.NET GC Internals

Generations

@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, ...
- **Allocations** - *bump pointer allocator*, free list allocator, *allocation context*, ...
- **Generations** - physical organization, *card tables*, demotion, ...
- **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?", ...

Partitioning

By object:

- size
- type/kind
- mutability
- lifetime
- ...

Partitioning

Size:

- copying cost!
- different quantities
  - many, many small objects often allocated 👉 **Small Object Heap**
  - rare large objects 👉 **Large Object Heap**

Partitioning

Size:

- copying cost!
- different quantities
  - many, many small objects often allocated 👉 **Small Object Heap**
  - rare large objects 👉 **Large Object Heap**

Type/kind:

- pinned or not pinned 👉 **Pinned Object Heap** (.NET 5+)

Partitioning

Size:

- copying cost!
- different quantities
  - many, many small objects often allocated 👉 **Small Object Heap**
  - rare large objects 👉 **Large Object Heap**

Type/kind:

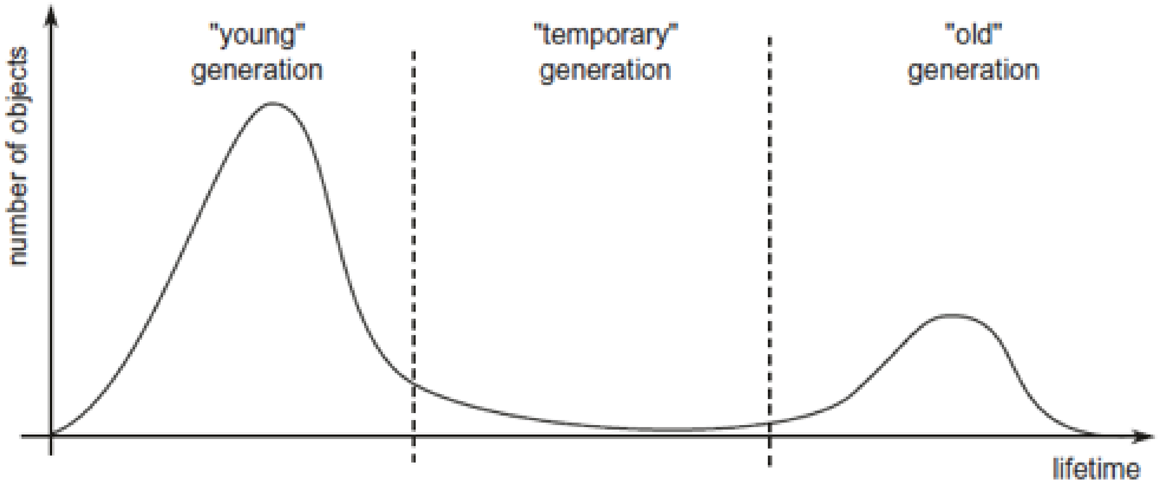
- pinned or not pinned 👉 **Pinned Object Heap** (.NET 5+)

Lifetime:

- *"many, many small objects often allocated"* - sensible to split SOH even further

Partitioning - lifetime

weak/strong generational hypothesis



Partitioning - logical

Partitioning - logical

- object allocation in gen0/LOH

Partitioning - logical

- object allocation in gen0/LOH
- GC collects given generation and all younger, so we have:
    - gen 0 GC
    - gen 1 GC - gen 0&1
    - Full GC - gen 0&1&2, LOH and POH (kind of LOH & POH treated as gen 2)

Partitioning - logical

- object allocation in gen0/LOH
- GC collects given generation and all younger, so we have:
  - gen 0 GC
  - gen 1 GC - gen 0&1
  - Full GC - gen 0&1&2, LOH and POH (kind of LOH & POH treated as gen 2)
- if an objects in generation X survives the GC considering this generation, is **promoted** to generation X+1 (or stays in gen2)

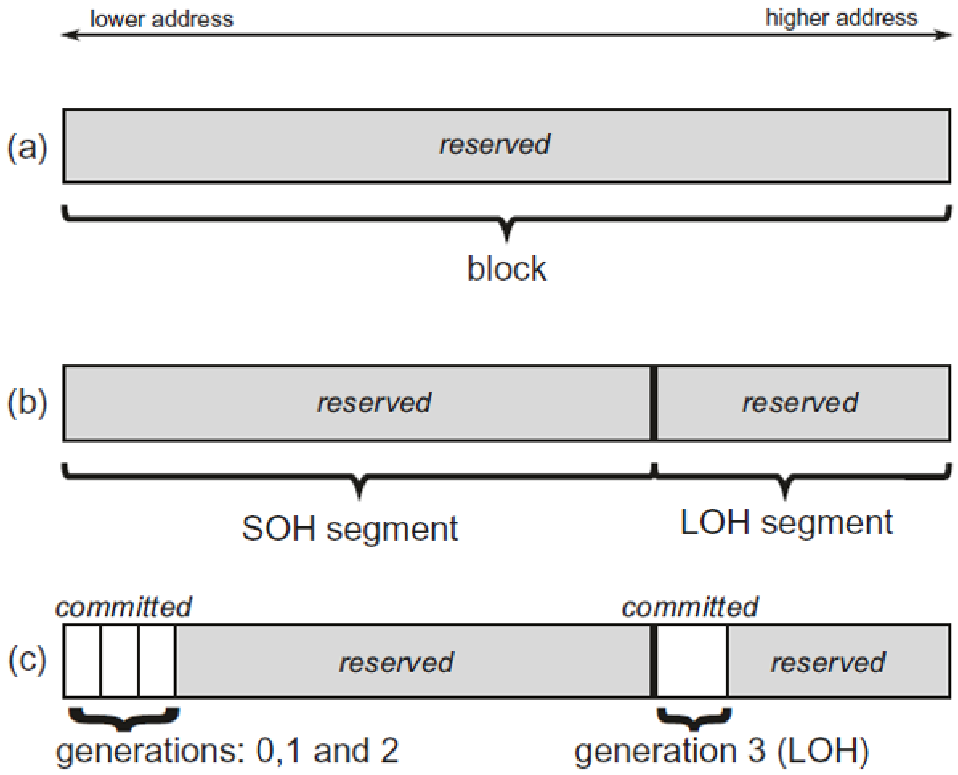
Partitioning - logical

- object allocation in gen0/LOH
- GC collects given generation and all younger, so we have:
  - gen 0 GC
  - gen 1 GC - gen 0&1
  - Full GC - gen 0&1&2, LOH and POH (kind of LOH & POH treated as gen 2)
- if an objects in generation X survives the GC considering this generation, is **promoted** to generation X+1 (or stays in gen2)
- when generation isn't collected, we simply treat all objects in that generation as live

Partitioning - logical

- object allocation in gen0/LOH
- GC collects given generation and all younger, so we have:
  - gen 0 GC
  - gen 1 GC - gen 0&1
  - Full GC - gen 0&1&2, LOH and POH (kind of LOH & POH treated as gen 2)
- if an objects in generation X survives the GC considering this generation, is **promoted** to generation X+1 (or stays in gen2)
- when generation isn't collected, we simply treat all objects in that generation as live
- generations are considered in various places, fe. finalization queue is generational too

Partitioning - physical

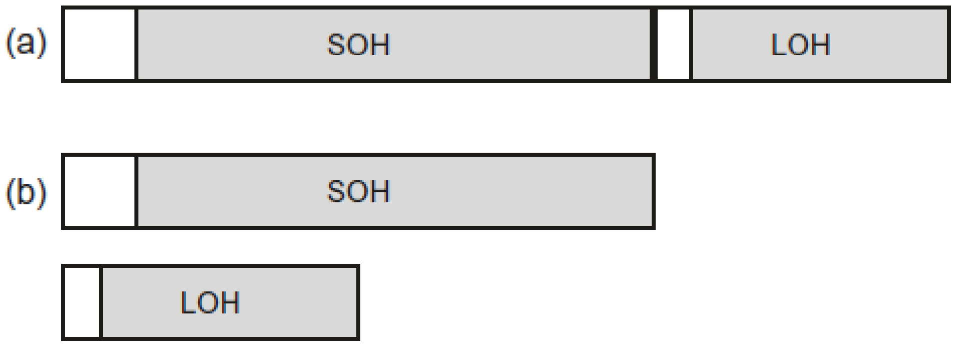


Partitioning - physical

0000026700000000	Managed Heap	393,216 K	336 K	336 K	224 K	224 K	4	Read/Write	GC
0000026700000000	Managed Heap	4 K	4 K	4 K	4 K	4 K		Read/Write	
0000026700001000	Managed Heap	24 bytes	24 bytes	24 bytes				Read/Write	Gen2
0000026700001018	Managed Heap	24 bytes	24 bytes	24 bytes				Read/Write	Gen1
0000026700001030	Managed Heap	259 K	259 K	259 K	204 K	204 K		Read/Write	Gen0
0000026700042000	Managed Heap	261,880 K						Reserved	
0000026710000000	Managed Heap	72 K	72 K	72 K	16 K	16 K		Read/Write	Large Object Heap
0000026710012000	Managed Heap	131,000 K						Reserved	

	Workstation		**Server**	
	32-bit	**64-bit**	**32-bit**	**64-bit**
SOH	16 MB	256 MB	64 MB (#CPU<=4) 32 MB (#CPU<=8) 16 MB (#CPU>8)	4 GB (#CPU<=4) 2 GB (#CPU<=8) 1 GB (#CPU>8)
LOH	16 MB	128 MB	32 MB	256 MB

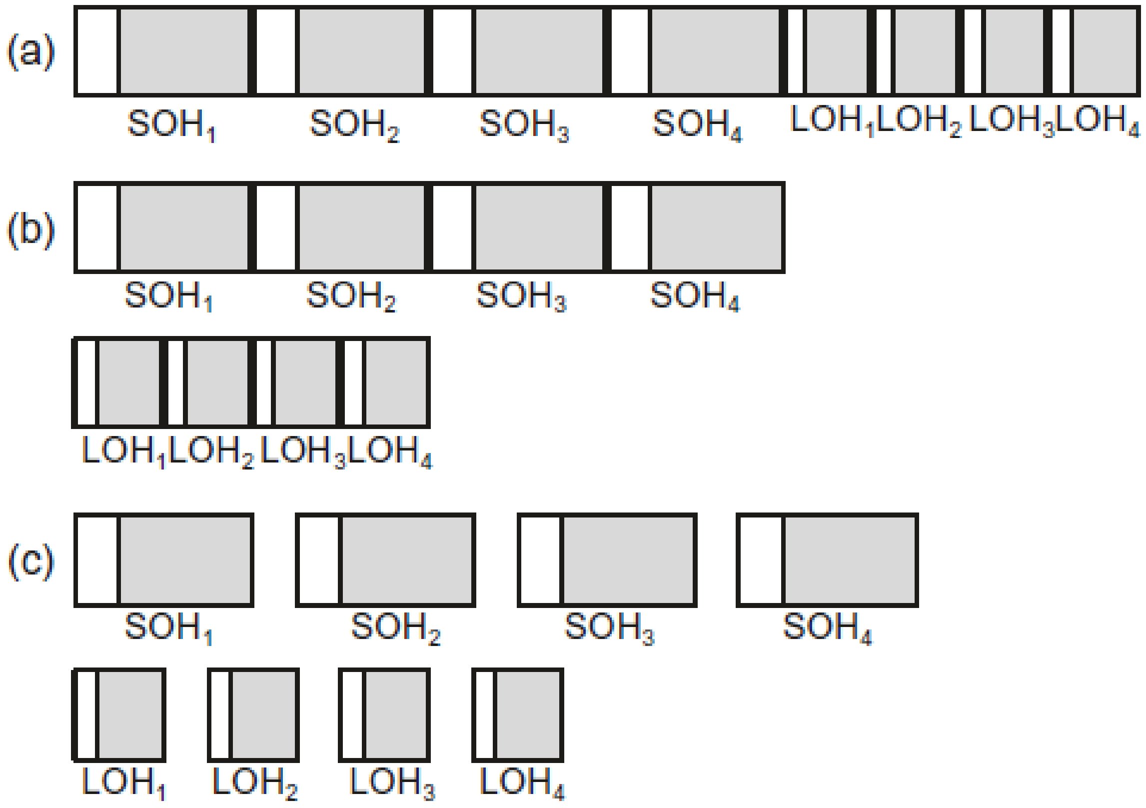
Partitioning - physical (Workstation GC)



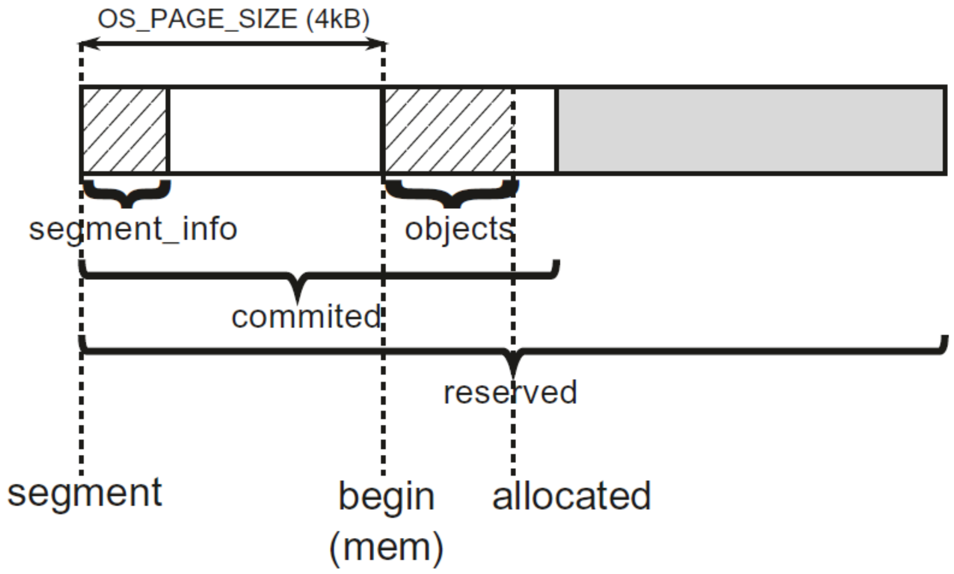
(a) all-at- once configuration,

(b) two-stage configuration (the same as each-block configuration)

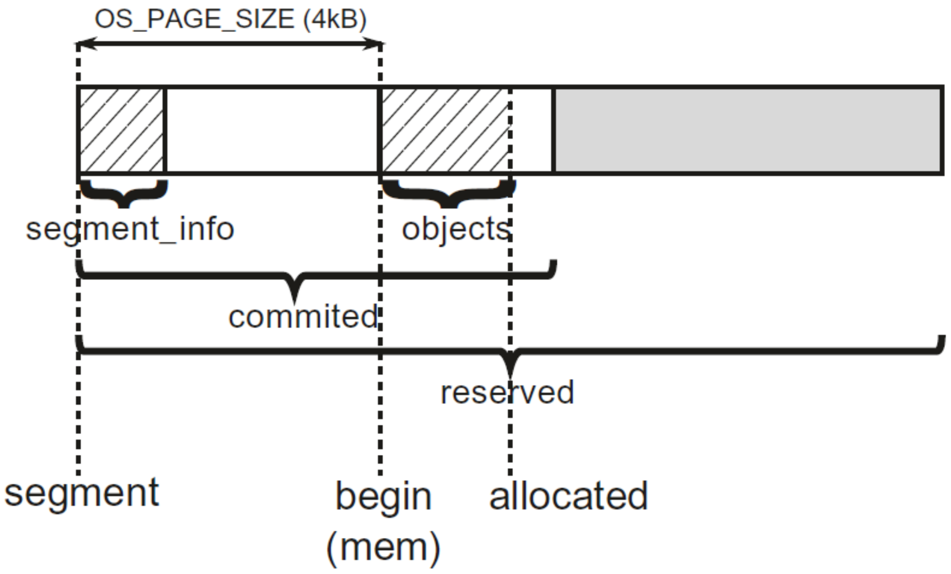
Partitioning - physical (Server GC)

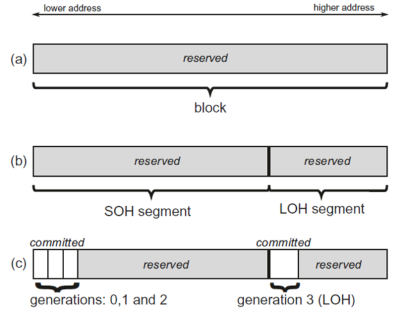


Partitioning - segment "anatomy"



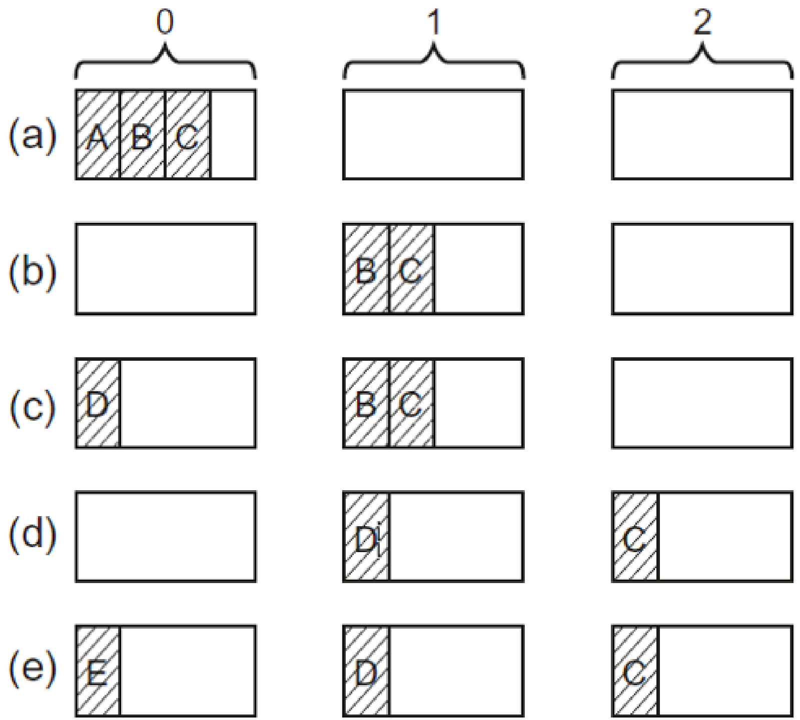
Partitioning - segment "anatomy"



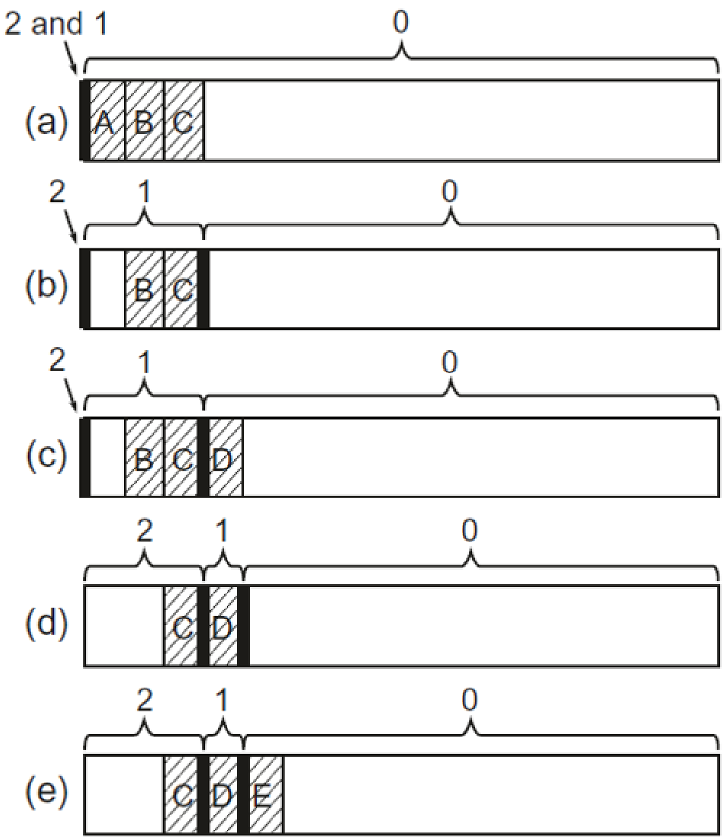


Generations

Generations



Generations - Sweeping overview



(a) Objects **A**, **B** and **C** were allocated.

(b) GC was triggered and only **B** & **C** survived. Gen1 boundary is extended to include promoted **B** and **C**.

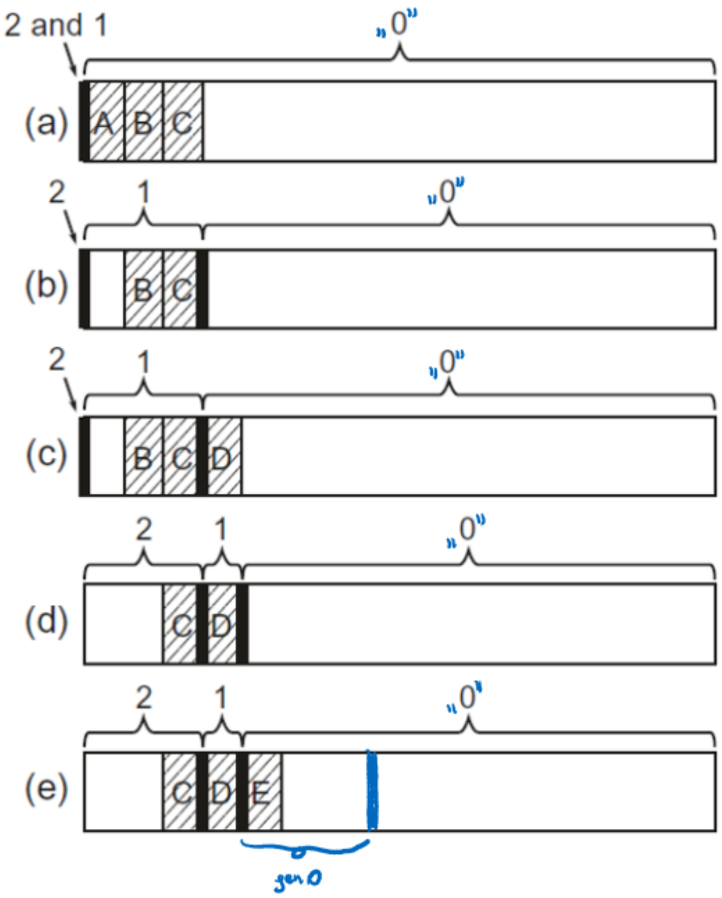
(c) Object **D** was allocated.

(d) GC was triggered and only **C** & **D** survived. Generations boundaries are extended to include **C** (promoted to gen2) and **D** (promoted to gen1).

(e) Object **E** was allocated.

(.) and the story continues...

Generations - Sweeping overview



(a) Objects **A**, **B** and **C** were allocated.

(b) GC was triggered and only **B** & **C** survived. Gen1 boundary is extended to include promoted **B** and **C**.

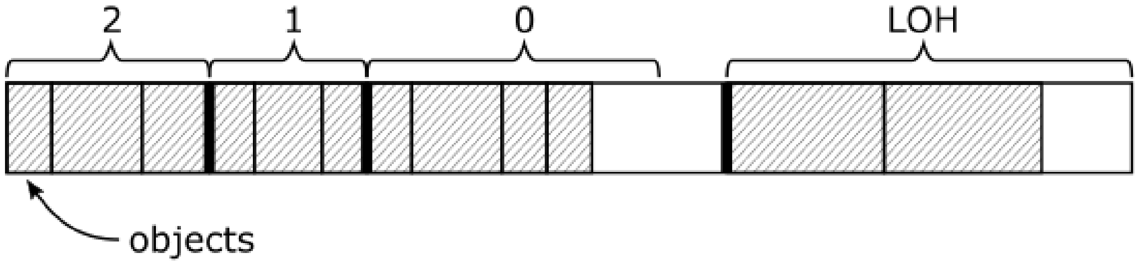
(c) Object **D** was allocated.

(d) GC was triggered and only **C** & **D** survived. Generations boundaries are extended to include **C** (promoted to gen2) and **D** (promoted to gen1).

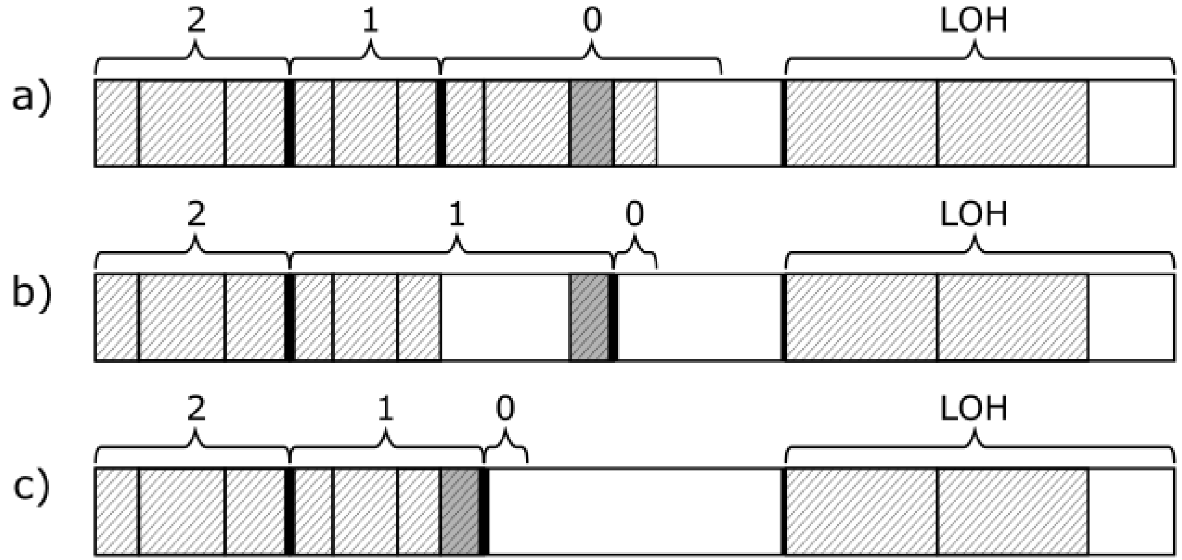
(e) Object **E** was allocated.

(.) and the story continues...

Generations



Generations - gen0 GC

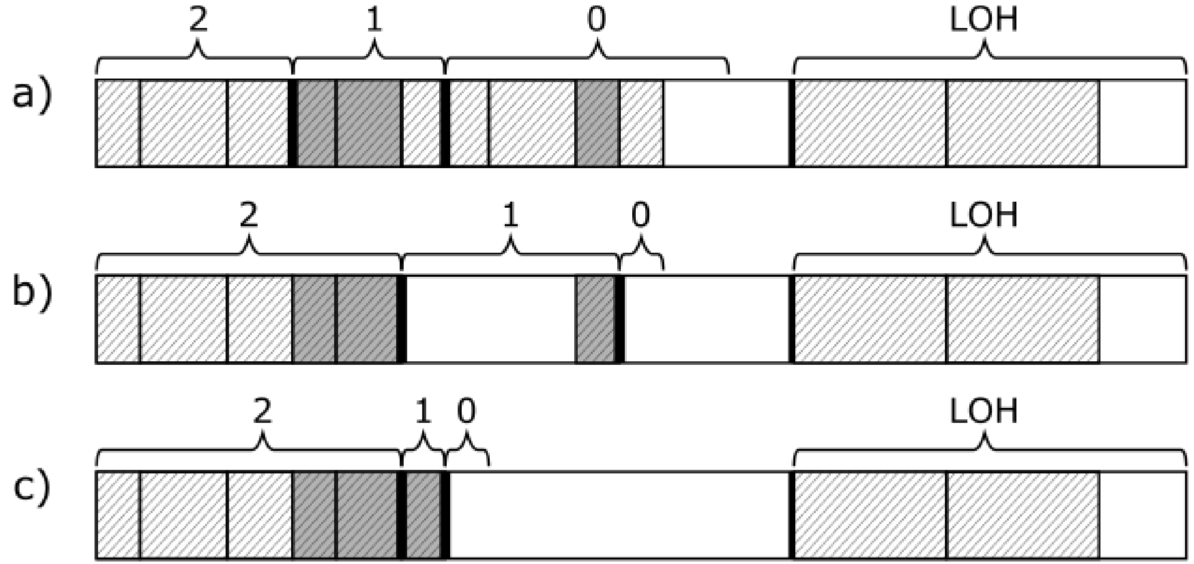


(a) After *Mark*

(b) After *Sweep* or...

(c) After *Compact*

Generations - gen1 GC

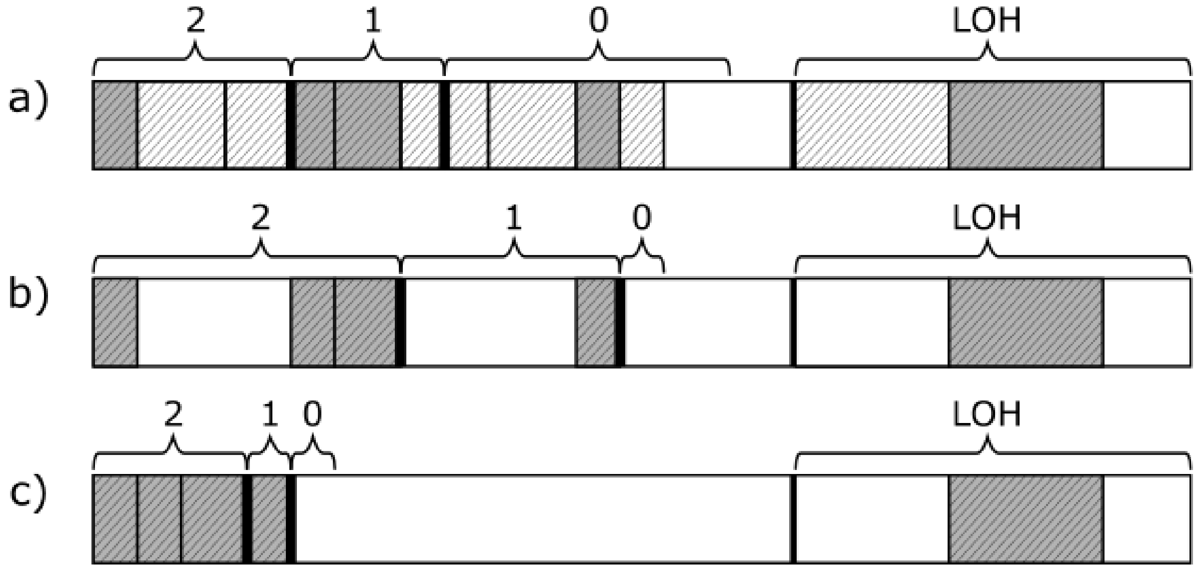


(a) After *Mark*

(b) After *Sweep* or...

(c) After *Compact*

Generations - gen2 GC ("Full GC")



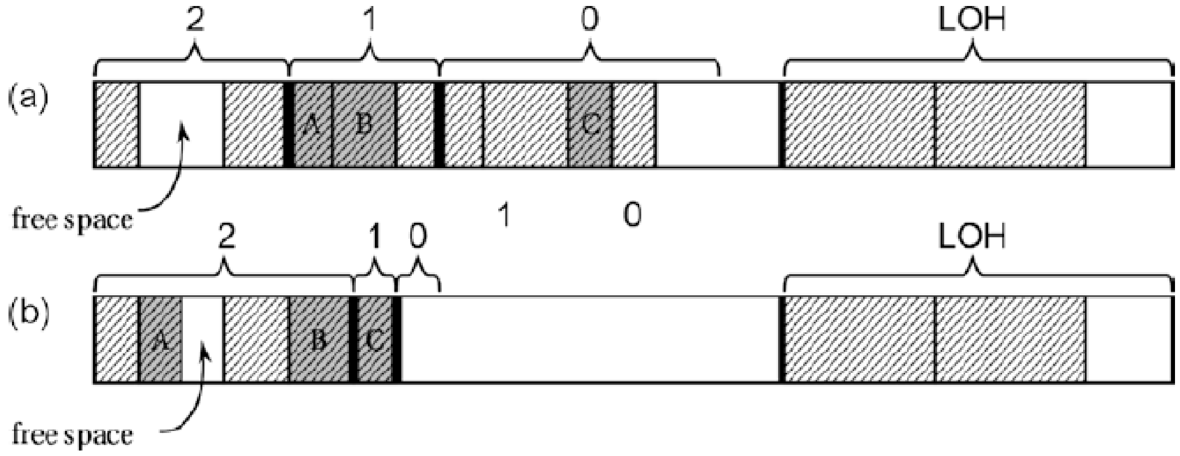
(a) After *Mark*.

(b) After *Sweep* or...

(c) After *Compact*

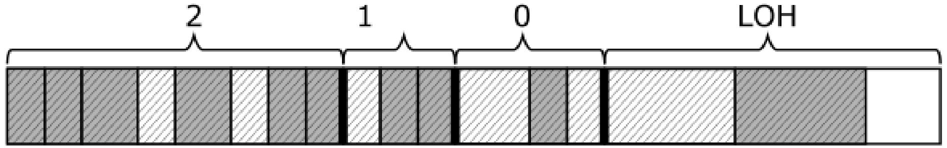
Generations - gen0 & gen1

While compacting, we may "allocate in the older generation" some promoted plugs - to make use of free-list allocations and fight/reuse fragmentation😍



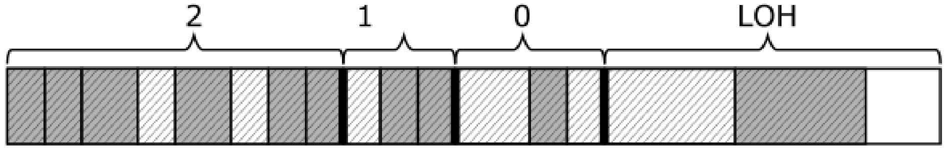
Generations - running out of SOH segment space

At some point gens may grow not to fit into SOH segment - gen0/1 may not have enough space:

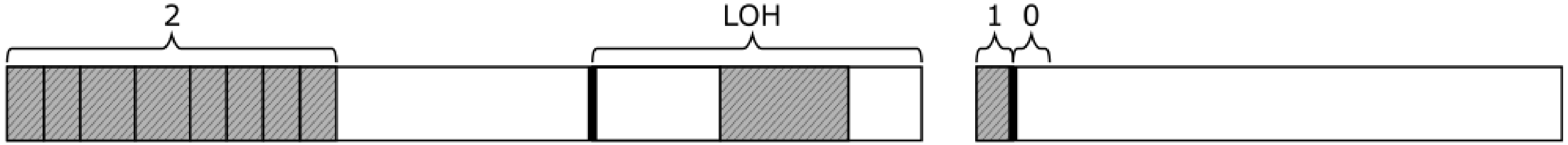


Generations - running out of SOH segment space

At some point gens may grow not to fit into SOH segment - gen0/1 may not have enough space:

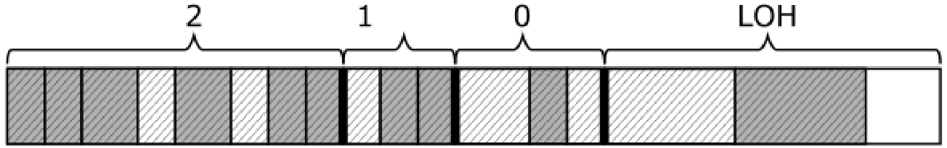


A new *ephemeral* segment will be created during compacting GC:

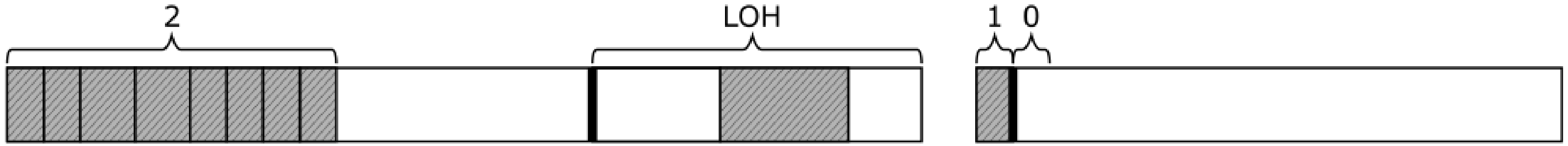


Generations - running out of SOH segment space

At some point gens may grow not to fit into SOH segment - gen0/1 may not have enough space:



A new *ephemeral* segment will be created during compacting GC:



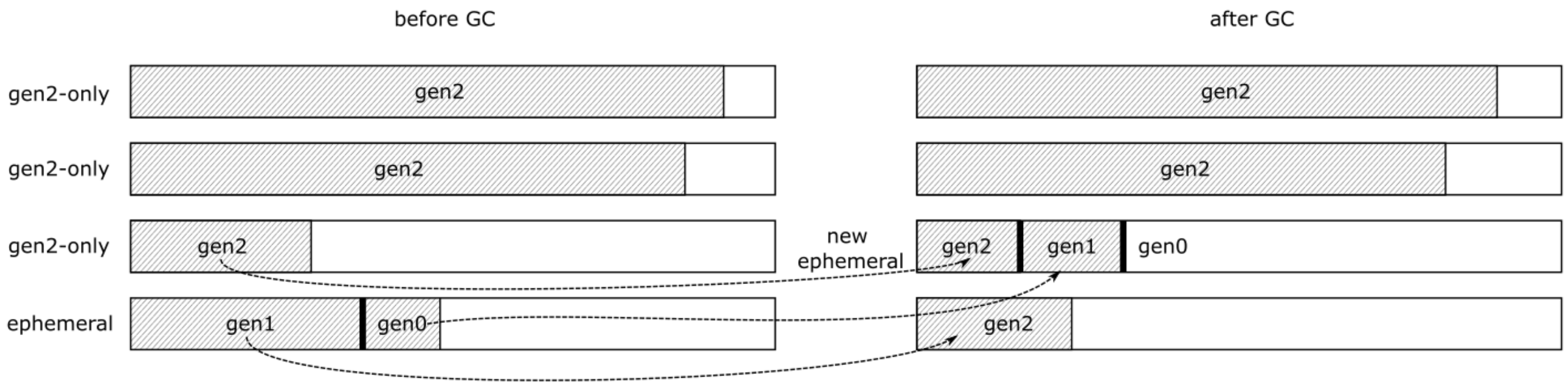
During this process:

- current ephemeral segment is changed into *gen2-only* - all reachable objects from gen 1 & 2 are being compacted there
- new ephemeral segment - all reachable objects from gen 0 are being compacted there (as gen 1 objects)
- LOH segment as usual

Generations - gen2 segments

And the story continues… current ephemeral segment may be filled, we need a new ephemeral segment:

- commit a new segment - as just presented
- reuse a segment from *standby list*
- an already existing gen2-only segment with small gen2 may be reused as a new ephemeral segment 🤯 - The old ephemeral segment will become gen2-only segment



Allocation budget

Allocation budget

- objects are allocated in gen0/LOH or "allocated" (promoted) in gen1/2

Allocation budget

- objects are allocated in gen0/LOH or "allocated" (promoted) in gen1/2
- thus, generations may grow in time and exceed their **allocation budget**

Allocation budget

- objects are allocated in gen0/LOH or "allocated" (promoted) in gen1/2
- thus, generations may grow in time and exceed their **allocation budget**
- GC tracks consumption of allocation budget per generation and uses it to decide on **condemned** generation

Allocation budget

- objects are allocated in gen0/LOH or "allocated" (promoted) in gen1/2
- thus, generations may grow in time and exceed their **allocation budget**
- GC tracks consumption of allocation budget per generation and uses it to decide on **condemned** generation
- thus, typical flow is:

Allocation budget

- objects are allocated in gen0/LOH or "allocated" (promoted) in gen1/2
- thus, generations may grow in time and exceed their **allocation budget**
- GC tracks consumption of allocation budget per generation and uses it to decide on **condemned** generation
- thus, typical flow is:
  - we allocate an object

Allocation budget

- objects are allocated in gen0/LOH or "allocated" (promoted) in gen1/2
- thus, generations may grow in time and exceed their **allocation budget**
- GC tracks consumption of allocation budget per generation and uses it to decide on **condemned** generation
- thus, typical flow is:
  - we allocate an object
  - EE/GC can't find space for a new allocation context (refer to *Episode 07. Allocations*)

Allocation budget

- objects are allocated in gen0/LOH or "allocated" (promoted) in gen1/2
- thus, generations may grow in time and exceed their **allocation budget**
- GC tracks consumption of allocation budget per generation and uses it to decide on **condemned** generation
- thus, typical flow is:
  - we allocate an object
  - EE/GC can't find space for a new allocation context (refer to *Episode 07. Allocations*)
  - the GC is triggered - initialy for gen0

Allocation budget

- objects are allocated in gen0/LOH or "allocated" (promoted) in gen1/2
- thus, generations may grow in time and exceed their **allocation budget**
- GC tracks consumption of allocation budget per generation and uses it to decide on **condemned** generation
- thus, typical flow is:
  - we allocate an object
  - EE/GC can't find space for a new allocation context (refer to *Episode 07. Allocations*)
  - the GC is triggered - initialy for gen0
  - the GC selects the condemned generation - and the oldest generation with its allocation budget exceeded is one of the main reasons. Running out of ephemeral segment may be another, ...

Allocation budget - **total size** the GC would like to allow to be spent on allocations in a particular generation:

Allocation budget - **total size** the GC would like to allow to be spent on allocations in a particular generation:

- changed dynamically on each GC that collects that generation

Allocation budget - **total size** the GC would like to allow to be spent on allocations in a particular generation:

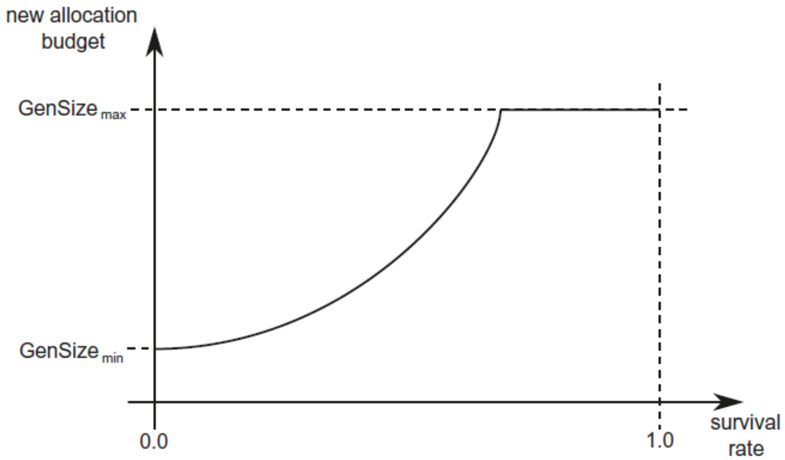
- changed dynamically on each GC that collects that generation
- lies in between given *minimum* and *maximum* (*)

Allocation budget - **total size** the GC would like to allow to be spent on allocations in a particular generation:

- changed dynamically on each GC that collects that generation
- lies in between given *minimum* and *maximum* (*)
- depends mostly on *the survival rate* (ratio of the size of objects that survived):
    - high survival rate - higher allocation budget as we don't want to promote prematurely (count on "better" ratio of dead to live objects next time)
    - small survival rate - smaller allocation budget as opposite to above (and we want to keep generations small)

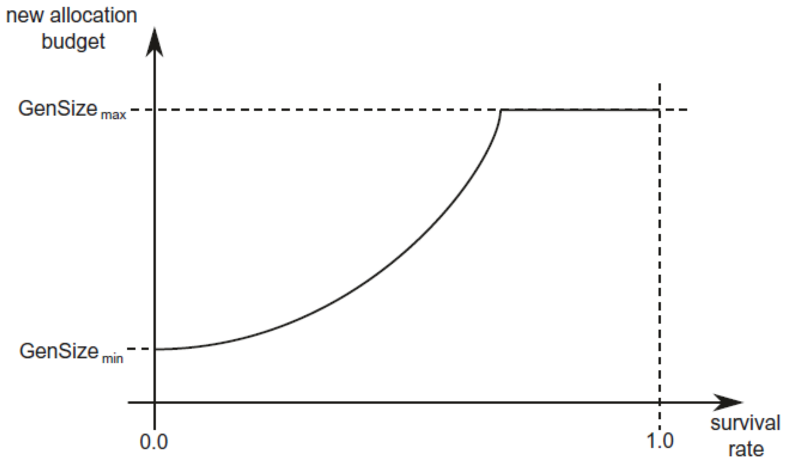
Allocation budget - **total size** the GC would like to allow to be spent on allocations in a particular generation:

- changed dynamically on each GC that collects that generation
- lies in between given *minimum* and *maximum* (*)
- depends mostly on *the survival rate* (ratio of the size of objects that survived):
  - high survival rate - higher allocation budget as we don't want to promote prematurely (count on "better" ratio of dead to live objects next time)
  - small survival rate - smaller allocation budget as opposite to above (and we want to keep generations small)
- the younger generation, the more dynamic change to the survival rate (*)



Allocation budget - **total size** the GC would like to allow to be spent on allocations in a particular generation:

- changed dynamically on each GC that collects that generation
- lies in between given *minimum* and *maximum* (*)
- depends mostly on *the survival rate* (ratio of the size of objects that survived):
  - high survival rate - higher allocation budget as we don't want to promote prematurely (count on "better" ratio of dead to live objects next time)
  - small survival rate - smaller allocation budget as opposite to above (and we want to keep generations small)
- the younger generation, the more dynamic change to the survival rate (*)



(*) controlled by per-generation *static data*

Per-generation static data

Table 7-1. Static GC Data - "Balanced" Mode (Assuming 8 MB LLC Cache)

	Min alloc budget	max alloc budget	fragmentation limit	fragmentation burdenlimit	limit	max_ limit	time_ clock	gc_ clock
Gen0	1) 4/15 MB	2) 6-200 MB	40000	0.5	9.0	20.0	1,000 ms	1
Gen1	160 kB	3) at least 6 MB	80000	0.5	2.0	7.0	10,000 ms	10
Gen2	256 kB	SSIZE_T_ MAX	200000	0.25	1.2	1.8	100,000 ms	100
LOH	3MB	SSIZE_T_ MAX	0	0.0	1.25	4.5	0 ms	0

1) related to the CPU cache size. In general, a little smaller in case of Workstation mode (first number) than in Server mode (second number).

2-3) For Workstation GC with Concurrent version - 6 MB. For Server GC and Workstation GC with Non-concurrent version - half of the ephemeral segment size

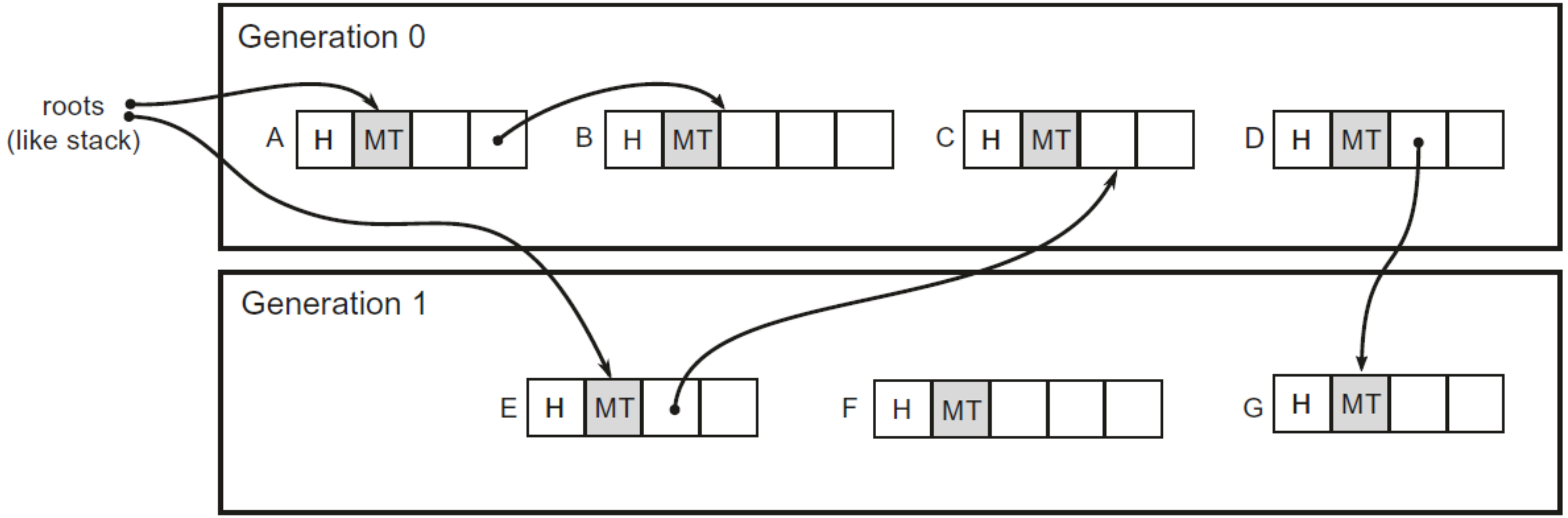
Card tables

Card tables

During GC's *Mark* phase we consider only given condemned and younger generations.

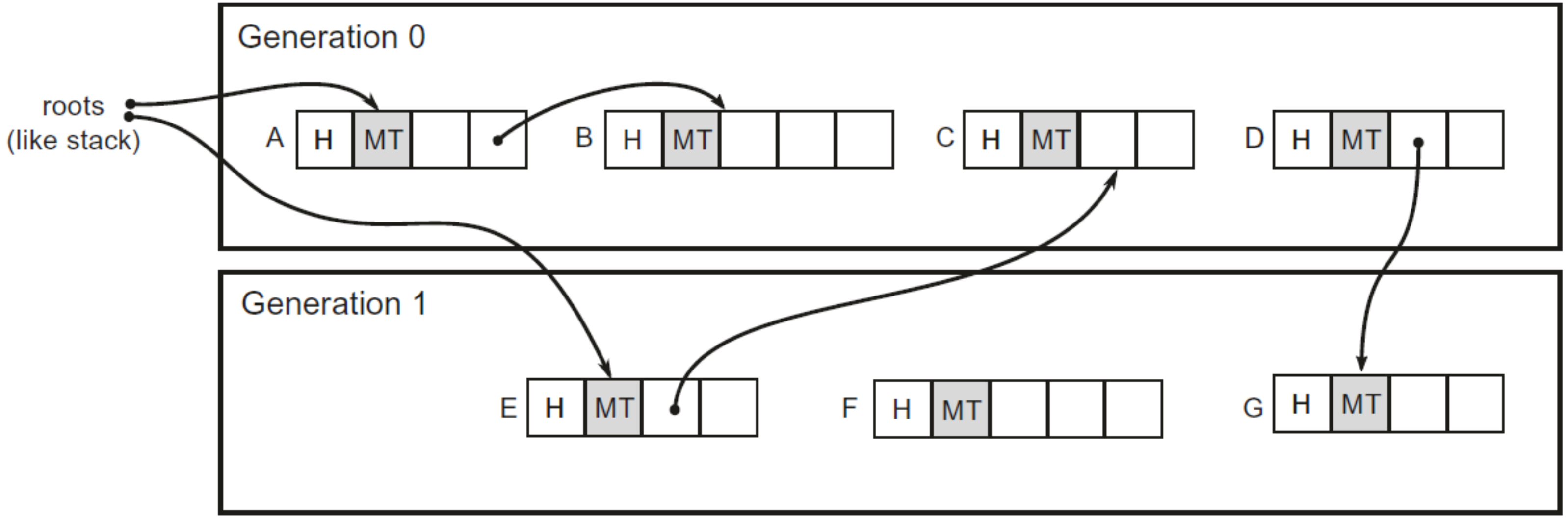
Card tables

During GC's *Mark* phase we consider only given condemned and younger generations. Imagine gen0 GC here:



Card tables

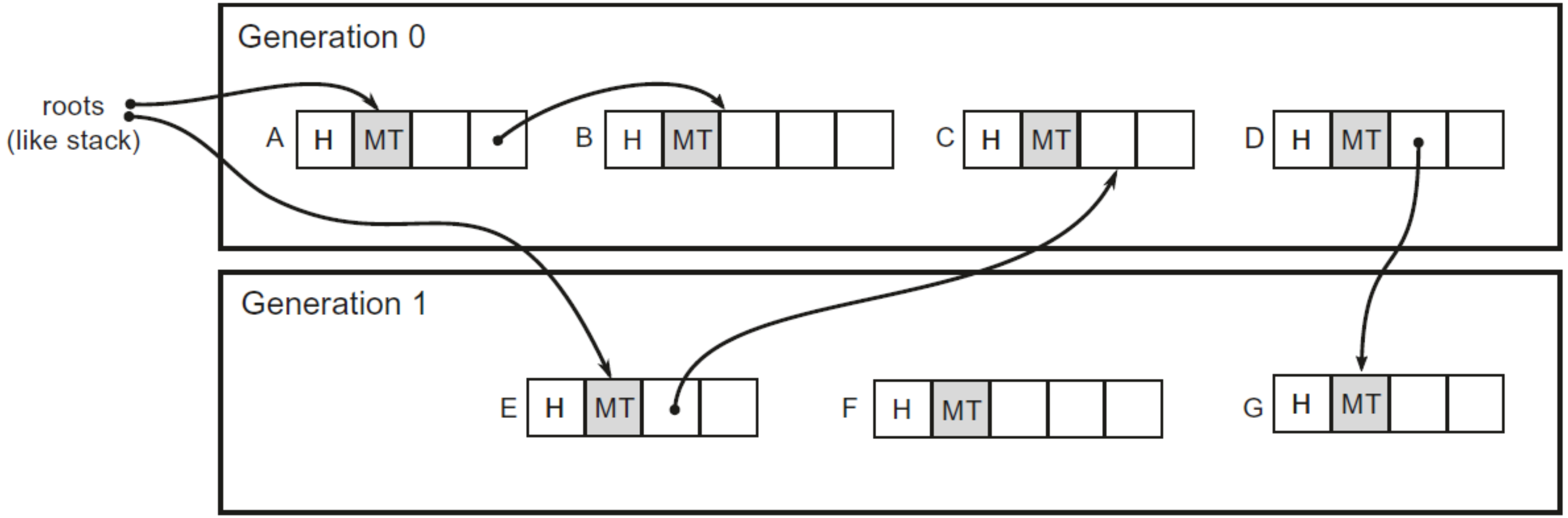
During GC's *Mark* phase we consider only given condemned and younger generations. Imagine gen0 GC here:



So, yes. We would "loose" object **C** 😱

Card tables

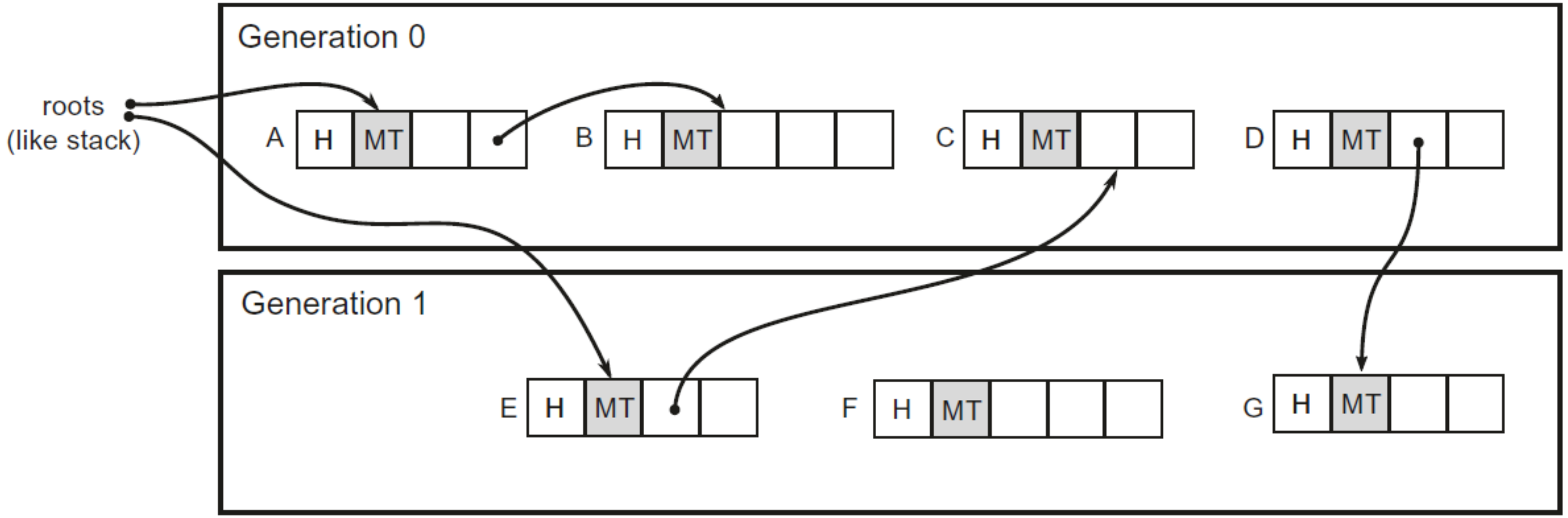
During GC's *Mark* phase we consider only given condemned and younger generations. Imagine gen0 GC here:



So, yes. We would "loose" object `C` 😱 We need to remeber somewhere such *"older-to-younger"* references.

Card tables

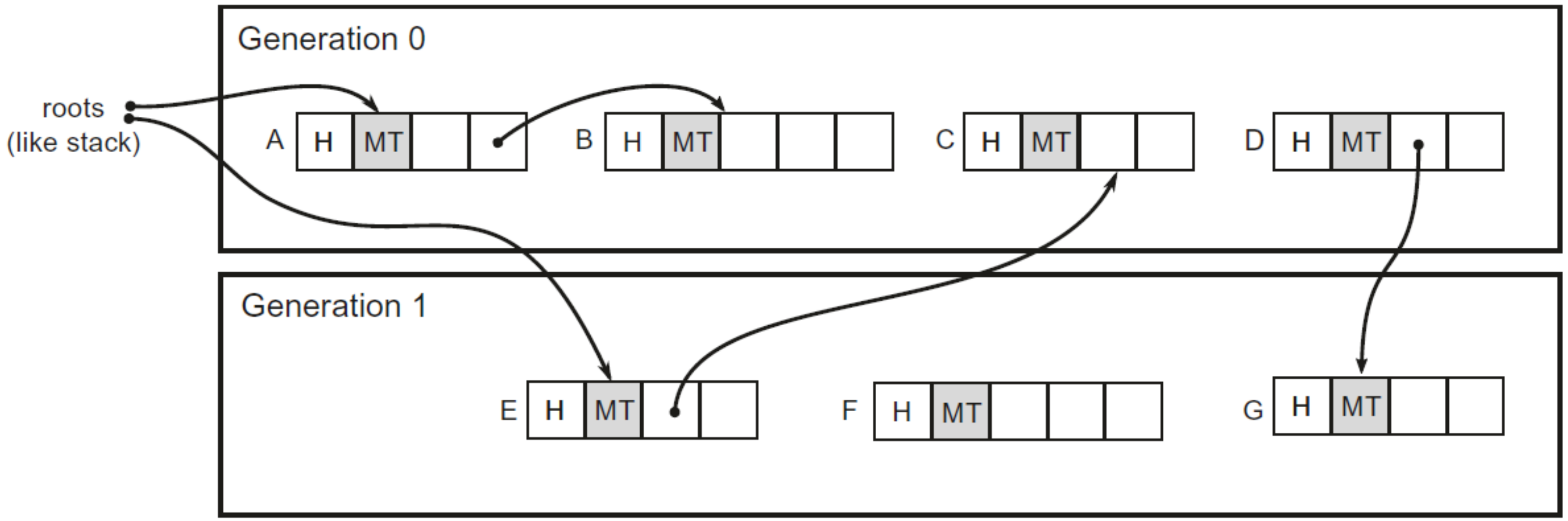
During GC's *Mark* phase we consider only given condemned and younger generations. Imagine gen0 GC here:



So, yes. We would "loose" object **C** 😱 We need to remeber somewhere such *"older-to-younger"* references. In literature, it is called *remembered set*.

Card tables

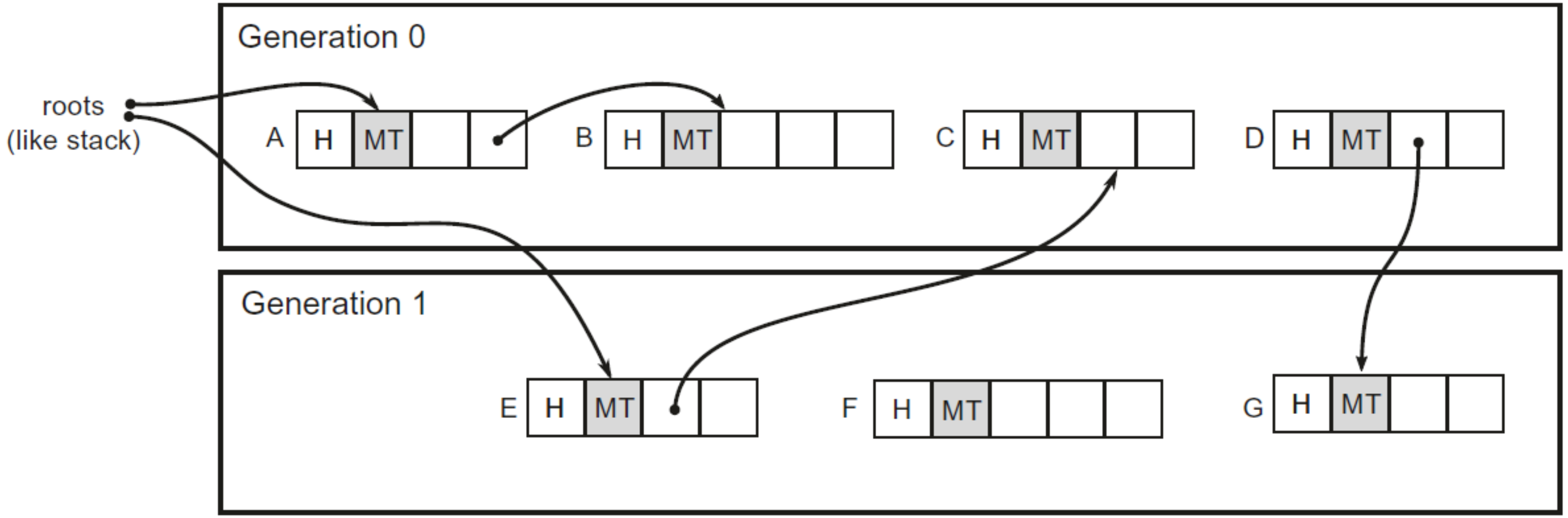
During GC's *Mark* phase we consider only given condemned and younger generations. Imagine gen0 GC here:



So, yes. We would "loose" object **C** 😱 We need to remeber somewhere such *"older-to-younger"* references. In literature, it is called *remembered set*.

BTW, *"younger-to-older"* references are not a problem due to the "always collect given and younger generation"👍

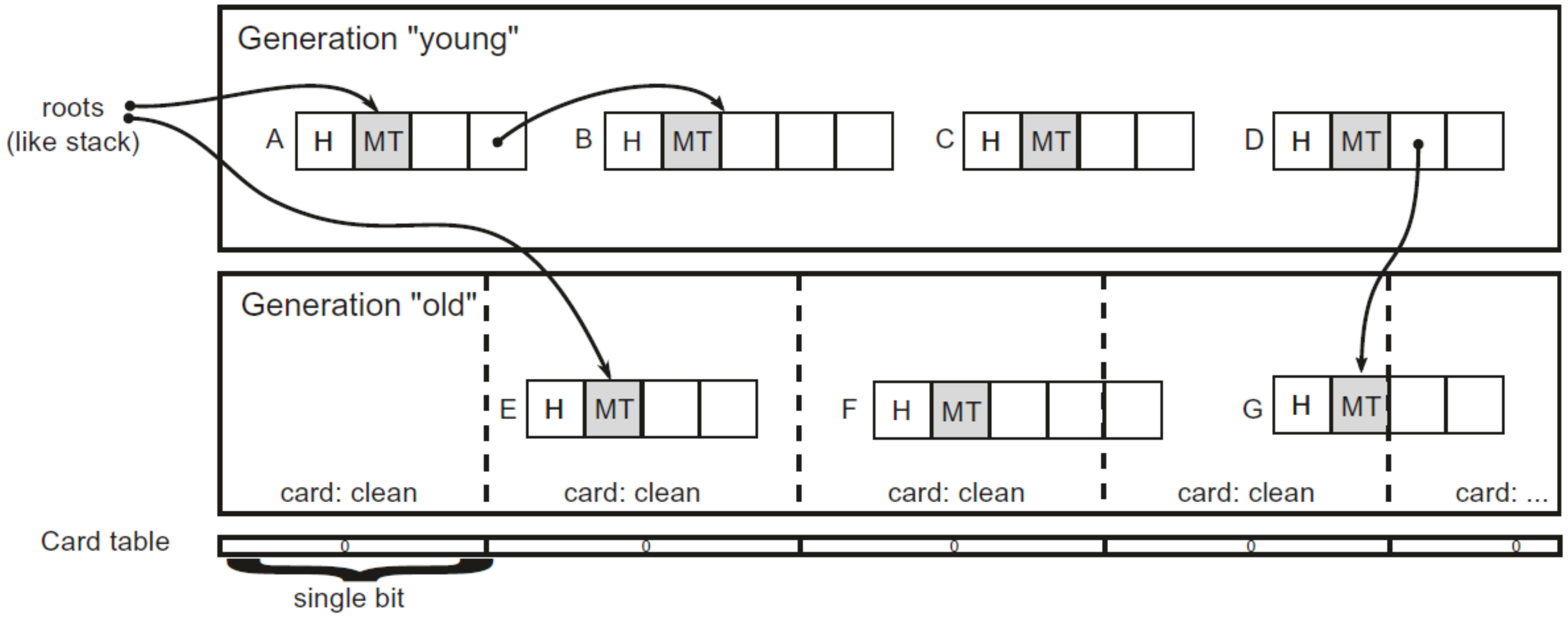
Card tables



We could store every single *"older-to-younger"* reference in some *remembered set* but it would introduce super overhead - we may have many such references changing all the time!

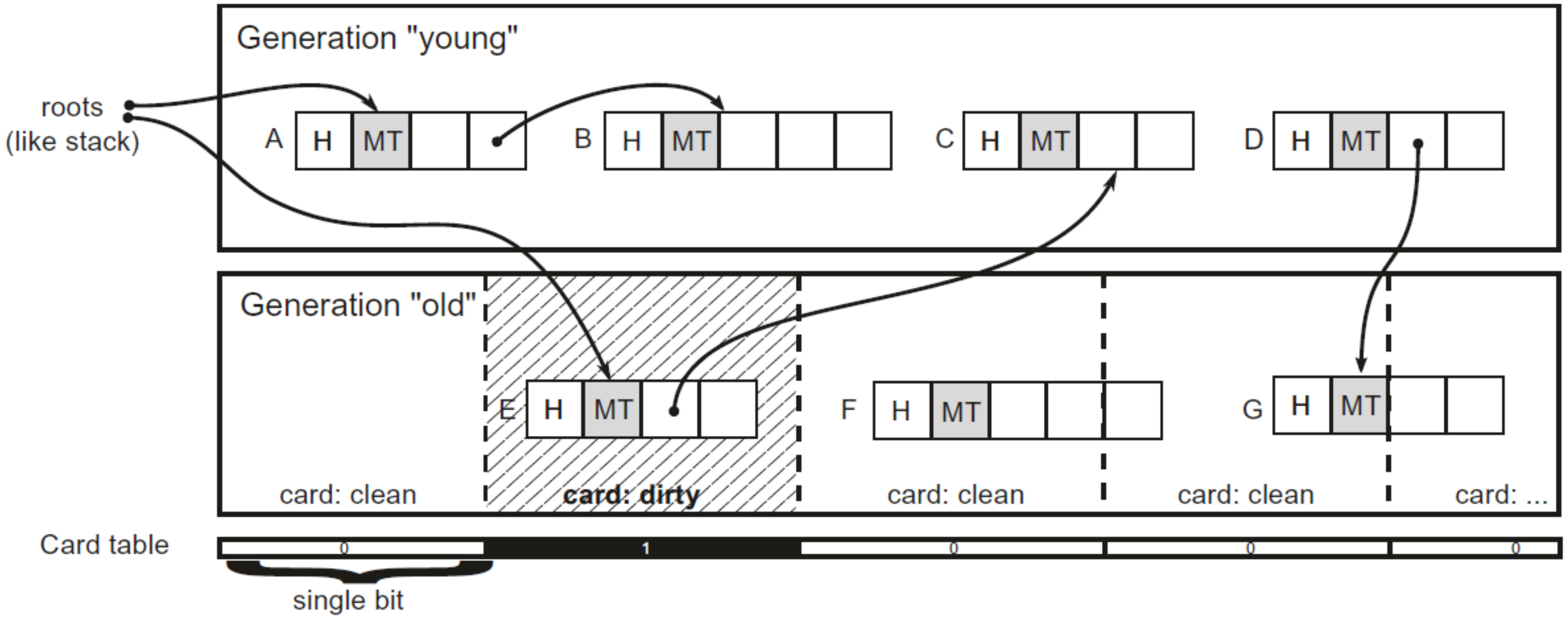
Instead, runtime tracks less granular information about it - covering not single object with "older-to-younger" reference, but for whole memory region.

Card tables



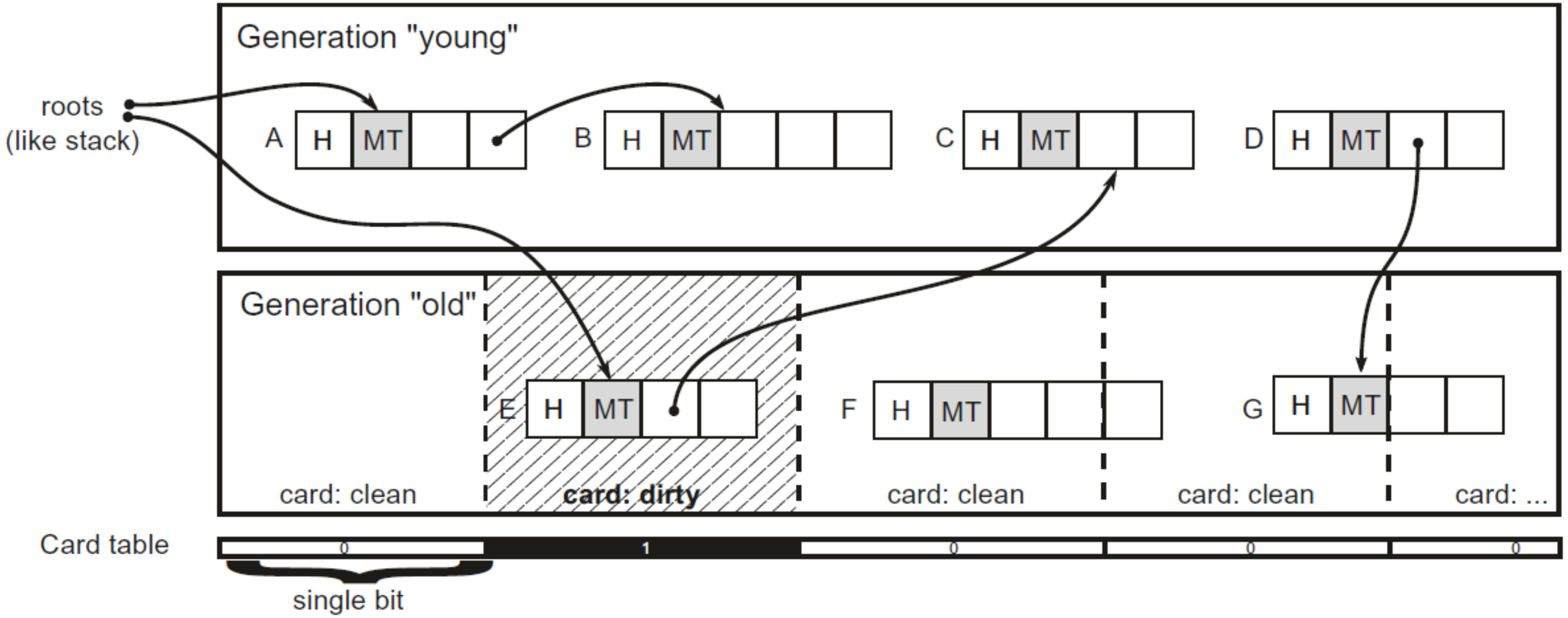
Generation "young"
roots
(like stack)
A
H
MT
B
H
MT
C
H
MT
D
H
MT
Generation "old"
E
H
MT
F
H
MT
G
H
MT
card: clean
card: clean
card: clean
card: clean
card: ...
Card table
0
0
0
0
0
single bit

Card tables



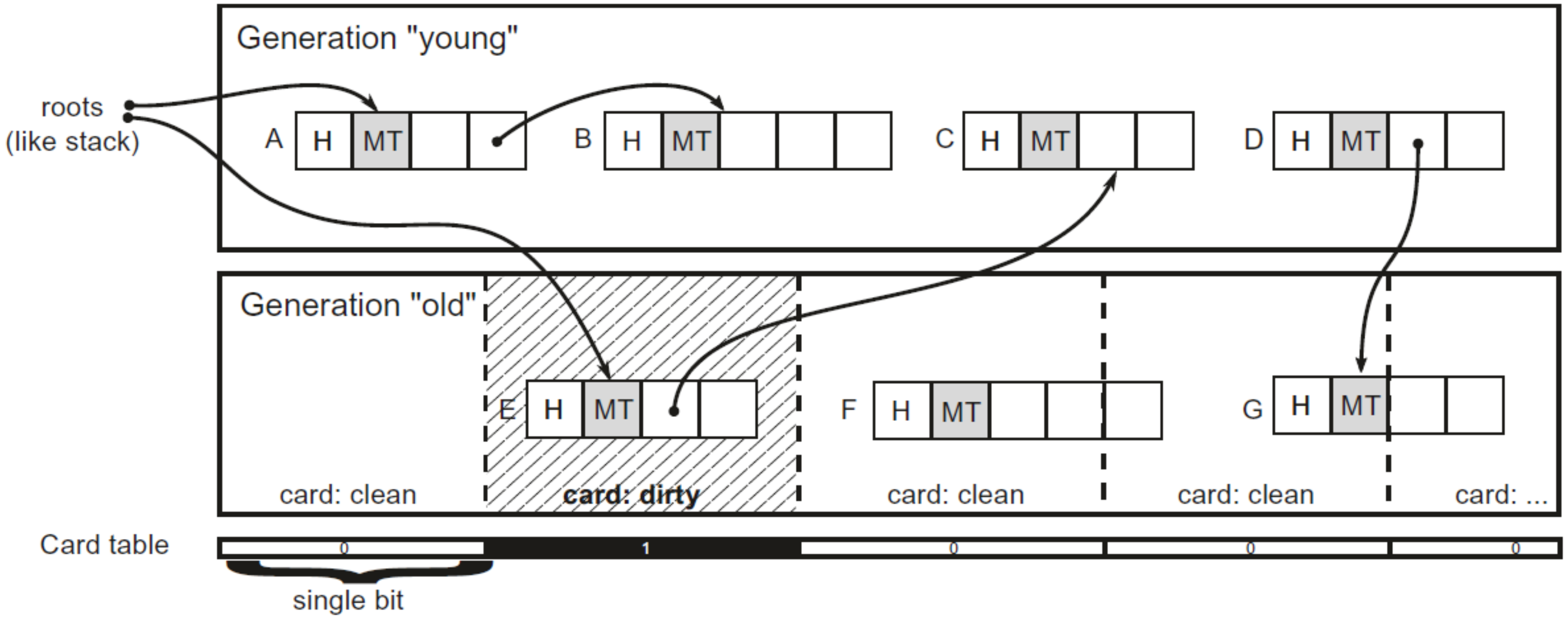
When executing `E.field = C`, write-barrier updates the card.

Card tables



When executing **`E.field = C`**, write-barrier updates the card. Single card covers 256/128 bytes (64/32-bit runtime).

Card tables



When executing `E.field = C`, write-barrier updates the card. Single card covers 256/128 bytes (64/32-bit runtime). But, for performance, write-barrier sets *the whole byte* (`0xFF`), so 2048/1024 bytes regions are treated "dirty".

Card tables

```
LEAF_ENTRY JIT_WriteBarrier_PostGrow64, _TEXT
        ...
        mov     [rcx], rdx
        ...
PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_Lower
        mov     rax, 0F0F0F0F0F0F0F0F0h
        ; Check the lower and upper ephemeral region bounds
        cmp     rdx, rax
        jb      Exit
        ...
PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_Upper
        mov     r8, 0F0F0F0F0F0F0F0F0h
        cmp     rdx, r8
        jae     Exit
        ...
PATCH_LABEL JIT_WriteBarrier_PostGrow64_Patch_Label_CardTable
        mov     rax, 0F0F0F0F0F0F0F0F0h
        ; Touch the card table entry, if not already dirty.
        shr     rcx, 0Bh
        cmp     byte ptr [rcx + rax], 0FFh
        jne     UpdateCardTable
        ...
    UpdateCardTable:
        mov     byte ptr [rcx + rax], 0FFh
        ...
LEAF_END_MARKED JIT_WriteBarrier_PostGrow64, _TEXT
```

Card bundles

On top of that, there is **card bundle** mechanism maintained by `MEM_WRITE_WATCH` or manually to have even less granular, high-level starting point to traverse card tables.

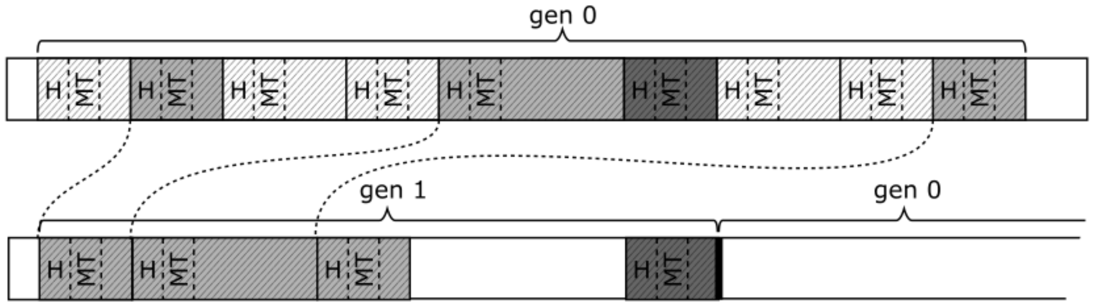
Demotion

Generations - Demotion

- *"if it survives it is **promoted** to older generation"*...
- but *pinning* may destroy this great idea... with fragmentation:

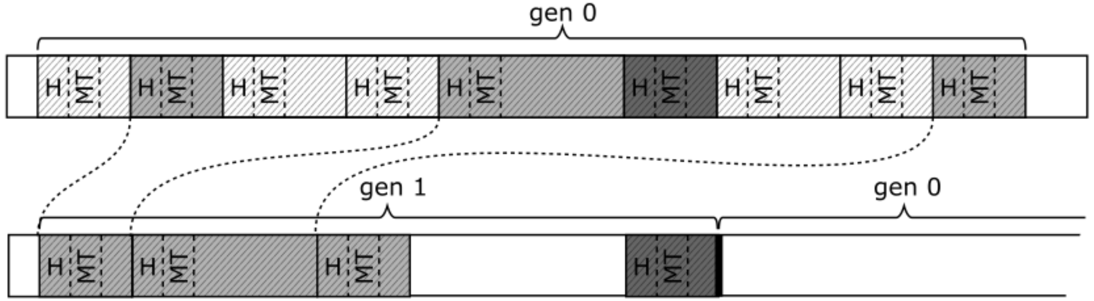
Generations - Demotion

- *"if it survives it is **promoted** to older generation"*...
- but *pinning* may destroy this great idea... with fragmentation:



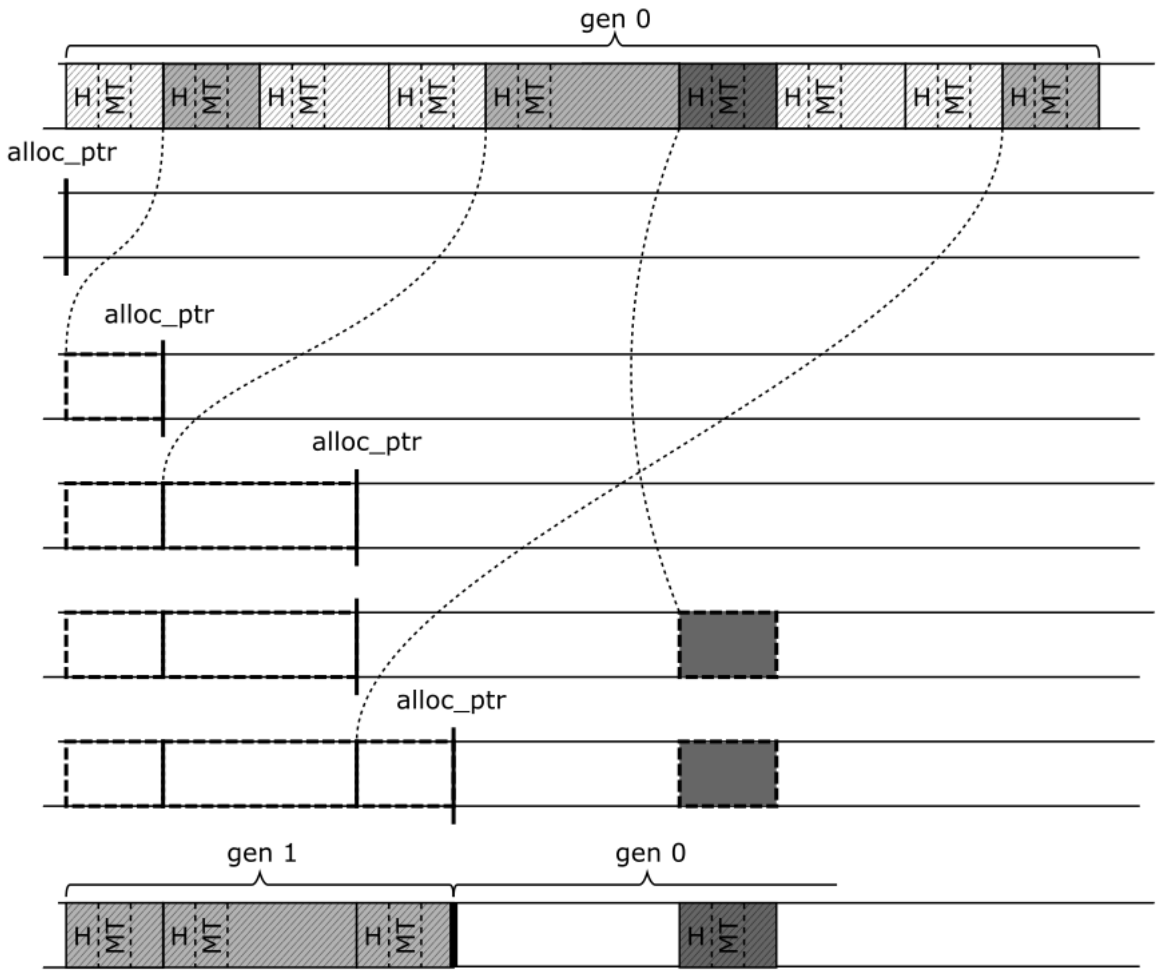
Generations - Demotion

- *"if it survives it is **promoted** to older generation"*...
- but *pinning* may destroy this great idea... with fragmentation:



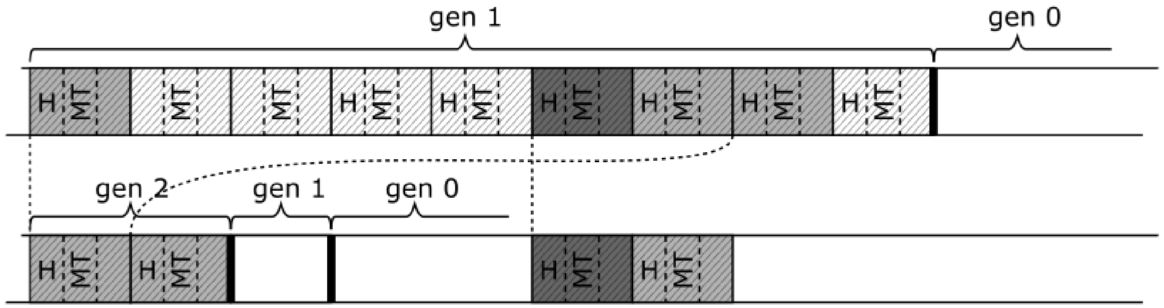
- so, let's introduce *demotion* - as the opposite of promotion

Generations - Demotion

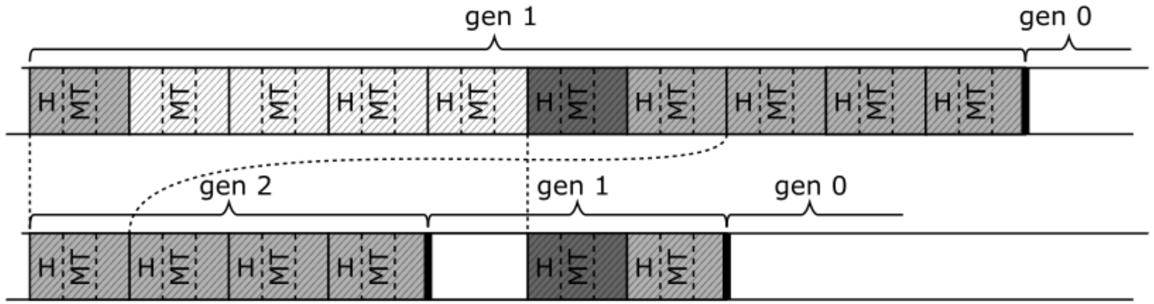


Generations - Demotion

Demotion from gen1 to gen0:

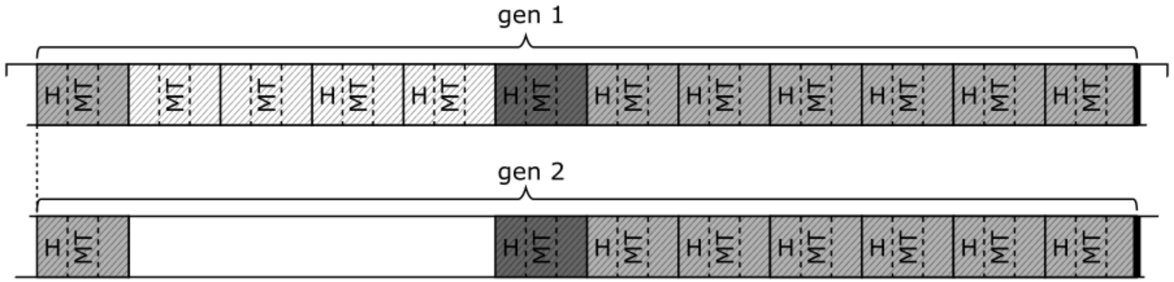


Demotion from gen1 to gen1:



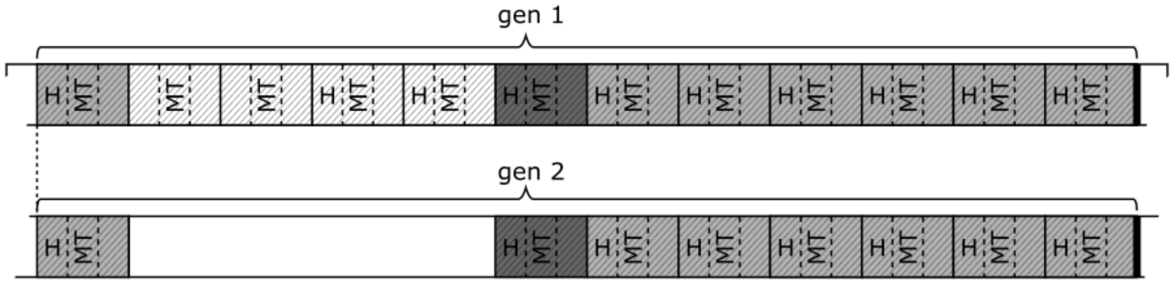
Generations - Demotion

Sometimes a plug will just not fit and demotion does not help:

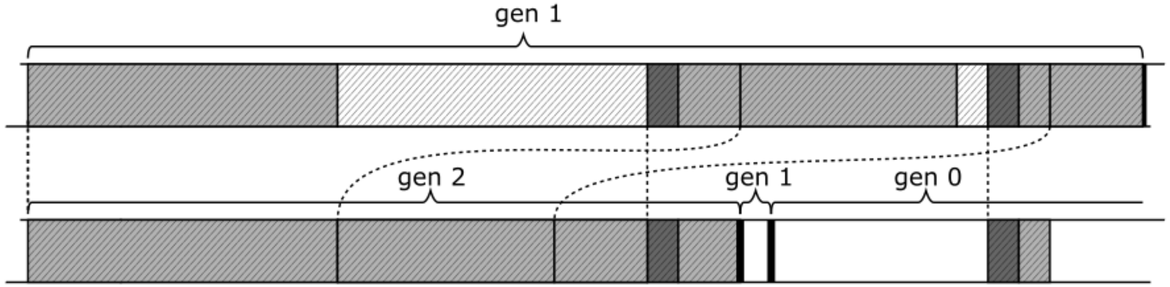


Generations - Demotion

Sometimes a plug will just not fit and demotion does not help:

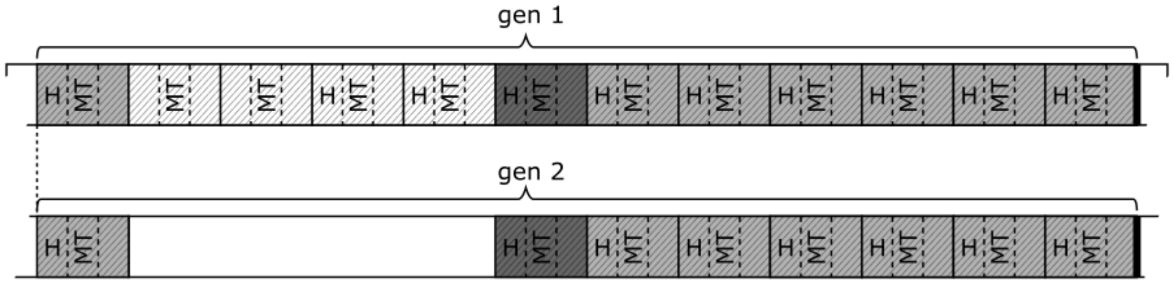


Only some pinned plugs may be demoted:

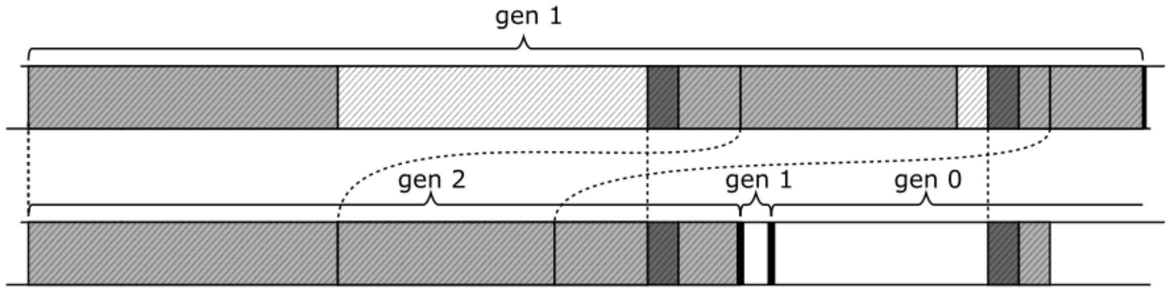


Generations - Demotion

Sometimes a plug will just not fit and demotion does not help:



Only some pinned plugs may be demoted:



Note. To emphasize it - in the current implementation, only pinned plugs may be demoted (which may include single non-pinned object in case of extended pinned plug)