7. fejezet

Szálkezelés és atomic támogatás a C++ standard library-ban

Grafikus Processzorok Tudományos Célú Programozása

Berényi Dániel – Nagy-Egri Máté Ferenc

Párhuzamosság a C++-ban

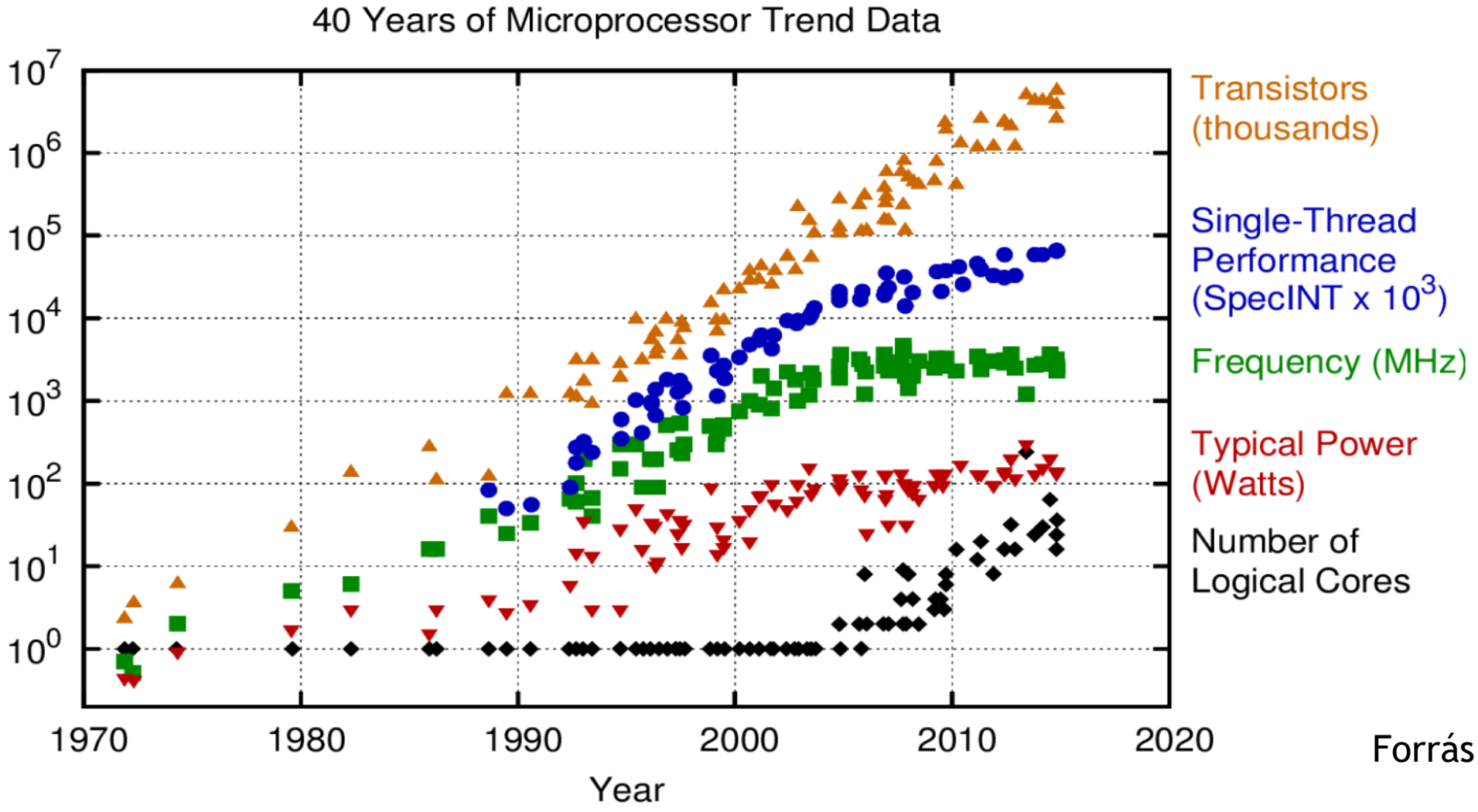
Wigner GPU Lab

Még két fontos részét nem néztük meg az STL-nek:

- Thread support library (C++11)
- Atomics operations library (C++11)

a mostani előadás minden C++ eleme legalább C++11,
csak az lesz feltüntetve ha ennél későbbi

Miért fontos a párhuzamosság?



Forrás: 1., 2.

Párhuzamosság általában

A 2000-es évek előtt a gépek azért lettek gyorsabbak, mert az órajel állandóan növekedett.

A hűtés azonban fizikailag limitált, ezért kb. 2-3 GHz fölé nem lehet praktikusan menni.

Az egyetlen esély növelni a (számítási)teljesítményt az, ha megtöbbszörözik a végrehajtókat, így az újabb tranzisztorokat párhuzamosításra fordítják.

Párhuzamosság általában

A párhuzamosítás több szinten is lehetséges:

- Bit szint
- Utasítás szint
- Vektor szint
- Feladat/Eszköz szint
- Folyamat/Klaszter szint

Párhuzamosság általában

A párhuzamosítás több szinten is lehetséges:

- Bit szint  Több bit van egy regiszterben
- Utasítás szint
- Vektor szint
- Feladat/Eszköz szint
- Folyamat/Klaszter szint

Párhuzamosság általában

A párhuzamosítás több szinten is lehetséges:

- Bit szint
- Utasítás szint
- Vektor szint
- Feladat/Eszköz szint
- Folyamat/Klaszter szint

Több utasítás hajtódik végre egyszerre a futószalagon

Párhuzamosság általában

A párhuzamosítás több szinten is lehetséges:

- Bit szint
- Utasítás szint
- Vektor szint
- Feladat/Eszköz szint
- Folyamat/Klaszter szint

Ugyan az az utasítás egyszerre több adaton hajtódik végre

Párhuzamosság általában

A párhuzamosítás több szinten is lehetséges:

- Bit szint
- Utasítás szint
- Vektor szint
- Feladat/Eszköz szint
- Folyamat/Klaszter szint

Több szálon történő végrehajtás, de van közös memória

Párhuzamosság általában

A párhuzamosítás több szinten is lehetséges:

- Bit szint
- Utasítás szint
- Vektor szint
- Feladat/Eszköz szint
- Folyamat/Klaszter szint

Folyamatok közötti feladat szétosztás, amik között nincs memória kapcsolat, csak üzenet küldés
(szerializálni kell az információt)

Párhuzamosság általában

A párhuzamosítás több szinten is lehetséges:

- Bit szint
- Utasítás szint
- Vektor szint
- Feladat/Eszköz szint
- Folyamat/Klaszter szint

Ezt a processzor és a fordító kezeli

Esetleg a vektorizálásnál néha kézzel be lehet segíteni, pl. intrinsic-ekkel

Párhuzamosság általában

A párhuzamosítás több szinten is lehetséges:

- Bit szint
- Utasítás szint
- Vektor szint
- Feladat/Eszköz szint
- Folyamat/Klaszter szint

} Ez viszont mindenképp a programozó feladata

Párhuzamosság általában

A párhuzamosítás több szinten is lehetséges:

- Bit szint
- Utasítás szint
- Vektor szint
- Feladat/Eszköz szint
- Folyamat/Klaszter szint

} Ha ezt nem használja ki az ember, akkor elpocsékolja a hardver erőforrásait és a kódja sose lesz gyorsabb, nem fog skálázódni

Párhuzamosság általában

A párhuzamosítás több szinten is lehetséges:

- Bit szint
- Utasítás szint
- Vektor szint
- Feladat/Eszköz szint ← De most csak erről lesz szó
- Folyamat/Klaszter szint

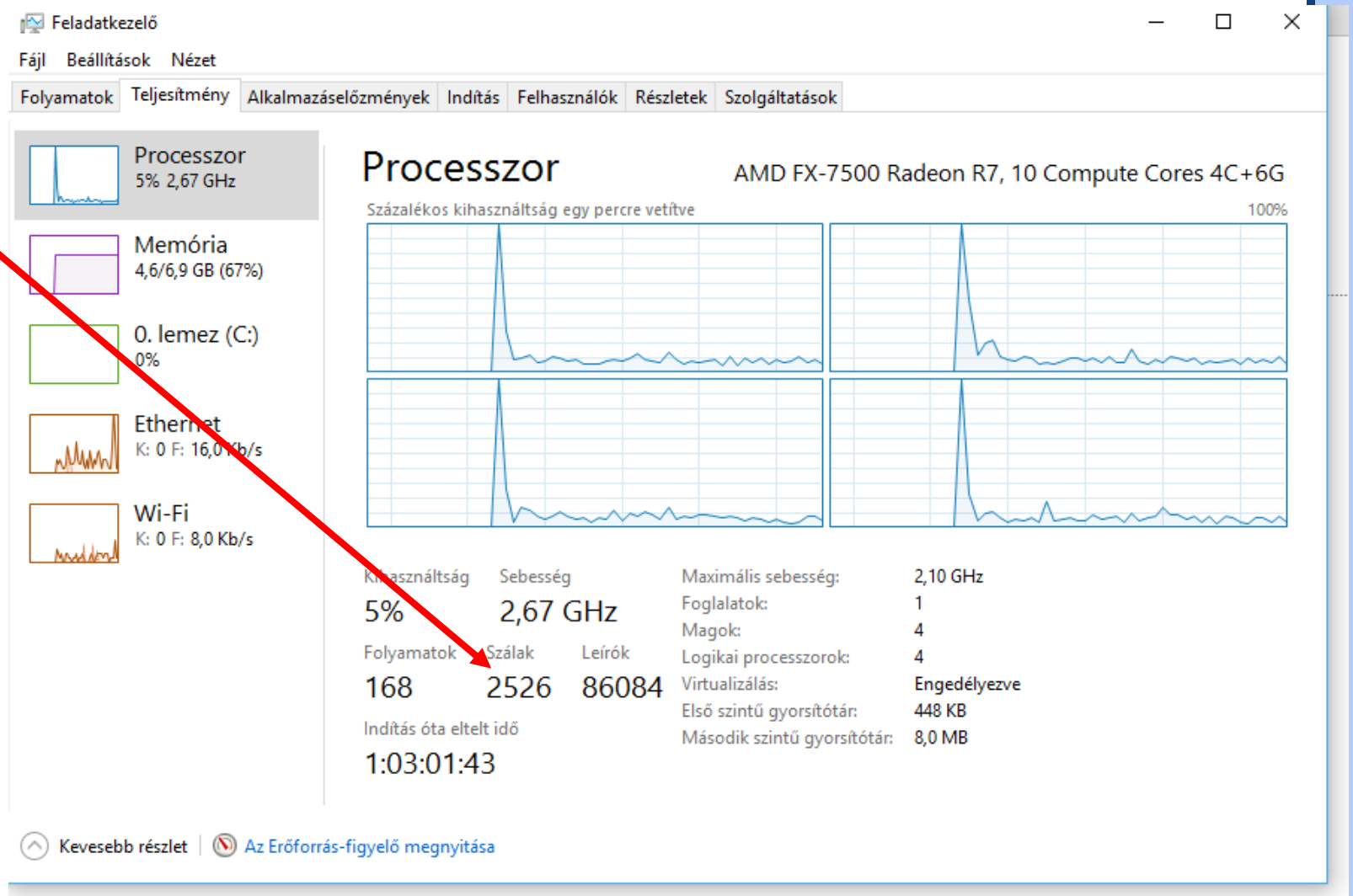
Párhuzamosság általában

Szál: a végrehajtás logikai egysége

- Saját stack-je van
- Saját állapota van
- Egy szálütemező dönti el, hogy mikor kerül sorra
  - CPU-kon az operációs rendszer része, azaz szoftveres
  - GPU-kon hardveres ütemező van
- Amikor dolgozik, akkor hardveres végrehajtóhoz van rendelve

Párhuzamosság általában

- Egy mai gép több ezer szálat kezel egyszerre
- Néhány magon...
- Hogyan?



Párhuzamosság általában

A szálak állapota lehet:

- Végrehajtás alatti
- Várakozó
- Felfüggesztett
- stb...

Csak ebben az állapotban használ végrehajtási hardver erőforrást

Párhuzamosság általában

A szálak állapota lehet:

- Végrehajtás alatti
- Várakozó
- Felfüggesztett
- stb...

Ekkor félre van téve,
és nem csinál semmit

Szálak C++-ban

- Legalább 1 szál mindig van a programunkban
- Ezt az operációs rendszer indítja el a `main()` függvénnyel
- A `main()` visszatérésekor pedig megsemmisül

Szálak C++-ban

Az aktuális függvényt végrehajtó szálat az `std::this_thread` névtérben levő függvényekkel lehet elérni:

- `std::yield`
  - Előre engedi a többi szálat, mert neki épp várnia kell valamire
- `std::sleep_for, std::sleep_until`
  - Felfüggeszti a szál végrehajtását valamennyi időre, vagy valamennyi ideig

Szálak C++-ban

Minden új szálat, amit létre hozunk az `std::thread` objektum reprezentál:

```
#include <thread>

std::thread t;
```

Szálak C++-ban

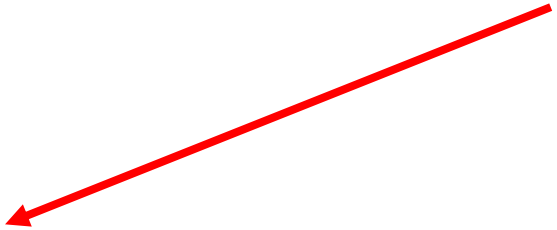
Ez a program elindít majd bevár egy szálat:

```
#include <thread>
#include <iostream>

int main()
{
	auto task = [](){ std::cout << "done.\n"; };
	std::thread t(task);
	t.join();
	return 0;
}
```

Szálak C++-ban

Ez a program elindít majd bevár egy szálat:

```
#include <thread>
#include <iostream>

int main()
{
      auto task = [](){ std::cout << "done.\n"; };
      std::thread t(task);
      t.join();
      return 0;
}
```



Ez az a lambda, amit az új szálban akarunk végrehajtani

Szálak C++-ban

Ez a program elindít majd bevár egy szálat:

```
#include <thread>
#include <iostream>

int main()
{
	auto task = [](){ std::cout << "done.\n"; };
	std::thread t(task);
	t.join();
	return 0;
}
```

A `thread` konstruktora átveszi a függvényt és elindítja a szálat

Szálak C++-ban

Ez a program elindít majd bevár egy szálat:

```
#include <thread>
#include <iostream>

int main()
{
	auto task = [](){ std::cout << "done.\n"; };
	std::thread t(task);
	t.join();
	return 0;
}
```

Itt a `main` bevárja, hogy befejeződjön a másik szál

Szálak C++-ban

Ez a program argumentumot is átad az elindított szálnak:

```
#include <thread>
#include <iostream>

int main()
{
	auto task = [](int x){ std::cout << x << "\n"; };
	std::thread t(task, 7);
	t.join();
	return 0;
}
```

Szálak C++-ban

```
std::thread t(task, 7);
```

- Ez most csak olyan szálakra működik, amik nem adnak vissza semmit
- Hogyan kapjunk vissza eredményt?

Promise és Future objektumok

```
#include <future>
std::promise<T> promise;
std::future<T> future = promise.get_future();
```

- A promise – future pár egy egyszeri aszinkron kommunikációs csatorna:
- Az a szál, amelyik eredményt akar közölni beállítja a promise-t egyszer
- A másik szál, akinek a kapcsolódó future van várhat az eredményre és ha elkészült, akkor lekérdezheti

Promise és Future objektumok


```
#include <thread>
#include <future>
int main()
{
   std::promise<int> promise;
   std::future<int> future = promise.get_future();
   auto f = [](std::promise<int> p, int x){ p.set_value(2*x); };
   std::thread t(f, std::move(promise), 21 );
   std::cout << "Result: " << future.get() << "\n";
   t.join();
}
```

Promise és Future objektumok

```
#include <thread>
#include <future>
int main()
{
    std::promise<int> promise;
    std::future<int> future = promise.get_future();
    auto f = [](std::promise<int> p, int x){ p.set_value(2*x); };
    std::thread t(f, std::move(promise), 21 );
    std::cout << "Result: " << future.get() << "\n";
    t.join();
}
```

Itt jön létre az összekapcsolt promise-future pár, ami egy `int`-et fog átvinni

Promise és Future objektumok

```
#include <thread>
#include <future>
int main()
{
    std::promise<int> promise;
    std::future<int> future = promise.get_future();
    auto f = [](std::promise<int> p, int x){ p.set_value(2*x); };
    std::thread t(f, std::move(promise), 21 );
    std::cout << "Result: " << future.get() << "\n";
    t.join();
}
```

A szál átveszi a promise-t és beállítja

Promise és Future objektumok

```
#include <thread>
#include <future>
int main()
{
    std::promise<int> promise;
    std::future<int> future = promise.get_future();
    auto f = [](std::promise<int> p, int x){ p.set_value(2*x); };
    std::thread t(f, std::move(promise), 21 );
    std::cout << "Result: " << future.get() << "\n";
    t.join();
}
```

Az elinduló szálba át kell mozgatni a promise-t

Promise és Future objektumok


```
#include <thread>
#include <future>
int main()
{
   std::promise<int> promise;
   std::future<int> future = promise.get_future();
   auto f = [](std::promise<int> p, int x){ p.set_value(2*x); };
   std::thread t(f, std::move(promise), 21 );
   std::cout << "Result: " << future.get() << "\n";
   t.join();
}
```

A fő szál a `.get()` függvényig fut, vár amíg a promise beállítódik, majd visszatér az értékkel

Aszinkron végrehajtás

Sokszor nem kell tetszőleges ponton a kommunikáció a két szál között, csak el akarunk indítani egy lambdát és visszakapni az eredményét.

Erre való az `std::async` és az std::packaged_task!

std::packaged_task

```
int main()
{
    std::packaged_task<double(double)> task(
     [](double x) { return std::sqrt(x); } );

    std::future<double> result = task.get_future();
    task(100.0);
    std::cout << "Result: " << result.get() << "\n";
}
```

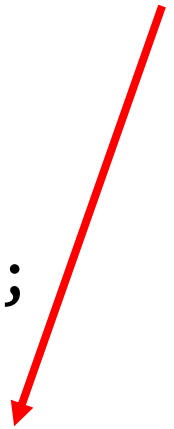
std::packaged_task

A konstruktorban adjuk meg a függvényt, akár lambda, akár függvény pointer formában

```
int main()
{
    std::packaged_task<double(double)> task(
     [](double x) { return std::sqrt(x); } );

    std::future<double> result = task.get_future();
    task(100.0);
    std::cout << "Result: " << result.get() << "\n";
}
```

std::packaged_task

Lekérdezzük a későbbi eredményt elérhetővé tevő future-t



```
int main()
{
    std::packaged_task<double(double)> task(
     [](double x) { return std::sqrt(x); } );

    std::future<double> result = task.get_future();
    task(100.0);
    std::cout << "Result: " << result.get() << "\n";
}
```

std::packaged_task

Itt indul el a függvény egy új szálban

```
int main()
{
    std::packaged_task<double(double)> task(
     [](double x) { return std::sqrt(x); } );

    std::future<double> result = task.get_future();
    task(100.0);
    std::cout << "Result: " << result.get() << "\n";
}
```

std::packaged_task

Itt várjuk meg, hogy befejeződjön

```
int main()
{
    std::packaged_task<double(double)> task(
     [](double x) { return std::sqrt(x); } );

    std::future<double> result = task.get_future();
    task(100.0);
    std::cout << "Result: " << result.get() << "\n";
}
```

std::async

```
int main()
{
    auto f = [](int x){ return 2*x; };

    auto future = std::async( std::launch::async, f, 21 );

    std::cout << "Result: " << future.get() << "\n";
}
```

std::async

Itt indul el a lambda egy új szálban, maga az async egy future-t ad vissza, ami majd az eredményt fogja tartalmazni


```
int main()
{
    auto f = [](int x){ return 2*x; };

    auto future = std::async( std::launch::async, f, 21 );

    std::cout << "Result: " << future.get() << "\n";
}
```

std::async

```
int main()
{
    auto f = [](int x){ return 2*x; };

    auto future = std::async( std::launch::async, f, 21 );

    std::cout << "Result: " << future.get() << "\n";
}
```

A "main" itt vár, hogy az eredmény elérhető legyen

Aszinkron szálkezelés

Ökölszabály:

- A legtöbbször elég az async vagy a packed_task. (a különbségekről itt)
  Elindítja a függvényt, visszaadja az eredményt.
- Ha köztes eredményt akarunk egyszeri módon átadni és szinkronizálni a szálak között, akkor érdemes promise-future párokat használni.
- Ha bonyolultabb kommunikáció kell, akkor a később bemutatandó eszközökből kell építkeznünk...

Példa: nagyvektor átlag

Feladat: számoljuk ki az átlagát egy nagy vektornak többszálon

```
std::vector<double> vec(10'000'000);


auto averager = [](auto it0, auto it1)
{
  auto sum = std::accumulate(it0, it1, 0.0);
  return sum / std::distance(it0, it1);
};
```

Példa: nagyvektor átlag

Feladat: számoljuk ki az átlagát egy nagy vektornak többszálon

```
std::vector<double> vec(10'000'000);
```

C++11: számjegy elválasztók, semmit nem változtat,
csak olvashatóvá teszi a nagy számokat

```
auto averager = [](auto it0, auto it1)
{
  auto sum = std::accumulate(it0, it1, 0.0);
  return sum / std::distance(it0, it1);
};
```

Példa: nagyvektor átlag

Feladat: számoljuk ki az átlagát egy nagy vektornak többszálon

```
std::vector<double> vec(10'000'000);
```

Segéd függvény, ez lesz egy részfeladat,
amit egy-egy szálon végre tudunk hajtani: két iterátor között részátlagot számol

```
auto averager = [](auto it0, auto it1)
{
  auto sum = std::accumulate(it0, it1, 0.0);
  return sum / std::distance(it0, it1);
};
```

Példa: nagyvektor átlag

```
auto n = std::thread::hardware_concurrency();
```

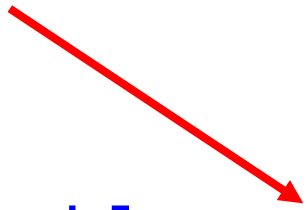
Visszaadja a párhuzamosan futtatható szálak számát
(általában a logikai magszámot)

```
std::vector<std::future<double>> futures(n);
```

Példa: nagyvektor átlag

```
auto n = std::thread::hardware_concurrency();
```

Lefoglalunk annyi eredményt tároló future-t, ahány szálunk van



```
std::vector<std::future<double>> futures(n);
```

Példa: nagyvektor átlag

A szálak elindítása: fel kell darabolni a teljes adatsort!

```
for(int k=0; k<n; ++k )
{
    auto it0 = vec.begin() +  k      * vec.size() / n;
    auto it1 = vec.begin() + (k + 1) * vec.size() / n;
    futures[k] = std::async( std::launch::async,
                             averager, it0, it1 );
}
```

Példa: nagyvektor átlag

A szálak elindítása: fel kell darabolni a teljes adatsort!

Osszuk fel a teljes intervallumot darabokra

```
for(int k=0; k<n; ++k )
{
    auto it0 = vec.begin() +  k      * vec.size() / n;
    auto it1 = vec.begin() + (k + 1) * vec.size() / n;
    futures[k] = std::async( std::launch::async,
                             averager, it0, it1 );
}
```

Példa: nagyvektor átlag

A szálak elindítása: fel kell darabolni a teljes adatsort!

```
for(int k=0; k<n; ++k )
{
    auto it0 = vec.begin() +  k      * vec.size() / n;
    auto it1 = vec.begin() + (k + 1) * vec.size() / n;
    futures[k] = std::async( std::launch::async,
                             averager, it0, it1 );
}
```

Elindítjuk az iterátorokkal a szálat és letároljuk a future-t.

Példa: nagyvektor átlag

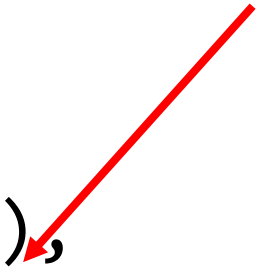
Eredmények összegyűjtése:

```
auto partial_avg =
     std::accumulate(futures.begin(),
                     futures.end(),
                     0.0,
        [](double acc, std::future<double>& f)
        {
            return acc + f.get();
        } );
```

Példa: nagyvektor átlag

Eredmények összegyűjtése:

Végigmegyünk a future tömbön...



```
auto partial_avg =
     std::accumulate(futures.begin(),
                     futures.end(),
                     0.0,
        [](double acc, std::future<double>& f)
        {
            return acc + f.get();
        } );
```

Példa: nagyvektor átlag

Eredmények összegyűjtése:

```
auto partial_avg =
     std::accumulate(futures.begin(),
                     futures.end(),
                     0.0,
        [](double acc, std::future<double>& f)
        {
            return acc + f.get();
        } );
```

Minden elemnél megvárjuk a részeredményt és összegezzük

Példa: nagyvektor átlag

És a végeredmény:

```
std::cout << "Average is: "
          << result / (double)n << "\n";
```

(ahol n a szálak száma volt☺)

A teljes kód itt.

Megosztott változók

És most kezdődik a bonyodalom...

Megosztott változók

A szálkezelés szempontjából az adatoknak két fő tulajdonsága van:

- Meg van-e osztva
- Változik-e, vagy sem

Megosztott változók

A szálkezelés szempontjából az adatoknak két fő tulajdonsága van:

- Meg van-e osztva
  Egynél több szál is hozzáfér-e ugyan ahhoz az adathoz, vagy erőforráshoz
- Változik-e, vagy sem
  Az adatot, erőforrást csak olvassa, vagy meg is változtatja a szál

Megosztott változók

	Nem megosztott	Megosztott
Konstans		
Változó		

Megosztott változók

	Nem megosztott	Megosztott
Konstans	✓	✓
Változó		

Megosztott változók

	Nem megosztott	Megosztott
Konstans	✓	✓
Változó	✓	

Megosztott változók

	Nem megosztott	Megosztott
Konstans	✓	✓
Változó	✓	!!!

Megosztott változók

- A megosztott változók kifejezetten veszélyesek
- Ha két szál egyszerre akar elérni egy megosztott változót, akkor versenyhelyzetről, *race condition*-ról beszélünk
- Kifejezetten az írási / módosítási versenyhelyzetek veszélyesek, mert megsérülhet, értelmetlenné válhat az érintett adat

Megosztott változók

- A megosztott változók kifejezetten veszélyesek

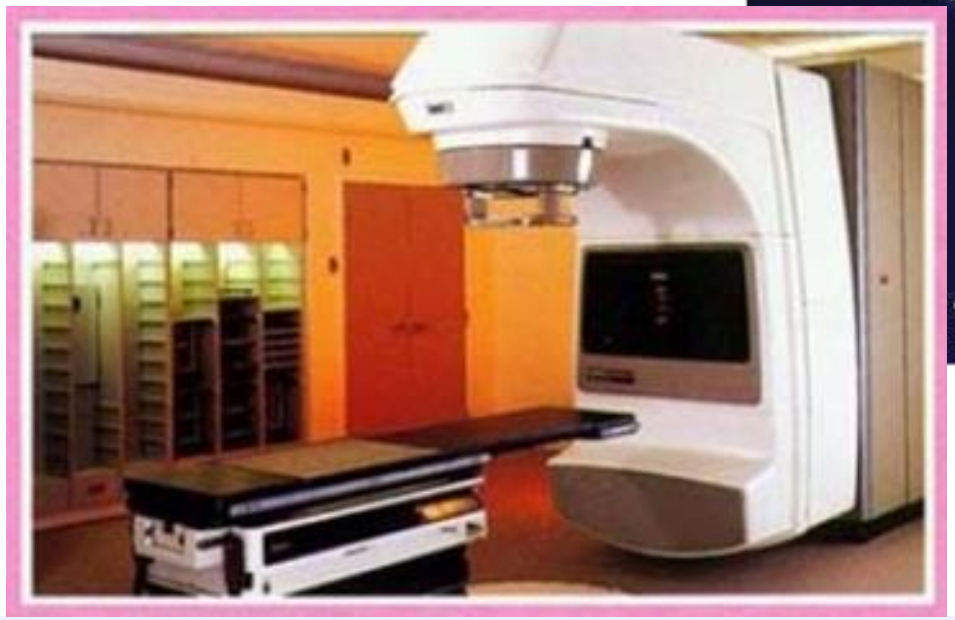
Továbbá!

- Mivel a szálak végrehajtási sorrendje és időzítése lényegében véletlen, ezért a versenyhelyzetek nagyon nehezen reprodukálhatóak és debuggolhatóak!

Megosztott változók

Emlékeztetőül a bevezető előadásból:

- A Therac-25-ös terápiás készülék hibája
- A 2003-as Észak-Amerikai áramszünet



Megosztott változók

Több módon is elkerülhetjük ezeket a problémákat:

- Nem használunk megosztott változókat!
- Használjuk a megfelelő szinkronizációs és szálvezérlési elemeket...

Szinkronizáció C++-ban

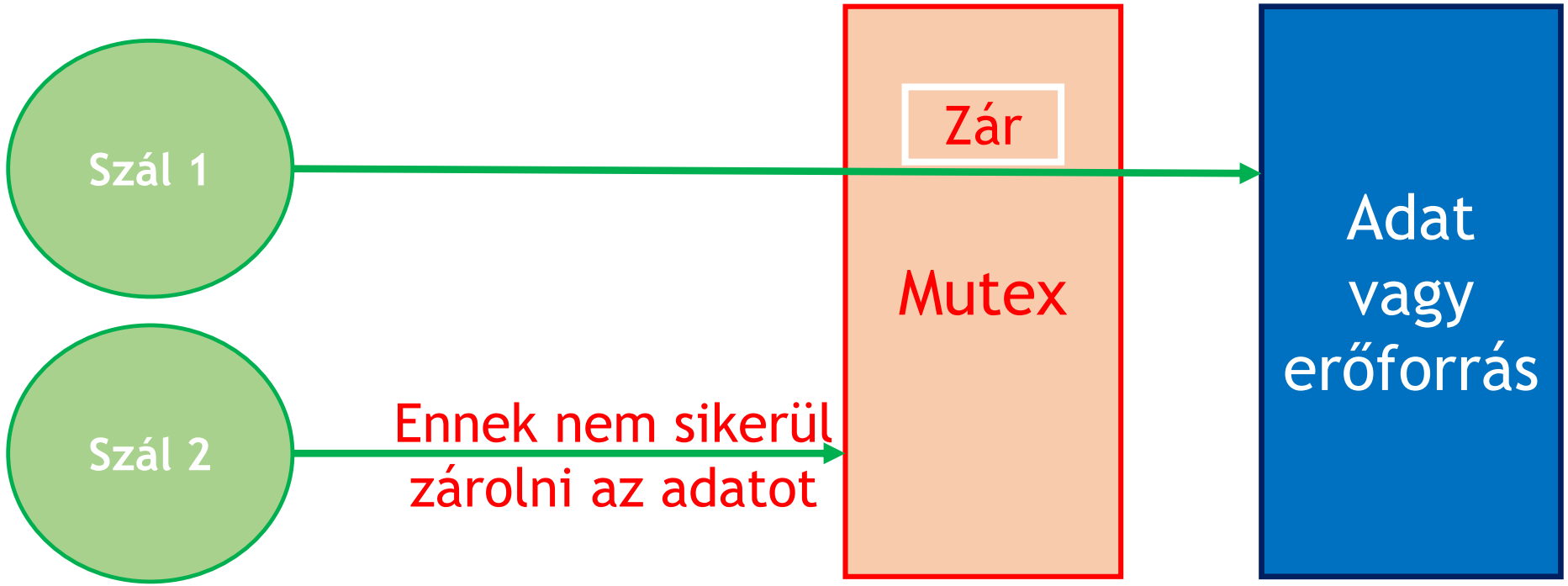
A C++ 11 a következő elemeket szabványosította:

- Mutexek
- Zárak (Locks)
- Feltételes változók (condition variables)

Mutexek

A mutex a kölcsönös kizárás rövidítése (mutual exclusion):

- Meggátolja, hogy a megosztott adatot egyszerre egynél több szál használhassa



Mutexek

A C++11 a következő fajta mutexeket szabványosította:

- `std::mutex` csak két művelet: zárás és feloldás
- `std::timed_mutex` ez adott ideig próbálkozik a zárással
- `std::recursive_mutex` ez többször is zárolható, de ugyanannyiszor kell utána feloldani
- `std::recursive_timed_mutex` mindkét előző funkció egyben

Mutexek és zárak

A mutexek csak a történet egyik fele.

A legbiztosabb zárakkal együtt használni őket:

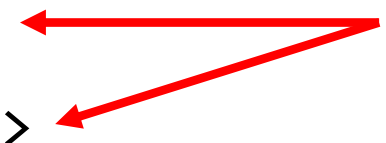
- `std::lock_guard< >`
- `std::unique_lock< >`

Mutexek és zárak

A mutexek csak a történet egyik fele.

A legbiztosabb zárakkal együtt használni őket:

- `std::lock_guard< >`
- `std::unique_lock< >`



A template paraméter a mutex típusa

Mutexek és zárak

`std::lock_guard< >` ez is RAII alapú

- a konstruktor zár,
- a destruktor feloldja a zárat.

Általában egy { } tartományban szokás használni:

```
std::mutex m;
{   //egy tartomány, pl. függvény törzse
    std::lock_guard<std::mutex> lock(m);
    //itt lehet használni a mutexhez rendelt megosztott változókat

}  //ezen a ponton a zár kiold, a mutex újra zárható másik szálból
```

Mutexek és zárak

Példa: megosztott vektor biztonságos átméretezése:

```
std::mutex mutex;
std::vector<int> data;

{
	std::lock_guard<std::mutex> guard(mutex);
	data.resize( 100 );
}
```

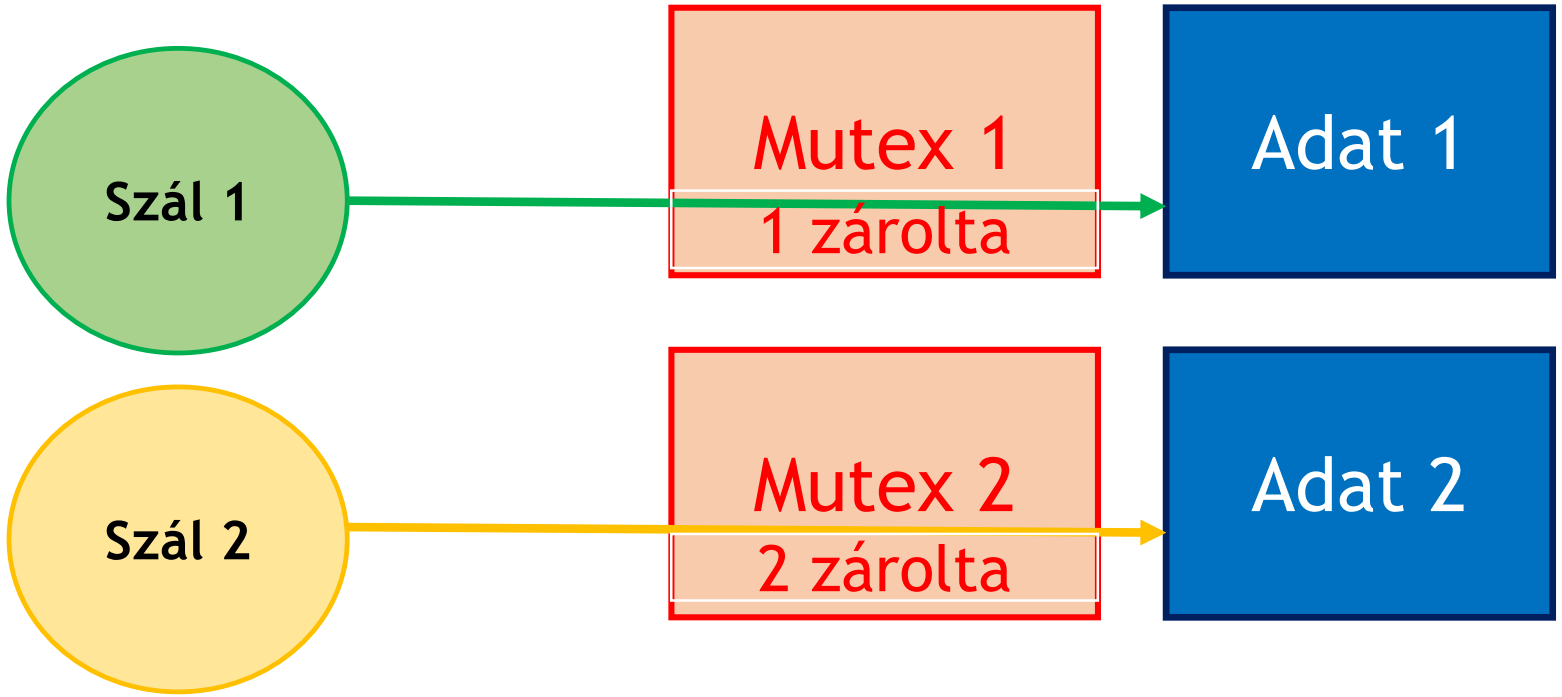
Mutexek és zárak

Az `std::unique_lock< >` egy segéd objektum a mutexekhez, ami:

- Mozgatható és értéküladható
- Szintén RAII alapú
- De olyan interfésze van, mint a mutexnek (zárható és feloldható)
- +még sokminden

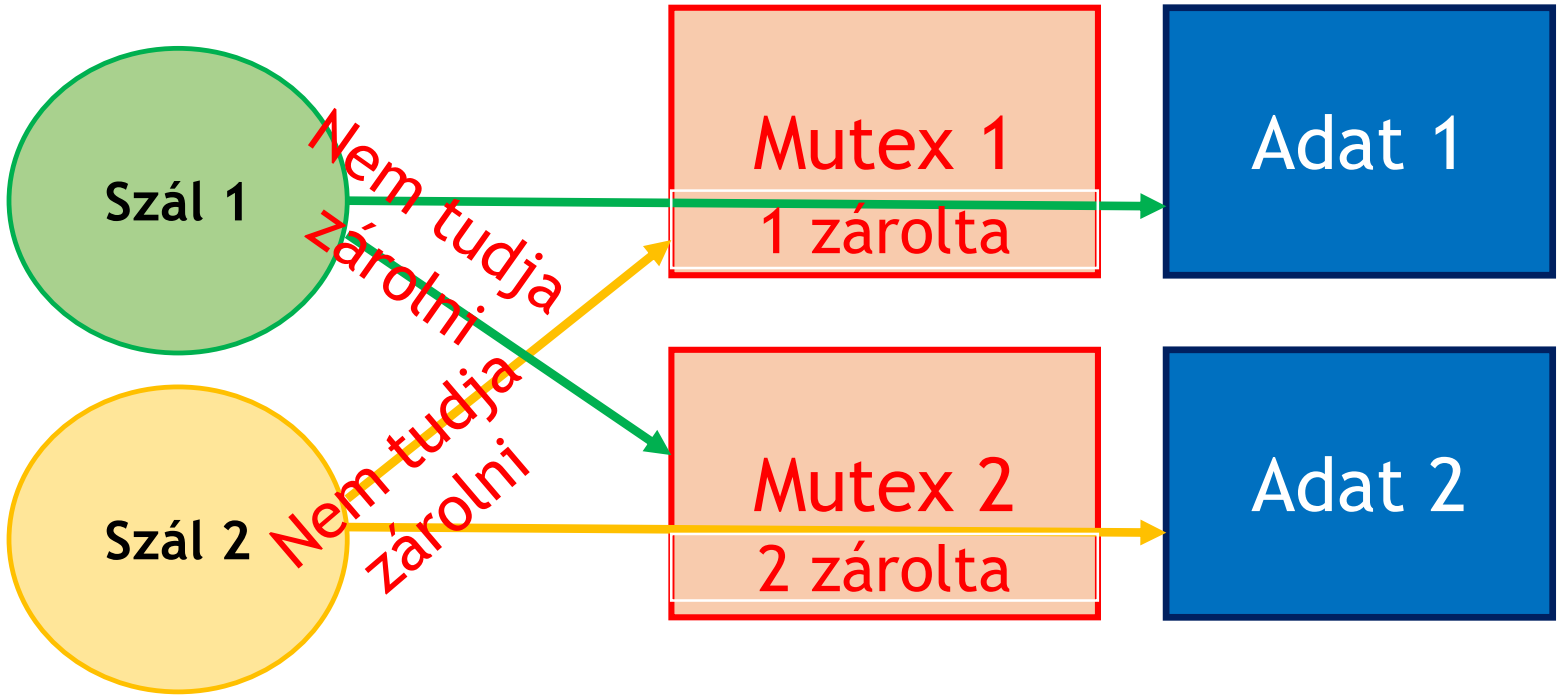
Mutexek és holtpontok

Ha egyszerre több mutexet is zárni akarunk, az nem megy egyszerre (atomikusan), és ha több szál próbálkozik egyszerre, más sorrendben, akkor előfordulhat holtpont (deadlock):



Mutexek és holtpontok

Ha egyszerre több mutexet is zárni akarunk, az nem megy egyszerre (atomikusan), és ha több szál próbálkozik egyszerre, más sorrendben, akkor előfordulhat holtpont (deadlock):



Mutexek és holtpontok

Léteznek holtpont elkerülő algoritmusok (ha egy mutexet a sok közül nem sikerül zárolni, akkor minden korábbi zárolt mutexet fel kell oldani az újrapróbálkozás előtt),

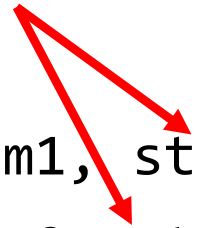
de ezeket nem kell feltétlenül ismerni, mert az `std::lock` tudja őket!

Mutexek és holtpontok

Példa az `std::lock` használatára több mutex biztonságos zárolása esetére:

```
std::mutex m1, m2;

{
        std::unique_lock<std::mutex> lock1(m1, std::defer_lock);
        std::unique_lock<std::mutex> lock2(m2, std::defer_lock);

        std::lock(lock1, lock2);
        //most lehet a védett adatokat használni
}
```

Nem a konstruktorban akarunk lockolni, hanem később!



Mutexek és holtpontok

Példa az `std::lock` használatára több mutex biztonságos zárolása esetére:

```
std::mutex m1, m2;

{
	std::unique_lock<std::mutex> lock1(m1, std::defer_lock);
	std::unique_lock<std::mutex> lock2(m2, std::defer_lock);

	std::lock(lock1, lock2);
	//most lehet a védett adatokat használni
}
```

Itt lesz a zárolás, és nem lesz deadlock!

Kiterjesztések

C++17-ben az előző példa egyszerűbben is megoldható az std::scoped_lock-al:

```
std::mutex m1, m2;

{
      std::scoped_lock<std::mutex, std::mutex> lock{m1, m2};
      //most lehet a védett adatokat használni
}
```

Kiterjesztések

A C++14 és C++17 tovább bővíti a készletet:

```
std::shared_mutex
std::shared_timed_mutex
std::shared_lock<   >
```

A "shared" arra utal, hogy ezeknek létezik nem kizárólagos módja is a kizárólagos mellett:

- Több szál is érvényesíthet megosztott zárást (pl.: többen akarják olvasni az adatot)
- De csak 1 szál szerezhet kizárólagos lock-ot (pl.: írási jogot)
- Minden megosztott lock-ot fel kell oldani, mielőtt a kizárólagos jogot megkaphatja valaki

Feltételes változók

Az `std::condition_variable` két funkcionalitást ötvöz:

- mutex szerű kizárást
- várakozást egy másik szál általi módosításra

Feltételes változók

`std::condition_variable:`

A módosító szál:	A várakozó szálak:
1.: Megszerzi a mutexet	1.: Megszerzik a zárat a mutexhez
2.: Módosítja a megosztott adatot	2.: Várakoznak a zár feloldódására a `condition_variable`-n keresztül (ezalatt a mutex nincs lockolva)
3.: Feloldja a mutexet és értesíti a többi szálat	3.: Amikor felébrednek, akkor övék a mutex

Feltételes változók

Példa:

```
std::condition_variable cv; std::mutex m;

//Író szál:
{
   std::unique_lock<std::mutex> lock(m);
   //a megosztott adat módosítása
   cv.notify_all();
}
```

```
//Olvasó / várakozó szálak:
{
   std::unique_lock<std::mutex> lock(m);
   cv.wait(lock);
   //itt használhatják a megosztott adatot
}
```

Feltételes változók

Az garantált, hogy csak azok a szálak kerülnek a felébresztési listába, akik az előtt kezdték el a várakozást, hogy az értesítés (`notify`) megtörtént.

Viszont: az nincs garantálva, hogy csak akkor ébrednek fel a szálak, ha valóban történt értesítés! Ez a jelenség a hamis ébresztés (*spurious wake-up*).
Ennek a hatékony szálütemezés implementálásával összefüggő okai vannak (link).

Hamis ébresztés

A hamis ébresztések kezelésére ajánlott ellenőrizni egy feltételt, ami megmondja, hogy valóban megtörtént-e a várt módosítás. Erre való a `wait` egy változata, ami ilyen feltételt vár:

```
template< class Predicate >
void std::condition_variable::wait
( std::unique_lock<std::mutex>& lock, Predicate pred )
{
    while (!pred()){ wait(lock); }
}
```

Atomikus műveletek

- Atomikus műveletek

Atomikus műveletek

Emlékeztető:
egy művelet akkor atomikus, ha másik szál nem tudja félbeszakítani.

- Csak néhány egyszerű típushoz érhető el néhány egyszerű művelet
- Ezeket a hardvernek kell biztosítania utasítás szinten
- Az összes szálkezeléssel összefüggő primitív közül ezek a legolcsóbbak (de persze drágábbak, mintha simán elvégeznénk a műveleteket)

Atomikus műveletek

C++11 óta van egy általános template: `std::atomic<T>`

A template maga bármilyen típusra példányosítható, de csak az egyszerű típusok esetén fogja működését atomikus műveletekkel végezni, egyéb esetekben lock-okat használ

Le lehet kérdezni, hogy melyiket választja az implementáció

Atomikus műveletek

A következő műveletek végezhetőek el az `std::atomic<T>`-n:

- Betöltés és kimentés
- Csere, összehasonlítás-és-csere
- Összeadás, kivonás
- Logikai műveletek: ÉS, VAGY, KIZÁRÓ-VAGY

Atomikus műveletek

A következő műveletek végezhetőek el az `std::atomic<T>`-n:

- Betöltés és kimentés
- Csere, összehasonlítás-és-csere
- Összeadás, kivonás
- Logikai műveletek: ÉS, VAGY, KIZÁRÓ-VAGY

} Ezek operátorként is elérhetőek

Atomikus műveletek

Az `std::atomic<T>` nem másolható a copy konstruktor és az `operator=` törölve van.

Ez azt jelenti, hogy nem hozhatunk létre egyszerűen tárolókat belőle, pl.: ez nem fog menni:

```
std::vector<std::atomic<T>>!
```

De, ezt meg lehet kerülni...

Atomikus műveletek

```
template <typename T>
struct atomic_wrapper
{
  std::atomic<T> data;
  atomic_wrapper():data(){}
  atomic_wrapper(atomic_wrapper const& copy) :
                                  data(copy.data.load()){}
  atomic_wrapper& operator=(atomic_wrapper const& copy)
  {
    data.store(copy.data.load());
    return *this;
  }
};
```

Atomikus műveletek

```
template <typename T>
struct atomic_wrapper
{
  std::atomic<T> data;
  atomic_wrapper():data(){}
  atomic_wrapper(atomic_wrapper const& copy) :
                                    data(copy.data.load()){}
  atomic_wrapper& operator=(atomic_wrapper const& copy)
  {
    data.store(copy.data.load());
    return *this;
  }
};
```

Atomikus műveletek

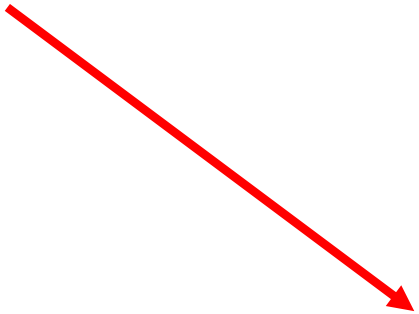
Ezzel most már lehet tárolót használni, mert az `atomic_wrapper<T>` már másolható.

```
std::vector<atomic_wrapper<T>> v;
```

Atomikus műveletek - példa

Példa: hisztogram számolás több szálon

```
//adatok, ismert terjedelemmel:
double low  = 0.0;
double high = 10.0;
std::vector<double> vec = { ... };


//hisztogram:
std::vector<atomic_wrapper<int>> histogram(30);
```



Atomikus műveletek - példa

Ez a segéd függvény egy iterátorokkal adott tartományt kategorizál be:

```
auto insert = [&](auto it0, auto it1)
{
  for( auto it=it0; it!=it1; ++it)
  {
    auto sz = histogram.size()-1;
    auto index = (size_t)(sz * (((*it)-low)/(high-low)));
    if( index > 0 && index < sz )
    {
      histogram[index].data += 1;
    }
  }
};
```

Atomikus műveletek - példa

Ez a segéd függvény egy iterátorokkal adott tartományt kategorizál be:

```
auto insert = [&](auto it0, auto it1)
{
  for( auto it=it0; it!=it1; ++it)
  {
    auto sz = histogram.size()-1;
    auto index = (size_t)(sz * (((*it)-low)/(high-low)));
    if( index > 0 && index < sz )
    {
      histogram[index].data += 1;
    }
  }
};
```

Atomikus műveletek - példa

A szálak elindítása hasonló a nagy vektor átlaghoz:

```
std::vector<std::future<void>> futures(max_num_of_threads);
for(int n=0; n<max_num_of_threads; ++n )
{
  auto it0 = vec.begin() +  n      * vec.size() / max_num_of_threads;
  auto it1 = vec.begin() + (n + 1) * vec.size() / max_num_of_threads;
  futures[n] = std::async( std::launch::async, insert, it0, it1 );
}
//itt várjuk be az elindított szálakat:
std::for_each(futures.begin(), futures.end(),
              [](std::future<void>& f){ f.get(); } );
```

Atomikus műveletek - példa

Teszt paraméterek:

- 10 millió adatpont
- 4 szál
- 30 bin a hisztogrammban

A teljes kód itt.

Eredmény:

- Ha nem használunk atomikus műveleteket, akkor kb. 3 millió adatpont nem kerül bele a hisztogramba az írási versenyhelyzetek miatt.
  A feldolgozási idő ekkor: kb. 470 ms
- Atomikus műveletekkel helyes az eredmény, de kicsit tovább tart: kb. 1070 ms

Kitekintés

A C++ nyelv fejlesztői számos kiegészítő / kísérleti könyvtárat is fejlesztenek, amik egy ponton bekerülhetnek a standard library-ba (pl. a C++20-ban):

Library fundamentals

Library fundamentals 2

Extensions for Parallelism

Extensions for Concurency

Concepts

Ranges

Kitekintés

Vannak dolgok, amik még nincsenek és a közeljövőben valószínűleg nem is kerülnek szabványosításra. Két példa:

- Szál prioritás
- Szál affinitás

Kitekintés

- Szál prioritás
  - A szálkezelő milyen gyakran adjon utat az adott szálnak. Ha kicsi, ritkán ébreszti fel, ha nagy, akkor gyakran. Ha sok nagy prioritású szál van, akkor azok teljesen kisajátíthatják a rendszert.

Linux: `pthread_setschedprio`

Windows: `SetThreadPriority`

A szükséges szálazonosító az `std::thread native_handle` függvényéből nyerhető.

Kitekintés

- Szál affinitás
  - A szálkezelő a szálakat átrakhatja a processzorok között. Ekkor az L1/L2 vagy a teljes cache elveszik a szál számára, ami költséges. A szál affinitással megmondható az ütemezőnek, hogy a szál melyik logikai processzoron fusson, elkerülve az ilyen átrakásokat

Linux: `pthread_setaffinity`

Windows: `SetThreadAffinityMask`

Olvasnivaló

A szükséges szálazonosító az `std::thread native_handle` függvényéből nyerhető.