Multicore

amborodin@acm.org

Салат асинхронный

Салат "Асинхронный": помиогурдоры,цымайон,ез.

Salad "Asynchronous": tomacucumtoes,bersmayonn,aise

Terminology

- Concurrency

Computation may interleave with other computations

- Multitasking

Device may execute more than one program at a time

- Parallel execution

Device is capable of advancing more that one computation in a point of time

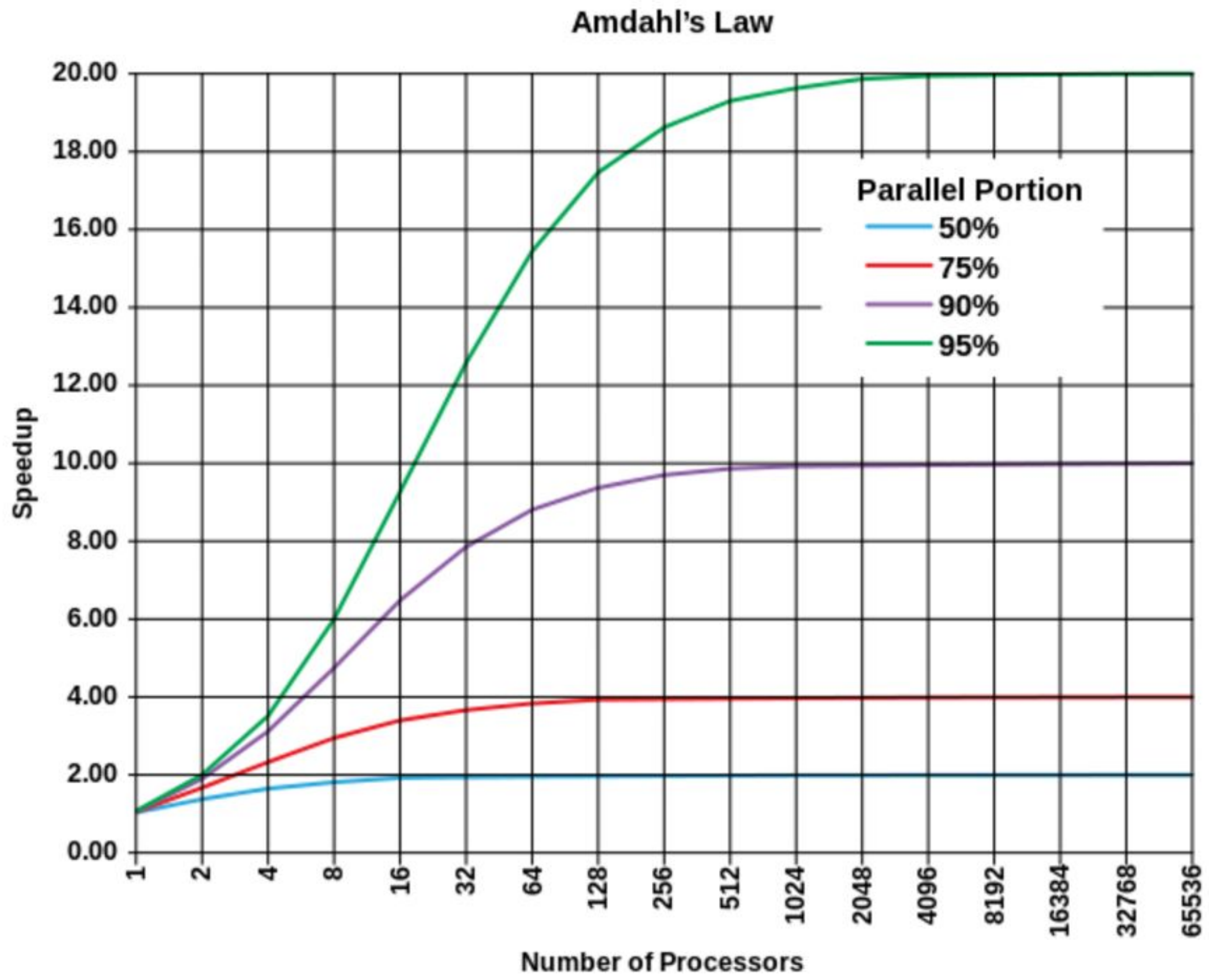
- Multithreading

Program is represented as a set of worker threads

- Asynchrony

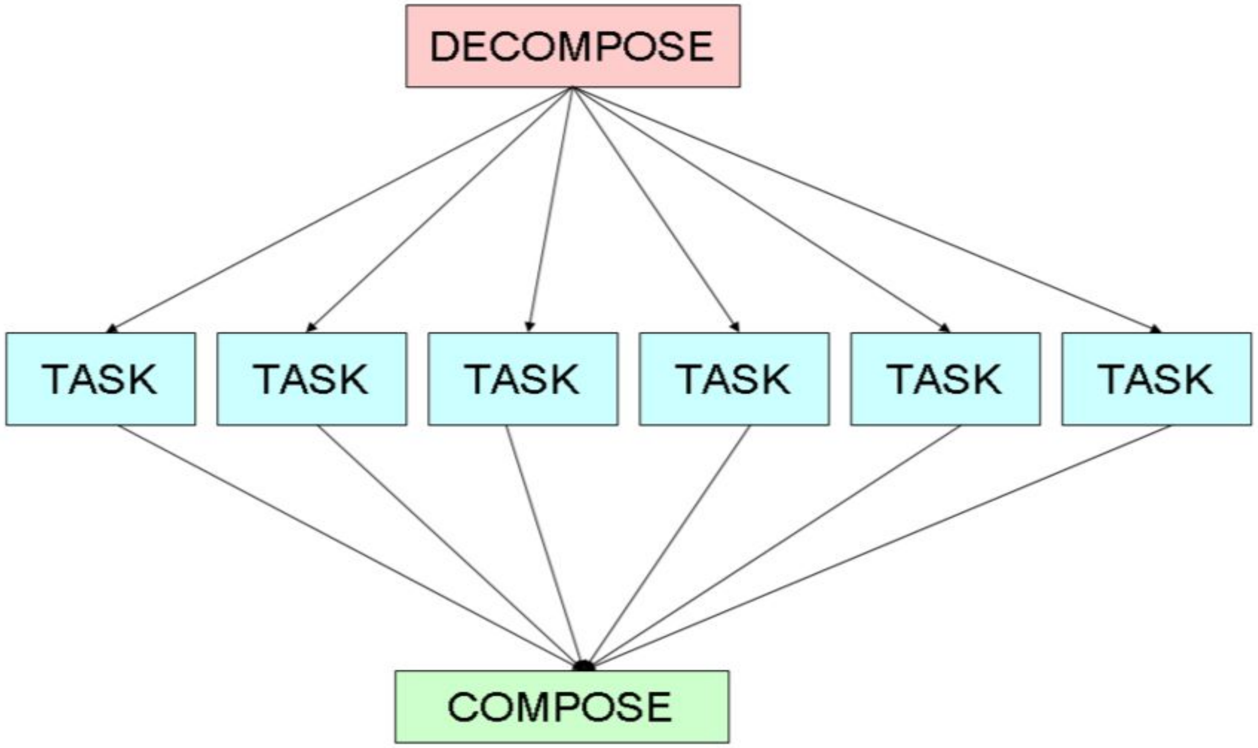
Program contain non-blocking calls

Amdahl law: so we have many CPUs



Embarrassingly parallel problems

- Solved mostly by SIMD



Hardware: Atomic operations

- Load-link/store-conditional
- Compare-and-swap

Hardware: Fences

- SFENCE
- LFENCE
- MFENCE

Processor #1:

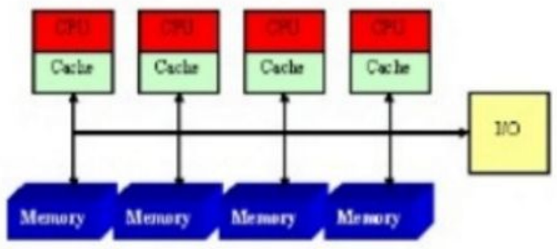
```
while (f == 0);
// Memory fence required here
print x;
```

Processor #2:

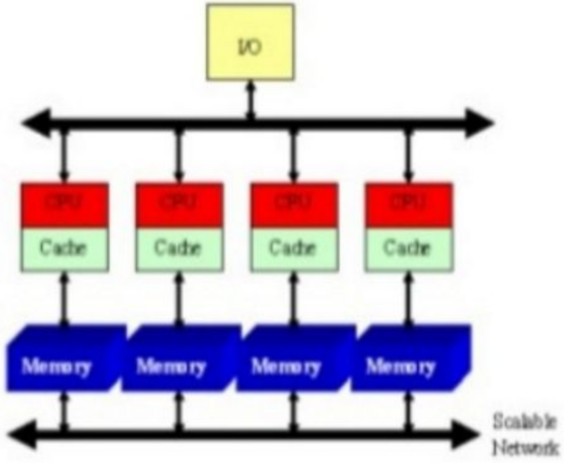
```
x = 42;
// Memory fence required here
f = 1;
```

Hardware: Non-uniform memory architecture

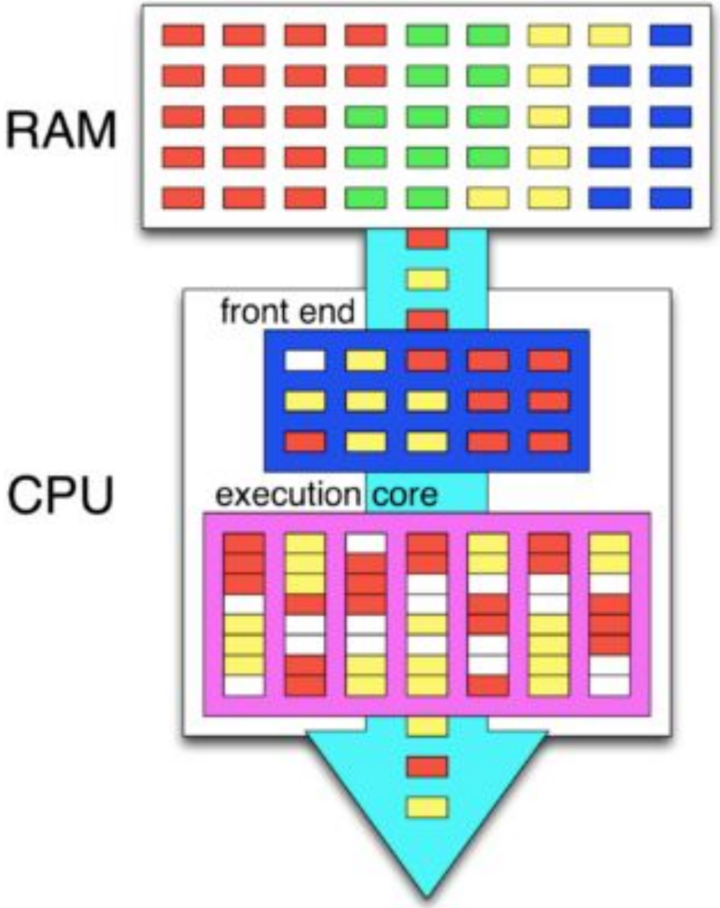
- **Uniform memory access**(UMA): all processors have same latency to access memory. This architecture is scalable only for limited nmber of processors.



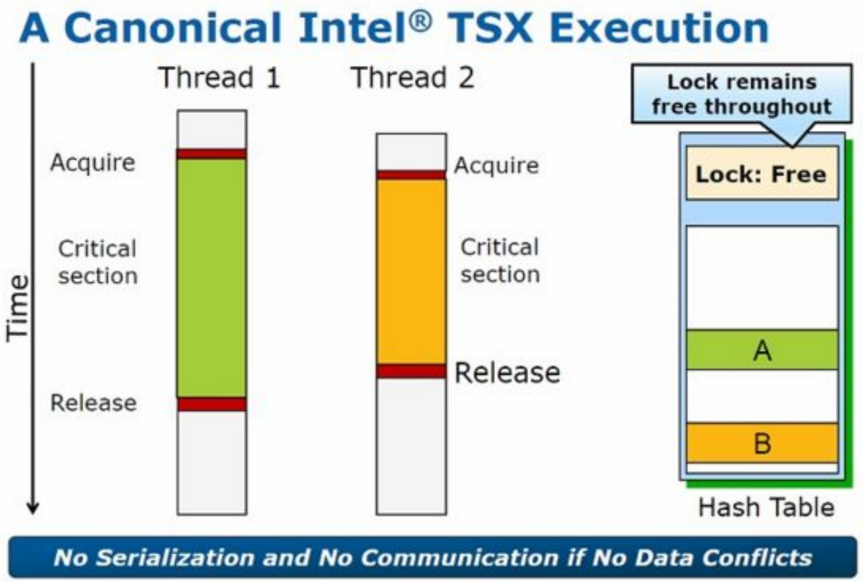
- **Nom Uniform Memory Access**(NUMA): each processor has its own local memory, the memory of other processor is accessible but the lantency to access them is not the same which this event called " remote memory access"



Hardware: Hyper Threading



Hardware: Intel's Transactional Synchronization Extensions



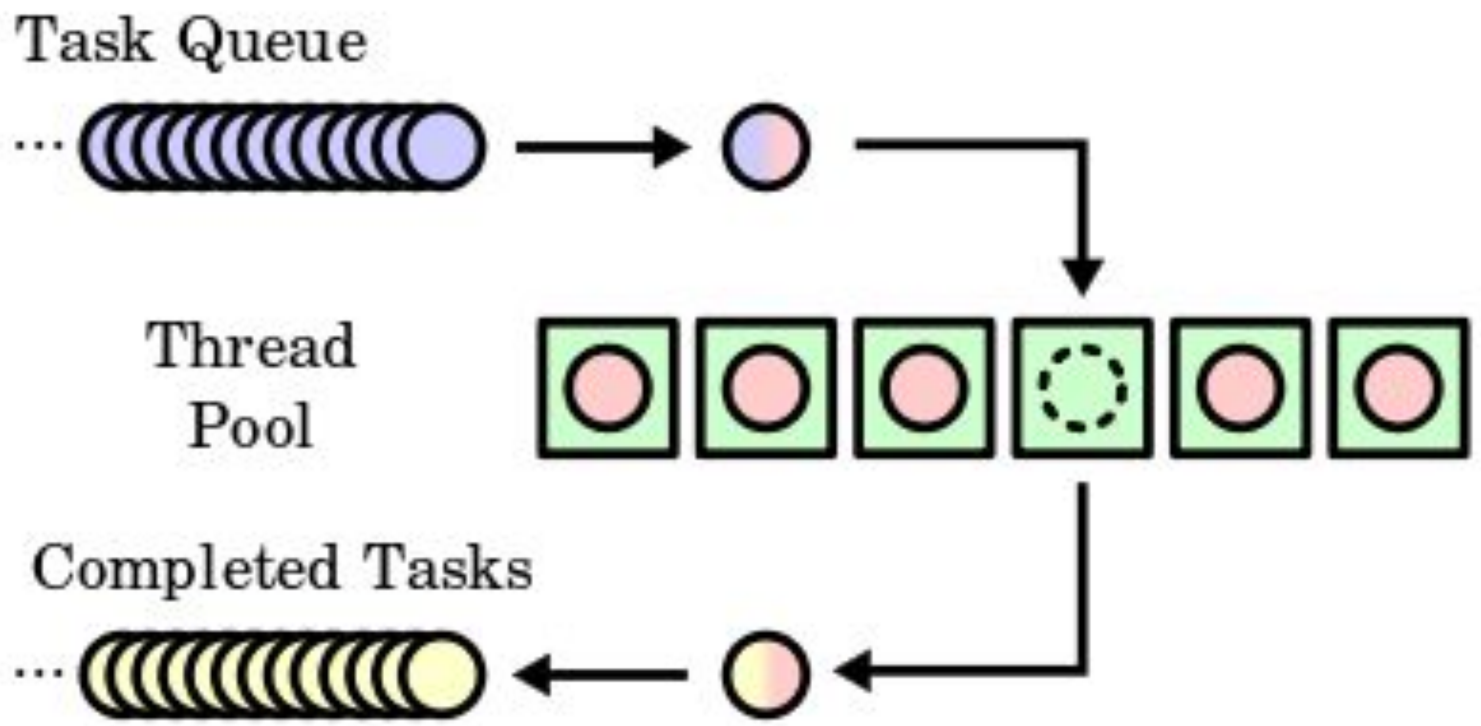
Intel® TSX Operational Aspects

1. **Identify and elide**
   - Identify critical section, start transactional execution
   - Elide locks, keep them available to other threads
2. **Execute transactionally**
   - Manage all transactional state updates
3. **Detect conflicting memory accesses**
   - Track data accesses, check for conflicts from other threads
4. **Abort or commit**
   - Abort discards all transactional updates
   - Commit makes transactional updates instantaneously visible

Spinlock\Futex\Crit Section\Fast Mutual Exclusion

- Context of exclusion
- Reliable exclusion mechanics
- Inter-process communication
- Waiting time work (spin\sleep\pump)
- Reentrancy
- Deadlock detection

Thread pool pattern



Reader-writer lock pattern

- Lock for read (many at a time)
- Lock for write (one at a time, excluding all readers)
- Upgradable: escalate from reader to writer

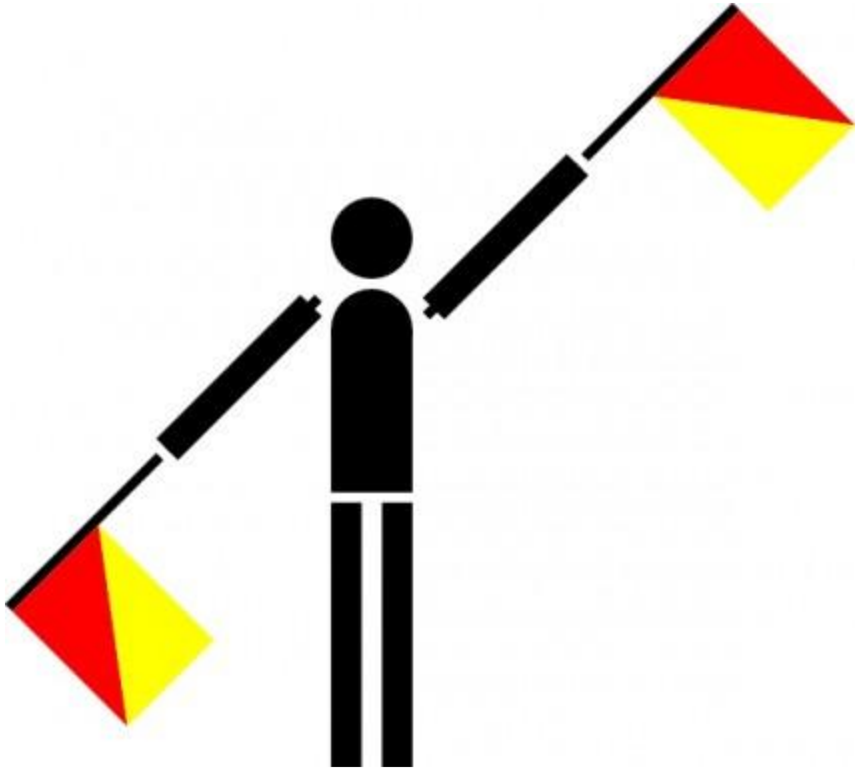
- Is sync primitives is really so hard?
- .NET 2.0 implementation had a deadlock bug

- Starvation is possible

Event pattern

- Simple event
- Autoreset event
- Countdown event \ rundown protection

Semaphore



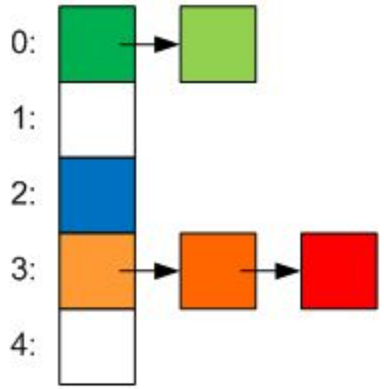
Double-checked locking

```
public class Singleton {
  private static volatile Singleton instance;

  public static Singleton getInstance() {
    if(instance == null) {
      synchronized(Singleton.class) {
        if(instance == null) {
          instance = new Singleton();
        }
      }
    }
    return instance;
  }
}
```

Striped locking

- Used in CuncurrentDictionary

```
private void GetBucketAndLockNo(int hashcode, out int bucketNo, out int lockNo, int bucketCount, int lockCount)
{
    bucketNo = (hashcode & 0x7fffffff) % bucketCount;
    lockNo = bucketNo % lockCount;
}
```



Pulse\Wait

A "Worker" thread:

```
lock(phone) // Sort of "Turn the phone on while at work"
{
    while(true)
    {
        Monitor.Wait(phone); // Wait for a signal from the boss
        DoWork();
        Monitor.PulseAll(phone); // Signal boss we are done
    }
}
```

A "Boss" thread:

```
PrepareWork();
lock(phone) // Grab the phone when I have something ready for the worker
{
    Monitor.PulseAll(phone); // Signal worker there is work to do
    Monitor.Wait(phone); // Wait for the work to be done
}
```

Producer-consumer synchronization

```
class ProducerConsumer<T> {
    private T value;
    private bool isEmpty = true;

    public void Produce(T t) {
        lock (this) {
            while (!isEmpty) {
                Monitor.Wait(this);
            }
            this.value = t;
            isEmpty = false;
            Monitor.Pulse(this);
        }
    }

    public T Consume() {
        lock (this) {
            while (isEmpty) {
                Monitor.Wait(this);
            }

            isEmpty = true;
            Monitor.Pulse(this);
            return this.value;
        }
    }
}
```

Futures and promises

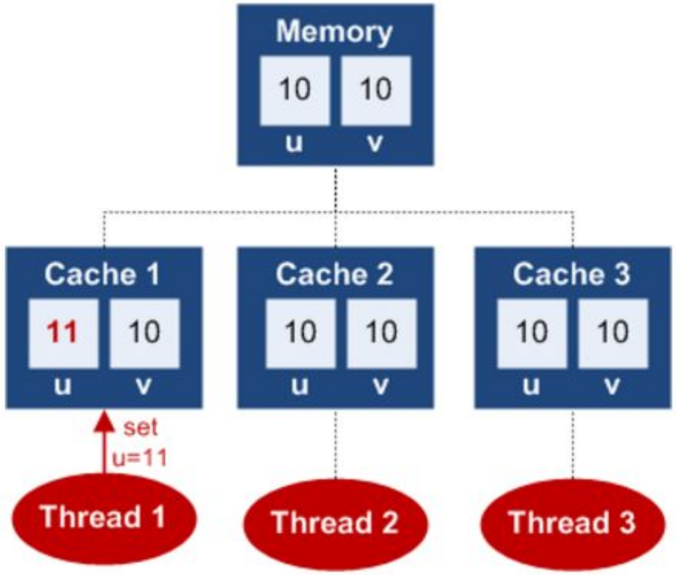
```
auto promise = std::promise<std::string>();

auto producer = std::thread([&]
{
    promise.set_value("Hello World");
});

auto future = promise.get_future();

auto consumer = std::thread([&]
{
    std::cout << future.get();
});

producer.join();
consumer.join();
```

Futures and promises

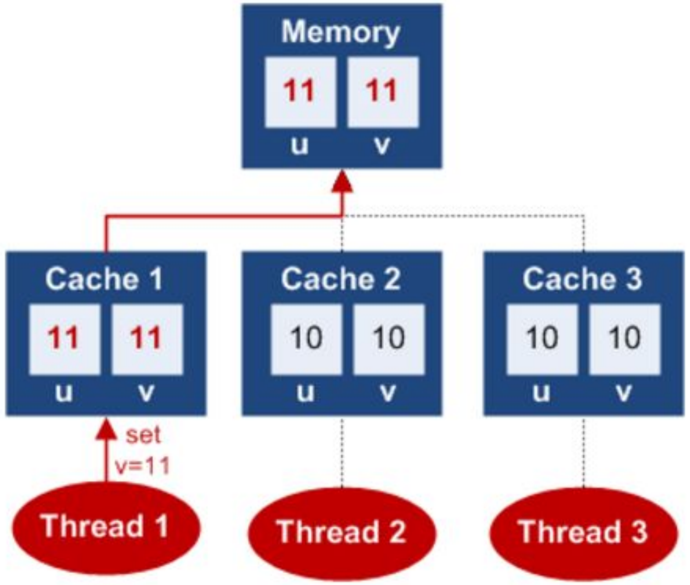
```
{
    // future from a packaged_task
    std::packaged_task<int()> task([](){ return 7; }); // wrap the function
    std::future<int> f1 = task.get_future();  // get a future
    std::thread(std::move(task)).detach(); // launch on a thread

    // future from an async()
    std::future<int> f2 = std::async(std::launch::async, [](){ return 8; });

    // future from a promise
    std::promise<int> p;
    std::future<int> f3 = p.get_future();
    std::thread( [](std::promise<int>& p){ p.set_value(9); },
                 std::ref(p) ).detach();

    std::cout << "Waiting...";
    f1.wait();
    f2.wait();
    f3.wait();
    std::cout << "Done!\nResults are: "
              << f1.get() << ' ' << f2.get() << ' ' << f3.get() << '\n';
}
```

volatile semantics

- Non-volatile write



Volatile write



There is no such thing as thread cache. This is an abstraction over compilers and hardware optimizations.
Source: http://igoro.com/archive/volatile-keyword-in-c-memory-model-explained/

volatile is not atomic

- volatile is not atomic
- volatile is not atomic
- volatile is not atomic
- volatile is not atomic
- volatile is not atomic
- volatile is not atomic
- volatile is not atomic
- volatile is not atomic

Examples of optimizations prevented by volatile semantics

- Register allocation
- Out-of-order execution
- Loop fusion
- Invariant hoisting
- Rematerialization

```
int i, a[100], b[100];
for (i = 0; i < 100; i++)
  a[i] = 1;
for (i = 0; i < 100; i++)
  b[i] = 2;
```

```
int i, a[100], b[100];
for (i = 0; i < 100; i++)
{
  a[i] = 1;
  b[i] = 2;
}
```

- Almost any compiler, JIT or CPU optimization
- https://en.wikipedia.org/wiki/Optimizing_compiler
- https://en.wikipedia.org/wiki/Program_optimization

Memory model

- Example: .NET memory model

Construct	Refreshes thread cache before?	Flushes thread cache after?	Notes
Ordinary read	No	No	Read of a non-volatile field
Ordinary write	No	**Yes**	Write of a non-volatile field
Volatile read	**Yes**	No	Read of volatile field, or Thread.VolatileRead
Volatile write	No	**Yes**	Write of a volatile field – same as non-volatile
Thread.MemoryBarrier	**Yes**	**Yes**	Special memory barrier method
Interlocked operations	**Yes**	**Yes**	Increment, Add, Exchange, etc.
Lock acquire	**Yes**	No	Monitor.Enter or entering a lock {} region
Lock release	No	**Yes**	Monitor.Exit or exiting a lock {} region

Java memory model is base on “happens before” memory model
C++ introduced memory model in C++11, most of questions were not even undefined behavior.
There is no such thing as thread cache. This is an abstraction over compilers and hardware optimizations.

Memory model

- Search for memory model of your platform\language

Non-blocking algorithms

- Obstruction-free
- Lock-free
- Wait-free

Lock-free stack: Interlocked Singly Linked List

```
template <class Entry>
class LockFreeStack{
    struct Node{
        Entry* entry;
        Node* next;
    };

    Node* m_head;

    void Push(Entry* e){
        Node* n = new Node;
        n->entry = e;
        do{
            n->next = m_head;
        }while(!CompareAndSwap(&m_head, n->next, n));
    }
```

```
    Entry* Pop(){
        Node* old_head;
        Entry* result;
        do{
            old_head = m_head;
            if(old_head == NULL){
                return NULL;
            }
        }while(!CompareAndSwap(&m_head, old_head, old_head->next));

        result = old_head->entry;
        delete old_head;
        return result;
    }
}
```

Based on work by Alex Skidanov https://habrahabr.ru/post/174369/

Lock-free stack: Interlocked Singly Linked List

```cpp
template <class Entry>
class LockFreeStack{
    struct Node{
        Entry* entry;
        Node* next;
    };

    Node* m_head;

    void Push(Entry* e){
        Node* n = new Node;
        n->entry = e;
        do{
            n->next = m_head;
        }while(!CompareAndSwap(&m_head, n->next, n));
    }
```

```cpp
    Entry* Pop(){
        Node* old_head;
        Entry* result;
        do{
            old_head = m_head;
            if(old_head == NULL){
                return NULL;
            }
        }while(!CompareAndSwap(&m_head, old_head, old_head->next));

        result = old_head->entry;
        delete old_head;
        return result;
    }
}
```

Not lock-free

Undefined behavior

Segfault

ABA

Based on work by Alex Skidanov https://habrahabr.ru/post/174369/ □ There you'll find working code

More on lock-free structures and concurrency http://www.1024cores.net/

Purity\Functionality

A program created using only ***pure functions***

No ***side effects*** allowed like:

- Reassigning a variable
- Modifying a data structure in place
- Setting a field on an object
- Throwing an exception or halting with an error
- Printing to the console
- Reading user input
- Reading from or writing to a file
- Drawing on the screen

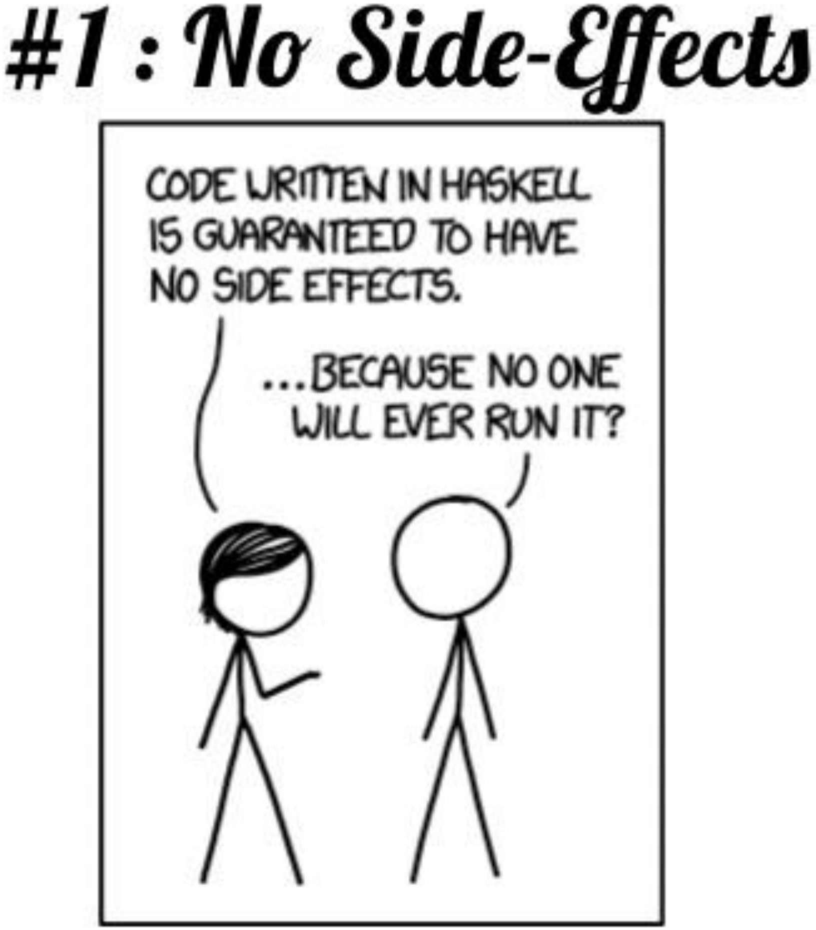
avoidable

deferrable

Functional programming is a restriction on ***how*** we write programs, but not on ***what*** they can do

Source: http://www.slideshare.net/mariofusco/why-we-cannot-ignore-functional-programming

Purity\Functionality



- Knock knock.
- Race condition.
- Who's there?