Többszálú programozás

Albert István

ialbert@aut.bme.hu

Q.B. 221, 1662

Automatizálási és
Alkalmazott
Informatikai Tanszék

Tartalom

Explicit szálkezelés

- Explicit szál indítás: Thread osztály

```
static void Threads()
{
    bool stop = false;
    var t1 = new Thread(() =>
    {
        while(!stop) Console.Write('-');
    });
    var t2 = new Thread(() =>
    {
        while (!stop) Console.Write('_');
    });
    t1.Start();
    t2.Start();
    Thread.Sleep(3000);
    stop = true;
    t1.Join();
    t2.Join();
}
```

```
-Done.
```

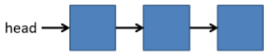
Szinkronizáció

- Párhuzamos futás esetén a közösen használt erőforrások védelme
- Sok tényező: tipikusan az átbocsátóképesség maximalizálása
- **Kernel szintű** megoldások szükségessége: a szál indítását/megállítását csak a kernel tudja hatékonyan megoldani
- **LockFree** algoritmusok: kerneltől függetlenek, de tipikusan több CPU-t használnak

A kölcsönös kizárás kritikája

- „Lassú”, mert kernelt hív
- Akkor is van overheadje, ha nincs blokkolás
- A késleltetési idő nagyon szórhat
- Nehezebb programozni
- Rosszul skálázódik sok szálra

Lock-free stack (1)



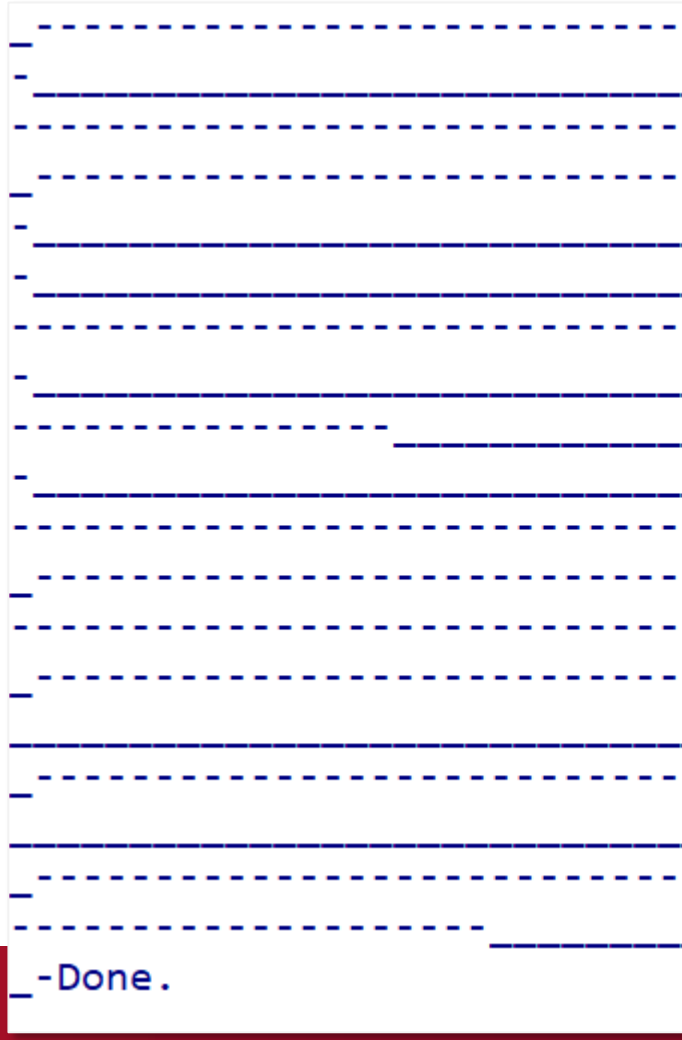
Parallel LINQ

- LINQ: Language Integrated Query
  - Standard Query Operators for IEnumerable
  - Where, Select, OrderBy, Join, Take és hasonló műveletek a gyűjtemények felett
  - Memóriában, objektumokon (NEM adatbázison)
- Parallel LINQ
  - A fenti műveletek végrehajtása több szálon
  - A bejövő adat felbontása több részre, hogy több szál párhuzamosan tudja feldolgozni
  - Nem mindig gyorsabb !

Explicit szálkezelés

- Explicit szál indítás: Thread osztály

```
static void Threads()
{
    bool stop = false;
    var t1 = new Thread(() =>
    {
        while(!stop) Console.Write('-');
    });
    var t2 = new Thread(() =>
    {
        while (!stop) Console.Write('_');
    });
    t1.Start();
    t2.Start();
    Thread.Sleep(3000);
    stop = true;
    t1.Join();
    t2.Join();
}
```



Thread osztály

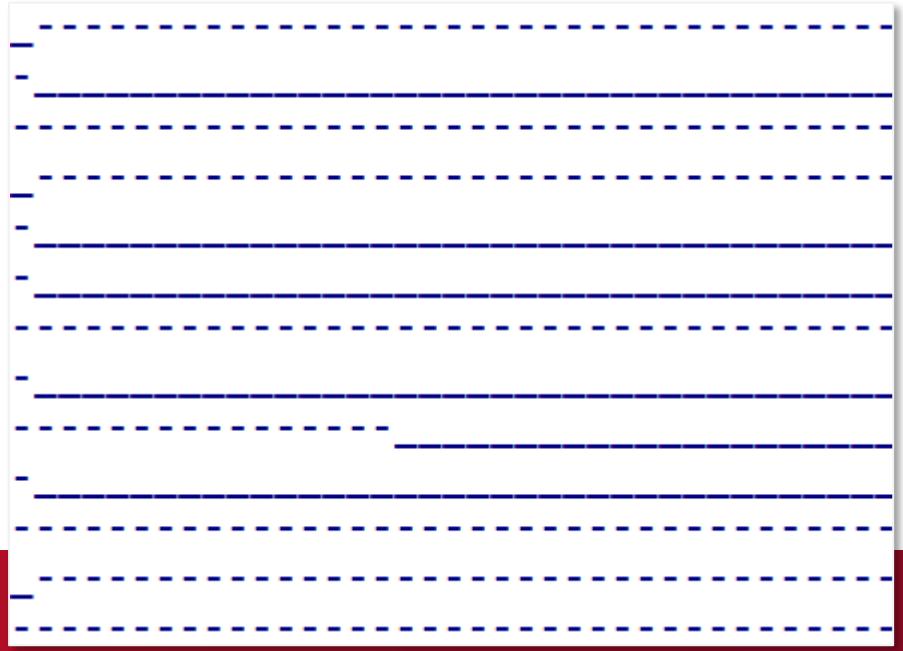
- Név, azonosító
- Hátter szál vagy sem
- Állapot: ThreadState, IsAlive
- Indítás: Start
- NINCS megállítás!

- Aktuális szál várakoztatása: Sleep

TreadPool

- Nem explicit hozzuk létre és szüntetjük meg a szálakat
  - > A szál létrehozása költséges - pool
  - > Túl sok szál csökkenti a teljesítményt
- Feladatot adunk a threadpoolnak

```
static void ThreadpoolSample()
{
    bool stop = false;
    ThreadPool.QueueUserWorkItem( d=>
    {
        while (!stop) Console.Write('-');
    });
    ThreadPool.QueueUserWorkItem( d=>
    {
        while (!stop) Console.Write('_');
    });
    Thread.Sleep(3000);
    stop = true;
}
```



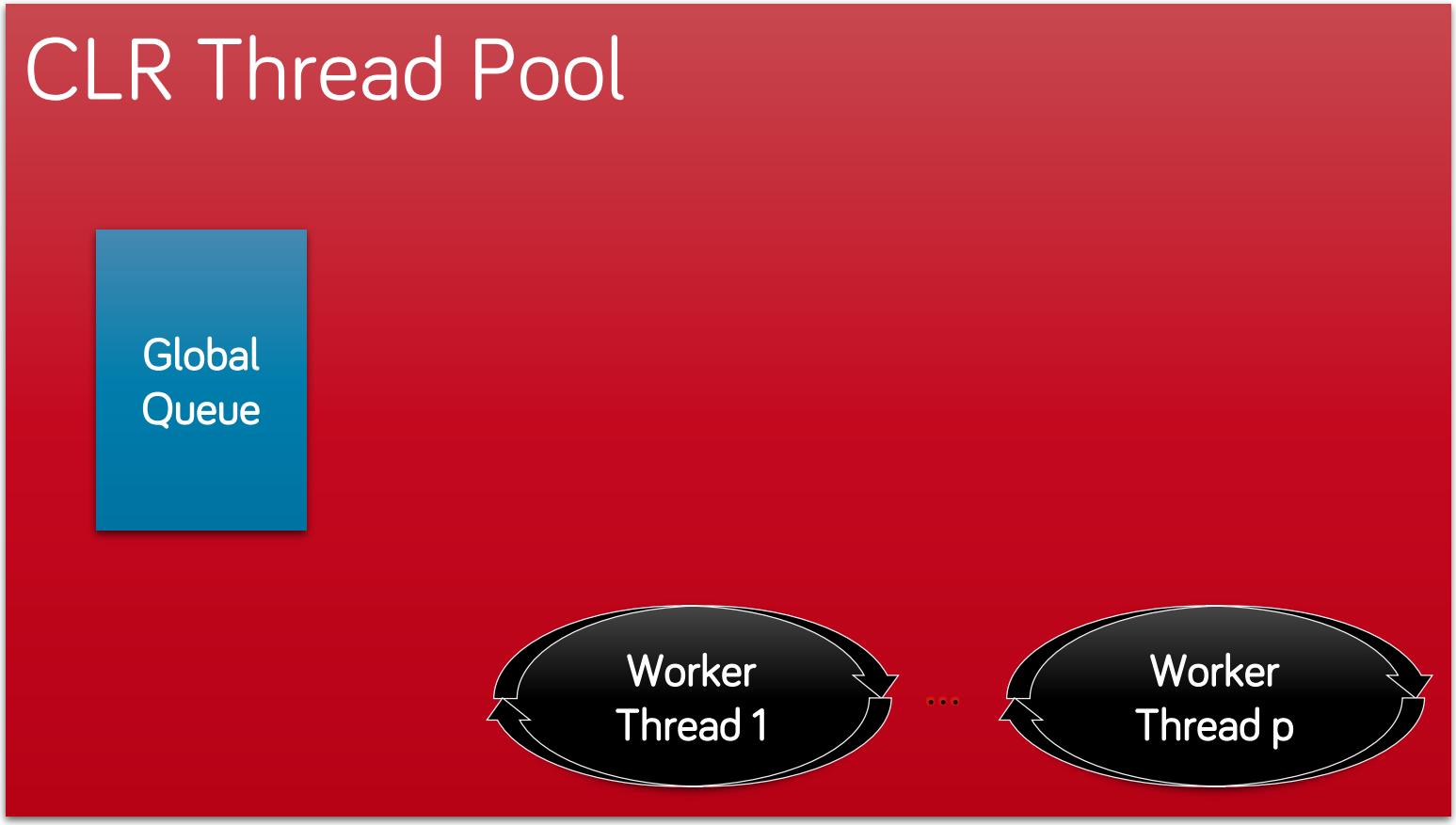
Thread pool jellemzők

- A TPL bevezetésével a feladatok több kisebb darabra bomlanak
  - Kis overhead kell!
  - Zármentes (lock-free) lista használata
  - GC barát adatstruktúra
  - .NET Core 2.1-ben is rengeteget optimalizáltak rajta
- A TPL alapértelmezett ütemezője a ThreadPool

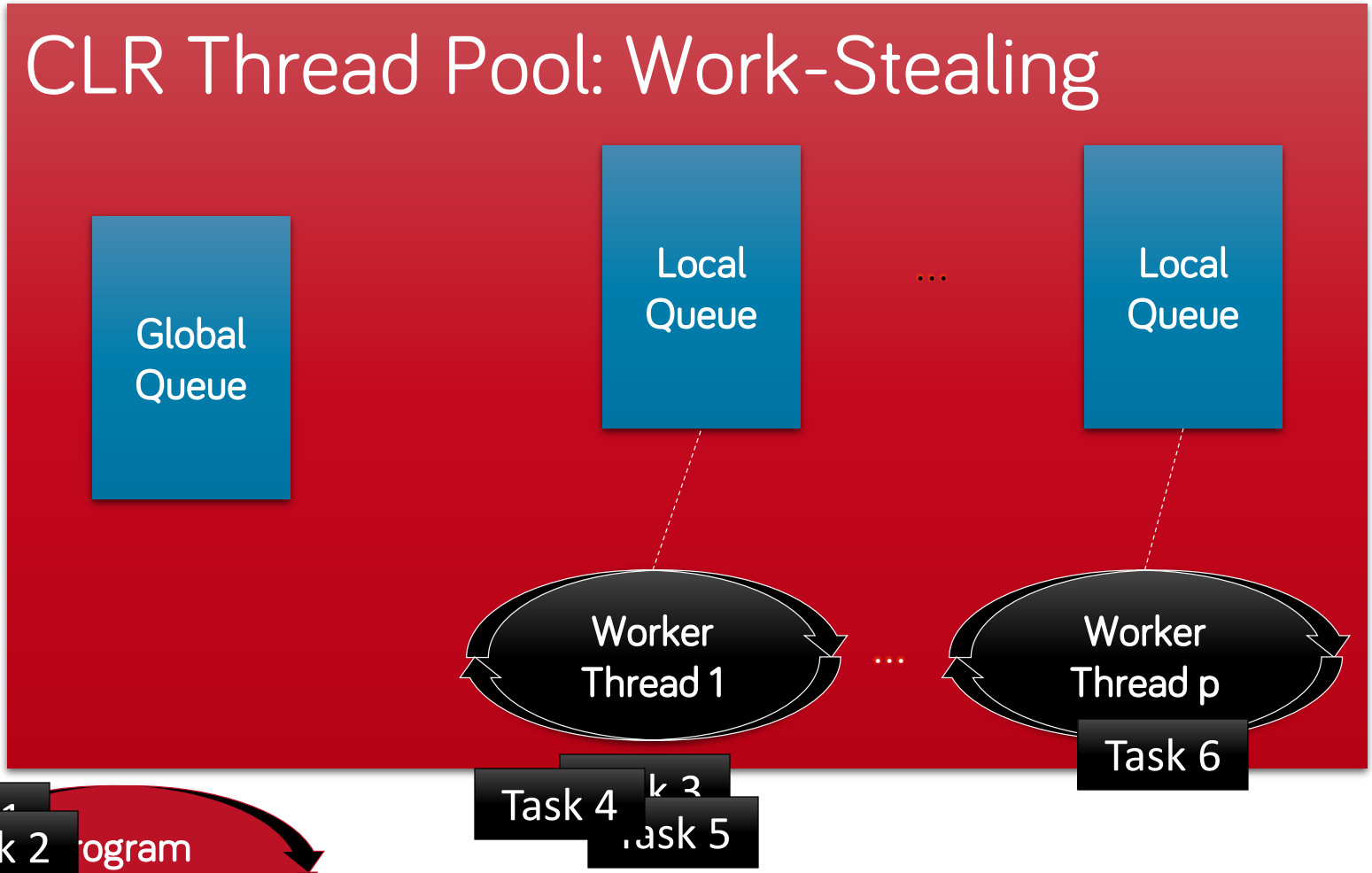
Work stealing queue

- Egyetlen globális és szálhoz tartozó lokális feladatsorok
- Thread pool szálból érkező feladatok a szál lokális sorába kerülnek (LIFO - cachebarát)
- Más szálból érkező feladatok a globális sorba kerülnek (FIFO – top level Taskok)
- A szálak a következők alapján dolgoznak
  1. Saját lokális sor
  2. Globális sor
  3. Másik szál lokális sora
- Task inlining: a várakozott task helyben fut szinkron módon
- Long-Running hint: külön szálat kap

User Mode Scheduler



User Mode Scheduler For Tasks



Szinkronizáció

- Párhuzamos futás esetén a közösen használt erőforrások védelme
- Sok tényező: tipikusan az átbocsátóképesség maximalizálása
- **Kernel szintű** megoldások szükségessége: a szál indítását/megállítását csak a kernel tudja hatékonyan megoldani
- **LockFree** algoritmusok: kerneltől függetlenek, de tipikusan több CPU-t használnak

Zárolás (locking)

- Erőforrás védelme: egyetlen szál férhet hozzá
- Monitor osztály, statikus metódusok (nem WaitHandle)
- Lock megszerzése: Enter, TryEnter, Exit
  - Thread affinitive (szálhoz kötött): Enter után Exitet kell hívni
- Várakozás/ébresztés: Wait, Pulse, PulseAll
  1. A szál megszerzi a zárat (Enter)
  2. Wait-tel átadhatja másnak, várakozik, hogy
  3. A másik visszaadja (Pulse vagy PulseAll + Exit)
- Ezzel implementálhatók magasabb szintű viselkedések
- lock kulcsszó
- Ne használjunk Stringet vagy Type-ot!
  - Hozz létre egy saját object-et!
- Ne az alacsony szinűt adattárat védjük!

Példa – manuális Enter / Exit

```
// Lock the queue and add an element.
public void Enqueue(T qValue)
{
    // Request the lock, and block until it is
       obtained.
    Monitor.Enter(m_inputQueue);
    try {
        // When the lock is obtained,add an element.
        m_inputQueue.Enqueue(qValue);
    } finally {
        // Ensure that the lock is released.
        Monitor.Exit(m_inputQueue);
} }
```

Példa – lock kulcsszó

```
// Lock the queue and add an element.
public void Enqueue(T qValue)
{
    // Request the lock, and block until it is
       obtained.
    lock(m_inputQueue) {
        // When the lock is obtained,add an element.
        m_inputQueue.Enqueue(qValue);
} }
```

Példa – AutoResetEvent

```
0 references
class AutoResetEvent
{
    readonly object key = new object();

    0 references | 0 exceptions
    void Sample()
    {

        bool block = true;

        // thread A
        lock (key)
        {
            while (block)
                Monitor.Wait(key);
            block = true;
        }

        // thread B
        lock (key)
        {
            block = false;
            Monitor.Pulse(key);
        }

    }
}
```

Mutex : WH

- Egy szál megszerezheti a Mutexet
  - Lehet lokális
  - Globális (nevesített): más folyamat is hivatkozhatja
    - Egyetlen példányban futó alkalmazások technikája
  - Másik szál is elengedheti
- WaitHandle-ből származik
  - Lehet rá várni, WaitAny-vel vagy WaitAll-lal
  - SignalAndWait: elengedi az egyiket és vár a másikra atomi műveletként (pl két AutoResetEvent)
  - Van handle-je

Semaphore : WH

- Több szál is megszerezheti az erőforrást
  - Beállítható, hogy legfeljebb hány darab
- WaitHandle-ös
  - WaitOne, …
- Release
- Lokális / globális (nevesíthető)

AutoResetEvent : WH

- Esemény, amire lehet várni, resetelni és jelezni
- Amint egy várakozó szál megkapja, automatikusan reseteli az eseményt
- EventWaitHandle-ös
  - Set, Reset metódusok
  - WaitHandle-ből származik
- Lokális / globális (nevesíthető)

ManualResetEvent : WH

- Esemény, amire lehet várni, resetelni és jelezni
- Több várakozó szál megkaphatja, amíg manuálisan nincs resetelve, addig szignálva marad
- WaitHandle-ös
- Lokális / globális (nevesíthető)

A kölcsönös kizárás kritikája

- „Lassú”, mert kernelt hív
- Akkor is van overheadje, ha nincs blokkolás
- A késleltetési idő nagyon szórhat
- Nehezebb programozni
- Rosszul skálázódik sok szálra

Pehelysúlyú szinkronizáció alapja

- Atomi utasítások
- Olvasás + írás egyetlen lépésben!
- Hardver által támogatva
- Csak architektúra szó méretű adatra: 32/64 bit
- Nem blokkol!
- Minden nem kernel alapú megoldás (zár vagy zármentes) erre épül

Interlocked

- Int32 vagy Int64 összeadás és lekérdezés atomi műveletként
- Increment, Decrement, Add, műveletek
- Exchange: két érték felcserélése
- CompareExchange: két érték felcserélése ha az egyik valamilyen állapotban van
  - Tipikus művelet a zármentes algoritmusoknál

Interlocked . CompareExchange

- Memória címen levő érték felülírása, ha az aktuális értéke megegyezik a megadottal
  - A módosítás nem történik meg, ha közben más az adatot felülírta
- „Optimista konkurencia kezelés.”
- Egyetlen atomi lépés

```
static Int32 CompareExchange(
    ref Int32 location,
    Int32 value,
    Int32 comparand )
{
    var currentValue = location;
    if (currentValue == comparand)
        location = value;
    return currentValue;
}
```

Interlocked – összegzés 0

```
int[] vector = Enumerable.Range(0, 1000000).ToArray();



int sum = 0;
Parallel.For(0, vector.Length,
                i => i + 1);
```

Interlocked – összegzés 1

```
int[] vector = Enumerable.Range(0, 1000000).ToArray();



int sum = 0;
Parallel.For(0, vector.Length,
                i => Interlocked.Add(ref sum, vector[i]));
```

Interlocked – összegzés 2

```
int[] vector = Enumerable.Range(0, 1000000).ToArray();
```

```
int sum = 0;
Parallel.For(0, vector.Length,
                i => Interlocked.Add(ref sum, vector[i]));
```

```
var totalSum = 0;

Parallel.For(0, vector.Length,

                () => 0,

                (i, loop, localSum) => localSum += vector[i],

                localSum => Interlocked.Add(ref totalSum, localSum));
```

Interlocked – maximum keresés 1

```
int[] vector = Enumerable.Range(0, 1000000).ToArray();

int max = vector[0];
Parallel.For(0, vector.Length, i => {
    if (vector[i] > max)
    {
        do
        {
            int myMax = max;
            if (vector[i] > myMax)
            {
                if (Interlocked.CompareExchange(
                            ref max, vector[i], myMax) == myMax)
                    break;
            }
           else
               break;
        } while (true);
} } );
```

Interlocked- maximum keresés 2

```
int[] vector = Enumerable.Range(0, 1000000).ToArray();


int max = vector[0];
Parallel.For(0, vector.Length,
   () => vector[0],
   (i, loop, localMax) => vector[i] > localMax ? vector[i] : localMax,
   localMax => {
        if (localMax > max)
        {
           do
           {
               int myMax = max;
               if (vector[i] > localMax)
               {
                   if (Interlocked.CompareExchange(
                              ref max, vector[i], localMax) ==
localMax)
                        break;
               }
           } while (true);
       } }
);
```

SpinWait

- „Vár egy kicsit”
  - SpinOnce, SpinUntil (feltétel, timeout)
- Single CPU esetén rögtön átadja a végrehajtást
  - HyperThreadingnél Pause hívás
- Tipikusan erőforrásra várunk anélkül, hogy átadnánk másik szálnak a végrehajtást
  - SpinOnce metódus: vár egy kicsit
  - NextSpinWillYield: a következő várás átadja-e a végrehajtást egy másik szálnak?
- Elsőre nem okoz szál váltást
- Nem jó CompareExchange-et használni egy szoros ciklusban – túl sok lesz a CPU cache ürítés

System.Threading.SpinWait

```
bool resultReady = false;

var t1 = Task.Factory.StartNew(() =>
{
        SpinWait sw = new SpinWait();
        while (!resultReady)
                sw.SpinOnce();
});

var t2 = Task.Factory.StartNew(() =>
{
        Thread.Sleep(100);
        resultReady = true;
});
```

Memória elérés optimalizáció

- A C# és JIT fordító (és a processzor ütemező) optimalizálhat(na)
- Ha a szálon nincs írás, akkor az olvasást előrébb hozza stb...

```
class Foo
{
   public int x;
   public int y;
}
```

Nem biztos,
hogy az 5-öt írja ki !

```
foo.y = 5;
foo.x = 1;
```

```
if (foo.x == 1) {
   Console.WriteLine(foo.y);
}
```

Optimalizáció kikapcsolása

- Interlocked . MemoryBarrier
  - Az olvasást nem lehet előrébb hozni, írást nem lehet késleltetni
- Volatile osztály
  - T Read(ref T location)
  - Write(ref T location, value)
- Írás, olvasás memória határral

SpinWait + Volatile példa

```
bool resultReady = false;

var t1 = Task.Factory.StartNew(() =>
{
    SpinWait sw = new SpinWait();
    while (!Volatile.Read(ref resultReady))
        sw.SpinOnce();
});

var t2 = Task.Factory.StartNew(() =>
{
    Thread.Sleep(100);
    Volatile.Write(ref resultReady, true);
});
```

SpinLock

- Enter, TryEnter, Exit metódusok
- Rövid várakozásnál jobb teljesítmény mint a Monitor
  - A CPU-t használja és nem megy le kernel módba
- Egy idő után átvált Monitorra
- Struct!
  - Nincs allokációs és GC overhead

SpinLock

```
private class LockedRegion : IDisposable
{
    private readonly DisposableSpinLock myLock;
    1 reference | 0 changes | 0 authors, 0 changes
    public LockedRegion(DisposableSpinLock myLock)
    {
        this.myLock = myLock;
    }
    0 references | 0 changes | 0 authors, 0 changes
    public void Dispose() { myLock.Exit(); }
}
```

```
class DisposableSpinLock
{
    int busy = 0;
    SpinWait spinWait = new SpinWait();

    0 references | 0 changes | 0 authors, 0 changes
    public IDisposable Enter()
    {
        while (Interlocked.CompareExchange(ref busy, 1, 0) != 0)
            spinWait.SpinOnce();
        return new LockedRegion(this);
    }

    1 reference | 0 changes | 0 authors, 0 changes
    public void Exit()
    {
        Interlocked.Exchange(ref busy, 0);
    }
```

ManualResetEventSlim

- Eleinte nem használ kernel objektumot
  - > De később létrehozhat
- Támogatja a CancellationToken-t
- Nincs AutoResetEventSlim

SemaphoreSlim

- Pehelysúlyú Semaphore
  - Eleinte nem használ kernel objektumokat
- Támogatja a CancellationToken-t

ReaderWriterLockSlim

- Több olvasó
- Egyetlen író
- Upgradeable
  - Egyetlen szál lehet ugradeable állapotban
- Rekurzív támogatás van, de nem javasolják
  - Overhead, bad practice

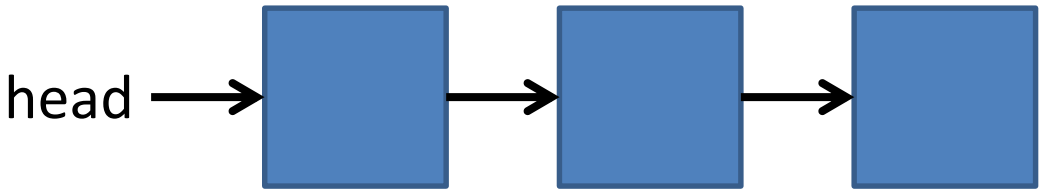
Barrier

- Több szál dolgozik egyetlen, fázisokra osztott feladaton
- A következő fázis mindig akkor kezdődhet, ha az összes résztvevő szál eljut az aktuális fázis végéig
  - Opcionálisan meg lehet hívni egy fázis utáni műveletet
- Létrehozásnál meg lehet adni a résztvevők számát
  - Dinamikusan állítható
- SignalAndWait metódus: kész az aktuális fázissal és vár a következő kezdetére
  - Timeout és CancellationToken támogatás

CountdownEvent : WH

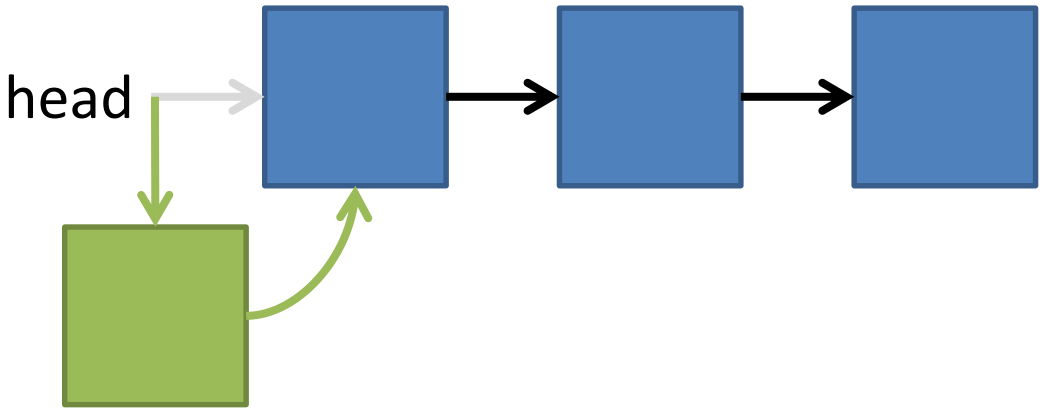
- Jelez, amikor eléri a nullát
- Reset, AddCount, Signal
- Wait, …
  - WaitHandle támogatás
- Támogatja a CancellationToken-t

Lock-free stack (1)



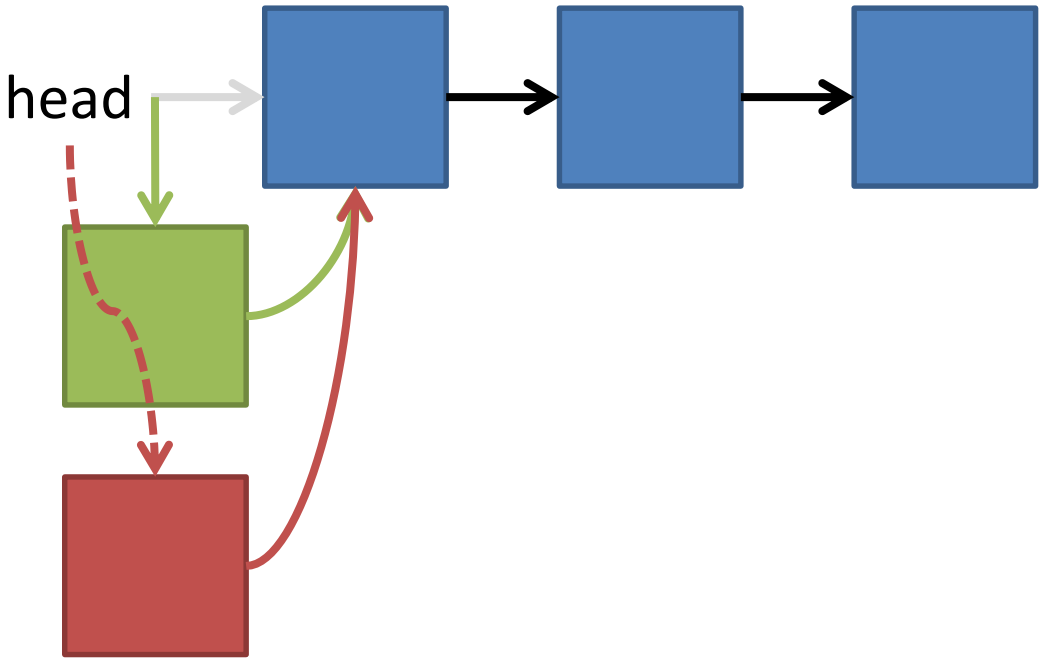
Lock-free stack (1)

- Push



Lock-free stack (1)

- Push

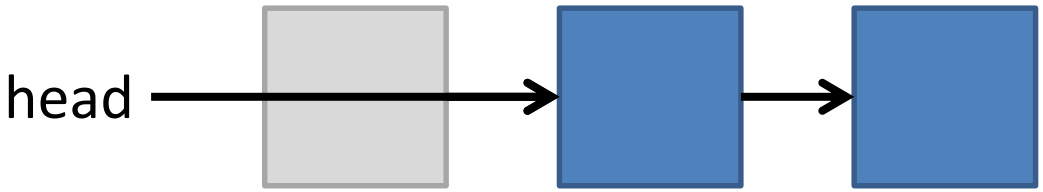


Lock-free stack (1)

```
public void Push(T value)
{
    var n = new Node<T>(value);
    while(true)
    {
        var myHead = head;
        n.Next = myHead;
        if (Interlocked.CompareExchange(
                                ref head, n, myHead) == myHead)
            break;
    }
}
```

Lock-free stack (1)

- Pop

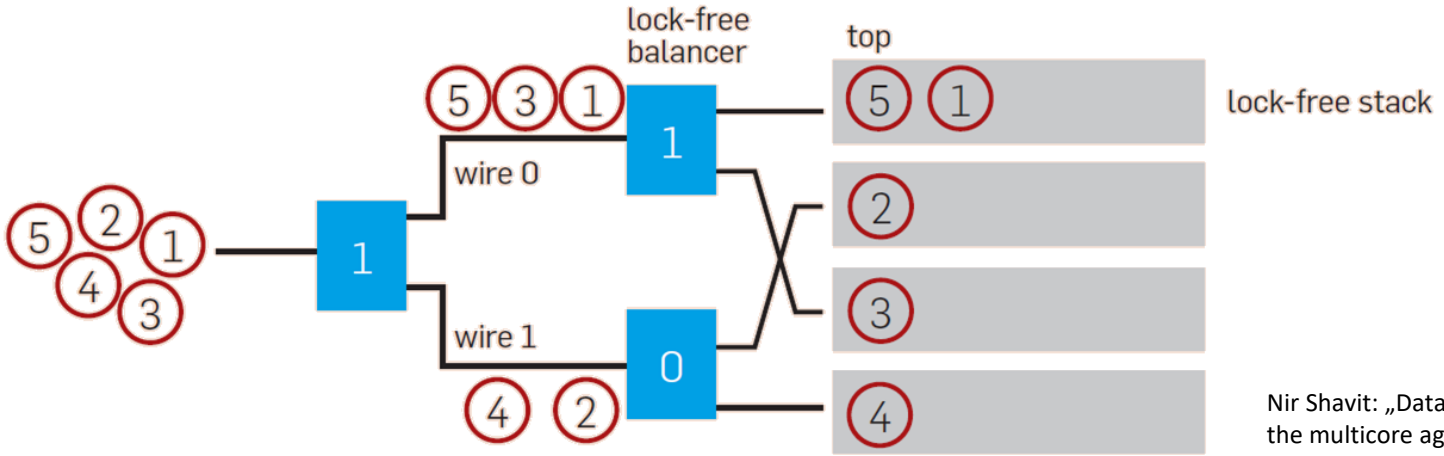


Lock-free stack (1)

```
public bool Pop(out T value)
{
    while (true)
    {
        if (head == null){ value = default(T); return false; }
        else
        {
            var myHead = head;
            value = myHead.Value;
            if (Interlocked.CompareExchange(
                        ref head, myHead.Next, myHead) == myHead)
                return true;
        }
    }
}
```

Lock-free stack (2)

- Skálázhatóság javítása érdekében több belső stack.
- Párhuzamos kérések más-más lock-free stack-hez mennek, valójában párhuzamosan futnak le.
- Stack-ek fa struktúrába rendezve, irányítók a csomópontokban, stack a levélben.
- Irányító: bool flag, balra vagy jobbra küldi a kérést.
- Push: irányító invertálása, *régi* irányba megy.
- Pop: irányító invertálása, *új* irányba megy.



Nir Shavit: „Data structures in the multicore age”

Lock-free stack (2)

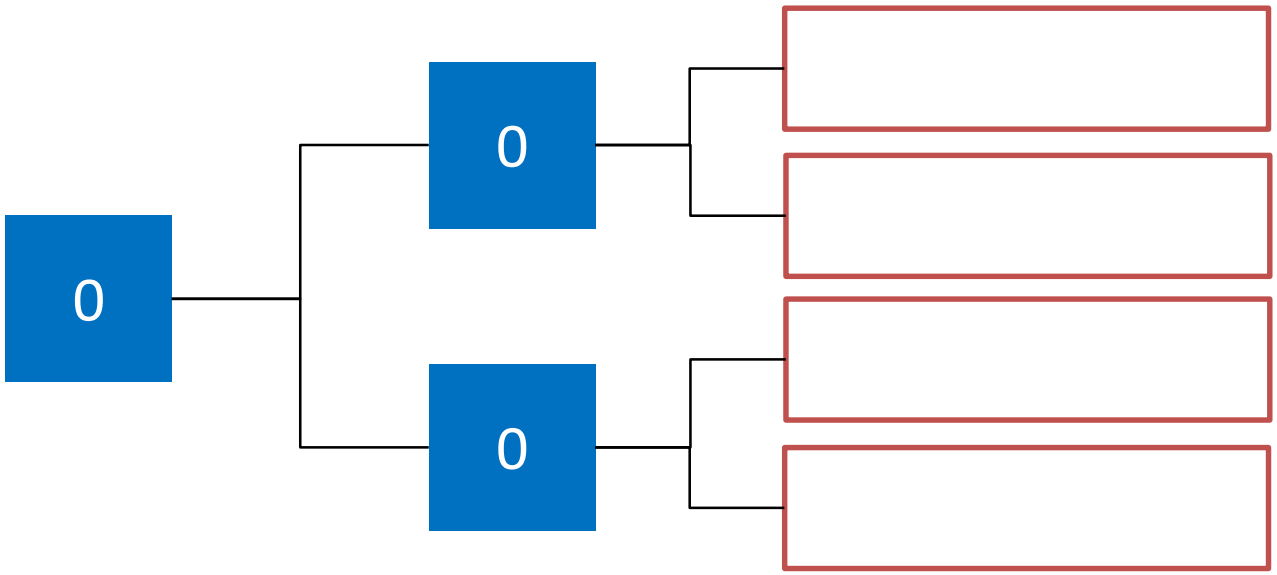
push 1

push 2

push 3

pop

pop



Lock-free stack (2)

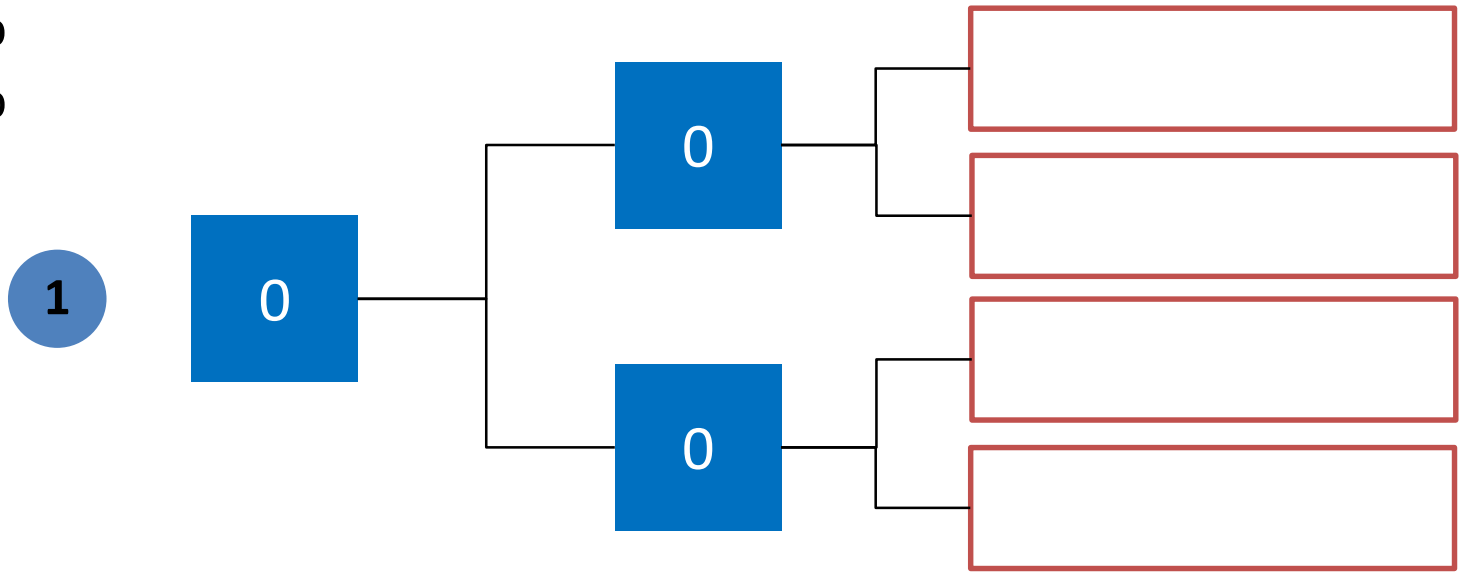
push 1

push 2

push 3

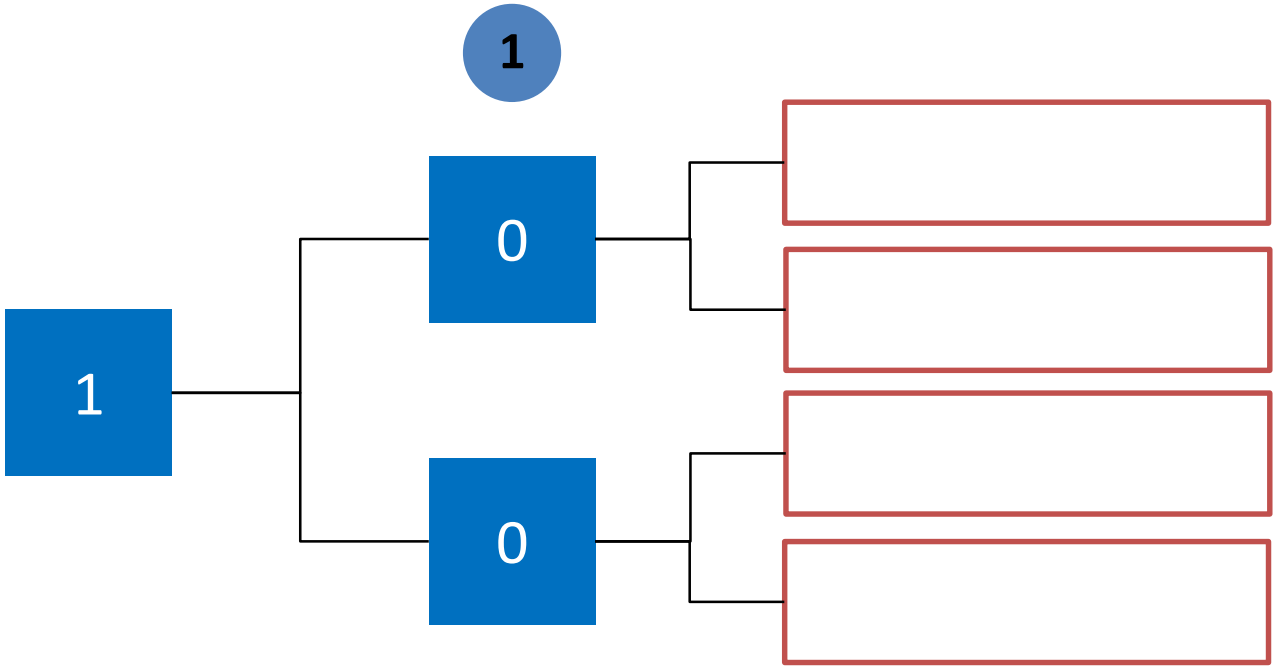
pop

pop



Lock-free stack (2)

push 1
push 2
push 3
pop
pop



Lock-free stack (2)

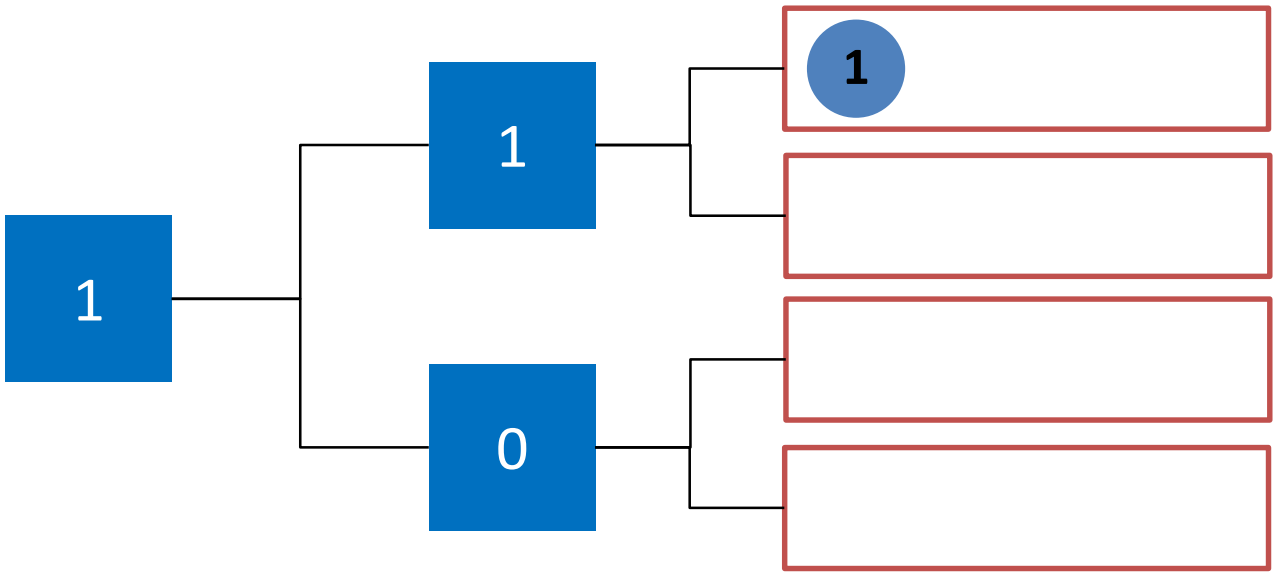
push 1

push 2

push 3

pop

pop



Lock-free stack (2)

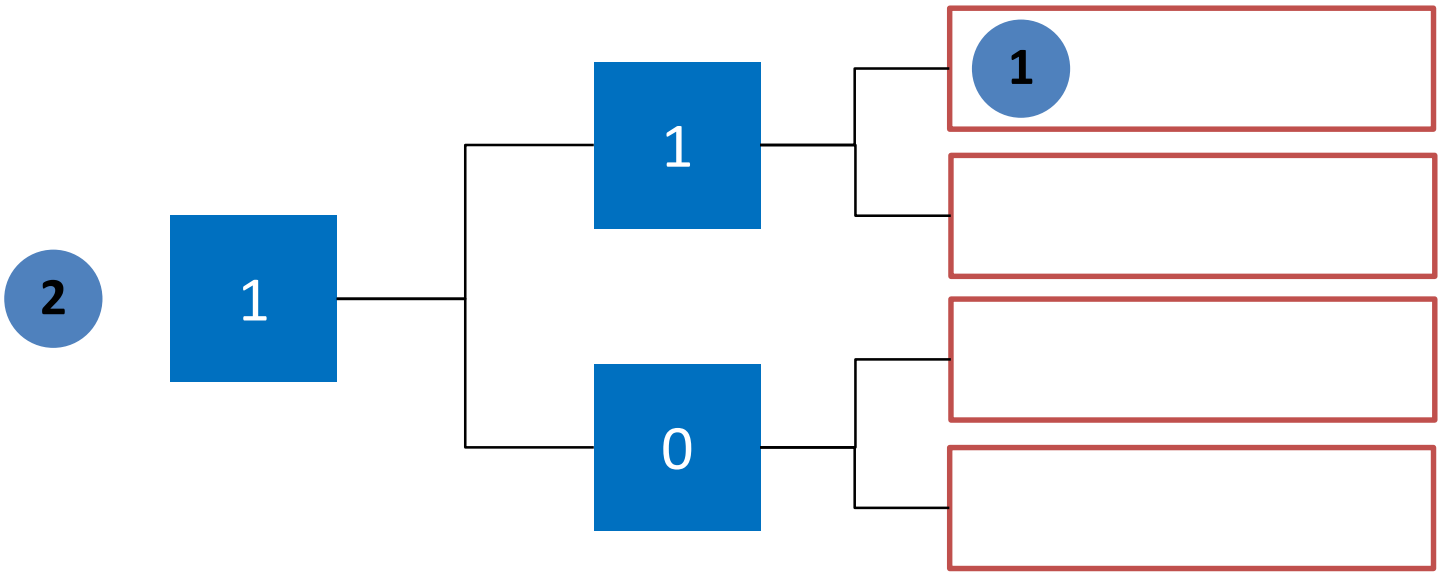
push 1

push 2

push 3

pop

pop



Lock-free stack (2)

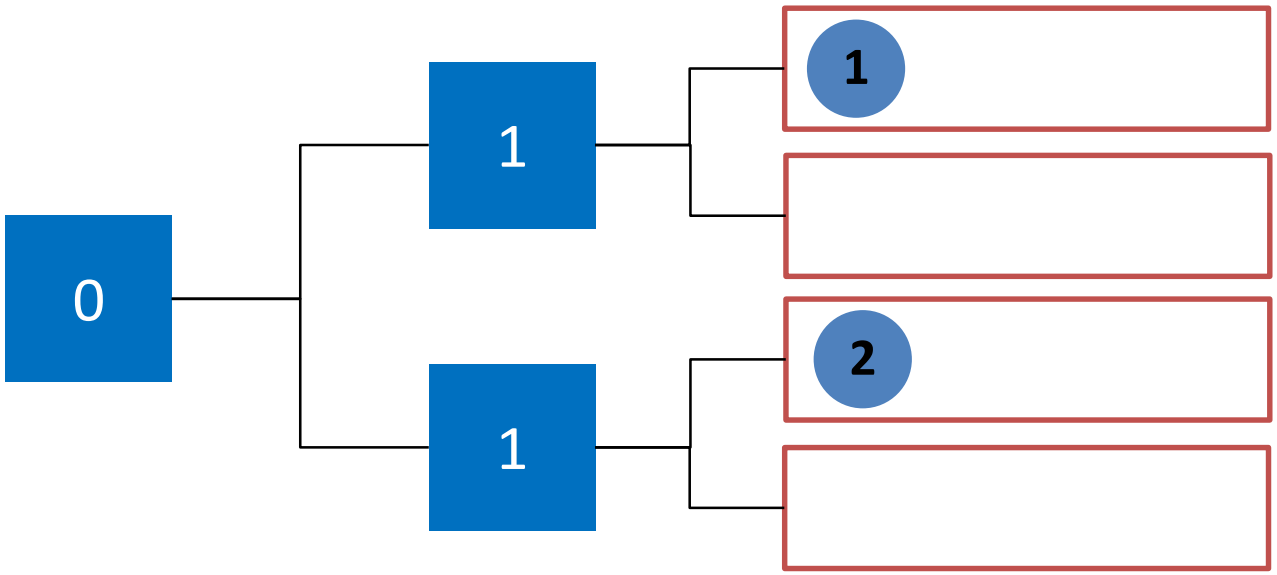
push 1

push 2

push 3

pop

pop



Lock-free stack (2)

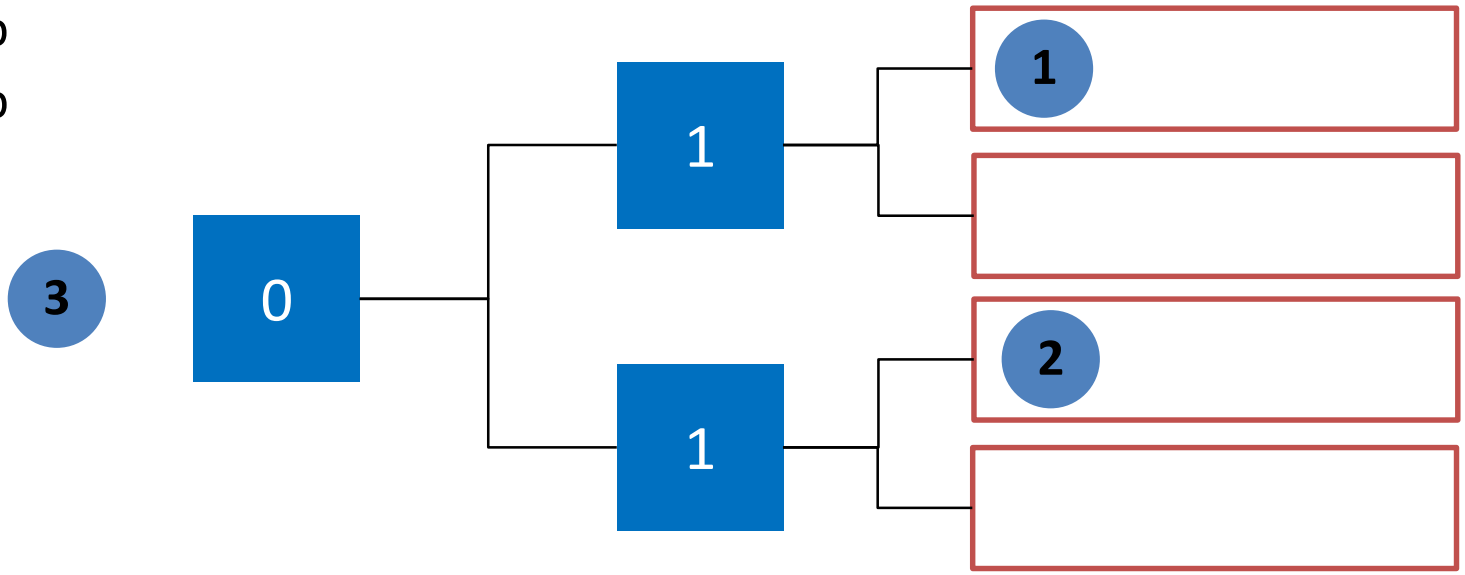
push 1

push 2

push 3

pop

pop



Lock-free stack (2)

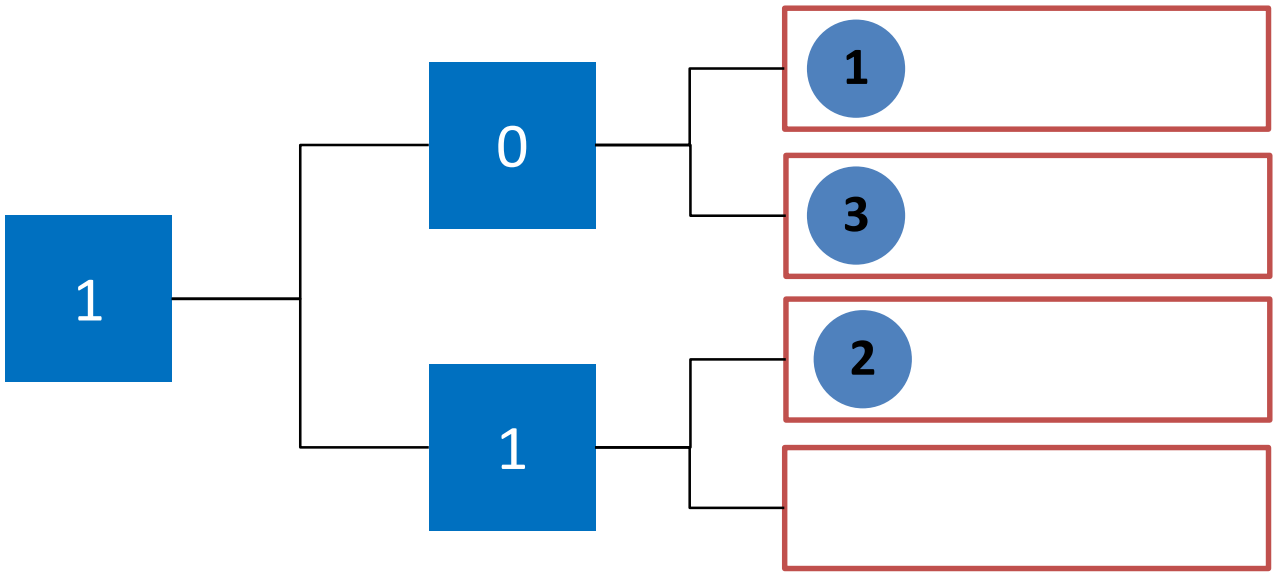
push 1

push 2

push 3

pop

pop



Lock-free stack (2)

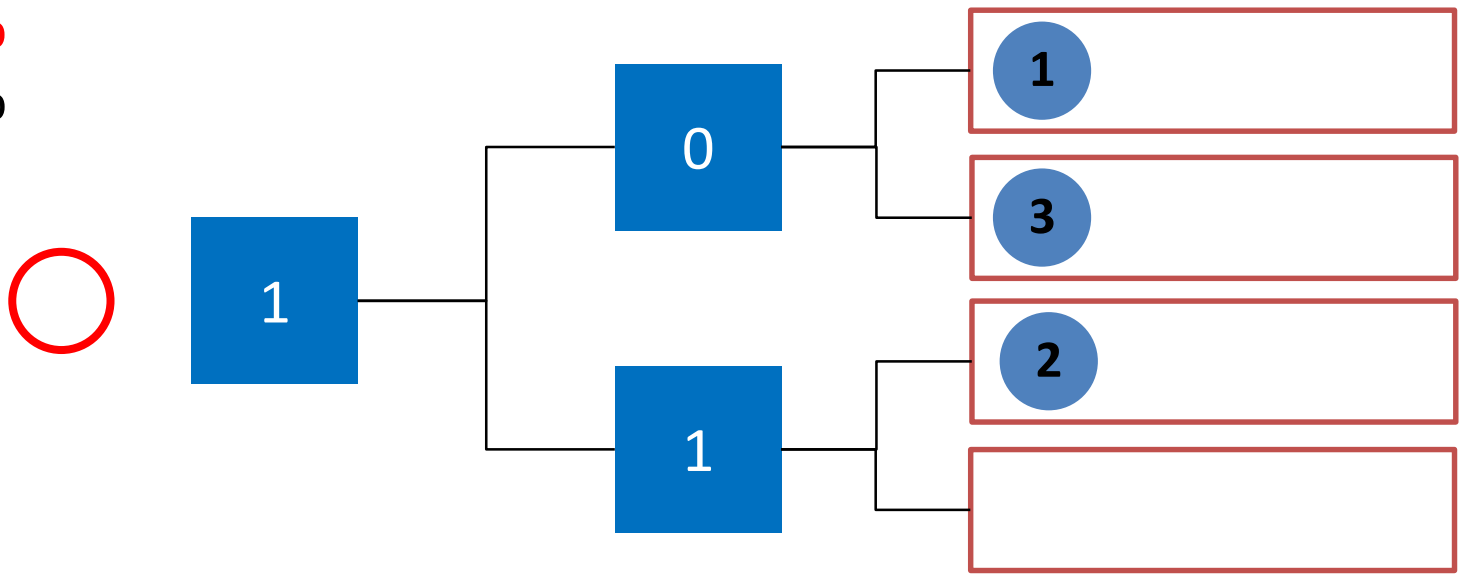
push 1

push 2

push 3

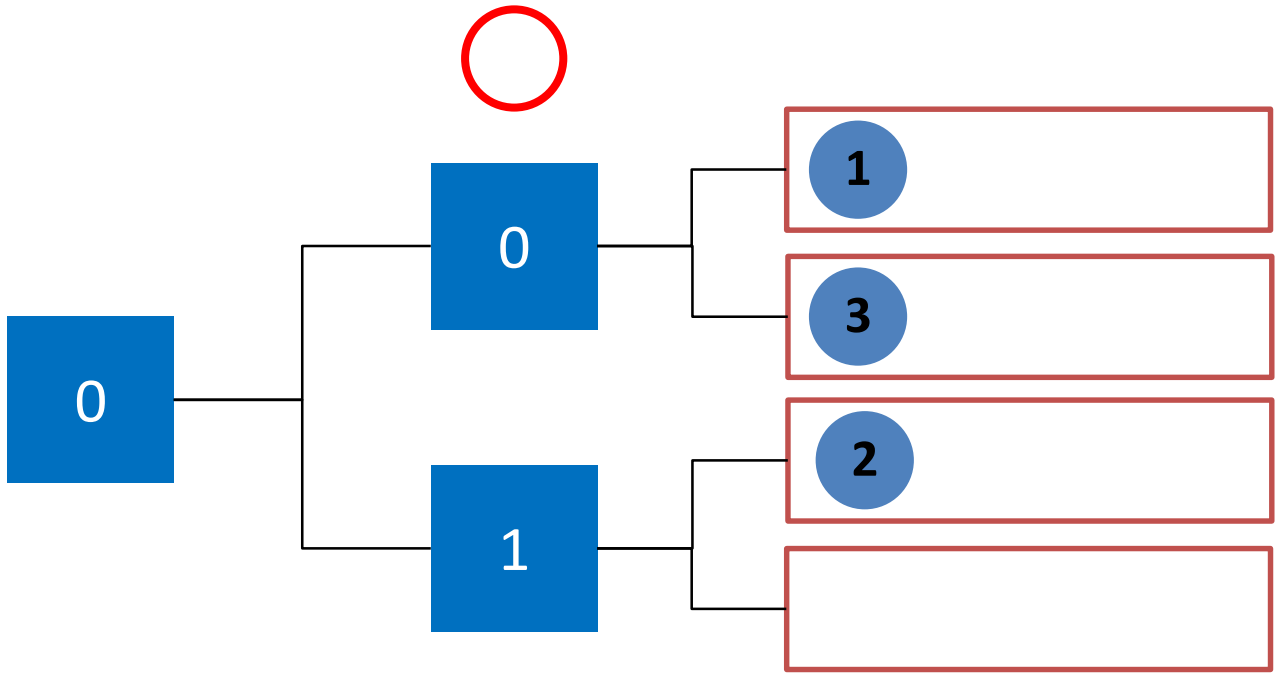
pop

pop



Lock-free stack (2)

push 1
push 2
push 3
pop
pop



Lock-free stack (2)

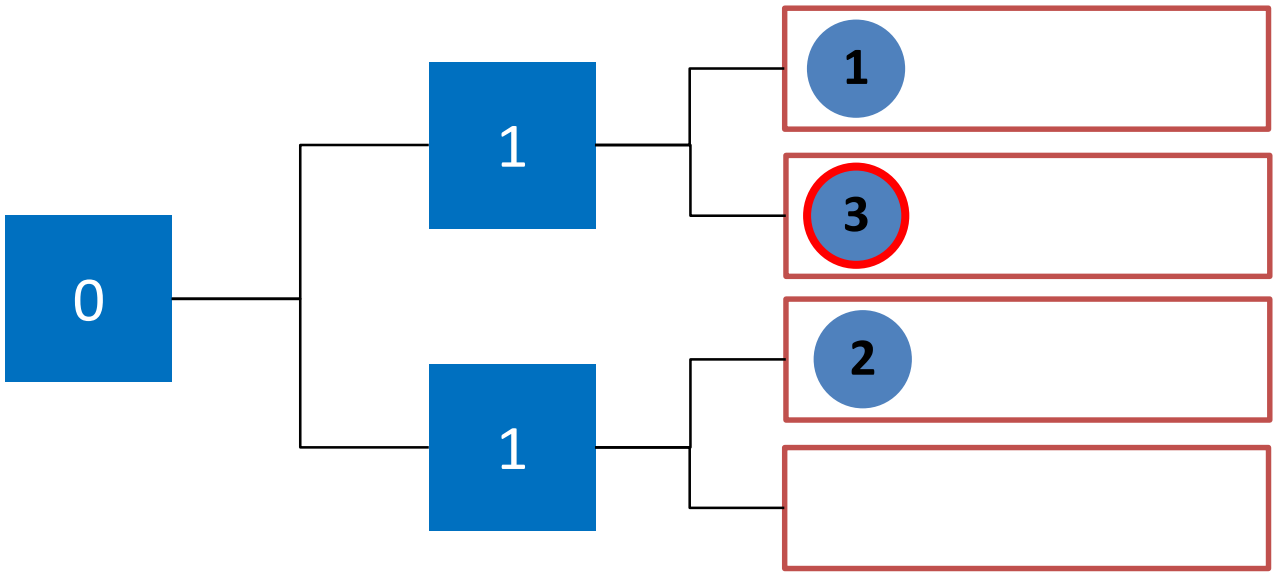
push 1

push 2

push 3

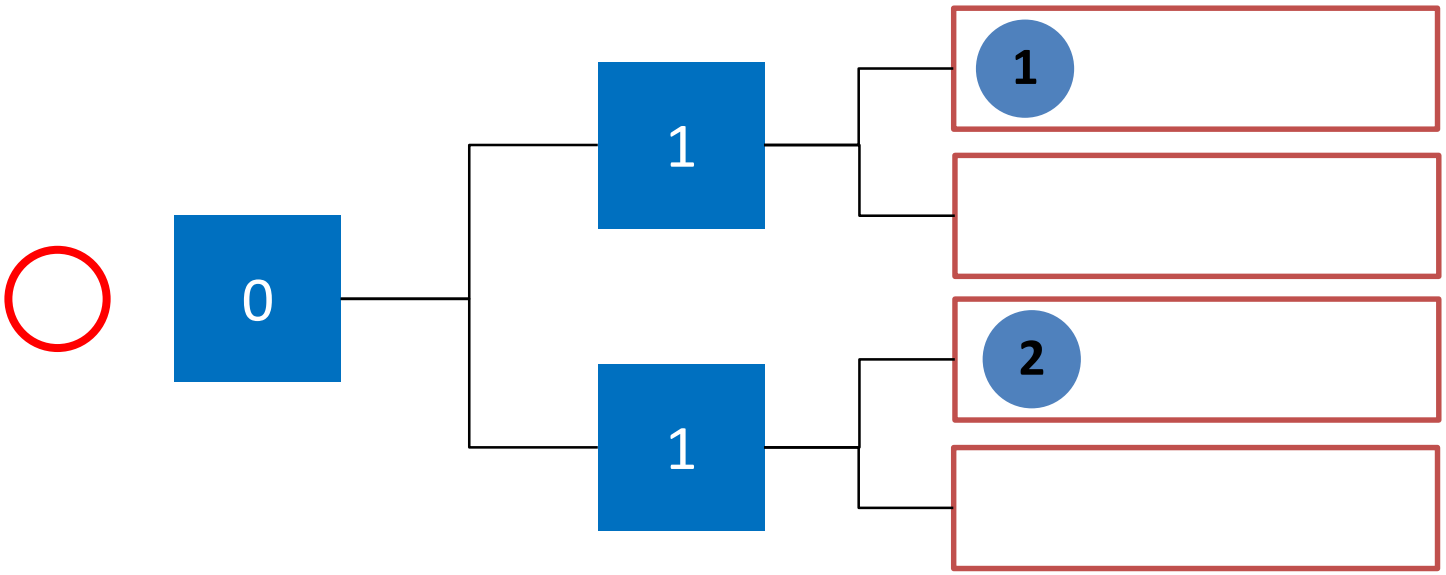
pop

pop



Lock-free stack (2)

push 1
push 2
push 3
pop
pop



Lock-free stack (2)

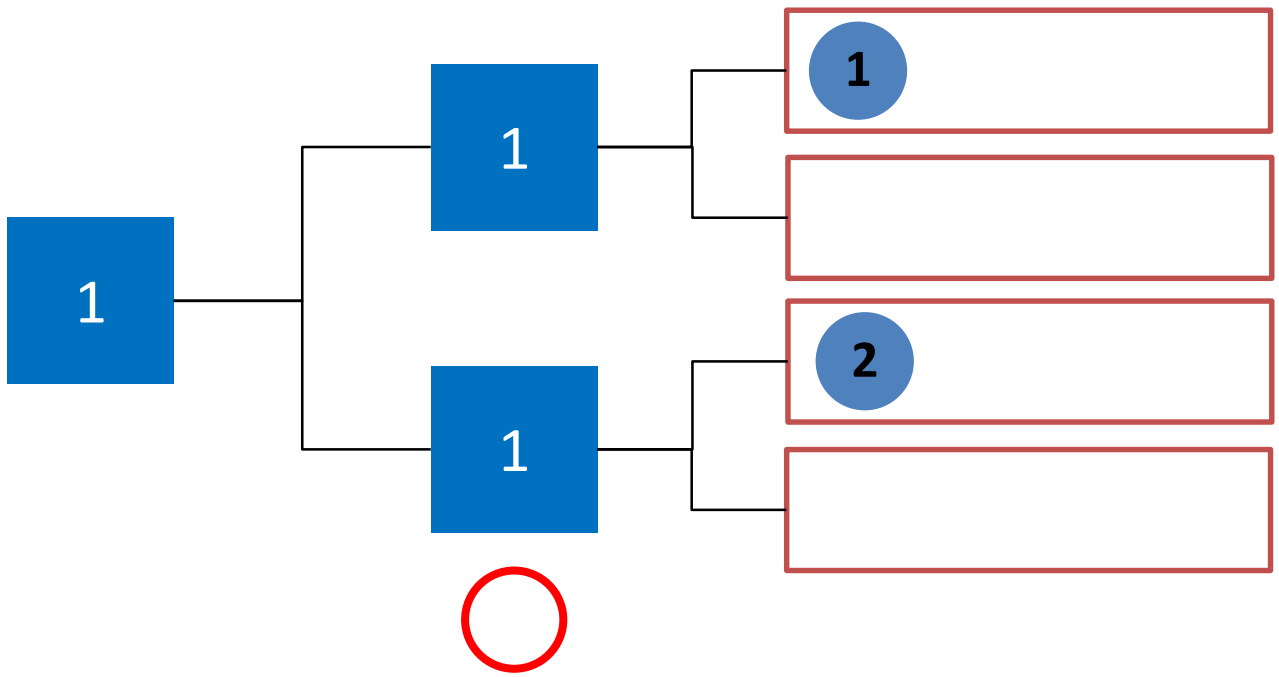
push 1

push 2

push 3

pop

pop



Lock-free stack (2)

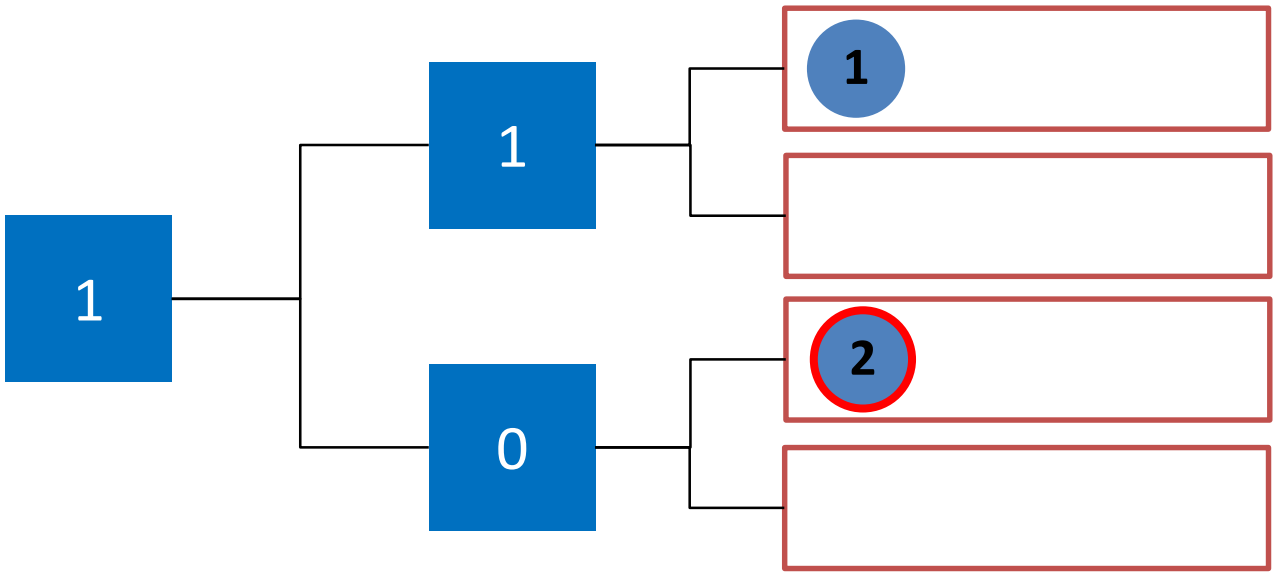
push 1

push 2

push 3

pop

pop



Lock-free stack (2)

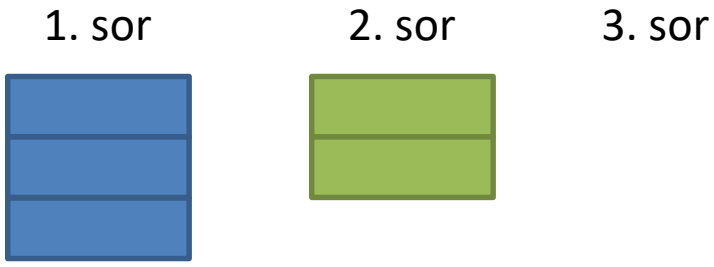
```
public void Push(T value)
{
   while (true)
   {
      var myDirection = direction;
      var newDirection = (myDirection + 1) % 2;
      if (Interlocked.CompareExchange(
               ref direction, newDirection, myDirection) == myDirection)
      {
          if (myDirection == 0)
              left.Push(value);
          else
              right.Push(value);
          return;
      }
}}
```

Lock-free bag

- Stack LIFO, queue FIFO
- Bag: nincs specifikált sorrend
- Minden szálnak saját terület, először csak ide írjon, olvasson. (Nincs konkurencia.)
- Ha nincs már olvasható elem, akkor forduljon másik szálhoz tartozó területhez.
- „Work stealing”

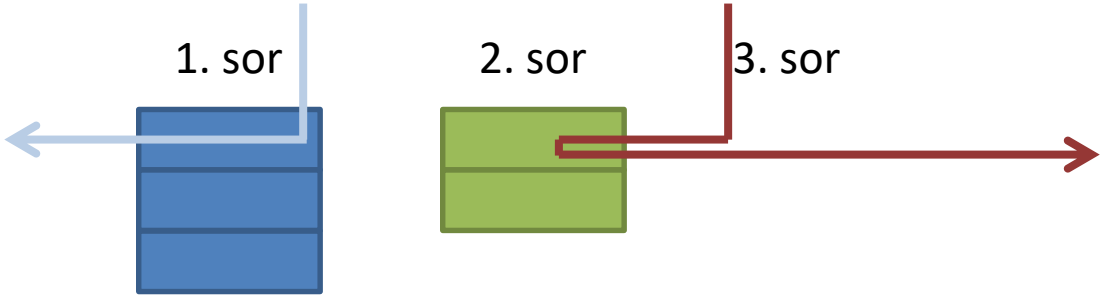
Lock-free bag

- Stack LIFO, queue FIFO
- Bag: nincs specifikált sorrend
- Minden szálnak saját terület, először csak ide írjon, olvasson. (Nincs konkurencia.)
- Ha nincs már olvasható elem, akkor forduljon másik szálhoz tartozó területhez.
- „Work stealing”



Lock-free bag

- Stack LIFO, queue FIFO
- Bag: nincs specifikált sorrend
- Minden szálnak saját terület, először csak ide írjon, olvasson. (Nincs konkurencia.)
- Ha nincs már olvasható elem, akkor forduljon másik szálhoz tartozó területhez.
- „Work stealing”



Lock-free bag

```
private SpinLock globalLock = new SpinLock();
private List<LockFreeStack<T>> allLists = new List<LockFreeStack<T>>();
private ThreadLocal<LockFreeStack<T>> myList=new ThreadLocal<LockFreeStack<T>>();

public void Add(T value)
{
    if (!myList.IsValueCreated) {
        myList.Value = new LockFreeStack<T>();
        bool lockTaken = false;
        try {
            globalLock.Enter(ref lockTaken);
            allLists.Add(myList.Value);
        }
        finally { if (lockTaken) globalLock.Exit(); }
    }
    myList.Value.Push(value);
}
```

Lock-free bag

```
public bool TryTake(out T value)
{
    if(myList.IsValueCreated && myList.Value.Pop(out value)) return true;

    LockFreeStack<T>[] allListsCache;
    bool lockTaken = false;
    try {
        globalLock.Enter(ref lockTaken);
        allListsCache = allLists.ToArray();
    }
    finally { if (lockTaken) globalLock.Exit(); }

    for (int i = 0; i < allListsCache.Length; ++i) {
        if (allListsCache[i].Pop(out value))
            return true;
    }
    return false;
}
```

System.Collections.Concurrent

- ConcurrentStack<T>
  - *Push, TryPop*
  - teljesen lock-free
- ConcurrentQueue<T>
  - *Enqueue, TryDequeue*
  - teljesen lock-free
- ConcurrentBag<T>
  - *Add, TryTake*
  - használ zárolást ha nem a saját szál területéhez fér hozzá
- ConcurrentDictionary<TKey,TValue>
  - *AddOrUpdate, GetOrAdd, TryAdd, TryRemove, TryUpdate*
  - nagyfelbontású zárolást használ írásra, olvasás lock-free
- BlockingCollection<T>
  - *Add, Take*
  - kölcsönös kizárást használ

BlockingCollection

- blokkoló Remove viselkedés (vár, amíg lesz adat)
- méret limit esetén Add szintén vár
- támogat CancellationToken-t
- belső tároló megválasztható (IProducerConsumerColection<T>)
  - alapértelmezett: ConcurrentQueue
- TakeFromAny, AddToAny

```
BlockingCollection<int> b = new BlockingCollection<int>();
var producer = Task.Factory.StartNew(() =>
{
    for (int i = 0; i < 100; i++)
        b.Add(i);
    b.CompleteAdding();
});
var consumer = Task.Factory.StartNew(() =>
{
    foreach (int data in b.GetConsumingEnumerable())
        process(data);
});
```

Parallel LINQ

- LINQ: Language Integrated Query
  - Standard Query Operators for IEnumerable
  - Where, Select, OrderBy, Join, Take és hasonló műveletek a gyűjtemények felett
  - Memóriában, objektumokon (NEM adatbázison)
- Parallel LINQ
  - A fenti műveletek végrehajtása több szálon
  - A bejövő adat felbontása több részre, hogy több szál párhuzamosan tudja feldolgozni
  - Nem mindig gyorsabb !

ParallelEnumerable osztály

- Hasonló az Enumerable osztályhoz
  - Bővítő SQO metódusok
  - ParallelQuery<TSource> felett
- AsParallel metódus (IEnumerable<T> felett)
  - A PLINQ belépési pontja
  - ParallelQuery<TSource>-t ad vissza
  - Utána minden művelet már ezt az osztályt használja
- További módosító metódusok
  - AsUnordered: nem kell megőrizni az eredeti sorrendet
  - WithCancellation: megszakítható
  - ForAll: az eredmény párhuzamos feldolgozása

Sorrend megőrzés

- A bejövő szekvenciát feldarabolja részekre
- Ezeket párhuzamosan dolgozza fel
- AsUnordered (default)
  - A végeredményt rögtön tovább adja a következő csomópontnak
- AsOrdered
  - A továbbiakban a sorrend megőrződik a csomópontok között
  - Az OrderBy, … után mindig megörződik a sorrend a legközelebbi AsUnordered hívásig

PLINQ példa 1

```
var source = Enumerable.Range(100, 20000);
// Result sequence might be out of order.
var parallelQuery = from num in source.AsParallel()
                    where num % 10 == 0
                    select num;
// Process result sequence in parallel
parallelQuery.ForAll((e) => Console.WriteLine(e));
// Or use foreach to merge results first.
foreach (var n in parallelQuery)
    Console.WriteLine(n);
```

PLINQ példa 2

```
// You can also use ToArray, ToList, etc
var parallelQuery2 = (from num in source.AsParallel()
                      where num % 10 == 0
                      select num).ToArray();

// Method syntax is also supported
var parallelQuery3 = source
    .AsParallel()
    .Where(n => n % 10 == 0)
    .Select(n => n);
```

Task Parallel Library (TPL)

- Új technológia a párhuzamos programozás segítésére
- Két összefüggő de mégis eltérő forgatókönyv:
  1. Munka párhuzamosítása (Task osztály, …)
  2. Adat feldolgozás párhuzamosítása (ParallelFor, …)
     - Támaszkodik a munka párhuzamosításra
- Alapértelmezett ütemező a ThreadPool, a Taskok thread pool szálakon futnak
  - Készíthető saját ütemező is (TaskScheduler)
  - . FromCurrentSynchronizationContext – UI szál
- A threadpool helyett a Task és ParallelFor a javasolt technológia párhuzamos programozásra

Parallel osztály - Invoke

- Invoke metódus: az átadott metódusokat párhuzamosan futtatja
  - Hasonló, mintha közvetlenül ThreadPool-t használnánk
  - De ezek a műveletek megszakíthatóak
  - Szinkron hívás: megvárja amíg mind lefut

```
Parallel.Invoke(
        () => DoSomeWork(),
        () => DoSomeOtherWork()
);
```

Parallel osztály – For, ForEach

- For metódus: párhuzamosítható ciklus futtatása
  - Mettől – meddig: int32/64 ciklus változó
  - Szállokális változó: minden résztvevő szálhoz külön példány tartozik, inicializálás, finalizálás, …
  - Break: a többi szálon a ciklus mag végrehajtása az aktuális iterációig
  - Stop: azonnali megállás
  - Cancel: CancellationToken alapú megszakítás
  - Aggregált kivételkezelés

Parallel . For

```
var source = Enumerable.Range(100, 20000).ToArray();
var results = new ConcurrentStack<int>();
// Store all values below a specified threshold.
Parallel.For(0, source.Length, (i, loopState) => {
    var d = source[i];
    results.Push(d);
    if (d > 13513 )
    {
        // Might be called more than once!
        loopState.Break();
        Console.WriteLine($"Break called at iteration {i}. d = {d}");
        Thread.Sleep(1000);
    }
});
Console.WriteLine("result count {0}", results.Count());
```

Kérdések?

Albert István
ialbert@aut.bme.hu