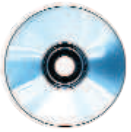
# Írjunk többszálú programot Linux alatt! (2. rész)



Sorozatunkat a QThread és a többi szálkezelő osztály áttekintésével folytatjuk.

Az elmúlt hónapban a többszálú programok készítésének és a Qt környezet által nyújtott lehetőségek alapjaival foglalkoztunk. Most folytatjuk a Qt többszálú programok készítését támogató moduljának bemutatását, célunk a részletek pontosabb megismertetése.
Minden szálosztálynak a QThread az őse, ezért nézzük meg egy kicsit részletesebben is!

### Public eljárások

- `bool wait(unsigned long time = ULONG_MAX)` – a szálak összekapcsolását (JOIN) teszi lehetővé.
  Az előző részben (Linuxvilág, októberi szám, 51. oldal) már részletesebben ismertettük a működését. Amikor a `time` valamilyen tényleges ezredmásodperc érték, akkor ez az eljárás csak a megadott ideig várakozik a meghívott szálra.
- A `wait()` sikertelen összeláncolás esetén `false` értéket ad vissza.
- A `void start()` – elindítja a szál párhuzamosan futó kódját (lásd `run()` tagfüggvény).
- A `bool running() const` – a szál fut, illetve nem fut állapotát kérdezhetjük le.

### Statikus public eljárások

- `void postEvent (QObject *receiver, QEvent *event)`
  A Qt legfontosabb része természetesen a grafikus GUI-készítési lehetőség. A jel/foglalat (signal/slot) szerkezet nagyon érdekes tervezési minta, de a Qt lehetőséget ad az aszinkron eseménykezelésre is. Ez az eljárás egy fogadó grafikus elem (widget) számára tud egy eseményt szálbiztosan elküldeni.
- `void exit()` – leállítja az ezt a tagfüggvényt hívó szál futását, valamint felébreszti az összes olyan szálat, ami eddig ennek a szálnak a lefutására várt.

### Protected eljárások

- `virtual void run()`
  A Java nyelvhez hasonlóan itt is a `run()` tagfüggvény képviseli a szál kódját, amit nekünk kell mindig újra megvalósítani.

### Statikus protected eljárások

- `void sleep(unsigned long masodperc)`
- `void msleep( unsigned long ezred_masodperc)`
- `void usleep( unsigned long millomod_masodperc)`

Ezek az eljárások a megadott ideig felfüggesztik a szálak futását (a szálakat elaltatják), ez egyben a hívó szál futási jogáról való lemondását is jelenti. Egy nagyon rövid várakoztatáshívás az együttműködő szálfuttatási rendszerek klasszikus `yield()` (azaz lemondok a futási jogomról) függvényének felel meg.

## A közösen használható erőforrások védelme

Az eddigi két program nem használt minden szál által látható, közös (globális) erőforrásokat, hiszen csak helyi változóink voltak. A párhuzamos programozás megvalósításához a szál fogalma nem elegendő, hiszen biztosítani kell a közös erőforrások megfelelő sorrendben (összehangolás) való használatát. Gondoljunk a közúti közlekedés példájára! Képzeljük el, hogy az egyes szálak a közlekedési eszközök vagy a gyalogosok. Ekkor a közösen használt erőforrások az útkereszteződések és a zebra. A KRESZ szabályai és a jelzőlámpák valósítják meg ebben a rendszerben az összehangolást. A Qt egyik összehangoló eszközét a `QMutex` osztály képviseli.
Az osztály célja a közösen használható objektumok védelme, illetve a forrásprogramok kényes szakaszainak kijelölése.
A `QMutex` kétállapotú sorompó (nyitott vagy zárt) módjára véd. Amennyiben egy `mutex` zárt, úgy az általa védett kódrészbe a többi szál nem léphet be, azaz várakozniuk kell.
Lássuk a `QMutex` osztályt!

- `QMutex(bool recursive = FALSE)`
  Létrehoz (constructor) egy `mutex` objektumot. Lehetőség nyílik a `recursive` beállítás használatára, ugyanakkor a `TRUE` érték megadása szükséges, amely esetben a `mutex`-re kiadott `lock()` hívások száma nyilvántartódik, és pontosan ugyanennyi `unlock()` meghívására lesz szükség a `mutex` felnyitásához.
- `void lock()` – lezárja a `mutex` objektumot. Amennyiben a `mutex` objektumot egy másik szál már lezárta, úgy ez a tagfüggvény blokkolja a hívó szálat, ami csak akkor futhat tovább, ha a `mutex` ismét nyitott állapotba kerül és utána sikerül a zárolás.
- `void unlock()` – felnyitja a `mutex` objektumot.
- `bool locked()` – visszaadja, hogy egy `mutex` objektum most éppen zárt vagy nyitott állapotú-e.
- `bool tryLock()` – a célja hasonló a `lock()` tagfüggvényéhez, de sikertelenség esetén a hívó szál nem blokkolódik, ezért programunknak lehetősége van egy másik logikai ágon továbbfutnia.

A Qt csomag egy másik összehangolási fogalmat is ad: a `QSemaphore` osztályt. Egy `QMutex` objektum kétállapotú jelzőnek (semaphor) feleltethető meg. Általános értelemben viszont jól jön egy olyan eszköz is, ami nemcsak egyetlen szálat enged be a védett területre, hanem annyit, amennyit a jelző létrehozójában (constructor) megadtunk. Miért jó ez nekünk? Képzeljük el, hogy egy vendéglőben *N* darab szék van, és a vacsoravendégek jelképezik az egyes szálakat. Ekkor úgy kell szervezni a vendéglő működését, hogy egyszerre legfeljebb csak *N* darab vendég mehessen be az ajtón, a többi kint várakozzon. Ha néhány vendég elhagyja a vendéglőt, akkor a várakozók közül már kiválaszthatók a következő szerencsés és éhes vendégek. A számítástechnikához kicsit közelebb álló példa lehet a TCP/IP-kapcsolatkészlet esete. Tegyük fel, hogy van 10 előre elkészített `Connection` objektumunk.

Ekkor egy jelző vezérelheti a szálak kapcsolatkérési igényeit. A `QSemaphore` osztály rendelkezik az atomi módon végrehajtható növelő, csökkentő műveletekkel, amiket részletesen a Qt leírásából tanulmányozhatunk.

## Az író–olvasó feladat megoldása

Gyakori helyzet, hogy létezik egy osztottan használható erőforrásunk (egy adatbázis, egy memóriaváltozó), amit néhány írófolyamat időnként frissít (általában egy lassú író művelettel), illetve sok olvasófolyamat olvas (ez általában egyetlen gyors művelet). Az nyilvánvaló, hogy az erőforrás írása során nincs értelme olvasni, hiszen ekkor átmeneti állapotban lehet a kiolvasandó adat. Ugyanakkor több olvasószál természetesen párhuzamosan olvashat. Ilyen jellegű gond merülhet fel például egy internetes tőzsdei index szolgáltató alkalmazás esetén. Példaképpen legyen a feladatunk az, hogy egy integer változót (neve: `TheValue`) írunk és olvasunk párhuzamosan. Egy lehetséges megoldást mutat be az *1. listán* található forrásprogram.
A 15–20. sorokban létrehozott `kiir()` függvény biztosítja azt, hogy a képernyőre írás megszakíthatatlan (atomi) művelet legyen. Ezt a kényes szakaszt a `kiiroMutex` védi. A Java-programozók ilyen kóddal oldották volna meg a feladatot:

```
synchronized ( kiirastVedo
_ Objektum ) {
kiiras...
}
```

A `countOfReader` változó tárolja a pillanatnyi olvasók számát. A 31. sorban látható létrehozónak egy egész értékű szálazonosítót tudunk átadni. A `TReaderWriter` osztály két lényegi tagfüggvénye a `reader()` és a `writer()`. Ezek működését a `reader_mutex` és a `writer_mutex` objektum hangolja össze. A `writer()` tagfüggvényt teljes mértékben védi a `writer_mutex`. A `TheValue` értékét azért változtattuk meg öt lépésben, hogy életszerűbb legyen, ugyanis az írási folyamat általában több lépésből áll, és emiatt a kényes szakasz bevezetése nélkül könnyen megszakítható lenne. Ekkor pedig a `TheValue` változó öttel nem osztható állapota jöhetne létre. A 44. sorban kezdődő `reader()` működése azon az ötleten alapul, hogy az olvasó szálakat számoljuk, és amennyiben a számuk nulla, úgy a `writer_mutex`-et is lezárt állapotba hozzuk. Lehetséges, hogy a `writer_mutex.lock()` nem sikerül, de ez pont jó, hiszen ekkor egy írófolyamat dolgozik és nincs is értelme addig olvasni. Ekkor itt várunk, amíg az író be nem fejezi a munkáját. Az is lehetséges, hogy a `writer_mutex.lock()` sikerült. Ezután természetesen az írók fognak várni addig, amíg az olvasók száma nulla nem lesz, mert a 60–63. sorok tanúsága szerint csak ekkor történik meg ismét a `writer_mutex` felnyitása.
A 84–101. sorokban megvalósított `run()` tagfüggvény az eddigiek alapján már biztosan érthető, a feladata az, hogy egy szál `reader()` és `writer()` műveletet hajtson végre.
A `main()` függvényben egy tízelemű, `TReaderWriter` mutatókból (pointer) álló tömböt hozunk létre, majd elindítjuk őket egy-egy új `TReaderWriter` objektummal, majd elindítjuk a tíz végrehajtási szálat. A 121. sorban egy tetszőleges billentyűleütésre várunk, ennek hatására szabályosan befejeződik a programunk.

## A szálak elaltatása és felébresztése: feltételes várakoztatás

A párhuzamosan futó szálak különféle feladatokat hajthatnak végre, de általában kijelenthető az is, hogy a teendőknek valamilyen előfeltételük szokott lenni. Amikor egy szál futási feltétele nem teljesül, elaltathatja magát. Egy másik szál figyeli azt, hogy a feltétel most már teljesül-e, ha igen, az alvó szálat felébreszti. Érezhető, hogy ebben a működésben a `QMutex` által kialakított szál összehangolásának is jelentős szerep jut, hiszen arra is vigyázni kell, hogy a felébredt szál futási feltételei ne váljanak ismét hamissá. A Qt a `QWaitCondition` osztály segítségével teszi elérhetővé a feltételes várakoztatást, ezért nézzük meg, mit nyújt számunkra ez az osztály!

```
bool wait(unsigned long time = ULONG_MAX)
```

Elaltatja a szálat, ami az alapértelmezett `ULONG_MAX` érték esetén addig alszik, amíg valaki fel nem ébreszti. A `wait()` tagfüggvényben valós időt is megadhatunk (ezredmásodpercben), amely time-outként működik, azaz ha a megadott idő letelik, a szál magától felébred.

```
bool wait(QMutex *mutex, unsigned long
➥time = ULONG_MAX)
```

Hasonlóan működik, mint az előző `wait()` tagfüggvény, de elalvás előtt kinyitja a zárolt `mutex`-et.

```
void wakeOne()
void wakeAll()
```

Legyen `qwc` egy `QWaitCondition` típusú változó (objektum). Ekkor a `qwc.wakeOne()` felébreszt egy olyan szálat, amit előzőleg egy `qwc.wait()` hívás altatott el. A `qwc.wakeAll()` az összes ilyen szálat felébreszti. A feltételes elalvás–felébredés jobb megértése érdekében nézzünk meg egy olyan példát, ahol három dolgozó szál csinál valami hasznosat (példánkban csak egy üzenetet jelenít meg a képernyőn, ezt most nem hangoljuk össze egy `kiir()` tagfüggvénnyel). A `main()` függvény szála (azaz a fő szál) egy gombnyomáseseményt figyel, ennek bekövetkeztekor felébreszti az erre a feltételre várakozó három dolgozószálat (2. lista, 53. CD Magazin/Qt könyvtára).
A 12. sorban meghatározott gombnyomásváltozóval kezeljük a feltételes elalvást, illetve felébredést. Itt jó programozási szokás, ha az ilyen változónak a feltétel vagy az esemény nevét adjuk meg. A 43. sorban látható, hogy a `TDolgozo::run()` szálkód végrehajtása lényegében azzal kezdődik, hogy egy gombnyomás eseményre várva elaltatja magát. A `main()` szál ciklusfeltételében egy gombnyomásra várakozás van.
A `while` ciklus mindaddig ismétlődni fog, amíg nem a 'v'-t (v=vége) nyomtuk le. Egy nem v billentyű lenyomása után a főprogramban (82. sor) a `gombnyomas.wakeAll()` felébreszti mindhárom dolgozószálat, hiszen mindegyik a `gombnyomas.wait()` hívásnál aludt el. A dolgozó szálak (d1, d2, d3) az idő haladtával ismét el fognak aludni, ha már mindegyik alszik, akkor az `ennyiSzalDolgozik` változó nulla lesz, így a fő szál belső `while` ciklusából újra és újra kiléphetünk, és az összes dolgozót felébreszthetjük.

## A termelő-, illetve fogyasztófeladat megoldása

A párhuzamos programozás másik klasszikus kérdése (nevezhetnénk desing patternnek, azaz tervezési mintának is) a termelő-, illetve fogyasztófeladat. Van egy raktár, ahová a termelőszál bizonyos típusú objektumokat helyez el, míg ezeket a fogyasztószál kiveszi innen, hogy valahol majd felhasználja őket. A raktár véges méretű, azaz legfeljebb ***K*** darab objektum befogadására alkalmas. Az összehangolásnak két feltétele van:

```
1. //
2. // Író/olvasó feladat
3. //
4.
5. #include <stdio.h>
6. #include <stdlib.h>
7. #include <iostream>
8. #include <string>
9. #include <qthread.h>
10.
11.using std::cout;
12.
13.static QMutex kiiroMutex;
14.
15.void kiir(char *sz, int th_szam,
            ➥int ertek)
16.{
17. kiiroMutex.lock();
18. cout << "\n" << sz << " (szál=" <<
    ➥th_szam << ", érték=" << ertek << ")";
19. kiiroMutex.unlock();
20.}
21.
22.class TReaderWriter : public QThread
23.{
24. static QMutex reader_mutex;
25. static QMutex writer_mutex;
26. static int countOfReader; // Az
olvasószálak pillanatnyi száma
27. static int TheValue; // Az olvasandó
    ➥érték tárolója
28. int th_ID; // A szál azonosítója
29. bool leallitott;
30.public:
31. TReaderWriter( int pID )
    ➥{ th_ID = pID; }
32. void reader();
33. void writer();
34. void leallit();
35. virtual void run();
36.};
37.
38.
39.QMutex TReaderWriter::reader_mutex;
40.QMutex TReaderWriter::writer_mutex;
41.int TReaderWriter::countOfReader = 0;
42.int TReaderWriter::TheValue = 0;
43.
44.void TReaderWriter::reader()
45.{
46. reader_mutex.lock();
47. if ( countOfReader == 0 )
48. {
49. writer_mutex.lock();
50. }
51. countOfReader++;
52. reader_mutex.unlock();
53.
54. kiir("Egy ovasás történt:", th_ID,
        ➥TheValue);
55.
56. reader_mutex.lock();
57. countOfReader--;
58. reader_mutex.unlock();
59.
60. if ( countOfReader == 0 )
61. {
62. writer_mutex.unlock();
63. }
64.}
65.
66.void TReaderWriter::writer()
67.{
68. writer_mutex.lock();
69. TheValue++;
70. TheValue++;
71. TheValue++;
72. TheValue++;
73. TheValue++;
74. kiir("Az er _ forrás értéke növelve:",
        ➥-1, TheValue);
75. writer_mutex.unlock();
76.}
77.
78.void TReaderWriter::leallit(void)
79.{
80. leallitott = true;
81.}
82.
83.
84.void TReaderWriter::run()
85.{
86. leallitott = false;
```

*folytatás a következő oldalon*

- A termelőnek aludnia kell addig, amíg a raktár tele van.
- A fogyasztónak aludnia kell addig, amíg a raktár teljesen üres.

Példaprogramunkban a raktárat egy `BUFFER_MERET` nagyságú, egészekből álló tömb testesíti meg. A termelt objektumok egész számok. A program ismertetése előtt kiemelném két érdekességét. Az első az, hogy a raktár egy ciklikus FIFO segítségével van megvalósítva. Ebben az adatszerkezetben a betett elemek számát is figyelemmel kell kísérnünk, különben az `ÜRES` és a `MEGTELT` eseteket nem tudnánk megkülönböztetni. A másik érdekesség, hogy a program az összehangolt adatkezeléshez (raktárkezeléshez) a monitorelgondolást használja. Ez a tervezési minta egységbe zárja a védett erőforrást (adatot) és az ezt kezelő műveleteket (függvényeket). A szálak az erőforrást csak a monitor által biztosított műveleteken keresztül kezelhetik. (3. lista, 53. CD Magazin/Qt könyvtára.)

A forráskód elején lévő `BUFFER_MERET` állandó (constans) a raktár nagyságát, a `TERMELENDO` pedig a legyártandó egész számok számát határozza meg. A program 24–29. sorában megvalósítottuk az összehangolt képernyőre írást, a szálak csak ennek a használatával jeleníthetik meg a szöveges adatokat. A 30. sortól egy segédfüggvényt találunk, ami a *libc* véletlenszám-generáló lehetőségét használja egy új (1 és 5000 közötti) egész érték létrehozásához.

*folytatás az előző oldalról*

```
87.
88. reader_mutex.lock();
89. kiir("Elindult egy új szál:", th_ID, -1);
90. reader_mutex.unlock();
91.
92. while ( !leallitott )
93. {
94. for(int i = 0; i < 20; i++)
95. {
96. reader();
97. }
98. writer();
99. }
100. kiir("Leált a következ _ szál:",
          ➥th_ID, -1);
101.}
102.
103.
104.//
105.// Indulás...
106.//
107.int main()
108.{
109. TReaderWriter *rw[10];
110.
111. for (int i=0; i<10; i++)
112. {
113. rw[i] = new TReaderWriter(i);
114. }
115.
116. for (int i=0; i<10; i++)
117. {
118. rw[i]->start();
119. }
120.
121. getchar();
122.
123. for (int i=0; i<10; i++)
124. {
125. rw[i]->leallit();
126. }
127.
128. for (int i=0; i<10; i++)
129. {
130. rw[i]->wait();
131. }
132.
133. for (int i=0; i<10; i++)
134. {
135. delete rw[i];
136. }
137.
138. return 0;
139.}
```

A 39. sortól található meg a `TMonitor` osztály felülete és megvalósítása.

```
39. class TMonitor
40. {
41. int buffer[BUFFER_MERET]; // Ide termelünk
    és innen fogyasztunk
42. int elso, utolso, betettElemekSzama;
43. public:
44. TMonitor();
45. void betesz(int e);
46. int kivesz(void);
47. };
```

Ez a típus lesz az a monitor, amelyik átmenetitárazza a termelő termelését, illetve szolgáltatja az objektumokat (most csak egy-egy egész szám) a fogyasztónak. Ezt a két műveletet a `betesz()` és a `kivesz()` tagfüggvények valósítják meg. Vegyük észre, hogy a `betesz()` a fogyasztószálat, a `kivesz()` a termelőszálat ébreszti, bár lehet, hogy erre nem lenne szükség, de nem lesz belőle baj. Az az igazság, hogy az így, feltétel nélkül kiadott `wakeOne()`, azaz felébresztés valószínűleg nem nagyobb költségű, mintha megvizsgálnánk, hogy a másik szál alszik-e. A 98–102. sorig a termelőosztály, míg a 107–111. sorig a fogyasztóosztály felülete van meghatározva. Látható, hogy mindkettőben csak a `run()` tagfüggvény került újraírásra. Mindkét `run()` tagfüggvény annyira egyszerű, hogy nem is érdemes hozzájuk magyarázó szöveget írni. A fő szálat megvalósító `main()` függvény pedig még ennél is egyszerűbb. Röviden ennyit szerettem volna elmesélni a Qt többszálúság (multithread) lehetőségeiről. A példaprogramok tanulmányozásához sok sikert kívánok.

Az összes példa egyszerre történő lefordításához mellékeltem egy `Makefile`-t. Nézze meg mindenki, hogy nála is a */usr/lib/qt3* könyvtár tartalmazza-e a Qt csomagot! Ha nem, akkor a `Makefile`-ban a jó elérési útra kell beállítani a `-I` és `-L` kapcsolókat. A `make` parancs kapcsolók nélküli futtatásával az összes példa futtatható változata előállítható.
***Mindenkinek kellemes programozást kívánok!***

***A cikkhez tartozó listák megtalálhatóak az 53. CD Magazin/Qt könyvtárában.***

*Nyíri Imre* (inyiri@mol.hu)
Jelenleg a MOL Rt.-nél dolgozik. Informatikai vállalkozásában az Internet, a Linux, valamint a Java-programozás gyakorlati hasznosításával foglalkozik. Örök szerelme a C++ maradt.

## KAPCSOLÓDÓ CÍMEK

1. A Qt-csomaghoz adott leírás
   ➲ http://doc.trolltech.com
   ➲ http://www.trolltech.com
2. Dr. Kacsuk Péter–Ferenczi Szabolcs
   Párhuzamos és konkurens programozás soktranszputeres rendszereken
   Budapest, 1993 BME jegyzet (ISBN 963 431 774 X)