Interprétation et compilation

Chapitre 6

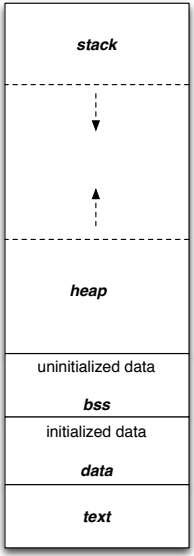
La mémoire

Pablo Rauzy <pr@up8.edu>
pablo.rauzy.name/teaching/ic

La mémoire

Organisation de 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* 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.

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

Compilation

- 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* 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)

- Les “valeurs de gauche” (à gauche du =) dans une assignation ne sont pas des expressions comme les autres.
- Étudions la compilation des instructions C suivantes :
  - `*a = 1312;`
  - `tab[5] = *a;`
- Le compilateur doit explicitement prendre en compte les différents cas de *lvalues*.

Codons !

- En TP, vous ajouterez la gestion des chaînes de caractères à un petit compilateur.
- Tout de suite, ajoutons l'allocation dynamique et le déréférencement.