.NET GC Internals

[Concurrent] Sweep phase

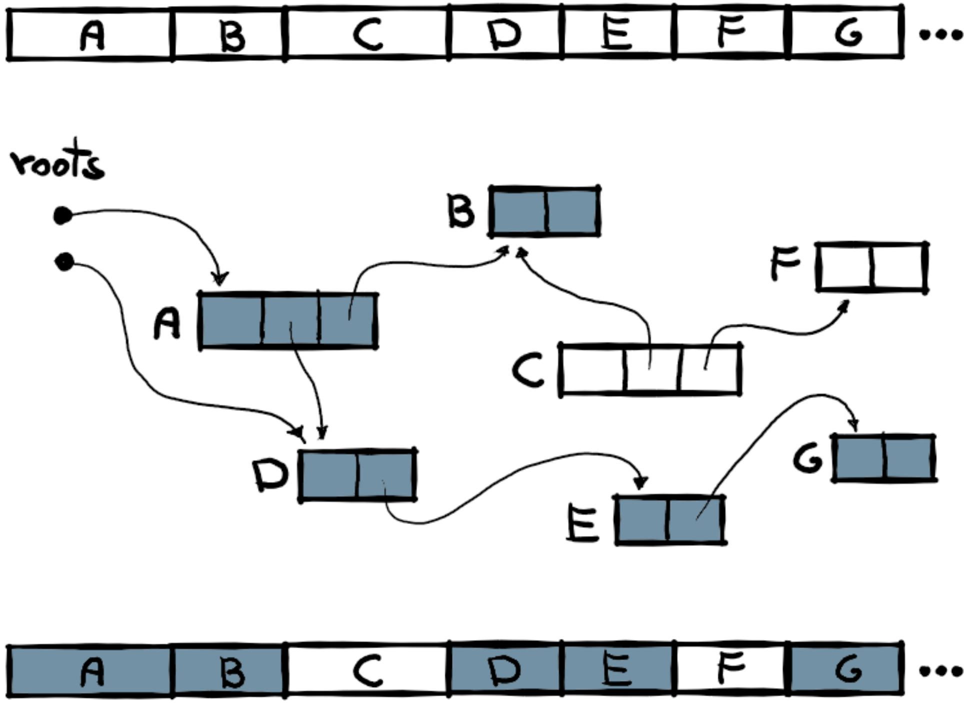
@konradkokosa / @dotnetosorg

.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?", ...

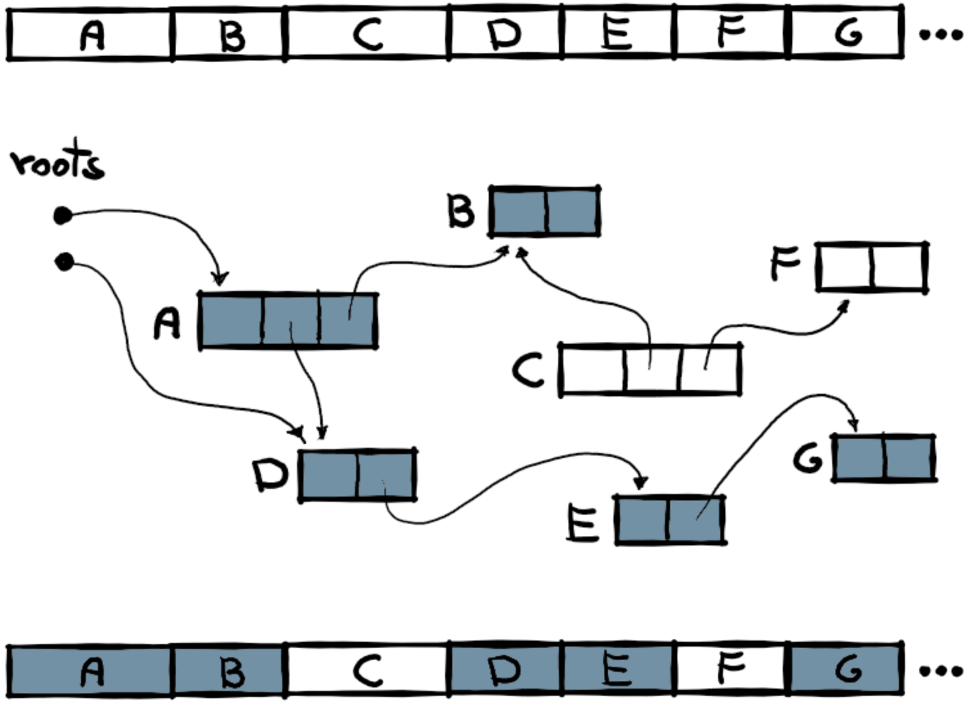
Sweep

All no-longer reachable objects must be turned into a free space:



Sweep

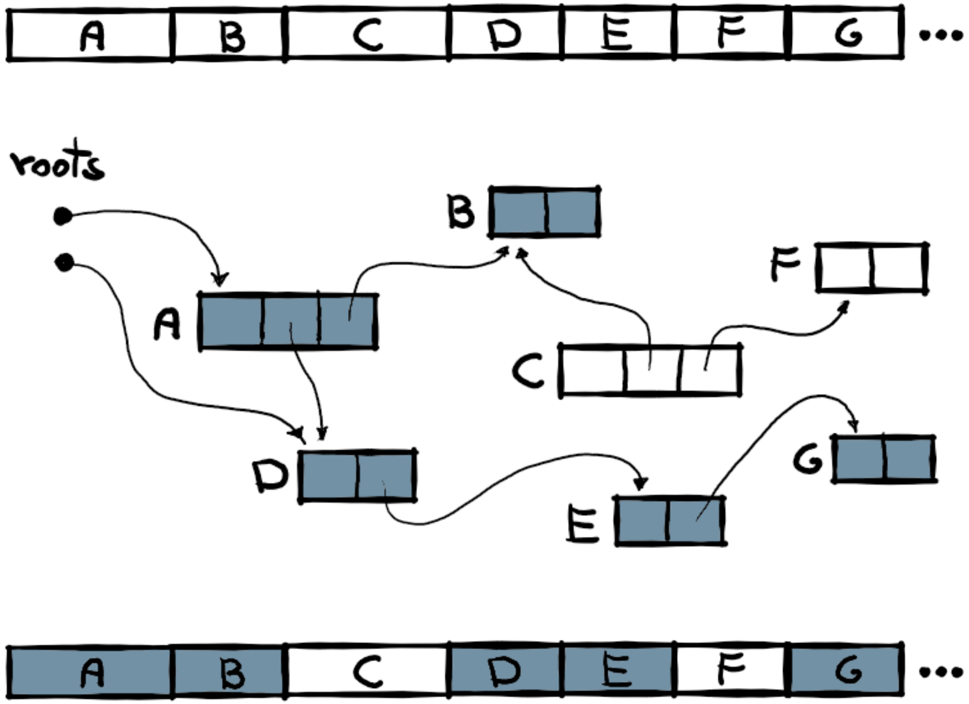
All no-longer reachable objects must be turned into a free space:



In the .NET GC terminology, it means that it must **transform all or some gaps into free-list items**.

Sweep

All no-longer reachable objects must be turned into a free space:

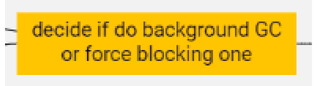


In the .NET GC terminology, it means that it must **transform all or some gaps into free-list items**. Free-list items are then used for "allocations".

Mark, Plan, Sweep, Compact...

Let's make a short stop here.

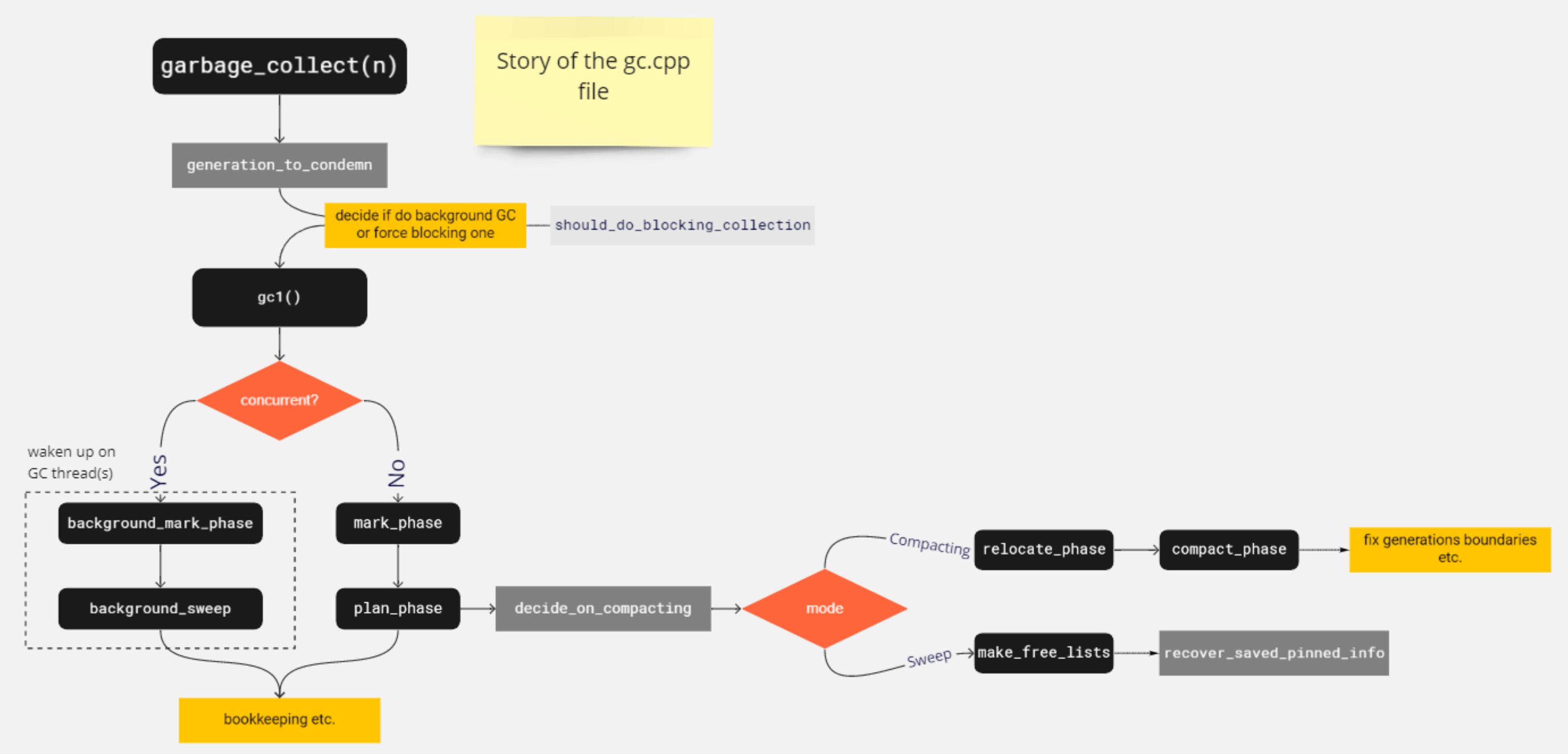
Concurrent GC?



is:

```
if ((settings.condemned_generation == max_generation) &&
    (should_do_blocking_collection == FALSE) &&
    gc_can_use_concurrent &&
    !temp_disable_concurrent_p &&
    ((settings.pause_mode == pause_interactive) || (settings.pause_mode == pause_sustained_low_latency)))
{
    keep_bgc_threads_p = TRUE;
    c_write (settings.concurrent, TRUE);
    memset (&bgc_data_global, 0, sizeof(bgc_data_global));
    memcpy (&bgc_data_global, &gc_data_global, sizeof(gc_data_global));
}
```

Non-Concurrent Sweep



So, we are after *Mark* & *Plan* phases.

Non-Concurrent Sweep

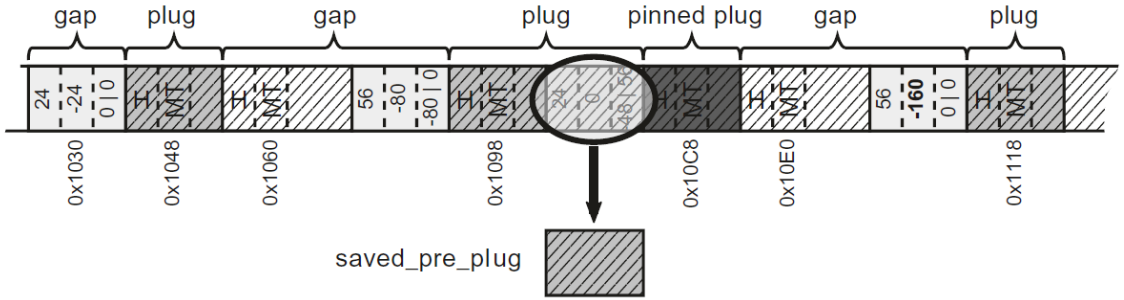
Using the knowledge from the *Plan* phase, go by plug-gap pair and:

- for every gap:
  - if bigger that 2x minimum object size - create *free-list items* from it
  - if smaller - treat as **unused free space/fragmentation**
- recover pre/post-plugs
- ... (additional bookkeeping)

Non-Concurrent Sweep

Using the knowledge from the *Plan* phase, go by plug-gap pair and:

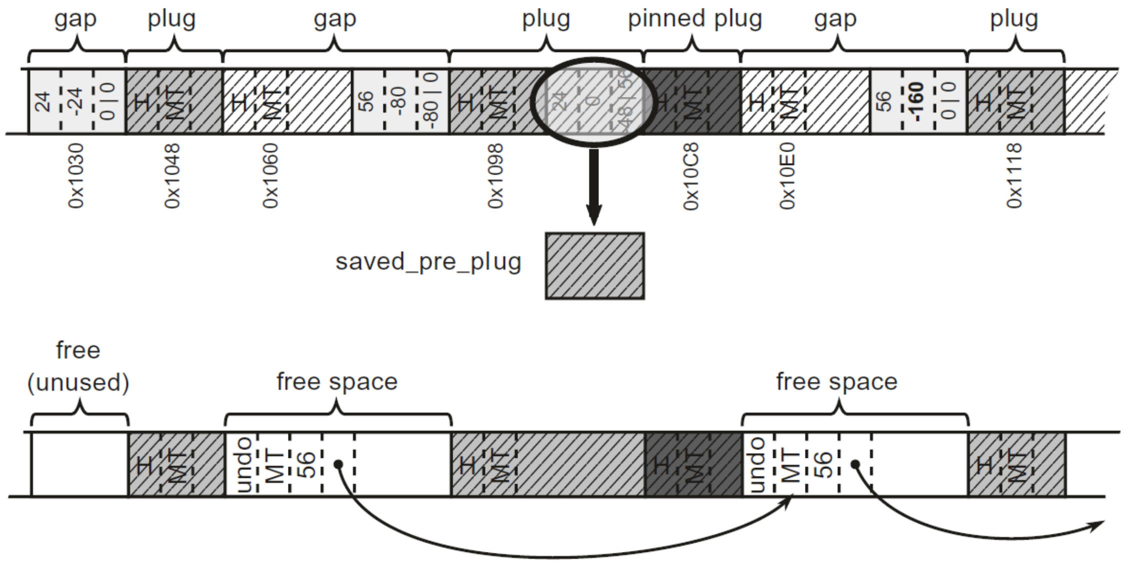
- for every gap:
  - if bigger that 2x minimum object size - create *free-list items* from it
  - if smaller - treat as **unused free space/fragmentation**
- recover pre/post-plugs
- … (additional bookkeeping)



Non-Concurrent Sweep

Using the knowledge from the *Plan* phase, go by plug-gap pair and:

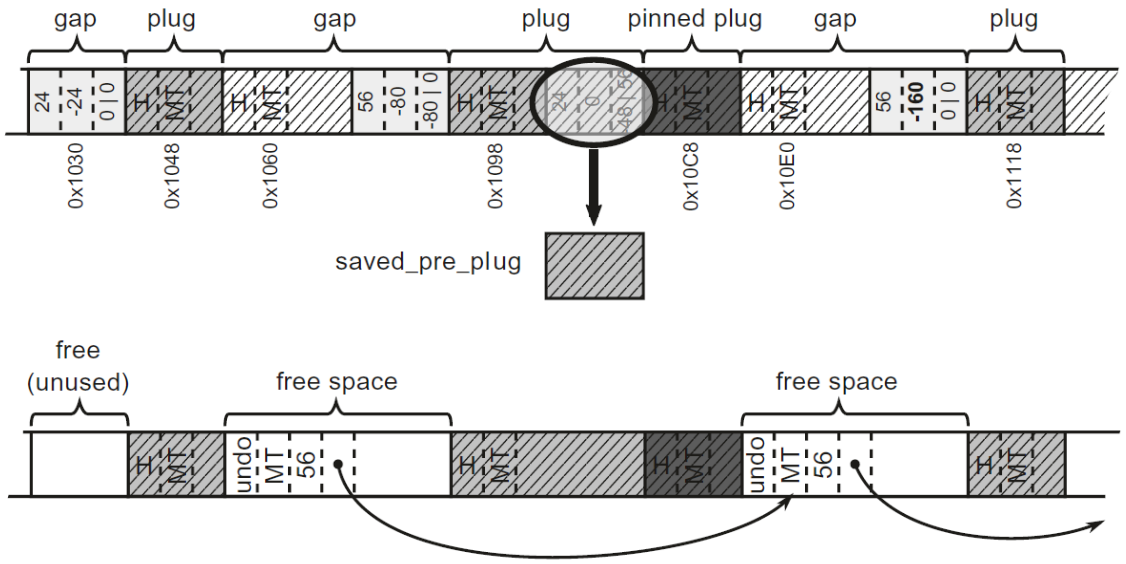
- for every gap:
  - if bigger that 2x minimum object size - create *free-list items* from it
  - if smaller - treat as **unused free space/fragmentation**
- recover pre/post-plugs
- … (additional bookkeeping)



Non-Concurrent Sweep

Using the knowledge from the *Plan* phase, go by plug-gap pair and:

- for every gap:
  - if bigger that 2x minimum object size - create *free-list items* from it
  - if smaller - treat as **unused free space/fragmentation**
- recover pre/post-plugs
- … (additional bookkeeping)

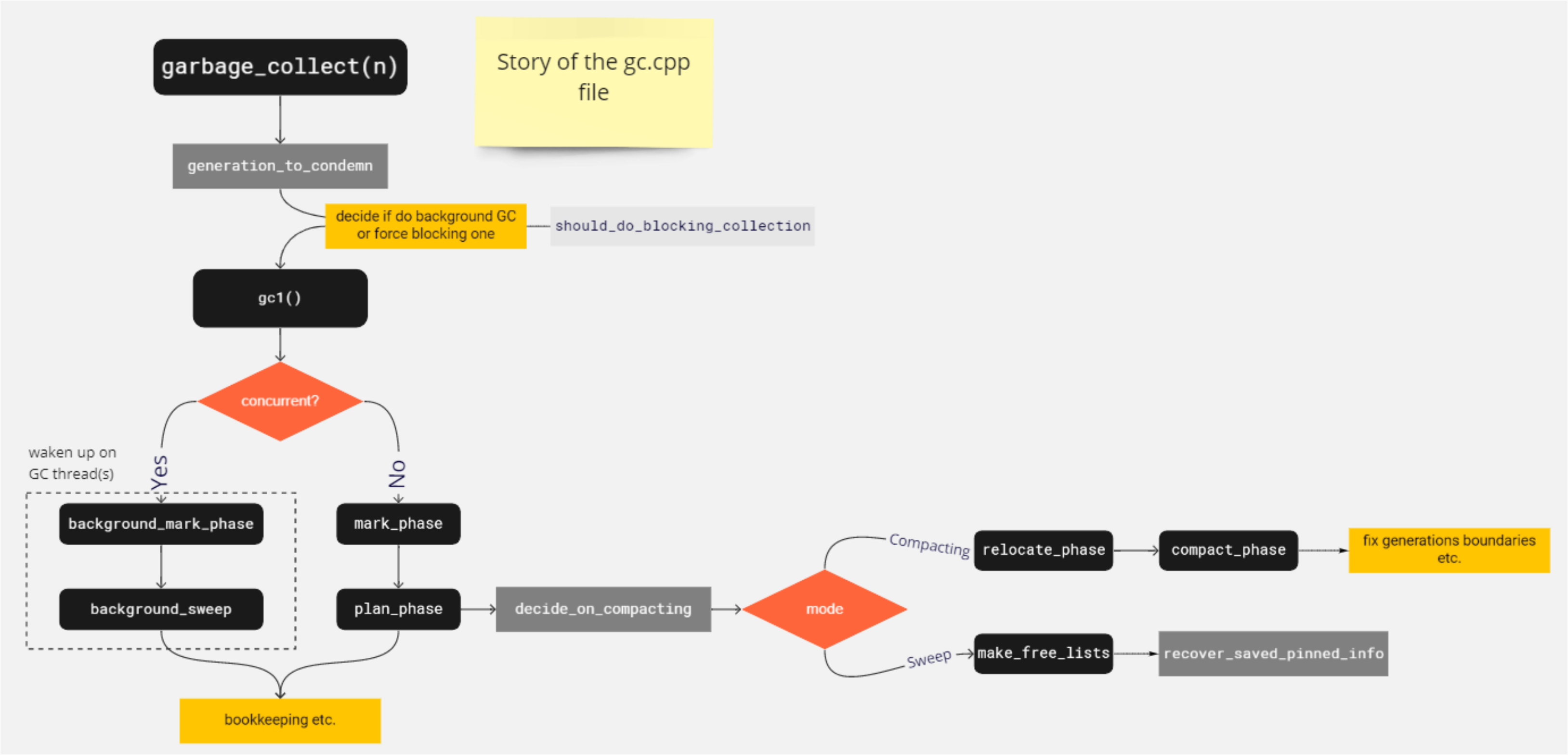


Important: we **don't zero** memory now - why bother?!

Sweep - Large Object Heap

If no compacting, there is no *Plan* phase for it. Just go object by object and thread *free-list* items from not marked ones.

Concurrent Sweep

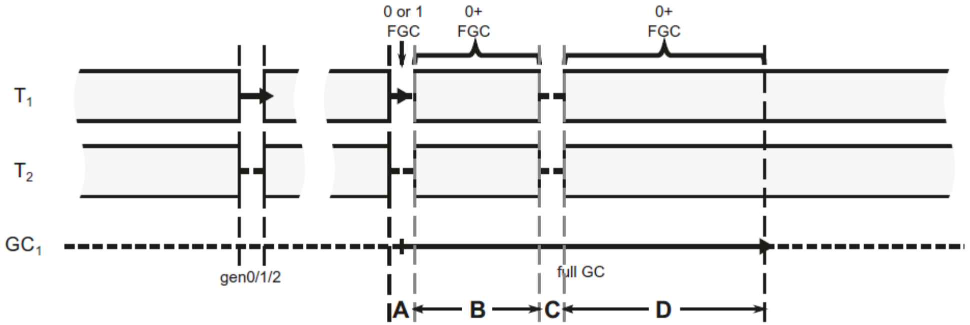


So, we didn't have *Plan* phase! We are just doing **`background_mark_phase`** (populating **`mark_array`** aka *mark list*) and **`background_sweep`**.

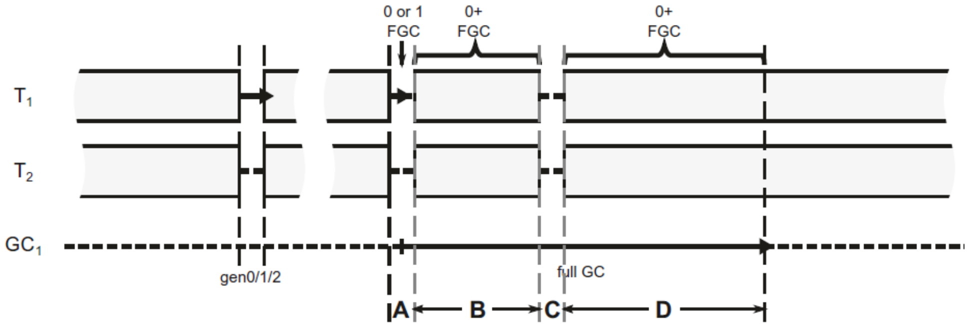
Concurrent Sweep

`background_sweep` is similar to the non-concurrent *Plan* phase - it scans object by object to group non-reachable objects into gaps and threads them into free-list items.

Concurrent Sweep

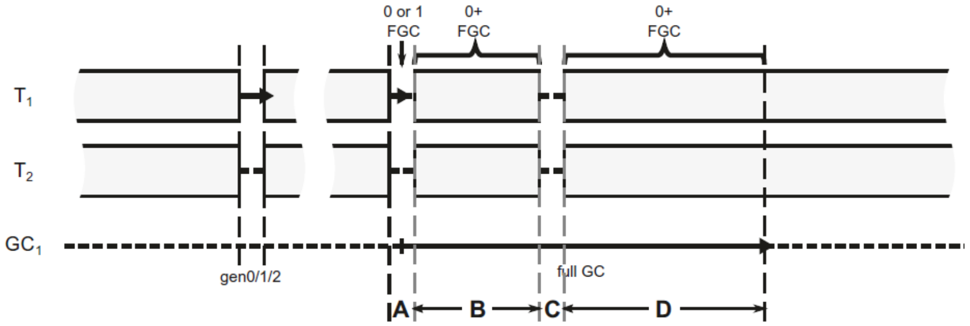


Concurrent Sweep



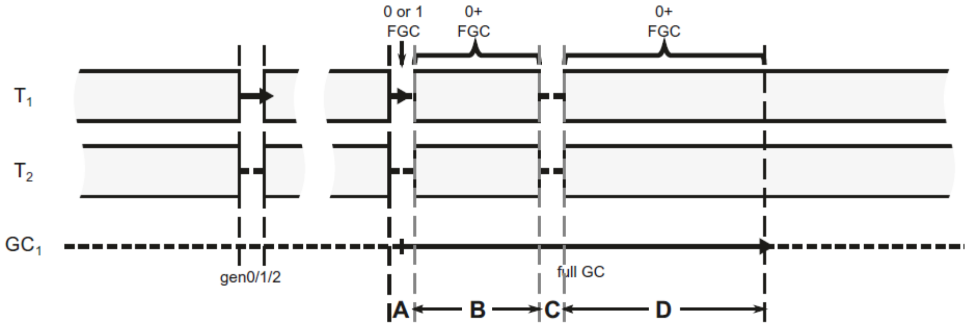
- `D` is Concurrent Sweep - the *mark array* contains information about all marked (reachable) objects

Concurrent Sweep

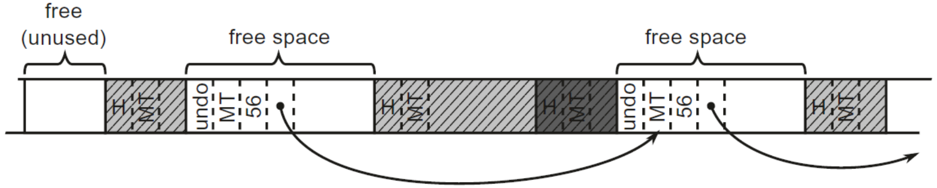


- `D` is Concurrent Sweep - the *mark array* contains information about all marked (reachable) objects
- we sweep **concurrently** with the application (which may be allocating) 😱

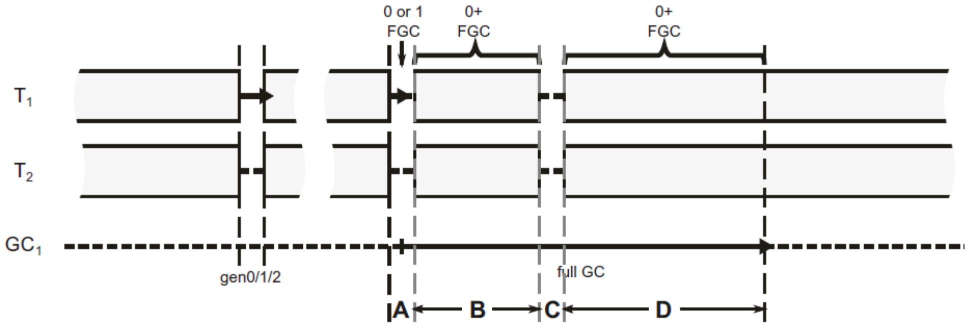
Concurrent Sweep



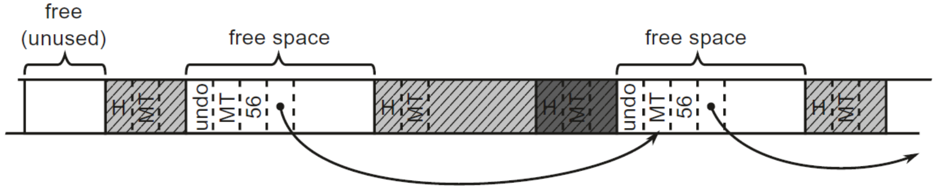
- `D` is Concurrent Sweep - the *mark array* contains information about all marked (reachable) objects
- we sweep **concurrently** with the application (which may be allocating) 😱
- however, we sweep only already not reachable object - no worries!



Concurrent Sweep

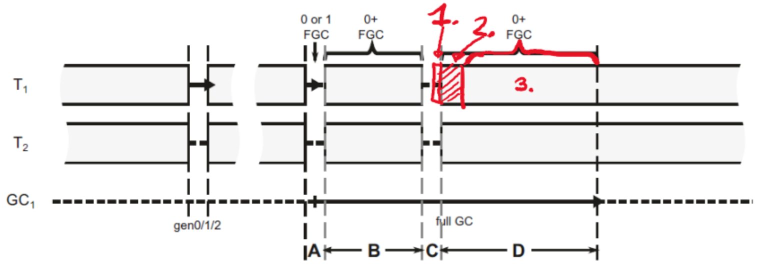


- `D` is Concurrent Sweep - the *mark array* contains information about all marked (reachable) objects
- we sweep **concurrently** with the application (which may be allocating) 😱
- however, we sweep only already not reachable object - no worries!



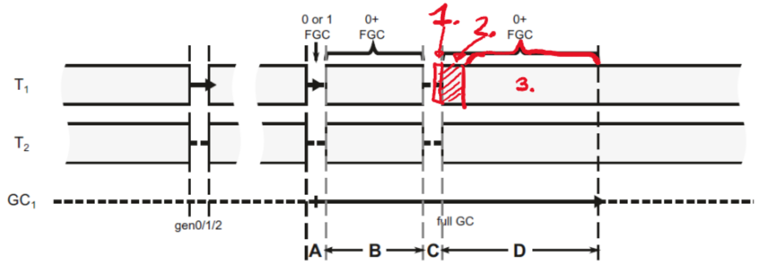
- but... allocators use free-list items, created during sweeping... so how does it all cooperate?!🤔

Concurrent Sweep



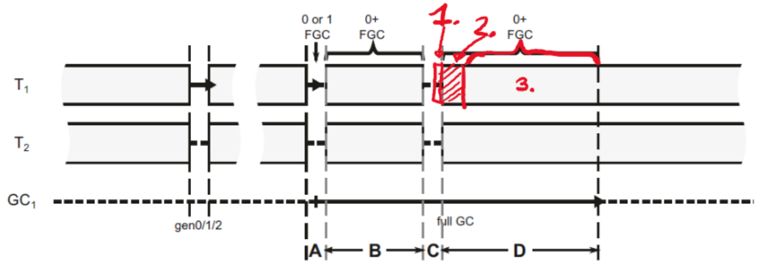
1. **before the runtime resumes threads** - free-item lists are cleared in all Gen(s)
    - yes, allocators will not be aware of free space for a short period of time (just allocating at the end of already consumed space)

Concurrent Sweep



1. **before the runtime resumes threads** - free-item lists are cleared in all Gen(s)
   - yes, allocators will not be aware of free space for a short period of time (just allocating at the end of already consumed space)
2. **concurrent Sweep of Gen0/1** - operating on a temporary *free-list*:
   - this temporary list is published when finished (and gen 0/1 starts to use it)
   - Foreground GC is prohibited

Concurrent Sweep



1. **before the runtime resumes threads** - free-item lists are cleared in all Gen(s)
    - yes, allocators will not be aware of free space for a short period of time (just allocating at the end of already consumed space)
2. **concurrent Sweep of Gen0/1** - operating on a temporary *free-list*:
    - this temporary list is published when finished (and gen 0/1 starts to use it)
    - Foreground GC is prohibited
3. **concurrent Sweep of Gen2/LOH** - operating on the main *free-list*:
    - Foreground GCs are allowed - if object gets promoted from gen1 to gen2 it uses already added free-list items. It is safe and without overhead as Background GC is suspended for the time of Foreground GC.
    - LOH allocations are not allowed - it would require multithreaded access to the free list

Concurrent Sweep

"LOH allocations are not allowed":

ETW/LLTng events `BGCAllocWaitBegin`/`BGCAllocWaitEnd` used to show *"LOH Allocation Pause (due to background GC) > 200 Msec"* section in PerfView’s GCStats report

Non-Concurrent Sweep

"Start from the **`gc_heap::plan_phase`** method. In the part enclosed by **`else`** block of **`should_compact`** conditional check, the two most important methods are called: **`gc_heap::make_free_lists`** creates free-list items from gaps and **`gc_heap::recover_saved_pinned_info`** recovers objects destroyed by pre and post plugs.

The main work horse is **`make_free_list_in_brick`** that recursively traverse plug tree to thread free items from gaps."

Concurrent Sweep

"In case of CoreCLR code, concurrent sweep phase is included in the `gc_heap::background_sweep` method. It calls `gc_heap::background_ephemeral_sweep` method scanning objects from generation 0 and 1, and then scans objects from generation 2 and Large Object Heap (calling `gc_heap::allow_fgc` method at some well-defined safe points, after each of 256 objects has been scanned). During object scanning, already known `gc_heap::thread_gap` or `gc_heap::make_unused_array` methods are used to create a free-list item or small unusable free space respectively.

Mentioned LOH allocations are blocked by global `gc_heap::gc_lh_block_event` which is used in `gc_heap::wait_for_background_planning` by calling `gc_heap::user_thread_wait` on it. This path is used at the beginning of the `gc_heap::a_fit_free_list_large_p` method, which is in fact the begging of the entire LOH allocation path."

Concurrent Sweep

"For code related to *free object*, start from `gc_heap::make_unused_array method`, which prepares it. As you will see it uses static global pointer to `g_pFreeObjectMethodTable` as a new MT. Then it adds such gap to the free list by calling `generation_allocator(gen)->thread_item (gap_start, size)`. However, threading is done only for gaps larger than the double size of the minimum object size. This helps to ignore the list management overhead for such small items."