Lock-szegény adatszerkezetek tervezése, elemzése és implementálása a C++11 memóriamodelljében

Szakdolgozat

Alkalmazott matematikus mesterszak - Számítástudomány szakirány

Készítette:

Danyluk Tamás

Témavezető:

Dr. Porkoláb Zoltán, egyetemi docens

Programozási Nyelvek és Fordítóprogramok Tanszék

Eötvös Loránd Tudományegyetem

Természettudományi Kar

2016

Köszönetnyilvánítás

Ezúton szeretném megköszönni Dr. Porkoláb Zoltánnak, hogy témavezetőként iránymutatásával és hasznos javaslataival nélkülözhetetlen segítséget nyújtott a dolgozat megírásában. Köszönet illeti a barátnőmet, a családomat és mindenki mást is, aki támogatásával hozzájárult a dolgozat létrejöttéhez.

Danyluk Tamás

Tartalomjegyzék

Az szakdolgozatban említett, általam írt programok forráskódja elérhető a következő git tárolóból: `https://gitlab.com/tomi92/lockfree`

1. fejezet

Bevezetés

1.1. Motiváció

A dolgozat áttekinti a lock-szegény adatszerkezetek C++11 nyelven való létrehozásának minden állomását a matematikai alapozástól a számítógépes validálásig.

E dolgozat megírását az alacsony szintű szinkronizálásról szóló, kellően átfogó, szabadon elérhető mű hiánya motiválta.

Az interneten található leírások sokszor átugorják a teljes megértéshez elengedhetetlen matematikai relációk ismertetését, és intuitív de matematikailag kevésbé precíz leírásokat adnak az egyes szinkronizálási primitívekről.

A formális matematikai megfogalmazás látszólagos hiánya nehézzé teszi a lock-free adatszerkezetekkel kapcsolatos gondolkodást. Szerencsére kiderül, hogy a C++11 szabványban nagy pontossággal definiálva vannak a kellő relációk, és egy kis összefoglalással és alapállítások belátásával kezelhető formába hozhatjuk őket. A dolgozat igyekszik a megfelelő arányban egymás mellé helyezni a formális és a közérthető megfogalmazású állításokat és emellett az implementáció kérdéseit is részletezi.

A processzor szintű / hardveres megvalósítás részleteire nem tértem ki, ezek megtalálhatóak például a [7] cikkben.

1.2. Eredmények

Összegyűjtöttem és összefoglaltam a témához szükséges alapfogalmakat és definíciókat. Formálisan összefoglaltam a C++11 memóriamodelljét. Ismertetem a memóriamodell történeti vonatkozásainak, kialakulása okának. Létrehoztam egy lock-free sor adatszerkezetet és a helyességéről meggyőződtem mind matematikai, mind kísérleti eszközökkel. Táblázatos formában összehasonlítottam a különböző szinkronizálási módokat használó sorok futásidejét. Átültettem C++11 nyelvre egy olvasás közben módosítható (RCU) hasítótáblát, majd mérésekkel alátámasztottam, hogy ez egyenletesebb és gyorsabb olvasási időket garantál, mint egy egyszerű vödrönként lockolt hasítótábla.

1.3. Dolgozat szerkezete

A 2. fejezet megindokolja miért is van szükség a használt technikai és matematikai eszközökre. Alapvető cikkekből idézve lerántja a leplet a köztudatban hibásan elterjedt programozási mintákról, például tételesen megcáfolja, hogy a volatile használható lenne szinkronizálásra.

A 3. fejezet bevezeti az olvasót a C++11 memóriamodelljének rejtelmeibe, matematikai pontossággal foglalja össze a szálak közötti szinkronizálást biztosító egyes relációkat. Az összefoglalás alapján belát egyszerű tételeket, amik segítenek az adatszerkezetek tervezésekor.

A 4. fejezet definiálja a lock-free adatszerkezetekhez kapcsolódó alapvető fogalmakat, majd bemutat egy egyszerű lock-free sor adatszerkezetet. Az adatszerkezet alapján rámutat a matematikai helyességbizonyítás szükségességére, és ismertet is rá egy példát. Bemutatja a számítógépes validálás mikéntjét a Relacy versenyhelyzet kereső segítségével. Táblázatos formában összehasonlítja a különböző sor adatszerkezetek gyakorlati futási időit.

Az 5. és a 6. fejezet kitekintést nyújt az olvasás közben módosítható (RCU) adatszerkezetekre, amik még nem teljesen honosodtak meg a C++11 nyelven. Áttekinti a portolás nehézségeit egy RCU megvalósítás C++11-re költöztetésével, majd egy nagy-teljesítményű konkurrens hasítótábla C++11-es implementációjával. Táblázatos formában összehasonlítja két különböző hasítótábla adatszerkezet gyakorlati futási időit.

Az A függelék további információkat közöl a Relacy race detector használatáról, míg a B függelék tartalmaz egy rövid leírást a többszálú programozáshoz használatos tesztkörnyezet beállításáról, és amellett érvel, hogy a legkedvezőbb környezetet a szinte mindenkinél megtalálható okostelefon nyújtja.

2. fejezet

A C++03 memória modell hiányosságai

Ez a fejezet arról szól, hogy miért volt szükség a memória modell definiálására a C++11 szabványban. A C++03 szabvány absztrakt számítógép modellje alapvetően egyszálú és emiatt nagyon nehéz helyes és kereszt-platformos többszálú programokat írni benne. A hivatkozott [10] cikk áttekinti, hogy miért nem elég ha egy függvénykönyvtár definiálja a több-szálú végrehajtás szabályait. A [2] cikk pedig egy olyan megoldást ír le, ami a Singleton pattern komoly gyorsításával kecsegtet, de a C++11 előtt lényegében nem lehet platform-specifikus kód nélkül megvalósítani.

2.1. A Double-Checked Locking Pattern problémái C++03-ban

E szakasz alapját a hivatkozott [2] cikk képezi. A Singleton pattern az egyik legelterjedtebb design pattern, annak ellenére, hogy sokak szerint káros a használata. Singleton típusból legfeljebb egy objektum-példány létezhet, és azt le is lehet kérni bármely programrészből, hasonlóan egy globális változóhoz. Sokszor az abstract factory-hoz hasonlóan működik, azaz kívülről nem látszik az objektum pontos típusa.

```
//.h file
class Singleton {
public:
    static Singleton* instance();
    // ...
private:
    static Singleton* pInstance;
}

//.cpp file
Singleton* Singleton::pInstance = 0;
Singleton* Singleton::instance() {
    if(pInstance == 0)
        pInstance = new Singleton;
    return pInstance;
}
```

2.1. Kódrészlet. Egyszálú Singleton

Nem szálbiztos Singleton A 2.1 kódrészletben szereplő triviális megvalósítása nem szálbiztos. (Tekintsünk el attól, hogy az objektum soha nem kerül destruálásra. A singleton-ok felszabadításáról további információt a [9] jegyzetben találhatunk). Egyszálú környezetben (és ha interrupt-okból nem használjuk az objektumot), akkor ez helyes megvalósítást ad. Több szál esetén például az a probléma vele, hogy akár több szál is létrehozhat egy-egy objektum-példányt, ha mind NULL-nak látja a pointert.

Szálbiztos Singleton A Singletont egyszerű szálbiztossá tenni (2.2 kódrészlet), csak védjük le az instance() függvény tartalmát egy Lock-kal. (A Lock jelentsen egy RAII osztályt ami konstruáláskor megszerez egy a platformunknak megfelelő mutexet, destruáláskor pedig elengedi.)

```
Mutex Singleton::mutex;

Singleton* Singleton::instance() {
    Lock lock(mutex);
    if(pInstance == 0)
        pInstance = new Singleton;
    return pInstance;
}
```

2.2. Kódrészlet. Helyes többszálú Singleton

A Double Checked Locking Pattern A 2.2 megoldás helyes, de sokak szerint nem elég hatékony, hogy minden egyes lekéréskor Lock-ol. Ha már tudjuk, hogy az objektum elkészült, akkor lehet, hogy nem is kellene lock-olni.

```
Singleton* Singleton::instance() {
    if(pInstance == 0) {
        Lock lock(mutex);
        if(pInstance == 0)
            pInstance = new Singleton;
    }
    return pInstance;
}
```

2.3. Kódrészlet. Hibás DCLP Singleton 1

A Double Checked Locking Pattern (DCLP) azt jelenti, hogy először lock-olás nélkül megnézzük, hogy elkészült-e már az objektum, és csak akkor lock-olunk, ha úgy látjuk, hogy még nem. Ezután újra meg kell néznünk elkészült-e. Itt már a lock-olás miatt biztos, hogy a pointer legfrissebb értékét fogjuk látni. A DCLP tekinthető az egyik legelterjedtebb lock-szegény programozási technikának.

A megvalósítással (2.3 kódrészlet) több probléma van, például a következők.

1. Nem biztos, hogy az utasítások helyes sorrendben fognak végrehajtódni. Lásd: 2.1.1.
2. Nem biztos, hogy a többmagos / többprocesszoros rendszereken a memória-írások helyes sorrendben fognak láthatóvá válni. Lásd: 2.1.2.
3. A pointer írások és olvasások atomiságát nem garantálja a C++03 szabvány (mivel ez a legtöbb modern architektúrán adott (kivéve pl. member-pointer-ek), ezért most nem foglalkozunk vele).

2.1.1. Sorrendiség

A `pInstance = new Singleton;` sor három tevékenységet is végez egyszerre.

1. Lefoglalja a memóriát egy Singleton objektum számára.
2. Meghívja a Singleton objektum konstruktorát a lefoglalt helyen.
3. A pInstance változót átállítja, hogy az újonnan lefoglalt területre mutasson.

Az egyértelmű, hogy az 1. műveletet kell először végezni, azonban a 2. és 3. műveleteket egy fordító felcserélheti. Ekkor egyes szálak egy olyan objektumra mutató pointer-t olvashatnak ki, ami még nem lett inicializálva. ([2] megjegyzi, hogy a 2. és 3. művelet felcserélése csak akkor lehet megengedett, ha a Singleton konstruktora nem dobhat kivételt.)

Jó lenne beleírni a programba, hogy a 2. műveletet a 3. előtt szeretnénk végrehajtani, azonban a C++03 erre nem ad lehetőséget.

C++03-ban minden utasítás vége szekvencia-pont, amire a következő szabályozás vonatkozik. Egy szekvencia-pont elérésekor minden korábbi művelet mellékhatásainak látszani kell, és még semelyik későbbi művelet mellékhatása nem jelenhetett meg.

Azonban ez nekünk nem elég, mivel a C++03 szabvány a program működését csak a megfigyelhető viselkedésen (observable behavior) keresztül szabályozza. Ilyennek az input-output műveletek és a volatile változókon végzett műveletek számítanak. A fordító minden mást (az egyszálú megfigyelhető viselkedés változtatása nélkül) szabadon átrendezhet.

2.1.1.1. A volatile nem elég

A sorrendiséget megköthetjük úgy, hogy bevezetünk egy volatile lokális változót és a pInstance pointer-t is volatile-nak jelöljük (2.4 kódrészlet).

```
Singleton * volatile Singleton::pInstance;
// ...
Singleton * volatile tmp = new Singleton();
pInstance = tmp;
```

2.4. Kódrészlet. Hibás DCLP Singleton 2

Azonban magát a pointer által mutatott objektumot is volatile-nak kell jelölnünk (2.5 kódrészlet), hiszen a konstruktorban lévő műveletek így még nem volatile változókon történnek, tehát áthelyezhetőek a pInstance beállítása után.

```
volatile Singleton * volatile Singleton::pInstance;
// ...
volatile Singleton * volatile tmp = new volatile Singleton();
pInstance = tmp;
```

2.5. Kódrészlet. Hibás DCLP Singleton 3

Azonban kiderül, hogy ez sem elég, mert a volatile objektumok csak a new kifejezés lefutása után válnak volatile-lá. Ezt úgy oldhatnánk meg, hogy a Singleton konstruktorban az összes változóeléréskor

volatile-lá kasztoljuk a változókat.

Ekkor az egy darab egy magos processzorral rendelkező gépeken már csak egy „univerzális” érvünk maradt a helyesség ellen: A szabvány csak egyszálú környezetben követeli meg a helyességet, tehát a fordító nyugodtan generálhat nem-szálbiztos kódot.

Azonban a többmagos/több processzoros gépeken ennél sokkal erősebb érvünk is lesz, ami azt támasztja alá, hogy ez a megvalósítás helytelen.

2.1.1. Megjegyzés. Az eddigi érvekből következik, hogy a volatile változók még egyszerű „volatile bool finished;” jelző-flag-ként sem használhatóak, ugyanis lehet, hogy átrendeződnének az elvégzendő feladat elé, vagy hamarabb jelenne meg az új értékük, mint az elvégzett feladat eredménye.

2.1.2. Megjegyzés. Bizonyos beállítások mellett a Microsoft Visual C++ által implementált volatile [3] (és az 5. verzió óta a Java volatile-ja [4], és a C# volatile-ja [5]) ennél erősebb garanciákkal rendelkezik. Ezek a (más nyelvekben szereplő) erősebb volatile-ok hasonlóan működnek, mint a C++11 atomi változói.

2.1.2. Láthatóvá válás sorrendje többmagos rendszereken

Több-magos rendszereken a különböző magok különböző gyorsítótárakkal (cache-sekkel) rendelkeznek. Előfordulhat, hogy egyes magok cache-sében hamarabb jelenik meg a pInstance pointer új értéke, mint magának az objektumnak a konstruktor által inicializált értéke.

Helyes DCLP singleton C++03-ban platform specifikus barrier-rel A cache-ek szinkronizálási sorrendjét semmilyen C++03 kifejezéssel nem tudjuk megváltoztatni, ezért egy platform-függő (pl. assembly nyelven implementált) memória barrier-t kell használnunk. Ez egy utasítás a fordító, a linker és a processzor számára, amely előírja a lehetséges átrendezések körét és azt, hogy szinkronizálni kell a cache-t. Tegyük fel most, hogy a `my_memory_barrier()` függvény semmilyen átrendezést nem enged az előtte és az utána lévő utasítások között, és a jelenlegi processzor(mag) cache-jét szinkronizálja a memóriával. Ekkor helyes a 2.6 kódrészletben látható megvalósítás.

```
Singleton* Singleton::instance()
{
    Singleton* tmp = pInstance;
    my_memory_barrier();
    if(tmp == 0) {
        Lock lock(mutex);
        tmp = pInstance;
        if(tmp == 0) {
            tmp = new Singleton;
            my_memory_barrier();
            pInstance = tmp;
        }
    }
    return pInstance;
}
```

2.6. Kódrészlet. Helyes DCLP singleton C++03-ban platform specifikus barrier-rel

2.1.3. Megjegyzés. A teljes memória barrier erősebb a szükségesnél, létezik hatékonyabb megoldás is, amit C++11-ben már szabványosan is meg tudunk valósítani.

2.1.3. C++11-es megoldás

C++11-ben már teljesen szabványosan tudjuk leprogramozni az előző megoldást [6] (2.7 kódrészlet). Mint azt később részletesen ismertetem, a release szemantikájú store és a consume szemantikájú load együttesen szinkronizációt biztosít a pointer által mutatott területre.

```
std::atomic<Singleton*> Singleton::pInstance;
std::mutex Singleton::mutex;

Singleton* Singleton::getInstance()
{
    Singleton* tmp = pInstance.load(std::memory_order_consume);
    if (tmp == nullptr)
    {
        std::lock_guard<std::mutex> lock(mutex);
        tmp = pInstance.load(std::memory_order_relaxed);
        if (tmp == nullptr)
        {
            tmp = new Singleton;
            pInstance.store(tmp, std::memory_order_release);
        }
    }
    return tmp;
}
```

2.7. Kódrészlet. Helyes DCLP singleton C++11-ben

2.1.3.1. Meyers singleton

C++11-ben ennél van egy sokkal egyszerűbb megvalósítása a Singleton pattern-nek (2.8 kódrészlet). Ez arra épül, hogy a C++11 szabvány szerint a statikus objektumok inicializálása szálbiztos módon történik. Ez nem biztos, hogy belül a DCLP módszert használja, de abban biztosak lehetünk, hogy a fordítók készítői a végletekig optimalizálták. Ez az ajánlott Singleton megvalósítás C++11-ben. Ennél a megvalósításnál az objektum destruktora automatikusan meghívódik a program végén.

```
Singleton& Singleton::getInstance()
{
    static Singleton instance;
    return instance;
}
```

2.8. Kódrészlet. Meyers singleton (C++11-ben már szálbiztos)

2.1.4. Call-once alapú singleton

A C++11 kínál egy call_once függvényt is, ami a következőt nyújtja: Programfuttatásonként legfeljebb 1-szer futtatja le a paraméterként megadott függvényobjektumot, akkor is ha több szálról és/vagy szálanként többször hívjuk. (Ha nem jut el hozzá a vezérlés, akkor 0-szor hívja meg a függvényt, ha eljut akkor 1-szer.) Emellett szinkronizációt is biztosít: A függvényobjektumot meghívó futását nevezzük aktív futásnak, minden más futását passzív futásnak. Minden passzív futás végén láthatóvá válnak számunkra az aktív futás alatt beállított változók értékei ([1, §30.4.4.2/2]). Ezzel a függvénnyel is megvalósíthatjuk a Singleton pattern-t (2.9 kódrészlet).

```
std::shared_ptr<Singleton> Singleton::pInstance;
std::once_flag Singleton::mOnceFlag;

std::shared_ptr<Singleton> Singleton::getInstance()
{
    std::call_once(mOnceFlag, []{
            pInstance.reset(new Singleton);
    });
    return pInstance;
}
```

2.9. Kódrészlet. Call-once alapú singleton (C++11-ben)

2.2. A függvénykönyvtár alapú megközelítés nem elégséges

A szakasz alapját a hivatkozott [10] cikk képezi. A C nyelvről és a Pthread szálkezelő könyvtárról szól, de a levont következtetései általános érvényűek.

Miért nem elég a függvénykönyvtár alapú megközelítés?

- Programhelyesség szempontjából
- Sebesség szempontjából - Lock-free programozás

A problémát általában az okozza, hogy mivel a nyelv definíciója nem tér ki a többszálú végrehajtásra, a fordító olyan átalakításokat is végezhet, amik csak egyszálú esetben helyesek.

2.2.1. Programhelyesség

A cikk három problématípust említ, ebből az első eddig nem jelentkezett a gyakorlatban, a másodikról keringenek anekdoták, a harmadik által okozott problémákkal viszont már a [10] cikk szerzője is találkozott.

2.2.1.1. Párhuzamos módosítás - versenyhelyzetek

A pthread könyvtár tiltja a versenyhelyzeteket, azaz azt, hogy elérjünk egy változót miközben egy másik szál módosítja.

```
int x = 0, y = 0;
thread_1: if(x == 1) ++y;
thread_2: if(y == 1) ++x;
```

Az a kérdés, hogy ez a pszeudokód tartalmaz-e versenyhelyzetet. Lehetséges/elfogadható az x==1 és y==1 eredmény? A versenyhelyzet definíciója szorosan függ a programozási nyelv szemantikájától. Mi van, ha egy optimalizáló fordító számára megengedett a következő átalakítás:

```
int x = 0, y = 0;
thread_1: ++y; if(x != 1) --y;
thread_2: ++x; if(y != 1) --x;
```

Ekkor lehetséges az x==1 és y==1 eredmény. Azonban az is vitathatatlan, hogy egyszálú végrehajtás esetén ez egy helyes átalakítás.

2.2.1.2. Szomszédos adat felülírása

Az viszonylag evidens, hogy egy bitmező különböző tagjai külön változók, de mégsem változtathatók egyszerre különböző szálakról. Ez azért van, mert egy bitmező tagjának módosítása általában nem csak az adott tag, hanem legalább egy egész bájt felülírásával jár. A jelenlegi processzorok nem támogatják a bit szintű címzést.

Az azonban meglepő lehet, hogy szabályozás hiányában akár byte vagy nagyobb méretű változók esetén is előfordulhat a szomszédos adat felülírása.

Tekintsük a következő (0-ra inicializált) struktúrát.

```
struct {char a,b,c,d,e,f,g,h;} x;
```

Mi történik, ha végrehajtjuk rajta a következő műveleteket:

```
thread_1: x.a = 'a';
thread_2: x.b = 'b'; x.c = 'c'; x.d = 'd';
          x.e = 'e'; x.f = 'f'; x.g = 'g'; x.h = 'h';
```

Ezután lehet x.a == 0?

```
thread_1: x.a = 'a';
thread_2: x = 'hgfedcb\0';
```

Egy 64 bites processzoron lehet, hogy a fordító ezt a kódot állítja elő. Ez gyors, hiszen csak egy gépi szó írásra van szükség hozzá és egyszálú esetben helyes is. Viszont a példánkban előfordulhat, hogy így x.a == 0 lesz.

A Pthread könyvtár egyébként nem csak a változókon, hanem a memória-helyeken értelmezett versenyhelyzeteket is tiltja. A memória-hely fogalma azonban nincs definiálva a Pthread szabványban, jelenthet például bájtot, vagy az adott platform gépi szavát.

2.2.1.3. Regiszterbe emelés

Tekintsük a következő ciklust, ami egy x globális változót módosít.

```
for(...)
{
    ...
    if(mt) pthread_mutex_lock(...);
    x = ... x ...
    if(mt) pthread_mutex_unlock(...);
}
```

Sokszor előfordul, hogy egy programrész egy- és több-szálú környezetben is használható. Csak akkor kell lock-olni, ha több-szálú módon használjuk (azaz be van állítva az mt változó). Ha a fordító úgy gondolja (akár profiling eredményeképpen), hogy az mt változó valószínűleg hamis lesz, akkor generálhatja a következő kódot. A kód az x változót az r regiszterben tárolja, de a könyvtári függvények hívása előtt természetesen vissza-menti a memóriába. (Egy függvényhívás általában invalidálja a regiszterek tartalmát.)

```
r = x;
for(...)
{
    ...
    if(mt){ x = r; pthread_mutex_lock(...); r = x; };
    r = ... r ...
    if(mt){ x = r; pthread_mutex_unlock(...); r = x; };
}
x = r;
```

Ilyenkor az x változó pont a lock-on kívül lesz írva, így értelmét veszti a szinkronizálás. Ha egy ilyen hibát észrevesznek, általában kijavítják a fordítót, de a hiba addig is okozhat problémákat. Az ilyen és a hasonló optimalizálások miatt fontos, hogy egy nyelvnek definiálva legyen a memória modellje. Hasonló „trükkös” optimalizálásokat találhatunk a hivatkozott [8] cikkben.

2.2.2. Sebesség - Lock-free programozás

Teljesítmény-kritikus rendszereken már régóta szokás olyan több-szálú programokat írni, amik tartalmaznak versenyhelyzeteket, mégis helyesek. Vannak problémák, ahol csak így érhető el a kellő gyorsulás. Ezek helyes és portolható írását megkönnyíti, ha a nyelv definiálja a megfelelő atomi műveleteket és a láthatóság (sorrendiség) szabályait. Ezen a szinten már az egyes változóírások szemantikájáról van szó, ezeket szinte lehetetlen lenne egy függvénykönyvtárból hatékonyan megoldani a nyelv segítsége nélkül.

3. fejezet

A C++11 memória modellje

Ez a fejezet főként a C++14 szabvány vázlata alapján készült [1].

3.1. Atomi műveletek

Oszthatatlanság Egy atomi változó írás egészben válik láthatóvá minden atomi olvasás számára (bármely szálon), tehát nem láthatjuk félig átírva a változót. (Signal handlerekben ez csak akkor igaz, ha az atomi művelet lock-free módon van megvalósítva.)

Sorrend Minden atomi változóra igaz az, hogy a rajta végrehajtott (atomi) módosítások egy közös (totális) sorrendben jelennek meg minden szálon (módosítási-sorrend / modification-order). A különböző változókon végzett műveletek egymáshoz képesti sorrendje szálanként eltérhet.

3.1.1. Példa. Íme néhány saját példa a lehetséges ütemezésekre. A következő kimeneteket kaphatnánk bizonyos időközönként kiíratva mit látnak az egyes szálak.

- 1.szál: $A = 1, 2, 2, 2, 3$
  2.szál: $A = 1, 1, 2, 3, 3$
  A értékei módosítási-sorrendben: $1, 2, 3$.
- Az is lehet, hogy egy szál nem *lát* minden változtatást, mert mire olvasná a változót, az már újból megváltozott:
  1.szál: $A = 1, 2, 3$
  2.szál: $A = 1, 1, 3$
  A értékei módosítási-sorrendben: $1, 2, 3$.
- Az előző eset egy speciális változata az úgynevezett ABA probléma:
  1.szál: $A = 1, 2, 1$
  2.szál: $A = 1, 1, 1$
  A értékei módosítási-sorrendben: $1, 2, 1$.

- 1.szál: $(A = 1, B = 1), (A = 2, B = 1), (A = 2, B = 2)$
  2.szál: $(A = 1, B = 1), (A = 1, B = 2), (A = 2, B = 2)$
  A és B értékei módosítási-sorrendben (egyaránt) $1, 2$.

3.1.2. Példa. *Nem* lehetséges a következő ütemezés, amennyiben A értékei módosítási-sorrendben $1, 2, 3$ (vagy $1, 3, 2$):

- 1.szál: $A = 1, 2, 3$
  2.szál: $A = 1, 3, 2$

3.2. Szinkronizáció

3.2.1. Definíció. ***Szinkronizálásnak*** *nevezzük azt, amikor meggyőződünk arról, hogy az egyik szálon történt memóriaírás hatása már látszik egy adott másik szál számára, illetve nem-atomi változó esetén megakadályozzuk, hogy két szál egyszerre módosítsa a memóriaterületet.*

Az atomi műveleteket alapvetően nem kell szinkronizálni. A nem-atomi műveleteket viszont kell, ha több szál is eléri az érintett adatterületet. A nem-atomi műveletek szinkronizálhatók megfelelő atomi műveletekkel, vagy mutex-ekkel.

C++11-ben alapvetően 3 féle szemantikájú szinkronizáló művelet létezik.

- Release szemantikájú atomi memóriaírások (jel.: release(x), ahol x egy változó), read-modify-write műveletek, vagy mutex unlock-olások.
- Acquire szemantikájú atomi memória olvasások, read-modify-write műveletek, vagy mutex lock-olások.
- Consume szemantikájú atomi memória olvasások, read-modify-write műveletek.

Egy release(x)-acquire(x) pár teljes szinkronizációt biztosít (a később leírt módon), míg egy release(x)-consume(x) pár csak az x változótól függő műveleteket szinkronizálja, például amelyek egy, az x-szel kezdődő pointer-láncon elért változót olvasnak.

A C++11-ben alapbeállítás `std::memory_order_seq_cst` minden atomi műveletnek release és acquire szemantikát is ad. Ez (főleg az x86-nál relaxáltabb platformokon) lassabb működést eredményez, mintha csak a release, acquire, vagy consume szemantikát használnánk. Létezik relaxált szemantika is, ekkor az atomi műveletnek semmilyen szinkronizációs hatása nincs.

3.2.2. Definíció. *Ha X egy release szemantikájú memóriaművelet egy m változón, akkor release-sequence(X) = „m módosítási sorrendjének leghosszabb X-től kezdődő összefüggő részsorozata, amiben csak X-el egy szálon lévő műveletek, vagy pedig atomi read-modify-write műveletek vannak".*

3.2.3. Definíció. *A C++11 definiál egy happens-before (korábban-történt) relációt, ami a később leírt módon láthatóságot biztosít a műveleteknek. Az, hogy mikor teljesül a happens-before reláció kiderül a következő táblázatból, ami különböző relációkat ír le. A jelölés-rendszert én vezettem be, elsősorban a dolgozaton belüli használatra.*

Legyenek A és B memóriaműveletek.

A ; B	A is-sequenced-before B	A és B egy szálon vannak és A egy korábbi teljes kifejezésben van mint B. Például pontosvessző választja el őket.
A $;_d$ B	A carries-dependency-to B	(A ; B és B értéke függ A értékétől a szabványban meghatározott módon) vagy $\exists X$: (A $;_d$ X $;_d$ B).
A $<_s$ B	A syncronizes-with B	A=release(x) és B=acquire(x) és B-ben az A által beírt értéket olvassuk ki.
A $<_d$ B	A is-dependency-ordered-before B	(A=release(x) és B=consume(x) és B-ben a release-sequence(A) egy tagját olvassuk) vagy (A $<_d$ X $;_d$ B).
A $<$ B	A inter-thread-happens-before B	A $<_s$ B vagy A $<_d$ B vagy $\exists X$: (A $<_s$ X ; B) vagy (A ; X $<$ B) vagy (A $<$ X $<$ B)
A $\prec$ B	A happens-before B	A ; B vagy A $<$ B. (Nem tranzitív.)

3.1. táblázat. A C++11-ben definiált happens-before-hoz kapcsolódó relációk

3.2.4. Példa. A következő példa azt mutatja, hogy $\prec$ nem tranzitív: $A <_d B; C$ -ből *nem* következik A $\prec$ C.

3.2.5. Definíció. *Egy m változón végzett A mellékhatást (memória írást) láthatónak nevezünk egy M-en történő B számítás számára, ha $A \prec B \wedge \nexists X : A \prec X \prec B$.*

3.2.6. Megjegyzés. Egyszerre több mellékhatás is lehet látható egy számítás számára. Ekkor a később definiált data-race következett be.

3.2.7. Definíció. *Egy nem-atomi M változón data-race következik be, ha $\exists A, B$ művelet: $A = write(M) \wedge (B = read(M) \vee B = write(M)) \wedge A \not\prec B \wedge B \not\prec A$. Ez nem-definiált állapotba hozza a programot.*

3.2.8. Állítás. *Ha $A; B = release(X)$ és $C = acquire(X); D$ és C a B által írt értéket olvassa, akkor D számára láthatóak az A által okozott mellékhatások. (Tehát a release-acquire párral lehet szinkronizálni.)*

Bizonyítás. A bizonyításban bezárójelezem az aktuálisan összevonandó részkifejezést és nyíllal választom el a levezetés egyes lépéseit.
$A; (B <_s C; D) \rightarrow A; B < D$
$(A; B < D) \rightarrow (A < D) \rightarrow A \prec D$ □

3.2.9. Állítás. *A release(X)-consume(X) szinkronizáció szinkronizál minden, az X változóból pointerláncon elérhető változót. Másképp megfogalmazva, ha D a C által írt értéket olvassa (és máshol nem módosul a változó), akkor answer = 42 a 3.1 kódrészletben.*

```
std::atomic<Object*> x;

thread_1:
A:    Object* tmp = new Object;
B:    tmp->m->...->m = 42;
C:    x.store(tmp, std::memory_order_release);

thread_2:
D:    Object* tmp = x.load(std::memory_order_consume);
E1:    auto& tmp1 = tmp->m;
E2:    auto& tmp2 = tmp1->m;
    ...
F:    int answer = tmp(n-1)->m;
```

3.1. Kódrészlet. Példa a happens-before levezetéséhez

Bizonyítás. $B;C <_d D;_d E1;_d E2;_d \cdots ;_d F$, ha D a C által írt értéket olvassa.
$B;C <_d (D;_d E1;_d E2;_d \cdots ;_d F) \to B;C <_d D;_d F$
$B;(C <_d D;_d F) \to B;C < F$
$(B;C < F) \to (B < F) \to B \prec F$ $\square$

3.2.10. Megjegyzés. A következő Prolog program segít a fentihez hasonló állítások eldöntésében a C++11 memória modellel kapcsolatban:
`https://gitlab.com/tomi92/lockfree/blob/master/relations/mem.pl`

3.2.1. Szekvenciális konzisztencia

Az atomi műveletek elvégezhetőek *szekvenciálisan konzisztens* (`std::memory_order_seq_cst`) módon, az ilyen műveletekre igaz az, hogy sorrendjük egymáshoz képest is minden szálon azonos. Ez az alapbeállítás C++11-ben, de ha nem ezt használjuk gyorsabb (és kellő validálás hiányában hibás) programokat kaphatunk.

3.2.11. Definíció. *A szekvenciálisan konzisztens műveleteknek létezik egy S abszolút sorrendje, ami konzisztens az egyes változók módosítási sorrendjeivel és a korábban-történt relációkkal.*

3.2.12. Következmény. Ebből következik az, hogy az egy szálon belüli szekvenciálisan konzisztens műveletek sorrendje ugyanaz lesz az S sorrenden belül is, mint ahogy a kódban követik egymást (sequenced-before).

Az S abszolút sorrendben található B: read(X) művelet a következő értékek *bármelyikét* láthatja:

- Az S-ben B előtti utolsó A: write(X) művelet eredményét, ha van ilyen.
- Ha létezik az előbbi A művelet és létezik C: write(X) nem szekvenciálisan konzisztens művelet, hogy C $\not\prec$ A, akkor C eredményét is.
- Ha nem létezik az előbbi A művelet, akkor bármely C: write(X) nem szekvenciálisan konzisztens művelet eredményét.

3.2.13. Példa. Lehetségesek például a következő esetek: (Az egyes felsorolt műveletek különböző szálakon is lehetnek, a sorrendjük tetszőleges.)

- $S_1 = A = write(X), S_2 = B = read(Y), S_3 = C = read(X)$, és máshol nem használják X-et, ekkor C csakis A eredményét láthatja.
- $S_1 = A = write(X), S_2 = B = read(Y), D = write(X), S_3 = C = read(X)$, $D \not\prec A$, és máshol nem használják X-et, ekkor a C művelet A vagy D eredményét láthatja.
- $D = write(X), S_1 = A = write(X), S_2 = B = read(Y), S_3 = C = read(X)$, $D \prec A$, és máshol nem használják X-et, ekkor a C művelet csakis A eredményét láthatja.

3.2.2. Kerítések (Atomic thread fences)

Motiváció: Előfordulhat, hogy a szálak közötti szinkronizációt külön szeretnénk választani az atomi változók módosításától. Például, tegyük fel, hogy egy új adat megérkezését jelző változót olvasunk. Amennyiben nem érkezett új adat, felesleges lenne szinkronizálni, ha viszont érkezett akkor jó lenne egy módszer amivel utólag acquire szemantikájúvá változtathatnánk ezt az olvasást. [24] Pont ezt tudjuk elérni a kerítésekkel.

A kerítések hatásai

Egy acquire szemantikájú kerítés (`std::atomic_thread_fence(std::memory_order_acquire)`) lényegében acquire szemantikájúvá változtatja az *előtte* lévő (pl. relaxált) atomi olvasást, egy release szemantikájú kerítés pedig lényegében release szemantikájúvá változtatja az *utána* lévő atomi írást.

A *lényegében* szóra azért volt szükség mert ilyenkor csak a release fence előtti műveletek eredményei lesznek láthatóak és csak az acquire fence után. Mégis van szerepük az atomi változókon végzett műveleteknek, mivel önmagukban a kerítések nem biztosítanak szinkronizációt.

Léteznek még acquire-release (egyszerre acquire és release), illetve szekvenciálisan konzisztens acquire-release szemantikájú kerítések is. Az utóbbiak részt vesznek a szekvenciálisan konzisztens műveletek abszolút sorrendjében.

A szekvenciálisan konzisztens kerítések használhatóak a szabványban leírt módon arra, hogy egy adott változón végzett írás és olvasás között kikényszerítsék a láthatóságot, vagy arra, hogy egy adott változón végzett írás és írás között kikényszerítsék a módosítási sorrendet. Viszont (idézet a szabványból) általánosságban nem használhatók relaxáltabb szemantikájú műveletek sorrendjének kikényszerítésére.

Szignál kerítések Az eddig említett `atomic_thread_fence` műveleten kívül létezik `atomic_signal_fence` is. Ez csak egy szál és az ugyanezen szálon futtatott szignál kezelő eljárás (signal handler) között végez szinkronizációt. Nem tartalmaz hardveres szinkronizáló utasítást, csak a fordító számára írja elő, hogy mely sorrendbeli átrendezések tilosak.

4. fejezet

Lock-free adatszerkezetek vizsgálata

4.1. Definíciók

A bevezetés alapjául a hivatkozott [11] könyv szolgált.

4.1.1. Definíció. ***Blokkolónak*** *nevezzük az olyan könyvtári eljárásokat, amik leállítják a jelenlegi szál végrehajtását, amíg egy másik szál el nem végez egy adott műveletet.*

4.1.2. Definíció. *A blokkoló hívásokat használó adatszerkezeteket* ***blokkoló adatszerkezeteknek*** *nevezzük.*

4.1.3. Definíció. *A nem blokkoló adatszerkezeteket* ***nem-blokkoló adatszerkezetnek*** *nevezzük.*

4.1.4. Megjegyzés. A mutexek, condition variable-ök, és future-ök általában blokkoló hívásokat használnak, ezért azok az adatszerkezeteket, amik ilyen szinkronizálási primitíveket használnak, általában blokkoló adatszerkezetek.

4.1.5. Definíció. *Az olyan adatszerkezeteket, amik lock-nál alacsonyabb szintű primitíveket is használnak szinkronizálásra, vagy információ-megosztásra, lock-szegény (****low-lock****) adatszerkezeteknek hívjuk. (Ez a kevésbé elterjedt elnevezés hasonló jelentéssel szerepel például a hivatkozott [13] könyvben.)*

4.1.6. Definíció. *A következő tulajdonságokat teljesítő adatszerkezeteket* ***lock-free****-nek nevezzük:*

- *Több mint egy szál érheti el egyszerre az adatszerkezetet. (A használható műveletek halmaza szálanként eltérő lehet, de legalább két szál esetében nem üres.)*
- *Ha bármely szál kivételével az összes többi szál végrehajtását felfüggesztjük, akkor a futó szálnak be kell tudnia fejeznie az aktuális eljárás végrehajtását a többi szálra való várakozás nélkül.*

4.1.7. Következmény. A lock-free adatszerkezetek egyben block-free adatszerkezetek is.

4.1.8. Megjegyzés. Ahhoz, hogy egy adatszerkezet ne legyen lock-free, nem kell mutexet alkalmaznia, elég az is ha maga az adatszerkezet úgy működik, mint egy spin-lock. Például, ha az egyik szál addig vár egy ciklusban, amíg a másik át nem állít egy változót.

4.1.9. Definíció. *Ha egy lock-free adatszerkezetre igaz a következő feltétel is, akkor* ***wait-free*** *adatszerkezetnek hívjuk.*

- *Minden t szálra létezik egy n_t véges és a többi szál működésétől független szám, hogy a t szál n_t lépés alatt befejezi az adatszerkezet aktuális eljárását.*

4.1.10. Megjegyzés. A lock-free adatszerkezetek általában egy ciklusban használt compare/exchange operációt használnak, ami addig működik, amíg egyszer le nem fut úgy, hogy nem történik közben másik szálon módosítás. Az ilyen adatszerkezetek általában lock-free, de nem wait-free tulajdonságúak, hiszen az egyik szál meg tudja váratni a másikat. Ezt éheztetésnek/starvation-nek nevezzük.

4.1.11. Definíció. ***Deadlock****-nak nevezzük azt a helyzetet, amikor egy szál-halmaz minden eleme a halmaz egy másik eleme által fogott mutexre vár. Általában ez azt jelenti, hogy a program működése nem tud folytatódni.*

4.1.12. Definíció. ***Livelock****-nak nevezzük azt a szerencsétlen esetet, amikor több szál közül az egyik sem tud tovább haladni, mivel mindkettő olyan módosításokat végez, ami arra kényszeríti a másikat hogy újrafuttassa az aktuális ciklusát. Ezek általában nem tartanak sokáig, mivel a livelock fennállása függ a pontos ütemezéstől.*

A lock-free adatszerkezetek előnyei a nem lock-free adatszerkezetekkel szemben.

- Jobban kihasználják a konkurrencia előnyeit, mivel általában nincs kölcsönös kizárás a műveletek között.
- Nem történhetnek miattuk deadlock-ok, csak a kevésbé veszélyes livelock-ok (wait-free esetben azok sem).
- Robosztusabbak, ugyanis, egy szál „halála” nem tudja megakadályozni a többi szál folytatását. (Azért még vigyázni kell, hogy ne romoljon el az adatszerkezet egyik invariánsa sem.)

A lock-free adatszerkezetek hátrányai a nem lock-free adatszerkezetekkel szemben.

- A szinkronizálás sokkal bonyolultabb (atomi műveletek segítségével történik). Ha nagyon sok helyen szinkronizálunk, ez akár lassabb működést is eredményezhet, mintha csak egy mutexet lock-olnánk.
- Figyelni kell, hogy az egyes módosításokat jó sorrendben lássák a szálak.
- A lock-free, de főleg a wait-free adatszerkezetek műveletei sokkal bonyolultabbak lehetnek a blokkoló műveleteknél. Ezek a műveletek akár egy-szálú működés esetén is lassabbak lehetnek mint nem-lockfree társaik.
- Ha ugyanazt az atomi változót több szál is írja az gyakori cache-invalidálást okozhat.

4.2. Queue

```
template<typename T, int Capacity>
class queue
{
    T mData[Capacity+1];
    alignas(CacheLineSize) std::atomic<int> mBase;
    alignas(CacheLineSize) std::atomic<int> mNext;
public:
    queue()
    : mBase(0)
    , mNext(0)
    {}
    bool push_t1(const T& value)
    {
        int base = mBase.load(std::memory_order_acquire);
        int next = mNext.load(std::memory_order_relaxed);
        if(next != wrap(base-1))
        {
            mData[next] = value;
            mNext.store(wrap(next+1),std::memory_order_release);
            return true;
        }
        return false;
    }
    bool pop_t2(T& out)
    {
        int base = mBase.load(std::memory_order_relaxed);
        int next = mNext.load(std::memory_order_acquire);
        if(base != next)
        {
            out = mData[base];
            mBase.store(wrap(base+1), std::memory_order_release);
            return true;
        }
        return false;
    }
private:
    static int wrap( int i )
    {
        while(i>=Capacity+1)
            i-=(Capacity+1);
        while(i<0)
            i+=(Capacity+1);
        return i;
    }
};
```

4.1. Kódrészlet. A lock-free queue adatszerkezet

4.2.1. Bevezetés

Az első adatszerkezet egy általam írt korlátozott méretű ciklikus sor, amely pontosan 2 szál közötti egyirányú adatátvitelre használható. Ez az adatszerkezet a korábban definiált értelemben lock-free és wait-free.

Az adatszerkezet három lépésben készült el.

1. Az első lépésben nem szóltam bele a változók alignolásába, sem pedig a load és store műveletek memória order-jébe.

2. A másodikban hozzáadtam az mBase és az mNext változók cache line-ra való align-olását. Ez azt jelenti, hogy külön cache line-ra kerültek, és így amikor az egyik szál beleír valamelyik változóba, akkor nem kell megvárnia amíg a másik által a másik változóba történt írás átszinkronizálódik (cache-koherens rendszereken sem). Ezt a jelenséget nevezzük false sharing-nek. Ez egy biztonságos lépés, csak több helyet igényel, de alapvetően nem ronthatja el a programot. Később bebizonyosodott, hogy ez esetünkben nem jelent (egyértelmű) gyorsítást minden platformon (lásd: 4.2.4).

3. A harmadik lépés egy kicsit kockázatosabb, itt átírtam a store-okat és loadokat az alapértelmezett szekvenciálisan konzisztens memóriamodellről relaxáltra, release-re, vagy acquire-re. Ez lényeges gyorsítást okozott (lásd: 4.2.4).

4.2.1. Megjegyzés. A dinamikus memóriafoglalású lock-free adatszerkezetek általában compare-and-swap (összehasonlítás-és-csere) műveletekből álló ciklusokat használnak. Ezek használatát itt nem mutatom be.

4.2.2. Helyesség

A queue működésétől a következőt várjuk el: Az n. sikeres olvasás az n. sikeres íráskor beírt adatot kell, hogy visszaadja. Az hogy az olvasófej a megfelelő helyen lesz nem vitás (írásonként és olvasásonként is 1-el növeljük ($\bmod c+1$)), csak az kérdéses, hogy ott van-e a megfelelő adat.

4.2.2. Állítás. *A következő feltételekből következik, hogy queue-ból mindig a megfelelő adatot olvassuk ki. Legyen c a capacity rövidítése.*

1. *Minden $n \geq 1$-re: Az n. írás korábban-történt az n. olvasásnál.*

2. *Minden $n \geq 1$-re: Az n. olvasás korábban-történt az (n+c). írásnál (ha van ilyen).*

Bizonyítás. Ezek pont azt jelentik, hogy az n. olvasáskor az n. írás eredménye már látszik, de az $n+c$. vagy későbbi írások még nem írták felül. (Az írások sorrendje alatt az író szálon való sorrendjüket értjük, ugyanígy az olvasások sorrendje alatt az olvasó szálon való sorrendjüket.) □

Most tegyük fel a következőket: az mNext és az mBase minden írásával/olvasásával egy műveletben, az írások számát (w) és az olvasások számát (r) is lementjük/beolvassuk. A t1-ben így beolvasott olvasásszám legyen r', a t2-ben így beolvasott írásszám pedig w'. Az if-ek feltételeit pedig a(z) 4.2.2 feltételekkel összhangban a következőkre cseréljük: push_t1-ben: ($r' \geq w+1-c$), pop_t2-ben pedig: ($r+1 \leq w'$).

Fogjuk fel úgy a számítógép működését, hogy egy összefüggő irányított aciklikus gráfot gyárt. Minden műveletkor 1 új pontot (és valamennyi élt) ad hozzá. Ebben a gráfban a korábban-történt ($\prec$) relációknak az élek felelnek meg. A kezdőpont maga a konstruktor, amiből él megy minden írási és olvasási kísérletbe. A sikeres írás-/olvasásokat jelölje kitömött piros/kék pont. Ha a feltétel miatt nem írunk/olvasunk azt jelölje üres piros/kék pont. Az írási és olvasási kísérletek között is haladnak néhol élek. Az újabb pont hozzá vételekor a színéhez tartozó if alapján döntünk a kitöltéséről.

4.2.3. Állítás. *Így csak (4.2.2-re nézve) helyes gráfot állíthatunk elő, azaz a módosított queue működése helyes.*

Bizonyítás. Használjunk indukciót. Az 1 pontú gráf helyes. Bizonyítsuk be, hogy ha egy eddig helyes gráfhoz hozzáveszünk egy újabb pontot akkor a pont hozzáadásával is helyes marad a gráf.

- Ha a pont piros ($w+1$. írást kíséreljük meg): Tudjuk, hogy legalább r' olvasás korábban-történt nála. Az $r' \geq w+2-c$ feltételből pedig következik, hogy csak akkor töltjük ki a pontot (tesszük meg a lépést), ha megfelelő számú olvasás korábban történt nálunk (teljesül 4.2.2.2).
- Ha a pont kék ($r+1$. olvasást kíséreljük meg): Tudjuk, hogy legalább w' írás korábban-történt nála. Az $r+1 \leq w'$ feltételből pedig következik, hogy csak akkor töltjük ki a pontot (tesszük meg a lépést), ha megfelelő számú írás korábban történt nálunk.

□

Most már csak azt kell belátnunk, hogy az eredeti feltételek is ugyanazt a gráfot produkálnák, mint az újak.

4.2.4. Állítás. *Az eredeti feltételekkel is ugyanazt a gráfot adja a számítógép, mint az előző állításban.*

Bizonyítás. Indukció. Az egy pontú gráfra ez igaz (ott még nem nézünk feltételt). Tegyük fel, hogy a meglévő gráfunkra is igaz (tehát ez a gráf helyes is), és most veszünk hozzá egy pontot. Lássuk be, hogy ekkor az új pontra vonatkozó új feltétel negálásából következik az eredeti feltétel negálása.

Ha a pont piros (írás): Ha a feltétel nem igaz ($r' < w+1-c$), abból (és az eddigi helyességből) következnie kell, hogy $next = wrap(base-1)$.
$r' < w+1-c$, de $r' \geq w-c$, mert különben már korábban megsérült volna a feltétel. Emiatt $r' = w-c \rightarrow r'-1 = w-c-1 \rightarrow r'-1 \equiv w \mod c+1, \rightarrow next = wrap(base-1)$. Ha a feltétel igaz ($r' \geq w+1-c$), akkor pedig kell, hogy $next \neq wrap(base-1)$. Indirekt tegyük fel, hogy $next = wrap(base-1)$. Ekkor $r'-1 = w+k(c+1)$. $r' \geq w+1$ nem lehetséges. $r' \leq w-c$ szintén nem lehetséges.

Ha a pont kék (olvasás): Ha a feltétel nem igaz ($r+1 > w'$), abból (és az eddigi helyességből) következnie kell, hogy $next = base$. $r+1 > w'$, de $r \leq w' \rightarrow r = w' \rightarrow r \equiv w' \mod c+1 \rightarrow base = next$. Ha pedig a feltétel igaz, ($r+1 \leq w'$), akkor kell, hogy $base \neq next$. Indirekt tegyük fel, hogy $base = next$. Ekkor $r = w' (\mod c+1)$, tehát $r \geq w'$ (nem lehetséges), vagy $r \leq w'-c-1$ (nem lehetséges). □

Az algoritmus helyességéhez természetesen az is kellett, hogy az egyes függvényhívásokon belüli műveletek eredményei megfelelő sorrendben látszódjanak a másik szálról, például, hogy az olvasás it-hamarabb-történjen, mint az mBase növelése. Az algoritmus egy korábbi változatában ez sajnos nem volt így, de egy Relacy nevű hibakereső programmal (mint később olvasható) szerencsére ez kiderült és javítva lett.

4.2.5. Állítás. *Ha valamit beírtunk, akkor idővel ki is tudjuk olvasni, és ha eleget kiolvastunk, ahhoz, hogy legyen üres hely, akkor idővel írni is tudunk az adatszerkezetbe. Tehát nem kell a végtelenségig várni az adatszerkezet használatának a folytatásához.*

Bizonyítás. A c++11 szabvány szerint az implementációknak minden atomi írást elfogadható időn belül láthatóvá kell tenni az atomi olvasások számára. □

4.2.3. Kerítések használata

Adatszerkezetünkben a feltételes kifejezések előtti acquire műveleteket relaxálttá tehetnénk, ha egy acquire fence-et tennénk a feltételes blokkok elejére. Ez azzal az előnnyel járna, hogy csak akkor szinkronizálnánk, ha tényleg van új adat, illetve tudunk beszúrni.

4.2.4. Futási idők összehasonlítása

4.2.4.1. Tesztelés módja

Az alábbi futási idők a tesztprogramok 100 egymás utáni futtatásának együttes idejei. Az egyes tesztprogramok 1000 000 sikeres push és ugyanennyi sikeres pop műveletet hajtottak végre külön-külön szálon (100 kapacitású sor használatával). Nem volt garantálva, hogy a külön szálak külön magon fussanak, de ez általában így történt (A `time` parancs majdnem 200%-os processzor használatot mutatott: ez csak akkor lehetséges, ha több mag van használatban). Az egyszálú tesztben a 100 push és 100 pop művelet váltogatta egymást, összességében ugyanannyiszor lefutva, mint a többszálú esetben. Az x86-64 processzor pontos típusa: Intel Core i5-3230M (2 mag $\times$ 2 logikai szál, 2,6 Ghz). Az arm-v7 processzor pontos típusa: Qualcomm Snapdragon 400 (4 mag, max. 1,4 Ghz (terhelés függvényében változó)).

4.2.4.2. Az egyes sorok

Az Lock-free 1.1 - 1.4 sorok kerültek ebben a fejezetben ismertetésre/említésre. Ezeknek és a többi sornak a megvalósítását lásd a(z) [34] git tárolóban. A sorok főbb tulajdonságai:

- Egyszálú sor: egyszálú so.
- Lock-free 1.1: Az aktuális fejezetben ismertetett szekvenciálisan konzisztens szemantikát használó lock-free sor.
- Lock-free 1.2: Mint az előző, csak az mBase és mNext változók cache-line-ra vannak igazítva (align-olva).
- Lock-free 1.3: Az aktuális fejezetben ismertetett release / acquire szemantikát használó lock-free sor az mBase és mNext változók cache-line-ra igazításával (align-olásával).
- Lock-free 1.4: Az aktuális fejezetben említett (4. fejezet) release / acquire szemantikájú kerítéseket használó lock-free sor az mBase és mNext változók cache-line-ra igazításával (align-olásával).

- Lock-free 2.1: Olyan release / acquire szemantikát használó lock-free sor, ahol minden elemet egy-egy „hasData” atomi boolean változó véd és 1-1 szálon belül használt mBase/mNext változók cache-line-ra vannak igazítva.
- Lock-free 2.2: Mint az előző, csak az egyes elemek is cache-line-ra vannak igazítva.
- Globális mutex: Olyan sor, ahol minden hozzáférést egy globális mutex véd.
- Elemenkénti mutex 1: Olyan sor, ahol a hozzáféréseket elemenkénti mutex védi.
- Elemenkénti mutex 2: Mint az előző, csak az 1-1 szálon belül használt mBase/mNext változók cache-line-ra vannak igazítva.
- Elemenkénti mutex 3: Mint az előző, csak még az egyes elemek is cache-line-ra vannak igazítva.

4.2.4.3. A táblázat

A táblázatban az „arány” jelű oszlopokban az időtartamok le vannak osztva a „Lockfree 2.1” megvalósítás futásidejével.

	x86-64		**arm-v7**	
	futási idő	arány	futási idő	arány
Egyszálú futási idő	0,65 mp	0,60	2,85 mp	0,32
Lock-free 1.1 (seq-cst)	7,18 mp	6,65	18,86 mp	2,14
Lock-free 1.2 (seq-cst, align)	6,84 mp	6,33	20,11 mp	2,28
Lock-free 1.3 (rel/acq, align)	1,85 mp	1,71	12,48 mp	1,42
L.-f. 1.4 (rel/acq fence, align)	1,81 mp	1,68	12,03 mp	1,37
Lock-free 2.1 (rel/acq, align)	1,08 mp	1,00	8,81 mp	1,00
L.-f. 2.2 (rel/acq, (item) align)	1,12 mp	1,04	8,04 mp	0,91
Globális mutex	26,16 mp	23,30	421,48 mp	47,84
Elemenkénti mutex 1	12,16 mp	11,26	12,42 mp	1,41
Elemenkénti mutex 2 (align)	7,45 mp	6,90	11,32 mp	1,28
El. mutex 3 ((item) align)	6,97 mp	6,45	11,93 mp	1,35

4.1. táblázat. A tesztek futásideje a különböző sor adatszerkezeteken

4.2.4.4. Értékelés

4.2.6. Megjegyzés. Ez a teszt úgy mérte fel a szinkronizálási primitívek futási időigényét, hogy közben az adaton végzett egyéb műveletek (másolás/konstruálás/feldolgozás) elhanyagolhatóak voltak. Ha az egyéb műveletek nagyobb erőforrás igényűek lennének, akkor értelemszerűen kevésbé térnének el a kapott időtartamok.

4.2.7. Megjegyzés. Az egyszálú tesztek azért is lehettek ilyen gyorsak, mert ott nem fordulhat elő, hogy az egyik szálnak várnia kell a másikra.

Az **x86-64 platformon** egyértelműen a release/acquire szemantikát használó lock-free sor nyújtotta a legjobb futási időt. Az elemenkénti mutex-et használó sor és a szekvenciálisan konzisztens lock-free sor egyaránt kb. 7-szer több ideig futott a „nyertesnél". A globális mutex-et használó sor 26-szor futott tovább a „nyertesnél". A jellemzően csak 1-1 szál által írt változók külön cache-line-ra igazításának érezhetően kedvező hatása volt (az „elemenkénti 2" futásideje 0,6-szorosára csökkent az „elemenkénti 1"-hez képest). Az elemek align-olásának nem egyértelmű a hatása (pl. azért is ronthat a teljesítményen, mert így az elemek egymás utáni olvasásakor nagyobb memóriaterületet kell betölteni).

Az **arm-v7 platformon** a release/acquire szemantikát használó lock-free sor csak egy kicsivel előzte meg az elemenkénti mutex-et használó sort (utóbbi 1,41-szer futott tovább). A szekvenciálisan konzisztens lock-free sor 2,35-ször futott tovább a „nyertesnél". A globális mutex-et használó sor 52-szer futott tovább a „nyertesnél". A jellemzően csak 1-1 szál által írt változók külön cache-line-ra igazításának itt nem volt egyértelmű a hatása. Az elemek align-olásának nem egyértelmű a hatása.

Összefoglalva azt gondolom, hogy általános használatra az elemenként lock-olt változat a legmegfelelőbb, mivel nagyon kedvező futási időt biztosít viszonylag egyszerűen érthető kód mellett. Ha pedig minden utolsó processzor ciklus számít, akkor érdemes a release/acquire szemantikájú lock-free változatot használni (ez a leggyorsabb, de a tervezése és a validálása nagyon munka-igényes). A többi változatnak (pl. szekvenciálisan konzisztens lock-free) nincs egyértelmű előnye ezekhez képest. A jellemzően csak 1-1 szál által írt változókat érdemes külön cache-line-on tárolni.

4.2.8. Megjegyzés. Ipari környezetben természetesen nem csak az adatszerkezet használatának gyorsaságát, hanem egyéb tényezőket is figyelembe kell venni a tervezésnél (szabványosság, karbantarthatóság, megbízhatóság).

4.3. Relacy race detector (RRD)

A fejezet forrása a Relacy forráskódja [14] és a készítővel készült interjú [15].

A Relacy egy viszonylag kevéssé ismert és karbantartott, de hasznos hibakereső, csak header fájlokból álló library, amely lock-free és mutex-alapú többszálú programok ellenőrzésére is nagyon egyszerűen használható. Működési elve az, hogy egy rövid kódfuttatást minden (a szálkezelési modell szerint megengedett) ütemezés szerint lejátssza egy szálon (fiber-ekkel) és közben nézi történnek-e data-race-ek. A consume operációt nem támogatja, acquire-el helyettesíti. Használata kisebb-nagyobb kód-módosításokat igényel (lásd: A függelék).

4.3.1. Jelölés. *A program nyilvántartja minden (t, u) szál-párról, hogy u-nak melyik az a legutóbbi release vagy acquire művelete, amit t lát (korábban-történt relációban van u valamelyik megtörtént műveletével). Ezt egy számmal írjuk le, ami az u-n végzett műveletek szálon belüli sorrendje. Jelölje ezt ord(t,u).*

4.3.2. Jelölés. *Továbbá minden általunk megjelölt M nem-atomi változóról és minden t szálról nyilvántartja, hogy t-n mikor lett utoljára írva/olvasva M. Ez is egy az előzőhöz hasonlóan definiált sorszám. Jelölje ezeket w(t,M)/r(t,M).*

Ha a következők egyike teljesül, akkor data-race-t talál (itt ord és r/w értékei közül mindig az egy szálon végzett szimulációban épp aktuálisra gondolunk):

- Egy t szálon írni próbál egy M nem-atomi változót és $\exists u$ szál, hogy $ord(t, u) < \max\{r(u, M), w(u, M)\}$.
- Egy t szálon olvasni próbál egy M nem-atomi változót és $\exists u$ szál, hogy $ord(t, u) < w(u, M)$.

Ez összhangban áll a C++11 data-race definíciójával (3.2.7).

A program azt mondja meg, hogy melyik sorban próbáltuk írni vagy olvasni a változót, mikor megtalálta a data-racet. Érdekes javítás lenne, hogy ilyenkor mondja meg, hogy pontosan melyik típusú data-race-ről van szó és melyik sorokkal van konfliktusban.

Példa egy talált hibára A $push_{t1}$-ben található mBase.load-ot és a pop_{t2}-ben lévő mBase.store-t állítsuk át relaxálttá. Ekkor a Relacy data-race-t jelez az „out = mData[base];" sorában. Ez egyedül az „mData[next] = value;" sorral állhat konfliktusban. Valóban, ilyenkor az n. olvasás és az (n+c). írás között nem volt korábban-történt reláció. Tehát megtörténhetett, hogy az író szál hamarabb látta az *mBase* növelését minthogy az olvasóban kiolvasódott a megfelelő elem. Hiába van mBase növelése a kiolvasás után a kódban, attól ezek hatásai látszódhatnak fordított sorrendben a többi szálról. Kellő szinkronizálás hiányában a C++ bármely olyan műveletek sorrendjét felcserélheti, ami *egyszálú program esetén* nem változtatna a látszólagos sorrenden. Hasonló data-race kereső nélkül valószínűleg sosem derült volna fény a hibára, hiszen elképzelhető, hogy az általunk használt gépeken csak nagyon kis valószínűséggel történik meg a baj. x86 platformon ráadásul minden írás release és minden olvasás acquire szemantikájú, tehát ott sosem lett volna ebből gond [16].

5. fejezet

RCU - Olvasás közben módosítható adatszerkezetek

5.1. Bevezetés

Az RCU (Read-copy-update) egy olyan szinkronizálási mechanizmus, ami lehetővé teszi, hogy úgy módosítsunk egy adatszerkezetet, hogy közben akár több szál is olvassa. Legtisztább formájában az olvasó szálakra semmilyen szinkronizálási költség nem hárul. 2002 októberében jelent meg a Linux kernelben. [17]

Alapvető működés Az adatszerkezet nagyobb elemei pointerek által vannak összekötve. Egy-egy ilyen elem olvasásakor, beszúrásakor, vagy törlésekor atomi műveleteken keresztül olvassuk vagy módosítjuk a rá mutató pointereket. Az egyes olvasó-szálak olyan elemeket is olvashatnak, amik időközben törlődtek (kiláncolódtak) az adatszerkezetből. Ahhoz, hogy ténylegesen deallokálhassunk egy objektumot, meg kell várnunk, hogy minden őt olvasó szál befejezze az olvasását. Ennek az elvárásnak a teljesítése adja a fő nehézséget az RCU implementálásakor. Tekinthetünk úgy is az RCU-ra mint egy speciális garbage collector típusra. Ehhez hasonló a Hazard Pointer módszer, ami 2010-ig az IBM szabadalma volt [20][21].

Példa a felhasználásra Az RCU módszer olyan esetben a leghasznosabb, ha sok olvasó szálnak kell működnie minimális várakozási időkkel, ugyanakkor a beszúrások/törlések ritkák, ezért elfogadható, ha kicsit lassabbak. Például egy Domain név szervert (DNS) elképzelhetünk úgy, hogy egy központi hasító-táblában tárolja a domain neveket, amiket több szálról, nagy teljesítménnyel olvas, amikor a kliensek lekérik egy-egy domain adatait. Frissítéseket ezzel szemben sokkal ritkábban végez. Amikor új elem szúródik be a hasító-tábla egy vödrébe (az azonos hash-kódú elemek egy listájába), akkor az olvasóknak nem kell várniuk, hanem zavartalanul olvashatják az adatszerkezetet. Ami meglepő és sokkal nagyobb előnyt jelent, hogy a hasító-tábla átméretezése is megoldható úgy, hogy közben az adatszerkezet folyamatosan olvasható marad.

Olvasás Az RCU-védett elemek olvasásait ún. olvasó-oldali kritikus-szakaszokban kell végezni. Egy ilyen kritikus-szakaszban akár több RCU védett elem olvasását is végezhetjük, ennek csak annyi hatása lesz, hogy a közben törölt elemek deallokálását későbbre halaszthatja. Az `rcu_read_lock` és `rcu_read_unlock` műveletek egyes megvalósításokban az üres utasításra fordulnak le, de szemantikai tartalmuk miatt továbbra is ajánlott kiírni őket. Az `rcu_dereference` művelet egy consume szemantikájú load-ot hajt végre atomi módon a paraméterén.

```
//C code

Elem* p;

void reader(void)
{
    rcu_read_lock();
    Elem* tmp = rcu_dereference(p);
    /* use tmp->member1, tmp->member2, etc. */
    rcu_read_unlock();
}
```

5.1. Kódrészlet. RCU olvasás

Írás A beszúrás/törlés nem feltétlenül lock-free módon történik. A `synchronize_rcu` művelet azt teszi, hogy megvárja, amíg minden szál kilép az oldElem olvasó-oldali kritikus-szakaszából. (A valóságban ennél bonyolultabb és gazdaságosabb módon végzik a törléseket. Például csak egy callback-et regisztrálnak (`call_rcu(oldElem, free)`), ami a megfelelő időpontban végzi a törlést, így nem kell elemenként végigvárnunk 1-1 szinkronizációt.)

```
//C code

Elem* p;
mutex_t update_side_mutex;

void writer(void)
{
    lock_mutex(&update_side_mutex);

    Elem* oldElem = rcu_dereference(p);

    Elem* newElem = /* create new elem */
    rcu_assign_pointer(p, newElem);

    synchronize_rcu();
    free(oldElem);

    unlock_mutex(&update_side_mutex);
}
```

5.2. Kódrészlet. RCU írás

5.2. RCU megvalósítások

A Linux kernel természetesen tartalmaz egy RCU megvalósítást, amit gyakorlatilag már a kernel minden részében használnak [22]. Kernel módban sok olyan eszköz és megkötés áll a fejlesztők rendelkezésére ami felhasználói módban nem, így tehát felmerült a probléma, hogy készítsenek egy felhasználói szintű RCU könyvtárat is. Egy ezeket tárgyaló alapmű a [18]. Főként ez alapján szeretnék bemutatni egy-két elterjedt RCU megvalósítást.

5.2.1. Megjegyzés. Az RCU primitíveket lehetséges lenne atomi módon írt `shared_ptr`-ekkel is megvalósítani, azonban a klasszikus megvalósítások nagyobb olvasó-oldali performanciát képesek elérni, és nem utolsó sorban C nyelven is „könnyen" megvalósíthatóak. (A `shared_ptr` referencia számlálása szálbiztos, azonban magának a pointernek az írása/olvasása csak akkor az, ha az `atomic_load` és társainak `shared_ptr`-re specializált változatait használjuk. A szabvánnyal foglalkozó bizottságnál már indítványozták az `atomic_shared_ptr` osztály bevezetését, ami ezt transzparens módon fogja támogatni.)

5.2.2. Definíció. *Nyugalmi állapot (Quiescent state): Egy szál nyugalmi állapotban van, ha nincs benne egy olvasó-oldali kritikus-szakaszban.*

5.2.3. Definíció. *Türelmi periódus (Grace period): Ha egy időintervallumban minden szál legalább egyszer nyugalmi állapotban van, akkor az időintervallumot türelmi periódusnak nevezzük.*

5.2.4. Következmény. Igazak a következő állítások:

- Minden olyan olvasó-oldali kritikus-szakasz ami egy adott türelmi periódus előtt kezdődött, az a türelmi periódus végére már befejeződik.
- A különböző türelmi periódusok átfedhetnek egymással.
- Minden olyan időszak, amelyben van türelmi periódus önmaga is türelmi periódus.
- Ha minden olvasó-oldali kritikus-szakasz hossza véges, akkor minden türelmi periódus véges időn belül befejeződik (Akkor is ha folyamatosan kezdődnek újabb és újabb kritikus-szakaszok).

5.2.5. Következmény. Ha egy elem kiláncolása és törlése között várunk egy türelmi periódust, akkor biztosítva van, hogy már egyik szál sem olvassa. Definiáljuk tehát úgy a `synchronize_rcu` műveletet, hogy éppen ezt csinálja.

5.2.1. Nem-preemptív változat

Egy nem-preemptív rendszeren (azaz olyanon, ahol a task-ok nem szakítódnak félbe az ütemező által) elég lehet a következő megvalósítás [23]. Az `rcu_read_lock`-nak és az `rcu_read_unlock`-nak itt csak dokumentációs szerepe van. A `synchronize_rcu` eljárás itt annyit tesz, hogy beütemezi önmagát egymás után az összes processzorra. Könnyű látni, hogy a végére minden szál kijött legalább egyszer a kritikus-szakaszaiból. A legtöbb rendszer preemptív, tehát ez nem lesz elég nekünk.

```
//C-like pseudocode

void rcu_read_lock(void){}

void rcu_read_unlock(void){}

void synchronize_rcu(void)
{
    int cpu;
    for_each_cpu(cpu)
        schedule_current_task_to(cpu);
}
```

5.3. Kódrészlet. Nem preemptív rendszereken alkalmazható RCU primitívek

```
//C-like pseudocode (without proper synchronization)

void rcu_read_lock(void){}

void rcu_read_unlock(void){}

uint64_t globalCounter=1;

__thread_local uint64_t myCounter = 1;

void rcu_quiescent_state(void)
{
    myCounter = globalCounter;
}

mutex sync_mutex;

void synchronize_rcu(void)
{
    mutex_lock(&sync_mutex);

    ++globalCounter;
    for_each_threads_counter(itsCounter)
        while(itsCounter != globalCounter);

    mutex_unlock(&sync_mutex);
}
```

5.4. Kódrészlet. Nyugalmi-állapot-alapú RCU primitívek

5.2.2. QSBR: Nyugalmi-állapot-alapú újrahasznosítás

A Quiescent-State-Based Reclamation RCU lényegében a leggyorsabb valóságban is alkalmazható RCU változat. Az `rcu_read_lock` és az `rcu_read_unlock` primitívek itt is az üres utasításra fordulnak le, mégis használható preemptív rendszereken. Az a trükk benne, hogy szüksége van arra, hogy az olvasó szálak (egy `rcu_quiescent_state(void)` hívással) időnként bejelentsék, hogy nyugalmi időszakban vannak (egy hosszabb nyugalmi időszak esetén akár többször is). Ha pedig blokkoló hívást végeznek, akkor külön jelenteniük kell, hogy most egy hosszú nyugalmi időszakba kerülnek. A blokkoló hívás után pedig azt, hogy kiléptek onnan. Ez elég sok kényelmetlenséget ró a programozóra de teljesítmény-kritikus rendszereken elfogadható lehet.

A megvalósítás vázlata Van egy globális atomi számláló, ami kezdetben 1 értékű. Minden olvasó szálnak van egy-egy szál-lokális (thread-local) atomi számlálója, ami mindig kisebb vagy egyenlő, mint a globális számláló. Az `rcu_quiescent_state` eljárás beállítja az adott szál számlálóját a globális számláló értékére. A `synchronize_rcu` eljárás pedig növeli a globális számlálót, majd addig olvassa az egyes szálak számlálóit, amíg mindegyik értéke el nem érte a globális számláló értékét. Ez azt jelenti, hogy eltelt egy türelmi periódus. Ez a megoldás csak 64 bites rendszereken használható, mivel 32 bites számláló esetén problémát jelenthet a gyakori túlcsordulás.

5.2.2.1. Megjegyzés a Linux kernel memória modelljéről

Jelenleg a legkomolyabb RCU megvalósítások (pl: [18], illetve maga a Linux) mind a Linux kernel memória modelljét [25] [26] alkalmazzák. Ez egy főleg különböző barrierekre épülő modell, aminek az egyes primitívjei GCC builtin-ek vagy inline assembly segítségével vannak megvalósítva (Ettől még természetesen egy cross-platform modellről van szó.). Emellett volatile típusra kasztolásokat is használnak olvasáskor és íráskor, ami az optimalizálást kikapcsolja, szemben a c++11 megfelelő atomi műveleteivel, amik egyes optimalizálásokat még engednek.

5.2.6. Példa. `smp_mb()`: Garantálja, hogy minden memória elérés a barrier előtt korábban fog látszani minden cpu számára, mint minden memória elérés a barrier után. [25]

5.2.7. Megjegyzés. Ajánlom a Linux Weekly News hivatkozott cikkét, ami arról szól, hogy a Linux-ban miért nem sietik el a C11-es atomi műveletekre való átállást [27]. Ennek egyik oka az lehet, hogy a Linux kernel és a C(++)11 szinkronizációs primitívjei között nincs egyszerű/egyértelmű megfeleltetés.

5.2.2.2. Megvalósítás C++11-ben

Ide nem szúrom be az egész megvalósítást, csak a lényeges tapasztalatokat foglalom össze. A hivatkozott [18] cikkből indultam ki. A C++ memória modelljére való portolás nehezebb volt mint elsőre hittem, főként az előző megjegyzésben tárgyaltak miatt. A megvalósításom logikai helyességbizonyítására nem vállalkozom, erre már íródott egy cikk [28].

Kibővített QSBR A korábban leírt QSBR vázlatnál kicsit bonyolultabb módszert programoztam le, olyat, ami megengedi, hogy egy szál offline vagy online állapotba kerülhessen. Ennek akkor van haszna,

ha az egyes szálak dinamikusan jöhetnek létre / szűnhetnek meg, vagy esetleg azt szeretnénk, hogy a szinkronizálás ne várja meg az olvasó-oldali kritikus-szakaszokon kívüli hosszabb blokkoló műveleteket. Ahhoz, hogy ez a módszer működhessen, szükség van a szekvenciális konzisztencia fogalmára, mint azt a hivatkozott [28] cikk is állítja. A cikk éppen az ehhez megfelelő logikai eszközök hiányában nem tárgyalja ezt a bővített módszert. Én se bizonyítottam a helyességét, csak Relacy-vel validáltam.

A szekvenciális konzisztencia szükségessége Számomra ez volt a megvalósítás legérdekesebb része, ráadásul a szükségességére a Relacy használatával jöttem rá (később a hivatkozott [28] cikkben is olvastam). Amikor offline állapotból jött online állapotba egy szál, majd elkezdte olvasni az rcu-védett adatstruktúrát, akkor ez versenyhelyzetbe került az adatstruktúra törlésével, így törölt-memória olvasás történt. Ez úgy volt lehetséges, hogy a törlő szál még offline-nak olvasta az olvasó szál státuszát.

5.2.8. Állítás. *A hiba eltűnik, ha szekvenciálisan konzisztensnek jelöljük a következő műveleteket:*

- *A szál-állapot online-ra módosítása az olvasó szálon (eredetileg release volt)*
- *Az rcu-pointer olvasása az olvasó szálon (eredetileg consume volt)*
- *Az rcu-pointer cseréje (exchange) az író szálon (eredetileg acquire-release volt)*
- *A szál-állapot olvasása az író szálon (eredetileg acquire volt)*

Bizonyítás. Ilyenkor ha az olvasó szál az rcu-pointer korábbi állapotát olvassa, mint a csere utáni állapot, akkor a műveletek abszolút sorrendje az állításban szereplő sorrend. Az első a másodikkal, a harmadik pedig a negyedikkel van korábban történt relációban (hisz sequenced-before relációban állnak). A 2. és a 3. között pedig a módosítási sorrend határozza meg a globális sorrendet. (Indirekt tegyük fel, hogy a globális sorrend fordítva van: 3.->2. Ekkor a 2. utasításnak a 3. utasítás eredményét kell látnia, tehát nem lehet igaz az, hogy korábbi értéket olvas.) Így a szekvenciális konzisztencia láthatósági előírásai miatt az utolsó műveletnek látnia kell az első eredményét, vagyis, hogy már online a szál. □

5.2.9. Megjegyzés. A Linux kernel memória modelljében elég lenne egy `smp_mb()` hívás az állapot online-ra módosítása után.

Konklúzió Bár ez nem eredményez „színtiszta” C++11-es megvalósítást, mégis azt ajánlanám az érdeklődőknek, hogy ha amellett döntenek, hogy C++-ban fognak RCU-t használni, akkor a megfelelő library megjelenéséig inkább használják a C-ben írt liburcu-t [19]. Esetleg ezt csomagolják be megfelelő C++ osztályokba. Amennyiben mégis saját megvalósítás írását fontolgatják, semmiképp ne kezdjék el megfelelő automatikus verifikáló eszköz (pl: Relacy Race Detector) nélkül. Ezen kívül szükség lehet a logikai helyességbizonyításra is, mivel a Relacy sokszor nem az összes lehetséges lefutást járja be (érdemes a korábban bemutatott módon magasabbra állítani a tesztesetek számát).

5.2.3. Egyéb RCU megvalósítások

Léteznek még egyéb RCU megvalósítások, amik esetleg nagyobb terhet rónak az olvasó szálra, viszont cserébe nem igényelnek különleges kódszervezést (mint az időnkénti `rcu_quiescent_state` hívás) [18]. Ezek alapozhatnak barrierekkel / atomi műveletekkel végzett szinkronizációra vagy szignálokra is.

6. fejezet

Esettanulmány: RCU alapú átméretezhető hasítótábla

Az ezen fejezet témájául szolgáló adatszerkezetet a hivatkozott [29] cikkből és az ezt összefoglaló Linux Weekly News cikkből [30] vettem és ültettem át C++14-re. Az adatszerkezet motivációjáról korábban írtam 5.1.

6.1. Tulajdonságok

A jelenleg tárgyalt hasítótábla a következő tulajdonságokkal rendelkezik:

- Az olvasó oldal teljesen lock-mentes és várakozás-mentes (beszúrás, módosítás, törlés és átméretezés közben is folyamatosan lehet olvasni).
- A beszúrás, módosítás, törlés vödrönkénti lock-okkal működik (tehát potenciálisan több is történhet egyszerre).
- Átméretezés közben nem lehet beszúrni, módosítani vagy törölni (és természetesen egyszerre csak egy átméretezés történhet). Ez a feltétel könnyen gyengíthető [29].

6.2. Ábrázolás

Az alábbi két fejezetben bemutatom az RCU lista és az RCU alapú hasítótábla felépítését.

6.2.1. RCU Lista

A hasítótábla vödrei (buckets) egyszeresen láncolt, fejelemes RCU-listával vannak ábrázolva. A lista-elemek egy értéket és egy következő pointert tartalmaznak. Maga a lista pedig a fej-elemet (ennek az értékét nem használjuk, hanem csak az egyszerűbb algoritmusírásban segít) és egy íráskor használandó mutexet tartalmaz. A rekurzív mutex kicsit nagyvonalú, nélküle is meg lehet oldani, csak azért hasznos, mert így könnyebben tudják lock-védett metódusok hívni egymást.

```
template<typename T>
struct list_elem
{
    T value;
    rcu::ptr<list_elem> next;
};

template<typename T>
struct list
{
    list_elem<T> head;
    std::recursive_mutex writeMutex;
};
```

6.1. Kódrészlet. RCU lista struktúrája

6.2.2. RCU Hasítótábla

A hasítótábla szerkezete is egyszerű. Fő elme egy rcu-pointer ami a tényleges táblára mutat (ami egy `std::vector`). Ezenkívül tartalmaz egy (később ismertetett) shared-mutexet és egy size atomi változót, ami a hasítótáblában lévő elemek számát tárolja és arra használatos, hogy megállapítsuk mikor van túlságosan tele a tábla és szorul átméretezésre.

```
template<typename K, typename V>
class hash_table
{
    using Bucket = rcu::list<std::pair<K,V>>;

    rcu::ptr<std::vector<Bucket>> table_;
    std::shared_timed_mutex writeMutex_;
    std::atomic<size_t> size_;
}
```

6.2. Kódrészlet. RCU hasítótábla struktúrája

6.2.2.1. Shared-mutex

A shared-mutex (másnéven readers-writer/olvasók-író mutex) egy olyan mutex, amit két féleképpen lehet lock-olni:

- Tehetünk rá shared-lock-ot, ilyet egyszerre bármennyi szál tehet a mutexre.
- Illetve tehetünk rá unique-lock-ot, ilyenből egyszerre csak egy lehet a mutexen, ráadásul kizárja azt, hogy shared-lock legyen rajta.

Klasszikus felhasználása, hogy egyszerre több olvasót enged egy adatszerkezeten, viszont csak egy írót és írás közben nem enged olvasni.

Mi kicsit másra használjuk: egyszerre több beszúrást, módosítást, törlést engedünk, viszont csak egy átméretezést, és az kizárja a többit. (Az olvasás lock nélkül történik.)

Ez az adatszerkezet jelenleg `std::shared_timed_mutex`-ként érhető el a C++14-ben. A C++17-be már szándékozzák bevenni az egyszerűbb `std::shared_mutex`-et is. Ha az megjelenik akkor áttérhetünk rá. Ez az egy mutex az oka annak, hogy az adatszerkezetünk C++14-et igényel, nem csak C++11-et. A [29] cikk szerint a probléma megoldható shared-mutex nélkül is, például úgy, hogy az átméretezés lock-olja az összes vödör saját mutexét.

6.3. Lock-ok és olvasó-oldali kritikus szakaszok

6.3.1. Az RCU lista esetén

A listához tartoznak kereső függvények, ezek használatához elégséges, ha a következő három feltétel közül legalább egy teljesül (ezek biztosítása a hívó felelőssége):

- Legyünk olvasó-oldali kritikus szakaszban.
- Birtokoljuk a lista write-mutexét.
- Birtokoljuk a listát birtokoló hasítótábla write-mutexét (ha van ilyen).

A módosító függvények lock-olják a lista write-mutexét.

Lista-elemek tartalmának módosításához az egész lista-elemet cseréljük. Az régi elem kiláncolódik, az új pedig beláncolódik egyetlen exchange lépésben.

6.3.2. A hasítótábla esetén

Az RCU hasítótábla a szokásos műveletekkel rendelkezik: get (elem lekérése), set (elem hozzáadása vagy felülírása), remove (elem eltávolítása). Privát member-függvényei közül a fontosabbak: expand (tábla átméretezése a duplájára), shrink (tábla átméretezése a felére). Részletekért lásd a [34] git tárolót.

A get, set és remove metódusokat csak olvasó-oldali kritikus-szakaszból szabad hívni (az ebbe való belépéshez, mint korábban említettem nem kell igazi lock művelet). Az olvasó-oldali kritikus-szakaszra azért van szükség az író műveleteknél is, mivel a `table_` rcu-pointer szempontjából ők is olvasók.

A set és remove függvények shared-lockolják a tábla mutexét, míg az expand és shrink függvények unique-lock-ot tesznek rá. Ezek azért kellenek, hogy átméretezés közben ne lehessen beszúrni, vagy törölni.

A konstruktor és a destruktor hívásakor csak a hívó szál használhatja az adatstruktúrát.

6.4. Megvalósítás

Olvasás Az olvasás a következőképp zajlik:

1. Elmentjük a tábla pointer jelenlegi értékét (hiszen ez változhat, ha közben történik egy átméretezés).
2. A kulcs hash-elésével kiszámítjuk a neki megfelelő vödröt.
3. A vödörben megkeressük az elemet.
4. Ha van ilyen, visszaadjuk az értékére mutató pointert, ha nincs akkor nullptr-t.

```
const V* get(const K& key) const
{
    assert(debug::isReadLocked());
    auto& table  = *table_.consume();
    auto& bucket = table[hash_(key) % table.size()];
    auto* elem   = bucket.find([&](const KV& kv){
        return kv.first == key;
    });
    return elem == nullptr ? nullptr : &elem->value.second;
}
```

6.3. Kódrészlet. Olvasás az RCU alapú hasítótáblából

Beszúrás, törlés, módosítás A beszúrás, törlés, módosítás műveletek hasonlóan működnek az olvasáshoz, azzal a kivétellel, hogy ha túl- vagy alul-telítődik a hasítótábla, akkor meghívják a megfelelő átméretező függvényt (lásd később).

Double checked locking a tábla zsugorítása és növelése előtt A zsugorítás és növelés metódusok hívása double-checked-locking pattern szerűen működik. Először a unique-lock nélkül megnézzük, hogy túl/alul van-e telítődve a tábla, majd ha igen, akkor megszerezzük a lock-ot. Majd a metódus hívása előtt újra-ellenőrizzük a feltételt. A lock-on belüli feltételre azért van szükség, hogy nehogy több különböző szál is átméretezze a táblát, így többszörösen megnövelődjön/összezsugorodjon. A lock-on kívüli feltétel pedig optimalizálás: ha nem szükséges, ne szerezzük meg a lock-ot.

Tábla összezsugorítása Az tábla összezsugorítása viszonylag egyszerű művelet. A felére való zsugorítást implementáltam, de más szorzókra is könnyen megvalósítható lenne.

1. Mentsük el a régi táblára mutató pointert. (Itt nem fenyeget a felülírás veszélye, de az `rcu::ptr`-t nem lehet közvetlenül használni, ezért muszáj külön elmentenünk.)
2. Elkészítjük az új táblát. Könnyen belátható, hogy minden új táblabeli vödörbe két régi vödör elemei fognak hasítódni.
3. Az új tábla minden vödrének fejeleme után láncoljuk be az első régi vödröt ami bele hasítódó elemeket tartalmaz (A régi táblát ne módosítsuk).
4. Ezeknek a listáknak a végére láncoljuk oda a második hozzá tartozó régi vödröt.
5. Cseréljük le a táblát az újra.
6. Szinkronizáljunk (várjuk meg amíg minden olvasó abbahagyja a régi tábla olvasását).
7. Szabadítsuk fel a régi tábla tárterületét.

```
void shrink_to_half_impl()
{
    Table* oldTable = table_.consume();
    Table* newTable = new Table(oldTable->size()/2);
    for(size_t i = 0; i<newTable->size(); ++i)
    {
        (*newTable)[i].head.next.exchange((*oldTable)[i].head.next.consume());
        (*newTable)[i].last()->next.exchange(
            (*oldTable)[i+newTable->size()].head.next.consume());
    }
    table_.exchange(newTable);
    rcu::synchronize();
    delete oldTable;
}
```

6.4. Kódrészlet. RCU alapú hasítótábla összezsugorítása a felére

Tábla megnövelése A tábla megnövelése egy több iterációból álló művelet.

1. Mentsük el a régi táblára mutató pointert.
2. Elkészítjük az új táblát. Könnyen belátható, hogy minden régi táblabeli vödör tartalma két-két új vödörbe fog kerülni.
3. Minden új vödörhöz láncoljuk hozzá a hozzá tartozó régi vödör tartalmát az első olyan elemtől kezdve, ami ő belé hasítódik.
4. Cseréljük le a táblát az újra. (Így egy helyesen használható táblát kapunk, csak még egyes vödrök néhány nem oda tartozó elemet is tartalmazni fognak.)
5. Szinkronizáljunk.
6. A régi tábla minden vödrére hajtsunk végre egy szétválogató lépést.
7. Szinkronizáljunk.
8. Ha a 6. lépésben történt módosítás, akkor menjünk vissza a 6. lépésre és onnan folytassuk a programot.
9. Szabadítsuk fel a régi tábla tárterületét.

Egy szétválogató lépés annyit tesz, hogy megkeressük a régi vödör első olyan összefüggő részsorozatát, ami nem abba az új vödörbe való, mint az első elem és kiláncoljuk. A régi tábla vödreit arra használjuk, hogy jelöljék hol tartunk a szétválogatásban. Ez után a lépés után a megfelelő régi vödör head.next-jét az első kiláncolt elemre állítjuk.

A 3, 6, 7 lépések tartalma között szoros összefüggés van. A 3. lépésben az, hogy minden új vödörhöz az első bele-való elemet láncoljuk azért jó nekünk, mert így az első szétválasztási lépésben (6.) nem tudnak eltévedni a korábbi olvasók. (Mindig egy olyan elem next pointerét állítjuk át, akit már csak olyan olvasók látnak, akik csak a vele egy (új) vödörbe valókra kíváncsiak.) A szinkronizálás (7.) azért kell, hogy megvárjuk, hogy ismét egy hasonló helyzetbe kerüljünk, mint a 3. lépés után. Lényegében azért van ez az egész iteráció, mert ha egyszerre több részlistát láncolnánk át egy listán belül, akkor eltévedhetnének a korábbi olvasók.

```
void expand_to_double_impl()
{
    Table* oldTable = table_.consume();
    assert(oldTable->size() <= std::numeric_limits<size_t>::max() / 2);

    Table* newTable = new Table(2*oldTable->size());

    for(size_t i = 0; i<oldTable->size(); ++i)
    {
        link_new_bucket_to_first_fitting(*oldTable, i, *newTable, i);
        link_new_bucket_to_first_fitting(*oldTable, i, *newTable, oldTable->size() + i);
    }

    table_.exchange(newTable);
    rcu::synchronize();

    while(unzip_step(*oldTable, newTable->size()))
        rcu::synchronize();

    delete oldTable;
}
```

6.5. Kódrészlet. RCU alapú hasítótábla megnövelése a duplájára

6.5. Mérések

6.5.0.1. Tesztelés módja

A köv. tesztet hajtottam végre (x86-64-en 1 író + 1 olvasó szálon, arm-v7-en 1 író + 3 olvasó szálon):

- Író szál: 1000 000 elem hozzáadása, majd fordított sorrendben eltávolítása a hasítótáblából.
- Olvasó szál(ak): Amíg az írás folyik, addig folyamatosan olvas(nak) (keres(nek) egy álvéletlen értéket a táblában).

A tesztprogramot minden esetben 10-szer futtattam egymás után.

6.5.1. Jelölés. *A táblázat a következő jelöléseket használja:*

- *átlag(átlag): Az átlagos olvasási idők átlaga a 10 futtatás között.*
- *átlag(max): Az maximális olvasási idők átlaga a 10 futtatás között.*
- *max(max): Az maximális olvasási idők maximuma a 10 futtatás között.*

A mérés célja elsődleges sorban az volt, hogy megmutassam, hogy az RCU technika valóban lecsökkenti az olvasó szálak maximális várakozási idő / átlagos várakozási idő arányát. (Lényegében egyenletesen haladnak az olvasó szálak, semmire sem kell várniuk, legfeljebb ha az operációs rendszer mást ütemez be helyettük.) Másodlagos volt annak a mérése, hogy meddig tart az átlagos olvasási idő (ezt valószínűleg mindkét megvalósítás esetén lehetne még javítani és nem feltétlenül azonos mértékben).

6.5.0.2. A tesztelt hasítótáblák

Ezt a két hasítótáblán teszteltem:

- RCU hasítótábla (RCU): Az e fejezetben ismertetett hasítótábla.
- (Vödrönként) lock-olt hasítótábla (LOCK): Olyan hasítótábla ami az olvasás/beszúrás/törlés műveletekhez vödrönkénti shared-mutex-et használ, míg az átméretezéshez az egész táblát lock-olja egy globális shared-mutex-el (lásd: [34] git tároló). (Így minden művelet elvégzéséhez 2 mutexet kell megszereznie.)

olvasási idő x86-64-en	**RCU**	**LOCK**	**LOCK/RCU**
átlag(átlag)	243 ns	1 374 ns	**5,7**
átlag(max)	192 324 ns	53 239 366 ns	276,8
átlag(max) / átlag(átlag)	792	38 758	**49,0**
max(max)	1077 417 ns	55 442 839 ns	51,5
max(max) / átlag(átlag)	4 434	40 362	**9,1**

6.1. táblázat. Hasítótábla olvasási idők x86-64-en (1 író és 1 olvasó szál, ns=nanoszekundum)

olvasási idő x86-64-en	**RCU**	**LOCK**	**LOCK/RCU**
átlag(átlag)	2 357 ns	3 739 ns	**1,6**
átlag(max)	9 704 596 ns	292 919 943 ns	30,2
átlag(max) / átlag(átlag)	4 118	78 337	**19,0**
max(max)	20 045 834 ns	332 816 198 ns	16,6
max(max) / átlag(átlag)	8 506	89 007	**10,5**

6.2. táblázat. Hasítótábla olvasási idők arm-v7-en (1 író és 3 olvasó szál)

6.5.0.3. Értékelés

A mérésből levont következtetések egy része független attól, hogy melyik platformon futtatuk a programot, illetve, attól is, hogy a 10 futtatás átlagát, vagy maximumát vettük, ezért tekintsük most az x86-64-es futtatások átlagát.

Az RCU hasítótáblában a leglassabb olvasás 792-szer volt lassabb az átlagnál, míg a vödrönként lock-olt táblánál ez az arány 38 758 volt. Tehát a futási időbeli növekedés arányában 49-szer volt nagyobb a lock-olt táblánál. Ez nyilván amiatt lehet, hogy azt az átméretezés alatt nem lehet olvasni. Annak, hogy az RCU-nál miért van szintén jelentős kiugrás az olvasási idők között, szerintem az lehet az oka, hogy az operációs rendszer átmenetileg szüneteltette, vagy másik magra helyezte az adott szál futását.

Az RCU átlagos olvasási ideje 5,7-szer volt gyorsabb a lock-olt változaténál x86-64-en, míg ez az arány csak 1,6 volt arm-v7-en. Ez számomra azt mutatja, hogy az RCU változattal valószínűleg tényleg gyorsabb olvasási időket lehet elérni. Az arm-v7-en tapasztalt gyorsulás azért lehet kisebb mértékű, mert ott költségesebbek lehetnek a különböző atomi műveletek.

7. fejezet

Összefoglalás

E dolgozatban áttekintettem a lock-szegény adatszerkezetek C++11 nyelven való létrehozásának főbb állomásait, beleértve a matematikai helyesség-bizonyítást. Arra törekedtem, hogy ez egy az alacsony szintű szinkronizálásról szóló, kellően átfogó dokumentum legyen. Igyekeztem a C++11 memóriamodelljének egy teljes formális áttekintését adni.

Megindokoltam miért is van szükség a használt technikai és matematikai eszközökre. Ismertettem a köztudatban tévesen elterjedt programozási minták hibáit. Matematikai pontossággal összefoglaltam a szálak közötti szinkronizálást biztosító egyes relációkat. Az összefoglalás alapján beláttam egyszerű tételeket, amik segítettek az adatszerkezetek tervezésekor. Felsoroltam a lock-free adatszerkezetekhez kapcsolódó alapvető fogalmakat, majd bemutattam egy egyszerű lock-free sor adatszerkezetet. Az adatszerkezet implementálásakor szerzett tapasztalatok alapján rámutattam a matematikai helyességbizonyítás szükségességére, és ismertettem is rá egy példát. Bemutattam a számítógépes validálás miként jét a Relacy versenyhelyzet kereső segítségével. Táblázatos formában összehasonlítottam a különböző szinkronizálási módokat használó sorok futásidejét. Kitekintést nyújtottam az olvasás közben módosítható (RCU) adatszerkezetekre, amik még nem teljesen honosodtak meg a C++11 nyelven. Áttekintettem a portolás nehézségeit egy RCU megvalósítás C++11-re költöztetésével, majd egy nagy-teljesítményű konkurrens hasítótábla C++11-es implementációjával. Mérésekkel alátámasztottam, hogy az RCU alapú hasítótábla egyenletesebb és gyorsabb olvasási időket garantál, mint egy egyszerű vödrönként lockolt hasítótábla.

Bízunk benne, hogy a dokumentum hozzájárul ahhoz, hogy az érdeklődők könnyebben elsajátítsák a C++11 memóriamodelljét, és amennyiben lock-szegény adatszerkezetet fejlesztenek ki, akkor többféle segéd-eszközt is használjanak a helyesség ellenőrzésére, köztük a matematikai bizonyítást. Ez reményeink szerint több helyes többszálú programot eredményez.

A. függelék

Függelék: További információk a Relacy race detector használatáról

Erre a függelékre azért van szükség, mert az RRD a szakdolgozat írása során nélkülözhetetlen eszköznek bizonyult, azonban „dokumentációja” messze nem teljes.

A.1. A Relacy program alkalmazása egy meglévő algoritmus/adatszerkezet teszteléséhez

Ez a szekció a Relacy 2.4-es verziójáról szól. A programok kódján a következő módosítások szükségesek a Relacy használatához:

- Az <atomic> és a <thread> headerek helyett a `<relacy/relacy_std.hpp>` headert kell includeolni.
- Minden *std* névtérbeli atomi vagy szinkronizációs operációt az *rl* névtérbelire ugyanilyen nevűre kell cserélni.
- Minden több szálról elért nem-atomi változó `T` típusát `rl:var<T>`-re kell cserélni. Ez a típus részt vesz a data-race vizsgálatban.
- Az írt vagy olvasott `rl::var`-okat és `rl::atomic`-okat ($) végződéssel kell ellátni. Ez egy makró, ami rögzíti a jelenlegi sor számát. Azokban a műveletekben nem szükséges megadni, ahol explicit memóriamodellt adunk meg, mert a Relacy-ben a memóriamodellek nevei is makrók, amikben ez már benne van.
- A tesztet pedig egy lejjebb látható formátumú `rl::test_suite` leszármazott osztályként kell megvalósítani.

Bár a program kétségkívül igényel változtatásokat, de az ezekkel való munka és hibalehetőség elenyésző más modell ellenőrzőkhöz képest, ahol a programot újra kell írni egy speciális nyelven. Ezeket a változtatásokat vélhetően automatikusan is el lehetne végeztetni egy programmal. Egyébként a Relacy nem arra

készült, hogy minden cég kódbázisát ezzel vizsgálják, hanem, hogy maguk a lock-free adatszerkezetek készítői (köztük a program írója) validálják ezzel a szerkezeteket.

```
template<typename T, int Capacity>
class queue
{
    T mData[Capacity+1];
    alignas(CacheLineSize) rl::atomic<int> mBase;
    alignas(CacheLineSize) rl::atomic<int> mNext;
public:
    queue()
    {
        mBase($) = 0;
        mNext($) = 0;
    }
    bool push_t1(const T& value)
    {
        int base = mBase.load(rl::memory_order_acquire);
        int next = mNext.load(rl::memory_order_relaxed);
        int before_base = wrap(base-1);
        if(next != before_base)
        {
            mData[next]($) = value($);
            mNext.store(wrap(next+1),rl::memory_order_release);
            return true;
        }
        return false;
    }
    bool pop_t2(T& out)
    {
        int base = mBase.load(rl::memory_order_relaxed);
        int next = mNext.load(rl::memory_order_acquire);
        if(base != next)
        {
            out($) = mData[base]($);
            mBase.store(wrap(base+1), rl::memory_order_release);
            return true;
        }
        return false;
    }
private: [...]
};
```

A.1. Kódrészlet. A queue program hibakeresésre átírt változata (queue.h)

```cpp
#include <relacy/relacy_std.hpp>
#include <memory>
#include "queue.h"

using TestType = rl::var<int64_t>;
constexpr int TestSize = 10;
using QueueType = queue<TestType,TestSize>();

struct race_test : rl::test_suite<race_test, 2>
{
    std::unique_ptr<QueueType> q;

    void before()
    {
        q = std::make_unique<QueueType>(); //c++14
    }

    void thread(unsigned thread_index)
    {
        if (thread_index == 0)
            t1();
        else
            t2();
    }

    void t1()
    {
        for(int i = 0; i<5*TestSize; i++)
            while(!q->push_t1(i));
    }

    void t2()
    {
        for(int i = 0; i<5*TestSize; i++)
            TestType j=-1;
            while(!q->pop_t2(j));
            assert(i == j($));
    }

    void after()
    { }

    void invariant()
    { }
};

int main()
{
    rl::simulate<race_test>();
}
```

A.2. Kódrészlet. A queue program hibakeresésre átírt változata (main.cpp)

A.1.1. Megjegyzés. Előfordulhat, hogy túl hosszú futású tesztek esetén a program live-lock-ot jelez.

A.2. A tesztfuttatások számának növelése

Néha egy hiba nem jön elő kevés teszt futtatás alatt ezért növelni akarjuk a futtatások számát. Ezt így tehetjük meg:

```
int main()
{
    rl::test_params params;
    params.iteration_count = 100000;
    rl::simulate<race_test>(params);
}
```

A.3. Globális, lokális-statikus és szál-lokális változók kezelése

A Relacy program lényegében csak azokat a változókat tudja megfelelően kezelni, amik legkorábban egy-egy tesztelési ciklus elején jönnek létre és legkésőbb ugyanazon tesztelési ciklus végén megszűnnek. Az előbb említett típusú változók nem ilyenek. Ha nem tudunk, vagy nem akarunk ilyen változóktól mentes programot írni, akkor a következő szabályokkal könnyedén átalakíthatjuk a programot a Relacy-vel való használatra:

- Lokális-statikus (static egy eljáráson belül) változók: Használjunk helyettük globálisokat és kövessük az utolsó pont utasításait. (Vigyázat, ez nem teljesen lesz ekvivalens az eredeti viselkedéssel.)
- Szál-lokális változók: Használjunk helyettük egy globális tömböt, aminek annyi eleme van ahány szál lehet a teszt-osztályunkban. Az adott szálhoz tartozó változó lekérésére írhatunk egy függvényt (lásd: példa), így a használat helyein csak egy myVar → myVar() cserét kell majd végrehajtani. A tömb létrehozását és törlését az utolsó pontnak megfelelően végezzük. Ennek a tömbös módszernek az is az előnye, hogy így könnyű egy adott szál-lokális változó összes példányán végig-iterálni.
- Globális változók: Alakítsuk át őket globális *pointer* változókká és írjunk egy createGlobals és egy deleteGlobals eljárást, amelyek inicializálják, illetve törlik őket és amelyeket meghívunk a teszt-osztály before, illetve after metódusában.

A példát lásd a következő oldalon.

A.3.1. Példa.

```
int globalCounter = 1;
thread_local int myCounter = 2;

void foo()
{
    static int staticCounter = 3;
    std::cout << myCounter << " " << staticCounter << " "
        << globalCounter << std::endl;
}
```

A.3. Kódrészlet. Programkód Relacy-hoz való átalakítás előtt

```
constexpr int ThreadCount = 4;

int* globalCounter;
int* threadLocalCounters;
int* staticCounter;

void createGlobals()
{
    globalCounter = new int(1);
    threadLocalCounters = new int[ThreadCount];
    for(int t=0; t<ThreadCount; ++t)
        threadLocalCounters[t] = 2;
    staticCounter = new int(3);
}

void deleteGlobals()
{
    delete globalCounter;
    delete [] threadLocalCounters;
    delete staticCounter;
}

int& myCounter()
{
    return threadLocalCounters[rl::thread_index()];
}

void foo()
{
    std::cout << myCounter() << " " << *staticCounter << " "
        << *globalCounter << std::endl;
}
```

A.4. Kódrészlet. Programkód Relacy-hoz való átalakítás után

B. függelék

Függelék: Tesztkörnyezet

Manapság szinte minden személyi számítógépben több magos processzor található, azonban ezek közül sok csak két magos, és szinte mind x86-64 (=AMD64) architektúrájú.

A két mag kevés ahhoz, hogy egy több olvasóból és egy vagy több íróból álló rendszert külön magokon futtassunk rajta. Pedig ez fontos lenne, hiszen így tudna a lehető legtöbb versenyhelyzet előjönni.

Az x86-64 architektúra sem a legszerencsésebb választás egy többszálú keresztplatformos C++ alkalmazás teszteléséhez, hiszen ez kifejezetten erősen szinkronizált (minden írás és olvasás alapból release és acquire szemantikájú) [16]. Emiatt sok olyan hiba nem is fordulhatna elő rajta, ami más platformokon szinte azonnal látszik. Hasonló okokból a különböző szemantikájú műveletek közötti performancia különbség sem látszik ezen a platformon.

Az előbbi okok miatt én otthoni kísérletezéshez az ARM(v7) architektúrát ajánlom. Megvásárolható különböző barkácsoláshoz szánt system-on-chip szettekben, azonban ezek sokszor gyengébb teljesítményűek, kevés magosak és még rendes gépházat se adnak hozzájuk. A legkézenfekvőbb választás a gyakorlatilag minden háztartásban megtalálható okos-telefon vagy tablet. Ezekben általában legalább 4, de nem ritkán 6-8 mag található. A teljesítményük talán nem olyan nagy, mint az asztali gépeknek, de ez itt nem is számít.

B.1. Androidos telefon felhasználása többszálú programok teszteléséhez

Bár meglepő lehet, de egy egyszerű (USB → Micro-USB) adatkábel segítségével akár root-olatlan telefonon is tudunk natív ARM binárisokat futtatni [31].

Környezet feltelepítése Szükségünk van egy Android SDK-ra és egy Android NDK-ra (Native Development Kit). Az NDK részét képezi több különféle G++ és Clang cross-compiler. A következő paranccsal feltelepíthetünk egyet az általunk megadott könyvtárba.

```
~/android-sdk/ndk-bundle/build/tools/make-standalone-toolchain.sh \
    --arch=arm --install-dir=/home/user-name/armcc
```

A C++14 használatához esetleg másik fordítóra lehet szükségünk, így megadhatjuk a kívánt fordító típust [32]:

```
~/android-sdk/ndk-bundle/build/tools/make-standalone-toolchain.sh \
    --arch=arm --install-dir=/home/user-name/arm-gcc-4.9 \
    --toolchain=arm-linux-androideabi-4.9
```

Fordítás Ezután például a következőképp fordíthatunk vele:

```
~/armcc/bin/arm-linux-androideabi-g++ \
    -std=c++11 -pthread -fPIE -pie -O2 main.cpp -o arm-progi
```

- Az -std=c++11 kapcsoló a megszokott módon c++11 módba állítja a fordítót.
- A -pthread kapcsolóra minden többszálú program esetén szükség van, ez a linker-nek jelzi, hogy építse be a pthread függvénykönyvtárat, a fordítónak pedig, hogy többszálú program számára megfelelő kódot szeretnénk generálni.
- Az -fPIE és -pie kapcsolókra is valószínűleg szükség lesz, ezek jelzik, hogy Position-Independent Executable formátumú binárist szeretnénk kapni. Sok Android verzió csak az ilyen binárisokat támogatja. Az ilyen kód pozíció független, tehát bármely memóriaterületre betöltve futtatható. Ezt főleg biztonsági okokból preferálják egyes rendszereken, ugyanis a binárist véletlenszerű helyre betöltve az esetleges támadók nem tudják, hogy melyik memória címre kerülnek az egyes kódrészek.
- Az -O2 vagy -O3 kapcsoló a fordítót állítja erősen optimalizáló módba. Ha performanciát szeretnénk mérni érdemes lehet bekapcsolni.

Telefon beállítása A telefonon való futtatáshoz be kell kapcsolnunk a fejlesztői menüt és az USB-debug módot a telefonon. Ezt a hivatkozott [33] weboldalon írják le részletesen.

Futtatás Ezután adb-push-sal feltölthetjük a telefon egy megfelelő könyvtárába (nem minden könyvtárból engedi futtatni a binárisokat) és adb-shell-el futtathatjuk.

```
~/android-sdk/platform-tools/adb push arm-progi /data/local/tmp
~/android-sdk/platform-tools/adb shell "/data/local/tmp/arm-progi"
```

Teljesítmény mérés A többszálú programok futásidejét az ütemezés esetlegessége miatt érdemes úgy mérni, hogy egy viszonylag rövid ideig futó programot írunk, amit (pl.: egy shell-script-tel) több százszor lefuttatunk.

Androidon általában viszonylag kevés parancs érhető el, ezért az ismétléshez ezt az egyszerű szkriptet alkalmazhatjuk. (A while parancs feltételénél a szóközök is fontosak.)

```
#!/bin/sh

N=$1
shift
i=0
while [ $i -lt $N ]
do
    $@
    i=$(($i + 1))
done
```

B.1. Kódrészlet. Ismétlés Android rendszeren (repeat.sh)

Összességében ez az a bash fájl, amit a Linuxos host számítógépen szoktam futtatni:

```
#!/bin/bash
~/armcc/bin/arm-linux-androideabi-g++ \
    -std=c++11 -pthread -fPIE -pie -O2 main.cpp -o arm-progi
~/android-sdk/platform-tools/adb push arm-progi /data/local/tmp
~/android-sdk/platform-tools/adb push repeat.sh /data/local/tmp
~/android-sdk/platform-tools/adb shell \
    "cd /data/local/tmp/; time sh ./repeat.sh 100 ./arm-progi"
```

B.2. Kódrészlet. Teljesítmény mérés Android rendszeren (android-test.sh)

A program outputját ugyanúgy látjuk a parancssorban, mintha a saját számítógépünkön futna.

Irodalomjegyzék

[1] 2014, *Working Draft, Standard for Programming Language C++*,
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf

[2] Scott Meyers, Andrei Alexandrescu
2004, *C++ and the Perils of Double-Checked Locking*
http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf

[3] *A volatile működése Microsoft fordító használatával*
https://msdn.microsoft.com/en-us/library/12a04hfd.aspx

[4] 2006, *Java 6 szabvány - memória modell*
http://docs.oracle.com/javase/specs/jls/se6/html/memory.html#17.4

[5] *C# 5.0 szabvány*
https://www.microsoft.com/en-us/download/details.aspx?id=7029

[6] Jeff Preshing
2013, *Double-Checked Locking is Fixed In C++11*
http://preshing.com/20130930/double-checked-locking-is-fixed-in-cpp11/

[7] Ulrich Drepper
2007, *What Every Programmer Should Know About Memory*
https://www.akkadia.org/drepper/cpumemory.pdf

[8] Francesco Zappa Nardelli
2015, *C Concurrency: Still Tricky*
http://llvm.org/devmtg/2015-04/slides/CConcurrency_EuroLLVM2015.pdf

[9] R. C. Lacher
2015, *Lecture notes - Singleton pattern*
http://www.cs.fsu.edu/~lacher/lectures/Output/loki6/script.html

[10] Hans-J. Boehm
2004, *Threads Cannot Be Implemented As a Library*
http://www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf

[11] Anthony Williams
2012, *C++ Concurrency in Action - Practical Multithreading*
https://www.manning.com/books/c-plus-plus-concurrency-in-action

[12] Herb Sutter
2012, *atomic<> weapons (C++ and Beyond 2012)*
https://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-1-of-2

[13] Joe Duffy
2008, *Concurrent Programming on Windows*
http://www.amazon.com/Concurrent-Programming-Windows-Joe-Duffy/dp/032143482X

[14] Dmitriy Vyukov
Relacy Race Detector
http://www.1024cores.net/home/relacy-race-detector

[15] Andrey Karpov
2009, *Interview with Dmitriy Vyukov - the author of Relacy Race Detector (RRD)*
http://www.viva64.com/en/a/0041/

[16] Anthony Williams
2008, *The Intel x86 Memory Ordering Guarantees and the C++ Memory Model*
https://www.justsoftwaresolutions.co.uk/threading/intel-memory-ordering-and-c++-memory-model.html

[17] Paul McKenney
2007, *What is RCU, Really? (3 részes cikksorozat a Linux Weekly News-ban)*
http://www.rdrop.com/~paulmck/RCU/whatisRCU.html

[18] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais and Jonathan Walpole
2011, *User-Level Implementations of Read-Copy Update*
http://www.efficios.com/pub/rcu/urcu-main.pdf
http://www.efficios.com/pub/rcu/urcu-supp.pdf

[19] Mathieu Desnoyers, Paul E. McKenney
LGPL-licensed Userspace RCU Library
http://liburcu.org/

[20] *Wikipédia - Hazard pointer*
https://en.wikipedia.org/wiki/Hazard_pointer

[21] Maged M. Michael
2004, *Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects*
https://www.research.ibm.com/people/m/michael/ieeetpds-2004.pdf/

[22] *Linux Cross Reference - Az `rcu_assign_pointer` név használata*
http://lxr.free-electrons.com/ident?i=rcu_assign_pointer

[23] *Wikipédia - Read-copy-update*
https://en.wikipedia.org/wiki/Read-copy-update

[24] *Cppreference.com: Atomic thread fence*
http://en.cppreference.com/w/cpp/atomic/atomic_thread_fence

[25] David Howells, Paul E. McKenney
Linux kernel memory barriers
https://www.kernel.org/doc/Documentation/memory-barriers.txt

[26] David S. Miller
Semantics and Behavior of Atomic and Bitmask Operations
https://www.kernel.org/doc/Documentation/atomic_ops.txt

[27] Jonathan Corbet
2014, *C11 atomic variables and the kernel*
https://lwn.net/Articles/586838/

[28] Joseph Tassarotti, Derek Dreyer, Viktor Vafeiadis
2015, *Verifying Read-Copy-Update in a Logic for Weak Memory*
http://plv.mpi-sws.org/gps/rcu/full-paper.pdf

[29] Joseph Tassarotti, Derek Dreyer, Viktor Vafeiadis
2011, *Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming*
https://www.usenix.org/legacy/event/atc11/tech/final_files/Triplett.pdf

[30] Jonathan Corbet
2014, *Relativistic hash tables, part 1: Algorithms*
https://lwn.net/Articles/612021/

[31] *Cross compiling [C++ for Android]*
http://janos.io/articles/cross-compile.html

[32] *Android NDK Documentation / Standalone Toolchain*
http://developer.android.com/ndk/guides/standalone_toolchain.html

[33] *How to Enable USB Debugging Mode [and Developer Options Menu] on Android*
https://www.kingoapp.com/root-tutorials/how-to-enable-usb-debugging-mode-on-android.htm

[34] *A dolgozathoz tartozó programok git tárolója*
https://gitlab.com/tomi92/lockfree

Kódrészletek jegyzéke

Táblázatok jegyzéke

NYILATKOZAT

Név: Danyluk Tamás

ELTE Természettudományi Kar, szak: Alkalmazott Matematikus MSc

NEPTUN azonosító: PKUOBW

Szakdolgozat címe:
Lock-szegény adatszerkezetek tervezése, elemzése és implementálása a C++11 memóriamodelljében

A **szakdolgozat** szerzőjeként fegyelmi felelősségem tudatában kijelentem, hogy a dolgozatom önálló munkám eredménye, saját szellemi termékem, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel.

Budapest, 2016. Május 20.

_____

a hallgató aláírása