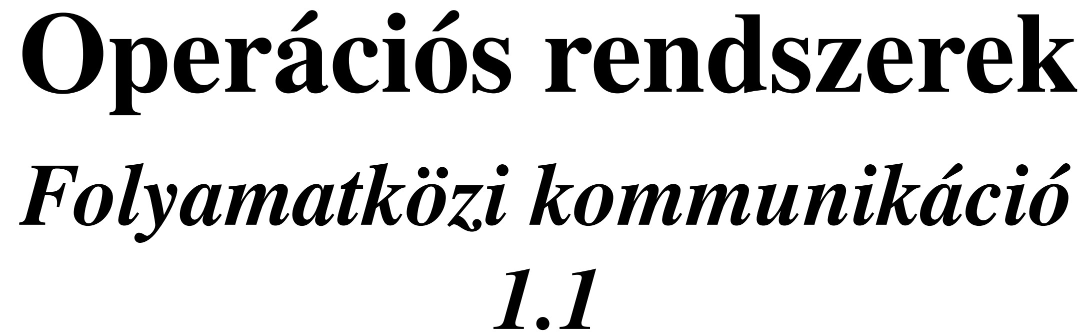


Pere László (pipas@linux.pte.hu)

Az IPC (*inter process communication*) kérdésköre:

- Hogyan tud információt átadni az egyik folyamat a másiknak?

Az IPC

Az IPC (*inter process communication*) kérdésköre:

- Hogyan tud információt átadni az egyik folyamat a másiknak?
- Hogyan bizonyosodhatunk meg arról, hogy két folyamat nem áll egymás útjába?

Az IPC

Az IPC (*inter process communication*) kérdésköre:

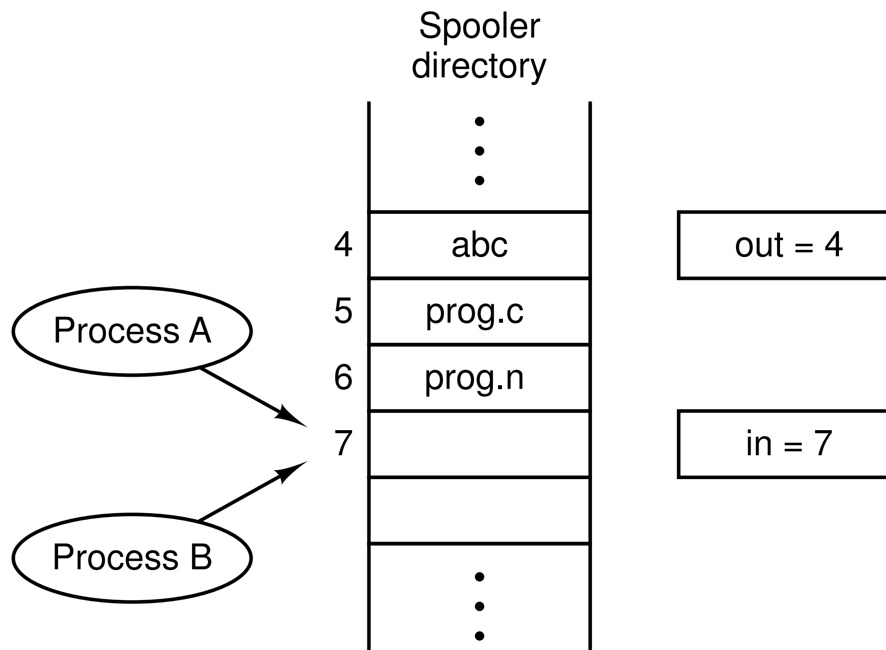
- Hogyan tud információt átadni az egyik folyamat a másiknak?
- Hogyan bizonyosodhatunk meg arról, hogy két folyamat nem áll egymás útjába?
- Milyen sorrendet kell tartanunk, ha függőségek léteznek?

IPC és multi-thread

Az itt tárgyalt módszerek multi-thread rendszerek esetében a szálak kezelésében is segítenek, hiszen a szálak közti kapcsolattartás hasonló problémákat vet fel.

A szálakat éppen ezért sokszor könnyű folyamatoknak is nevezzük (*lightweight processes*).

Versenyhelyzet



1. ábra. A versenyhelyzet és következménye

Versenyhelyzet

Két vagy több folyamat egy időben használ egy közös erőforrást, a futás végeredménye attól függő, hogy pontosan mikor, melyik folyamat futott.

A folyamatok futása így nem jósolható, esetlegessé válik.

Ezt a szituációt versenyhelyzetnek (*race condition*) nevezzük.

Vegyük észre, hogy a versenyhelyzet nem azt jelenti, hogy két folyamat verseng az erőforrás használatáért, a baj tehát megtörtént.

Kölcsönös kizárás

A versenyhelyzet megelőzésének érdekében biztosítanunk kell, hogy a folyamatok ne használják egyidőben a közös erőforrásokat.

A szokásos módszer az, hogy a folyamatok időről időre kizárólagos jogokat szereznek az erőforrás használatához.

Ez a kölcsönös kizárás (*mutual exclusion*) módszere.

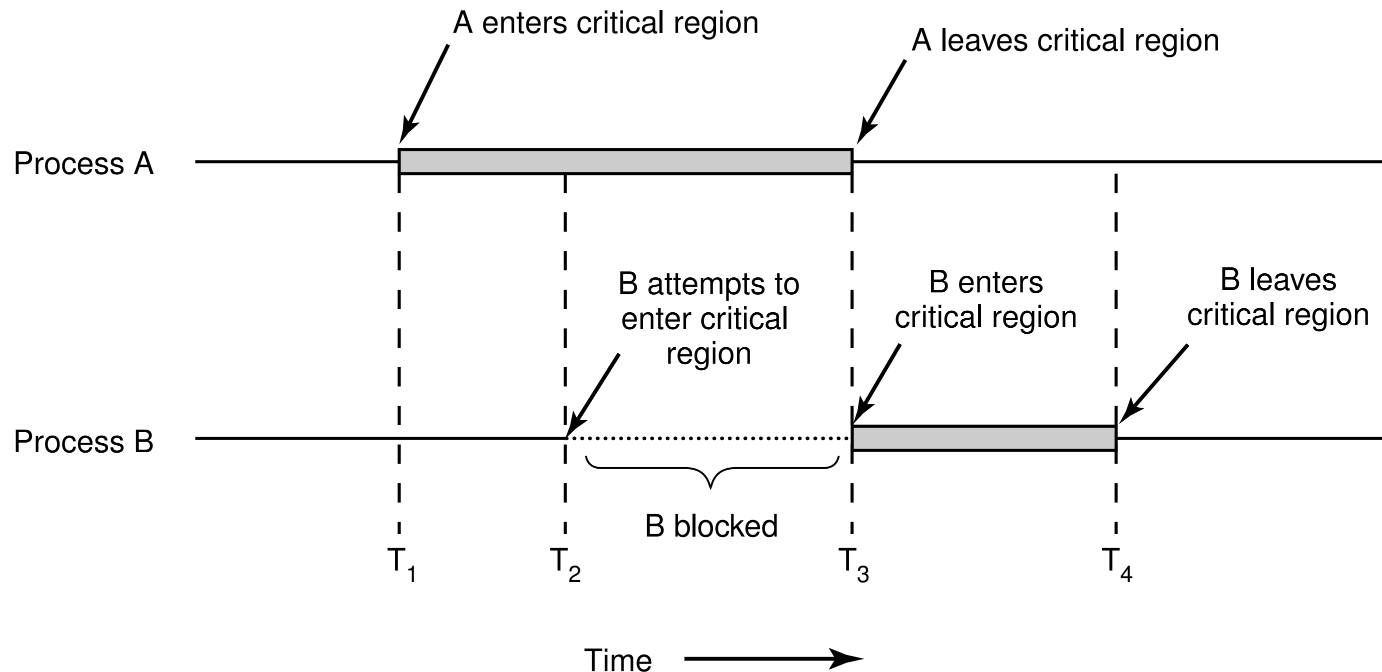
Vegyük észre, hogy a kölcsönösség nem jelentheti az egyidejűséget, nem egy időpontban zárják ki egymást, mert az katasztrófához fevetne!

Kritikus szakaszok

A kommunikáló folyamatokban vannak olyan szakaszok, amelyekben a megosztott erőforrásokat kezelik, olyan műveleteket végeznek, amelyek versenyhelyzethez vezetnek. Ezeket a szakaszokat kritikus szakaszoknak nevezzük.

Ha elkerülhetjük, hogy egyszerre két folyamat legyen kritikus szakaszban, a versenyhelyzet elkerülhető.

Kölcsönös kizárás



2. ábra. A kölcsönös kizárás megvalósítása kritikus szakaszok kezelésével

A helyes kapcs. feltételei

A tökéletes kapcsolattartáshoz a következő feltételeknek kell teljesülniük:

- Egyszerre legfeljebb egy folyamat van kritikus szakaszban.

A helyes kapcs. feltételei

A tökéletes kapcsolattartáshoz a következő feltételeknek kell teljesülniük:

- Egyszerre legfeljebb egy folyamat van kritikus szakaszban.
- Nincs beépített előfeltételezés az időzítés és a CPU-k számára nézve.

A helyes kapcs. feltételei

A tökéletes kapcsolattartáshoz a következő feltételeknek kell teljesülniük:

- Egyszerre legfeljebb egy folyamat van kritikus szakaszban.
- Nincs beépített előfeltételezés az időzítés és a CPU-k számára nézve.
- A folyamat, amely nincsen kritikus szakaszban, nem blokkolhatja a többi folyamatot.

A helyes kapcs. feltételei

A tökéletes kapcsolattartáshoz a következő feltételeknek kell teljesülniük:

- Egyszerre legfeljebb egy folyamat van kritikus szakaszban.
- Nincs beépített előfeltételezés az időzítés és a CPU-k számára nézve.
- A folyamat, amely nincsen kritikus szakaszban, nem blokkolhatja a többi folyamatot.
- Egyetlen folyamatnak sem kell végtelen hosszú ideig várnia, hogy beléphessen a kritikus szakaszba.

A megszakítás tiltása

A kölcsönös kizárás biztosítására használható a megszakítás tiltása, de ez nem szerencsés, mert:

- Ha a felhasználói folyamat tiltja a megszakítást, esetleg nem kapcsolja vissza.

Mindazonáltal a megszakítás tiltása az OS magjában használható megoldás.

A megszakítás tiltása

A kölcsönös kizárás biztosítására használható a megszakítás tiltása, de ez nem szerencsés, mert:

- Ha a felhasználói folyamat tiltja a megszakítást, esetleg nem kapcsolja vissza.
- Több processzor esetén nem hatásos a védelem.

Mindazonáltal a megszakítás tiltása az OS magjában használható megoldás.

Legyen egy változó, amely jelzi, hogy hány folyamat van kritikus szakaszban. A folyamat a kritikus szakasz előtt ellenőriz, csak akkor lép be a kritikus szakaszba, ha a változó 0, de előtte beállítja 1-re.

A zárolás

Legyen egy változó, amely jelzi, hogy hány folyamat van kritikus szakaszban. A folyamat a kritikus szakasz előtt ellenőriz, csak akkor lép be a kritikus szakaszba, ha a változó 0, de előtte beállítja 1-re.

Versenyhelyzet van a változó olvasása és az írása között. Ez az új kritikus szakasz, a változó pedig a közösen használt erőforrás.

A szigorú váltás

```
1  while( TRUE ){                while(TRUE){
2    while(ford != 0);             while(ford != 1);
3    kritikus();                   kritikus();
4    ford = 1;                       ford = 0;
5    egyeb();                      egyeb();
6  }                             }
```

A két folyamat közül mindig csak az egyik lehet a kritikus szakaszban, a versenyhelyzet teljesen kizárt.

A szigorú váltás hátrányai

A szigorú váltás a legtöbb esetben nem használható a következő okok miatt:

- Ha az egyik folyamat sokkal lassab, mint a másik, nem használható.

A szigorú váltás hátrányai

A szigorú váltás a legtöbb esetben nem használható a következő okok miatt:

- Ha az egyik folyamat sokkal lassabb, mint a másik, nem használható.
- Két művelet nem végezhető egymás után.

A szigorú váltás hátrányai

A szigorú váltás a legtöbb esetben nem használható a következő okok miatt:

- Ha az egyik folyamat sokkal lassab, mint a másik, nem használható.
- Két művelet nem végezhető egymás után.
- Sérti a 3. feltételt, a folyamat akkor is akadályozhat, ha nincs kritikus szakaszban.

Peterson megoldása (1981)

```
1  void belep(int folyamat){
2  int masik = 1 - folyamat;
3    keres[folyamat] = TRUE;
4    ford = folyamat;
5    while(ford == folyamat &&
6      keres[masik] == TRUE );
7  }
8
9  void kilep(int folyamat){
10   keres[folyamat] = FALSE;
11 }
```

TSL (test and set lock)

A TSL gépi kódú utasítás beolvas egy memóriacellát a regiszterbe és beleír egy 0-tól különböző értéket. A TSL atomos (multiprocesszoros környezetben is).

```
1  enter:                    leave:
2    TSL RX, Lock              MOV Lock, 0
3    CMP RX, 0                 RET
4    JNE enter
5    RET
```

Elfoglalt várakozás

A megoldásban megfigyelhető az elfoglalt várakozás (*busy waiting*), ami azt jelenti, hogy a folyamat az erőforrásra való várakozás közben utasításokat hajt végre.

Az elfoglalt várakozás csak különleges esetekben elfogadható, mert:

- Pazarolja az erőforrást, akkor is használja a processzort, amikor várakozik.

Elfoglalt várakozás

A megoldásban megfigyelhető az elfoglalt várakozás (*busy waiting*), ami azt jelenti, hogy a folyamat az erőforrásra való várakozás közben utasításokat hajt végre.

Az elfoglalt várakozás csak különleges esetekben elfogadható, mert:

- Pazarolja az erőforrást, akkor is használja a processzort, amikor várakozik.
- Prioritás inverzióhoz vezet (a magasabb prioritású folyamat nem engedi a társának, hogy beengedje a kritikus szakaszba).

Alvás ébresztés

A legegyszerűbb módszer a busy waiting elkerülésére az alvás-ébresztés (sleep and wakeup) eljárás.

A folyamat, amelyik nem tud belépni a kritikus szakaszba, a blokkolását kéri abban a reményben, hogy a másik folyamat felébreszti.

Az alvás-ébresztés módszer tahát a folyamatok állapotait és az operációs rendszer szolgáltatásait használja a kölcsönös kizárás megvalósítására.

Termelő-fogyasztó probléma

A számítástechnikában nagyon sokszor előforduló, tipikus IPC feladat a termelő-fogyasztó (*producer-consumer problem*):

- Ha a termelő azt látja, hogy az átmeneti tároló tele, aludni tér.

Termelő-fogyasztó probléma

A számítástechnikában nagyon sokszor előforduló, tipikus IPC feladat a termelő-fogyasztó (*producer-consumer problem*):

- Ha a termelő azt látja, hogy az átmeneti tároló tele, aludni tér.
- Ha a fogyasztó azt látja, hogy a tároló üres, aludni tér.

Termelő-fogyasztó probléma

A számítástechnikában nagyon sokszor előforduló, tipikus IPC feladat a termelő-fogyasztó (*producer-consumer problem*):

- Ha a termelő azt látja, hogy az átmeneti tároló tele, aludni tér.
- Ha a fogyasztó azt látja, hogy a tároló üres, aludni tér.
- Ha a termelő által elhelyezett elem az utolsó amelynek helye volt, felébreszti a fogyasztót.

Termelő-fogyasztó probléma

A számítástechnikában nagyon sokszor előforduló, tipikus IPC feladat a termelő-fogyasztó (*producer-consumer problem*):

- Ha a termelő azt látja, hogy az átmeneti tároló tele, aludni tér.
- Ha a fogyasztó azt látja, hogy a tároló üres, aludni tér.
- Ha a termelő által elhelyezett elem az utolsó amelynek helye volt, felébreszti a fogyasztót.
- Ha a fogyasztó azt látja, hogy az elfogyasztott elem az utolsó, felébreszti a termelőt.

A probléma

A versenyhelyzet egyrészről a közös terület, másrészről a közös változók miatt jelentkezik.

A termelő-fogyasztó folyamatok versenyhelyzetbe kerülhetnek, mivel az elhelyezett elemek számát tároló változó nem védett.

Ilyenkor fennáll annak a veszélye, hogy a folyamatok mindegyike alvó állapotba kerül, ahonnan semmi nem mozdítja ki őket.

Szemaforok (Dijsktra, 1965)

A szemafor védett – atomos kezelésű, versenyhelyzet mentes – változó két művelettel:

- DOWN – csökkentés:
  a) ha a szemafor értéke nagyobb mint 0, eggyel csökken az értéke
  b) ha a szemafor értéke 0, a hívó folyamat blokkolódik
- UP – növelés: A szemafor értéke növekszik eggyel.

  Ha van folyamat, amely e miatt a szemafor miatt blokkolódott, egyet kiválaszthatunk és az azonnal csökkentheti az értéket.

Szemaforok a gyakorlatban

A szemaforokat az operációs rendszer szolgáltatja:

- Az OS a szemafor kezelésének idejére tiltja a megszakítást.

A szemaforokat egyenként védjük egy-egy lock változóval és TSL utasítással.

A szemafor kezelése olyan gyors, hogy nem okoz gondot a busy waiting.

Szemaforok a gyakorlatban

A szemaforokat az operációs rendszer szolgáltatja:

- Az OS a szemafor kezelésének idejére tiltja a megszakítást.
- Több processzor esetén TSL és busy waiting, de ez használható, mert rövid időről van szó.

A szemaforokat egyenként védjük egy-egy lock változóval és TSL utasítással.

A szemafor kezelése olyan gyors, hogy nem okoz gondot a busy waiting.

Producer-consumer

A következő definíciók teszik lehetővé a program működését:

```
1  #define N 100
2  #typedef int semaphore
3  semaphore mutex = 1;
4  semaphore empty = N;
5  semaphore full = 0;
```

Producer-consumer

```
1  void producer(void){
2    int item;
3    while( TRUE ){
4      item = produce_item();
5      down(&empty);
6      down(&mutex);
7      insert_item();
8      up(&mutex);
9      up(&full);
10   }
11 }
```

Producer-consumer

```
1   void consumer(void){
2     int item;
3     while( TRUE ){
4       down(&full);
5       down(&mutex);
6       item = remove_item();
7       up(&mutex);
8       up(&empty);
9       consume_item(item);
10    }
11  }
```

Szemaforhasználat

A termelő-fogyasztó példában a szemaforok kétféle használata is látható.

- A `mutex` szemafor a versenyhelyzet elkerülésére, a kölcsönös kizárás megvalósítására szolgál (bináris szemafor).
- Az `empty` és a `full` szemaforok szinkronizálásra szolgálnak.

Mutexek

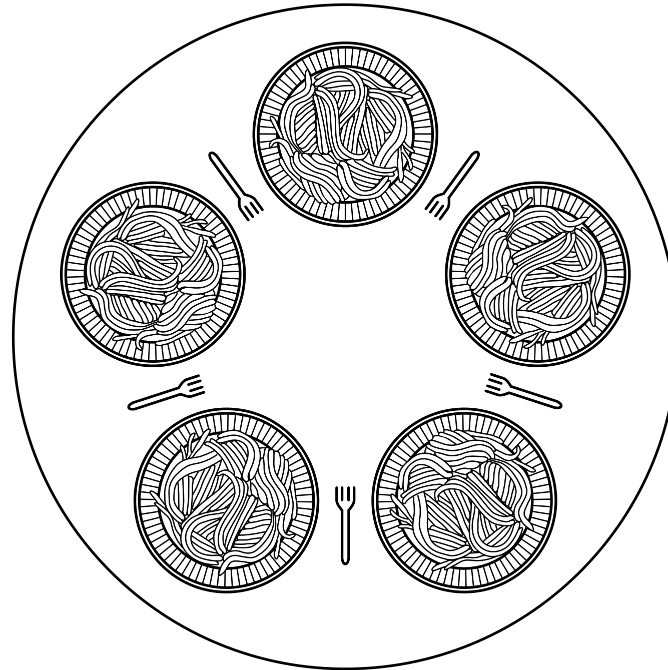
Ha a szemafort bináris szemaforként használjuk, megfelelő lehet a mutex is, amely alkalmas a kölcsönös kizárás megvalósítására. A Mutex könnyedén implementálható user területen, így alkalmas user space thread scheduler készítésére. (Mutual exclusion=kölcsönös kizárás)
A megvalósításhoz kell a TSL, a következő módon:

Mutex példa

A következő példaprogram mutexet valósít meg multi-thread környezetben TSL utasítás segítségével:

```
1  mutex_lock:                      mutex_unlock:
2    TSL RX, mutex                    MOV mutex, 0
3    CMP RX, 0                        RET
4    JZE ok
5    CALL thread_yield
6    JMP mutex_lock
7  ok:
8    RET
```

Ebédelő filozófusok esete



3. ábra. Az asztal elrendezése

Ebédelő filozófusok esete

Kör alakú asztal, rajta 5 tányér spagetti és 5 villa, minden két villa között egy tányér.

A filozófus vagy eszik, vagy gondolkodik, az evéshez két villára van szükség, ellenben egyszerre csak egy villát képes felvenni a filozófus. A közös erőforráshasználat versenyhelyzethez vezet.

Vegyük észre, hogy az egyszerű, versenyhelyzetet kiküszöbölő megoldás deadlock-hoz vezet!

Ebédelő filozófusok I.

```
1  #define N 5
2  #define LEFT (i+N-1)%N
3  #define RIGTH (i+1)%N
4  #define THINKING 0
5  #define HUNGRY 1
6  #define EATING 2
7
8  typedef int semaphore;
9  int state[N];
10 semaphore mutex = 1;
11 semaphore s[N];
```

Ebédelő filozófusok II.

```
1   /* Főciklus, folyamatonként egy.
2    * Gondolkodunk, villát szerzünk, eszünk,
3    * letesszük a villákat majd újra.
4    */
5   void philosopher( int i){
6     while( TRUE ){
7       think();
8       take_forks(i);
9       eat();
10      put_forks(i);
11    }
12  }
```

Ebédelő filozófusok III.

```
1  /* Megpróbálunk 2 villát szerezni. Ha nem
2   * megy blokkolunk a szomszédaink fel-
3   * ébresztenek, ha letették a villát.
4   */
5  void take_forks(int i){
6    down(&mutex);
7    state[i]=HUNGRY;
8    test(i);        /* bróbálkozunk */
9    up(&mutex);
10   down(&s[i]);    /* blokkolunk   */
11 }
```

Ebédelő filozófusok IV.

```
1   /* Letesszük a villát, ha valamelyik
2    * szomszédunk emiatt enni tud,
3    * felébresztjük.  */
4   void put_forks(int i){
5     down(&mutex);
6     state[i]=THINKING;
7     test(LEFT);  /* bal oldal tud enni?  */
8     test(RIGHT); /* jobb oldal tud enni? */
9     up(&mutex);
10  }
```

Ebédelő filozófusok V.

```
1  /* Amikor ezt a segédfüggvényt hívjuk, a
2   * mutex már tiltva van.
3   */
4  void test( int i ){
5    if( state[i]==HUNGRY &&
6        state[LEFT]!=EATING &&
7        state[RIGHT]!=EATING ){
8      state[i]=EATING;
9      up(&s[i]);
10   }
11 }
```

Az írók-olvasók klasszikus problémája esetén egy közös területet írnak és olvasnak folyamatok.

Az olvasók – mivel nem változtatnak az adatokan – működhetnek egyidőben, az írók azonban természetesen nem.

Ha egy író kritikus szakaszban van, az olvasók nem lehetnek abban, de tetszőleges sok olvasó lehet kritikus szakaszban, ha nincs író.

Írók és olvasók I.

```
1 typedef int semaphore;
2 semaphore mutex=1; /* rc változót védi */
3 semaphore db=1;    /* adatbázist védi  */
4 int rc=0;          /* hányan olvasnak v.
5                       akarnak olvasni  */
```

Írók és olvasók II.

```
void reader(){
  while(TRUE){
    down(&mutex);
    rc++;
    if( rc==1 )    /* Ha ez az első olvasó,
      down(&db);    * kizárja az írót        */
    up(&mutex);
    read_data_base();
    down(&mutex);
    rc--;
    if( rc==1 )   /* Ha ez az utolsó olvasó,
      up(&db);     * beengedjük az írót      */
    up(&mutex);
    use_data_read();
  }
}
```

Írók és olvasók III.

```
1  void writer(){
2    while(TRUE){
3      think_up_data();
4      down(&db);
5      write_data_base();
6      up(&db);
7    }
8  }
```

Írók és olvasók IV.

Figyeljük meg, hogy a vázolt megoldást használva az olvasók kiéheztethetik az írót, hiszen ha folyamatosan érkeznek, az író nem jut hozzá az adatbázishoz!