Szemétgyűjtés

Memóriakezelés – statikus memóriafoglalás

- Minden memóriát a fordító osztott ki, a változók tárhelye rögzített
- Nincs változó méretű memória
  - Fix méretű tömbök, nincs láncolt lista
  - Nincs rekurzió
- Megbízhatóság: nem fogyhat el a memória menet közben
- Fortran, '50-es évek

Memóriakezelés – automatikus memóriafoglalás

- Eljárás hívásakor memória foglalódik le a paramétereknek, lokális változóknak a stacken (stack frame)
- Eljárás befejeződésekor a stack frame felszabadul, folytatódik a hívó eljárás futása
- Van rekurzió
- Különböző méretű stack frame-k jöhetnek létre, de a visszatérési érték mérete rögzített
- Elfogyhat a memória rekurzió során

Memóriakezelés – dinamikus memóriafoglalás

- Különböző méretű memória foglalása, felszabadítása
- Hívott eljárás tetszőleges méretű eredményt visszaadhat
- Dinamikus adatszerkezeteket támogat: láncolt lista, fa
- Ha nincs felszabadítva a memória, miután már nem kell, szeméttévá válik, és nem lesz elérhető
- Memóriafoglalás költségesebb az előzőekhez képest

Memóriakezelés

- A mai nyelvek támogatják mindhárom memóriakezelést
- Számít, melyiket választjuk

```
void foo()
{
    MyType m1;                                          // 1. stack foglalás
    MyType* m2 = AllocateOnHeap(sizeof(MyType)); // 2. heap foglalás
    ...
    if (SomeCond)
       foo();      // rekurzív hívás
    ...
    DeAllocate(m2);
}
```

Szemétgyűjtés

- Memóriát csak foglalni kell, felszabadítás magától zajlik
- Szemét gyűjtése és csellengő pointerek elkerülése
- Költség: nem használják valós idejű rendszerekben, eszközmeghajtókban, játékokban.

Referenciaszámlálás

```
Object * obj1 = new Object(); // RefCount(obj1) 1-ről indul
Object * obj2 = obj1;         // RefCount(obj1) nő 2-re
Object * obj3 = new Object();

obj2->SomeMethod();
obj2 = NULL;                  // RefCount(obj1) csökken 1-re
obj1 = obj3;                  // RefCount(obj1) csökken 0-ra, felszabadítható
```

Referenciaszámlálás

```
// cb is the number of bytes to be allocated
PVOID GC_Alloc(size_t cb)
{
    MemHeader* pHdr = (MemHeader*)PlatformAlloc(MEMHEADERSIZE + cb);
    if (pHdr == NULL)
        return NULL;

    pHdr->refCount = 1;

    ++pHdr;

    return (PVOID)pHdr;
}
```

Referenciaszámlálás

- Felszabadításkor be kell járni a mutatott objektumokat is
- Előny
  - A szemét azonnal felszabadul
  - A felszabadítás eloszlik a futás során, nem kell megállítani a programot
- Hátrány
  - Ez így ebben a formában nem kezeli a körkörös hivatkozásokat
  - Extra tárhelyigény
  - Minden egyes pointer értékadásnál kell frissíteni kell a számlálót

Mark and sweep

- Nem követi nyomon az objektumokat, nem szabadít fel azonnal
- A szemétgyűjtés egy eseményre indul be (pl. fogytán a memória, de még van)

Mark and sweep

1. Gyökerek felsorolása
   - Futtatórendszer ad támogatást
2. Objektumok bejárása, megjelölése
   - Megállás levélnél vagy kör detektálásakor
3. Jelöletlen objektumok felszabadítása, jelölők visszaállítása
4. Tömörítés

Mark and sweep – indulás

```
void GC()
{
    HaltAllProcessing();
    ObjectCollection roots = GetRoots();
    for(int i = 0; i < roots.Count(); ++i)
        Mark(roots[i]);
    Sweep();
    ContinueProcessing();
}
```

Mark and sweep – bejárás, jelölés

```
void Mark(Object* pObj)
{
    if (!Marked(pObj))
    {
        MarkBit(pObj);

        ObjectCollection children = pObj->GetChildren();
        for(int i = 0; i < children.Count(); ++i)
        {
            Mark(children[i]);
        }
    }
}
```

~~

Mark and sweep – takarítás, jelölő visszaállítása

```
void Sweep()
{
    Object *pHeap = pHeapStart;
    while(pHeap < pHeapEnd)
    {
        if (!Marked(pHeap))
            Free(pHeap);
        else
            UnMarkBit(pHeap);

        pHeap = GetNext(pHeap);
    }
}
```

Mark and sweep

- Előnyök
  - Kezeli a körkörös hivatkozásokat
  - Értékadás során nincs extra költség
  - Tömörítés közel helyezi az objektumokat egymáshoz: jobb teljesítmény
- Hátrányok
  - Hosszú megállást okozhat (de: inkrementális szemétgyűjtő)
  - Nem jó valós idejű rendszereknél

Másoló szemétgyűjtő

- Két régió, egyszerre egy van használatban
- Ha betelik, a nem szemét objektum átmásolódik a másik régiba

Másoló szemétgyűjtő

- Előnyök
  - Gyors foglalás
  - Gyorsan detektálja, ha elfogyott a memória
  - Az objektumok közel vannak egymáshoz
- Hátrány
  - Kétszeres memóriaigény

Generációs szemétgyűjtés

- A legtöbb objektum rövid életű
- A szemét 90%-a az előző szemétgyűjtő ciklus óta keletkezett
- Az objektum, mely túlél egy szemétgyűjtési ciklust, kevés valószínűséggel lesz szemét rövidtávon

Generációs szemétgyűjtés

- Objektumok szétválogatása életkor szerint
- A szemétgyüjtés különböző gyakorisággal
- Csökken az átvizsgálandó objektumok száma