Langages : interprétation et compilation

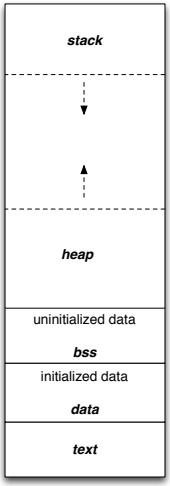
Pablo Rauzy
pr@up8.edu
pablo.rauzy.name/teaching/liec

UNIVERSITÉ PARIS 8 VINCENNES-SAINT-DENIS
UFR MITSIC / L3 informatique

Séance f
La mémoire

La mémoire

- La mémoire d'un programme est organisée en cinq sections :
  - le segment de code,
  - le segment de données initialisées,
  - le segment de données non-initialisées,
  - le tas,
  - la pile.
- (Ceci est au moins vrai pour les programmes type C sous UNIX.)
- Les trois segments sont encodés statiquement dans un exécutable.
- Le tas et la pile sont alloués dynamiquement par le système au lancement du programme.



- Le *segment de code* (aussi appelé *segment texte*) est celui qui contient les instructions exécutables du programme.
- Le plus souvent, il est de taille fixe et accessible en lecture seule pour des questions de sécurité.

Le segment de données initialisées

- Le *segment de données initialisées* (aussi appelé juste *segment de données*) est celui qui contient les variables globales et statiques du programme.
- Il est de taille fixe et accessible en lecture-écriture.
- Il arrive qu'une portion soit en lecture seule (constantes globales).

Le segment de données non-initialisées

- Le *segment de données non-initialisées* (aussi appelé *segment BSS*) est celui qui contient les variables globales et statiques du programme qui n'ont pas de valeur d'initialisation.
- Il est de taille fixe et accessible en lecture-écriture.
- Le plus souvent, le système d'exploitation met la mémoire correspondante à zéro au lancement du programme.

Le tas

- Le *tas* commence au dessus des segments de taille fixe.
- Il est généralement initialement de taille nulle.
- Sa limite est appelée le *program break*.
- C'est ici qu'on alloue dynamiquement de la mémoire (qui sera accessible en lecture-écriture) :
  - l'appel système `brk` change la position du *program break* par celle reçu en argument,
  - l'appel système `sbrk` ajoute la valeur de son argument au *program break*.

- La *pile* grossit généralement vers le bas, en direction du tas.
- Elle est accessible en lecture-écriture et est gérée directement par le programme via le *stack pointer*.
- Elle sert à contenir les variables locales des fonctions.
- L'espace utilisée par une fonction est appelé son *tableau d'activation*.
- À noter :
  - au lancement du programme, `argc`, `argv`, et `env` sont poussées sur la pile par le système.

La commande `size`

- On peut regarder la taille des segments statiques d'un exécutable avec la commande `size`.
- Regardons ensemble quelques exemples...

- Avec SPIM on a :
  - le segment de code (section `.text`),
  - le segment de données (section `.data`),
  - le tas (appel système `sbrk`),
  - la pile (pointeur de pile dans le registre `$sp`).
- On a déjà vu comment utiliser le pointeur de pile et compiler le code.

Le segment de données

- Lors de la compilation d'un programme, il est nécessaire de trouver dans le code source :
  - les variables globales / statiques,
  - certaines constantes qu'on ne peut pas laisser telles quelles dans les instructions (chaîne de caractères par exemple).
- Ces informations sont collectées lors d'une *passe* d'analyse sémantique sur l'arbre de syntaxe abstraite.
- La section `.data` est écrite à partir de ces informations au moment de la production de code.

- La mémoire des variables créées dynamiquement mais qui ne sont pas locales doit être allouée sur le tas.
- Ce qu'on transmets alors, effectivement sur la pile, est un pointeur vers la zone mémoire allouée sur le tas.
- Exemple : une liste retournée par une fonction.
- Rappel :

code	fonction	argument(s)	résultat
`$v0` = 9	sbrk	`$a0` (taille)	`$v0` (adresse)

Exemple

- Partons du code suivant :
  - `[ 42, "coucou" ]`.

- Partons du code suivant :
  - `[ 42, "coucou" ]`.
- Après l'analyse syntaxique on se retrouver avec un AST du genre :
  - `(Pair (Num 42) (Pair (Str "coucou") (Nil)))`.

- Partons du code suivant :
  - `[ 42, "coucou" ]`.
- Après l'analyse syntaxique on se retrouver avec un AST du genre :
  - `(Pair (Num 42) (Pair (Str "coucou") (Nil)))`.
- Après la passe d'analyse sémantique qui s'occupe de ça, on obtient :
  - d'un côté un nouvel AST : `(Pair (Num 42) (Pair (Data 'str_123) (Nil)))`,
  - de l'autre un environnement `#hash(… (str_123 . "coucou") …)`.

- Partons du code suivant :
  - `[ 42, "coucou" ]`.
- Après l'analyse syntaxique on se retrouver avec un AST du genre :
  - `(Pair (Num 42) (Pair (Str "coucou") (Nil)))`.
- Après la passe d'analyse sémantique qui s'occupe de ça, on obtient :
  - d'un côté un nouvel AST : `(Pair (Num 42) (Pair (Data 'str_123) (Nil)))`,
  - de l'autre un environnement `#hash(… (str_123 . "coucou") …)`.
- Lors de la production de code, dans le segment de données on écrira :
  - `str_123: .asciiz "coucou"`.
- Il nous reste à compiler l'AST.

Compilation de l'AST : stratégie

- On a donc l'AST :
  - `(Pair (Num 42) (Pair (Data 'str_123) (Nil)))`.
- Notre but est d'en faire une liste d'instructions pour le segment de code.
- Il y a bien sûr plusieurs façons de s'y prendre, on va en voir une.
- Pour compiler une paire :
  1. on compile un membre,
  2. on compile l'autre membre,
  3. on alloue deux mots mémoires sur le tas,
  4. on copie la valeur du premier membre dans le premier mot, celle du second dans le second.

Compilation de l'AST : stratégie

- On a donc l'AST :
  - `(Pair (Num 42) (Pair (Data 'str_123) (Nil)))`.
- Notre but est d'en faire une liste d'instructions pour le segment de code.
- Il y a bien sûr plusieurs façons de s'y prendre, on va en voir une.
- Pour compiler une paire :
  1. on compile un membre,
  2. on compile l'autre membre,
  3. on alloue deux mots mémoires sur le tas,
  4. on copie la valeur du premier membre dans le premier mot, celle du second dans le second.
- Peut-on utiliser des registres pour garder en mémoire le résultat de la compilation des membres avant de les copier dans la mémoire allouée ? Pourquoi ?

Compilation de l'AST : stratégie

- On a donc l'AST :
  - `(Pair (Num 42) (Pair (Data 'str_123) (Nil)))`.
- Notre but est d'en faire une liste d'instructions pour le segment de code.
- Il y a bien sûr plusieurs façons de s'y prendre, on va en voir une.
- Pour compiler une paire :
  1. on compile un membre,
  2. on compile l'autre membre,
  3. on alloue deux mots mémoires sur le tas,
  4. on copie la valeur du premier membre dans le premier mot, celle du second dans le second.
- Peut-on utiliser des registres pour garder en mémoire le résultat de la compilation des membres avant de les copier dans la mémoire allouée ? Pourquoi ?
- Du coup, qu'est ce qu'on peut utiliser ?

Compilation de l'AST : stratégie

- On a donc l'AST :
  - `(Pair (Num 42) (Pair (Data 'str_123) (Nil)))`.
- Notre but est d'en faire une liste d'instructions pour le segment de code.
- Il y a bien sûr plusieurs façons de s'y prendre, on va en voir une.
- Pour compiler une paire :
  1. on compile un membre,
  2. on compile l'autre membre,
  3. on alloue deux mots mémoires sur le tas,
  4. on copie la valeur du premier membre dans le premier mot, celle du second dans le second.
- Peut-on utiliser des registres pour garder en mémoire le résultat de la compilation des membres avant de les copier dans la mémoire allouée ? Pourquoi ?
- Du coup, qu'est ce qu'on peut utiliser ?
  - Évidemment, on va devoir stocker ces valeurs sur la pile.
  - Pour la même raison qu'on a besoin de la pile pour les variables locales de fonctions récursives.

Compilation de l'AST : code

- Voici la fonction de compilation, réduite au nécessaire pour compiler notre exemple :

```
(define (comp ast env fp-sp)
  (match ast
    ((Nil) (comp (Num 0) env fp-sp))
    ((Num n) (list (Li 'v0 n)))
    ((Data d) (list (La 'v0 (Lbl d))))
    ((Pair a b)
     (append
      (comp a env fp-sp)
      (list (Addi 'sp 'sp -4)
            (Sw 'v0 (Reg 0 'sp))
      (comp b env (- fp-sp 4))
      (list (Lw 't0 (Reg 0 'sp))
            (Addi 'sp 'sp 4)
            (Move 't1 'v0)
            (Li 'a0 8)
            (Li 'v0 9)
            (Syscall)
            (Sw 't0 (Reg 0 'v0))
            (Sw 't1 (Reg 4 'v0)))))))
```

- Regardons ensemble le reste du code...