resources

Erőforrások kezelése:
a dinamikus memória és a file

A jegyzetben és a feladatok során két erőforrást ismerünk meg és használunk: a dinamikus memóriát és a file-t. Azonban minden erőforrás – például portok (usb, printer, ethernet, …) – kezelésére hasonló általános kijelentések mondhatóak és hasonlóan kell őket is kezelni. Ezek az általános szabályok:

- kérni kell a hozzáférést
- a hozzáférés nem garantált, ezt ellenőrízni szükséges
    - siker esetén használhatjuk
    - a sikertelen kérést kezelnünk kell
- az erőforrást fel kell szabadítani

Az elsőt nyilvánvaló okok miatt nem szokás elfelejteni, azonban a másik két teendő elfelejtődik, pedig a hozzáférés sikertelensége esetén jogosulatlan erőforrást használnánk (memória esetén segmentation fault hiba), a felszabadítás elmaradásakor pedig feleslegesen fogjuk az erőforrást, akár a program befejezése után is (memória esetén ezt memóriaszivárgásnak vagy angolul memory leaknek nevezzük). Az erőforrások felszabadítását meg kell tennünk akkor is, ha a program hibára fut.

Dinamikus memória

Fizikailag a memória egy és oszthatatlan. A *stack*, amit ezidáig használtunk, a memória egyik felén található, - a legtöbb architektúrán - a legnagyobb memóriacímről indítva lefelé haladva. Ezen lépdel a processzor és hajtja végre a tárolt utasításokat vagy a tárolt adatokat olvassa. Minden függvényhívásnál a stack tetejére kerül a függvény, argumentumjaival együtt és a függvényen belüli utasításokkal. Mikor visszatér a függvény, úgy a hívó utasítás memóriacímére ugrik vissza a processzor futása a *stack*-n, ennek következményeként pedig a magasabb (alacsonyabb) memórián lévő függvényünk minden lokálisa elveszik (törlés nem történik, ezért célszerű inicializálni egy változót, hogy ne az ott található memóriaszemét – értsd: más utasítások, adatok emlékei – legyen tárolva az aktuális változóban).

A memória másik feléről indítva ugyanígy foglalható. Ekkor a foglalt terület tetszőleges ideig a rendelkezésünkre áll, egészen annak felszabadításáig. Az ilyen memória több függvényen keresztül is használható.

A három feladatkört egyesével vizsgáljuk meg. Ezen műveletekhez a C nyelv standard *stdlib.h* könyvtárában implementált függvények állnak a rendelkezésünkre.

Hozzáférés kérése:

- *void * malloc(size_t size)*

  A megadott *size* méretű memóriát próbálja lefoglalni byte-ban mérve (a *size_t* architektúra és fordító függően valamilyen *unsigned* egész típus). Sikeres foglalás esetén visszatér egy általános *void* mutatóval a memória első címére, ellenkező esetben 0-val.

- *void * calloc(size_t num, size_t size)*

  Ritkábban használt allokáló függvény. A *num* számú és *size* méretű objektumok foglalására lehet használni, a foglalt memória nem feltétlenül a két szám szorzata. Sikeres foglalás esetén visszatér egy általános *void* mutatóval a memória első címére, ellenkező esetben 0-val. A sikeresen foglalt memória minden bitje nullára lesz inicializálva.

- *void* realloc(void * original, size_t new_size)*

  Az *original* mutató által hivatkozott *heap*-n található, már lefoglalt (és nem felszabadított) memóriaterületet allokálja át egy másik memóriaterületre, a *size* által meghatározott mérettel.

    - Sikeres foglalás esetén:
        - visszatér egy általános *void* mutatóval a memória első címére
        - a régi memóriaterület felszabadul
        - a régi memóriaterületen lévő adat átmásolódik az új területre
            - ha a *new_size* nagyobb, mint az eredeti, úgy az eredeti méret feletti memóriában memóriaszemét lesz (nincs inicializálva)
            - ha a *new_size* kisebb, mint az eredeti, úgy csak az ekkora méretnek megfelelő adat kerül másolásra, a többi elveszik
    - Sikertelen foglalás esetén:
        - null pointerrel tér vissza
        - a régi memóriaterület megmarad

Általános típusú adat nincs, azaz ha szeretnénk dolgozni a foglalt memóriával, konvertálni kell a megfelelő típusra. Ez megtehető implicite is. Például:

```
int * data = malloc(10 * sizeof(int)); // 10 méretű int tömb foglalása
```

ahol egy *(int*)* implicit cast történik foglalás után.

Ellenőrzés:

Mint a fenti függvények leírásánál láttuk, mindegyik allokáló függvény null pointerrel (szimplán 0-val) tér vissza, ha sikertelen a kérés. Ezt könnyen felhasználhatjuk a kérés sikertelenségének kezeléséhez:

```
if(data == 0) // ekvivalens ezzel: !data
{
// sikertelen foglalás kezelése
}
```

Sikertelen foglalást számos módon kezelhetünk, például újra megkíséreljük a foglalást, hibaüzenetet írunk ki vagy terminálunk a programot. A null pointeren keresztül természetesen nem érhető el semmi, felhasználása segmentation fault-t dob.

Felszabadítás:

Bármely módszerrel is foglalunk vagy újrafoglalunk memóriát sikeresen, annak felszabadítása a programozó feladata, melyet a következő standard könyvtári függvénnyel oldhatunk meg.

*void free(void * pointer)*

Paraméterként a felszabadítandó területre mutató pointer használjuk. A felszabadítás mindig sikeres. A felszabadított memóriát a továbbiakban nem használhatjuk, a jól ismert segmentation fault hibát eredményezi, ha mégis megtesszük. Célszerű a pointert 0-ra állítani a felszabadítás után.

Hibák:

- *memory leak*

  Amennyiben nem szabadítjuk fel a foglalt memóriát, úgy memóriaszivárgás történik. Tipikus példája, amikor a *free* utasítást meg sem adjuk, de rejtettebb módon is elkövethetjük ezt a hibát, amikor a pointert, ami a foglalt memóriára hivatkozik, egyszerűen átirányítjuk egy másik memóriacímre.

- *double free*

  A másik gyakran előforduló hiba, ha kétszer akarjuk ugyanazt a memóriaterületet felszabadítani. Háromszoros vagy többszörös felszabadítás nem létezik, mivel már a másodiknál futási hibát kapunk.

Feladatok:

- Foglaljunk le 12 byte dinamikus memóriát. Ellenőrízzük, hiba esetén írjunk ki hibaüzenetet. Írjunk rá három integer számot. Írjuk ki a memória tartalmát *printf*-fel. Szabadítsuk fel a pointeren keresztül.
  - Próbáljuk ki, mi történik, ha kétszer szabadítjuk fel a memóriát.
  - Próbáljuk ki, mi történik, ha nem 12 byte-t, hanem 12 Gbyte-t próbálunk foglalni.
  - Próbáljuk ki, mi történik, ha az első foglalás után egy másodikat hajtunk végre és az első foglaláshoz használt pointert használjuk a másodikhoz is.
  - Definiáljunk egy *char** pointert, ami a foglalt memóriaterületre mutat. Írjuk felül a tárolt adatot az ábc betűvel.
- Az önhivatkozó struktúrákat felhasználva, készítsük most el a teljes láncolt listát, azaz dinamikus memória kezelésével oldjuk meg a láncolt lista felépítését. A node adattagja legyen egyszerűen *int* típusú. Készítsük el az alábbi függvényeket:
  - *push_back*: a láncolt lista végére fűz egy elemet, a bejövő *int* argumentummal feltöltve
  - *push_front*: a láncolt lista elejére fűz egy elemet, a bejövő *int* argumentummal feltöltve
  - *pop_front*: a láncolt lista első elemét törli és visszatér a node-ban tárolt adattag értékével

- *pop_back*: a láncolt lista első elemét törli és visszatér a node-ban tárolt adattag értékével
- *insert*: az argumentumként kapott node (pointer) után helyez egy új node-t, a szintén argumentumként kapott adattal feltöltve
- *erase*: az argumentumként kapott node-t (pointer) törli

  Figyeljünk a node-k egymásra hivatkozásaira. Minden műveletnél újra kell értelmeznünk a kapcsolatokat!

File

A dinamikus memóriához hasonlóan végighaladunk a lépéseken. Azonban itt nem közvetlenül dolgozunk a tárolt adattal, azt különböző, a standard könyvtárakban megtalálható függvényekkel tudjuk írni és olvasni. Mást nem. Ahhoz be kell töltenünk a memóriába (onnan registerekbe), majd visszaírnunk azt a hard drive-ra. Az írást, olvasást egy *FILE** típusú pointeren keresztül valósítjuk meg. Ez sok információt tartalmaz, többek között indikátorokat, a típusát, a file-n belüli pozíció mutatóját. Bővebben: https://en.cppreference.com/w/cpp/io/c/FILE .

A szükséges standard könyvtári header itt a *stdio.h*.

Hozzáférés kérése:

FILE fopen(const char* filename, const char* mode)*

A file nevének megadása egy karakterláncon keresztül történik, melynek létezésének ellenőrzése nem a függvény feladata. A *mode* argumentum a *printf* formatálási argumentumához hasonló, de jóval korlátozottabb listából választhatunk: “r”, “w”, “a” az olvasás, írás és hozzáfűzés megadására, valamint mindegyik után tehetünk “+” karaktert, ekkor írni és olvasni is lehet. Megnyitható a file bináris formában is a “b” karakterrel. Bővebben: https://en.cppreference.com/w/cpp/io/c/fopen .

A függvény sikeres nyitáskor visszatér a file-ra mutató változóval. Ellenkező esetben null pointerrel. Például:

*FILE const * const fp = fopen(“temp.txt”, “w”);*

Ellenőrzés:

Nincs új a nap alatt, a null pointer ellenőrzést használjuk erre az esetre is: *if(fp == 0)*.

Felszabadítás:

A nem-null file pointeren keresztül a file bezárása:

*int fclose(FILE * fp)*

A függvény siker esetén 0-val tér vissza, EOF-fal ellenkező esetben (ritkán ellenőrízzük).

Bővebben: https://en.cppreference.com/w/cpp/io/c/fclose

File műveletek:

Sok standard könyvtári függvény áll a rendelkezésünkre írásra és olvasásra.

- *fscanf*: https://en.cppreference.com/w/c/io/fscanf: használata ugyanaz, mint a *scanf*, csak első argumentumként a file pointert átadjuk
- *fprintf*: https://en.cppreference.com/w/c/io/fprintf: szintén *printf* és argumentumként a file pointer
- előzőek mindent is olvasnak és írnak, karakterenkénti írásra és olvasásra a következő két függvény:
  - *fgetc*: https://en.cppreference.com/w/c/io/fgetc
  - *fputc*: https://en.cppreference.com/w/c/io/fputc
- soronkénti olvasás:
  - *getline*: https://en.cppreference.com/w/c/experimental/dynamic/getline

Feladatok:

- Írd file-ba a "Hello World!"-t.
  - Olvasd be egy másik programmal.
- Készíts egy file-t random tartalommal feltöltve, tetszőleges hosszúsággal.
  - Készíts egy másik programot, ami ezt beolvassa és kiírja terminal-ra a szöveget.
  - Módosítsd a programot, hogy tárolja is a beolvasott szöveget. Tegyük fel, hogy maximum 124 karakter hosszú lehet egy sor és maximum 32 sor van.
  - Módosítsuk a programot, hogy fordítva írja ki a file tartalmát terminalra. Utolsó sor elsőként és minden sor utolsó karaktere elsőként.
  - Módosítsuk a programot, ami nem tartalmaz beégetett konstans-t, hanem dinamikusan foglaljunk memóriát és oda töltsük be a file tartalmát.

Extrém Haladó : Memory pool

A memory pool a valós fizikai memória egy lokális, a programunkra megvalósított módszer. Lényege, hogy a program a futásának elején foglal egy nagyobb méretű (a *heap*-n) memóriát és a program terminálása előtt szabadítja fel csak azt. Ugyanis mint mindennek, a dinamikus memóriakezelésnek is van költsége és ha sok memóriaallokáció és felszabadítás van, az erősen tudja lassítani a program futását.

A memory pool továbbá egy jó absztrakció a valós, az operációs rendszer általi memóriakezelésre is.

Részletesen:

- van egy *void * const* pointerünk, többnyire globális, ez mutat a memory pool elejére
  - a program indulásakor rögtön foglalunk dinamikus memóriát, általános típussal (nincs cast-olás)
  - a program végeztével felszabadítjuk ezt
- van egy *void * *mutatónk a memory pool-ban az aktuálisan elérhető memóriacímre, célszerű ezt is globálisnak deklarálni

- a foglalt memória méretéből, az aktuális és a kezdő memóriacímből megállapítható, hogy mennyi szabad lokális memóriánk van még:
- amikor szeretnénk memóriát "foglalni" adott mérettel, úgy egy saját függvény módosítja az aktuális memóriacímet a foglalási méretnek megfelelően és visszatér a régi aktuális memóriacímmel: így az ezen a memóriacímen elérhető memóriát tudja a hívó használni, természetesen a "foglalt" méretig
  - akár 0-ra is állíthatjuk a "foglalt" memóriát
- felszabadítás esetén tudnunk kell, mennyi memóriát is kell "elengednünk":
  - megoldhatjuk ezt egy tömbbel (*void** tömb), mely tárolja a memóriacímeket a program objektumainak elejére
  - a felszabadító függvényünknek átadjuk a felszabadítandó memória méretét