.NET GC Internals

Mark phase

@konradkokosa / @dotnetosorg

.NET GC Internals

(Non-Concurrent) Mark phase

.NET GC Internals Agenda

- Introduction - roadmap and fundamentals, source code, ...
- **Mark** phase - roots, object graph traversal, *mark stack*, mark/pinned flag, *mark list*, ...
- **Concurrent Mark** phase - *mark array/mark word*, concurrent visiting, *floating garbage*, *write watch list*, ...
- **Plan** phase - *gap*, *plug*, *plug tree*, *brick table*, *pinned plug*, *pre/post plug*, ...
- **Sweep** phase - *free list threading*, concurrent sweep, ...
- **Compact** phase - *relocate* references, compact, ...
- **Generations** - physical organization, *card tables*, ...
- **Allocations** - *bump pointer allocator*, free list allocator, *allocation context*, ...

.NET GC Internals Agenda

- Introduction - roadmap and fundamentals, source code, ...
- **Mark** phase - roots, object graph traversal, *mark stack*, mark/pinned flag, *mark list*, ...
- **Concurrent Mark** phase - *mark array/mark word*, concurrent visiting, *floating garbage*, *write watch list*, ...
- **Plan** phase - *gap*, *plug*, *plug tree*, *brick table*, *pinned plug*, *pre/post plug*, ...
- **Sweep** phase - *free list threading*, concurrent sweep, ...
- **Compact** phase - *relocate* references, compact, ...
- **Generations** - physical organization, *card tables*, ...
- **Allocations** - *bump pointer allocator*, free list allocator, *allocation context*, ...
- **Roots internals** - stack roots, *GCInfo*, *partially/full interruptible methods*, statics, Thread-local Statics (TLS), ...

.NET GC Internals Agenda

- Introduction - roadmap and fundamentals, source code, ...
- **Mark** phase - roots, object graph traversal, *mark stack*, mark/pinned flag, *mark list*, ...
- **Concurrent Mark** phase - *mark array/mark word*, concurrent visiting, *floating garbage*, *write watch list*, ...
- **Plan** phase - *gap*, *plug*, *plug tree*, *brick table*, *pinned plug*, *pre/post plug*, ...
- **Sweep** phase - *free list threading*, concurrent sweep, ...
- **Compact** phase - *relocate* references, compact, ...
- **Generations** - physical organization, *card tables*, ...
- **Allocations** - *bump pointer allocator*, free list allocator, *allocation context*, ...
- **Roots internals** - stack roots, *GCInfo*, *partially/full interruptible methods*, statics, Thread-local Statics (TLS), ...
- **Q&A** - "but why can't I manually delete an object?", ...

02. .NET GC Internals - Mark phase

This module agenda:

- introduction
  - object graph
  - object graph traversal
- implementation
  - traversal
  - pin/mark flag
  - mark stack & mark list
  - vectorized mark list sorting "story"
- inside code .NET runtime 😍

Mark phase

We need to know which objects are "live"...

Object graph

In memory:

Object graph

In memory:

Type data:

```
record A(B b, D d);
record B(int X);
record C(B b, F f);
record D(E e);
record E(G g);
record F(int X);
record G(int Z);
```

Object graph

In memory:

Type data:

```
record A(B b, D d);
record B(int X);
record C(B b, F f);
record D(E e);
record E(G g);
record F(int X);
record G(int Z);
```

Current "state":

```
var a = new A(..., ...);
var d = new D(...);
...we are here...
```

Object graph

In memory:

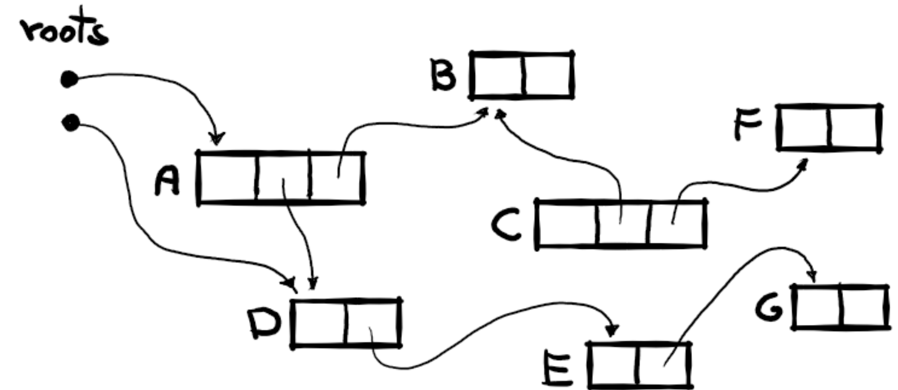
Type data:

```
record A(B b, D d);
record B(int X);
record C(B b, F f);
record D(E e);
record E(G g);
record F(int X);
record G(int Z);
```

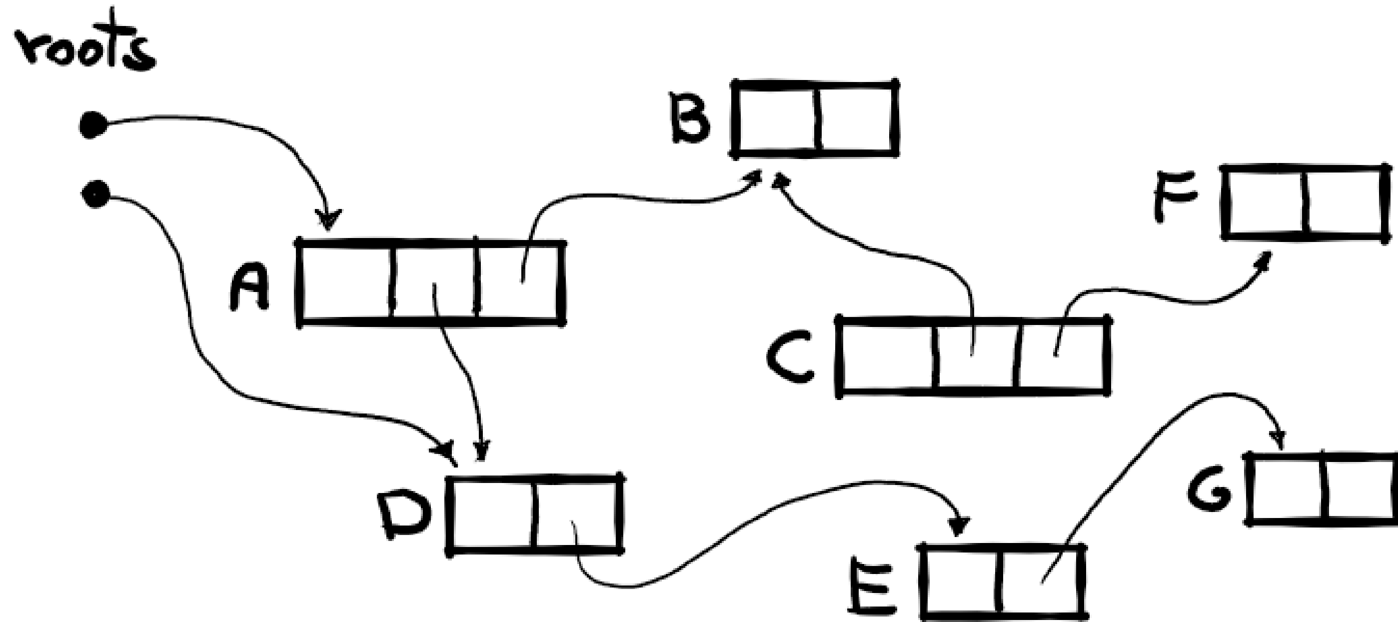
Current "state":

```
var a = new A(..., ...);
var d = new D(...);
...we are here...
```

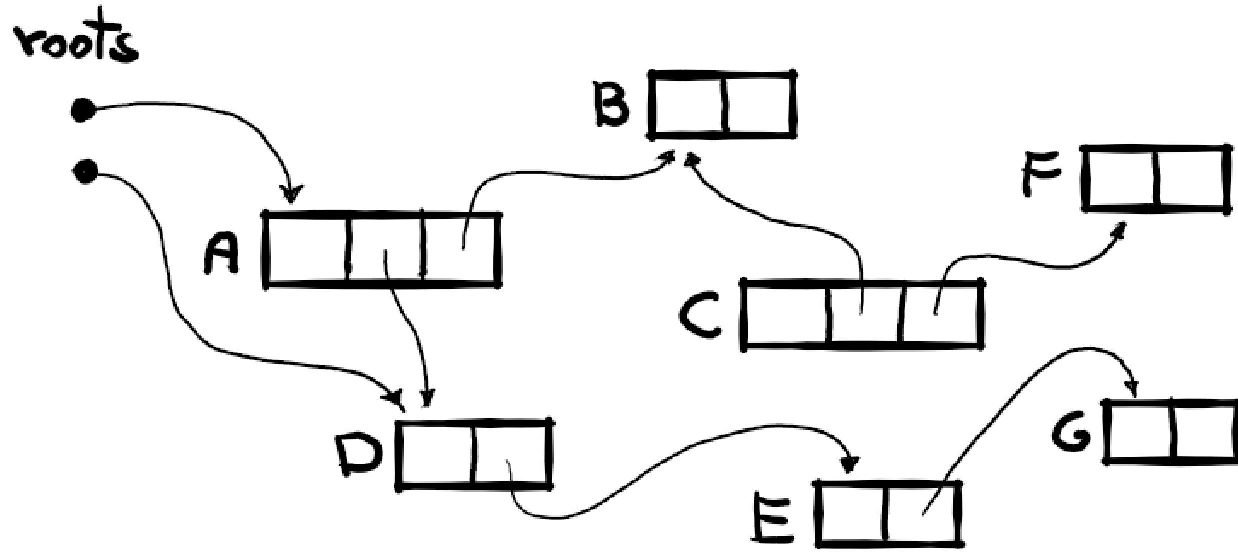
Object graph:



Object graph traversal

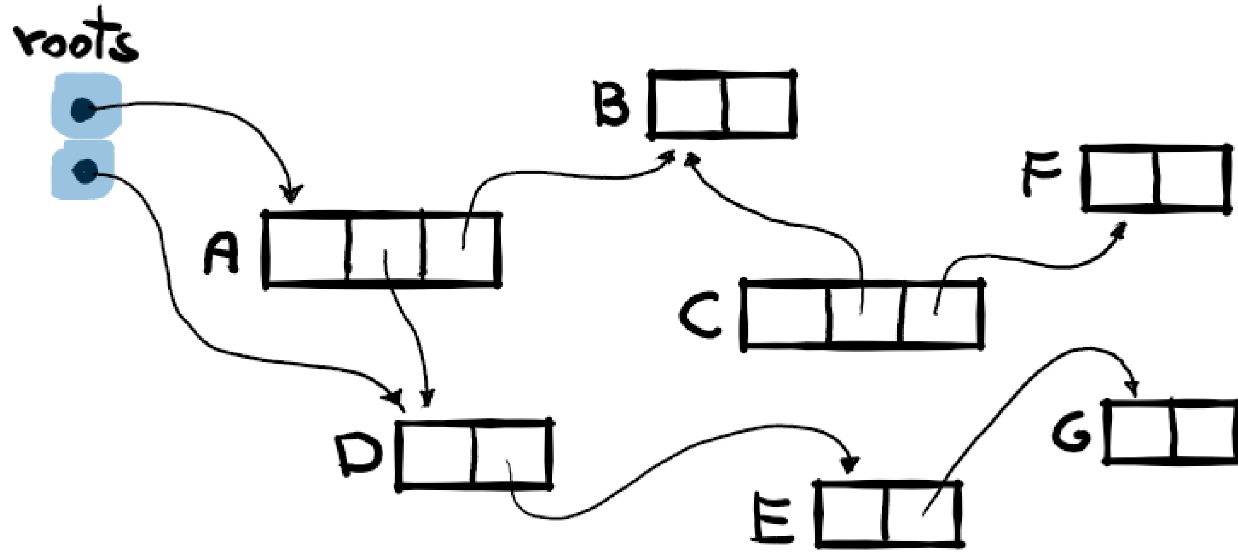


Object graph traversal

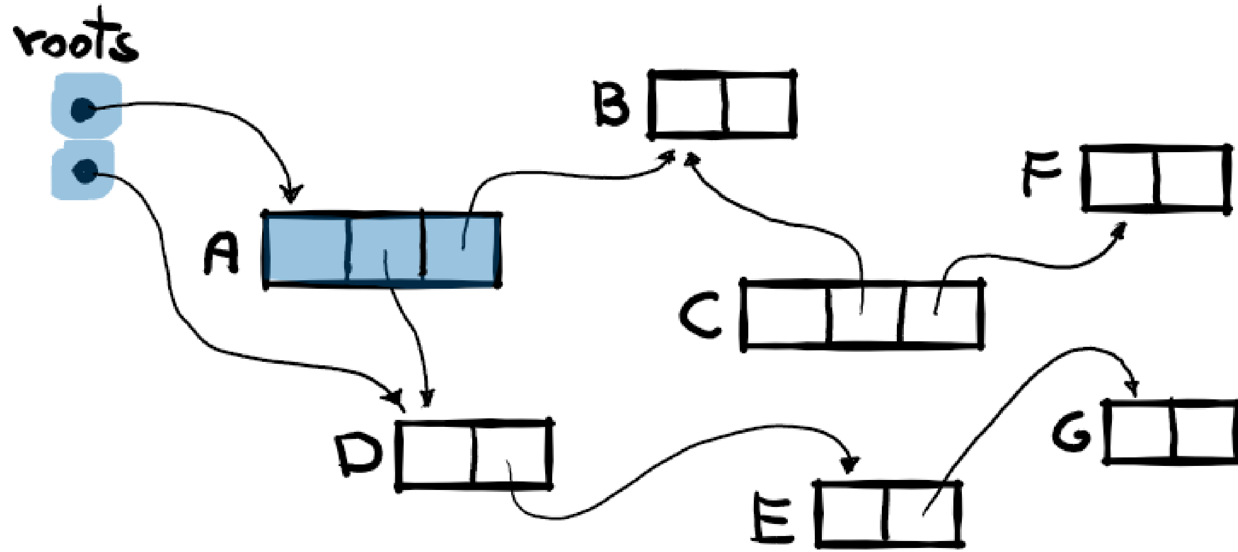


To visit:

Object graph traversal

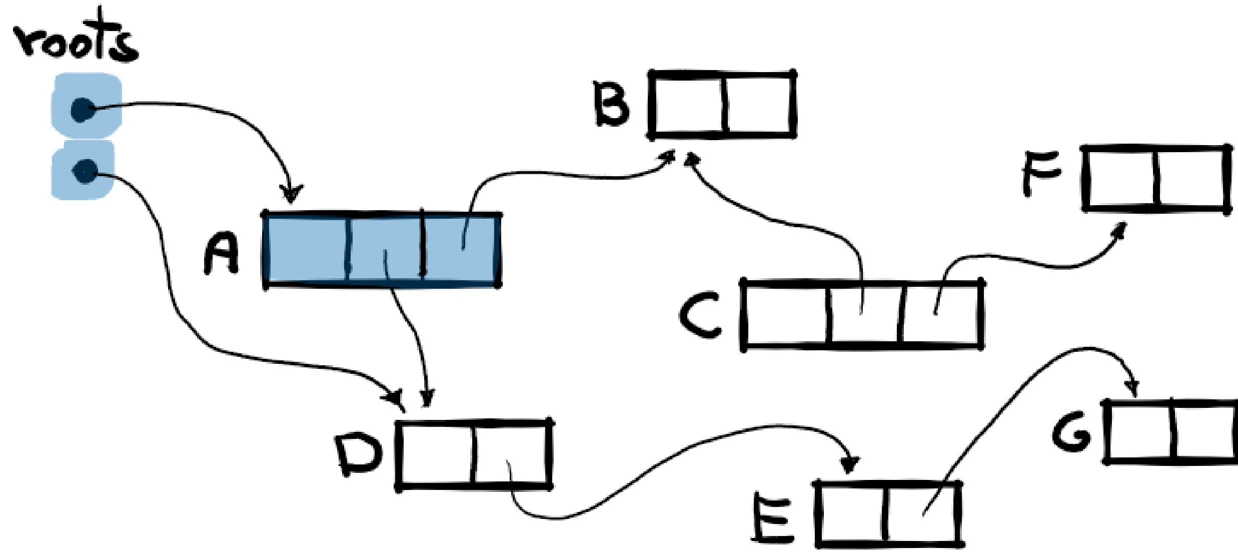


Object graph traversal



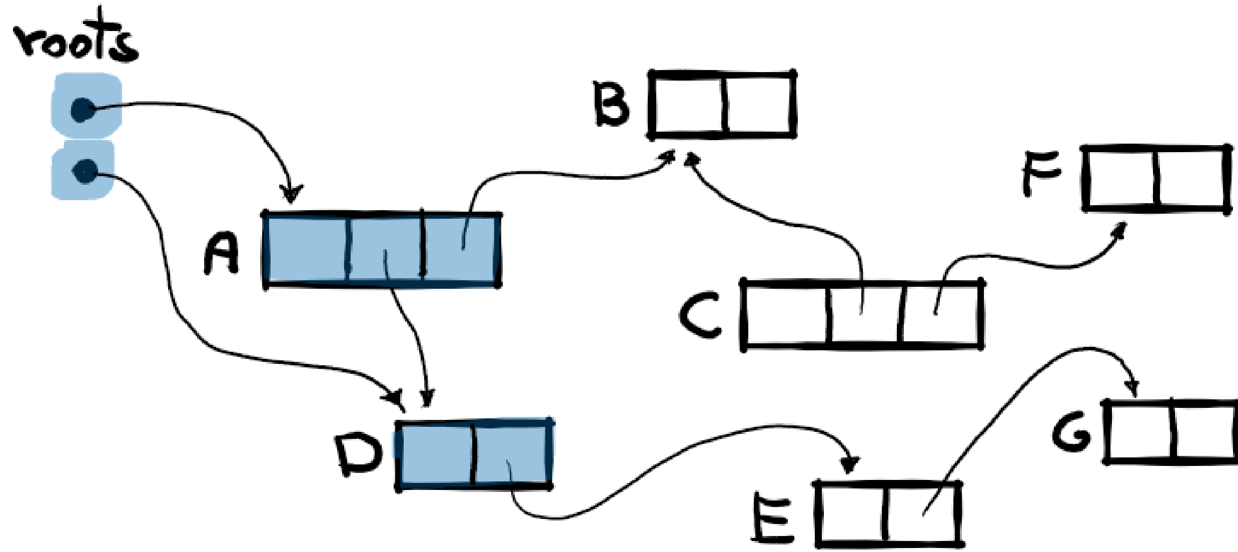
To visit: A, D, D, B

Object graph traversal

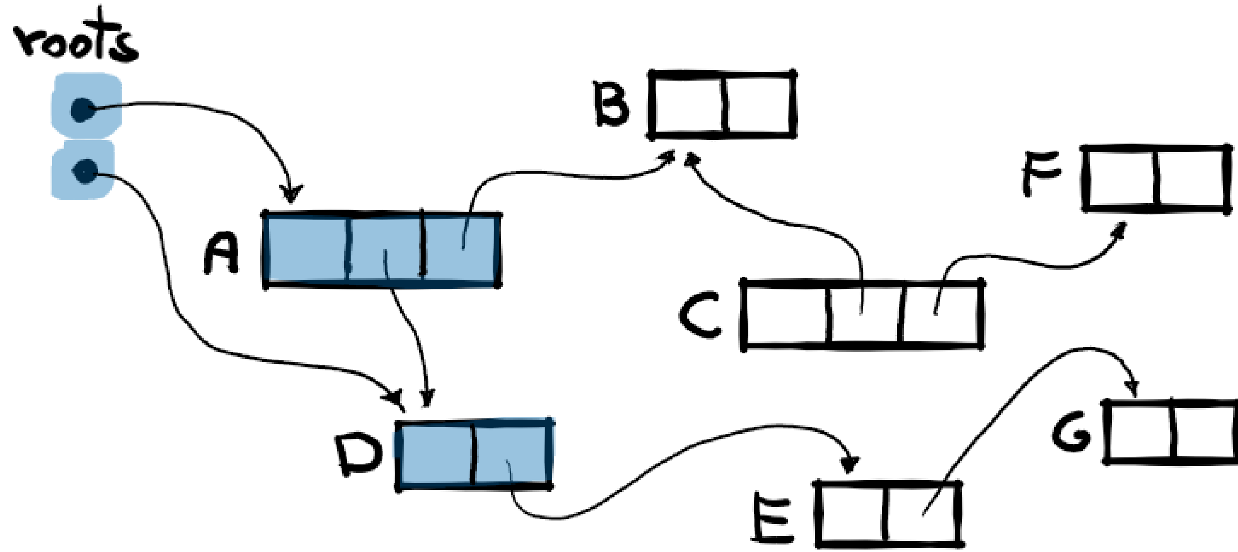


To visit: ~~A~~, D, D, B

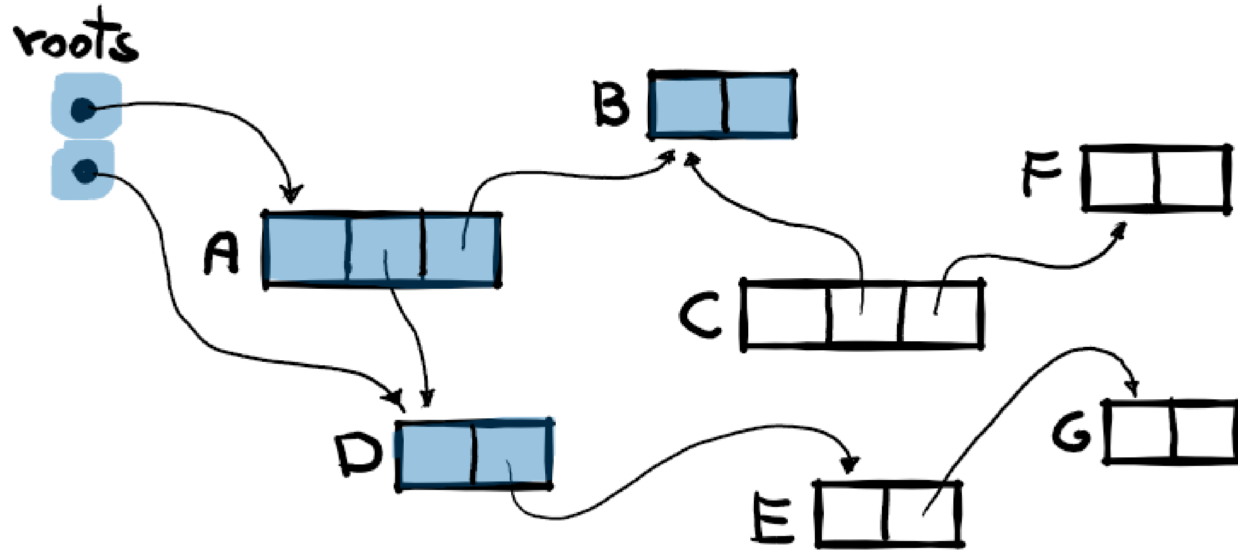
Object graph traversal



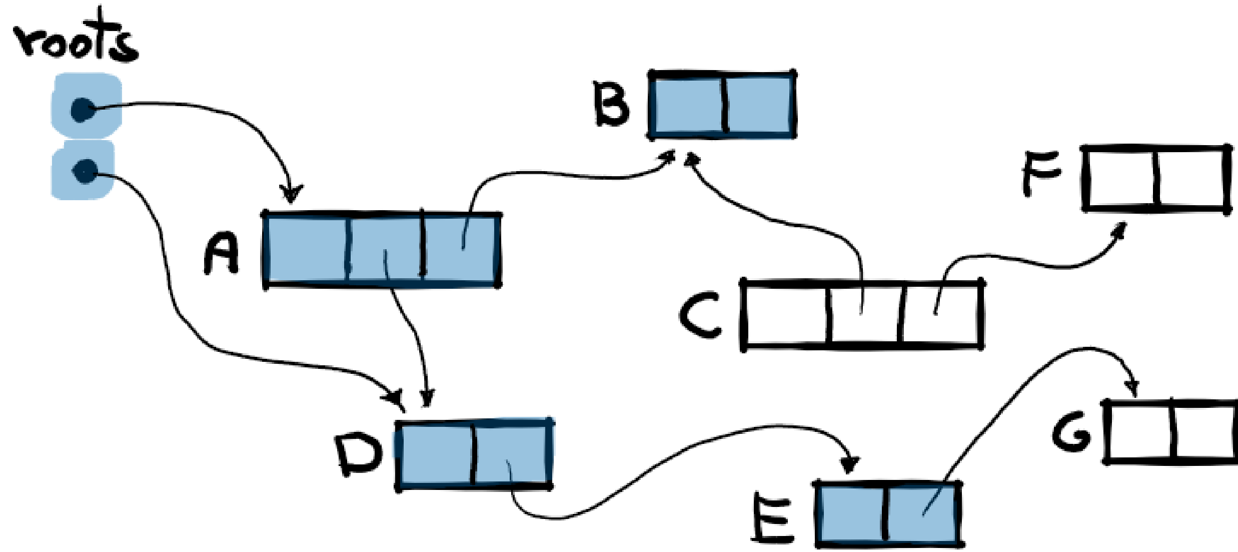
Object graph traversal



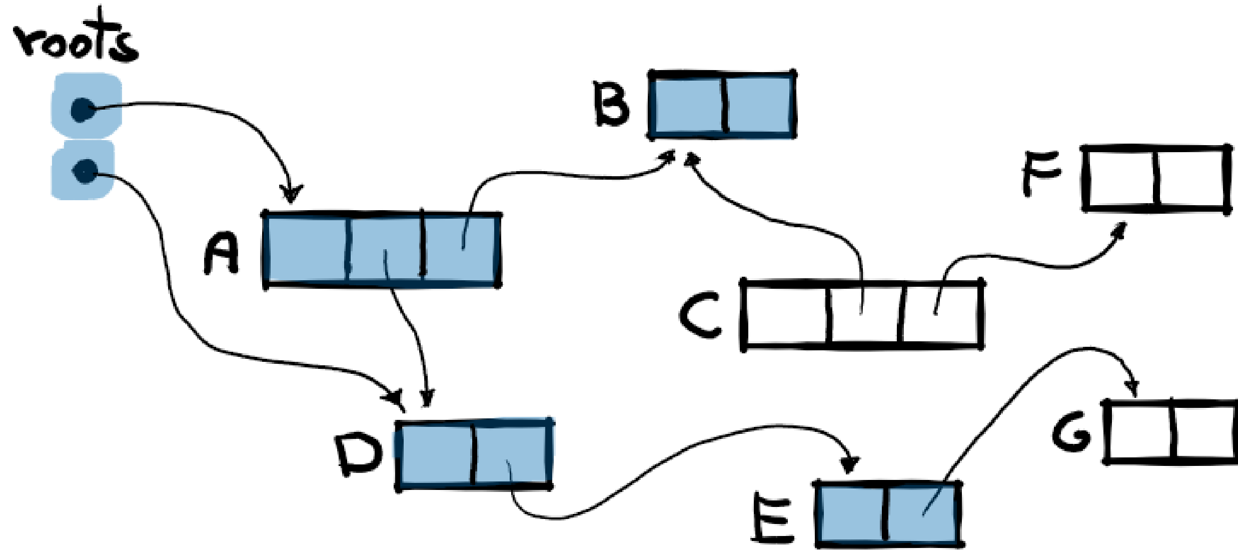
Object graph traversal



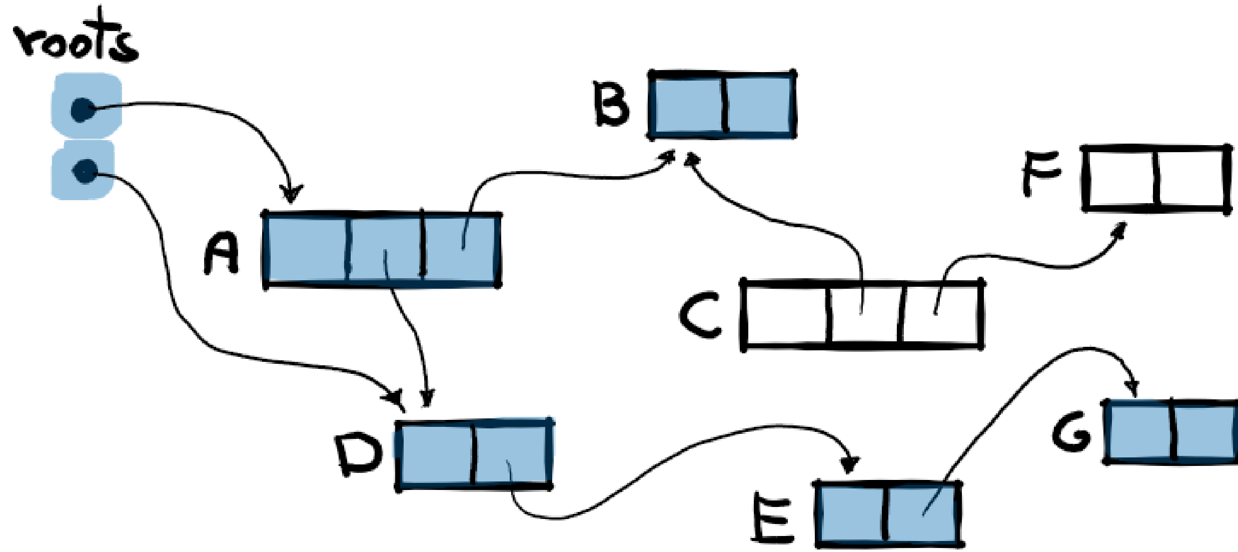
Object graph traversal



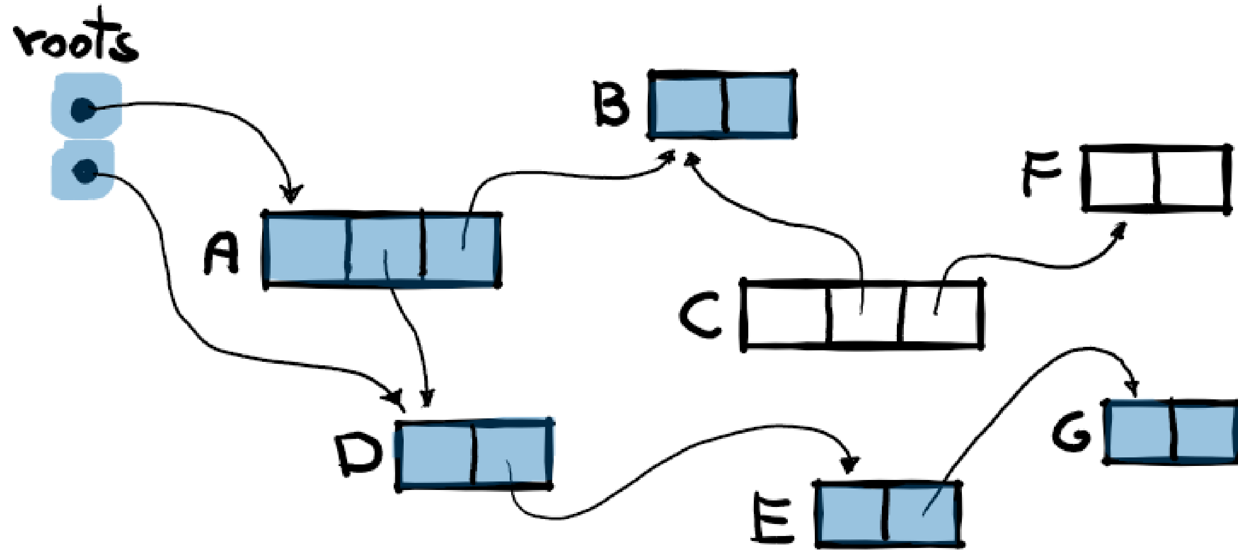
Object graph traversal



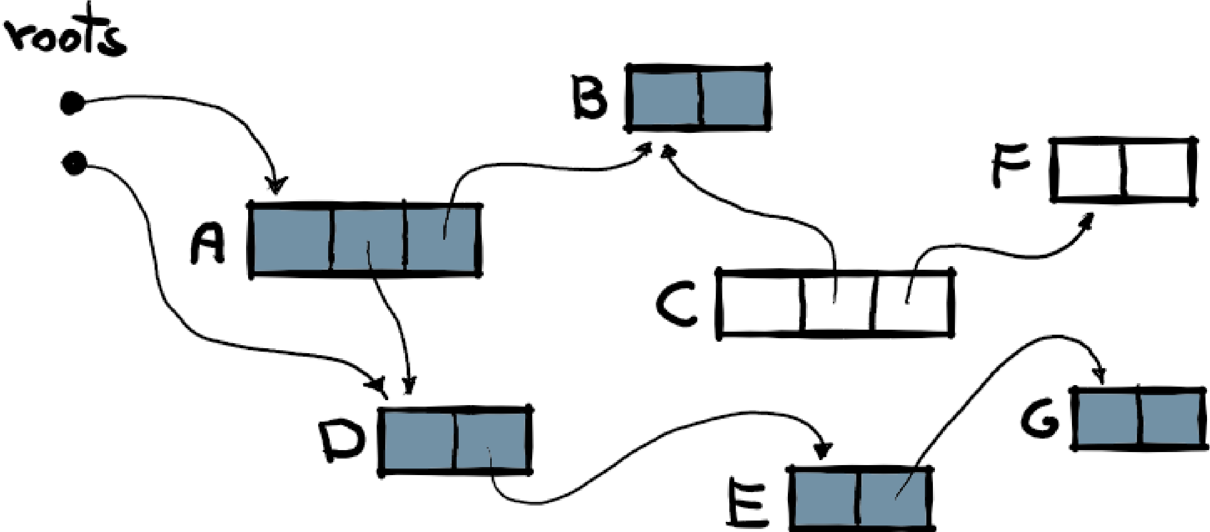
Object graph traversal



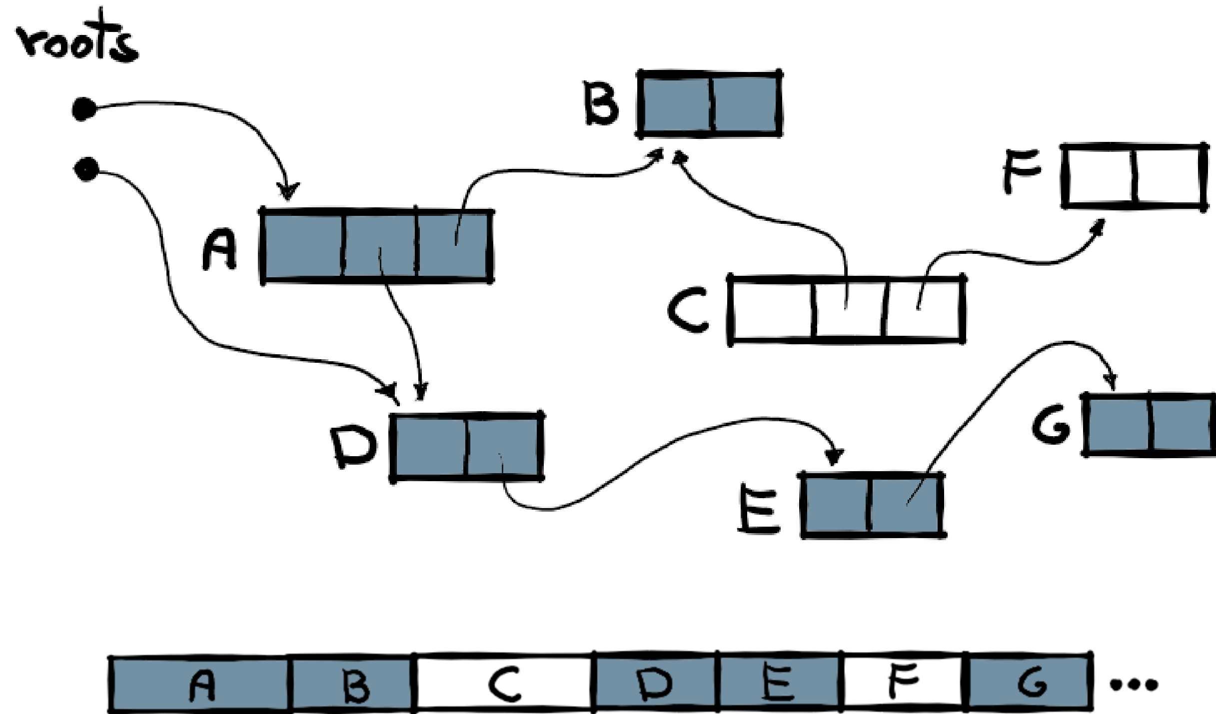
Object graph traversal



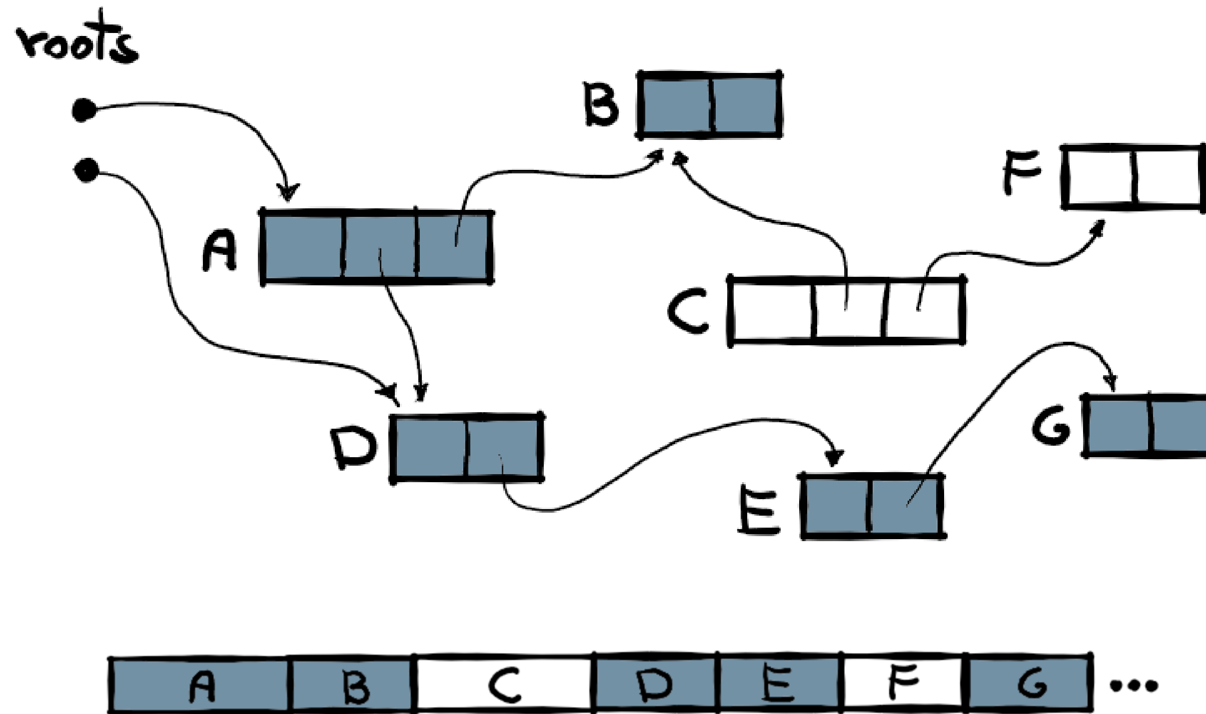
Object graph traversal



Object graph traversal

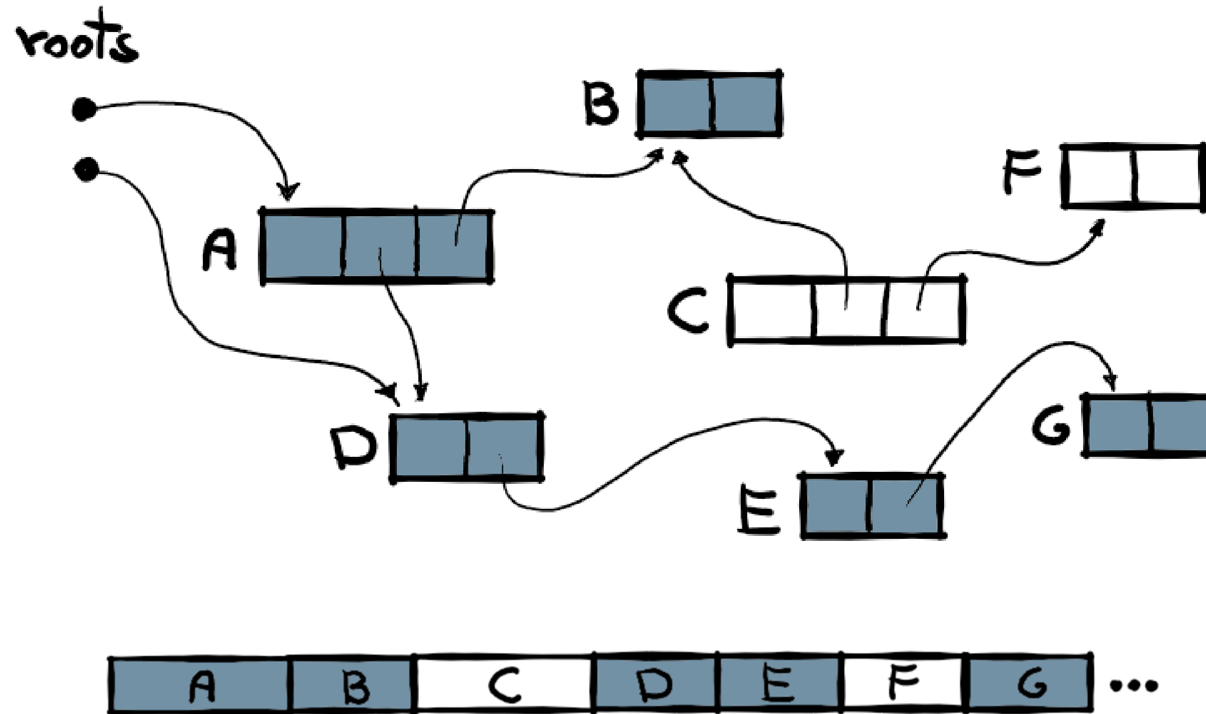


Object graph traversal



We have just discovered **reachability** of the objects (from at least one root) by *marking algorithm*.

Object graph traversal



We have just discovered **reachability** of the objects (from at least one root) by *marking algorithm*.

Reachability is the closest we can get to true "usability" - we don't know the future.

Object graph traversal - roots

- stack
- registers
- static/thread-local static data
- finalization queue
- inter-generational references ("cards", "card tables") - *we will return to that...*
- ...

Mark phase implementation

Sequentially for every root type (like stack, finalization, ...):

1. Collect the roots into the "to visit list" (**the mark stack**)
2. For each given target address `addr` from the mark stack:
   - set **pinning flag** (in the `Header`) - if the runtime says so
   - start traversal:
     - skip already visited object
     - **mark** an object (in the `MT`)
     - add outgoing references to the **mark stack**

Mark phase implementation

Sequentially for every root type (like stack, finalization, …):

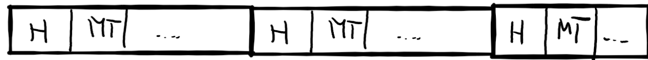
1. Collect the roots into the "to visit list" (**the mark stack**)
2. For each given target address `addr` from the mark stack:
    - translate it to the proper address of a managed object - *we will return to that…*
    - set **pinning flag** (in the `Header`) - if the runtime says so
    - start traversal:
        - skip already visited object
        - **mark** an object (in the `MT`)
        - add outgoing references to the **mark stack**

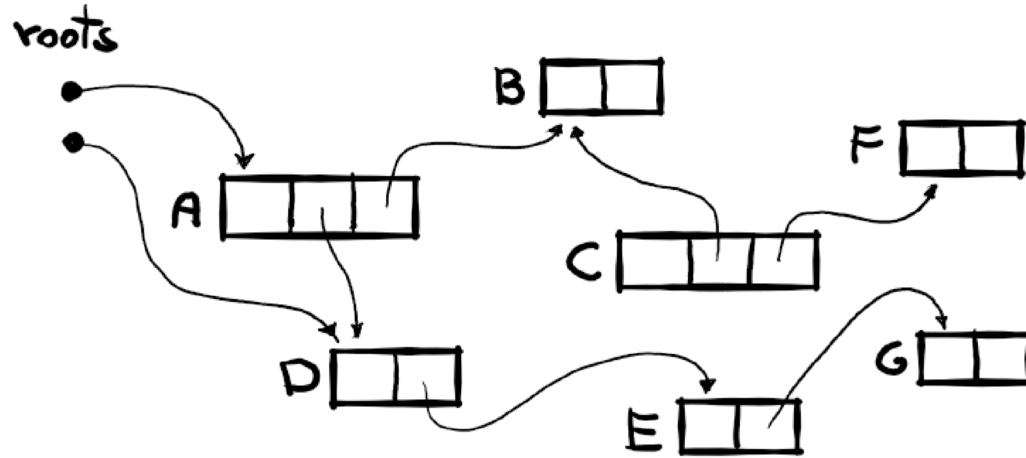
Mark phase implementation

Sequentially for every root type (like stack, finalization, ...):

1. Collect the roots into the "to visit list" (**the mark stack**)
2. For each given target address `addr` from the mark stack:
    - translate it to the proper address of a managed object - *we will return to that...*
    - set **pinning flag** (in the `Header`) - if the runtime says so
    - start traversal:
        - skip already visited object
        - **mark** an object (in the `MT`)
        - add its address to **the mark list** (if not overflowed)
        - add outgoing references to the **mark stack**

Mark phase - let's draw!





Mark stack:

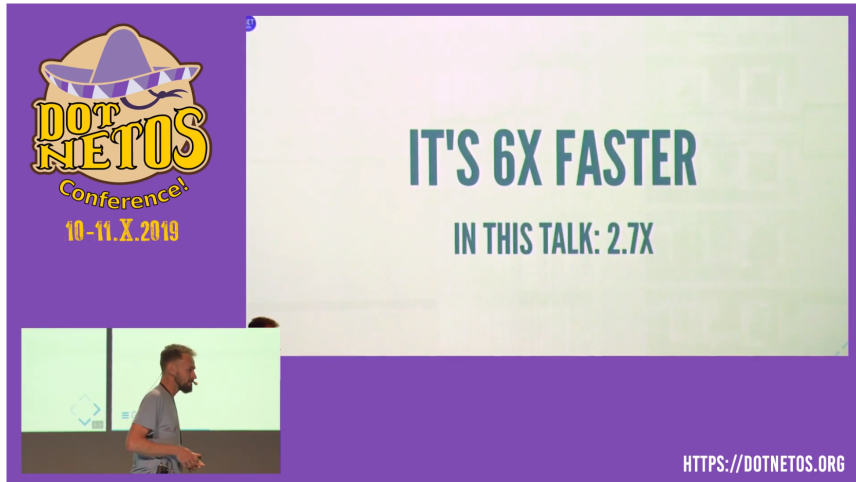
Mark list:

Mark phase

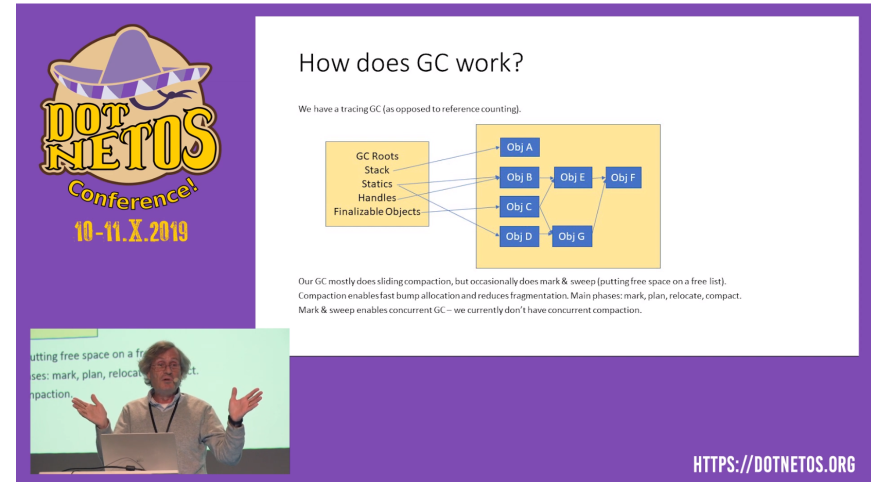
Findings:

- *mark* and *pinned* flags are **added only during the GC** - and **cleared afterwards** at *Plan* phase
  - ie. by looking at a regular memory dump in-between GCs, we won't see those bits set
  - diagnostics tool needs to traverse the graph from roots to notice an object is *pinned*
- mark stack is used as safier approach that recursion
- mark list will help us later, **if sorted**
- it is pretty a lot of work to do (and non-sequential memory access...)!

Mark phase - "mark list sorting story"

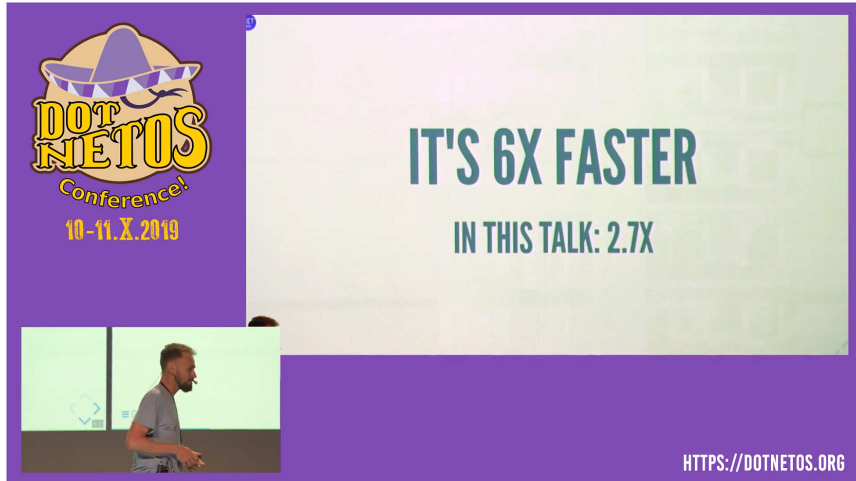


Dan Schechter - .NET Intrinsics in CoreCLR 3.0

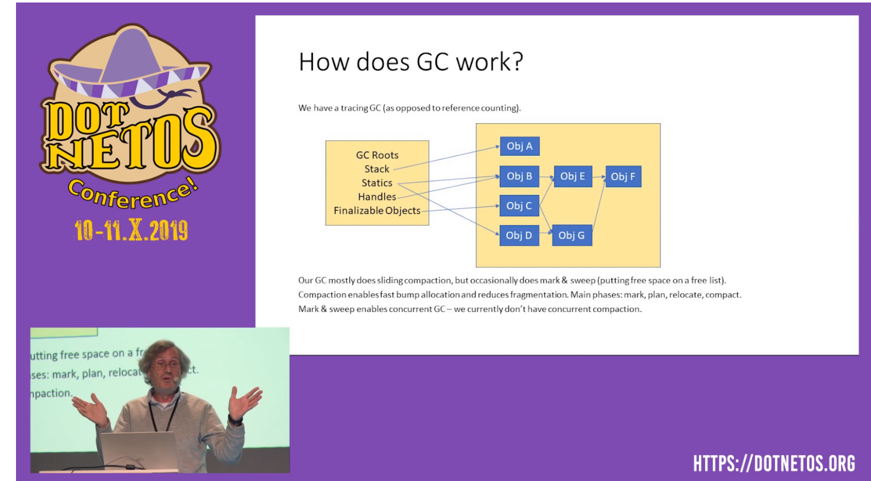


Peter Sollich - The .NET Garbage Collector

Mark phase - "mark list sorting story"



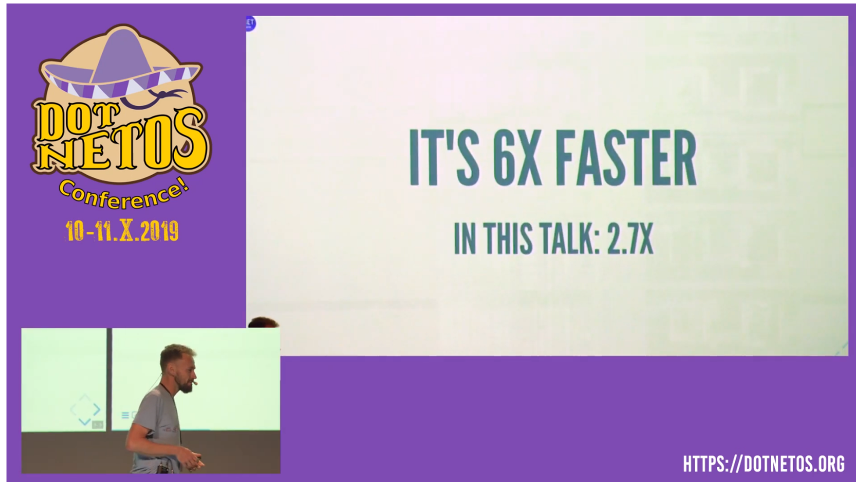
[Dan Schechter - .NET Intrinsics in CoreCLR 3.0]



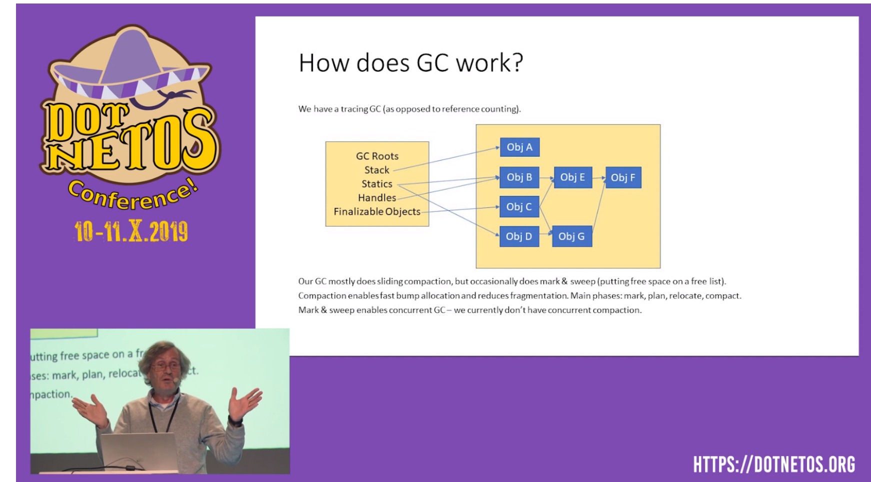
[Peter Sollich - The .NET Garbage Collector]

- Oct 2019 - *"Dan, we could improve our mark list sorting with that..."* - Peter

Mark phase - "mark list sorting story"



[Dan Schechter - .NET Intrinsics in CoreCLR 3.0]



[Peter Sollich - The .NET Garbage Collector]

- Oct 2019 - *"Dan, we could improve our mark list sorting with that..."* - Peter
- Jul 2020 - [https://github.com/dotnet/runtime/pull/37159] - "Vxsort"
  - **faster sorting** code from Dan Shechter, and **bigger mark list**, used for Marking, using AVX2/AVX512F
  - 👉 shorter GC pauses 😍

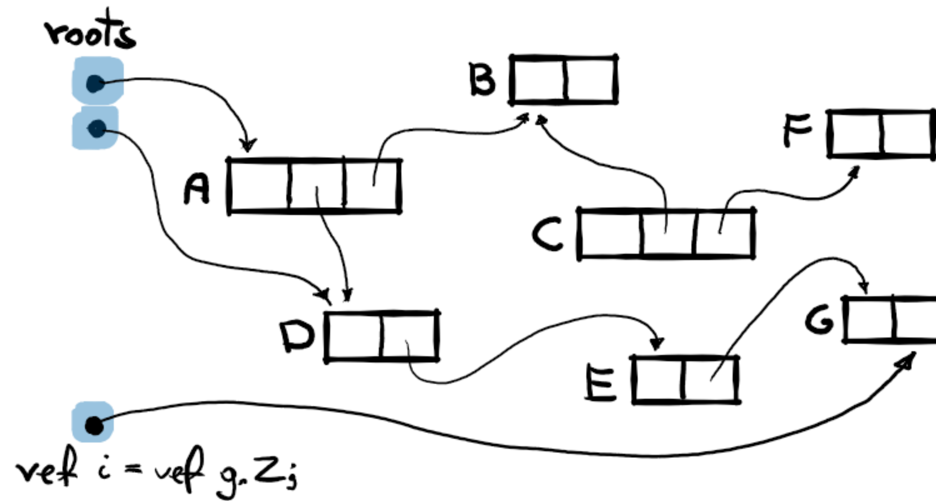
Mark phase - sidenote #1

Mark phase - sidenote #1

"translate it to the proper address of a managed object" - aka interior pointers

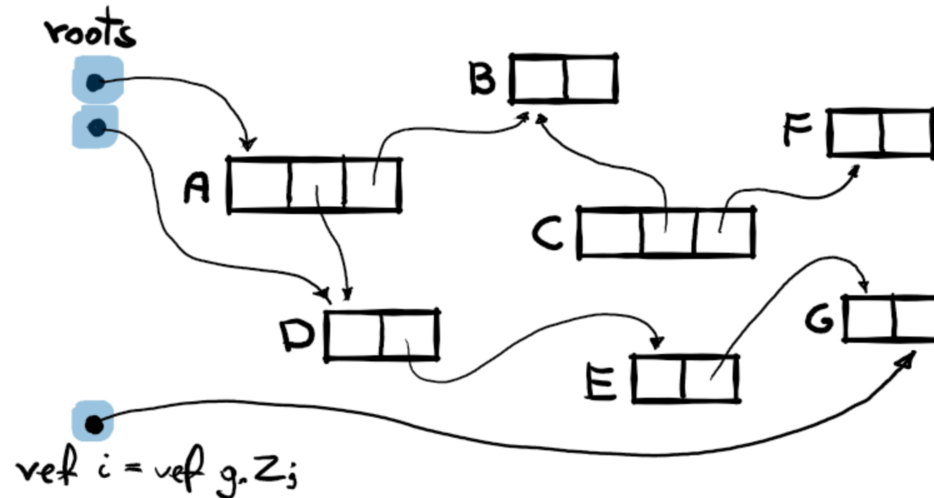
Mark phase - sidenote #1

"translate it to the proper address of a managed object" - aka interior pointers



Mark phase - sidenote #1

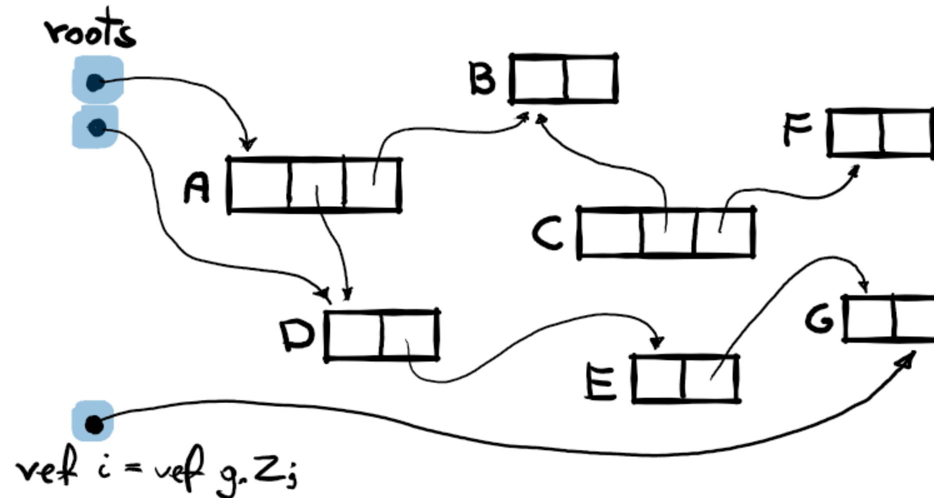
"translate it to the proper address of a managed object" - aka interior pointers




```
void GCHeap::Promote(Object** ppObject, ..., uint32_t flags)
{
    // ...
    if (flags & GC_CALL_INTERIOR)
    {
        if ((o = hp->find_object (o)) == 0)
        {
            return;
        }
```

Mark phase - sidenote #1

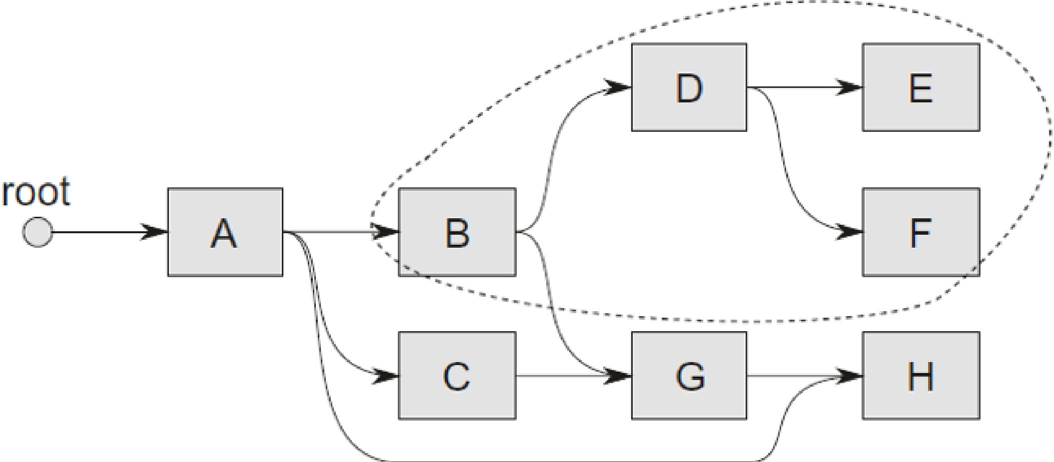
"translate it to the proper address of a managed object" - aka interior pointers



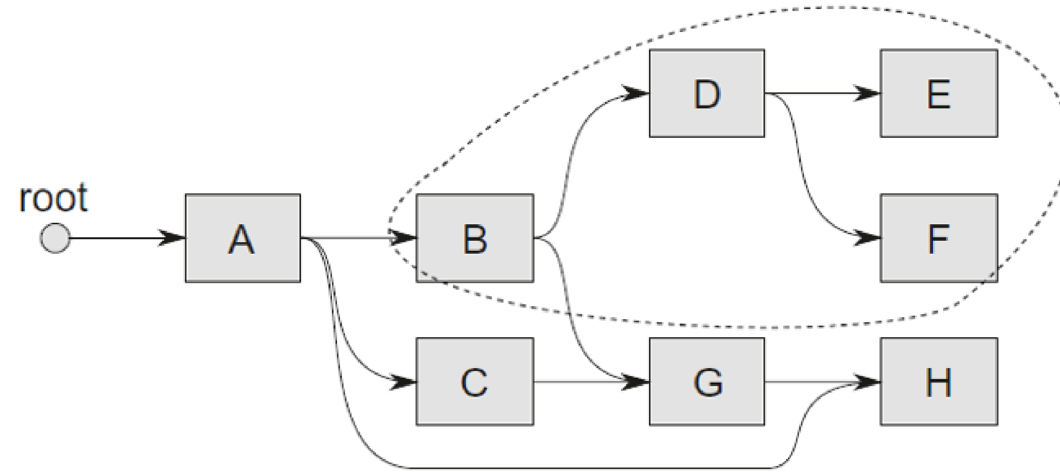

```
void GCHeap::Promote(Object** ppObject, ..., uint32_t flags)
{
    // ...
    if (flags & GC_CALL_INTERIOR)
    {
        if ((o = hp->find_object (o)) == 0) // based on bricks - although not yet available at Mark phase
        {
            return;
        }
```

Mark phase - sidenote #2

Mark phase - sidenote #2



Mark phase - sidenote #2



- shortest root path
- dependency subgraph – total size
- retained subgraph – retained size

Mark phase - inside code

Mark phase starts in the `gc_heap::mark_phase` and has calls to:

- `GCScan::GcScanRoots` that calls `Thread::StackWalkFrames` with `GCHeap::Promote`
- `CFinalize::GcScanRoots` using `GCHeap::Promote`
- `GCScan::GcScanHandles` (with `GCHeap::Promote` callback) methods that calls `Ref_TracePinningRoots` (for types `HNDTYPE_PINNED` and `HNDTYPE_ASYNCPINNED`), `Ref_TraceNormalRoots` (fe. for type `HNDTYPE_STRONG`) etc.
- `gc_heap::mark_through_cards_for_segments` (for SOH) and `gc_heap::mark_through_cards_for_large_objects` (for LOH)
- `GCScan::GcDhInitialScan` and `scan_dependent_handles` handle scanning "dependent handles"

`GCHeap::Promote` method calls the `go_through_object_cl` macro that triggers traversal through objects' references. The main work is done in `gc_heap::mark_object_simple1` that realizes depth-first object graph traversal using "mark stack" called `mark_stack_array` (with `mark_stack_bos` and `mark_stack_tos` indexes pointing to the bottom and the top of the stack). Setting "mark bit" happens in `gc_mark(o)`/`gc_mark1(o)` methods.

Mark phase - inside code

Additionally, **`gc_heap::mark_object_simple`**/**`gc_heap::mark_object_simple1`** methods, while traversal, are using **`m_boundary`** macro to populare **`mark_list`**.

Mark list is maintained/sorted only for "non concurrent" ephemeral GC (*"multiple segments are more complex to handle and the list is likely to overflow"*). Sorting happens in:

- **`mark_phase`** (only iff **`PARALLEL_MARK_LIST_SORT`** & **`MULTIPLE_HEAPS`**) calling **`gc_heap::sort_mark_list`** (using **`do_vxsort`** if **`USE_VXSORT`**)
- **`plan_phase`** (only iff **`!MULTIPLE_HEAPS`**) using **`do_vxsort`** (if **`USE_VXSORT`**)

If mark list overflows, we expand it up to maximum in **`gc_heap::grow_mark_list`**:

```
// with vectorized sorting, we can use bigger mark lists
#ifdef USE_VXSORT
#ifdef MULTIPLE_HEAPS
    const size_t MAX_MARK_LIST_SIZE = IsSupportedInstructionSet (InstructionSet::AVX2) ? 1000 * 1024 : 200 * 102
#else //MULTIPLE_HEAPS
    const size_t MAX_MARK_LIST_SIZE = IsSupportedInstructionSet (InstructionSet::AVX2) ? 32 * 1024 : 16 * 1024;
#endif //MULTIPLE_HEAPS
#else
...
```

Thank you! Any questions?!