Hatékony memóriakezelési technikák

Smidla József
Operációkutatási Laboratórium
2014. január 16.

Tartalom

- A cache áthidalása
- Cache optimalizálás
- Adatszerkezetek tervezése
- A programkód szerkezete
- Prefetch
- Optimalizált memória menedzser

A cache áthidalása

- A cache-ben olyan adatokat érdemes tárolni, amiket sűrűn használunk
- Írás a memóriába:
  - a célterületet a cache-be olvassuk
  - az információt a cache-be írjuk
  - majd valamikor ezt az információt visszaírjuk a cache-be
- Nagy memóriaterületek másolásakor elvesznek a fontosabb információk a cache-ben

A cache áthidalása

- Használjunk olyan írás utasítást, amely nem olvassa be a célterületet, hanem közvetlenül a memóriába ír

```
#include <emmintrin.h>
void _mm_stream_si32(int *p, int a);
void _mm_stream_si128(int *p, __m128i a);
void _mm_stream_pd(double *p, __m128d a);
#include <xmmintrin.h>
void _mm_stream_pi(__m64 *p, __m64 a);
void _mm_stream_ps(float *p, __m128 a);
#include <ammintrin.h>
void _mm_stream_sd(double *p, __m128d a);
void _mm_stream_ss(float *p, __m128 a);
```

A cache áthidalása

- Probléma: Ha sokszor írunk kis adatokat a memóriába, az rontja a teljesítményt
- Megoldás: Write-combining
- A módszert főleg perifériák esetén (például videokártyák) alkalmazzák
- Egy speciális bufferben gyűjtik az írandó adatokat
- Ha elég adat összejött, akkor utána kerülnek kiírásra

A cache áthidalása

- Példa: Állítsuk egy buffer minden bájtját azonos értékűre

```
#include <emmintrin.h>
void setbytes(char *p, int c)
{
  __m128i i = _mm_set_epi8(c, c, c, c,
                           c, c, c, c,
                           c, c, c, c,
                           c, c, c, c);
  _mm_stream_si128((__m128i *)&p[0], i);
  _mm_stream_si128((__m128i *)&p[16], i);
  _mm_stream_si128((__m128i *)&p[32], i);
  _mm_stream_si128((__m128i *)&p[48], i);
}
```

A cache áthidalása

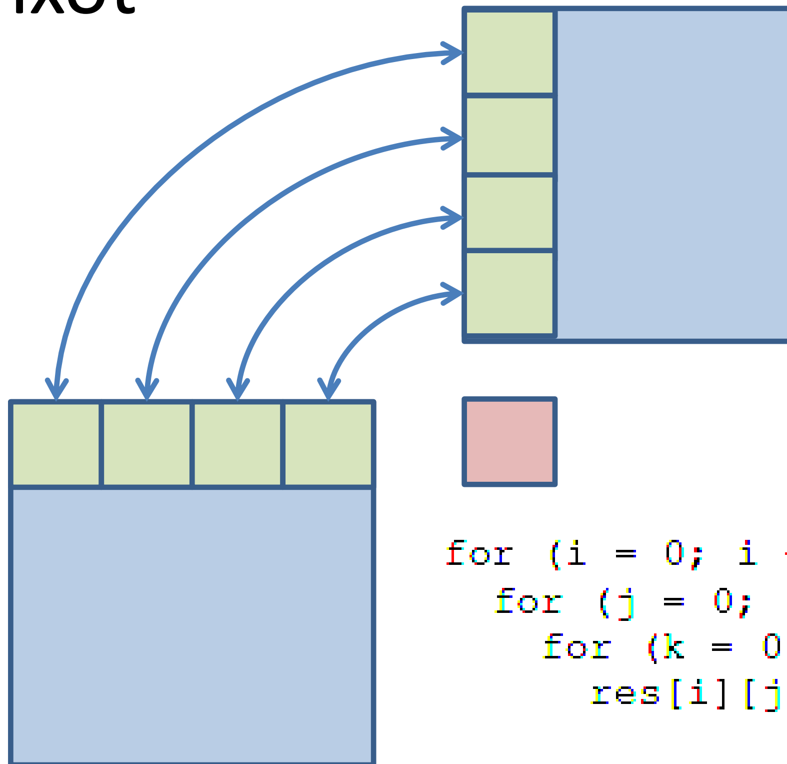
- Olvasás a memóriából cache nélkül
- A programozók az XMM, vagy YMM regiszterekbe tudnak beolvasni adatokat (128, illetve 256 bites regiszterek)
- movntdqa utasítás assemblyben

```
#include <smmintrin.h>
__m128i _mm_stream_load_si128 (__m128i *p);
```

- A cache helyett egy speciális bufferbe olvas
- Fontos: ezek az utasítások 16 bájtos határra igazított memóriacímeket várnak!

Cache optimalizálás: mátrix szorzás

- Szorozzunk össze két 1000*1000 méretű mátrixot



```
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    for (k = 0; k < N; ++k)
      res[i][j] += mul1[i][k] * mul2[k][j];
```

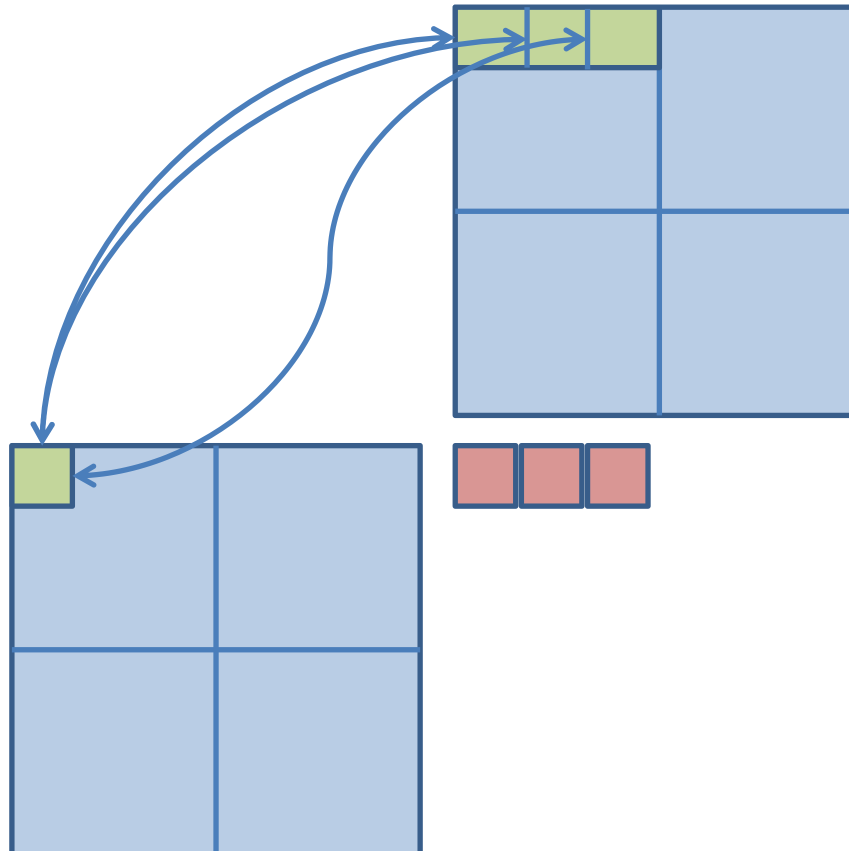
Mátrix szorzás

- Probléma: A C nyelv sorfolytonosan tárolja a mátrixokat
- A cache-t jobban kihasználhatjuk, ha transzponáljuk a jobb oldali mátrixot

```
double tmp[N][N];
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    tmp[i][j] = mul2[j][i];
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    for (k = 0; k < N; ++k)
      res[i][j] += mul1[i][k] * tmp[j][k];
```

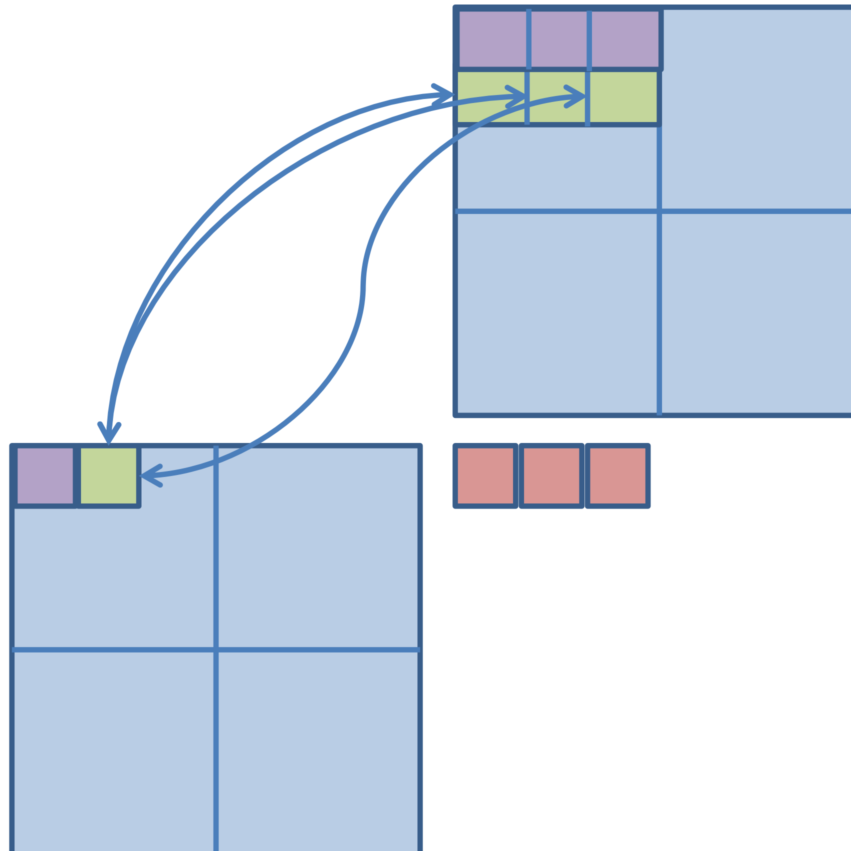
Mátrix szorzás

- Bontsuk blokkokra a mátrixot úgy, hogy kis cache-ben is elférjünk



Mátrix szorzás

- Majd odébb mozdulunk a blokkon belül



Mátrix szorzás

```
#define SM (CLS / sizeof (double))

for (i = 0; i < N; i += SM)
  for (j = 0; j < N; j += SM)
    for (k = 0; k < N; k += SM)
      for (i2 = 0, rres = &res[i][j],
           rmul1 = &mul1[i][k]; i2 < SM;
           ++i2, rres += N, rmul1 += N)
        for (k2 = 0, rmul2 = &mul2[k][j];
             k2 < SM; ++k2, rmul2 += N)
          for (j2 = 0; j2 < SM; ++j2)
            rres[j2] += rmul1[k2] * rmul2[j2];
```

Mátrix szorzás

- Végül használhatunk SIMD utasításokat a műveletek alacsony szintű párhuzamosítására (vektorizálás)

	Original	Transposed	Sub-Matrix	Vectorized
Cycles	16,765,297,870	3,922,373,010	2,895,041,480	1,588,711,750
Relative	100%	23.4%	17.3%	9.47%

Adatszerkezetek tervezése

- Hozzunk létre egy C struktúrát
- Nézzük meg, hogy a struktúra elemei hogyan helyezkednek el a memóriában

test.c

```c
struct foo {
    int a;
    long fill[7];
    int b;
};

int main() {

    struct foo f;

    return 0;
}
```

Struktúrák

- Használjuk a linuxos pahole programot:

smidla@notesmidla: ~/DCS/panopt

Fájl Szerkesztés Nézet Keresés Terminál Súgó

```
smidla@notesmidla:~/DCS/panopt$ gcc -g -o test test.c
smidla@notesmidla:~/DCS/panopt$ pahole test
struct foo {
        int                          a;                    /*     0     4 */

        /* XXX 4 bytes hole, try to pack */

        long int                     fill[7];              /*     8    56 */
        /* --- cacheline 1 boundary (64 bytes) --- */
        int                          b;                    /*    64     4 */

        /* size: 72, cachelines: 2, members: 3 */
        /* sum members: 64, holes: 1, sum holes: 4 */
        /* padding: 4 */
        /* last cacheline: 8 bytes */
};
smidla@notesmidla:~/DCS/panopt$
```

Struktúrák

- A vizsgált struktúra elemeinek összege 64 bájt
- A fordító az első elem után kihagyott 4 bájtot
- 64 bájtos cache line esetén ez a struktúra nem fér el egy cache line-ban
- Megoldás: rendezzük át a struktúra adattagjait!

test.c

```c
struct foo {
    int a;
    int b;
    long fill[7];
};

int main() {

    struct foo f;

    return 0;
}
```

Struktúrák

- A pahole kimenete:

smidla@notesmidla: ~/DCS/panopt

Fájl Szerkesztés Nézet Keresés Terminál Súgó

```
smidla@notesmidla:~/DCS/panopt$ gcc -g -o test test.c
smidla@notesmidla:~/DCS/panopt$ pahole test
struct foo {
        int                         a;                    /*     0     4 */
        int                         b;                    /*     4     4 */
        long int                    fill[7];              /*     8    56 */
        /* --- cacheline 1 boundary (64 bytes) --- */

        /* size: 64, cachelines: 1, members: 3 */
};
smidla@notesmidla:~/DCS/panopt$
```

- A struktúra most elfér egy cache line-ban

Struktúrák

- Egyéb szempontok:
- A struktúra elejére helyezzük el azokat az adattagokat, amelyeket a legnagyobb valószínűséggel használunk
- Ha tehetjük, akkor olyan sorrendben járjuk végig a struktúra elemeit, amilyen sorrendben definiáltuk őket

Memória objektumok igazítása

- A struktúra elemeinek gondos átrendezése nem ér semmit, ha a struktúra nem egy cache line határán kezdődik
- Ekkor átlóghat egy másik cache lineba
- Hozzuk létre az objektumokat úgy, hogy a cache line-ok határaira illesszük őket (azaz 32 vagy 64 bájtos határokra)

Memória objektumok igazítása

- Dinamikus objektumok lefoglalására használjunk speciális lefoglaló függvényt a malloc helyett, Linux alatt a posix_memalgin-t:

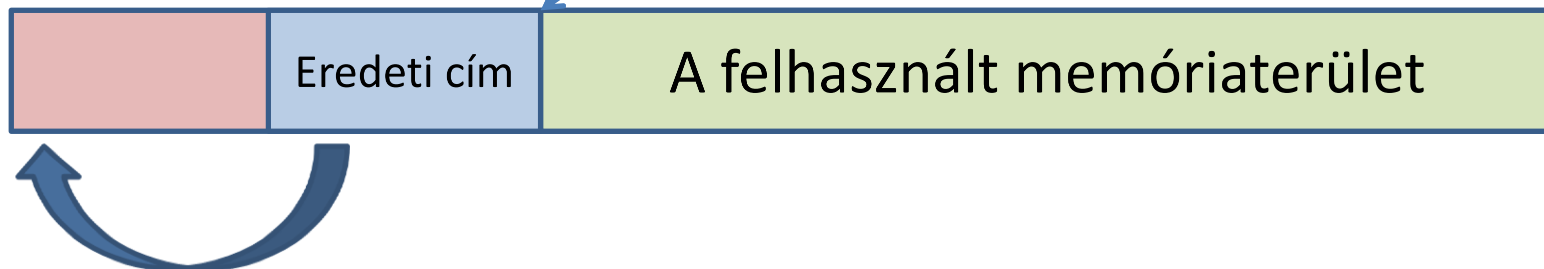
```
#include <stdlib.h>
int posix_memalign(void **memptr,
                   size_t align,
                   size_t size);
```

Memória objektumok igazítása

- Mit tehetünk, ha nem áll rendelkezésünkre ilyen memória foglaló? Írjunk egyet!
- Foglaljunk le egy némileg nagyobb buffert
- A bufferen belül keressünk egy alkalmas határt
- Ezt a határt adjuk vissza
- Felszabadításkor az eredeti címre van szükség
- Az eredeti címet rejtsük el a visszaadott cím elé

Memória objektumok igazítása

Ezt a címet adjuk vissza



A felszabadító megkapja a visszaadott címet, eltolja balra, kiolvassa az eredeti címet, majd meghívja a free-t az eredeti címre

Igazított lefoglalás

```
union Pointer {
    void * ptr;
    unsigned long int bits;
};

void * alloc(int size) {
    char * originalPtr = static_cast<char*>(malloc(size + 64 + sizeof(void*)));
    char * ptr2 = originalPtr + 64 + sizeof(void*);
    Pointer ptr;
    ptr.ptr = ptr2;
    ptr.bits >>= 6;
    ptr.bits <<= 6;
    ptr2 = static_cast<char*>(ptr.ptr);
    ptr2 -= sizeof(void*);
    *((char**)ptr2) = originalPtr;
    return ptr2 + sizeof(void*);
}
```

Felszabadítás

- Kinyerjük az eredeti címet, majd a teljes buffert felszabadítjuk

```
void release(void * ptr) {
    char * ptr2 = (char*)ptr;
    ptr2 -= sizeof(void*);
    free(*((char**)ptr2));
}
```

Statikus objektumok igazítása

- A gcc a változók és struktúrák számára biztosít egy speciális kulcsszót, amellyel a fordítót utasíthatjuk arra, hogy igazítottan helyezze el a változókat a memóriában

```
struct strtype variable
    __attribute((aligned(64)));

struct strtype {
   ...members...
} __attribute((aligned(64)));
```

A programkód szerkezete

- A programkódot a processzor az L1i cache-be tölti be
- Az utasítások általában egymás után jönnek
- Ez előnyös, mert lehet prefetch-elni a kódot
- Lehet alkalmazni a pipeline technikát
- Az ugró utasítások lerontják a teljesítményt, mert sok utasítást feleslegesen prefetch-eltünk, és a pipeline is megtörik

A programkód szerkezete

- Optimalizáljuk úgy a kódot, hogy minél kevesebb ugrást tartalmazzon
- Ehhez nem elég a C nyelv ismerete
- Például az int a = b > 4 ? 2 : 1; tartalmazhat ugrást gépi kód szinten
- A függvényhívások is megtörik a pipeline-t, használjunk inline függvényeket
- Inline függvények esetén a fordító nagyobb kódrészletet képes optimalizálni

Inline függvények

- Bánjunk óvatosan az inline függvényekkel!
- Az inline függvények nagyobb kódot eredményeznek
- A nagyobb kód nehezebben fér el a cache-ben, ez pedig ronthatja a teljesítményt
- A kisebb kód hatékonyabb

Inline függvények

```
start f1
  code f1
  inlined inlcand
  more code f1
end f1

start f2
  code f2
  inlined inlcand
  more code f2
end f2
```

```
start inlcand
  code inlcand
end inlcand

start f1
  code f1
end f1

start f2
  code f2
end f2
```

Inline függvények

- Vannak esetek, biztosítanunk kell, hogy az adott függvény mindig inline legyen
- A hagyományos inline kulcsszó csak egy jelzés, a fordító felülbírálhatja
- A gcc biztosít egy kulcsszót, amellyel utasíthatjuk a fordítót, hogy mindig inline legyen a függvény:
- `__attribute__((always_inline)) void foo();`

Ugrások optimalizálása

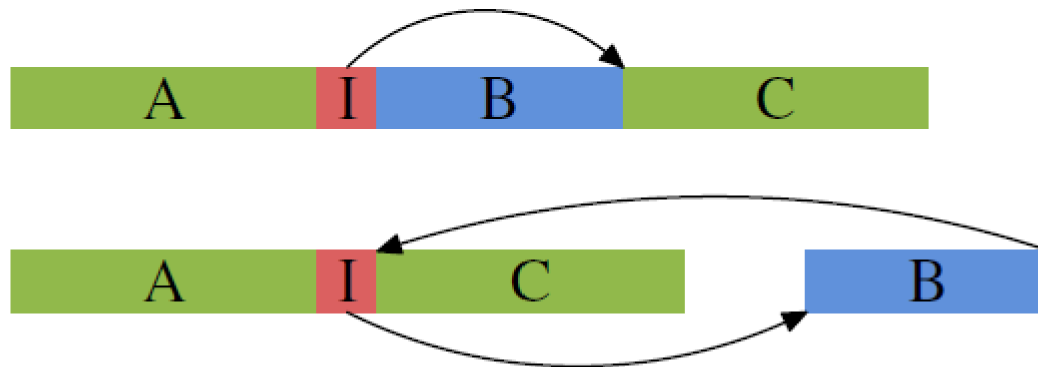
- Tekintsük az alábbi kódot:

```
void fct(void) {
  ... code block A ...
 if (condition)
   inlfct()
 ... code block C ...
}
```

- Tegyük fel, hogy a feltétel nagy valószínűséggel hamis, azaz csak ritkán hajtuk végre a feltétel igaz ágát

Ugrások optimalizálása

- Az ilyen ugrások miatt hiába prefetch-eljük a kód igaz ágát, valamint a pipeline-t is megtörjük, hiszen ha a feltétel igaz, akkor az igaz ágat át kell ugrani
- Rendezzük át a kód blokkokat:



Ugrások optimalizálása

- A fordítót felszólíthatjuk az ilyen blokk átrendezésekre egy speciális kulcsszóval:

```
long __builtin_expect(long EXP, long C);
```

- Ebből létrehozhatunk két makrót is:

```
#define unlikely(expr) __builtin_expect(!!(expr), 0)
#define likely(expr) __builtin_expect(!!(expr), 1)
```

- Példa: az a > 1 feltétel az esetek többségében igaz:

```
if (likely(a > 1))
```

- Ehhez szükség van a –O2 kapcsolóra is!

Prefetch

- A processzor igyekszik felismerni, hogy mely cache line-okra lesz szükségünk, így ezeket előre beolvassa, prefetch-eli
- Például ha azonos távolságra lévő adatokhoz férünk hozzá egymás után, akkor ezt a processzor felismeri
- Probléma: ez nem működik, mikor a programunk nem sorban halad a memóriában

Prefetch

- Amennyiben nem tudjuk biztosítani, hogy az adatokat sorban dolgozzuk fel, használjunk szoftveres prefetch-et!

```
#include <xmmintrin.h>
enum _mm_hint
{
   _MM_HINT_T0 = 3,
   _MM_HINT_T1 = 2,
   _MM_HINT_T2 = 1,
   _MM_HINT_NTA = 0
};
void _mm_prefetch(void *p,
                  enum _mm_hint h);
```

Optimalizált memória menedzser

- Ha túl sűrűn használjuk a malloc / free, vagy new / delete párost, az érezhetően csökkentheti a teljesítményt
- A felszabadítás hatására a felszabadított memóriaterület visszajut a heap-be
- Később egy újabb lefoglalás hatására akár ezt a korábban felszabadított területet is visszakaphatjuk
- A lefoglalás és felszabadítás sok kommunikációt és adminisztrációt igényel a rendszer más részeivel

Optimalizált memória menedzser

- Megoldás: Fejlesszünk saját memória menedzsert (vagy használjunk egy mások által megírt, hasonló célú megoldást)
- Elv:
- Lefoglalunk egy buffert malloc-al, és visszaadjuk
- A felszabadító rutinunk nem hívja meg a free-t, hanem a felszabadítandó buffert egy láncolt listába fűzi
- Később a lefoglaló rutin megnézi, hogy van-e szabad buffer a láncolt listában, ha van, akkor visszaadja azt, egyébként meg foglal egyet malloc-al

Optimalizált memória menedzser

- Előzzük meg a memória töredezését: A nagyjából azonos méretű puffereket ugyan abba a láncolt listába fűzzük
- Példa: Használjunk külön láncolt listát a 4, 8, 12, 16… 1024 méretű pufferek számára, majd alkalmazzunk 1024, 2048, stb méretű puffereket tartalmazó listákat
- A lefoglaló rutin O(1) idő alatt el tudja dönteni, hogy melyik listára van szükség

Optimalizált memória menedzser

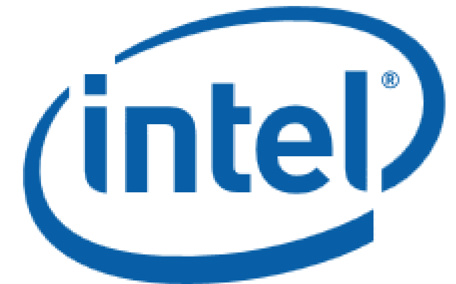
- A felszabadító függvény csak egy pointert kap, de valahonnan tudnia kell, hogy melyik láncolt listába kell befűzni a buffert
- Alkalmazzuk az igazított memóriafoglalásnál látott technikát, azaz a lefoglaló rutin helyezze el a szükséges információkat a visszaadott pointer által címzett terület elé
- A felszabadító függvény kiolvashatja ezeket az információkat

Optimalizált memória menedzser

- Tovább fokozhatjuk a teljesítményt, ha a program indulásakor előre lefoglalunk egy nagyobb buffert (például 1 Mb)
- Ebből a nagyobb bufferből csípünk le kisebb buffereket igény szerint
- Ha ez a nagy buffer elfogy, akkor újat foglalunk le
- A nagy buffereket is fűzzük láncolt listába

Források

- Ulrich Drepper: What Every Programmer Should Know About Memory



Intel® 64 and IA-32 Architectures Software Developer's Manual

Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z

Köszönöm a figyelmet!

A publikáció az Európai Unió, Magyarország és az Európai Szociális Alap társfinanszírozása által biztosított forrásból a TÁMOP-4.2.2.C-11/1/KONV-2012-0004 azonosítójú "Nemzeti kutatóközpont fejlett infokommunikációs technológiák kidolgozására és piaci bevezetésére" című projekt támogatásával jött létre.