13

Párhuzamosság

Brett L. Schuchert

„Az objektumok a feldolgozást ábrázolják elvont formában, a szálak pedig az ütemezést."
– James O. Coplien[1]

[1] Magánlevelezés.

Tiszta párhuzamos végrehajtású programokat írni nehéz – nagyon nehéz. Sokkal könnyebb olyan kódot készíteni, amely egyetlen szálban hajtódik végre, vagy olyan többszálas kódot, ami a felszínen rendben levőnek látszik, de a mélyben valójában hibás. Az ilyen kód addig megfelelően működik, amíg a rendszer komolyabb terhelés alá nem kerül.

Ebben a fejezetben a párhuzamos programozás szükségességét és az általa támasztott nehézségeket vitatjuk meg. Számos tanácsot adunk arra vonatkozóan, hogy miként küzdhetjük le ezeket a nehézségeket, és írhatunk tiszta párhuzamos kódot. A fejezetet végül a párhuzamos kódok tesztelésével kapcsolatos kérdések vizsgálatával zárjuk.

A tiszta párhuzamosság összetett témakör, amely önmagában megérne egy könyvet. *Ebben* a könyvben mi azt a megközelítést választottuk, hogy itt csak rövid áttekintést adunk, és az „A" függelékben szerepeltetünk egy részletesebb útmutatót. Ha csupán általánosságban érdeklődünk a párhuzamos programozás iránt, akkor ez a fejezet elegendő lesz a számunkra, ha viszont mélyebben meg szeretnénk érteni a párhuzamosság működését, akkor a függelékben szereplő útmutatót is érdemes elolvasnunk.

Mi a célja a párhuzamosságnak?

A párhuzamos programozás a csatolások megszüntetésének stratégiája. Segít, hogy elválasszuk azt, *amit* a program elvégez, attól, hogy *mikor* végzi el. Az egyszálas alkalmazásokban a *mit* és a *mikor* annyira szorosan összefonódik, hogy gyakran a teljes alkalmazás állapotát meg lehet állapítani a veremlenyomatra pillantva. Ha programozóként ilyen rendszeren végzünk hibakeresést, töréspontokat állíthatunk be, és az alapján pontosan *ismerhetjük* a rendszer állapotát, hogy melyik törésponthoz értünk.

A *mit* és a *mikor* szétválasztása drámai módon javíthat egy alkalmazás áteresztőképességén és szerkezetén. Szerkezeti szempontból az alkalmazás ilyenkor sok apró együttműködő számítógépnek tűnik, nem pedig egyetlen nagy ciklusnak. A rendszer ezáltal könnyebben áttekinthető lesz, és a felelősségi körök szétválasztása hatékonyan oldható meg.

Vegyük például a webalkalmazások szabványos „servlet" modelljét. Az ilyen rendszerek egy webes vagy EJB tároló ernyője alatt futnak, amely *részben* gondoskodik a párhuzamosságról helyettünk. A servlet-ek (kiszolgálói kisalkalmazások) végrehajtása aszinkron módon történik – akkor indulnak el, amikor egy webes kérelem érkezik. A servlet-et készítő programozónak nem kell az összes bejövő kérelemmel foglalkoznia. *Elvben* minden kiszolgálói kisalkalmazás végrehajtása a saját kis világában történik, és elválik a többi servlet végrehajtásától.

Természetesen ha a dolog valóban ilyen egyszerű lenne, ezt a fejezetet meg sem kellett volna írnunk. Valójában a webes tárolók által biztosított szétválasztás messze van a töké-

letestől. A kiszolgálói kisalkalmazások programozóinak nagyon körültekintőnek és óvatosnak kell lenniük, hogy a párhuzamos végrehajtású programjaik helyesen működjenek. A servlet modell szerkezeti előnyei ettől függetlenül jelentősek.

A szerkezet azonban nem az egyetlen érv a párhuzamosság mellett. Egyes rendszerek válaszideje és áteresztőképessége olyan korlátokba ütközik, ami kézi kódolású párhuzamos megoldást igényel. Vegyünk például egy egyetlen szálon futó információösszegzőt, amely számos webhelyről gyűjt össze információkat, és ezeket napi összefoglalókká gyúrja össze. Mivel a rendszer egyszálas, a webhelyeket egymás után kell meglátogatnia, és csak akkor kezdhet egy újabb feldolgozásába, ha az előzővel már végzett. A napi futásnak kevesebb mint 24 óra alatt be kell fejeződnie, ahogy azonban újabb és újabb webhelyeket kell belevennie az összefoglalóba, a szükséges idő addig nő, amíg 24 óra már nem elég az adatok összegyűjtéséhez. Az egyetlen szál miatt sok várakozásra van szükség a webes csatolóknál, a bemeneti–kimeneti műveletek befejeződésére várva. A program teljesítményét (sebességét) egy többszálas algoritmussal növelhetjük, amely egyszerre több webhely feldolgozására is képes.

Nézzünk egy másik példát, mondjuk egy olyan rendszert, amely egyszerre egy felhasználót kezel, és felhasználónként csupán egy másodpercre van szüksége. Ennek a rendszernek a válaszképessége kevés felhasználó esetén igen jó, de ahogy a felhasználók száma növekszik, a válaszidő is nőni kezd, márpedig senki nem akar százötven másik felhasználó mögött sorba állni! A rendszer válaszidején úgy javíthatunk, ha a felhasználókat párhuzamosan kezeljük.

Utolsó példaként gondoljunk egy olyan rendszerre, amely nagy méretű adathalmazokat értelmez, de csak az összes feldolgozása után tud teljes megoldást adni. Ha az adathalmazokat más-más számítógépen dolgozzuk fel, párhuzamosan sok-sok adathalmazzal végezhetünk.

Mítoszok és tévhitek

Láttuk tehát, hogy jó okunk lehet a párhuzamosság alkalmazására. Ugyanakkor, ahogy már utaltunk rá, a párhuzamos programozás *nehéz*. Ha nem vagyunk elég körültekintőek, igen csúnya helyzetekbe kerülhetünk. Tekintsük át az alább felsorolt elterjedt mítoszokat és tévhiteket:

- *A párhuzamosság mindig növeli a sebességet.*
  A párhuzamosság *néha* valóban növelheti a sebességet, de csak akkor, ha jelentős az a várakozási idő, amelyet több szál vagy több processzor között el lehet osztani – és egyik elosztási módszer megvalósítása sem könnyű.
- *A felépítésen nem kell változtatni párhuzamos programok írásakor.*
  Valójában egy párhuzamos algoritmus felépítése jelentősen különbözhet egy egyszálas rendszerétől. A *mit* és *mikor* szétválasztása általában hatalmas hatással van a rendszer szerkezetére.

- *A párhuzamossági kérdések ismerete nem fontos, ha webes vagy EJB tárolókkal dolgozunk.*
  Az igazság az, hogy nem árt, ha tudjuk, mit csinál a tároló, és hogyan védekezhetünk a párhuzamos frissítések és a holtpontok veszélyei ellen (ezekről a fejezet későbbi részében beszélünk).

Lássunk még néhány szabályt (ezúttal kiegyensúlyozottabbakat) a párhuzamos programok írásával kapcsolatban:

- *A párhuzamosságnak ára van,* mind a teljesítményt, mint a kód méretét tekintve.
- *A párhuzamosság helyes megvalósítása bonyolult,* még az egyszerű feladatok esetében is.
- *A párhuzamossági hibák általában nem megismételhetők,* ezért gyakran egyszeri hibának[2] tekintve figyelmen kívül hagyják azokat, ahelyett, hogy valódi hiányosságként kezelnék őket.
- *A párhuzamosság gyakran alapvető változtatásokat igényel a felépítési stratégiában.*

Kihívások

Mi teszi a párhuzamos programozást olyan nehézzé? Vegyük az alábbi egyszerű osztályt:

```
public class X {
    private int lastIdUsed;
    public int getNextId() {
        return ++lastIdUsed;
    }
}
```

Tegyük fel, hogy létrehozzuk az `X` egy példányát, a `lastIdUsed` mezőt `42`-re állítjuk, majd a példányt megosztjuk két szál között. Most tegyük fel, hogy mindkét szál meghívja a `getNextId()` tagfüggvényt. Három kimenet lehetséges:

- Az első szál a `43` értéket kapja, a második szál a `44`-et, a `lastIdUsed` értéke pedig `44` lesz.
- Az első szál a `44` értéket kapja, a második szál a `43`-at, a `lastIdUsed` értéke pedig `44` lesz.
- Az első szál a `43` értéket kapja, a második szál szintén a `43`-at, a `lastIdUsed` értéke pedig `43` lesz.

A meglepő harmadik eredményt[3] akkor kapjuk, ha a két szál keresztezi egymást. Ez azért eshet meg, mert a két szál sokféle útvonalon haladhat végig ezen az egyetlen sornyi Java-

[2] Kozmikus sugárzásnak, koszfoltnak stb.

[3] Lásd a *Mélyebbre ásva* című részt az „A" függelékben.

kódon, és egyes végrehajtási útvonalak helytelen eredményt adnak. Hány útvonal lehetséges? Ahhoz, hogy válaszolhassunk erre a kérdésre, meg kell értenünk, hogy mit csinál a futásidejű (Just-In-Time) fordító az előállított bájtkóddal, illetve hogy mit tekint a Java memóriamodell atomi (oszthatatlan) műveletnek.

Ha csak az előállított bájtkódot tekintjük, a gyors válasz az, hogy 12 870 különböző végrehajtási útvonal lehetséges[4], amelyen a két szál lefuthat a `getNextId()` tagfüggvényen belül. Ha a `lastIdUsed` típusát `int`-ről `long`-ra változtatjuk, a lehetséges útvonalak száma 2 704 156-ra nő. Természetesen ezeknek az útvonalaknak a többsége helyes eredményt ad – a gond csak az, hogy *néhány nem.*

A párhuzamos programozás védekezési elvei

A következőkben olyan elveket és eljárásokat mutatunk be, amelyeket követve megvédhetjük a rendszereinket a párhuzamos kódok okozta problémáktól.

Az egyetlen felelősségi kör elve

Az egyetlen felelősségi kör elve (SRP, Single Responsibility Principle) azt mondja ki, hogy egy adott tagfüggvény, osztály vagy összetevő csak egyetlen okból szabad, hogy megváltozzon. A párhuzamos felépítés azonban önmagában elég bonyolult ahhoz, hogy okot adjon a változtatásra, ezért külön kell választanunk a kód többi részétől. Sajnos nagyon gyakori, hogy a párhuzamosság megvalósítási részleteit közvetlenül más műveleti kódokba ágyazzák. A következőket érdemes figyelembe vennünk:

- *A párhuzamossághoz kapcsolódó kódoknak saját fejlesztési, módosítási és hangolási életciklusuk van.*
- *A párhuzamossághoz kapcsolódó kódok saját kihívásokat támasztanak,* amelyek különböznek a nem párhuzamos kódokétól, és gyakran nehezebb is megoldani őket.
- A rosszul megírt párhuzamos végrehajtású kódok olyan sokféle hibára adnak lehetőséget, hogy az őket körülvevő alkalmazáskód terhe nélkül is komoly kihívást jelentenek.

Javaslat: a *párhuzamos végrehajtású kódokat válasszuk el a többi kódtól*[6].

Folyomány: korlátozzuk az adatok hatókörét

Ahogy láttuk, ha két szál egy megosztott objektumnak ugyanazt a mezőjét módosítja, keresztezhetik egymás útját, ami váratlan viselkedéshez vezethet. Erre az egyik megoldást a `synchronized` kulcsszó használata jelenti, amellyel védelem alá helyezhetjük a meg-

[4] Lásd a *Lehetséges végrehajtási útvonalak* című részt az „A" függelékben.

[5] [PPP]

[6] Lásd az *Ügyfél–kiszolgáló példa* című részt az „A" függelékben.

osztott objektumot használó kód egy *kritikus szakaszát*. Lényeges, hogy ezeknek a kritikus szakaszoknak a számát korlátok közé szorítsuk. Minél több helyen módosulhatnak a megosztott adatok, annál nagyobb a valószínűsége a következőknek:

- Elfelejtjük védetté tenni valamelyik helyet – amivel lényegében minden kódot működésképtelenné teszünk, ami a kérdéses megosztott adatot módosítja.
- Kétszeres erőfeszítésre lesz szükség, hogy minden kódrész megfelelő védelmét biztosítsuk (ami megsérti a DRY[7] elvet).
- Nehéz lesz meghatározni azoknak a hibáknak a forrását, amelyek amúgy is nehezen deríthetők fel.

Javaslat: *vegyük komolyan az adatok egységbe zárását, és szigorúan korlátozzuk minden olyan adat elérését, amely megosztva használható.*

Folyomány: használjuk az adatok másolatát

A megosztott adatok elkerülésére jó módszer, ha eleve kerüljük az adatmegosztást. Egyes esetekben lehetséges lemásolni az objektumokat, és csak olvashatóként kezelni azokat, míg máskor lemásolhatjuk őket, több szállal összegyűjthetjük az eredményeket a másolatokból, majd egyetlen szálban egyesíthetjük azokat.

Ha van rá egyszerű mód, hogy elkerüljük az objektumok megosztását, a kapott kód sokkal kevésbé valószínű, hogy gondokat fog okozni. A több objektum létrehozásának persze vannak költségei, ami aggodalomra adhat okot, de megéri kísérletezni, hogy kiderítsük, ez valóban gondot jelent-e. Mindazonáltal, ha az objektummásolatok használata lehetővé teszi, hogy a kódban elkerüljük az összehangolást, a belső zárak kiküszöböléséből eredő megtakarítás valószínűleg ellensúlyozza a létrehozási és szemétgyűjtési többletterhet.

Folyomány: a szálaknak a lehető legfüggetlenebbnek kell lenniük

Célszerű a szálakra osztott kódokat úgy megírni, hogy minden szál a saját világában létezzen, és semmilyen adatot ne osszon meg egyetlen más szállal sem. Minden szál egyetlen ügyfélkérelmet dolgozzon fel, és az összes szükséges adat megosztatlan forrásból származzon és helyi változókban tárolódjon. Ennek köszönhetően a szálak úgy viselkednek majd, mintha ők lennének az egyetlen szál a világon, és nem lenne szükség semmilyen összehangolásra.

A `HttpServlet` osztályból származtatott alosztályok például minden információt a `doGet` és `doPost` tagfüggvényekben átadott paraméterekként kapnak meg, aminek köszönhetően minden `Servlet` úgy viselkedik, mintha önálló gépen futna. Amíg a `Servlet` kódja csak helyi változókat használ, a `Servlet` semmiképpen nem okozhat összehangolási problémákat. Természetesen a `Servlet`-ekre támaszkodó alkalmazások többsége végül belefut majd olyan megosztott erőforrásokba, mint egy adatbázis-kapcsolat.

---

[7] [PRAG]

Javaslat: *próbáljuk olyan független részhalmazokra felosztani az adatokat, amelyeken önálló szálakkal, lehetőleg különböző processzorokon végezhetünk műveleteket.*

Ismerjük meg a könyvtárunkat!

A Java 5-ben a korábbi változatokhoz képest számos helyen javították a párhuzamos fejlesztés támogatásán. Ha a Java 5-ben írunk többszálas kódot, az alábbiakat érdemes szem előtt tartanunk:

- Használjuk a készen kapott szálbiztos gyűjteményeket.
- Az egymáshoz nem kapcsolódó feladatok végrehajtására használjuk az `Executor` (végrehajtó) keretrendszert.
- Alkalmazzunk nem blokkoló megoldásokat, amikor csak lehetséges.
- Sok könyvtári osztály nem szálbiztos.

Szálbiztos gyűjtemények

Doug Lea akkor írta meg *Concurrent Programming in Java* című korszakalkotó könyvét[8], amikor a Java még újdonságnak számított. A könyv megírásával egyidejűleg számos szálbiztos gyűjteményt is kifejlesztett, amelyek később a JDK részévé váltak, és a `java.util.concurrent` csomagban kaptak helyet. Az ebben a csomagban található gyűjtemények biztonságosan alkalmazhatók és jól teljesítenek a többszálas programokban. Valójában a `ConcurrentHashMap` megvalósítás szinte minden helyzetben jobb teljesítményt nyújt, mint a `HashMap`. Ezenkívül lehetővé teszi az egyidejű olvasást és írást, és tagfüggvényekkel támogat olyan szokványos összetett műveleteket, amelyek egyébként nem lennének szálbiztosak. Ha a telepítési környezet a Java 5, mindig a `ConcurrentHashMap`-ből induljunk ki.

A párhuzamos felépítést további osztályok is támogatják. Lássunk néhány példát:

`ReentrantLock` (újrahívható zár)	Zár, amely megszerezhető egy tagfüggvényben, és feloldható egy másikban.
`Semaphore` (szemafor)	A klasszikus szemafor, vagyis a számlálóval rendelkező zár megvalósítása.
`CountDownLatch` (visszaszámláló retesz)	Zár, amely megvárja, amíg egy adott számú esemény bekövetkezik, mielőtt továbbengedné a rá várakozó szálakat. Ez lehetővé teszi, hogy az összes szál jó eséllyel nagyjából ugyanakkor induljon el.

[8] [Lea99]

Javaslat: *tekintsük át a rendelkezésünkre álló osztályokat – a Java esetében ismerkedjünk meg a java.util.concurrent, a java.util.concurrent.atomic és a java.util.concurrent.locks osztályokkal.*

Ismerjük meg a végrehajtási modelleket!

Egy párhuzamos végrehajtású alkalmazásban a viselkedés többféleképpen is felosztható. Ahhoz, hogy ezeket a módszereket megérthessük, tisztában kell lennünk néhány alapfogalommal:

Korlátos erőforrás	Rögzített méretű vagy számú erőforrás egy párhuzamos környezetben. Ilyen erőforrások lehetnek például az adatbázis-kapcsolatok, illetve a rögzített méretű olvasó/író átmeneti tárak.
Kölcsönös kizárás	Egyszerre csak egyetlen szál férhet hozzá a megosztott adathoz vagy erőforráshoz.
Éheztetés	Egy szál vagy szálak egy csoportja kivételesen hosszú ideig vagy soha nem haladhat tovább. Például ha mindig a gyorsan futó szálakat engedjük át először, azzal a hosszabb futási idejű szálakat éhezésre kárhoztathatjuk, amennyiben a gyorsan futó szálak nem fogynak el.
Holtpont	Két vagy több szál egymás befejeződésére vár. Mindegyik szálnak van egy olyan erőforrása, amelyre a másik szálnak van szüksége, és egyik sem tudja befejezni a feladatát, amíg meg nem szerezte a másik erőforrását.
Élőzár	Egymás után érkező szálak a saját feladatukat próbálják elvégezni, de a másik szál „útban van". A szálak megpróbálnak előrehaladni, de erre kivételesen hosszú ideig képtelenek – vagy soha nem tudnak továbbmenni.

Most, hogy tisztáztuk ezeket a fogalmakat, rátérhetünk a párhuzamos programozásban alkalmazott különféle végrehajtási modellek tárgyalására.

Termelők és fogyasztók[9]

Egy vagy több termelő szál feladatokat határoz meg, és egy átmeneti tárba vagy várakozási sorba helyezi azokat, ahonnan egy vagy több fogyasztó szál kiemeli és végrehajtja őket. A termelők és a fogyasztók közötti várakozási sor egy *korlátos erőforrás*. Ez azt jelenti, hogy a termelőknek meg kell várniuk, amíg a várakozási sorban hely szabadul fel, mielőtt írnának bele, a fogyasztóknak pedig meg kell várniuk, hogy legyen valami a sorban, amit elfogyaszthatnak. A termelők és a fogyasztók összehangolása a várakozási soron keresztül abból áll, hogy a termelők és a fogyasztók jelzéseket adnak egymásnak. A termelők bele-

[9] http://en.wikipedia.org/wiki/Producer-consumer

írnak valamit a várakozási sorba, és jelzik, hogy a sor már nem üres, a fogyasztók pedig kiolvassák a feladatot a várakozási sorból, és jelzést adnak, hogy a sor már nincs tele. Mind a termelők, mind a fogyasztók várakozásra kényszerülhetnek, amíg értesítést nem kapnak, hogy folytathatják a működésüket.

Írók és olvasók[10]

Amikor egy olyan megosztott erőforrással rendelkezünk, amely elsősorban olvasó szálaknak szolgál információforrásul, de amelyet időnként író szálak is frissítenek, az áteresztőképesség lényeges. Az áteresztőképesség növelése éhezést és az elavult információk felgyülemlését idézheti elő. A frissítés engedélyezése hatással lehet az áteresztőképességre, az olvasók összehangolása pedig – annak érdekében, hogy ne olvassanak, miközben egy író frissít, és viszont – kényes egyensúlyozási művelet. Az író szálak hajlamosak sok olvasót hosszú ideig blokkolni, ami gondokat okozhat az áteresztőképesség terén.

A kihívást az olvasók és írók igényeinek olyan kiegyensúlyozása jelenti, ami biztosítja a helyes működést és a megfelelő áteresztőképességet, és kiküszöböli az éhezést. Egy egyszerű megoldás lehet, ha az írókat addig várakoztatjuk, amíg minden olvasó nem végzett, és csak ez után engedjük meg nekik a frissítés végrehajtását. Ha azonban folyamatosan vannak olvasó szálaink, az írók éhezésre lesznek kárhoztatva. Másrészről, ha gyakran történik írás, és az író szálaknak adunk elsőbbséget, annak az áteresztőképesség látja kárát. A feladatot az egyensúly megtalálása és a párhuzamos frissítéssel kapcsolatos problémák kiküszöbölése jelenti.

Étkező filozófusok[11]

Képzeljünk el néhány filozófust, akik egy kerek asztal körül ülnek. Mindegyikőjük baljára teszünk egy villát, és az asztal közepén elhelyezünk egy nagy tál spagettit. A filozófusok elmélkedéssel töltik az idejüket, amíg meg nem éheznek. Aki éhes, megfogja a kézre eső villákat, és nekilát az evésnek. Mindenki csak akkor ehet, ha két villa van a kezében. Ha akár a balján, akár a jobbján ülő filozófus már elvette az egyik villát, amelyre szüksége van, az illetőnek meg kell várnia, amíg a másik befejezi az evést, és visszateszi a villát. Aki már evett, mindkét villát visszateszi az asztalra, és megvárja, amíg újból éhes nem lesz.

Ha a filozófusok helyére szálakat, a villák helyére pedig erőforrásokat helyettesítünk be, egy olyan problémát kapunk, amelyhez hasonlóval sok üzleti alkalmazásban találkozhatunk, amelyben folyamatok versengenek az erőforrásokért. Az ilyen módon versengő rendszerekben – hacsak nincsenek gondosan megtervezve – holtpontok és élőzárak léphetnek fel, az áteresztőképesség és a hatékonyság pedig romlik.

---

[10] http://en.wikipedia.org/wiki/Readers-writers_problem

[11] http://en.wikipedia.org/wiki/Dining_philosophers_problem

A legtöbb párhuzamossági probléma, amivel valószínűleg találkozni fogunk, az említett három probléma valamilyen variációja. Tanulmányozzuk ezeket az algoritmusokat, és írjunk a felhasználásukkal saját megoldásokat, hogy amikor párhuzamossági problémákkal szembesülünk, jobban fel legyünk készülve az elhárításukra.

Javaslat: *tanuljuk meg ezeket az alapvető algoritmusokat, és próbáljuk megérteni a rájuk adott megoldásokat.*

Óvakodjunk a függőségektől az összehangolt tagfüggvények között!

Az összehangolt tagfüggvények közötti függőségek nehezen felderíthető hibákat okozhatnak a párhuzamos végrehajtású kódokban. A Java nyelvben a `synchronized` (összehangolt) kulcsszóval védhetjük az egyes tagfüggvényeket, ugyanakkor ha egy megosztott osztályban egynél több összehangolt tagfüggvény tevékenykedik, lehet, hogy a rendszert helytelenül írtuk meg.[12]

Javaslat: *ne használjunk egynél több tagfüggvényt egy megosztott objektumon.*

Mindazonáltal időnként előfordulhat, hogy egynél több tagfüggvényt kell használnunk egy megosztott objektumon. Ha ez a helyzet, háromféleképpen tehetjük a kódot helyessé:

- *Ügyfél alapú zárolással* – Az ügyféllel zároltatjuk a kiszolgálót, mielőtt meghívnánk az első tagfüggvényt, és gondoskodunk róla, hogy a zárolás időtartamába beleessen az utolsó tagfüggvényt meghívó kód.
- *Kiszolgáló alapú zárolással* – A kiszolgálón belül létrehozunk egy tagfüggvényt, amely zárolja a kiszolgálót, meghívja az összes tagfüggvényt, majd feloldja a zárat, és az ügyféllel ezt az új tagfüggvényt hívatjuk meg.
- *Közvetítő kiszolgálóval* – Létrehozunk egy közvetítőt, amely végrehajtja a zárolást. Ez a kiszolgáló alapú zárolás egyik formája, amelyet akkor alkalmazhatunk, ha az eredeti kiszolgáló nem módosítható.

Az összehangolt szakaszok legyenek kicsik!

A `synchronized` kulcsszó egy zárat hoz létre. Minden kódrészen, amelyet ugyanaz a zár véd, garantáltan csak egyetlen szál végrehajtása haladhat át egyidejűleg. A zárak költségesek, mert késleltetést és többletterhelést okoznak, ezért a kódot nem szabad `synchronized` utasításokkal teleszórni. Másrészről viszont a kritikus szakaszokat[13] védeni kell. A kódunkat tehát úgy kell felépítenünk, hogy a lehető legkevesebb kritikus szakaszt tartalmazza.

---

[12] Lásd *A tagfüggvények közötti függőségek működésképtelenné tehetik a párhuzamos kódokat* című részt az „A" függelékben.

[13] A kritikus szakasz (létfontosságú szakasz) olyan kódrész, amelyet meg kell védeni az egyidejű használattól ahhoz, hogy a program működése helyes legyen.

Egyes naiv programozók ezt úgy próbálják elérni, hogy jelentősen megnövelik a kritikus szakaszok hosszát. Az összehangolás kiterjesztése a minimális kritikus szakaszon túlra azonban növeli a versengés mértékét, és rontja a teljesítményt.[14]

Javaslat: *az összehangolt szakaszokat igyekezzünk a lehető legkisebb méretűvé tenni.*

Helyes leállítási kódot írni nehéz

Olyan rendszert készíteni, amelynek folyamatosan üzemelnie kell, más, mint olyasmit írni, ami működik egy darabig, aztán elegánsan leáll. Az elegáns leállás helyes működésének elérése nem mindig egyszerű. A leggyakrabban fellépő problémák között találjuk a holtpontokat[15], amikor a szálak egy olyan jelzésre várnak, ami soha érkezik meg.

Képzeljünk el például egy olyan rendszert, amelyben egy szülő szál számos gyermekszálat indít, majd megvárja, amíg mindegyik befejezi a munkáját, mielőtt felszabadítaná az erőforrásait, és leállítaná magát. Mi történik, ha az egyik gyermekszál holtponthoz érkezik? A szülő így örökké várhat, és a rendszer soha nem fog leállni. Vagy vegyünk egy hasonló rendszert, amely *utasítást kap* a leállásra. A szülő felszólítja az összes gyermekét, hogy hagyják abba a munkájukat, és álljanak le. De mi történik, ha a gyermekek közül kettő temelő–fogyasztó párként működik? Tegyük fel, hogy a termelő megkapja a jelzést a szülőtől, és gyorsan leállítja magát. Előfordulhat, hogy a fogyasztó éppen üzenetet várt a termelőtől, és ezért olyan blokkolt állapotban várakozik, amelyben nem érhet el hozzá a leállásra felszólító jelzés. A fogyasztó tehát a termelőre várva elakad, és soha nem tudja befejezni a futását, ami a szülőt is megakadályozza a leállásban.

Az ilyen helyzetek egyáltalán nem ritkák, ezért ha olyan párhuzamos kódot kell írnunk, amelynek része az elegáns leállítás, számítsunk rá, hogy sok időt kell majd töltenünk azzal, hogy biztosítsuk a megfelelő leállítást.

Javaslat: *már az elején gondoljunk a leállításra, és minél hamarabb tegyük azt működőképessé, mert tovább fog tartani, mint gondolnánk, és vegyük igénybe a már létező algoritmusokat, mert a feladat valószínűleg nehezebb, mint hinnénk.*

A többszálas kódok tesztelése

Egy kód helyességének bizonyítása nem egyszerű, mert a tesztelés nem garantálja a helyességet – viszont a jó tesztek jelentősen csökkenthetik a kockázatot. Nos, ez az egyszálas programok esetében teljesen igaz, de amint kettő vagy több szál használja ugyanazt a kódot, és megosztott adatokkal dolgozik, a dolog már lényegesen bonyolultabb.

---

[14] Lásd *Az áteresztőképesség növelése* című részt az „A" függelékben.

[15] Lásd a *Holtpontok* című részt az „A" függelékben.

Javaslat: *írjunk olyan teszteket, amelyek képesek lehetnek felfedni a problémákat, és futtassuk őket gyakran, más-más program- és rendszerbeállításokkal, illetve terheléssel. Ha a tesztek hibát jeleznek, keressük meg az okát. Ne hagyjunk figyelmen kívül egy hibát csak azért, mert egy későbbi futtatásnál a tesztek esetleg gond nélkül továbbhaladnak.*

Ha a fenti javaslat túl sok szempont figyelembe vételét igényelné, íme néhány „finomabb felbontású" tanács:

- A nem ismétlődő hibákat tekintsük lehetséges szálkezelési problémáknak.
- Először a szálak nélküli kódot tegyük működőképessé.
- Tegyük a többszálas kódokat tetszőlegesen beszúrhatóvá.
- Tegyük a többszálas kódokat hangolhatóvá!
- Több szálat futtassunk, mint processzort.
- Futtassuk a programot különböző rendszereken.
- Rendezzük a kódot a hibák kikényszerítéséhez.

A nem ismétlődő hibákat tekintsük lehetséges szálkezelési problémáknak!

A többszálas kódokban olyan dolgok is hibásan működhetnek, amelyek „egyszerűen nem romolhatnak el".[*] A legtöbb fejlesztő (ennek a kötetnek a szerzőit is beleértve) nem érzi, hogyan kapcsolódnak a szálak a többi kódhoz. A szálakra osztott kódban megbúvó programhibák tünetei ezer vagy egymilió végrehajtás alatt talán csak egyszer jelentkeznek, ezért a rendszer viselkedésének megismétlése reménytelen munka lehet. Ez gyakran vezet ahhoz, hogy a fejlesztők a hiba okaként a kozmikus sugárzást, a hardver csuklását vagy valamilyen más, „egyszeri" eseményt jelölnek meg, hiszen a nem ismétlődő hibákat a legjobb úgy tekinteni, mintha nem is léteznének. Mindazonáltal, minél tovább nem veszünk tudomást ezekről az „egyszeri" hibákról, annál több kód épül majd egy olyan megoldásra, ami lehet, hogy hibás.

Javaslat: *ne tekintsük a rendszerhibákat egyszeri, figyelmen kívül hagyható eseményeknek.*

Először a szálak nélküli kódot tegyük működőképessé!

Ez nyilvánvalónak tűnhet, de nem árt, ha még egyszer az eszünkbe véssük: gondoskodjunk róla, hogy a kód a szálakon belüli használaton kívül is működőképes legyen. Általánosságban ez azt jelenti, hogy POJO-kat hozunk létre, amelyeket a szálaink meghívnak. A POJO-k nem tudnak a szálakról, ezért a szálas környezeten kívül tesztelhetők. Minél nagyobb részét tudjuk a rendszerünknek ilyen POJO-kban elhelyezni, annál jobb.

Javaslat: *ne próbáljuk egyszerre felderíteni az egyszálas és a többszálas hibákat, és győződjünk meg róla, hogy a kódunk a szálakon kívül is működik.*

Tegyük a többszálas kódokat tetszőlegesen beszúrhatóvá!

A párhuzamosságot támogató kódot úgy írjuk meg, hogy többféle felépítés esetén is futtatható legyen:

- egyszálas, többszálas és végrehajtás közben változó környezetben;
- a szálakra osztott kód valódi és tesztelési célú helyettes objektumokkal is működjön;
- gyorsan, lassan és változó sebességgel futó tesztelési célú helyettes objektumokkal is végrehajtható legyen;
- a teszteket úgy állítsuk be, hogy többször is megismételhetők legyenek.

Javaslat: *tegyük a szálakra épülő kódokat különösen sokoldalúan beszúrhatóvá, hogy a legkülönfélébb felépítések esetén is futtathatók legyenek.*

Tegyük a többszálas kódokat hangolhatóvá!

A szálak megfelelő egyensúlyának megtalálása általában próba-szerencse dolga. Már a fejlesztés elején találjunk módot rá, hogy többféle felépítés esetén is megmérhessük a rendszerünk teljesítményét, és a szálak számát tegyük egyszerűen módosíthatóvá. Azt is érdemes lehet megengedni, hogy a szálak száma a rendszer futása közben hangolható legyen, és lehetővé tenni az önhangolást az áteresztőképesség és a rendszer kihasználtsága alapján.

Több szálat futtassunk, mint processzort!

Akkor történnek érdekes dolgok, amikor a rendszer vált a feladatok között. A feladatváltások kikényszerítéséhez futtassunk több szálat, mint processzort vagy processzormagot. Minél többször történik feladatváltás, annál valószínűbb, hogy olyan kóddal találkozunk, amelyből hiányzik egy kritikus szakasz, vagy holtpontot okoz.

Futtassuk a programot különböző rendszereken!

2007 közepén kurzust tartottunk a párhuzamos programozásról. Az órák anyagát elsősorban OS X rendszerre dolgoztuk ki, és egy virtuális gépen keresztül Windows XP-n mutattuk be. A hibakörülmények szemléltetésére írt tesztek XP környezetben ritkábban vallottak kudarcot, mint OS X-en.

A tesztelt kódról minden esetben tudtuk, hogy helytelen. Ami történt, az csak megerősítette azt a tényt, hogy a különböző operációs rendszerek más-más szálkezelési rendet alkalmaznak, ami hatással van a kód végrehajtására. A többszálas kódok különbözőképpen viselkednek az eltérő környezetekben[16], ezért a teszteket minden lehetséges telepítési környezetben futtatnunk kell.

Javaslat: a *többszálas kódokat gyakran és minél korábban futtassuk le minden lehetséges célrendszeren.*

[16] Az Olvasó tudta, hogy a Java szálkezelési modellje nem garantál közbeavatkozó szálvégrehajtást? A mai operációs rendszerek azonban támogatják, ezért a közbeavatkozó szálvégrehajtást „ingyen kapjuk". Ennek ellenére a JVM nem ad rá garanciát.

Rendezzük a kódot a hibák kikényszerítéséhez!

Nem szokatlan, hogy a párhuzamos kódokban levő hibák rejtve maradnak. Az egyszerű tesztek gyakran nem fedik fel őket, sőt sokszor normál feldolgozásnál nem is jelentkeznek, csak néhány óránként, naponként vagy akár hetenként. Annak, hogy a szálkezelési hibák ritkán és rendszertelenül jelentkeznek, és nehéz megismételni őket, az az oka, hogy a sérülékeny szakaszokon keresztül vezető sok ezer lehetséges végrehajtási út közül ténylegesen csak nagyon kevés okoz hibát. Annak a valószínűsége tehát, hogy a program hibás útvonalat választ, meglepően csekély, ami a hibák észlelését és elhárítását nagyon megnehezíti.

Hogyan növelhetjük az ilyen ritkán jelentkező hibák elfogásának az esélyét? Úgy, hogy olyan tagfüggvényhívásokkal rendezzük a kódot, mint az `Object.wait()`, az `Object.sleep()`, az `Object.yield()` vagy az `Object.priority()`, amelyekkel különböző rendezés szerinti futásra kényszeríthetjük a programot. Ezek a tagfüggvények mind hatással vannak a végrehajtási sorrendre, ezért növelik a hibák észlelésének esélyét. Minél hamarabb és minél gyakrabban okoz hibát egy helytelen kódrész, annál könnyebben javíthatjuk ki a hibát.

A kód rendezésére két módszert alkalmazhatunk:

- kézi kódolású rendezést, és
- automatizált rendezést.

Kézi kódolású rendezés

A `wait()`, `sleep()`, `yield()` és `priority()` hívásokat beszúrhatjuk a kódba saját kezűleg is, és lehet, hogy éppen ezt kell tennünk, amikor egy különösen tüskés kódrészt tesztelünk. Íme egy példa:

```
public synchronized String nextUrlOrNull() {
      if(hasNext()) {
         String url = urlGenerator.next();
         Thread.yield(); // ezt a teszteléshez szúrtuk be.
         updateHasNext();
         return url;
      }
      return null;
}
```

A beszúrt `yield()` hívás megváltoztatja a kód végrehajtási útvonalát, és valószínűleg hibára készteti a kódot egy olyan helyen, ahol korábban nem vallott kudarcot. Amennyiben a kód valóban hibázik, annak nem az az oka, hogy hozzáadtuk a `yield()` hívást[17], hanem inkább az, hogy a kód eleve hibás volt – mi csak felszínre hoztuk a hibát.

[17] Ez nem feltétlenül igaz. Mivel a JVM nem garantál közbeavatkozó szálvégrehajtást, egy adott algoritmus mindig működőképes lehet egy olyan operációs rendszeren, amely nem közbeavatkozó módon kezeli a szálakat. Ennek a fordítottja is lehetséges, de más okokból kifolyólag.

A fenti megoldással azonban több probléma is van:

- Saját kezűleg kell megkeresnünk a megfelelő helyeket, ahová beszúrhatjuk a hívást.
- Nem tudhatjuk, hol kell elhelyeznünk a hívást, és milyen hívást kell alkalmaznunk.
- Ha egy ilyen kódot bennehagyunk az üzemi környezetben működő programban, az szükségtelenül lelassítja a kódot.
- A megoldás kissé olyan, mint ágyúval lőni verébre. A hibákat vagy megtaláljuk, vagy nem – az esélyek azonban nem a mi oldalunkon állnak.

Amire valójában szükségünk van, az egy olyan módszer, amellyel a fenti megoldást tesztelés közben alkalmazzuk, de éles környezetben nem. Ezenkívül azt is meg kell oldanunk, hogy az egyes futtatások között könnyen megváltoztathassuk a beállításokat, ami összességében növeli a hibák megtalálásának az esélyét.

Világos, hogy ha a rendszerünket olyan POJO-kra osztjuk fel, amelyek semmit sem tudnak a szálakról és a szálakat vezérlő osztályokról, könnyebben megtalálhatjuk azokat a helyeket, ahol rendezhetjük a kódot. Ezen kívül így különféle tesztrostákat hozhatunk létre, amelyek különböző `sleep`, `yield` stb. hívásokkal hívják meg a POJO-kat.

Automatizált rendezés

Ha programozottan szeretnénk rendezni a kódot, olyan eszközöket használhatunk, mint az Aspect-Oriented Framework (aspektus- vagy szempontközpontú keretrendszer), a CGLIB vagy az ASM. Használhatunk például egy egyetlen tagfüggvénnyel rendelkező osztályt:

```
public class ThreadJigglePoint {
   public static void jiggle() {
   }
}
```

Ezt az osztályt a kódunk különböző részeiről hívhatjuk meg:

```
public synchronized String nextUrlOrNull() {
      if(hasNext()) {
           ThreadJigglePoint.jiggle();
          String url = urlGenerator.next();
           ThreadJigglePoint.jiggle();
          updateHasNext();
           ThreadJigglePoint.jiggle();
           return url;
      }
      return null;
}
```

Így csupán egy egyszerű aspektust kell használnunk, amely véletlenszerűen választ a semmittevés, az alvás és a szálindítás között.

Vagy, tegyük fel, hogy a `ThreadJigglePoint` osztálynak két megvalósítása van. Az első úgy valósítja meg a `jiggle`-t (a rostát), hogy ne csináljon semmit – ezt használjuk az éles üzemben. A második megvalósítás egy véletlenszámot állít elő, amelynek alapján a program választ az alvás, a szálindítás vagy az egyszerű átesés között. Ha a tesztjeinket ezerszer lefuttatjuk véletlenszerű rostálással, felszínre hozhatunk néhány hibát, ha pedig a tesztek mindig sikerrel járnak, akkor legalább azt mondhatjuk, hogy mi megtettünk minden tőlünk telhetőt. Bár kissé leegyszerűsített megoldás, ésszerű alternatíva lehet egy kifinomultabb eszköz hiányában. Létezik egy ConTest[18] nevű eszköz (az IBM fejlesztése), amely valami hasonlót csinál, csak éppen sokkal furfangosabban.

A lényeg az, hogy a kódot úgy kell rázogatnunk a rostában, hogy a szálak minden esetben más-más sorrendben fussanak le. A jól megírt tesztek és a rostálás kombinációja drámai módon megnövelheti a hibák megtalálásának az esélyét.

Javaslat: *alkalmazzunk rostálási módszereket a hibák kifürkészéséhez.*

Összefoglalás

A párhuzamos kódok helyességét nem egyszerű biztosítani. Még egy eredetileg könnyen követhető kód is rémálommá válhat, ha több szálat és megosztott adatokat keverünk hozzá. Ha párhuzamos végrehajtású kódot kell írnunk, szigorúan ragaszkodnunk kell a tiszta kódhoz, különben rejtélyes és ritkán felbukkanó hibákkal fogunk szembenézni.

Az első és egyben a legfontosabb dolog, hogy kövessük az egyetlen felelősségi kör elvét. Bontsuk fel a rendszerünket POJO-kra, amelyek elválasztják a szálakra osztott kódot a szálakat nem használó kódrészektől. Gondoskodjunk róla, hogy amikor a többszálas kódot teszteljük, csak azt teszteljük, és semmi mást. Ez egyben azt is jelenti, hogy a többszálas kódrésznek kicsinek és fókuszáltnak kell lennie.

Legyünk tisztában a párhuzamossági problémák lehetséges forrásaival: azzal, amikor több szál megosztott adatokon végez műveleteket, vagy amikor közös erőforrásgyűjtőt használunk. Az olyan határesetek, mint a tiszta leállítás vagy egy ciklus ismétléseinek befejezése, különös odafigyelést igényelnek.

Ismerjük meg a programkönyvtárunkat és az alapvető algoritmusokat. Értsük, hogyan támogatják a könyvtár által nyújtott szolgáltatások a problémák megoldását az alapvető algoritmusokhoz hasonlóan.

Tanuljuk meg, hogy találhatjuk meg azokat a kódrészeket, amelyeket zárolni kell, és zároljuk őket. Ne zároljunk olyan kódrészeket, amelyeket nem kell zár alá helyezni. Kerüljük a zárolt szakaszok meghívását egy másik zárolt szakaszból. Ehhez szükséges, hogy

[18] `http://www.alphaworks.ibm.com/tech/contest`

pontosan tudjuk, mi megosztott, és mi nem. A megosztott objektumok számát és a megosztás hatókörét csökkentsük a lehető legkisebbre. Módosítsuk a megosztott adatokat tartalmazó objektumok felépítését úgy, hogy az objektumok igazodjanak az ügyfelekhez, ahelyett, hogy az ügyfeleket kényszerítenék a megosztott állapotok kezelésére.

Problémák biztosan fognak jelentkezni. Azokat a hibákat, amelyek nem bukkanak fel hamar, gyakran egyszeri jelenségeknek tekintve félresöprik. Az ilyen egyszeri hibák jellemzően komolyabb terhelés esetén, illetve látszólag véletlenszerűen bukkanak fel, ezért képesnek kell lennünk arra, hogy a többszálas kódrészeket sokféle beállítással, több rendszeren is ismétlődően és folyamatosan futtassuk. A tesztelhetőség, ami természetesen következik a tesztvezérelt fejlesztés három törvényének betartásából, bizonyos fokig tetszőleges beszúrhatóságot is jelent, ami biztosítja az ahhoz szükséges támogatást, hogy a kódot sokféle felépítésű rendszeren futtathassuk.

Jelentősen növelhetjük a hibás kódok megtalálásának az esélyét, ha időt szánunk a kód rendezésére. Ezt megtehetjük saját kezűleg vagy valamilyen automatizált eszközzel. Ebbe már a fejlesztés elején érdemes energiát fektetnünk. A többszálas kódrészeket a lehető leghosszabb ideig futtassuk, mielőtt az éles üzemi környezetben munkába állítanánk őket. Ha a tiszta megközelítést választjuk, drámai módon növelhetjük az esélyünket arra, hogy helyes kódot készítsünk.

Irodalomjegyzék

[Lea99]: Doug Lea: *Concurrent Programming in Java: Design Principles and Patterns*, 2. kiadás, Prentice Hall, 1999.

[PPP]: Robert C. Martin: *Agile Software Development: Principles, Patterns and Practices*, Prentice Hall, 2002.

[PRAG]: Andrew Hunt és Dave Thomas: *The Pragmatic Programmer*, Addison-Wesley, 2000.