

Android alkalmazásfejlesztés

Optimalizálás

Hatékony alkalmazás Androidra

Sicz-Mesziár János

sicz.mj@gmail.com

2011. május 13.

OE-NIK



Miről is lesz szó?

Hogyan optimalizáljunk teljesítményre
Android rendszer alatt.

© Források:

- [Designing for Performance](#)
- Google I/O
- Saját tapasztalatok
- Hello Android – Ed Burnette
- Ti kódjaitokban látottak



Irányelvek

- ◉ Első sorban arra kell törekedni, hogy jó programot írjunk, ne minden áron gyorsat!
- ◉ Teljesítmény szempontjából fontoljuk meg az API-k tervezését, használatát.
- ◉ Mérjük a teljesítményt az optimalizálás előtt és után.
- ◉ Optimalizáljunk, ahol ésszerű, és lehetséges, **de ne rombolja a felhasználói élményt.**
- ◉ És akkor a felhasználói élmény:

Hasonló 😊



◎ Objektumok létrehozásának elkerülése:

- Például több dimenziós tömbök helyett, 2 párhuzamos egy dimenziós tömb használata.

```
Class X{  
    Foo a;  
    Bar b;  
}
```

```
Foo[];  
Bar[];
```

◎ Belső Getter/Setter használatának mellőzés:

- OOP elvek követése erősen ajánlott. Kifelé public Getter/Setter használata, de belső értékadás közvetlenül történjen!

```
Class X{  
    private int a;  
    public void do(){  
        setA(1027);  
    }  
}
```

```
Class X{  
    private int a;  
    public void do(){  
        this.a = 1027;  
    }  
}
```

Gyorsítás: **3x**
JIT-el: **7x**

◎ ENUM használatának elkerülése

- ENUM használata kényelmes, de ne használjuk ha a sebesség számít! Helyette alkalmazzunk integer egészeket!

◎ Static használata

- Ha nem szükséges egy objektum mezőjéhez hozzáférni, akkor érdemes static megkötést használni.

Gyorsítás:
15-20%

◎ Final static megkötés konstansoknál

- A fordító generál egy osztály inicializálót (<clinit>), ami első használatkor fut le.
- Ha static megszorítást használunk, akkor a továbbiakban nincs szüksége a <clinit>-re.

```
static int intVal = 42;  
static String strVal = "Hello, world!";
```

```
static final int intVal = 42;  
static final String strVal = "Hello, world!";
```

Ez az optimalizálás csak primitív típusokra és String konstansokra érvényes!

For(each) előnyben részesítése

```
static class Foo { int mSplat; }  
Foo[] mArray = ...
```

```
public void zero() {  
    int sum = 0;  
    for (int i = 0; i < mArray.length; ++i)  
        sum += mArray[i].mSplat;  
}
```

Leglassabb:

Mert a JIT még nem tudja optimalizálni a tömb hosszának egyszeri számítását.

```
public void one() {  
    int sum = 0;  
    Foo[] localArray = mArray;  
    int len = localArray.length;  
    for (int i = 0; i < len; ++i)  
        sum += localArray[i].mSplat;  
}
```

Gyorsabb:

Mindent helyi változóba tesz → csökkenti a kereséseket. Tömb hosszának számítása gyorsabb.

```
public void two() {  
    int sum = 0;  
    for (Foo a : mArray)  
        sum += a.mSplat;  
}
```

Leggyorsabb:

Gyorsulás a JIT nélküli készülékeken. De a JIT-el rendelkezőkön nincs észlelhető különbség az előző megoldással szemben.

Rendszer API-k és egyéb trükkök

◎ StringBuilder

- String: ha a szöveg nem változik
- StringBuffer: változik a szöveg – több szálon (thread safe)
- StringBuilder: változik a szöveg, **gyorsabb** – csak 1 szálon (ha a szöveg hosszát előre megadjuk **még gyorsabb**)

◎ System.arraycopy()

Gyorsítás: **9x**

- Körülbelül 9x gyorsabb egy Nexus One készüléken - JIT-el, mintha kézzel írnánk meg.

◎ Listener objektumok elkerülése

Futási időben spórolunk: **1KB**

- Listener-ek megvalósításakor inkább a this kulcsszót használjuk, új Listener objektumok helyett!

◎ Logika: & vs. &&

◎ Ciklusok megszakítása break; utasítással

Számok

◎ Lebegőpontos számokról jó tudni

- Android készülékeken szemmértékre a lebegőpontos ábrázolás 2x lassabb, mint az egészszámok esetén.
Lásd.: **Location(double, double) vs. GeoPoint(int, int)**
- Sebességre a float és a double között nincs különbség. 😊
De a double 2x nagyobb. → ha lehet float-ot használjunk!

◎ Shiftelés

- Ha kettő hatványaival végzünk osztást, vagy szorzást, akkor a biteltolás módszere sokkal gyorsabb.

```
int a = 4320;  
int x = a / 2; // 2160  
int x = a / 4; // 1080  
int x = a / 8; // 540  
int x = a * 2; // 8640  
int x = a * 4; // 17280
```

```
int a = 4320;  
int x = a >> 1;  
int x = a >> 2;  
int x = a >> 4;  
int x = a << 1;  
int x = a << 2;
```


Teljesítmény mérése

- ⦿ Ajánlott optimalizálás előtt és után is mérni.



- ⦿ Így látni fogjuk, hogy a gyorsítás ért-e egyáltalán valamit.

- ⦿ Példakód az idő mérésére:



```
long start = System.currentTimeMillis();
```

```
/* Kódok, amelyek teljesítményére kíváncsiak vagyunk. */
```

```
long end = System.currentTimeMillis();
```

```
Log.i("M", String.valueOf(end - start));
```

Memory leak

◎ Drawable, Bitmap resource-ok

- Telefon megdöntésekor az Activity újraindul és újratölti a forrásokat. → Képek esetén ez memória szivárgást jelent.
- Megoldás:

```
static Bitmap d;  
public void onCreate(Bundle ...){  
    if(d == null) d = Bitmap.decodeResource(...);  
}
```

◎ Erőforrás felszabadításokról ne feledkezzünk meg!

- DB.close();
- Input/OutputStream.close();
- Bitmap.recycle();
- Camera.release();
- System.GC(); // Csak ha szükségesnek látjuk

UI gyorsítások

◎ Background drawable eltávolítása

- Alapértelmezett háttér eltávolítása gyorsít. (Csak ha nincs rá szükségünk, mert sajátot használunk)
- Gyorsulás oka a memória buszsebességéből ered.

```
<resources>  
  <style name="Theme.NoBackground" parent="android:Theme">  
    <item name="android:windowBackground">@null</item>  
  </style>  
</resources>
```

◎ Gyors orientáció váltás

- AndroidManifest.XML / adott Activity :
configuration change = "orientation"
- Következményei:
 - Döntéskor nem indul újra az életmodell ciklus.
 - Nem működik az alternatív minősítő az orientációra.

UI gyorsítások (2)

◎ ListView widget ListItem újrahasznosítása

- Egy ListView-nak sok eleme lehet (akát több ezer), így működése során a régi ListItem-eket újrahasznosítja.

◎ Touchscreen érintésének eseménygyakorisága

- A DOWN és az UP action jellemzően egy érintés alatt 1x-1x fut le, míg MOVE számtalanszor a mozgás alatt.



Ennek ismeretében **összehasonlítást spórolhatunk**, ha MOVE action-t előbb vizsgáljuk!

```
switch(event.getAction()){  
    case MotionEvent.ACTION_MOVE: break;  
    case MotionEvent.ACTION_DOWN: break;  
    case MotionEvent.ACTION_UP: break;  
}
```

Adatbázis gyorsítások

- ◎ Csak azon adatok lekérése, amikre szükségünk van.
 - Luxus a * alkalmazása → ha nincs szükségünk valamelyik mező értékére, akkor ne is kérjük le feleslegesen!
- ◎ Elsődleges kulcs használata
 - Mindig használjunk elsődleges kulcsot! (ID) Gyorsabb a kívánt sor megtalálása.
- ◎ Egy tábla sorainak száma

Hallgató kódja

```
Cursor c = adatb.rawQuery("Select * from fotabla", null);  
Log.d("NIK", "Count c: " + String.valueOf(c.getCount()));
```

```
Cursor c = adatb.rawQuery("Select count(1) from fotabla", null);  
int count = c.getInt(1);
```

- ◎ Multi-insert használata több sor beviteléhez

```
for(Data d : datas) db.insert(...);
```

```
db.rawQuery("insert into fotabla values(...),(...),(...)");
```

További gyorsítások

◎ Natív fejlesztés JNI-n keresztül

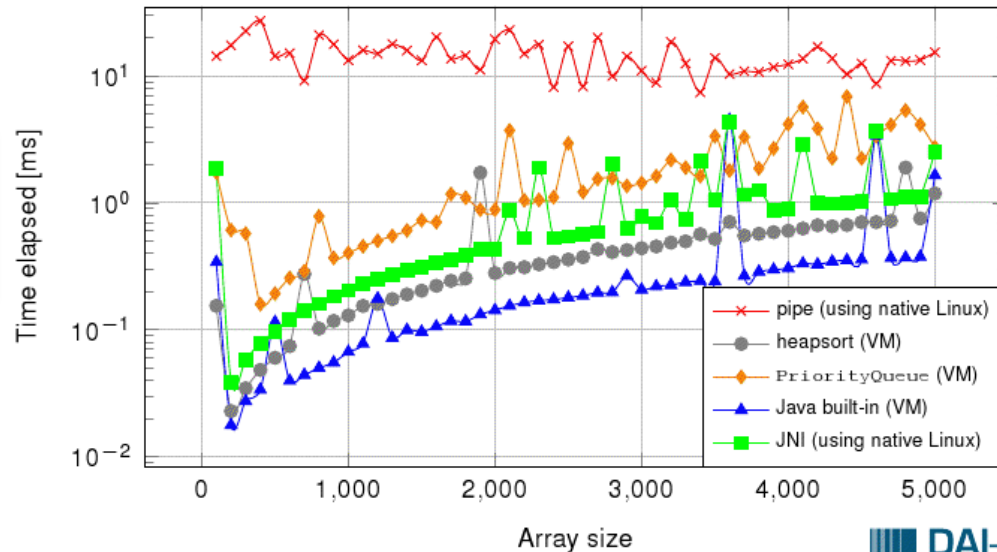
- Java kódból hívhatunk C/C++ kódot, memóriára mi ügyelünk!
- Mit jelent ez? – néhány számpélda

0 és 1000 közötti integer rendezése növekvőbe és csökkenőbe – és mindezt 10x

	Time (ms)
Dalvik	4668
Mono	411
Java SE for Embedded	895

De a natív kód meghívása némi többlet költséggel jár!

Sorting Integers on a Linux PC (for comparison)



◎ OpenGL

- Komoly grafikát igénylő alkalmazásoknál (pl.: játék) ajánlott OpenGL használata a hardveres gyorsítás miatt.
- 2D / 3D egyaránt.

Teljesítményt javító eszközök

◎ Zipalign tool

- A forráskezelő akkor a leghatékonyabb, ha a forrás 4 byte-os egységekhez van igazítva. (32 bit) → Zipalign erre jó!
- ADT 0.9.3-as óta, projekt exportálásánál automatikus: Projekten jobb klikk / Andorid tools / Export Signed Application Package...
- Manuálisan:
tools/zipalign -v 4 source.apk destination.apk

◎ DDMS memórafoglalás figyelése

- DDMS perspektívában lehetőségünk van a memórafoglalásokat követni.

Jöhet a pihenés 😊

