milyen portra érkező szegmenseket kell az adott programnak átadnia. Általában a kliens kezdi a kommunikációt azzal, hogy küld valamit a szervernek, aki erre válaszol. A kommunikáció végén mindkét fél lezárja a kommunikációt.

A kapcsolatorientált kommunikáció (TCP) menetében is vannak hasonló lépések, de a csatlakozó és a címek összerendelése után további teendőink vannak. Először a szervernek egy figyelő állapotba kell kerülnie, ahol arra vár, hogy egy kliens kapcsolódni akarjon hozzá. Miután a kliens a kapcsolat kiépítési kérést elküldte a szerver elfogadhatja azt, így kiépül a kapcsolat (lásd három-utas kézfogás). Az üzenetek küldése csak ezután következhet be. A kiépült kapcsolatot végül le kell bontani.



Ezek után részletesen tekintsük át az alkalmazható rendszerhívásokat! Többségüket a `sys/socket.h` header állomány definiálja.

- `int socket(int family, int type, int protokcol);`

Célja a socket létrehozása. Az első paramétere az alkalmazandó címcsalád specifikációja (`AF_INET`). Második paramétere a socket típusa (`SOCK_STREAM` vagy `SOCK_DGRAM`). A harmadik paraméter többnyire mindig lehet az alapértelmezett `0` egész érték. Hiba esetén -1 értékkel tér vissza,

míg siker esetén egy egész típusú fájlleírót kapunk, amivel mint azonosítóval később a socketre tudunk hivatkozni.

- `int setsockopt(int fd, int level, int cmd, char *arg, int len);`

Az első paraméterben megadott csatlakozóra vonatkozó opciókat állíthatunk be vele. A második paraméterként a SOL_SOCKET értéket szokás megadni. A harmadik paraméterrel azonosítjuk be az opciót. A negyedik az opció értéke pszeudo-cím szerinti paraméterátadással, mely címen általában egy igaz jellegű érték (1) foglal helyet. Ennek az opcióértéknek a bájtokban megadott mérete az utolsó paraméter. A harmadik paraméterben megadhatjuk például a SO_KEEPALIVE nevesített konstanst, ezáltal a kiépített kapcsolatot a rendszer életben tartja akkor is, ha sokáig nem történik üzenetküldés. Egy másik lehetséges érték a SO_REUSEADDR, melynek hatására a felhasznált helyi címet újabb futtatás esetén újra felhasználhatóvá teszi a rendszer. (Például tesztek esetén hasznos lehet.) Hiba esetén -1 értékkel tér vissza, míg siker esetén nullával.

- `int bind(int fd, struct sockaddr *addrp, int alen);`

A socket hozzárendelése a hálózati címhez ezzel történik a szerveren. Első paramétere a socket azonosítója, második egy pointer, amely arra a `sockaddr` struktúrára mutat, amiben eltároltuk a használni kívánt címet. A harmadik paraméter az előbbi struktúra méretét adja meg bájtokban. Hiba esetén -1 értékkel tér vissza, egyébként 0-val.

- `int listen(int fd, int backlog);`

Szerver oldalon a kapcsolat felvételi kérés figyelésére és a sorméret beállítására szolgál (kizárólag kapcsolatorientált esetben). Első paramétere annak a socketnek a fájlleírója, amelyen figyelni szeretnénk a bejövő kéréseket. Második paramétere a beérkező kapcsolódási igények (`connect`) maximális száma, melyek egy várakozási sorba kerülnek. Hiba esetén -1 értékkel tér vissza, egyébként nullával.

- `int connect(int fd, struct sockaddr *addrp, int alen);`

A kliens kapcsolati kiépítési kérése (a szerver felé) ezzel valósul meg. Első paramétere a kommunikációra használandó socket azonosítója. Második paramétere annak a címstruktúrának a címe, melyben előzetesen eltároltuk a cél szerver elérhetőségét. Harmadik paraméter az előbbi címstruktúrának a méret bájtokban. Hiba esetén -1 értékkel tér vissza, egyébként nullával.

- `int accept(int fd, struct sockaddr *addrp, int *alenp);`

A kapcsolati kérés elfogadása szerver oldalon (kapcsolatorientált esetben). Első paramétere a kommunikációra használandó socket leírója. Második paramétere annak a címstruktúrának a címe, melybe az elfogadás során elmentésre kerül a kapcsolatot kérő kliens elérhetősége. Harmadik paraméter az előbbi címstruktúrának a méret bájtokban. Hiba esetén -1 értékkel tér vissza, sikeres kapcsolatkiépítés után viszont egy új socket jön létre és ennek az azonosítóját kapjuk vissza. (Ezt kell később a szerveren az adatküldés és fogadás során használni.) A régi (paraméterként is megadott) socket fájlleíró továbbra is azt a socketet azonosítja, ahol a további bejövő kéréseket figyeljük.

- ```
  int sendto(int fd, char *buff, int len, int flags,
             struct *addrp, int alen);
  ```

Kapcsolatmentes (UDP alapú) üzenetküldésre szolgál. Az első paraméterében kell megadni, hogy melyik socketen keresztül történjen az információtovábbítás. Második paramétere a küldendő bájtsorozat kezdőcíme, ahonnan a harmadik paraméterben megadott számú bájt kerül elküldésre. A negyedik paraméterének egyes bitjei különböző lehetőségeket írhatnak le. Alapesetben ennek a paraméternek az értéke lehet 0. Az ötödik paraméterben megadott pointer arra a memóriaterületre utal, hol a fogadó fél címe van eltárolva annyi bájton, amennyi a hatodik paraméterben szerepel. Hiba esetén -1 értékkel tér vissza, egyébként a ténylegesen elküldésre került bájtok számát kapjuk meg.
  ```

- `int send(int fd, char *buff, int len, int flags);`

Kapcsolatorientált (TCP alapú) üzenetküldésre szolgál. Mivel itt a kapcsolat már kiépült, nem kell külön megmondani célt, így a `sendto(fd, buff, len, flags, NULL, 0);` hívással egyenértékű. A `send` helyett használhatjuk a `write(fd, buff, len);` rendszerhívást is.

- `int recvfrom(int fd, char *buff, int maxlen, int flags, struct *addrp, int *alenp);`

Kapcsolatmentes (UDP alapú) üzenet fogadást tesz lehetővé. Az első paraméterében adjuk meg, hogy melyik socketen keresztül történjen az információtovábbítás. Második paramétere egy korábban lefoglalt memóriaterület címe, ahová mentésre kerül a fogadott bájtsorozat. A harmadik paraméter az előbbi memóriaterület mérete bájtokban. A negyedik paramétere hasonló a `send` és a `sendto` negyedik paraméteréhez. Az ötödik paraméterben megadott pointer arra a memóriaterületre mutat, ahová a küldő fél címe tárolásra kerül a hívás során. A hatodik paraméterben van megadva az előző rekord méretének eltárolására szolgáló cím. Hiba esetén -1 értékkel tér vissza, egyébként a ténylegesen fogadott bájtok számát kapjuk meg. Amennyiben a `recvfrom` meghívásakor az üzenet még nem érkezett meg a hálózaton, akkor alapesetben a folyamat várakozó állapotba kerül, és csak akkor fejeződik be, ha adat érkezett.

- `int recv(int fd, char *buff, int maxlen, int flags);`

Kapcsolatorientált (TCP alapú) üzenetfogadásra szolgál. Mivel itt a kapcsolat már kiépült, nem kell megadni a cél elérhetőségét, így a recvfrom(fd, buff, len, flags, NULL, NULL); hívással egyenértékű. Ha a visszatérési érték 0, akkor az arra utal, hogy a túloldalon a kapcsolatot bezárták. A `recv` helyett használhatjuk a `read(fd, buff, len);` rendszerhívást is.

- `int close(int fd);`

A bináris fájlkezelésnél már megismert unistd.h header állománybeli rendszerhívás használható a socket lezárására a kommunikáció végén. A lezárandó csatlakozó fájlleírója az egyetlen paraméter. Hiba esetén -1 értékkel tér vissza, siker esetén 0-val.

- `int shutdown(int fd, int how);`

Kapcsolat-orientált socket egyoldalú lezárására szolgál. Első paramétere a lezárandó socket azonosítója. Ha a második paramétere 0, az azt jelenti, hogy nem lehet adatot fogadni többé a csatlakozón. Ugyanitt az 1 érték jelentése: nem küldhető itt több adat. A `shutdown(fd, 2);` hívás esetén sem küldeni sem fogadni nem lehet, tehát egyenértékű a `close(fd);` rendszerhívással.

Konverziós függvények

Az IP címek és port számok binárisan kerülnek tárolásra/felhasználásra, azonban az ember szöveges formában szereti megadni/megjeleníteni őket. Ezért különböző konverziós függvényeket használhatunk a programfejlesztés során. Például egy karaktertömbként megadott IPv4 címet 32 bites előjel nélküli egésszé alakíthatunk az `inet_addr` függvénnyel. Egy szövegesen megadott IP címmel fel is tölthetjük egy sockaddr_in struktúra megfelelő mezőjét az `inet_aton` függvény révén. Ennek ellentéte az `inet_ntoa` függvény, amellyel `sockaddr_in` struktúra mezője pontozott decimális alakú karaktersorozattá alakítható. Az alábbi kódrészlet ezekre mutat példát.

```
struct sockaddr_in IP;
IP.sin_addr.s_addr = inet_addr("127.0.0.1");
inet_aton("192.168.0.1",&(IP.sin_addr));
printf("%s \n",inet_ntoa(IP.sin_addr));
```

Az intel alapú számítógépeken a több bájtos értékek (pl. IP cím, port szám) little-endian bájtsorrendben vannak eltárolva a memóriában. Ezt a socket programozás terén gazdagépi

bájtsorrendben is hívjuk. Ezzel szemben a hálózati forgalomban ezek a több bájtos adatok (mint PDU fejrész mezők) big-endian bájtsorrendben kerülnek továbbításra, amit gyakran hálózati bájtsorrendnek hívunk. Emiatt bájtsorrendet konvertáló függvényekre is szükség van. Ezek a következők: `htons`, `ntohs`, `htonl`, és `ntohl`. A függvénynévben a 'h' a gazdagépre, az 'n' a hálózatra, az 's' a 16 bites értékre, az 'l' pedig a 32 bites értékre utal. A paraméterként megadott érték konvertált változatával térnek vissza a függvények. Egy példa felhasználást láthatunk a következő kódban, ahol a https szolgáltatás szerver oldali jól ismert port száma kerül beállításra.

```
struct sockaddr_in server;
server.sin_addr.s_addr = inet_addr("193.6.135.80");
server.sin_port = htons(443);
```

Egyéb lehetőségek

A `netdb.h` header állományban definiálva van egy `hostent` rekord, amely révén a domain neveket is kezelni tudjuk.

```
struct hostent {
  char *h_name;
  char **h_aliases;
  int h_addrtype;
  int h_length;
  char **h_addr_list;
  };
#define h_addr h_addr_list[0]
```

Ennek `h_name` mezője egy csomópont hivatalos (kanonikus) nevére hivatkozik. A `h_aliases` mező egy sztringeket tartalmazó tömböt jelöl, ahol a csomópont további álnevei szerepelnek. A `h_addrtype` és a `h_length` mezők a címcsalád típusát és a cím hosszát tárolják. A `h_addr_list` egy sztringeket tartalmazó tömb, amelyben fel vannak sorolva a csomópont címei (mivel lehet több is). Ez a tömb egy `NULL` értékkel végződik. A `h_addr` makró révén közvetlenül hivatkozhatunk az elsődleges címre.

Ezt a hostent struktúrát gethostbyname vagy a gethostbyaddr függvényekkel tudjuk feltölteni. Ezek fejléce a következő:

- `struct hostent *gethostbyname(char *hname);`
- `struct hostent *gethostbyaddr(char *addrp, int len, int family);`

A `gethostbyname` paramétere egy domain nevet tartalmazó sztring és az ehet tartozó adatokkal feltöltött `hostent` struktúra címével tér vissza. A `gethostbyaddr` első paramétere egy pontozott decimális formában megadott IP címet tartalmazó sztring, második paraméter a cím hossza, harmadik a cím családjának megadására szolgál. Szerepeljen itt egy példa is!

```
struct hostent *Computer;
Computer = gethostbyname("irh.inf.unideb.hu");
printf("IP: %s\n",(*Computer).h_addr);
Computer = gethostbyaddr("193.6.135.80");
printf("Name: %s\n",(*Computer).h_name);
```

Érdemes még megemlíteni itt a `netent`, `protent` és `servent` struktúrákat valamint a `getpeername`, `gethostname`, `getservbyname`, `getservbyport` és a `getsockname` függvényeket.

Irodalom:

- Michael J. Donahoo , Kenneth L. Calvert: *TCP/IP Sockets in C: Practical Guide for Programmers*, Elsevier 2009.

Folyamatok és a fork rendszerhívás

A folyamat (process) alatt a memóriába betöltött, végrehajtás alatt álló programot és az ehhez kapcsolódó operációs rendszer tevékenységeket értjük. Egy folyamaton belül egy vagy több (akár párhuzamosan is elvégezhető) végrehajtási szál (thread) lehet. A folyamatok (szemben a szálakkal) nem osztoznak az erőforrásokon, saját memóriaterülettel és címtérrel rendelkeznek, külön fileleíróik vannak, stb. Minden folyamat egyedi azonosítóval rendelkezik, amely PID (Process ID) néven gyakorlatilag egy nem negatív egész számot takar.

Multi-programozott környezetben egyszerre több folyamat is jelen van, amelyek különböző állapotokban lehetnek. Minden folyamat (kivéve az 1 PID-ű process) van pontosan egy szülő folyamata és lehetnek gyermek folyamatai (nulla vagy több), ugyanis minden folyamatot egy másik hoz létre. Az ún. `init` folyamat (PID 1) minden folyamat őse. Minden folyamat tudja ki a szülője.

Egy program indításakor keletkező folyamat először átmenetileg egy „új" állapotba kerül, majd beállva az ütemezési sorba „futásra kész" állapotúvá válik. Az ütemezési sorban lévő folyamatok közül az ütemező (scheduler) dönti el, melyik folyamat kapja meg a CPU-t, azaz válik „futó" állapotúvá. Egy process lemondhat a CPU-ról vagy az ütemező is elveheti tőle. Utóbbi esetben visszakerül futásra kész állapotba, vagyis az ütemezési sorba kerül. Ha a folyamat önként mond le az erőforrásról (mert például, inputra/erőforrásra/szignálra vár), akkor „várakozó" állapotba kerül. Miután a szükséges esemény bekövetkezett a folyamat ismét bekerül az ütemezési sorba, azaz futásra kész állapotú lesz. Amint a folyamat végrehajtotta utolsó utasítását (esetleg hibát érzékelt vagy külsőleg leállították) ideiglenesen egy speciális „halott" (zombie) állapotba kerül.

A folyamatok állapotait és egyéb jellemzőit (PID, programszámláló, regisztertartalmak, memória infó, stb.) az operációs rendszer az ún. Process Control Block-ban (PCB) tárolja. Linux rendszerben vannak parancsok, melyek segítségével informálódhatunk a folyamatok process táblában tárolt adatairól illetve a process-ekkel kapcsolatos műveleteket végezhetünk (pl. `ps`, `top`, `renice`). Ezeken kívül a Linux rendszerben a folyamatok is megjelennek (speciális módon) a fájlrendszerben. (Lásd `\proc` könyvtár.)

Egy C nyelvű programban a hívó folyamat saját PID lekérdezésére a `getpid` függvény szolgál. A visszaadott PID érték `pid_t` (egész) típusú. Ezen kívül a programunkban lekérdezhetjük a szülő folyamat (pl. terminálból indított program esetén a parancsértelmező) PID-jét is a `getppid` függvény segítségével. Mindkét függvény az `unistd.h` header-ben van deklarálva.

Az `unistd.h` header állományban található `fork` rendszerhívás lehetővé teszi azt, hogy egy folyamat létrehozzon egy gyermek folyamatot. A `fork` paraméter nélküli függvény visszaad egy pid_t típusú értéket és létrehozza az aktuális folyamat egy másolatát. Ez kívülről nézve azt jelenti, hogy egy programot indítunk el, de egyszer csak megjelenek egy másik folyamat is, amelynek a kódja, állapota, változóinak tartalma és minden egyéb jellemzője pontosan megegyezik az eredeti folyamattal és a végrehajtásuk ugyanott a `fork` hívás utáni utasításon folytatódik. (Majdnem olyan, mintha kétszer indítottuk volna el ugyanazt a programot és éppen ugyanott tartanának.) Az egyetlen különbség a két folyamatban a `fork` függvény által visszaadott érték. Az újonnan létrejövő (tehát gyermek) folyamatban a visszaadott érték mindig 0 (nulla) lesz. Az eredeti folyamatban a visszaadott érték megegyezik az éppen most létrejött gyermek folyamat PID értékével. Így a szülő megismeri a gyermeke azonosítóját. (A gyermek mindig ismeri a szülőét.) A `fork` által visszaadott érték alapján a programkódok meg lehet úgy írni (feltételes utasítás segítségével), hogy mindegyik folyamat tudja azt, hogy ő szülő-e vagy gyermek és ez alapján akár eltérő utasítássorozatot hajtson végre.

Például:

```
if(fork()==0)
  /* Things to do in child. */
else
```

```
/* Things to do in parent. */
```

Természetesen a gyerek folyamat is fork-olhat és a szülő is megismételheti a fork rendszerhívást, így összetettebb hierarchia is létrehozható. Ha több folyamat is lehet egyszerre futó állapotban az adott számítógépen, akkor ezek a folyamatok akár párhuzamosan is futhatnak. (Ez még nem párhuzamos programozás.) A szülő és a gyermek folyamatok nem látják egymás memóriaterületeit (külön változóik vannak), így köztük a kommunikáció nem valósítható meg „közös/globális” változókon keresztül.

Irodalom:

- Neil Matthew, Richard Stones: *Beginning Linux Programming,*
  Third Edition (2004), 445-461. oldal

Szignál

Egy folyamat számára a szignál egy esemény bekövetkeztét jelzi, egyfajta információtartalom nélküli jelzés, a folyamatok közötti kommunikáció (IPC) egyik eszköze. A szignál kezelésére a programot fel kell készíteni. Kezeletlen szignál érkezése esetén a folyamat leáll. A szignálok két csoportba sorolhatók: elkapható és nem elkapható szignálok. Előbbi érkezésére fel lehet készíteni a programot, meg lehet határozni, mi történjen ilyen szignál érkezésekor, át lehet definiálni az alapértelmezett jelentést egy szignálkezelő eljárás segítségével. Utóbbi esetben a szignál érkezésekor azonnal aktiválódik az alapértelmezett jelentés, nem lehet késleltetni, módosítani.

Néhány fontosabb szignál:

- `KILL`: Nem elkapható. Hatására a folyamat azonnal és visszavonhatatlanul befejeződik.
- `TERM`: Általában arra használjuk, hogy egy folyamatot „megkérjünk”, hogy tegye meg a szükséges lépéseket a befejezéshez.
- `INT`: Alap esetben megszakítja a folyamat futását és a program leáll, azonban elkapható és átdefiniálható. CTRL+c billentyűkombináció ezt a szignált küldve állítja le az éppen (előtérben) futó programot.
- `STOP`: Nem elkapható. Megállítja a folyamat futását, így a folyamat várakozó állapotba kerül, és ott marad mindaddig, amíg egy `CONT` szignált nem kap.
- `CONT`: Egy `STOP` szignállal megállított folyamatot újra futásra kész állapotúvá tesz, így az bekerül az ütemezési sorba. Miután az ütemező kiválasztja a korábban elmentett állapotból folytatja futását.
- `ALRM`: Időzítő lejártáról értesíti a folyamatot.
- `CHILD`: Egy gyermek folyamat megállításáról vagy befejeződéséről értesíti a szülő folyamatot.
- `USR1`: A programozó szabadon definiálhatja a jelentését. Saját célokra használható.
- `USR2`: A programozó szabadon definiálhatja a jelentését. Saját célokra használható.

Parancssorból a „`kill`” parancs segítségével küldhető szignál. Első argumentum a szignált írja le a második pedig a cél folyamat PID-je. Például: `kill -USR1 3928` (Programozói szignál küldése a 3928 PID-ű folyamatnak.)

A C programokban a szignálokra nevesített konstansokkal lehet hivatkozni, amelyek „SIG” karakterekkel kezdődnek . Például: `SIGINT` (2), `SIGKILL` (9), `SIGALRM` (14), stb.

A C nyelvű programokban szignálkezelő egy tetszőleges nevű (pl. Do_If_Sigint) eljárás lehet, amely egyetlen int típusú paraméterrel rendelkezhet (ez az aktuálisan kapott szignál azonosítóját/számát kapja értékül). Az eljárás törzse írja le, mit kell tenni adott szignál érkezése esetén. Tartalmilag tetszőleges eljárástörzs, bár gyakori alkalmazási forma, hogy csak flag-ek átállítása történik a szignálkezelőben.

A szignálkezelő eljárásnak nincs explicit hívása, szignál érkezésekor a rendszer hívja meg, ha korábban sikeresen lefutott egy megfelelő `signal` függvény. A `signal` függvénynek két paramétere van. Az első egy (általában nevesített konstanssal megadott) szignál azonosító/szám, második paraméter a korábban megírt szignálkezelő eljárás neve (pl. `signal(SIGINT, Do_If_Sigint);`).

Működést tekintve (általában a program elején) meghívjuk a `signal` függvényt, mellyel megmondjuk a rendszernek, hogy a megadott szignal érkezése esetén melyik szignálkezelőt kell elindítania, majd a végrehajtás megy tovább. Ha ezután bármikor érkezik egy adott szignál, akkor a program végrehajtása megszakad, aztán lefutnak a definiált szignálkezelő eljárás utasításai, végül a végrehajtás a megszakított eredeti utasítássorozat következő utasításával folytatódik.

Két „előre gyártott” speciális szignálkezelő is alkalmazható a `signal` függvény második paraméterében. A `SIG_IGN` jelentése: ha az adott szignál érkezik, hagyd figyelmen kívül. A `SIG_DFL` visszaállítja az adott szignál alapértelmezett szignálkezelőjét.

A C nyelvű programból a `kill(pid_t pid, int sig);` függvény segítségével tudunk szignált küldeni egy másik folyamatnak, ahol az első paraméter a cél folyamat azonosítója, a második pedig a küldendő szignál azonosítója.

Az eddig tárgyalt szignálozással kapcsolatos eszközök, azonosítók használatához inkludálni kell a `signal.h` header-t.

Az unistd.h header-ben található `pause` paraméter nélküli alprogram segítségével elérhetjük, hogy a C program egy adott pontján álljon meg a végrehajtás mindaddig, amíg egy (tetszőleges) szignál nem érkezik. Azaz a folyamat a `pause` hívás hatására várakozó állapotba kerül, majd szignál érkezése esetén kerül csak át „futásra kész” állapotba. (Normál esetben előbb-utóbb az ütemező átadja neki a vezérlést, lefut a definiált szignálkezelő eljárás, majd a végrehajtás a `pause` utáni következő utasításon folytatódik. Segítségével tehát a folyamatok szinkronizációját hajthatjuk végre.

Ha egy folyamat befejeződik (és néhány más esetben), a rendszer automatikusan küld egy `SIGCHLD` szignált a szülő folyamatnak. A szülő folyamat kódjában lévő `wait();` utasítás lehetővé teszi, hogy a szülő várakozó állapotba kerüljön mindaddig, amíg egy `SIGCHLD` szignált nem kap. Így tehát elérhető, hogy egy folyamat végrehajtása csak akkor folytatódjon, ha (egyik) gyermek folyamata befejeződött. Ha egy gyermek folyamat befejeződött, de a (még futó) szülőben még nem került meghívásra a `wait` akkor a gyermek folyamat nem szűnik meg, hanem „zombie” állapotba kerül. A `wait` használatához inkludálni kell a `sys/wait.h` header-t.

Egy szignálozáshoz kapcsolódó speciális függvény az `alarm`. Segítségével egy időzítőt indíthatunk el. Egy paraméterrel rendelkezik, az itt magadott (egész) számú másodperc lejárta után a folyamat kapni fog egy `SIGALRM` szignált (ezzel figyelmezteti magát a program, hogy lejárt a kiszabott idő). Természetesen a `SIGALRM` érkezésére fel kell készíteni a programot (szignálkezelő írás és `signal` hívás segítségével). Egy folyamatban egyszerre csak egy időzítő lehet aktív. Ha egy `alarm` hívás történik, mielőtt lejárna egy másik, akkor a korábbi időzítés hatására nem fog `SIGALRM` érkezni, de a függvény visszatér az így „kihagyásra ítélt” szignál érkezéséig hátralevő másodpercek (egész) számával. Az `alarm(0);` utasítás hatására az aktuális időzítő leáll, de nem indul új. Az időzítéshez szükséges header állomány az `alarm.h`.

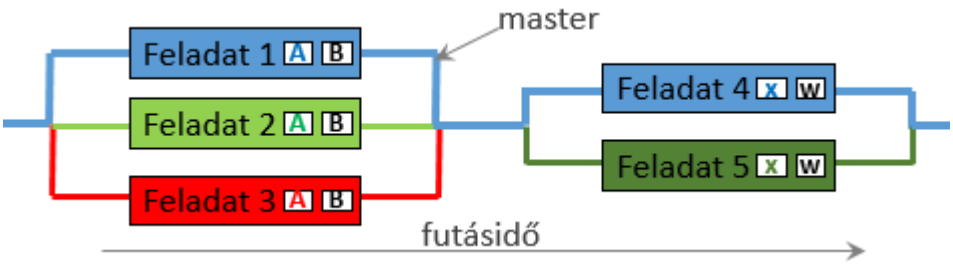
Irodalom:

- Neil Matthew, Richard Stones: *Beginning Linux Programming,*
  Third Edition (2004), 463-476. oldal
- Brian W. Kernighan, Dennis M. Ritchie: *A C programozási nyelv*, Műszaki kiadó (2008)

Párhuzamos programozás

A C nyelven az OpenMP API segítségével osztott memória modellen alapuló párhozamosan futó szállakat tartalmazó programokat írhatunk. Az OpenMP által biztosított fordítási direktívák és könyvtári függvények révén a programunk/folyamatunk egyes végrehajtási szálai a mindnyájuk által hozzáférhető memóriaterületeken keresztül képesek kommunikálni (információt cserélni) egymással, bár mindegyiküknek lehetnek saját memóriaterületei is. Így feladat és adatpárhuzamosságot is megvalósíthatunk. Ezen eszközök eléréséhez inkludálnunk kell az `omp.h` header állományt. Ezen kívül a `gcc` compilerrel történő fordítás esetén a `-fopenmp` kapcsolót is alkalmazni kell.

A programunk révén létrejött folyamat végrehajtása kezdetben mindig egyetlen szálon indul, majd később a megadott helyen ez több azonos időben is végrehajtható szálra osztható. Meg tudjuk azt csinálni, hogy program egy adott pontján a szálak újra egyesüljenek, majd később esetleg újra párhozamos végrehajtást kérhetünk. Minden szálnak van egy `int` típusú egyedi azonosítója. Az egyszerre jelen lévő szálak közül az egyiket (mindig pontosan egyet) master szálnak nevezünk, ennek azonosítója a `0`. Minden szálnak lehetnek ún. privát változói (alapértelmezés), amelyekhez csak az adott szál fér hozzá, illetve lehetnek megosztott változói, amikhez mindegyik szál ugyanúgy hozzáfér. Az alábbi ábrán egy példát láthatunk, hol az `A` és `x` változók privátak, míg a `B` és `w` osztottak. Ha a körülmények engedik, a 1., 2. és 3. feladat akár egyszerre is futhat külön processzormagokon. Ugyanígy a 4. és 5. feladat is, de ezek biztos, hogy csak az után kezdődhetnek el, miután az előző szálcsoport mindegyike befejeződött.



Fordítási direktívák

Vegyük sorra a fontosabb fordítási direktívákat és azok jelentését! Ezek a fordítás menetét befolyásolják a kívánt cél elérése érdekében.

- `#pragma omp parallel`

  Az utasítás (vagy blokk), ami előtt áll, több szálon is le fog futni.

- `#pragma omp parallel num_threads(4)`

Az utasítás (vagy blokk), ami előtt áll, 4 szálon fog lefutni. Ha rendelkezésre áll 4 processzormag, akkor ezek párhuzamosan futnak, ha nincs, akkor az ütemező dönt a végrehajtás sorrendjéről (egymástól függetlenül végrehajtva a szálakat).

- `#pragma omp parallel private(x,y) shared(a,b)`

A következő (párhuzamosan futó) blokkban szereplő változók közül `x` és `y` legyen minden szálon egyedi (több példány), `a` és `b` legyen minden szálon közös (egy példány) lesz. Explicit megadás nélkül a változók privátak lesznek.

- `#pragma omp parallel for`

A következő `for` ciklus iterációs lépéseit oszd fel a szálak között egyenletesen! A „ciklusváltozó” egyes értékeire a ciklusmag különböző processzormagokon futhat le. A tartományok azonos méretűek lesznek.

- `#pragma omp parallel for schedule(guided)`

A következő `for` ciklus lépéseit oszd fel a szálak között terheléstől függően! (A `guided` paraméter helyett megadhatunk `dynamic` vagy `static` értéket is kiegészítve a „csonk” mérettel.)

- `#pragma omp parallel for reduction(+:Sum)`

A következő `for` ciklus elosztott lefutása után (!) az egyes szálak privát `Sum` változóinak értékeit add össze és az eredményt tárold el szintén a `Sum` változóba! Más operátorokkal is használható, mint például a `*` vagy a `min` és `max` azonosítók.

- `#pragma omp parallel sections`

  Hozz létre párhuzamosan futó szakaszokat! Ezek tartalma lehet különböző.

- `#pragma omp section`

A következő blokk legyen az előző direktívával létrehozott egyik párhuzamosan futó szakasz! Ilyenekből tehát elhelyezhetünk többet is az `omp parallel sections` után.

- `#pragma omp master`

  A következő utasítást vagy blokkot csak a master szál futassa egyedül!

- `#pragma omp single`

A következő utasítást vagy blokkot csak az egyik szál futassa! Nem határozzuk meg melyik szál legyen az, majd az ütemező dönt.

- `#pragma omp barrier`

Hozz létre itt egy szinkronizációs pontot! Egyik szál sem mehet addig tovább, amíg mindegyik el nem ért idáig. Így lesz egy pillanat, amikor mindegyik szál egy ismert állapotban van.

- `#pragma omp critical`

A következő utasítás (vagy blokk) alkosson ún. kritikus szakaszt, azaz egyszerre csak egy szál futtathassa ezt a részt (a többiek csak azután hogy az előző befejezte). Ezzel valósíthatjuk meg a kölcsönös kizárást. Például egy `shared` változónak ne akarjon egyszerre egynél több szál értéket adni, mert az fizikailag megoldhatatlan és így az eredmény határozatlan lenne.

Könyvtári függvények

- `int omp_get_num_procs()`

  Visszaadja az elérhető processzormagok számát.

- `int omp_get_num_threads()`

A függvény visszaadja a szálak aktuális számát abban a folyamatban, amelyben éppen meghívták.

- `void omp_set_num_threads(int)`

Beállítja hány szálon fussanak a párhuzamos szakaszok. Egyenértékű a `num_threads` fordítási direktíva elemmel.

- `int omp_get_thread_num()`

  Visszaadja annak a szálnak az azonosítóját, amelyben meghívták.

Irodalom:

- Barbara Chapman, Gabriele Jost, Ruud van der Pas: *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation),* MIT Press (2008).