.NET GC Internals

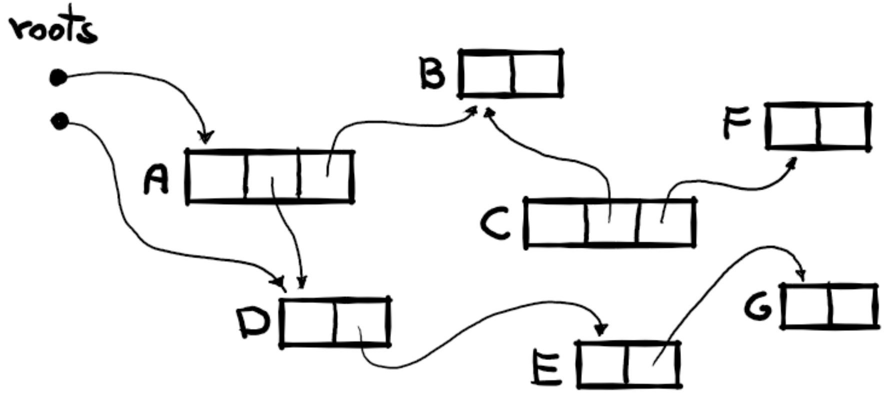
Concurrent Mark phase

@konradkokosa / @dotnetosorg

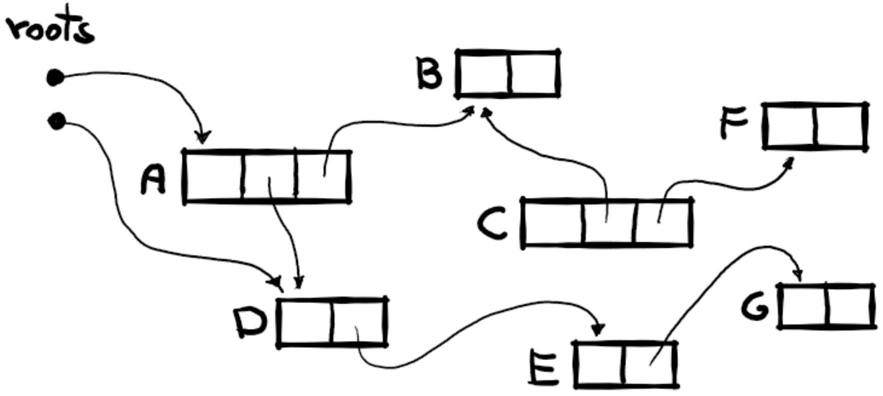
.NET GC Internals Agenda

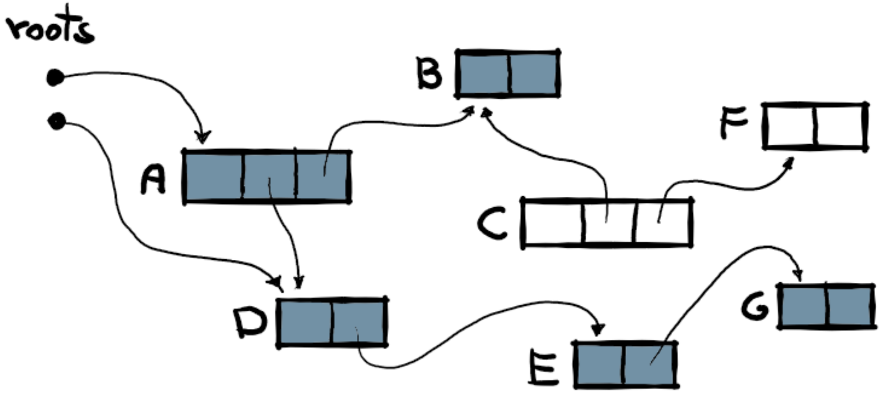
- Introduction - roadmap and fundamentals, source code, ...
- **Mark** phase - roots, object graph traversal, *mark stack*, mark/pinned flag, *mark list*, ...
- **Concurrent Mark** phase - *mark array/mark word*, concurrent visiting, *floating garbage*, *write watch list*, ...
- **Plan** phase - *gap*, *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?", ...

Concurrent Mark phase

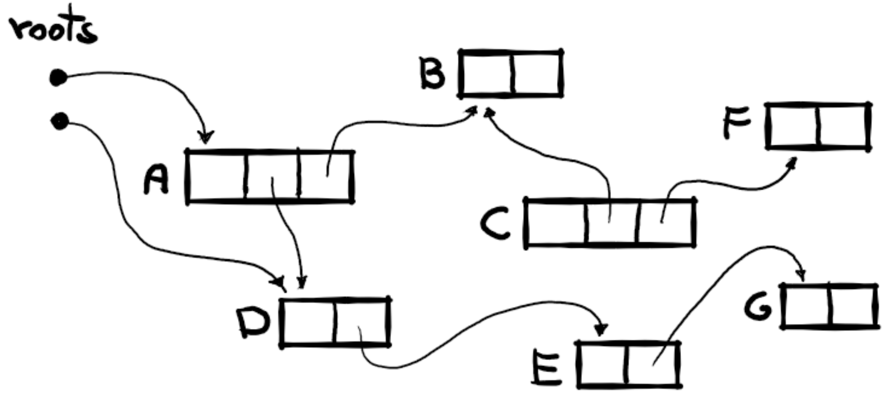


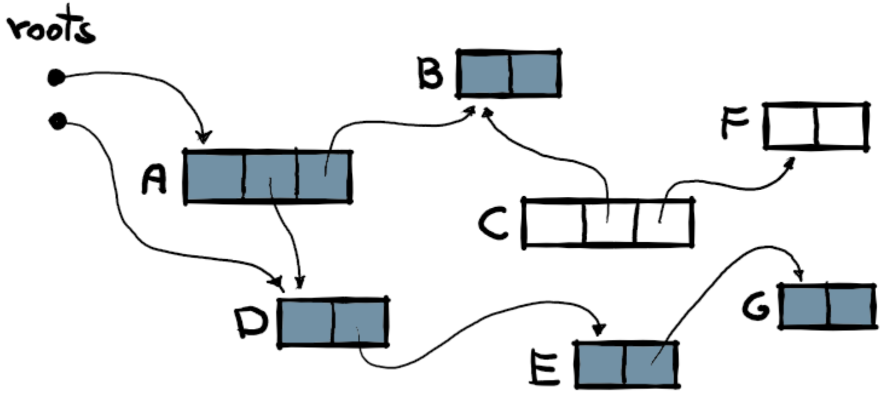
Concurrent Mark phase





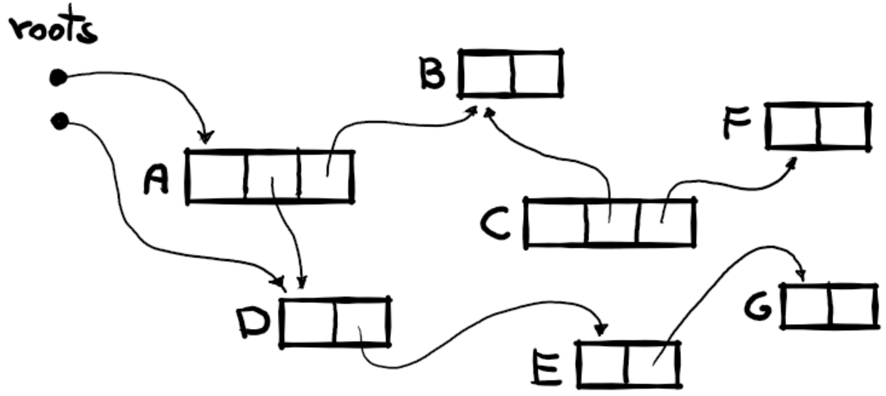
Concurrent Mark phase

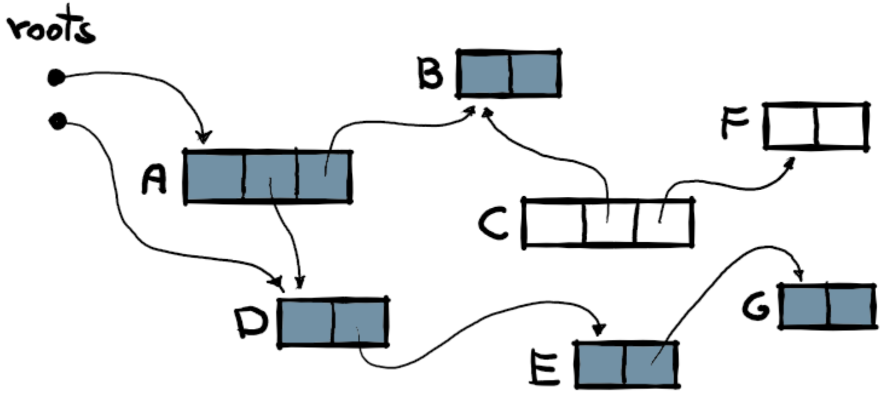




- Concurrent GCs mark objects while the application is running…😱

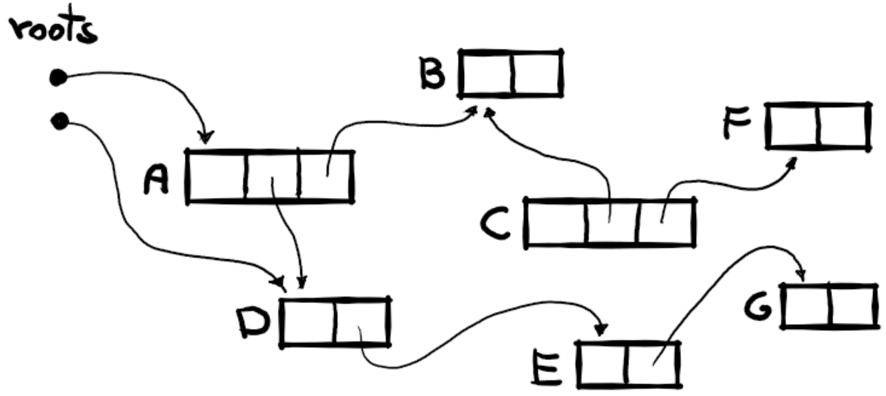
Concurrent Mark phase

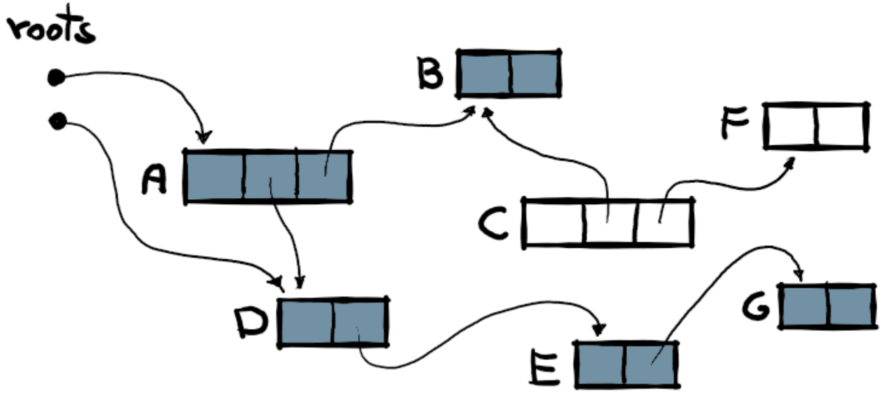




- Concurrent GCs mark objects while the application is running…😱
- two problems:

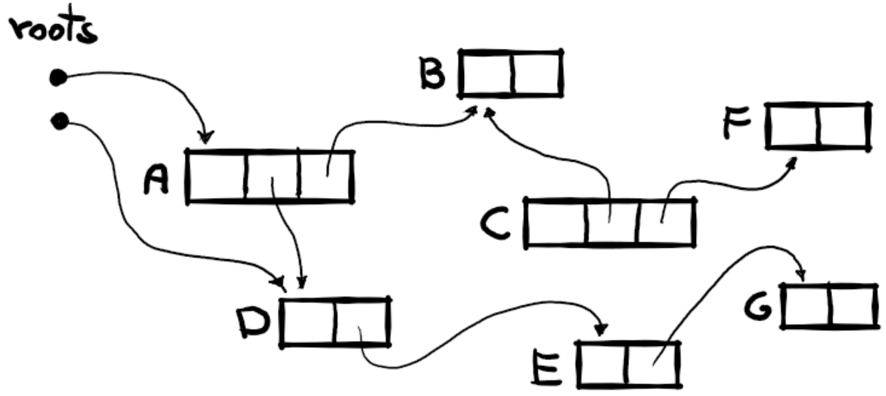
Concurrent Mark phase

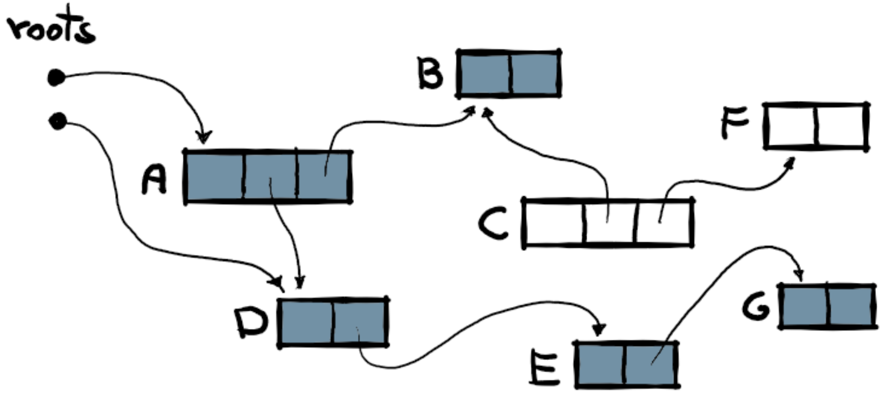




- Concurrent GCs mark objects while the application is running...😱
- two problems:
  - #1 - how to mark an object while being used? - we were using MT for this...

Concurrent Mark phase





- Concurrent GCs mark objects while the application is running...😱
- two problems:
  - #1 - how to mark an object while being used? - we were using MT for this...
  - #2 - how to get a consistent view while references are changing? - ups...

Problem #1 - marking used objects

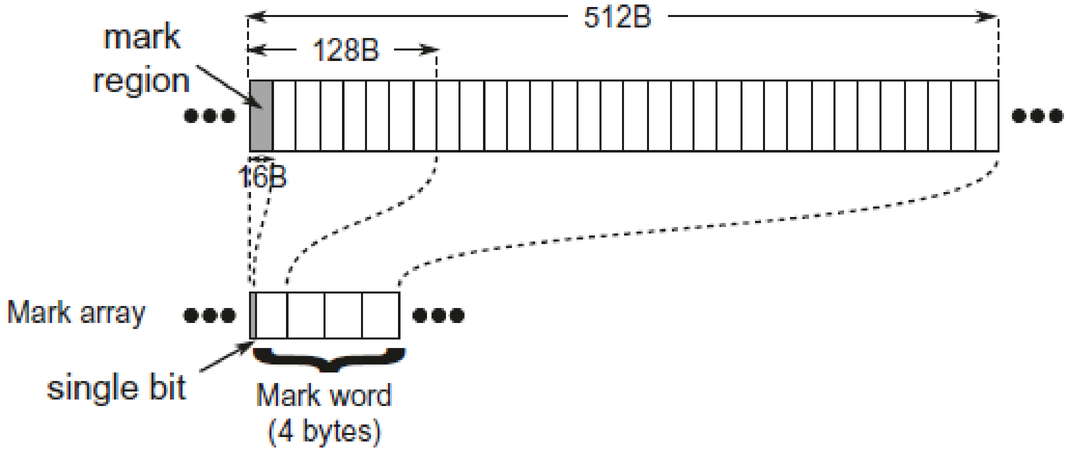
- can't use *MethodTable* - manipulating live pointer and cache invalidation

Problem #1 - marking used objects

- can't use *MethodTable* - manipulating live pointer and cache invalidation
- we need to store mark information elsewhere - ***mark array***

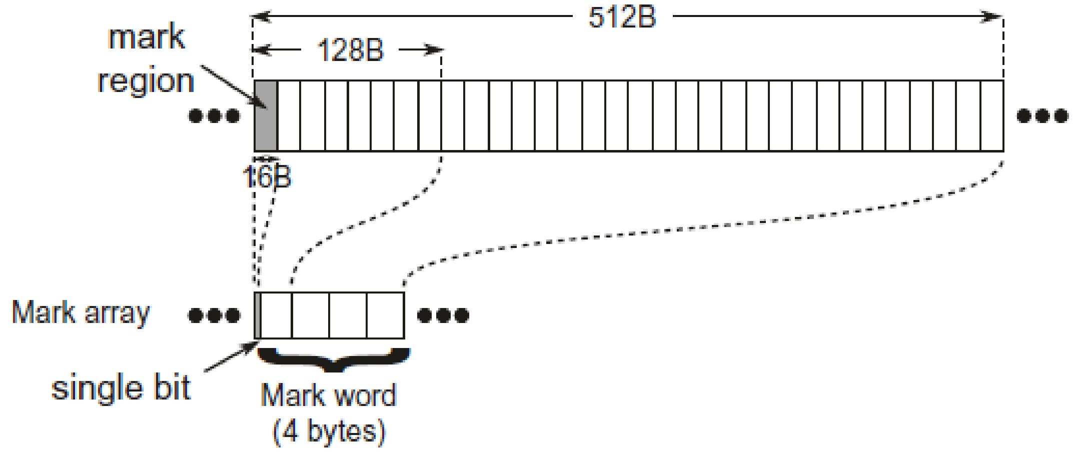
Problem #1 - marking used objects

- can't use *MethodTable* - manipulating live pointer and cache invalidation
- we need to store mark information elsewhere - ***mark array***
- each bit maps to 16 bytes (64-bit) or 8 bytes (32-bit) region
  - so, single byte covers 128B
  - and *mark word* covers 512B



Problem #1 - marking used objects

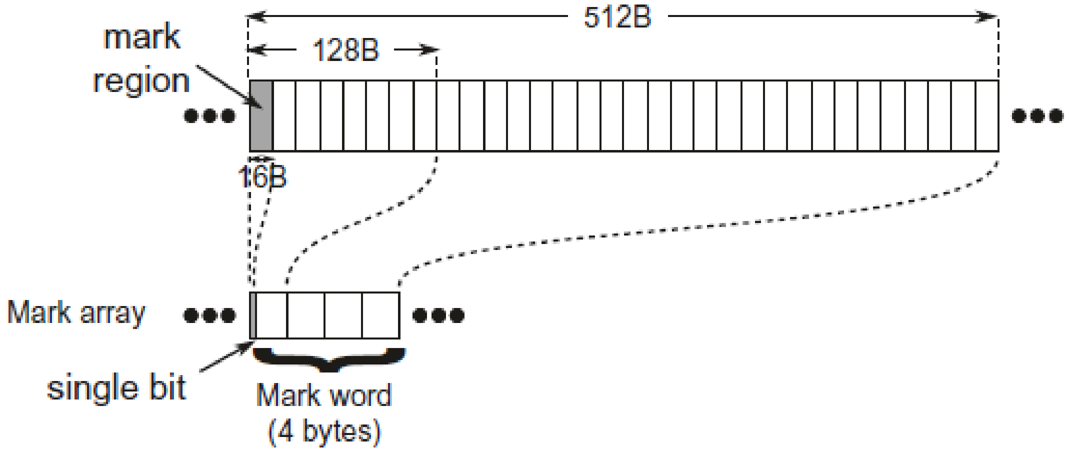
- can't use *MethodTable* - manipulating live pointer and cache invalidation
- we need to store mark information elsewhere - ***mark array***
- each bit maps to 16 bytes (64-bit) or 8 bytes (32-bit) region
  - so, single byte covers 128B
  - and *mark word* covers 512B



- 16B granularity is enough - minimum object size is 24B

Problem #1 - marking used objects

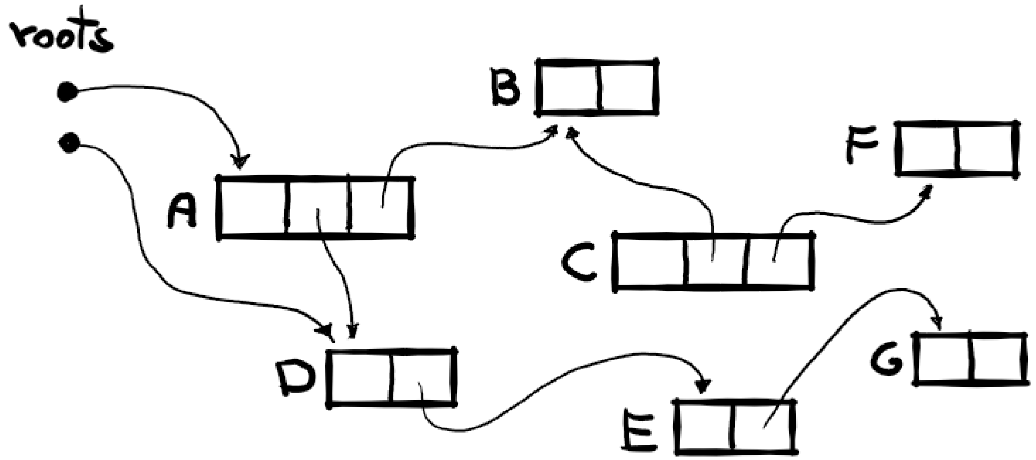
- can't use *MethodTable* - manipulating live pointer and cache invalidation
- we need to store mark information elsewhere - ***mark array***
- each bit maps to 16 bytes (64-bit) or 8 bytes (32-bit) region
  - so, single byte covers 128B
  - and *mark word* covers 512B



- 16B granularity is enough - minimum object size is 24B
- so, in case of 64-bit, we need 8MB of mark array per 1GB of data

Problem #1 - marking used objects

Using mark array during marking graph traversal:



Problem #2 - consistent view

When references between objects are changing while *marking*, we may end up in a few situations:

Problem #2 - consistent view

When references between objects are changing while *marking*, we may end up in a few situations:

- Not-yet-visited object has modified references to some other objects

Problem #2 - consistent view

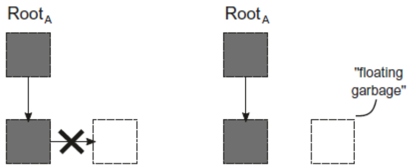
When references between objects are changing while *marking*, we may end up in a few situations:

- Not-yet-visited object has modified references to some other objects - **fine**! We will visit it anyway.

Problem #2 - consistent view

When references between objects are changing while *marking*, we may end up in a few situations:

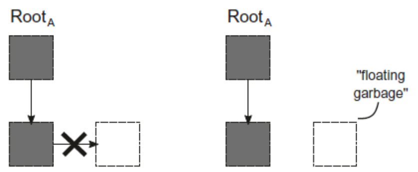
- Not-yet-visited object has modified references to some other objects - **fine**! We will visit it anyway.
- Already visited object has removed reference to the otherwise unreachable object



Problem #2 - consistent view

When references between objects are changing while *marking*, we may end up in a few situations:

- Not-yet-visited object has modified references to some other objects - **fine**! We will visit it anyway.
- Already visited object has removed reference to the otherwise unreachable object

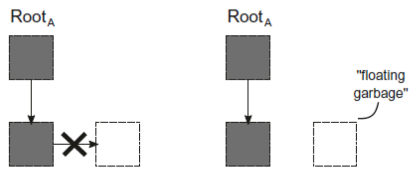


well... **fine**! We've just created some *"floating garbage"* to collect next time.

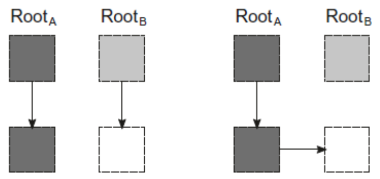
Problem #2 - consistent view

When references between objects are changing while *marking*, we may end up in a few situations:

- Not-yet-visited object has modified references to some other objects - **fine**! We will visit it anyway.
- Already visited object has removed reference to the otherwise unreachable object

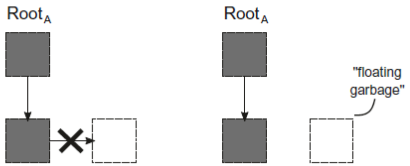
  well... **fine**! We've just created some *"floating garbage"* to collect next time.
- Already visited object has added reference to otherwise unreachable object (or new one)

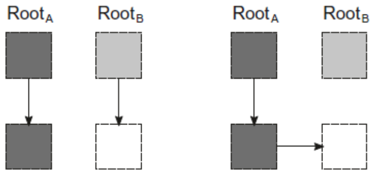
Problem #2 - consistent view

When references between objects are changing while *marking*, we may end up in a few situations:

- Not-yet-visited object has modified references to some other objects - **fine**! We will visit it anyway.
- Already visited object has removed reference to the otherwise unreachable object

  well... **fine**! We've just created some *"floating garbage"* to collect next time.
- Already visited object has added reference to otherwise unreachable object (or new one)

  well... this is **NOT fine**! *"The lost object"* problem - we will not visit it, and it will be GCed! 😲

Problem #2 - consistent view

Problem #2 - consistent view

- Already visited object has added a reference to an otherwise reachable object - whether it is “the lost object” require checking whether we will have chance to visit such an object.

Problem #2 - consistent view

- Already visited object has added a reference to an otherwise reachable object - whether it is “the lost object” require checking whether we will have chance to visit such an object.
- Currently visiting object has modified its references - it would require checking whether such reference has been already visited. If not, it's #1. If yes, it's one of #2-#4.

Problem #2 - consistent view

- Already visited object has added a reference to an otherwise reachable object - whether it is “the lost object” require checking whether we will have chance to visit such an object.
- Currently visiting object has modified its references - it would require checking whether such reference has been already visited. If not, it's #1. If yes, it's one of #2-#4.

Solution: We may 🤯 while trying to solve this.

Problem #2 - consistent view

- Already visited object has added a reference to an otherwise reachable object - whether it is “the lost object” require checking whether we will have chance to visit such an object.
- Currently visiting object has modified its references - it would require checking whether such reference has been already visited. If not, it's #1. If yes, it's one of #2-#4.

Solution: We may 🤯 while trying to solve this. But... what if, just, problematic objects should be revisited?

Problem #2 - consistent view

- Already visited object has added a reference to an otherwise reachable object - whether it is “the lost object” require checking whether we will have chance to visit such an object.
- Currently visiting object has modified its references - it would require checking whether such reference has been already visited. If not, it's #1. If yes, it's one of #2-#4.

Solution: We may 🤯 while trying to solve this. But… what if, just, problematic objects should be revisited? *Problematic* ⊆ *modified*.

Problem #2 - consistent view

- Already visited object has added a reference to an otherwise reachable object - whether it is “the lost object” require checking whether we will have chance to visit such an object.
- Currently visiting object has modified its references - it would require checking whether such reference has been already visited. If not, it's #1. If yes, it's one of #2-#4.

Solution: We may 🤯 while trying to solve this. But… what if, just, problematic objects should be revisited? *Problematic* ⊆ *modified*. So, revisit all *modified*!

Concurrent Mark

Concurrent Mark

- various algorithms balance the amount of "floating garbage", number of "revisits" to make and overall synchronization costs

Concurrent Mark

- various algorithms balance the amount of "floating garbage", number of "revisits" to make and overall synchronization costs
- in .NET, it is based on ***write barriers***:
  - a code injected to "assign reference" operation (for the time of Concurrent Mark) - like `S.X = t`,
  - responsible for notyfing that the *source object* `S` has changed
  - unconditionally adds it to the ***write watch list***

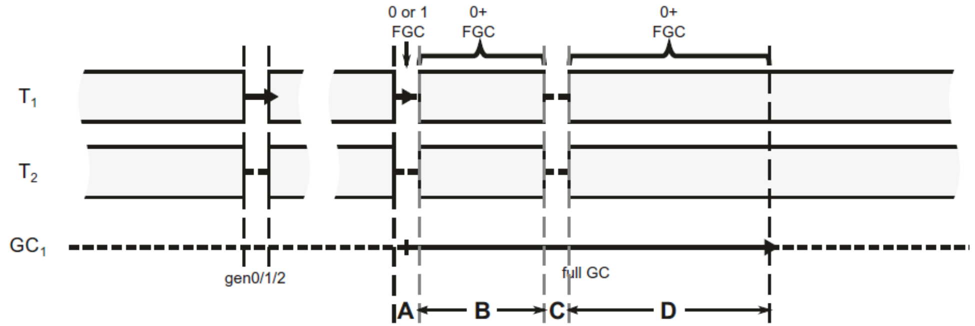
Concurrent Mark

- various algorithms balance the amount of "floating garbage", number of "revisits" to make and overall synchronization costs
- in .NET, it is based on ***write barriers***:
  - a code injected to "assign reference" operation (for the time of Concurrent Mark) - like `S.X = t`,
  - responsible for notyfing that the *source object* `S` has changed
  - unconditionally adds it to the ***write watch list***
- later on write watch list is just a set of additional roots (to revisit and start traversal from there)

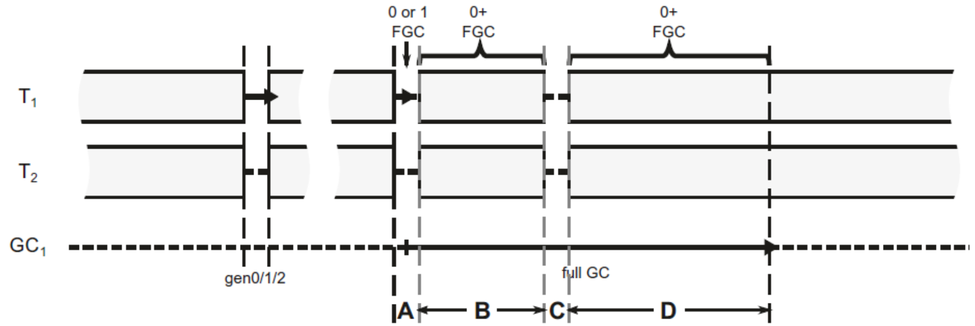
Concurrent Mark

- various algorithms balance the amount of "floating garbage", number of "revisits" to make and overall synchronization costs
- in .NET, it is based on ***write barriers***:
  - a code injected to "assign reference" operation (for the time of Concurrent Mark) - like `S.X = t`,
  - responsible for notyfing that the *source object* `S` has changed
  - unconditionally adds it to the ***write watch list***
- later on write watch list is just a set of additional roots (to revisit and start traversal from there)
- so, yes - the more references modifications during Concurrent Mark, the bigger write watch list and then revisiting cost

Concurrent Mark

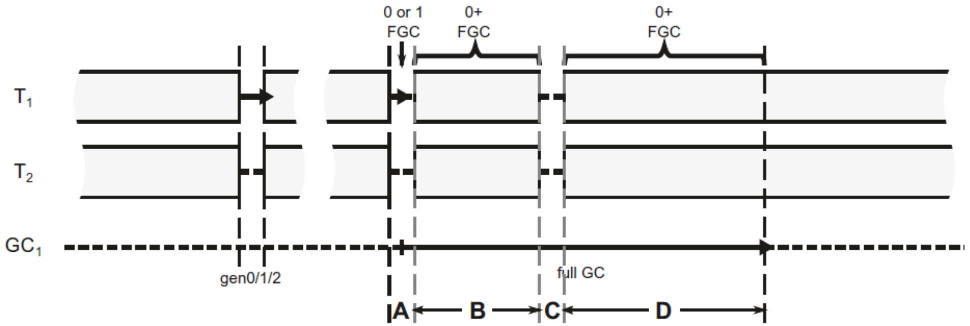


Concurrent Mark



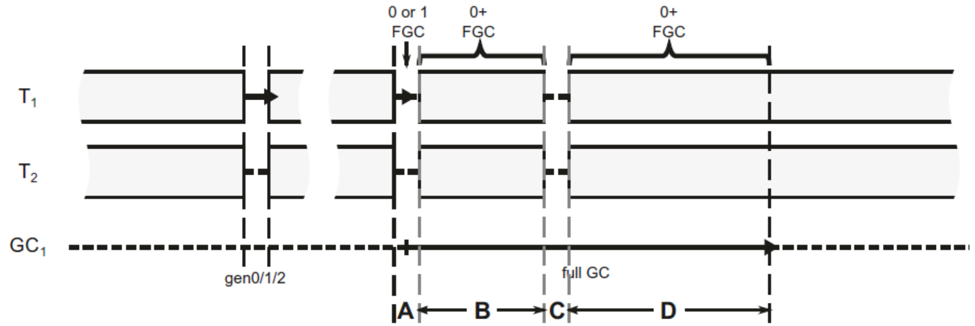
- `A` - initial "stop the world" phase - **initial work** list is being prepared from stack and finalization roots

Concurrent Mark



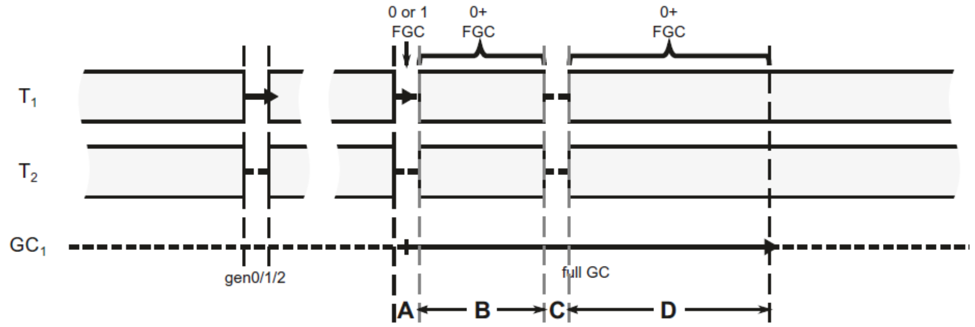
- `A` - initial "stop the world" phase - **initial work** list is being prepared from stack and finalization roots
- `B` - concurrent mark phase - the main work:
  - write barriers start to track modifications and store them in the **write watch list**
  - concurrent traversal happens - using "to visit list"
  - at the end, revisit objects from the write watch list

Concurrent Mark



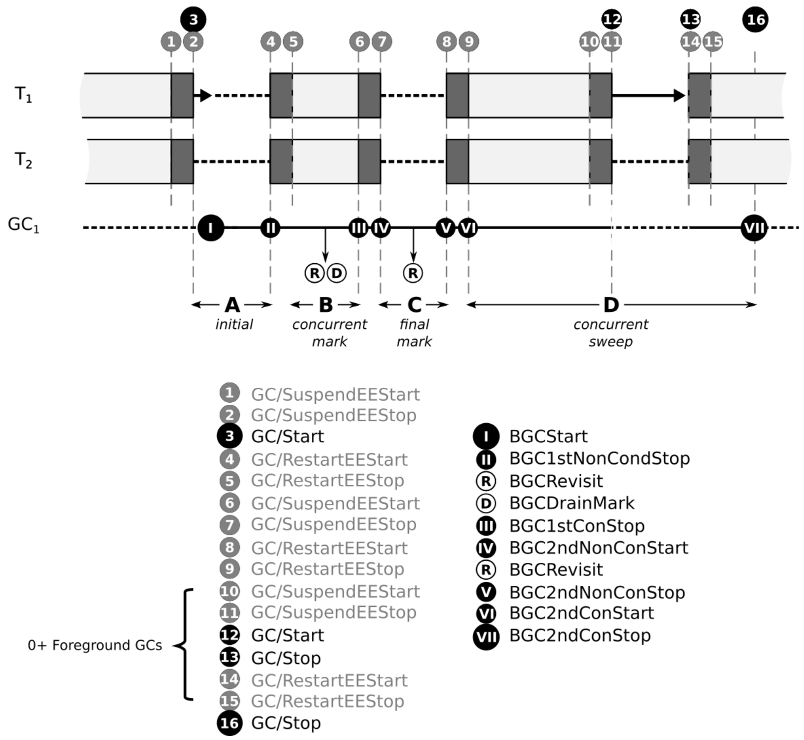
- `C` - final "stop the world" phase - to get "the final truth":
  - at this point the mark array should pretty well reflect the truth
  - we traverse again from the stack, finalization, handles, ...
    - it should be not a lot of work - most objects are already visited!
  - some final bookkeeping - scanning dependent handles and weak references

Concurrent Mark



- `C` - final "stop the world" phase - to get "the final truth":
  - at this point the mark array should pretty well reflect the truth
  - we traverse again from the stack, finalization, handles, ...
    - it should be not a lot of work - most objects are already visited!
  - some final bookkeeping - scanning dependent handles and weak references
- `D` - "garbage collection"

Concurrent Mark - events



- **`BGCDrainMark`** - information about the number of objects in a "initial work list"
- **`BGCRevisit`** - how many pages were "dirty" and how many objects have been eventually marked because of that

Concurrent Mark - challenge #1

- efficient implementation of the write barrier and/or write watch list **is not trivial**

Concurrent Mark - challenge #1

- efficient implementation of the write barrier and/or write watch list **is not trivial**
- historically, on Windows, instead of write barrier, there was *WriteWatch* mechanism usage to track *write watch list*-like data:
  - when allocating a page (4kB), enable *write watching* by `MEM_WRITE_WATCH` flag
  - later on you can retrieve a list of modified pages by `GetWriteWatch` system call

Concurrent Mark - challenge #1

- efficient implementation of the write barrier and/or write watch list **is not trivial**
- historically, on Windows, instead of write barrier, there was *WriteWatch* mechanism usage to track *write watch list*-like data:
  - when allocating a page (4kB), enable *write watching* by `MEM_WRITE_WATCH` flag
  - later on you can retrieve a list of modified pages by `GetWriteWatch` system call
- which is good enough compromise:
  - no write barrier overhead 👍👍
  - a lot of "false positive" roots - every `S.X = t` records the whole 4kB page 👎
  - blocked usage of the "large pages" 👎

Concurrent Mark - challenge #1

- efficient implementation of the write barrier and/or write watch list **is not trivial**
- historically, on Windows, instead of write barrier, there was *WriteWatch* mechanism usage to track *write watch list*-like data:
  - when allocating a page (4kB), enable *write watching* by `MEM_WRITE_WATCH` flag
  - later on you can retrieve a list of modified pages by `GetWriteWatch` system call
- which is good enough compromise:
  - no write barrier overhead 👍👍
  - a lot of "false positive" roots - every `S.X = t` records the whole 4kB page 👎
  - blocked usage of the "large pages" 👎
- BUT... during Linux port, it was hard to find WriteWatch counterpart API
  - instead of OS-magic, write barrier was used

Concurrent Mark - challenge #1

- efficient implementation of the write barrier and/or write watch list **is not trivial**
- historically, on Windows, instead of write barrier, there was *WriteWatch* mechanism usage to track *write watch list*-like data:
  - when allocating a page (4kB), enable *write watching* by `MEM_WRITE_WATCH` flag
  - later on you can retrieve a list of modified pages by `GetWriteWatch` system call
- which is good enough compromise:
  - no write barrier overhead 👍👍
  - a lot of "false positive" roots - every `S.X = t` records the whole 4kB page 👎
  - blocked usage of the "large pages" 👎
- BUT... during Linux port, it was hard to find WriteWatch counterpart API
  - instead of OS-magic, write barrier was used
- which worked so well that now even Windows uses it:
  - it has some, but tolerable, overhead👎
  - given that it is now the CLR code so...
  - much more "smart" - like ignoring no references `S.I = 44`👍
  - much more flexible - like it may have page-, large page-, "whatever-you-like"-granularity👍

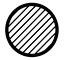
Concurrent Mark - challenge #1

- efficient implementation of the write barrier and/or write watch list **is not trivial**
- historically, on Windows, instead of write barrier, there was *WriteWatch* mechanism usage to track *write watch list*-like data:
  - when allocating a page (4kB), enable *write watching* by `MEM_WRITE_WATCH` flag
  - later on you can retrieve a list of modified pages by `GetWriteWatch` system call
- which is good enough compromise:
  - no write barrier overhead 👍👍
  - a lot of "false positive" roots - every `S.X = t` records the whole 4kB page 👎
  - blocked usage of the "large pages" 👎
- BUT... during Linux port, it was hard to find WriteWatch counterpart API
  - instead of OS-magic, write barrier was used
- which worked so well that now even Windows uses it:
  - it has some, but tolerable, overhead👎
  - given that it is now the CLR code so...
  - much more "smart" - like ignoring no references `S.I = 44`👍
  - much more flexible - like it may have page-, large page-, "whatever-you-like"-granularity👍 However, it is still (current) page-size.

Concurrent Mark phase - inside code

In case of CoreCLR, the core code responsible for concurrent marking exists in `gc_heap::background_mark_phase` method. The three most important data structures are:

- `mark_array`, as we know it already,
- `background_mark_stack_array` for concurrent "to visit list" (aka "mark stack")
- `c_mark_list`, realizing "initial work list" populated at the initial phase

`c_mark_list` is populated with `gc_heap::background_promote_callback` method during stack and finalization queue scanning and then consumed by `gc_heap::background_drain_mark_list` method. This method calls `background_mark_object` [] for all objects in `c_mark_list` and fires a single `BGCDrainMark` event at the end (with the initial list size).

Concurrent Mark phase - inside code

As `FEATURE_USE_SOFTWARE_WRITE_WATCH_FOR_GC_HEAP` is defined, it enables the software write watch mechanism. You may see its usage in write barriers like `JIT_WriteBarrier_WriteWatch_PreGrow64`. The software write watch list is then consumed by `gc_heap::revisit_written_pages` method. It calls `revisit_written_page` (using the same `background_mark_object` [ ] on objects inside, one by one) on pages returned from `get_write_watch_for_gc_heap` method. At the end, a single `BGCRevisit` event is called with the "dirted" pages & marked objects counts.

`get_write_watch_for_gc_heap` uses 4kB-wide ("page") granularity and is tracked per byte of the table (to avoid multithreading issues) - see the `AddressToTableByteIndexShift` in `softwarewritewatch.h`.

Historically, the write watch list in case of Windows managed by the system itself and is consumed in the GC within `gc_heap::revisit_written_pages` method by calling `GCToOSInterface::GetWriteWatch`.

Concurrent Mark phase - inside code

All "regular" concurrent marking is done with the help of **`gc_heap::background_promote(obj,...)`** method that through **`background_mark_simple(obj)`** and **`background_mark_simple1(obj)`** (the one utilizing/consuming **`background_mark_stack_array`**) traverses the object's graph (marking corresponding bits in **`mark_array`** inside **`background_mark1(obj,...)`** method).