Exécution des programmes

Compléments sur la liaison - Organisation d'un *shell*

Sacha Krakowiak
Université Joseph Fourier
Projet Sardes (INRIA et IMAG-LSR)

http://sardes.inrialpes.fr/~krakowia

Plan de la présentation

Objectif : synthèse sur les schémas d'exécution des programmes, faisant le lien entre le présent cours et les notions vues ailleurs

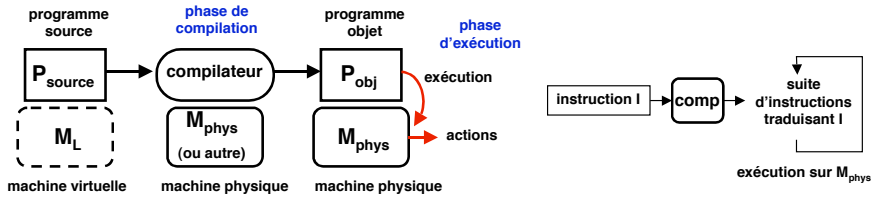
Plan :

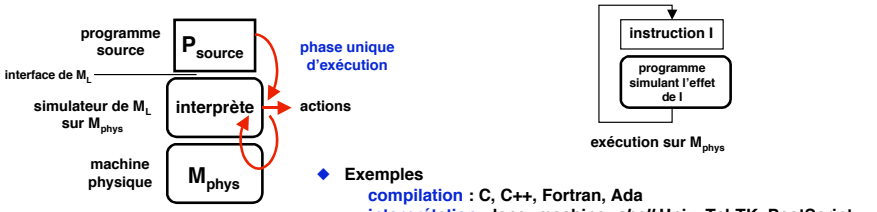
1. Rappel rapide sur les schémas d'interprétation et de compilation
   - Principes, comparaison
   - Fonctionnement d'un interprète
2. Schémas d'exécution d'un programme compilé
   - Rappel rapide sur le cycle de vie d'un programme
   - Compléments
     - Bibliothèques statiques
     - Bibliothèques dynamiques
     - Gestion des dépendances
3. Organisation et fonctionnement d'un *shell*
   - sera poursuivi et développé en TD

Schémas d'exécution d'un programme (rappel)

- **Un programme P_{source} dans un langage (impératif) L décrit une suite d'actions à exécuter par une machine (virtuelle) M_L**
  - Le programme est une suite d'instructions dont chacune spécifie une action (passage d'un état initial à un état final)
  - La machine M_L est capable d'exécuter (interpréter) le programme P_{source}, c'est-à-dire de traduire en actions de M_L la suite d'instructions de P_{source}.
- **Comment faire si on ne dispose pas de la machine M_L, mais d'une machine différente, M_{phys} ?**
  - Deux solutions de base
    1. Traduire le programme P_{source} dans un programme P_{obj} "équivalent" pour la machine M_{phys}, et faire exécuter P_{obj} sur M_{phys} ("équivalent" = qui a le même effet). C'est un schéma de compilation (la traduction de P_{source} en P_{obj} est faite par un compilateur)
    2. Construire (par programme) sur M_{phys} un simulateur de la machine M_L, et faire exécuter le programme original P_{source} sur ce simulateur. C'est un schéma d'interprétation (le simulateur de M_L sur M_{phys} est un interprète)
  - Une solution mixte est possible
    - compiler P_{source} pour une machine intermédiaire M_{int} ; construire un interprète de M_{int} sur M_{phys}

Compilation et interprétation : principe





- **Exemples**
  compilation : C, C++, Fortran, Ada
  interprétation : lang. machine, *shell* Unix, Tcl-TK, PostScript
  l'un ou l'autre : Lisp, Scheme
  schéma mixte : Java (cf plus loin), Smalltalk

Compilation et interprétation : comparaison

Compilation

- ↑ **Efficacité**
  - le code engendré s'exécute directement sur la machine physique
  - ce code peut être optimisé
- ↓ **Mise au point**
  - pas toujours facile de relier une erreur d'exécution au texte source
- ↓ **Cycle de modification - réexécution**
  - toute modification du texte source impose de refaire le cycle complet (compilation, édition de liens, exécution)

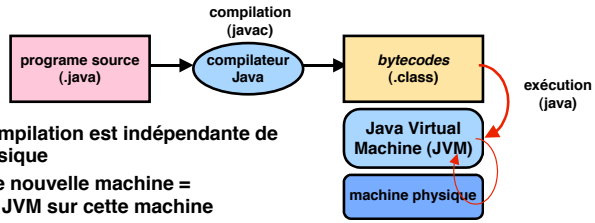
Interprétation

- ↓ **Efficacité**
  - l'interprétation directe est souvent longue (appel de sous-programmes)
  - pas de gain sur les boucles
  - facteur de x10 à x100 …
- ↑ **Mise au point**
  - lien direct entre instruction et exécution
  - possibilités étendues d'observation et trace intégrées
- ↑ **Cycle de modification - réexécution**
  - cycle très court (modifier et réexécuter)

L'augmentation de puissance des processeurs amène un regain des techniques d'interprétation

Schéma mixte d'exécution

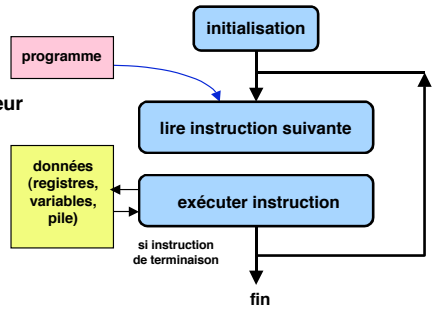
- **Objectifs**
  - Essayer de combiner les avantages des schémas de compilation et d'interprétation
  - Améliorer la portabilité des programmes entre machines, via un langage intermédiaire standard
- **Principe**
  - Définir un langage intermédiaire et une machine virtuelle capable d'interpréter ce langage
  - Écrire un compilateur du langage initial (source) vers le langage intermédiaire
  - Écrire un interprète du langage intermédiaire (c'est-à-dire un simulateur de la machine virtuelle)
- **Exemple : Java**



  - La phase de compilation est indépendante de la machine physique
  - Portage sur une nouvelle machine = réécriture de la JVM sur cette machine

Principe de fonctionnement d'un interprète (1)

- **Définition de la machine virtuelle**
  - Éléments du "pseudo-processeur" (analogie avec processeur physique)
    - pseudo-registres (zones de mémoire réservées)
    - pseudo-instructions (réalisées par une bibliothèque de programmes)
  - Structures d'exécution
    - allocation des variables
    - pile d'exécution
- **Cycle d'interprétation**
  - Analogie avec cycle d'un processeur
  - Pseudo-compteur ordinal



Principe de fonctionnement d'un interprète (2)

- **Raffinement de la "boîte" *exécuter instruction***
  - **Format de l'instruction : instruction ::= <code opération>, <suite de paramètres>**
  - **Opérations**

    Décoder instruction (isoler code op., paramètres)
    selon code op faire
    code$_1$: exécuter programme de code$_1$ (paramètres)
    …
    code$_n$: exécuter programme de code$_n$ (paramètres)
    autre : signaler erreur
  - **Exécution du programme correspondant à un code op.**

    Dépend de l'instruction, mais quelques points communs
    allouer éventuellement place en mémoire pour données
    charger paramètres dans registres ou pile
    exécuter l'opération sur les paramètres
    mettre éventuellement à jour les résultats en mémoire
    déterminer l'instruction suivante à exécuter
  - **Autres opérations possibles**

    Observation, mise au point (exécution pas à pas), trace
    Transformation préalable du programme dans une forme adaptée à l'exécution (postfixé, etc.)

Cycle de vie d'un programme compilé (rappel)

- **Compilation vers programme objet absolu (adresses fixées en mémoire)**

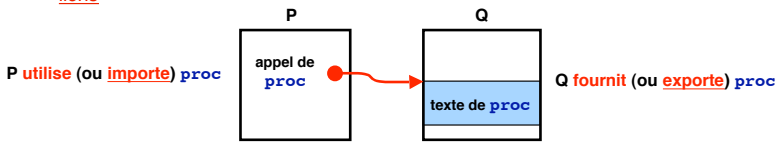
  - contrainte : le programme ne peut pas être déplacé en mémoire (par ex. pour le combiner avec d'autres)
- **Compilation vers programme objet translatable (adresses définies à une translation près)**

  - nécessité de passer par un chargeur pour rendre le programme exécutable (le programme translatable n'est pas exécutable tel quel)
- **Exemples**

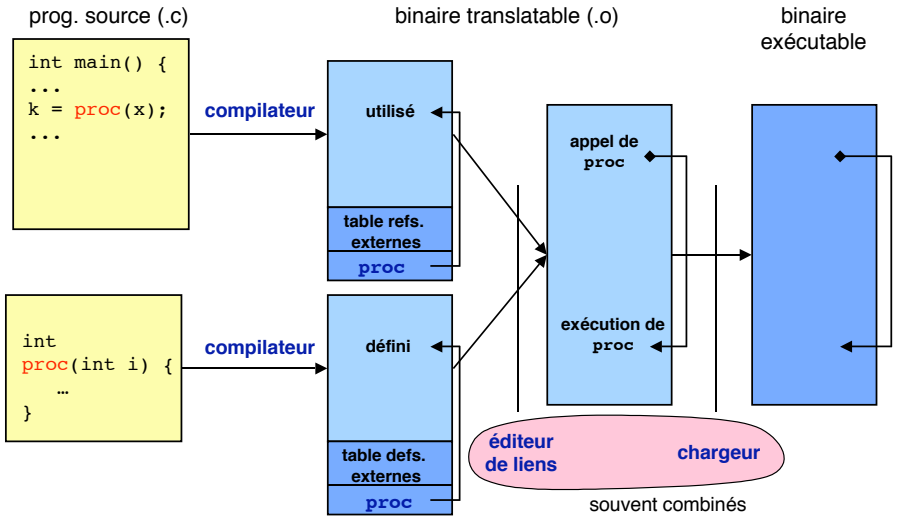
  `gcc -c prog.c` ; produit un programme objet translatable dans le fichier `prog.o`
  `gcc -o prog prog.c` ; produit un programme objet absolu dans le fichier `prog`
  (appelle le compilateur et le chargeur)

Composition de programmes

- **Pourquoi parler de composition de programmes ?**
  - La plupart des applications sont réalisées par assemblage de parties
    - construction modulaire, facilite la conception et l'évolution des programmes
  - La plupart des applications utilisent des fonctions founies par des "bibliothèques"
  - Conséquence : un programme ne s'exécute pratiquement jamais "seul"
- **Le problème de la composition : un exemple de liaison**
  - Soit un programme P faisant appel à une procédure `proc` incluse dans un programme Q construit séparément
  - Question : lors de la compilation de P, quelle adresse va-t-on associer à `proc`?
  - Réponse : il est impossible de le savoir tant qu'on ne connaît pas le programme Q
  - La liaison dans P entre la procédure `proc` et son adresse ne peut être faite que dans une phase postérieure à la compilation, utilisant simultanément P et Q : c'est l'édition de liens



Édition de liens



Exemples

- **La commande *gcc* permet d'appeler à la fois le compilateur, l'éditeur de liens et le chargeur**
  - `gcc prog.c` ; produit (par défaut) le binaire exécutable dans le fichier `a.out`
    `gcc -o prog prog.c` ; produit le binaire exécutable dans le fichier `prog`
    `gcc -c prog.c` ; produit (par défaut) le binaire translatable dans le fichier `prog.o`
  - Supposons qu'un programme soit composé de deux parties `prog1.c` et `prog2.c`. Alors :
    `gcc prog1.c prog2.c` produit le binaire exécutable du programme complet dans `a.out`
    `gcc - c prog1.c prog2.c` produit les deux binaires translatables `prog1.o` et `prog2.o`
    `gcc -o prog prog.o prog1.o` produit le binaire exécutable du programme complet dans `prog`, à partir des binaires translatables `prog1.o` et `prog2.o`
- **Utilisation de bibliothèques**
  - On peut rechercher les références externes dans des collections de programmes appelées **bibliothèques**, ou **archives**, fournies par le système sous forme de binaires translatables pour les utilitaires courants (détails plus loin)

Quelques problèmes de l'édition de liens

```
k = proc(x);
...
```

```
proc
non défini
```

Définition manquante

```
<unix> gcc -o prog prog1.c prog2.c prog3.c
/tmp/cckdgmLh.o(.text+0x19): In function 'main':
undefined reference to 'proc'
collect2: ld returned 1 exit status
<unix>
```

```
k = proc(x);
...
```

```
int proc(int i){
...}
```

```
int proc;
...
```

Définitions multiples (sans déclarations `extern`)

```
<unix> gcc -o prog prog1.c prog2.c prog3.c
<unix>
```

Pas de diagnostic d'erreur...
Quelles sont les règles ?

Définitions multiples en C

On distingue 2 catégories de **symboles externes** (globalement visibles par l'éditeur de liens, donc hors du corps des fonctions)

symboles forts : définition de procédures, variables globales initialisées
symboles faibles : variables globales non initialisées

faible
fort

```
int z;
int proc(int i){
      ...
}
```

fort
faible

```
int y = 5;
double d;
...
```

Règles de l'éditeur de liens pour les définitions multiples :

- deux symboles forts de même nom : interdit ! (erreur détectée)
- un symbole fort et plusieurs faibles de même nom : c'est le symbole fort qui est retenu
- plusieurs symboles faibles de même nom : l'un d'eux est choisi au hasard

Pièges des définitions multiples

```
int z;
int proc(int i){
```

```
int x;
int proc(){
```

interdit (diagnostic) :
2 symboles forts

```
int x;
int p1(int i){
```

```
int x;
int p2(){
```

les deux variables x (faibles) désignent le même objet…

```
int x;
int y;
int p1(int i){
```

```
double x;
int p2(){
```

2 variables x faibles :
si x est modifié dans p2, on peut écraser y... catastrophe !

```
int x=7;
int y=5;
int p1(int i){
```

```
double x;
int p2(){
```

la variable x initialisée est forte :
si x est modifié dans p2, on va écraser y... catastrophe !

Le problème vient de ce que le langage C n'impose pas de **contrôle de types** strict entre modules. Il faut donc utiliser une déclaration **explicite** des références externes

Déclarations des variables externes

Règle 1 : éviter d'utiliser des variables globales ! on peut le plus souvent s'en passer

Règle 2 : si on utilise une variable globale, la déclarer dans UN module (soit `M`), et la déclarer comme `extern` dans tous les autres (avec son type). Ne la modifier QUE dans le module `M` où elle a été déclarée (si on doit la modifier à partir d'autres modules, le faire via des fonctions déclarées dans `M`)

Règle 3 : déclarer explicitement les fonctions externes (avec leur signature) ; on peut ne pas indiquer `extern`, la signature (ou prototype) suffit.

```
extern double y;
int x;
int p1(int i){
      …
}
/* utiliser y */
```

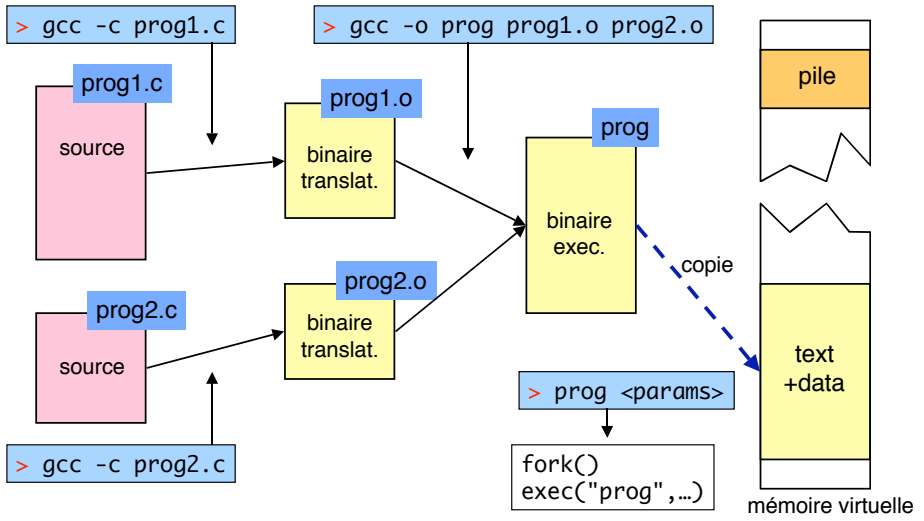
modifier `x` seulement ici

```
extern int x;
double y;
/* utiliser x */
```

modifier `y` seulement ici

```
extern double y;
int p1(int i);
/* utiliser
   y, p1  */
```

Cycle de vie d'un programme : résumé



Bibliothèques statiques (1)

De nombreux programmes utilisent des fonctions d'usage courant, fournies par le système, telles que `printf()`, `strcpy()`, `sin()`, etc.

Comment un programme exécutable accède-t-il à ces fonctions ?
Les fonctions sont regroupées sous forme binaire translatable dans des **bibliothèques**, ou **archives**, spécialisées. Par exemple

`/usr/lib/libc.a`	bibliothèque standard du langage C
`/usr/lib/libm.a`	bibliothèque mathématique

Il suffit de spécifier les bibliothèques utilisées dans la commande d'édition de liens (en fait `libc.a` est automatiquement incluse ; inutile de l'indiquer)

```
gcc -o prog prog.c /usr/lib/libc.a /usr/lib/libm.a
```

Remarque importante : l'éditeur de liens ne choisit dans la bibliothèque que les fonctions **effectivement utilisées**. Par exemple si le programme `prog.c` n'utilise que `print`, `strcpy` et `sin`, seules ces fonctions seront incluses dans `prog` (heureusement : `libc.a` a une taille de 8 Megaoctets environ)

Bibliothèques statiques (2)

Quelques aspects pratiques d'utilisation :

- notation abrégée : `-l<nom>` est un raccourci pour `/usr/lib/libnom.a` par exemple : `gcc -o prog prog.c -lsocket` (inclut `/usr/lib/libsocket.a`)
- chemin de recherche : la variable d'environnement `LIBPATH` indique dans quels répertoires chercher les bibliothèques (analogue à `PATH`). On peut aussi spécifier ces répertoires dans la commande avec l'option `-L`

Il est possible de construire sa propre bibliothèque avec la commande `ar` :

```
ar rcs libprivee.a fonct1.o fonct2.o fonct3.o
```

construit la bibliothèque `libprivee.a` contenant `fonct1.o`, `fonct2.o`, et `fonct3.o`

Difficultés et inconvénients des bibliothèques statiques :

- problèmes de dépendance mutuelle (détails plus loin)
- si plusieurs processus utilisent la même fonction de bibliothèque, celle-ci est incluse dans chaque programme exécutable (donc dupliquée)

Bibliothèques dynamiques (1)

Les **bibliothèques dynamiques** visent à éviter le principal inconvénient des bibliothèques statique : la duplication des fonctions usuelles dans les mémoires virtuelles des processus qui les utilisent.
Exemples : en Unix, *shared objects* (`.so`) ; en Windows, *Dynamically Linkable Libraries* (DLL)

Avec une bibliothèque dynamique,

- une fonction de bibliothèque utilisée par plusieurs processus est **partagée** par tous ces processus (à des adresses virtuelles qui peuvent être différentes) ; elle ne figure qu'une fois dans la mémoire physique
- une fonction de bibliothèque est incluse **dynamiquement** dans la mémoire virtuelle d'un processus qui l'utilise (une zone est prévue à cet effet)

Plus précisément, la liaison peut être faite soit en deux étapes (liaison partielle avant exécution, puis liaison dynamique au moment de l'exécution) soit entièrement de manière dynamique.

Bibliothèques dynamiques (2)

Pour fabriquer une bibliothèque dynamique partagée `libpart.so` contenant `fonct1.o`, `fonct2.o`, et `fonct3.o`

```
gcc -shared -fPIC -o libpart.so fonct1.c fonct2.c fonct3.c
```

partagée(un seul exemplaire)

PIC = *position-independent code*

Le code PIC peut être placé à n'importe quelle adresse. Nécessaire pour pouvoir exécuter le même programme à des adresses différentes

La bibliothèque `libpart.so` s'utilise ensuite de la même manière qu'une bibliothèque statique (si on veut une liaison partielle avant exécution) ou bien à travers des primitives spécialisées (si on veut une liaison entièrement dynamique).

Détails sur ce dernier aspect non fournis ici (cf livres de référence)

Gestion des dépendances dans une commande

Dans une commande utilisant des bibliothèques statiques, l'éditeur de liens essaie de résoudre les références non encore résolues en examinant les fichiers dans l'ordre dans lequel ils apparaissent dans la commande.
Exemple :

```
gcc -o prog prog.c -ltruc -lmachin
```

Supposons que `prog.c` utilise la fonction `f()` fournie par `libmachin.a`, et que `f()` utilise `g()` fournie par `libtruc.a`

Dans ce cas, l'éditeur de liens diagnostique une référence non résolue !

Il aurait fallu écrire :

```
gcc -o prog prog.c -lmachin -ltruc
```

Dans des cas de dépendance circulaire, il faut faire figurer 2 fois le nom d'une bibliothèque.
La règle est que la déclaration d'une fonction apparaisse toujours **après** la première référence à cette fonction.

Gestion des dépendances entre commandes

L'outil `make` permet de gérer automatiquement diverses dépendances :

- Dépendance de fichiers entre eux (exemple : si on modifie un fichier source, il faut le recompiler et refaire l'édition de liens pour mettre à jour le fichier exécutable
- Dépendance entre commandes (via les fichiers produits) : une commande qui utilise un fichier dépend de la commande qui produit ce fichier

Les dépendances sont indiquées dans un fichier `Makefile`. Exemple simple :

```
prog.o: prog.c truc.h
	gcc -c prog.c
monprog: prog.o truc.c
	gcc -o monprog prog.o truc.c
all: monprog fich1
	monprog fich1 | gv
```

`toto.o` dépend de `prog1.c`, `prog2.c`, `truc.h` si l'un d'eux change, on réexécute la commande

`monprog` dépend de `toto.o`, `truc.c` si `toto.o` n'existe pas ou est périmé, on applique la règle précédente

Une modification de `truc.h` amène à réexécuter les 3 commandes

Voir document technique n°1 (placard) et référence en ligne citée dans ce document

Fonctionnement d'un *shell*

Un ***shell*** est un programme qui réalise une interface entre le système d'exploitation et un utilisateur interactif. Il fonctionne comme un interprète d'un langage de commande, et utilise lui-même l'interface des appels systèmes

Il existe plusieurs *shells* pour Unix (`sh`, `bash`, `csh`, `tcsh`, …)

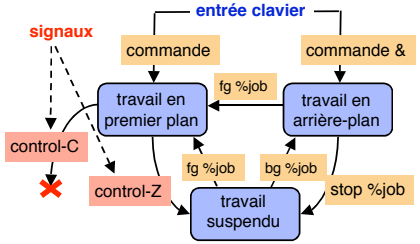
Fonctions principales d'un *shell*

- Interpréter les commandes introduites par les usagers. Ces commandes donnent accès aux principales fonctions du système d'exploitation (création et exécution de processus, accès aux fichiers, entrées-sorties, utilisation des outils tels qu' éditeurs, compilateurs, éditeurs de liens, outils de mesure, etc.).
- Permettre d'exécuter des travaux en mode interactif ou en travail de fond
- Fournir un environnement de travail personnalisé à chaque utilisateur (choix des entrées-sorties, chemins d'accès aux commandes et aux bibliothèques, etc.)

Langage de commande

Le langage de commande interprété par le *shell* a les fonctions suivantes :

- Exécution de commandes intégrées (directement exécutées par le *shell*)
- Exécution de commandes contenues dans les fichiers (préexistantes ou fournies par l'utilisateur)
- Constructions permettant l'exécution conditionnelle (`if`) ou la répétition (`while`) de commandes
- Accès aux variables d'environnement
- Exécution et coordination de processus (exécution de travaux en parallèle, pipelines)
- Navigation dans les répertoires
- Contrôle des travaux (rappel ci-contre, cf cours 2)
- Gestion des flots d'entrée-sortie

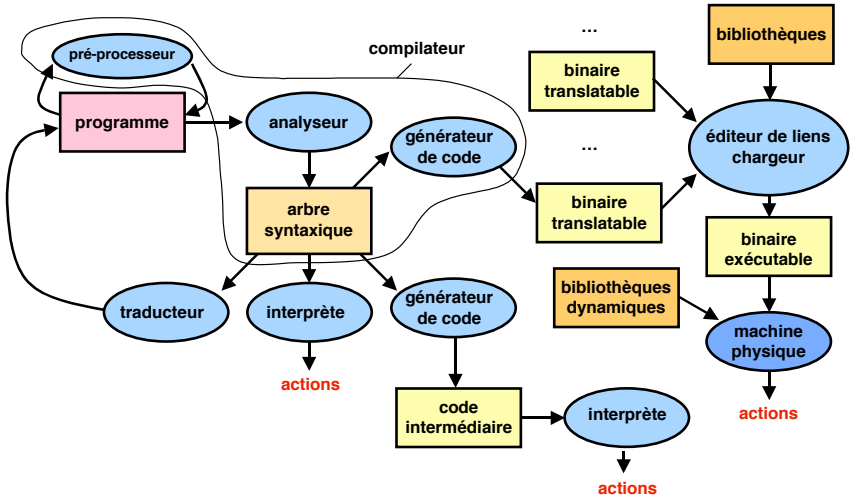


Organisation (très schématique) d'un *shell*

```
initialisation (installation traitants, etc;)
while (TRUE) {
      Lire une commande (sur stdin)
      Analyser la commande
      switch {
  commande intégrée :   exécuter commande; break;
      pipeline      :   créer tube(s) et processus;
                        lancer commandes; break;
 travail de fond    :   créer processus (fork+exec)
                          en travail de fond; break;
 travail interactif :   créer processus (fork+exec+wait)
                          au premier plan
           }
      Ramasser les processus zombis (travaux de fond finis)
      }
```

Davantage de détails en TD et TP

Schéma général d'exécution de programmes



Résumé de la séance 5

- **Exécution de programmes**
  - Programmes compilés, programmes interprétés
  - Avantages respectifs des deux schémas, schémas mixtes
- **Cycle de vie d'un programme compilé**
  - Programmes translatables, principe d'un chargeur
  - Composition de programmes, principes d'un éditeur de liens
  - Bibliothèques statiques
  - Bibliothèques dynamiques
- **Principe de fonctionnement d'un interprète**
  - Exemple du *shell* Unix

Rappels (non présentés en cours)

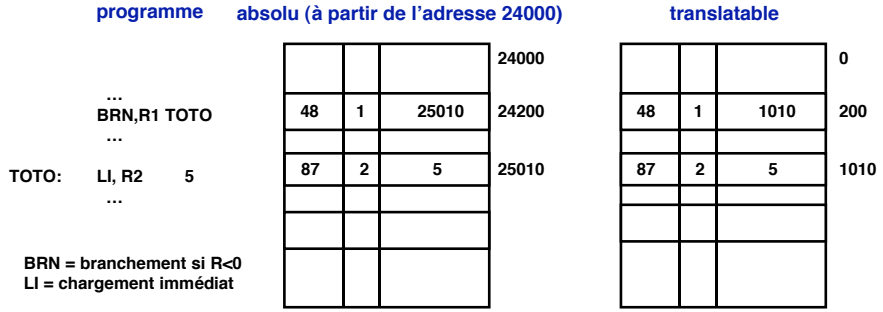
Rappels sur le principe de fonctionnement des chargeurs et éditeurs de liens (vu en ALM)

Code exécutable et chargeurs

Nous illustrons le fonctionnement du chargeur avec un format simple d'instructions machine

<code-op> <registre><adresse>

Programme objet **absolu** (exécutable) : toutes les adresses sont absolues
Programme objet **translatable** (ou relogeable) : les adresses sont relatives au début du programme



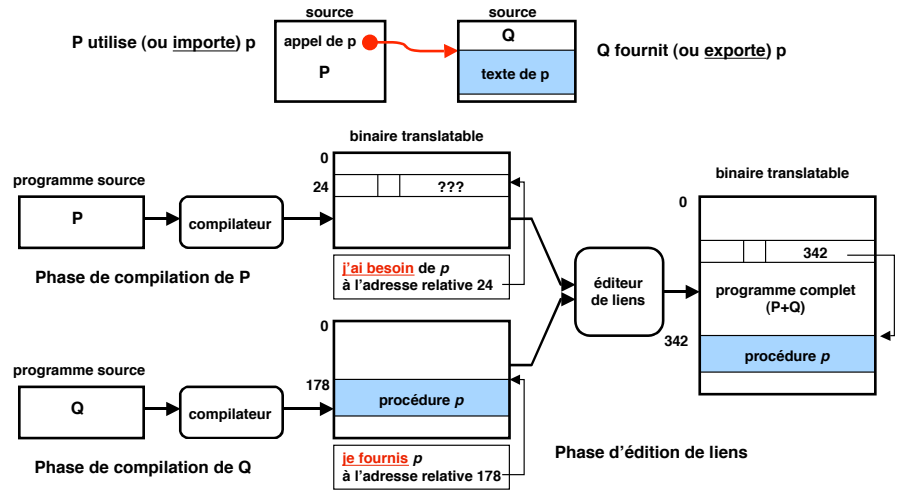
Fonctionnement d'un chargeur

- **Format du binaire translatable**
  toutes les informations dans la partie adresse ne sont pas translatables ; il faut donc un indicateur associé à toute adresse, signalant si elle est translatable ou non ; cet indicateur est placé par le compilateur lors de la génération du code
- **Fonctionnement du chargeur**
  l'adresse absolue de chargement *A* est fournie en paramètre
  algorithme du chargeur:

> **copier le segment *St* contenant le binaire translatable dans un segment *Sa***
> **pour toute adresse *a* contenue dans le segment *Sa* faire**
> **si *a* translatable alors *a* = *a* +*A***
> **le segment *Sa* contient maintenant le binaire absolu**

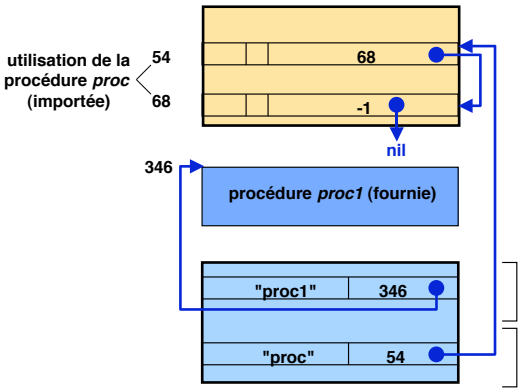
- **Illustration de la notion de liaison**
  liaison = établissement de la relation entre une entité (abstraite) et sa réalisation physique

Fonctionnement d'un éditeur de liens (1)



Fonctionnement d'un éditeur de liens (2)

Format du binaire translatable (très simplifié)



- En général, un programme importe certains noms et en exporte d'autres
- Dans le binaire translatable, le compilateur crée une table des références externes pour les noms importés et une table des définitions externes pour les noms exportés.
- Ces tables sont utilisées par l'éditeur de liens

table des définitions externes

table des références externes

Principe de l'éditeur de liens à 2 passes (simplifié)

- Phase initiale : l'éditeur de liens réunit bout à bout l'ensemble des segments binaires translatables pour faire un segment objet unique. Toute information a alors une adresse (relative) dans ce segment objet (le binaire translatable global)
- Passe 1 : collecte des références. À partir des tables des définitions externes, l'éditeur de liens construit une table globale des symboles. Une entrée de cette table contient le nom d'une procédure fournie et l'adresse (relative) correspondante dans le segment objet.
- Passe 2 : résolution des références externes. À partir des tables des références externes, l'éditeur de liens retrouve toutes les instructions faisant référence à une procédure externe, et met dans la partie adresse de chacune de ces instructions l'adresse relative trouvée dans l'entrée de la table globale des symboles correspondant à la procédure externe utilisée.
- À la fin de la passe 2, toutes les références externes doivent avoir été résolues. Si ce n'est pas le cas, l'éditeur de liens signale les références non résolues, et le programme global ne peut pas être exécuté.
- Le programme global doit encore passer par le chargeur pour obtenir un binaire exécutable. Souvent, la fonction du chargeur est incluse dans l'éditeur de liens, qui produit alors directement le binaire exécutable.