**FONCTIONNEMENT DU GARBAGE COLLECTOR DE BOEHMS:**

Le Garbage collector de Boehms généralement fonctionne en 05 étapes:

**I- ALLOCATION INITIALE DE LA MEMOIRE A L’OS:**

La première étapes dans le processus du Garbage collector de Boehms est l’allocation d’une grande quantité de mémoire au système d’exploitation. GC utilise 03 méthodes d’allocation de la mémoire au tout début:

- mmap: Ici GC utilisel’appel système mmap pour allouer de la mémoire à l’OS.

- sbrk(): Ici GC utilise l’appel système sbrk pour allouer de la mémoire.

- malloc: Ici GC utilise l’appel système malloc pour allouer de la mémoire.

Dans la commande ./configure lors de l’installation de GC lorsqu’on désactive la configuration de mmap c’est l’appel système sbrk qui est utilisé par GC par défaut, lorsqu’on active la configuration mmap à l’aide du paramètre –enable-mmap ca sera l’appel système mmap qui sera utilisé.

Pour le moment je ne peux pas encore te dire quand et dans quelle condition GC utilise l’appel système malloc car je n’ai aucune idée la dessus.

La portion de code utilisée pour faire de l’allocation de mémoire se trouve dans le fichier **os\_dep.c**. La première partie de fichier avant l’implémentation des différentes méthodes de tracking des dirty pages toute la première partie de ce fichier implémente des fonctions qui permettent de faire de l’allocation de mémoire en utilisant les 03 méthodes ci-dessus.

Les fonctions qui ont attiré mon attention dans cette partie sont:

**ptr\_t GC\_unix\_get\_mem(size\_t bytes),**

**STATIC ptr\_t GC\_unix\_mmap\_get\_mem(size\_t bytes),**

La première fonction elle appelle simplement la deuxième tandis que la seconde elle implémente l’allocation de la mémoire en utilisant mmap uniquement.

**ptr\_t GC\_unix\_sbrk\_get\_mem(size\_t bytes) :** Cette fonction implément l’allocation de la mémoire en utilisant sbrk

**ptr\_t GC\_unix\_get\_mem(size\_t bytes) :** Cette fonction bien qu’elle aye le même nom que la première ici GC tente de faire l’allocation de la mémoire en combinant à la fois mmap et sbrk (je ne sais pas trop pourquoi).

Voila en gros ce que je peux dire pour la première étape.

**II-** **AUGMENTATION DE LA TAILLE DU HEAP ET ALLOCATION DES OBJETS ROOTS AU DEBUT:**

Une fois la mémoire allouée GC commence par definir une taille initiale au heap et commence à faire les premières allocations des objets.

La fonction qui implémente cela se trouve dans le fichier **alloc.c** et s’appelle **GC\_INNER ptr\_t GC\_allocobj(size\_t gran, int kind).** (c’est la dernière fonction dans le fichier alloc.c)

**III- TRACKING DES DIRTY PAGES:**

Avant de commencer cette partie je tiens à preciser que le fonctionnement de GC est cyclique et ont distingue deux types de cycle: les cycle globaux appelé de façon technique **full\_collection** et les cycles partiels appelés de façon technique **partial\_collection.**

Une fois que la grande mémoire a été alloué, que la taille du heap a été défini et que tous les objets roots ont été alloué GC initie le premier cycle global ou **full\_collection**.

Un **Full\_collection** consiste en plusieurs étapes:

- On arrête l’application cliente (ici l’application de phoenix) le terme communément utilisé est STOP THE WORLD

- On traque les dirty pages

- On démarre l’application cliente (START THE WORLD)

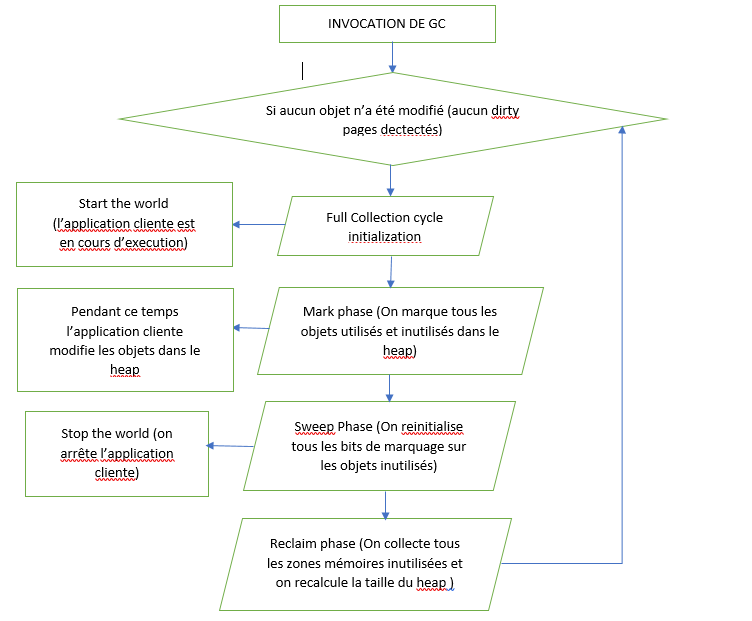
- Pendant que l’application cliente est en court de fonctionnement ont commence le marquage des objets utilisés et inutilisés

- Après un certain intervalle de temps ont arrête l’application cliente

- On fait un balayage (sweep phase) ou ont reinitialise tout les bits de marquage des objets inutilisés

- On recalcule la taille du heap (reclaim phase)

- On démarre l’application cliente



Un **Partial\_collection** consiste aussi en plusieurs étapes:

- On arrête l’application cliente

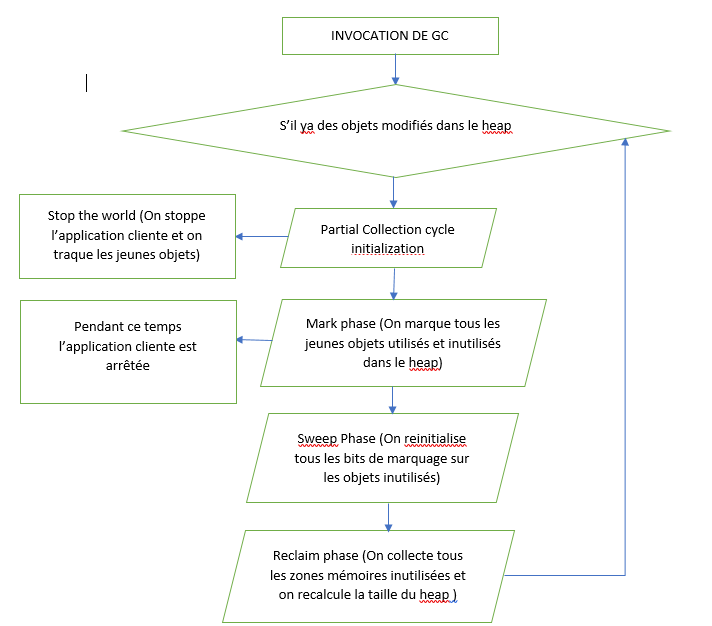
- On tracque les dirty pages des jeunes objets uniquement

- S’il y a les objets modifiés dans le heap

- On initiallise la phase de marquage des jeunes objets (mark phase)

- On reinitialise tous les bits de marquage sur les objets utilisés et inutilisés (sweep phase)

- On recalcule la taille du heap (reclaim phase)



La différence entre un Full\_collection et un Partial\_collection est que dans le Full\_collection GC traque à la fois les jeunes et les vieux objets tandis que dans le Partial\_collection GC ne traque que les jeunes objets uniquement. L’alternance (c’est à dire quand GC alterne entre un Full\_collection et un Partial\_collection) entre ces deux cycles ne se fait que lorsque la taille total des objets utilisés en mémoires devient supérieure à la taille du heap. Je veux dire quoi durant son fonctionnement GC initie un cycle full\_collection lorsque la taille total des objets utilisés en mémoire devient supérieure à la taille du heap. Dans ce cas GC initialise un full\_colection pour traquer aussi les vieux objets pour savoir lesquels ne sont plus aussi utilisés.

La premère fonction qui est exécutée par GC une fois que l’allocation de la mémoire initiale a été faite et l’allocation des objets roots s’appelle **GC\_INNER void GC\_collect\_a\_little\_inner(int n)** qui est définie dans le fichier **alloc.c** et est appelée par la fonction **GC\_INNER ptr\_t GC\_allocobj(size\_t gran, int kind).**

C’est la fonction **GC\_collect\_a\_little\_inner** qui appelle la fonction **STATIC void GC\_maybe\_gc(void)** définie dans le fichier **alloc.c**. Cette fonction par la suite appelle la fonction **STATIC GC\_bool GC\_stopped\_mark(GC\_stop\_func stop\_func)** cette fonction est définie dans le fichier **alloc.c**. c’est cette fonction qui initialise soit le **Full\_collection** soit le **Partial\_collection.**

Dans la définition de cette fonction GC appelle une autre fonction appelée **GC\_INNER void GC\_initiate\_gc(void)** définie dans le fichier **mark.c**.

C’est la fonction **GC\_initiate\_gc** qui lance le tracking des dirty pages puis ensuite initialise la valeur de l’etat du marqueur appelé **GC\_mark\_state.** Les différentes valeurs prisent par cette variable sont définies à la fin du fichier d’entête **gc\_pmark.h**

Pour savoir dans quel cycle se trouve GC, j’utilise les lignes de code suivantes dans la fonction **GC\_stopped\_mark**:

**if(GC\_mark\_state == MS\_INVALID){**

**GC\_log\_printf("\n\*\*\*\*\*\*\*\*\*\*INITIALISATION D'UN CYCLE GLOBAL\*\*\*\*\*\*\*\*\*\*\n\n\n");**

**}else if(GC\_mark\_state != MS\_INVALID){**

**GC\_log\_printf("\n\*\*\*\*\*\*\*\*\*\*INITIALISATION D'UN CYCLE PARTIEL\*\*\*\*\*\*\*\*\*\*\n\n\n");**

**}**

Avant de terminer cette partie je tiens à preciser que l’implémentation des différentes méthodes de tracking des dirty pages se trouve dans le fichier **os\_dep.c** après la description de toutes les méthodes de tracking.

Toutefois pour activer une méthode de tracking il faut aller dans le code du programme de test de phoenix et ajouter la directive:

**#define NOM\_DE\_LA\_METHODE**

par exemple pour le cas que je faisais au début j’utilisais SOFT\_VDB donc je faisais:

**#define SOFT\_VDB**

**IV- MARQUAGE DES OBJETS:**

Sur la capture d’écran suivante on constate que:

Une fois que GC a appelé la fonction **GC\_initiate\_gc** pour traquer les dirty pages il ouvre une boucle infinie puis fait appel à la fonction **GC\_mark\_some().** Cette fonction est défine dans le fichier **mark.c** et permet de marquer tout les objets utilisés et inutilisés en mémoire.

Dans cette fonction, en fonction de l’état du marqueur **GC\_mark\_state** GC collecte tous les objets utilisés et inutilisés en mémoire.



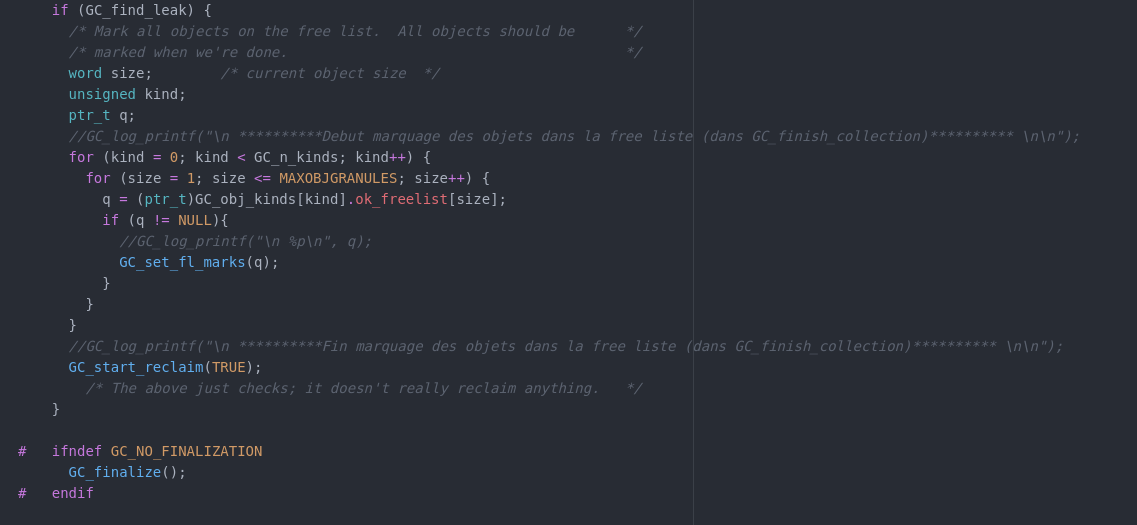
**V- ETAPE DE BALAYAGE ET DE RECLAMATION:**

Une fois que la phase de marquage des objets est terminée GC dans la fonction GC\_maybe\_gc fait appel à la fonction **STATIC void GC\_finish\_collection(void)** qui est définie dans le fichier **alloc.c**.

Cette fonction est utilisée pour marquer tous les objets qui seront supprimés:



Puis ont supprime tous les objets inutilisés en mémoire à l’aide de la méthode **GC\_finalize()**:



Ensuite ont supprime les bits de marquage des objets inutilisés qui sont dans la freelist:



puis ont recalcule la taille du heap à l’aide de la méthode **GC\_start\_reclaim(FALSE)**  et ont reinitialise les variables pour le prochain cycle:



Pour faire simple Voici la séquence des appels des fonctions dans GC:

- On commence avec **GC\_INNER ptr\_t GC\_allocobj(size\_t gran, int kind)** pour définir la taille du heap.

- Ensuite **GC\_allocobj** appelle **GC\_INNER void GC\_collect\_a\_little\_inner(int n)**

- Ensuite **GC\_collect\_a\_little\_inner** appelle **GC\_maybe\_gc**

**-** Puis **GC\_maybe\_gc** appelle **GC\_stopped\_mark**

- Et enfin **GC\_maybe\_gc** appelle **GC\_finish\_collection**

**Voila en gros la séquence des appels dans GC.**

**NB:** Tout ce que je viens de t’expliquer n’est valable uniquement quand le mode incrémental est activé, lorsque celui-ci est désactivé je n’ai aucune idée du comportement de GC puisque dépuis c’est en mode incrémental qu’on a travaillé.

Initialisation:

GC\_enable\_incremental() :: misc.c

|

GC\_init() → GC\_dirty\_init() :: os\_dep.c

GC\_read\_dirty |

| GC\_mark\_init() :: mark.c

GC\_soft\_read\_dirty() :: os\_dep.c |

alloc\_mark\_stack()

A l’allocation:

malloc

|

GC\_allocobj() :: alloc.c

|

GC\_collect\_a\_little\_inner()

|

GC\_maybe\_gc

|

GC\_stopped\_mark

|

GC\_initiate\_gc :: mark.c

|

GC\_read\_dirty :: os\_dep.c → GC\_soft\_read\_dirty

GC\_check\_dirty :: checksums.c |

GC\_soft\_dirty\_open\_files :: to open pagemap and clear\_refs

soft\_set\_grungy\_pages

write(refs…) :: echo “4” > clear\_refs