TD 1 : la pile, le tas et le segment de données

Communication et fonctionnement bas niveau

BUT informatique, première année

Lorsqu'on exécute un programme informatique on le stocke, ainsi que les données qu'il manipule (les constantes et variables du programme), dans la mémoire de l'ordinateur. Ce programme dispose alors d'un espace mémoire qui lui est réservé et qui s'organise en différentes zones (figure 1).

La **pile** est une zone mémoire dans laquelle seront stockées les données utilisées de manière temporaire par les fonctions (leurs paramètres par exemple) uniquement au moment où elles sont nécessaires.

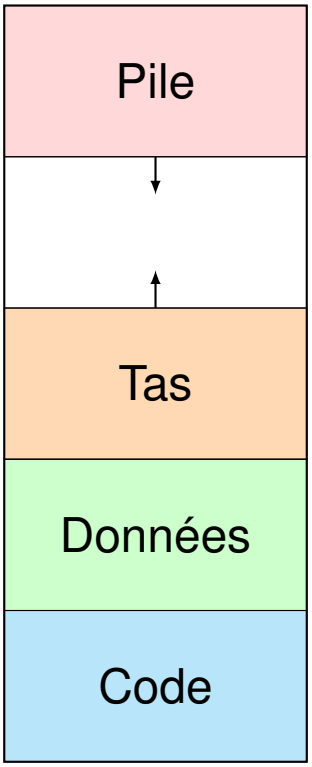

FIGURE 1 – Schéma de l'organisation de la mémoire d'un programme

Le **tas** est une zone mémoire dans laquelle seront stockées les données créées durant l'exécution du programme mais dont la durée de vie n'est pas directement liée à celle d'une fonction (par exemple car elles continuent à être utilisées après la fin de la fonction qui les a créées).

Le segment de **données** est une zone mémoire dans laquelle seront stockées les données créées à la compilation du programme (les constantes par exemple). Cette zone est en fait séparée en plusieurs parties (.data, .bss, etc).

Enfin, le **code** du programme lui même est aussi stocké dans la mémoire.

L’objectif de ce TD est de comprendre comment se fait en général le choix de stocker des données dans la pile, le tas ou le segment de données.

1 La pile

La pile (*stack)* est une zone mémoire qui fonctionne comme un pile au sens algorithmique du terme (i.e. une structure de données LIFO). À chaque appel de fonction, un espace appelé **cadre** (*frame*) de la fonction est créé au sommet de la pile (on utilise donc l'opération *push*). Ce cadre contient notamment les paramètres de la fonction et les variables déclarées dans celle-ci [1]. À chaque retour de fonction, le cadre de cette fonction est retiré de la pile (on utilise donc l'opération *pop*) et on se retrouve donc avec le cadre de la fonction qui l'avait appelée au sommet de la pile.

> En pratique on n'enlève pas vraiment de données de la pile, on se contente de déplacer un pointeur indiquant le sommet de la pile. Ce pointeur est appelé *stack pointer* (SP).

La figure 2 montre à quoi peut ressembler la pile au cours de l'exécution d'un programme. Ici, la fonction main a appelé la fonction foo, qui elle-même a appelé la fonction bar. Au moment où la pile est représentée, la fonction bar vient de se terminer.

1. En fait c'est un peu plus compliqué que ça : les variables locales et, dans certaines architectures, les paramètres peuvent parfois être stockés dans des registres.

```
package main

func main() {
    foo()
}

func foo() {
    bar()
}

func bar() {}
```

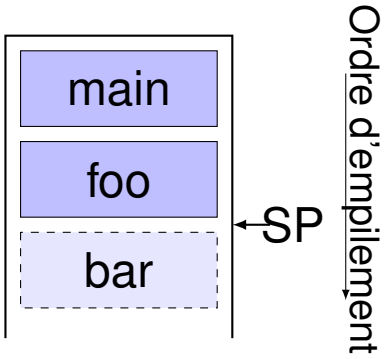

FIGURE 2 – Un programme avec plusieurs fonctions et un aperçu de l'état de la pile juste après le retour de la fonction bar

Exercice 1.

Récupérer, sur MADOC, le programme pile2.go puis dessiner l'état de la pile au début et à la fin de chaque appel de fonction.

2 Le tas

Comme on a pu le comprendre en étudiant le fonctionnement de la pile dans la partie précédente, une donnée qui appartient au cadre d'une fonction (donc qui est mise dans la pile au moment de l'appel à cette fonction) n'existe plus (ou en tout cas n'est plus accessible) une fois que cette fonction a fini de s'exécuter. (On rappelle aussi que – même si ce n'est pas évident puisque ces données existent dans la pile – sans pointeurs [2] les données du cadre d'une fonction ne sont pas non plus accessibles directement par les autres fonctions, même si ce cadre est encore dans la pile.) Ces données qui ne peuvent pas être placées dans la pile seront placées dans le tas (*heap*).

Exercice 2.
Récupérer, sur MADOC, le programme tas.go et le jeu de tests associé tas_test.go, puis consulter ces programmes et expliquer leur fonctionnement.

— À votre avis, quelles variables devraient être placées dans le tas et quelles variables devraient être placées dans la pile ?

2. On reparlera plus en détails des pointeurs dans la suite de ce cours.

Le commentaire `go:noinline` qu'on trouve au dessus de chacune des fonctions est une directive donnée au compilateur (on parle parfois de *pragma*). Elle lui interdit de faire une optimisation particulière qui consiste à remplacer les appels à une fonction par le code de cette fonction (ce qui ferait notamment, dans le cas de ce programme, que tout serait dans le main et donc que l'utilisation du tas ne serait plus jamais nécessaire). À tout moment dans la suite de ce TD vous pouvez supprimer ce commentaire pour voir ce que cela change (mais n'oubliez pas de le remettre ensuite).

On rappelle qu'on peut exécuter les *benchmarks* du fichier tas_test.go par la commande

```
go test -bench=. tas.go tas_test.go
```

En ajoutant l'option -benchmem à cette commande on obtient, en plus des informations sur les temps d'exécution, des informations sur la gestion de la mémoire :

```
go test -bench=. -benchmem tas.go tas_test.go
```

Exercice 3.

Exécuter les *benchmarks* du fichier tas_test.go avec l'option -benchmem. Expliquer les résultats obtenus (pour cela il ne faut pas hésiter à se documenter sur la signification des informations affichées par go test).

- Quel(s) appel(s) de fonction(s) utilise(nt) le tas ?
- Qu'est-ce qui vous semble plus efficace en termes de temps de calcul entre l'utilisation du tas et l'utilisation de la pile ?

On peut obtenir plus d'informations sur les variables exactes qui sont placées dans le tas on peut compiler le programme avec l'option -gcflags -m :

```
go build -gcflags -m tas.go
```

Quand on utilise la commande go build avec -gcflags comme argument, on passe des options de compilation (*flags*) au compilateur Go (gc signifie ici *Go compiler*). En effet, la commande go build est quelque chose de plus général que simplement un compilateur (elle va vérifier les dépendances, gérer les modules, etc) mais elle utilise un compilateur pour produire un exécutable. Ici, on passe l'option -m qui permet d'afficher des informations sur les décisions d'optimisation (comme par exemple le choix de mettre une variable dans le tas ou dans la pile) [3].

Exercice 4.

Utiliser la commande go build -gcflags -m tas.go.

— Quelles sont les variables qui sont placées dans le tas ? Expliquer.

3. On peut obtenir plus d'informations en utilisant go help build qui nous explique que -gcflags passe des arguments à go tool compile. En faisant go help tool on apprend qu'on peut obtenir des explication plus précise en utilisant go doc cmd/compile. Cette dernière commande nous donne la signification exacte de -m et nous indique aussi toutes les autres options possibles. On y trouve aussi des infos sur `go:noinline` dont on a parlé plus tôt. Et bien sûr, une version en ligne de cette page de documentation existe : `https://pkg.go.dev/cmd/compile`

Vous aurez peut-être noté que, quand on utilise go build -gcflags -m tas.go et go test -bench=. -benchmem tas.go tas_test.go ce n'est pas le même code qui est analysé : dans le premier cas c'est la fonction main qui est le point de départ du programme alors que dans le deuxième ce sont les fonction de *benchmark* situées dans le fichier tas_test.go

On a observé des variables qui sont stockées dans le tas et des variables qui sont stockées dans la pile. Cependant, ce qu'on a constaté dans l'exercice précédent n'est pas un règle absolue et, en pratique, les décisions prises par le compilateur Go sont plus complexes que cela. Il peut être difficile de prévoir à l'avance si une variable va être dans le tas ou non (et cela peut varier d'une version à l'autre du compilateur). On peut obtenir un peu plus d'informations dans la FAQ de Go

Exercice 5.

Consulter le point sur l'allocation de la mémoire dans la FAQ de Go : `https://go.dev/doc/faq#stack_or_heap`.

— Est-ce que seules les variables dont on récupère l'adresse dans un programme sont susceptibles d'être stockées dans le tas ?

— Est-ce qu'une variable dont on récupère l'adresse dans un programme est obligatoirement stockée dans le tas ?

Les espaces mémoires qui sont occupés dans le tas doivent être libérés quand ils ne sont plus utilisés pour éviter des fuites de mémoire (*memory leaks*) menant à une occupation trop importante de la mémoire et donc, lorsqu'il n'y a plus d'espace disponible, à des ralentissements importants des programmes, voir à un blocage complet de l'ordinateur. Par rapport à d'autres langages de bas niveau (comme le C) où le programmeur doit lui même réserver et libérer la mémoire, le Go a une différence majeure : il y a un ramasse-miettes (*garbage collector*) qui se charge de temps en temps de libérer la mémoire qui n'est plus utile (les espaces du tas qui ne seront plus jamais utilisés dans le reste du programme). Ceci simplifie la vie au programmeur mais ajoute un surcoût à l'utilisation du tas. Le fonctionnement du ramasse-miettes est assez complexe et nous ne l'aborderons pas dans ce cours.

3 Le segment de données

Le segment de données contient les données qui peuvent être déterminées à la compilation (donc dont on connaît la taille et dont on sait qu'elles existeront dans toute exécution sans avoir a exécuter le programme). On peut voir ces données, ainsi que le code assembleur du programme segment_donnees.go en utilisant la commande suivante :

go tool compile -S -S segment_donnees.go

Attention, il y a bien deux fois le -S, en le mettant une seule fois on ne voit que le code et pas les données.

Exercice 6.
Récuperer le fichier segment_donnees.go sur MADOC et tester la commande go tool compile -S -S segment_donnees.go et repérer la partie données et la partie code dans le résultat produit. Dans le code, SP signifie *stack pointer* et SB signifie *static base pointer*.

- La variable b est-elle allouée statiquement ?
- La variable c est-elle allouée statiquement ?
- Que remarquez-vous sur la constante a ?
- Que remarquez-vous sur les chaînes de caractères utilisées dans le programme ?

Références

1. https://povilasv.me/go-memory-management/ : des explications générales sur la gestion de la mémoire, mises en lien avec le cas particulier de Go.
2. https://medium.com/eureka-engineering/understanding-allocations-in-go-stack-heap-memory-9a2631b5035d : des explications sur les choix faits par Go d'allouer la mémoire dans la pile ou dans le tas.
3. https://go.dev/doc/asm : quelques informations sur le langage assembleur utilisé par le compilateur Go.
4. https://go.dev/doc/faq : la FAQ officielle du langage Go.