Elosztott zármentes feladatütemezés

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Paral Zsolt
Konzulens: Dr. Dudás Ákos

2016. október 27.

Tartalomjegyzék

1. Bevezető

A feladatümezés napjaink számítógépes rendszereinek egyik fontos alkotóeleme. A legalsó, hardverközeli szinten a processzorok egy ütemezőhöz hasonló algoritmus szerint osztják szét a feladatokat különböző feldolgozási egységeik között. Ehhez hasonló működés figyelhető meg a heterogén rendszerek világában is, ahol az egyes segédprocesszorok között válogatjuk szét a beérkező feladatok úgy, hogy azokat a lehető leghatékonyabb módon tudjuk végrehajtani. A feladatütemezés ugyancsak meghatározó a szoftveres rendszerekben is: az operációs rendszerek a felhasználói és rendszerfolyamatok, míg a szerveralkalmazások a bejövő kérések ütemezésére használják őket. A koncepció hasonló módon megállja a helyét az elosztott - és különös módon a grid - rendszerek világában is.

A többmagos processzorok térnyerésével a hagyományos, szekvenciális programozási módszerek egyre kevésbé használhatók nagy teljesítményű alkalmazások implementálásához. Az új, párhuzamos programozási módszerek közül kitűnnek a zármentes algoritmusok, mivel nem várt szálhibák és késleltetések mellett is jó teljesítményt biztosítanak. A kedvező skálázódás mellett védenek a deadlock és livelock jelenségek kialakulásától, melyek a hagyományos, zárakat alkalmazó megoldások legsúlyosabb problémái közé tartoznak.

A cikkben két új algoritmust mutatok be, amelyek képesek a feladatok elosztott, zármentes ütemezésére úgy, hogy közben feladattípusok szintjén kölcsönös kizárást is megvalósítanak. Ez a jó skálázódás és teljesítmény mel-

lett lehetővé teszi azt is, hogy az algoritmusokat akár alacsony, hardverközeli, akár magas, az elosztott rendszerek szintjén valósítsuk meg. Mindkét megoldás legfontosabb eleme egy round-robin ütemező - esetleges particionálással -, amelyet olyan fogyasztó szálak vesznek igénybe, amik bizonyos szabályokat követnek a feladatok magukhoz rendelésekor. A cikkben bemutatok egy kísérleti implementációt, amely teljesítményének növelése érdekében a szálankénti memóriaterület kihasználásával csökkenti a termelői és fogyasztói szálak versengését. Ez a megoldás csak egy szóhosszú compare-and-swap utasításokat használ, ezért a legtöbb modern architektúrán implementálható. Végül egy életszerű példán mutatom be mérési eredményeimet, melyeken látható, hogy az új, zármentes algoritmusok nagyobb megbízhatóság mellett is képesek hasonló teljesítményt nyújtani mint a hagyományos, zárakat alkalmazó megoldások.

1.1. Dolgozat felépítése

Dolgozatom következő fejezetében bemutatom a feladatütemezés általános problémáját és bevezetem a dolgozat további részében használt kifejezéseket. Ezután megmutatom, hogy a kölcsönös kizárás feltétele milyen további követelményeket támaszt a zármentességgel kapcsolatban.

A harmadik és negyedik fejezetekben bemutatom a két új ütemező algoritmust: először egy áttekintő képet adok az általános architektúráról, majd egy konkrét implementáción bemutatok néhány optimalizációs lehetőséget, bizonyítva az algoritmus rugalmasságát.

Ezt követően az ötödik fejezetben egy életszerű példán mutatok be egy konkrét implementációt, kitérve annak felépítésére, illetve a vizsgált algoritmusok részleteire. Itt több, zárakat alkalmazó és alternatív ütemezőt is összehasonlítok a zármentes algoritmusokkal, részletezve az implementáció-, és használhatóságbeli különbségeket. Ezután több szempont alapján kiértékelem a tesztelt algoritmusok teljesítményét: vizsgálni fogom a bemenetekkel, a feladattípusok számával, illetve a termelői és fogyasztói szálak számával történő skálázódásukat is.

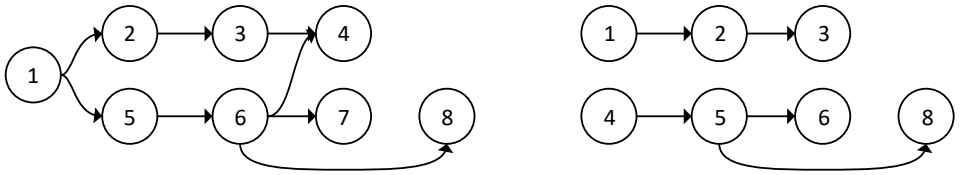
A gyakorlati példa után részletezek néhány olyan terület, ahol az algoritmus még hasznos lehet. Itt kitérek a terület által állított nehézségekre, lehetséges megoldásaikra, illetve mutatok egy elméleti modellt az algoritmusok egy lehetséges implementációjára.

Az utolsó fejezetben összefoglalom dolgozatom eredményeit, illetve kitekintést nyújtok arról, hogy ezen eredményeket milyen további kutatásokban lehetne használni.

2. Feladatütemezés

A feladatütemezés legalapvetőbb modellje szerint az ütemezésben résztvevő szálakat két alapvető csoportra bonthatjuk: a termelőkre, amelyek képesek feladatok előállítására, illetve fogyasztókra, amelyeken a feladatok tényleges végrehajtása történik. A feladatütemezés általános esetben NP-nehéz probléma [1], de különböző megkötésekkel ez a komplexitás csökkenthető. A

dolgozatban bemutatott ütemező például maximum egy függőséget engedélyez egy meglévő és egy újonnan létrehozott feladat között: ez a feladatokat úgynevezett feladatcsoportokba (feladattípusokba) osztja, ahol egy csoportba azon feladatok kerülnek, amelyeknek csak ugyanabban a feladatcsoportban van függősége. A feladatcsoportok legnagyobb előnye, hogy a különböző csoportok feldolgozása történhet egymástól teljesen függetlenül (és így párhuzamosan). Egy általános ütemező és a dolgozatban tárgyalt megoldás közötti koncepcionális különbségeket az 1. ábrán láthatjuk.



1. ábra. Az általános és feladattípusos ütemező közötti különbség

Ezen felül gyakori követelmény, hogy az ütemezés közben valamilyen extra szabályrendszert is be kell tartani ahhoz, hogy az ütemezett rendszer optimális teljesítménnyel működjön és/vagy konzisztens állapotban maradjon. Ilyen szabályok lehetnek például különböző feladatprioritások, valamilyen követendő heurisztika [2] vagy pedig kölcsönös kizárás megvalósítása egyes feladatok/feladatcsoportok között [3]. Legtöbbször ezen szabályok betartatása egy központi ütemező adatstruktúra feladata. Ebben a dolgozatban azt vizsgálom meg, hogy milyen előnyökkel jár az, ha ezen szabályok kezelését minél inkább a fogyasztó szálakra bízzuk. Ez a megközelítés több előnnyel

is jár, ha figyelembe vesszük az elsző bekezdésben bemutatott megkötést is: mivel a feladatcsoportok egymástól függetlenül feldolgozhatóak, ezért a központi adatstruktúra feladata vagy jelentősen leegyszerűsödik, vagy a használt szabályrendszertől függően akár teljesen elhagyható is lehet.

Fontos továbbá tisztázni a zármentesség definícióját. Mivel a magyar fordítás nem tükrözi az eredeti definíciókban található apró különbségeket, ezért amikor a helyzet megköveteli, akkor az angol szakkifejezéseket fogom használni:

1. *obstruction-free algorithm*: egy algoritmus obstruction-free, ha egy szál teljes izolációban futtatva garantáltan befejezi futását véges lépésszám után
2. *lock-free algorithm*: egy lock-free rendszerben legalább egy szál garantáltan munkát végez véges lépésszám után. Minden *lock-free* algoritmus egyben *obstruction-free* is. A továbbiakban a zármentesség fogalmát a *lock-free* algoritmusok szinonimájaként használom.
3. *wait-free algorithm*: egy wait-free rendszerben minden szálnak adható egy felső határ, amennyi lépésszám alatt garantáltam elvégzi a ráosztott feladatot. Minden *wait-free* algoritmus egyben *lock-free* is.

Az ütemező tervezésénél fontos szempont, hogy az algoritmusok változatos architektúrákon implementálhatóak legyenek: a hardverközeli megvalósításoktól kezdve egészen az elosztott rendszerekig. Ennek érdekében minden algoritmus moduláris felépítésű, ahol az egyes modulok könnyen cserélhetők, egymással egy standardizált interfész segítségével kommunikálnak. A szink-

ronizációt igénylő műveletek (mint például a kölcsönös kizárás) legtöbbször valamilyen központi, egységet igényelnek, azonban megfelelő absztrakciók használatával ez a probléma is áthidalható. Az architektúrális szempontok mellett az alábbi teljesítmény és megbízhatósági garanciákra törekedtem:

1. egy feladat végrehajtása *atomi* az összes, azonos feladattípusba tartozó feladatra nézve
2. zármentes feladatütemezés, amennyiben az ehhez szükséges és elégséges feltétel teljesül
3. kedvező memóriafogyasztás és skálázódás
4. minimális beavatkozás a termelői és fogyasztói oldalon, amennyiben az architektúra ezt lehetővé teszi

2.1. Zármentesség és kölcsönös kizárás

A feladatütemezés zármentes megvalósítása aktív kutatási terület [4, 5], azonban ezek jellemzően a feladatütemezés általános problémájával foglalkoznak; a korábban tárgyalt megkötések lehetőségeket biztosítanak új algoritmusok kifejlesztéséhez. A teljes zármentesség elérhető egy általános célú feladatütemezőnek, azonban a feladattípusokon belüli kölcsönös kizárás szabályának bevezetésével ez elérhetetlenné válik:

Tétel. *A kölcsönös kizárást biztosító feladattípusonkénti ütemezés zármentességének feltétele, hogy a feladattípusok száma legalább az egyidejű fogyasztó szálak számával legyen egyenlő.*

Bizonyítás. Az alábbi informális bizonyítás segítségével az elözö állítás belátható:

1. Tételezzünk fel egy feladatütemezöt T feladattípussal és $C = T + 1$ fogyasztó szállal.
2. A legrosszabb eshetőséget vizsgálva mind az N feladattípust egy-egy fogyasztó szálhoz osztjuk.
3. Ekkor ha egy kiosztott szál sem képes munkát végezni, akkor a $T+1-ik$ szálnak nem tudunk feldolgozandó feladatot adni (a kölcsönös kizárás feltételének megsértése nélkül).

Ezzel szemben ha $C = T$, akkor mindig találunk egy olyan feladattípust, amelyik vagy szabad volt. Az alábbi két eset képzelhető el, a legrosszabb eshetőséget figyelembe véve (az összes feladattípust kiosztottuk egy-egy fogyasztó szálnak, és egy szál éppen befejezte saját feladattípusának feldolgozását):

1. Ha megpróbálunk egy másik feladattípust ütemezni, és ez sikeres, akkor a vizsgált szál munkát végzett, ezzel kielégítve a zármentesség feltételét.
2. Ha nem sikerül egy másik feladattípust ütemezni, akkor egy másik szálnak osztottuk ki a korábban felszabadult feladattípust, ezzel ugyancsak teljesítve a zármentesség feltételét (ez a garancia azonban a *wait-freedom* eléréséhez már nem elegendő).

□

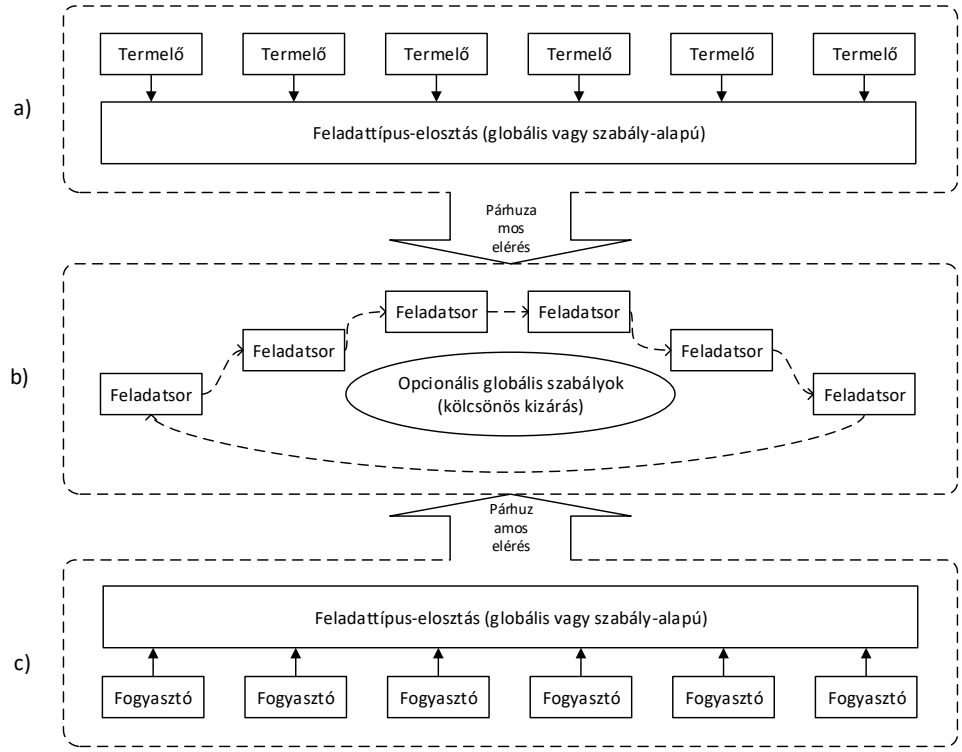
3. Round-robin ütemező

3.1. Leírás

Az első ütemező algoritmus az úgynevezett globális round-robin ütemező. Működésének alapja, hogy a fogyasztók szabadon be tudják járni az ütemezendő feladattípusokat tartalmazó sorokat olyan módon, hogy véges lépésszám után visszaérnek ahhoz a sorhoz, ahonnan az ütemezés indult (amennyiben nem vettünk fel újabb feladattípust). Az architektúrát a(z) 2. ábra írja le. Ez három fő részből épül fel:

a) Termelői oldal, ahol a feladattípusnak megfelelően meghatározzuk a megfelelő feladatsort. Ez két módon történhet, több körülménytől függően:
   1) A feladattípusok száma előre nem ismert, egy feladattípus csak egyszer szerepelhet (például kölcsönös kizárás esetén): szükség van egy globális szinkronizációs pontra egy új feladattípus hozzáadása esetén. Ilyenkor sincs azonban szükség szinkronizációra egy egyszerű feladat hozzáadása esetén.
   2) Egy feladattípus többször is szerepelhet, és/vagy tudunk olyan szabályt megadni, amely egy feladatsort egyértelműen hozzárendel egy feladattípushoz: nincs szükség globális szinkronizációra, a termelők képesek maguk meghatározni a használandó feladatsort.

b) Az ütemező, amely elosztott jellegéből adódóan implementációtól függően csak koncepcionálisan létezik. Itt találhatóak az egyes feladattí-



2. ábra. Az ütemező architektúrája

pusokhoz tartozó feladatsorok, illetve - amennyiben szükség van - rá egy globális adatstruktúra, amely a szinkronizációt igénylő szabályok kezelésére szolgál.

c) A fogyasztói oldal, ahol a feladatsorokat meghatározó algoritmus megegyezik az első pontban tárgyalttal.

Ez az ütemező számos előnnyel jár, amelyek közül a legfontosabb, hogy a változó feldolgozási idővel rendelkező feladattípusokat is kiegyensúlyozott késeltetéssel tudja futtatni. Ez annak köszönhető, hogy az algoritmus kölcsönös

kizárás esetén képes átugrani a feldolgozás alatt álló feladatokat, így előnyben részesítve a nem zárolt feladatsorokat. Az algoritmus további előnyökkel bír amennyiben egy memóriatérben dolgozó implementációt használunk:

1. Egy feladatsorok közötti lépést teljesítmény szempontjából olcsó műveletekkel tudjuk megvalósítani, aminek köszönhetően a fogyasztói oldal szinte soha sem szűk keresztmetszete az ütemezőnek.
2. A használt adatstruktúra - amennyiben garantált egy destruktor/finalizer futtatása szálhiba esetén - nem igényel termelői/fogyasztói közreműködést a kölcsönös kizárás megvalósításához (a fogyasztók optimalizációs céllal továbbra is jelezhetik egy feladattípus feldolgozásának befejezését)

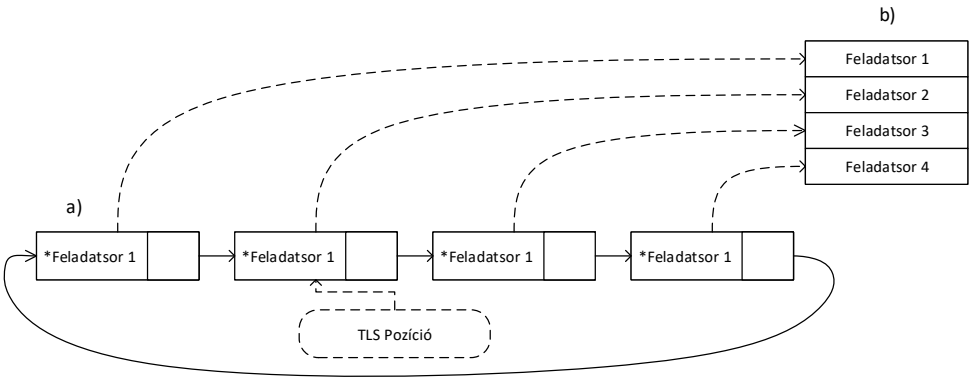
3.2. Implementáció

Példaképpen egy C# nyelven írt implementációt mutatok be. Mivel az ütemező egyes részei közös memóriatérben futnak, ezért több optimalizációt is alkalmazhatunk. Ezek közül a legjelentősebb, hogy a feladattípusok hozzáadásának kezelését egy nagy teljesítményű, zármentes hashtábla segítségével érjük el. Ez az adatstruktúra egy, a szakirodalomban megtalálható algoritmusra, a Split-Ordered List-re [6] épül, azonban néhány fontos változtatással javít az eredeti cikkben tárgyalt változat teljesítményén:

1. A .NET keretrendszer sajátosságait figyelembe véve és a gazdag futási idejű típusinformációt kihasználva olyan referenciákat használunk, amelyek egy objektumon kívül képesek egy logikai (igaz/hamis) érték

tárolására. A módszer alkalmazásával jelentősen csökkenthető a felesleges memóriaallokációk mérete, ezzel mérsékelve a *Garbage Collector-ra* nehezedő nyomást.

2. A hashtábla implementálásánál a vödörlista növelésére egy új, a vödrök számát geometriai módon növelő algoritmust használtam. Ennek segítségével a korábban $O(n)$ beszúrás művelet komplexitását sikerült $O(1) - re$ csökkenteni.



3. ábra. Az implementáció architektúrája

Ezeket az optimalizációkat figyelembe véve a 3 ábrán látható implementációt vázolhatjuk fel. Az imént említett központi hashtáblát a 3 ábra b) pontjában láthatjuk. Az ütemező középpontjában azonban az a) pontban látható körkörösen láncolt lista áll, amely zármentes hozzáférést és beszúrást biztosít a termelők és fogyasztók számára egy MPSC (multiple producer, single consumer) sor [7] segítségével. Mivel nem kell támogatnia az egyes feladattípusok eltávolítását, ezért a beszúrás művelete (a 1. kódrészleten lát-

ható `AddTaskGroup`) megvalósítható egy egyszerű *compare-and-swap* ciklus segítségével. Ez a művelet mindig zármentes viselkedést biztosít: A 8 sorban található CAS utasítás csak akkor nem lesz sikeres, ha egy másik szál már elvégezte az inicializációt. Hasonló módon a 21 sorban lévő CAS ciklus egy másik szál sikeres `AddTaskGroup` művelete esetén lesz sikertelen.

```
1  public void AddTaskGroup(ILockfreeQueue<T> queue)
2  {
3      var node = new Node(queue);
4
5      var currentHead = Volatile.Read(ref head);
6      node.Next = node;
7      if (currentHead == null &&
8          Interlocked.CompareExchange(ref head, node, null) == null)
9
10     {
11         Interlocked.Add(ref numQueues, 1);
12         return;
13     }
14
15     currentHead = Volatile.Read(ref head);
16     Node next;
17     do
18     {
19         next = Volatile.Read(ref currentHead.Next);
20         node.Next = next;
21     } while (Interlocked.CompareExchange(ref currentHead.Next,
22                                          node, next) != next);
23     Interlocked.Add(ref numQueues, 1);
24 }
```

Kódrészlet 1. Az `AddTaskGroup` művelet

A `GetTask` képezi az ütemező második elengedhetetlen metódusát: egy feladat kinyerését az ütemezőből. Ez a művelet zármentes, amennyiben teljesül az előző fejezetben bemutatott minimális követelmény, azaz *feladattípusok száma* $\geq$ *fogyasztók száma*. A feladatlekérdezés implementációját a 2. kódrészletben láthatjuk. Az algoritmus lényege a következő:

1. A fogyasztó szál lefoglal egy feladattípust, amit egy atomi flag bebillentésével jelez (22. sor)
2. A rákövetkező fogyasztók átugorják a lefoglalt feladattípust
3. A feladat befejeztével az eredeti fogyasztó szál új feladattípust keres, de először felszabadítja az általa korábban lefoglaltat.
4. Szálhiba esetén a destruktorban kerül felszabadításra a foglalt erőforrás

Az utolsó, és egyben opcionális művelet az úgynevezett `SignalFinished`. Ez olyan esetekre szolgál, amelyekben a fogyasztó szál nem végez tartósan munkát a feladatütemezőnél: ilyenkor az elvégzett munka után a `SignalFinished` metódus hívásával tudjuk jelezni a művelet befejezését, aminek hatására az adott feladattípus felszabadíthatóvá válik. Bár az ütemező törekszik a termelők és fogyasztók segítsége nélkül működésre, azonban egy fogyasztó szál futási állapotának megállapítása egyenlő lenne a Halting probléma megoldásával.

```
1  public bool GetTask(out T task)
2  {
3      var node = threadNode.Value;
4      if (node != null)
5      {
6          Volatile.Write(ref node.IsReserved, 0);
7          node = Volatile.Read(ref node.Next);
8      }
9      else if (Volatile.Read(ref head) != null)
10     {
11         node = head;
12     }
13     else
14     {
15         task = default(T);
16         return false;
17     }
18
19     var retries = Volatile.Read(ref numQueues) * 2;
20     for (var i = 0; i < retries; i++)
21     {
22         while (Interlocked.Exchange(ref node.IsReserved, 1) == 1)
23         {
24             node = Volatile.Read(ref node.Next);
25         }
26
27         if (node.Queue.TryPop(out task))
28         {
29             threadNode.Value = node;
30             return true;
31         }
32
33         Volatile.Write(ref node.IsReserved, 0);
34         node = Volatile.Read(ref node.Next);
35     }
36     task = default(T);
37     threadNode.Value = null;
38     return false;
39 }
```


Kódrészlet 2. A `GetTask` müvelet

4. Szegmentált ütemező

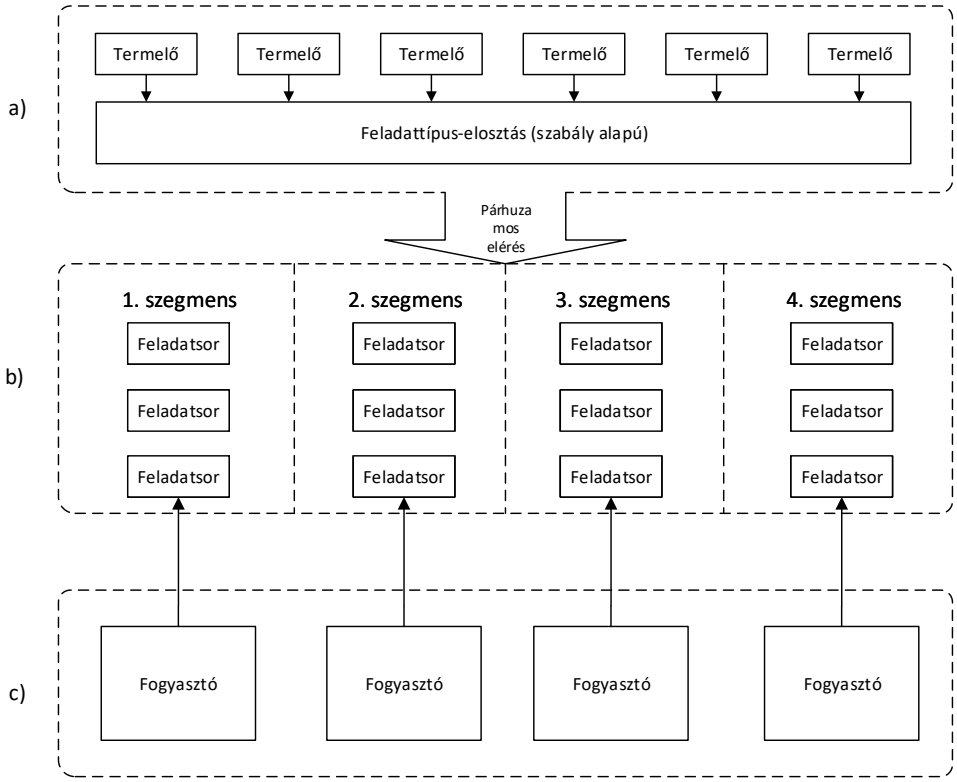
4.1. Leírás

A második ütemező algoritmus a feladattípusok szerinti szegmentálhatóságot használja ki a fogyasztók közötti versengés minimalizálásának érdekében. Itt a fogyasztók maximális számának előzetes ismeretével megadathatunk egy olyan fogyasztó-feladatsor hozzárendelési szabályt, ahol minden feladattípushoz pontosan egy fogyasztót rendelünk. Ez automatikus biztosítja a kölcsönös kizárás feltételét is, amennyiben egy feldolgozó egység egyszerre csak egy feladatot tud végrehajtani. Az architektúra, hasonlóan a round-robin ütemezőhöz, három rétegből épül fel, amelyet a 4. ábra mutat be.

a) Az első réteg - ahol a termelők feladatsorokhoz rendelése történik - lényegében megegyezik a round-robin ütemezőjével, azzal a különbséggel, hogy itt minden esetben elegendő a szabályalapú hozzárendelés; globális adatstruktúrára nincs szükség.

b) A második réteg, amely implementációtól függően ugyancsak hiányozhat, már jelentősen eltér az eddigiektől. A feladattípusok fogyasztókhoz vannak rendelve, a párhuzamosság helyes működését (és így a zármentességet is) csak a termelői oldalon kell garantálni. Ez jelentősen csökkentheti a szinkronizáció költségét abban az esetben, ha nagy volt a versengés a fogyasztók között, azonban a bonyolultabb működés növeli az algoritmus futási idejű komplexitását.

c) A végső rétegben láthatjuk, hogy a fogyasztók ismerik a hozzájuk ren-

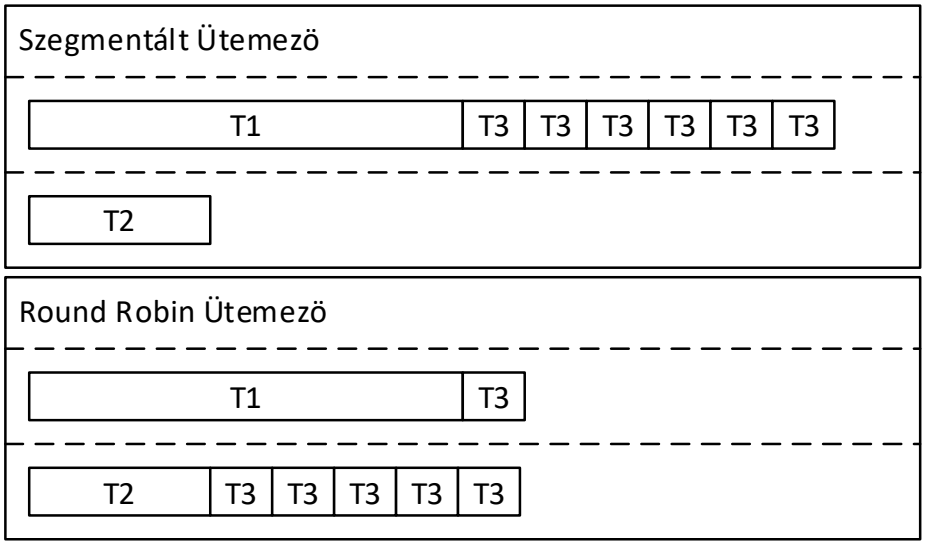


4. ábra. A szegmentált ütemező architektúrája

delt feladatsorokat, így nincs szükség egy további lépésre azoknak meghatározásához.

A szegmentált ütemezőnek a hangsúlyozott előnyök mellett természetesen hátrányai is vannak. Az első, hogy kölcsönös kizárás megvalósítása esetén ismernünk kell az egy időben aktív fogyasztó szálak maximális számát. A második, és egyben súlyosabb probléma, hogy az algoritmus kedvezőtlen helyzetben vett futási ideje jelentősen rosszabb, mint a round-robin ütemezőé, mivel nem tudja átlapolni a műveleteket az egyes fogyasztó egységek

között. Erre egy szemléletes példát a 5. ábrán láthatunk.



5. ábra. Worst-case ütemezési különbségek

4.2. Implementáció

A round-robin ütemezőhöz hasonlóan a szegmentált ütemező is három műveletet (`AddTaskGroup`, `GetTask`, illetve `SignalFinished`) definiál. Az előző implementációhoz képest egy feladattípus hozzáadása jelentősen leegyszerűsödött: a 3. kódrészleten láthatjuk, hogy egy tömb ciklikus bufferként használatával érjük el a feladattípusok egyenletes elosztását. A megoldás legnagyobb hátránya, hogy nem terjeszthető ki könnyen egy teljesen elosztott rendszerre: ilyen esetekben a termelői oldalon, illetve a feladattípusokat

tartalmazó csomópontokban definiált heurisztikára kellene hagyatkoznunk.

```
1  public void AddTaskGroup(ILockfreeQueue<T> queue)
2  {
3      var target = Volatile.Read(ref index);
4      while (Interlocked.CompareExchange(ref index,
5                                         (target + 1) % consumers,
6                                         target) != target)
7      {
8          target = Volatile.Read(ref target);
9      }
10     threadBlocks[target].AddQueue(queue);
11 }
```

Kódrészlet 3. Az `AddTaskGroup` müvelet

A komplexitás jelentős része a `GetTask` műveletben található, amelynek működését a 4. kódrészlet részletezi. Az algoritmus két fő részből épül fel:

1. A 7. sorban kezdődő `if` blokk inicializálja a szálhoz tartozó szegmenst, amennyiben ez még nem történt meg. A 12. sorban található feltétel betartása csak abban az esetben fontos, ha kölcsönös kizárást akarunk megvalósítani a feladattípusok között, elhagyásával az algoritmus tetszőleges fogyasztó szál ütemezésére képes.
2. A második, 25. sornál kezdődő részben történik egy feladat tényleges kinyerése a szálhoz rendelt feladatsorokból. A művelet annyira gyors, hogy sok szál és gyors feladatlefutási idők esetén a thread local storage-ben található érték kinyerése bizonyul az algoritmus szűk keresztmetszetének. Ilyenkor a 35. sorban látható backoff stratégiát finomhangolásával jelentős (1-2 nagyságrendnyi) teljesítménynövekedés érhető el.

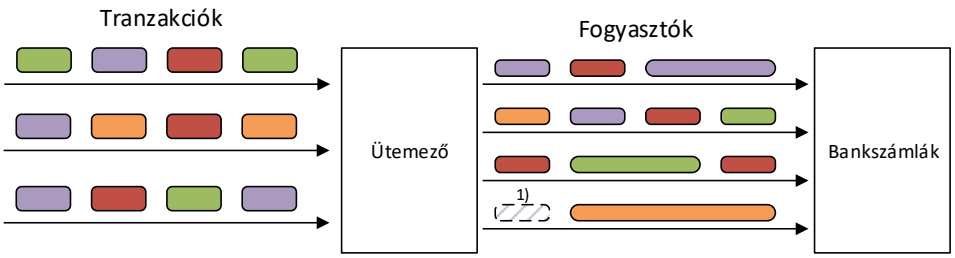
```
1  public bool GetTask(out T task)
2  {
3      var tr = threadRef.Value;
4      var node = tr.Current;
5      var id = tr.Id;
6
7      if (node == null)
8      {
9          if (id == -1)
10         {
11             id = Interlocked.Increment(ref threadId);
12             if (id >= threadBlocks.Length)
13                 throw new Exception();
14             threadRef.Value.Id = id;
15         }
16         var head = Volatile.Read(ref threadBlocks[id].Head);
17         if (head == null)
18         {
19             task = default(T);
20             return false;
21         }
22         node = threadRef.Value.Current = head;
23     }
24
25     var numQueues = threadBlocks[id].NumQueues;
26     var retries = numQueues * 1000;
27     for (var i = 0; i < retries; i++)
28     {
29         if (node.Queue.TryPop(out task))
30         {
31             threadRef.Value.Current = Volatile.Read(ref node.Next);
32             return true;
33         }
34         node = Volatile.Read(ref node.Next);
35         Thread.Yield();
36     }
37     threadRef.Value.Current = node;
38     task = default(T);
39     return false;
40 }
```

Kódrészlet 4. A `GetTask` művelet

5. Egy gyakorlati példa

5.1. Leírás

Egy banki rendszerben *tranzakciókat* dolgozunk fel olyan módon, hogy az egyes tranzakciókat csak atomi módon hajthatjuk végre a tranzakció céljaként megjelölt *bankszámlán*. A rendszer általános architektúrája a 6. ábrán látható, az implementáció pedig C# nyelven, a .NET keretrendszer segítségével készült.



6. ábra. A banki tranzakciókezelő rendszer

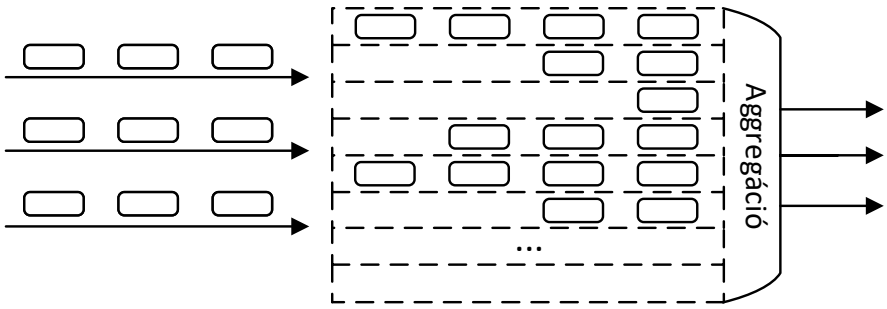
A `1)` számmal jelölt rész mutatja a kölcsönös kizárás megvalósítását: az adott bankszámlára egy másik szálon éppen folyamatban van egy tranzakció, így ugyanarra a bankszámlára másik művelet nem ütemezhető. Az implementált és tesztelt megoldások több csoportba oszthatók:

1. Globális zárolási táblát használó megoldások
2. Zárakat alkalmazó, sor alapú algoritmusok
3. Zármentes, sor alapú megoldások

5.1.1. Globális zárolási tábla

Ezen megoldások kihasználják a .NET beépített párhuzamos adastruktúráit és algoritmusait. A megoldások modulárisak, és mivel rendkívül kevés módosítást igényelnek egy meglévő kódbázis esetén, ezért könnyen integrálhatóak meglévő rendszerekbe. Az algoritmusok finomhangolt (bankszámla-szintű) zárolást alkalmaznak, így teljesítmény szempontjából reprezentatívak lehetnek egy valóélet-beli rendszerben használt általános megoldásról. Ebben a kategóriában két algoritmust teszteltem:

1. Egy ThreadPool és `Task-alapú` párhuzamosságot használó rendszer: ez a megoldás teljesen megkerüli a tranzakciók tárolását, azokat közvetlenül futtatja egy `Task` formájában. Segítségével kedvező teljesítmény érhető el, amennyiben a szálak közötti versengés nem túl nagy. Legnagyobb hátránya, hogy a függő tranzakciók csak egy saját `TaskScheduler` ütemező implementálása után érhetők el. Nem támogatja a tranzakciók rendezését.
2. Egy `BlockingCollection<T>` típust használó algoritmus, amely 62 párhuzamos sorba osztja a beérkező tranzakciókat majd ezekből tetszőleges sorrendben ütemezi a tranzakciókat a fogyasztó szálakra. Kedvező teljesítményt biztosít, azonban néhány területen skálázódása elmarad a többi megoldástól. Nem támogatja a tranzakciók rendezését. Az algoritmus architektúráját a 7. ábra mutatja be.



7. ábra. A `BlockingCollection` algoritmus architektúrája

5.1.2. Sor alapú megoldások

A sor alapú megoldások legnagyobb előnye, hogy a belső sorok implementációinak cseréjével változatos funkciók támogatására képesek: FIFO/LIFO viselkedés, prioritások, halmaztulajdonság betartatása, vagy akár a sorrendiség teljes elhagyása egy `Bag` adatstruktúra alkalmazásával. Az algoritmusok nehezebben integrálhatók meglévő rendszerekbe, mivel használatuk legtöbb esetben architektúrális változtatást igényel. A dolgozatban bemutatott két zármentes algoritmus mellett implementáltam a round robin ütemező egy zárakat alkalmazó változatát is: ez jó összehasonlítási alapként szolgál a zármentes algoritmusok skálázódási és teljesítménybeli viselkedéséhez.

5.2. Teljesítménymérések

A mérések elvégzésénél a .NET keretrendszer 4.6.1-es változatát használtam, ahol a párhuzamos, *szerver Garbage Collector-t* választottam a teljesítmény

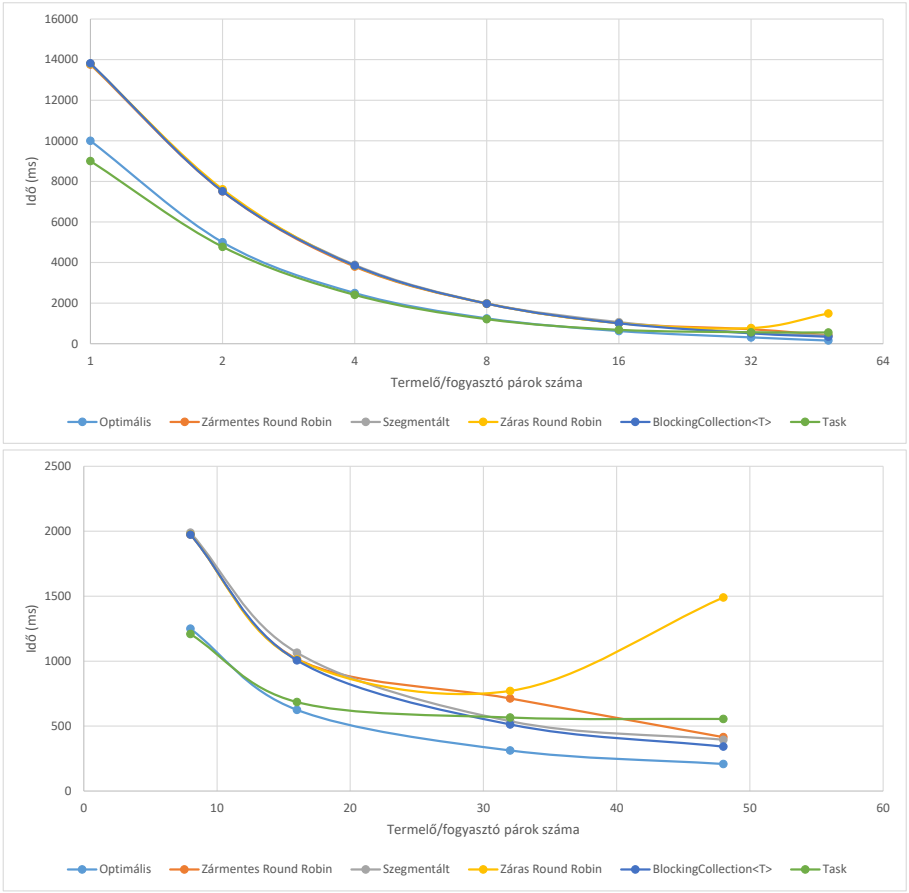
növelése érdekében. Mivel a zármentes algoritmusok jellemzően sok memóriaallokációval járnak, ezért egy nem-optimális GC választás akár 2-3x teljesítményromláshoz is vezethet. A méréseket egy modern, 8 magos Intel processzoron végeztem. A skálázódási méréseket kétszer végeztem el: először teljes versengés mellett, majd egy mesterséges, $1ms$ hosszúságú tranzakciót szimulálva.

5.2.1. Szálak

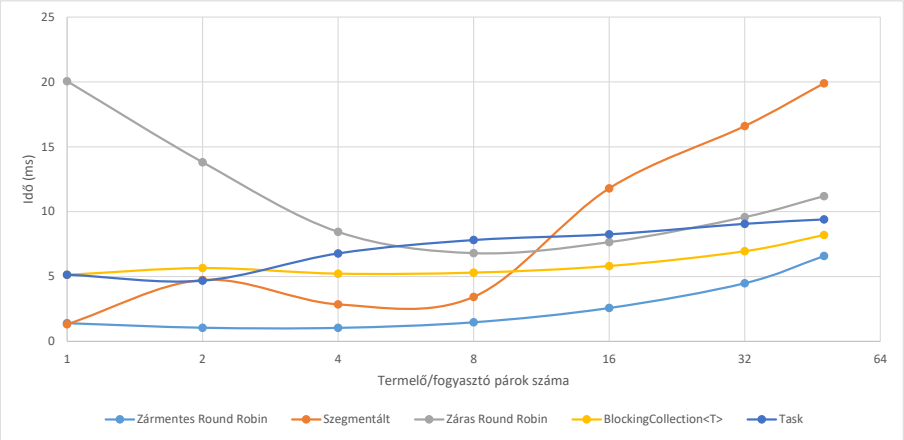
A 8. ábrán az algoritmusok szálak számával való skálázódását figyelhetjük meg abban az esetben, amikor alacsony a szálak közötti versengés. Itt jól látszik, hogy minden ütemező algoritmus közel lineárisan skálázódik az aktív szálak számával (egy konstans overhead mellett). A `Task-alapú` megoldás a szálak alacsony száma mellett még az optimális esetet is képes megelőzni. Ennek hátterében a termelő és fogyasztó szálak egységesített threadpool-ja áll: ennek segítségével kevés szál esetén is képes a feldolgozás párhuzamosítására. A szálak számának növekedésével ez az előny fokozatosan elveszik, és 16 száltól kezdve a zármentes megoldások már jobban teljesítenek, mint a `Task-alapú` algoritmus. Megfigyelhető továbbá, hogy a zárakat alkalmazó round-robin ütemező - ellentétben zármentes változatával - nem skálázódik egy bizonyos határ felett. A várt eredménynek megfelelően a szegmentált ütemező magasabb versengés mellett 10-20%-os teljesítményelőnyre tesz szert az addig vezető zármentes round robin ütemezővel szemben.

Magas versengés mellett más képet mutatnak a tesztelt algoritmusok: ek-

kor lényegében az egyes megoldások overhead-jét mérjük. A 9. ábrán láthatjuk, hogy csak a zármentes round robin ütemező teljesítményvesztése mutat lecsengő tendenciát. A szegmentált ütemező teljesítménye a szálak számának növekedésével jelentősen romlik. Ennek oka, hogy egy `GetTask` művelet jelentősen gyorsabb, mint a termelő szálak `AddTask` művelete. A kedvezőtlen hatás az ütemező backoff stratégiájának finomhangolásával javítható (illetve egyes esetekben teljesen kiküszöbölhető).



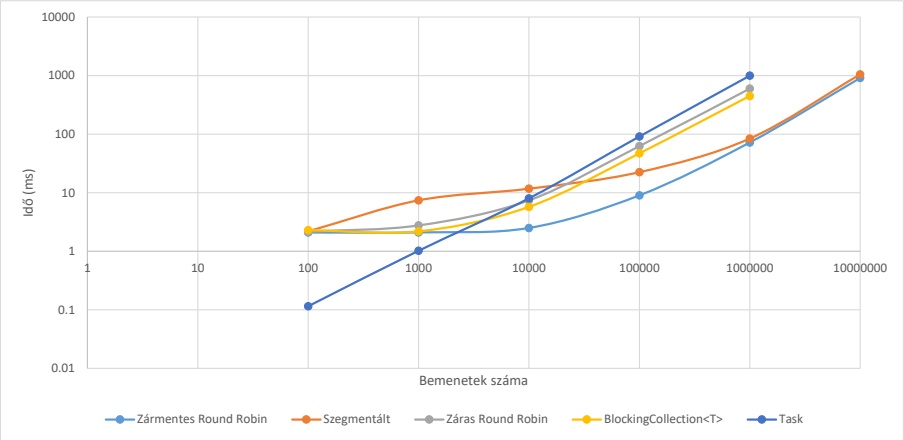
8. ábra. Skálázódás szálakkal (alacsony versengés)



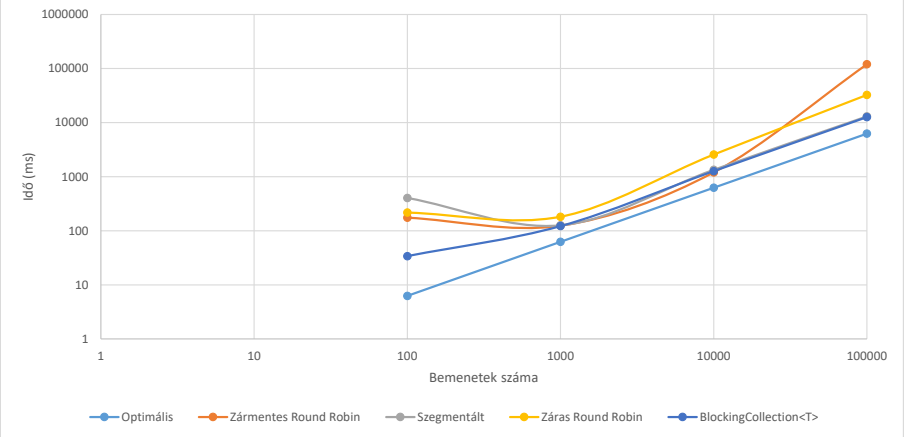
9. ábra. Skálázódás szálakkal (magas versengés)

5.2.2. Tranazkciók

Ezután megvizsgáltam az ütemezők bemenetekkel való skálázódását. A 10. ábrán látható, hogy a legtöbb nem zármentes ütemező lineárisan skálázódik a bemenetek számával. A zármentes változatok skálázódása szublineáris mindaddig, amíg a használt adatstruktúrák által okozott overhead-et el nem érjük, amely után ezek is lineárisként folytatódnak. A zármentes round robin és szegmentált ütemező konstans teljesítményköltsége jelentősen alacsonyabb, mint a többi vizsgált algoritmusé: ez teszi lehetővé egy további mérés elvégzését (10000000 elemre). Alacsony versengés mellett minden ütemező skálázódása lineáris, ahogyan ezt a 11. ábra is mutatja.



10. ábra. Skálázódás bemenetekkel (magas versengés)

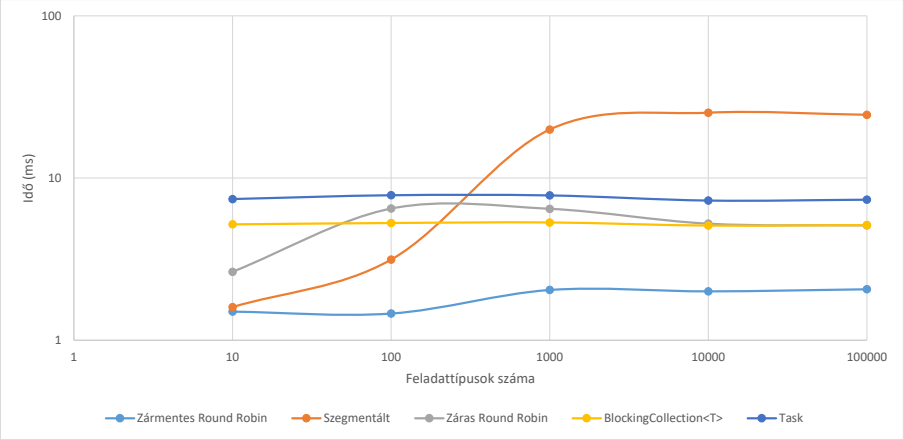


11. ábra. Skálázódás bemenetekkel (alacsony versengés)

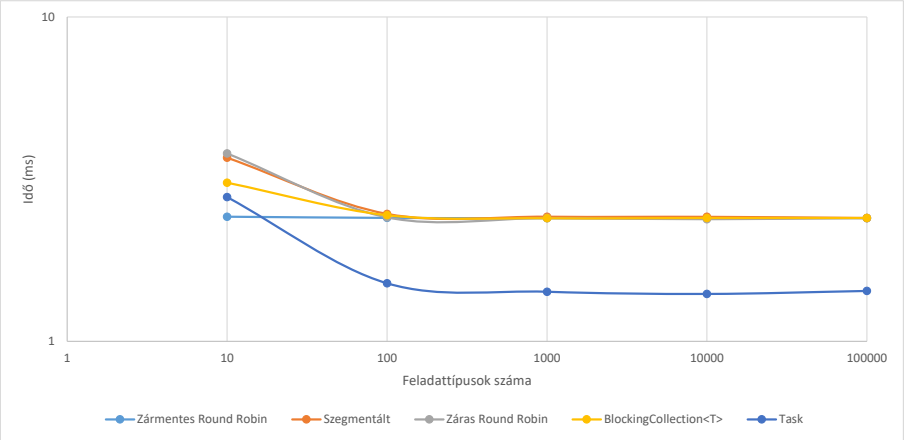
5.2.3. Bankszámlák

Fontos ezen felül a feladattípusokkal (bankszámlákkal) történő skálázódás. Itt mind a magas, mind az alacsony versengési helyzet esetén konstans komplexitást figyelhetünk meg. Az alacsony versengési esetben (12. ábra) a zármentes round robin ütemező jelentősen jobb teljesítményt biztosít, mint bár-

mely másik alternatíva, és a magas versengési szituációban is lépést tart versenytársaival (13. ábra).



12. ábra. Skálázódás feladattípusokkal (magas versengés)



13. ábra. Skálázódás feladattípusokkal (alacsony versengés)

5.3. Kiértékelés

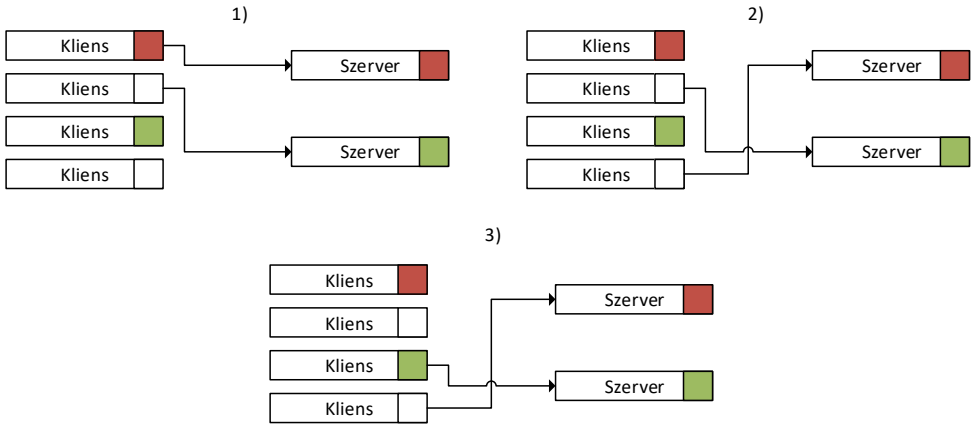
A mérések eredményéből kitűnik, hogy a legtöbb esetben a zármentes algoritmusok teljesítménye megközelíti, és sok esetben meg is haladja a hagyományos, zárakat alkalmazó alternatívákét. Amennyiben az implementáció bonyolultsága és az esetleges architektúrális változtatások nem jelentenek problémát, addig a bemutatott zármentes algoritmusok alkalmazása előnyös lehet valóélet-beli helyzetekben is. A deadlock és livelock jelenségek kialakulásától történő védelem miatt pedig jó választások lehetnek a megbízhatóságot előnyben részesítő rendszerek esetében is.

6. Felhasználási területek

6.1. Webszerverek és grid rendszerek

A weben számos példáját találjuk annak, hogy több különböző feladatcsoportot kell szétosztani feldolgozóegységek között valamilyen szabály vagy szabályok betartatása mellett. A legegyszerűbb példa erre a legtöbb webes stackekben megtalálható load balancer, amelynek hatékony implementálása rendkívül aktív kutatási terület [8, 9]. Itt a beérekező forgalmat osztjuk szét különböző szerverek között különböző heurisztikák, statisztikák, aktuális terheltség, illetve felhasználók szerint Az új algoritmus segítségével a felhasználók részben (vagy teljesen) maguk döntenének arról, hogy melyik szerver szolgálja ki őket. A kölcsönös kizárást, illetve szinkronizációt igénylő

műveletek egy globális szerver alkalmazásával továbbra is megvalósíthatók.

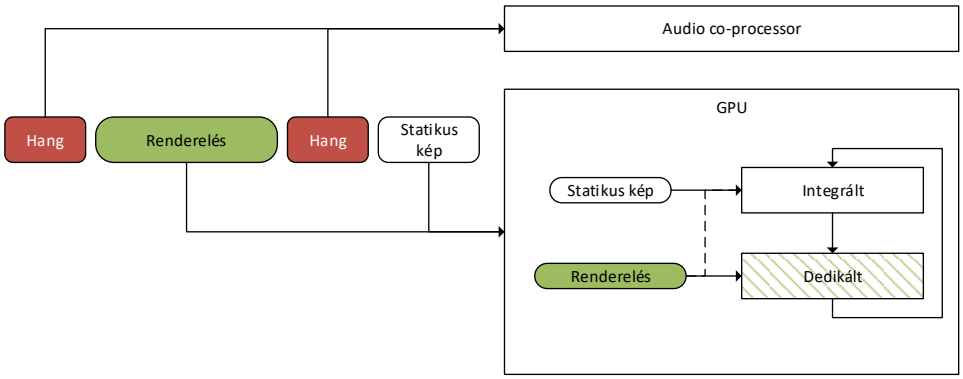


14. ábra. Egy egyszerű round-robin load-balancer

Itt egy kezdetleges szabályrendszert alkalmazó rendszert figyelhetünk meg, azonban kevés kérés esetén már ezzel is jó ütemezést kapunk. Ennél azonban egy modern load-balancer algoritmus jelentősen bonyolultabb, ezért a jövőben érdekes kutatási terület lenne egy olyan szabályrendszer megalkotása, amely versenyre kelhet napjaink load-balancer megoldásaival. Ennek két nehézségét látom, amelyek közül első a szabályok megtervezése. Egy központosított rendszer több statisztikához fér hozzá a használt szerverek állapotáról, amelyek kliensoldali heurisztikákkal történő helyettesítése gondos tervezést igényel. A második probléma a kliens-oldali szerverlista karbantartása olyan módon, hogy az ütemezés során a kliens és szerver közötti kommunikációt minimálisra csökkentsük.

6.2. Heterogén rendszerek belső ütemezése

A feladatütemezés egy másik jellemző felhasználási területe, amikor egy azonos feladatot mindig ugyanahhoz a feldolgozóegységhez (vagy csoporthoz) kell továbbítani. Erre egy jellemző példa lehet a heterogén rendszerek megfelő ütemezése, amellyel számos kutatás foglalkozik [10, 11]. Egy optimális szabályrendszer megtalálása, a load-balancer példájához hasonlóan, egy olyan terület, amelyben további kutatásokat lehet végezni dolgozatom eredményei alapján. Egy egyszerüsített implementáció, amely egy többszintű ütemező összeállításával mindkét tárgyalt algoritmus előnyeit kihasználja a 15 ábrán látható.



15. ábra. Egy heterogén rendszer processzorainak ütemezése

A felhasználási terület jelentősen különbözik a webszerverektől: egy nem megfelelően ütemezett feladatot a segédprocesszor valószínűleg le sem tud futtatni. Mivel tudjuk, hány processzortípusunk van, ezért ki tudjuk használni a szegmentált ütemező által nyújtott lehetőségeket: egy fix hozzárende-

lés mindig adott a feladat és annak típusa között. Az azonos feladatot ellátó processzorok között, ahol a hozzárendelés nem egyértelmű, a round-robin ütemezőt használjuk a megfelelő heurisztikák alkalmazása mellett.

7. Következtetések, kitekintés

Bár a feladatütemezők elmélete aktív kutatási területnek számít, az általános probléma különböző megkötésekkel történő szűkítésével fontos teljesítménybeli előnyökre tehetünk szert. A feladatok közötti függőségi kapcsolatok megszorításával két olyan ütemező algoritmust mutattam be, amelyek nem igénylik egy központi adatstruktúra meglétét az ütemezés végrehajtásához. Ezek jó skálázódást mutatnak mind az aktív szálak, a bemenetek és feladattípusok számával, és sokszor elérik vagy meghaladják a hagyományos, zárakat alkalmazó megoldások teljesítményét. Zármentes tulajdonságuk miatt jó alternatívát nyújtanak a magas megbízhatóságot és teljesítményt igénylő valóélet-beli felhasználási területeken. Ezen felül tovább kutatás alapjaként szolgálnak egy zármentes, elosztott load-balancer, vagy heterogén rendszer ütemezőjének fejlesztéséhez is.

8. Hivatkozások

[1] Yu-Kwong Kwok and Ishfaq Ahmad. “Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors”. In: *ACM Comput. Surv.* 31.4 (Dec. 1999), pp. 406–471. ISSN: 0360-0300. DOI: 10.1145/344588.344618. URL: http://doi.acm.org/10.1145/344588.344618.

[2] Wei Zhao and Krithi Ramamritham. “Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints”. In: *J. Syst. Softw.* 7.3 (Sept. 1987), pp. 195–205. ISSN: 0164-1212. DOI: 10.1016/0164-1212(87)90041-0. URL: http://dx.doi.org/10.1016/0164-1212(87)90041-0.

[3] Lincoln Quirk. “Ownership of a queue for practical lock-free scheduling”. Undergraduate Thesis. URL: https://cs.brown.edu/research/pubs/theses/ugrad/2008/quirk.pdf.

[4] James H. Anderson and Srikanth Ramamurthy. “A Framework for Implementing Objects and Scheduling Tasks in Lock-free Real-time Systems”. In: *Proceedings of the 17th IEEE Real-Time Systems Symposium*. RTSS ’96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 94–. ISBN: 0-8186-7689-2. URL: http://dl.acm.org/citation.cfm?id=827268.828962.

[5] Florian Negele et al. “On the Design and Implementation of an Efficient Lock-free Scheduler”. In: 2015.

[6] Ori Shalev and Nir Shavit. “Split-ordered Lists: Lock-free Extensible Hash Tables”. In: *J. ACM* 53.3 (May 2006), pp. 379–405. ISSN: 0004-5411. DOI: 10.1145/1147954.1147958. URL: http://doi.acm.org/10.1145/1147954.1147958.

[7] Dmitry Vyukov. *Non-intrusive MPSC node-based queue*. http://www.1024cores.net/home/lock-free-algorithms/queues/non-intrusive-mpsc-node-based-queue.

[8] Y. Murata et al. "A distributed and cooperative load balancing mechanism for large-scale P2P systems". In: *International Symposium on Applications and the Internet Workshops (SAINTW'06)*. Jan. 2006, 4 pp.-129. DOI: 10.1109/SAINT-W.2006.2.

[9] Harshitha Menon and Laxmikant Kalé. "A Distributed Dynamic Load Balancer for Iterative Applications". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: ACM, 2013, 15:1–15:11. ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503284. URL: http://doi.acm.org/10.1145/2503210.2503284.

[10] Husheng Zhou and Cong Liu. "Task Mapping in Heterogeneous Embedded Systems for Fast Completion Time". In: *Proceedings of the 14th International Conference on Embedded Software*. EMSOFT '14. New Delhi, India: ACM, 2014, 22:1–22:10. ISBN: 978-1-4503-3052-7. DOI: 10.1145/2656045.2656074. URL: http://doi.acm.org/10.1145/2656045.2656074.

[11] Michael Boyer et al. "Load Balancing in a Changing World: Dealing with Heterogeneity and Performance Variability". In: *Proceedings of the ACM International Conference on Computing Frontiers*. CF '13. Ischia, Italy: ACM, 2013, 21:1–21:10. ISBN: 978-1-4503-2053-5. DOI: 10.1145/2482767.2482794. URL: http://doi.acm.org/10.1145/2482767.2482794.