Írjunk többszálú programot Linux alatt! (1. rész)

A Qt könyvtár multithread-támogatásának bemutatása.

A Linuxon immár szabadon használható Qt fejlesztői keretrendszert bizonyára sokan ismerik a világon. Unix (Linux), az összes Windows, illetve Mac OS X operációs rendszerekre történő fejlesztésekhez egyaránt használható, de létezik olyan változata is, amelyik a beágyazott rendszerek fejlesztéséhez lett összeállítva. Fontos tudni azt is, hogy a KDE asztali környezetnek is a Qt csomag az alapja. A Qt-t sokan egy GUI-építéshez használható elemgyűjteményként, könyvtárként ismerik, ami olyan remek eszközökkel van felvértezve, mint a Qt Designer és a Qt Linguist. E tervezőeszközök tudása a Borland Kylixéhoz hasonlít. A Qt azonban sokkal több, mint egy grafikus felületet megvalósító program GUI-motorja.

Mik is azok a szálak?

A többszálú programozást lehetővé tevő környezetekben a szálak időosztásos vagy együttműködő módon futhatnak. Elsőként tisztázzuk, hogy mi is az a szál, illetve mit értünk többszálú program alatt. A hagyományos szekvenciális programok egyik tulajdonsága az, hogy mindig egyetlen aktív végrehajtási ponttal rendelkeznek, ami megmutatja, hogy mi lesz a következő végrehajtandó utasítás. Ezzel szemben egy többszálú program több aktív végrehajtási pontot tartalmazhat, ezek között az ütemező program osztja szét a CPU-erőforrást (általában több CPU is lehet). Ezek szerint a szál (thread) a folyamat (process) legkisebb önállóan ütemezett egysége. A szálak saját veremmel, gépi regiszterkészlettel, fontossággal (priority) és időszelettel rendelkezhetnek, de öröklik az őket befoglaló folyamat memóriacímterét, megnyitott fájljait, jeleit (signal) és általában a közösen használható erőforrásokat. A szálak használatának az a célja, hogy a folyamat párhuzamosan futtatható részei explicit módon is meg legyenek határozva, így a program-végrehajtási környezet rendelkezhet azzal az adattal, hogy mely tevékenységek párhuzamosíthatók. A szálak kezelése történhet mag- vagy felhasználói módban – mindkét felfogásnak megvannak a maga előnyei és hátrányai. A Linux több szálkezelő megoldással is bír, talán a legrégebbi a Posix `pthread` alrendszer. Ebben a cikkben a Qt 3.x által megvalósított, objektumközpontú szálkezelést ismertetjük, amit a Qt GUI-felületek dinamikusabb működtetése mellett az egyszerű textalapú programok készítésénél is használhatunk.

Szálak létrehozása és használata

A szálak létrehozásának megismerése közben írjunk egy olyan programot, ami kipróbálja, hogy a Qt könyvtár az időosztásos módszert alkalmazza-e a szálak ütemezésére. Az elv az, hogy elindítunk három szálat, ezek között semmilyen kapcsolat nincs, viszont mindegyik rendelkezni fog egy-egy helyi `szamlalo` változóval, ami folyamatosan növekszik, amikor éppen az őt tartalmazó szálnál van a vezérlés. A programot hagyjuk néhány másodpercig futni, majd valamilyen billentyű megnyomásával léphetünk ki belőle. A program utolsó lépéseként a `main()` függvény mindhárom szál `szamlalo` nevű helyi változójának értékét kiírja a képernyőre. Amennyiben a három szám hasonló nagyságrendű, úgy a Qt időosztásos ütemezést használ. Lássuk a programot (a CD-mellékleten ez a ***teszt0.cpp***)!

```
1. //
2. // teszt0.cpp
3. //
4. #include <iostream>
5. #include <qthread.h>
6.
7. using std::cout;
8.
9. // A TMyThread osztály felülete
10.class TMyThread : public QThread
11.{
12. bool leallitott; // A szál programozott
                      ➥leállítására
13. long int szamlalo; // helyi számláló
14. public:
15. virtual void run();
16. void leallit(void);
17. long int getSzamlalo() { return szamlalo; }
18.};
19.
20.// Programozott leállítás (Hasonló, mint a
Java 2 ajánlása)
21.void TMyThread::leallit(void)
22.{
23. leallitott = true;
24.}
25.
26.// Ez a szál kódja
27.void TMyThread::run()
28.{
29. leallitott = false;
30. szamlalo = 0;
31.
32. while ( !leallitott )
33. {
34. szamlalo++;
35. }
36.}
37.
38.
39.//--- A program indulási pontja ---
40.int main()
41.{
42. TMyThread t1, t2, t3;
43.
44. t1.start(); t2.start(); t3.start();
45. getchar();
46. t1.leallit(); t2.leallit(); t3.leallit();
47.
48. // A main() függvény bevárja, hogy az
       ➥összes szál lefusson
49. t1.wait(); t2.wait(); t3.wait();
```

```
50.
51. cout << "\nAz első szál számlálója: " <<
t1.getSzamlalo();
52. cout << "\nA második szál számlálója: "
            ➥<< t2.getSzamlalo();
53. cout << "\nA harmadik szál számlálója: "
            ➥<< t3.getSzamlalo() << "\n";
54.
55.
56.}
```

A program egyik futási eredménye a következő:

- Az első szál számlálója: 72 459 958.
- A második szál számlálója: 83 693 735.
- A harmadik szál számlálója: 81 453 880.

Látható, hogy a számlálók értéke azonos nagyságrendű, így a Qt önállóan kezelt időosztásos módban futtatja a szálakat. Nézzük végig a program forráskódját!
Az 5. sor mutatja, hogy a Qt szál (thread) használatához mindig a *qthread.h* beépített (include) fájlt kell beilleszteni. A 10. sorban egy `TMyThread` osztályt kezdünk létrehozni, ami nyilvános utódosztálya a `QThread` osztálynak. A `QThread` osztály valósítja meg a szál fogalmát a Qt könyvtárban. Osztályunknak két saját tagváltozója van. Az egyik a már ismertetett `szamlalo`. A másikat (melynek neve: `leallitott`) a programozott szálleállítási megoldás használja. Érdemes megjegyezni, hogy a Java 2 is ezt a fajta `iarendesenl`, programozott szálleállítási megoldást ajánlja. A 17. sor `getSzamlalo()` tagfüggvénye a `kiiráskor` lesz hasznos, hiszen a mi számláló adattagunk `privat`, így csak egy ilyet elérő függvénnyel kérhetjük le az értékét. A 20–24. sor a szálleállító eljárást valósítja meg. Ebben a kódban nem volt fontos ennek a változónak a kizárólagos használata. A 27. sortól kezdődik a `public` elérésűre meghatározott `run()` tagfüggvény, ami a végrehajtási szál kódját tartalmazza, és feladata lényegében csak a számláló folyamatos növeléséből áll. A 40. sorban kezdődő főprogram `t1`, `t2`, `t3` néven három szálat hoz létre, majd a 44. sorban a `QThread` osztálytól örökölt `start()` tagfüggvénnyel elindítja őket. Ezután a főprogram a `getchar()` hívásnál elakad, és arra vár, hogy valaki leüssön egy billentyűt. Eközben a `t1`, `t2`, `t3` szál dolgozik. Egy billentyű megnyomása után a 46. sorban leállítjuk a szálainkat. Miért van szükség a 49. sor `wait()` hívásaira? Mi az a `wait()` tagfüggvény? A `QThread` osztály `wait()` tagfüggvénye a szálak összekapcsolását (`JOIN`) valósítja meg. Legyen `sz1` és `sz2` két szálváltozó. Ekkor az `sz1` `run()` kódjából hívott `sz2.wait()`-nek az a hatása, hogy az `sz1` szál addig blokkolódik, amíg az `sz2` véget nem ér, mely után az `sz1` folytatja a munkáját. Az `sz1` szál tehát így tudja bevárni `sz2` befejeződését. A mi programunkban is ez a cél. A `main()` függvénynek mindegyik szál befejeződését be kell várnia, és ő maga csak ezután fejeződhet be, de előbb az 51–53. sorban kiírja a számlálók értékét. A Qt GUI-alkalmazásokban a `wait()` használata nem szükséges, hiszen úgyis csak egy kiléptető eseménnyel fejezhetjük be a programunkat.
Befejezésül nézzük meg, hogyan tudjuk lefordítani a *teszt0.cpp* programot. A hozzá tartozó `Makefile` az 52. CD Magazin/Qt könyvtárában található, a HOGYAN-fájl a honlapunkon olvasható.

A szálak ütemezésének hangolása

Láttuk, hogy a Qt-szálak önműködően időosztásosak, sokszor jó lenne azonban az időszeletek hosszát saját kezűleg is befolyásolni. A *teszt1.cpp* programot az előzőek alapján már valószínűleg mindenki érti. Létrehozunk két szálat a főprogramban, majd mindkettő folyamatosan kiírja a szálazonosítóját, illetve azt, hogy hol tart a 3000-ig történő számolásban.

```
1. //
2. // teszt1.cpp
3. //
4. #include <iostream>
5. #include <string>
6. #include <qthread.h>
7.
8. using std::cout;
9. using std::string;
10.
11.class TMyThread : public QThread
12.{
13. string szalnev;
14.public:
15. TMyThread(string szalnev) { this->szalnev
            ➥= szalnev; }
16. virtual void run();
17.};
18.
19.void TMyThread::run()
20.{
21. for(int i=0; i<3000; i++ )
22. {
23. cout << "\n " << szalnev << " = " << i;
24. //if ( i % 100 == 0 ) QThread::usleep(100);
25. }
26.}
27.
28.//
29.// Indulás...
30.//
31.int main()
32.{
33. TMyThread t1("T1 szál");
34. TMyThread t2("T2 szál");
35. t1.start(); t2.start();
36. t1.wait(); t2.wait();
37. cout << "\n";
38.}
```

A program lehetővé teszi a szálak vizsgálatát, azaz egy fájlba átirányítva az eredményt megtekinthető, hogyan alakult a `t1` és `t2` szál aktív futási ideje. Ennek finomítási lehetősége az, ha a 24. sorból kivesszük a megjegyzést, és úgy futtatjuk a `teszt1.exe` programot. Az `i` ciklusváltozó minden 100. értékénél az éppen aktív szál önként lemond a futásáról, ennek eredményeképpen a másik folytatni tudja a futását. A futási eredmény mutatja, hogy a szálak időbeni aktivitása finomodott (becsempésztünk egy kis együttműködő jelleget az ütemezésbe), fontos azonban kiemelni, hogy ezzel a lehetőséggel a végsőkig visszaélve a hatékonyságot is ronthatjuk, hiszen növekszik a szálak közötti kapcsolgatás felügyeleti költsége.
Sorozatunk következő részében a QThread osztály részletes áttekintésével folytatjuk, továbbá szót ejtünk a közösen használható erőforrások védelméről is.

Nyíri Imre (inyiri@mol.hu)