

Quatrième chapitre

Exécution des programmes

- Schémas d'exécution
- Interprétation
 - Principe de fonctionnement d'un interpréteur
 - Exemple d'interpréteur : le shell
- Compilation et édition de liens
 - Principes de la compilation
 - Liaison, éditeur de liens et définitions multiples en C
 - Bibliothèques statiques et dynamiques
- Conclusion

Schémas d'exécution d'un programme

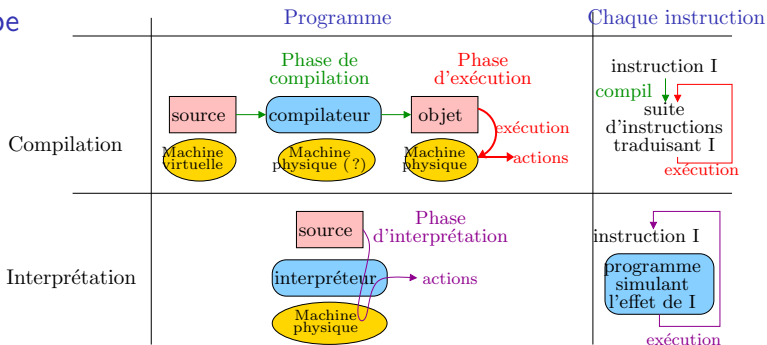
- ▶ **Programme** (impératif) : suite d'instructions, d'actions successives
 - ▶ **Exécution** : faire réaliser ces actions par une machine dans un état initial pour l'amener à un état final
 - ▶ Langage [de la] machine : suite de 0 et de 1, difficile pour humains
 - ▶ Langues humaines ambiguës et complexes, inaccessibles aux machines
- ⇒ les humains utilisent des langages informatiques traduits en langage machine

Convertir les langages informatiques en langage machine

- ▶ **Compilation** : conversion à priori, génération d'un fichier *binaire exécutable* depuis un fichier *source* par un **compilateur**
- ▶ **Interprétation** : programme auxiliaire (**interpréteur**) traduit au fur et à mesure
interpréteur \approx simulateur de machine virtuelle ayant un autre langage machine
- ▶ **Mixte** : compilation pour une machine intermédiaire
+ interprétation par une *machine virtuelle*

Compilation et interprétation

Principe



Exemples

- ▶ **Compilation** : C, C++, Fortran, Ada, assembleur
- ▶ **Interprétation** : shell Unix, Tcltk, PostScript, langage machine
- ▶ **Autre** : Lisp (les deux), Java/Scala (machine virtuelle), Python (interprétation ou machine virtuelle), Ocaml (interprétation, compilation ou machine virtuelle)
- ▶ On peut faire de la compilation croisée voire pire

Compilation et interprétation : comparaison

Compilation

😊 Efficacité

- ▶ Génère du code machine natif
- ▶ Ce code peut être optimisé (lien)

☹ Mise au point

- ▶ Lien erreur ↔ source complexe

☹ Cycle de développement

- ▶ cycle complet à chaque modification :
compilation, édition de liens,
exécution

Interprétation

☹ Efficacité

- ▶ 10 à 100 fois plus lent
- ▶ appel de sous-programmes
- ▶ pas de gain sur les boucles

😊 Mise au point

- ▶ Lien instruction ↔ exécution trivial
- ▶ Trace et observation simples

😊 Cycle de développement

- ▶ cycle très court :
modifier et réexécuter

- ▶ L'interprétation regagne de l'intérêt avec la puissance des machines modernes
- ▶ On peut parfois (eclipse/Visual) recharger à chaud du code compilé

Schéma mixte d'exécution

Objectifs

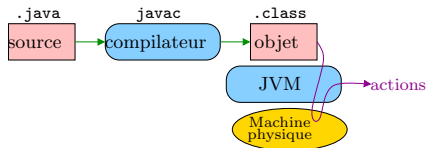
- ▶ Combiner les avantages de la compilation et de l'interprétation
- ▶ Gagner en portabilité sans trop perdre en performances

Principe

- ▶ Définir une machine virtuelle (qui n'existe donc pas)
- ▶ Écrire un simulateur de cette machine virtuelle
- ▶ Écrire un compilateur du langage source vers le langage machine virtuel

Exemple : Java inventé pour être (le langage d'internet)

- ▶ Programmes (applets) téléchargés doivent fonctionner sur toutes les machines
- ▶ Phase de compilation et objets indépendants de machine physique
- ▶ Portage sur une nouvelle machine : écrire une JVM pour elle



Quatrième chapitre

Exécution des programmes

- Schémas d'exécution
- **Interprétation**
 - Principe de fonctionnement d'un interpréteur
 - Exemple d'interpréteur : le shell
- Compilation et édition de liens
 - Principes de la compilation
 - Liaison, éditeur de liens et définitions multiples en C
 - Bibliothèques statiques et dynamiques
- Conclusion

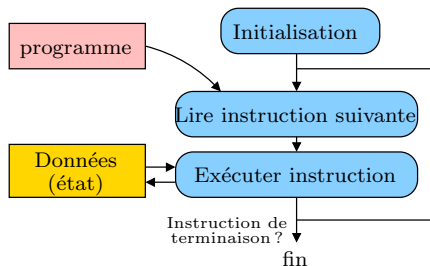
Principe de fonctionnement d'un interpréteur

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

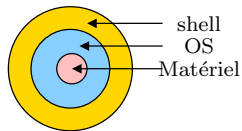
- ▶ Analogie avec cycle processeur
- ▶ Pseudo-compteur ordinal (PC)



Exemple d'interpréteur : le shell

Définition

- ▶ Programme interface entre OS et utilisateur interactif
Interpréteur le langage commande (→ appels systèmes)
- ▶ Mot à mot, shell=coquille, car ça «englobe» l'OS
- ▶ Il existe plusieurs shells (sh, csh, tcsh, bash, zsh, etc.)



Fonctions principales d'un shell

- ▶ **Interpréteur commandes des usagers** ; accès aux fonctions de l'OS
 - ▶ Création et exécution de processus
 - ▶ Accès aux fichiers et entrées / sorties
 - ▶ Utilisation d'autres outils (éditeur, compilateur, navigateur, etc.)
- ▶ **Gestionnaire de tâches** : travaux en mode interactif ou en tâche de fond
- ▶ **Personnaliser l'environnement de travail** : variables d'environnement
- ▶ **Scripting** : programmation en langage de commande (if, while, etc.)

Quatrième chapitre

Exécution des programmes

- Schémas d'exécution
- Interprétation
 - Principe de fonctionnement d'un interpréteur
 - Exemple d'interpréteur : le shell
- **Compilation et édition de liens**
 - Principes de la compilation
 - Liaison, éditeur de liens et définitions multiples en C
 - Bibliothèques statiques et dynamiques
- Conclusion

Composition de programmes

Les programmes ne s'exécutent jamais «seuls»

- ▶ Applications modernes = assemblage de modules (conception et évolution 😊)
- ▶ Applications modernes utilisent des «bibliothèques» de fonctions
- ▶ Cf. programmation objet et prolongement «par composants» (en PAR, 3A)



Problème : la liaison

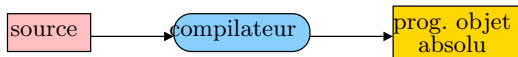
- ▶ Un programme P appelle une procédure $proc$ définie dans Q
- ▶ Appel de routine en assembleur : `jump` à une adresse donnée
- ▶ **Question** : quelle adresse associer à $proc$ dans le code machine de P ?
- ▶ **Réponse** : on ne sait pas tant que le code machine de Q est inconnu
- ▶ **Solution** : Dans P , liaison entre $proc$ et son adresse faite après compilation

Édition de liens

Cycle de vie d'un programme compilé

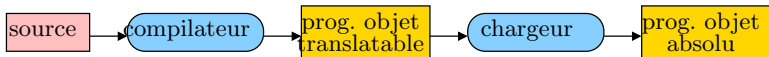
► Compilation en programme objet absolu

- Adresses (des routines) en mémoire fixées



► Compilation en programme objet translatable

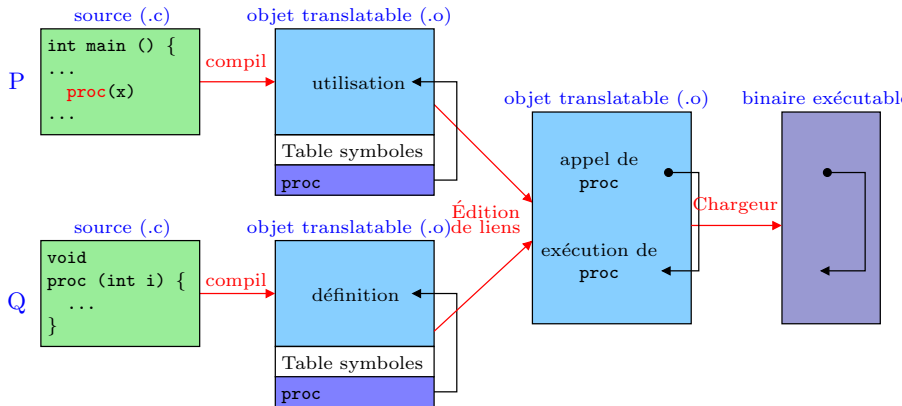
- Adresses définies à une traduction près
- ☺ On peut donc regrouper plusieurs objets translatables dans un seul gros
- ☹ Ce n'est pas exécutable (le CPU ne translate pas)
- ⇒ **chargeur** convertit code translatable (combinable) en code absolu (exécutable)



► Exemples

- `gcc -c prog.c` ⇒ `prog.o` = programme objet translatable
- `gcc -o prog prog.c` ⇒ `prog` programme objet absolu (compilo + chargeur)

Édition de liens



- L'éditeur de lien et le chargeur sont souvent regroupés dans le même utilitaire : `ld`

Exemples de commandes de compilation

- ▶ La commande `gcc` invoque compilateur, éditeur de liens et chargeur

`gcc prog.c` → `a.out` = binaire exécutable

`gcc -o prog prog.c` → `prog` = binaire exécutable

`gcc -c prog.c` → `prog.o` = binaire translatable

- ▶ Si un programme est composé `prog1.c` et `prog2.c`, alors :

`gcc prog1.c prog2.c` → binaire exécutable complet dans `a.out`

`gcc -c prog1.c prog2.c` → `prog1.o` et `prog2.o` = 2 objets translatables

`gcc -o prog prog.o prog1.o` → `prog` depuis objets translatables

Quelques problèmes de l'édition de liens

Théorie relativement simple...

► Définition manquante

```
...  
proc(x);  
...  
/* pas de définition de proc */
```

```
$ 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  
$
```

► Définitions multiples

```
...  
proc(x);  
...  
int proc; /* déf. comme entier */  
...  
void proc(int x) {  
... /* déf. comme fonction */  
}
```

```
$ gcc -o prog prog1.c prog2.c prog3.c  
$
```

- Pas de message d'erreur !
- Que se passe-t-il ?

... pratique souvent déroutante

Définitions multiples en C

Deux catégories de symboles externes

- ▶ Symboles «externes» : visible de l'éditeur de liens (hors de toute fonction)
- ▶ **Symbole fort** :
 - ▶ définition de procédures
 - ▶ variables globales initialisée
- ▶ **Symbole faible** :
 - ▶ déclaration de procédures en avance (sans le corps de fonction)
 - ▶ variables globales non-initialisée

Règles de l'éditeur de liens pour les définitions multiples

- ▶ deux symboles forts : interdit ! (erreur détectée)
- ▶ un symbole fort et plusieurs faibles : le fort est retenu
- ▶ plusieurs symboles faibles : choix aléatoire

Pièges des définitions multiples

Module A

```
int proc(int i) {
```

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

```
int x; /* 4o */  
int y; /* 4o */  
int p1(int i) {
```

```
int x=3;  
int y;  
int p1(int i) {
```

Module B

```
int proc(int i) {
```

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

```
double x; /* 8o */  
int p2(int i) {
```

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

Deux symboles forts : interdit

Deux symboles faibles
désignent le même objet

Deux symboles faibles.
Modifier B.x écrase A.y!!

A.x fort ; B.x faible ;
Modifier B.x écrase encore A.y!!

► Contrôle de types strict entre modules pas obligatoire en C

⇒ Déclaration explicite des références externes nécessaire

Exercice : que se passe-t-il dans l'exemple précédent (deux pages avant) ?

int proc = symbole faible ; void proc(int x) = symbole fort, c'est lui qui est choisi (mais comportement étrange quand on change int proc)

Déclarations des variables externes

- ▶ Règle 1 : utiliser des entête (.h) pour unifier les déclarations
- ▶ Règle 2 : éviter les variables globales autant que possible!
 - ▶ déclarer `static` toute variable hors des fonctions (visibilité = fichier courant)
- ▶ Règle 3 : bon usage des globales (si nécessaire)
 - ▶ chaque globale déclarée fortement dans `un` module *M*
 - ▶ `extern` dans tous les autres (avec son type)
 - ▶ Seulement consultation depuis l'extérieur, modification seulement depuis *M*
 - ▶ Fonctions dans *M* pour la modifier depuis ailleurs, au besoin (*setters*)
- ▶ Règle 4 : bon usage des fonctions
 - ▶ déclaration explicite des fonctions externes (avec signature)
 - ▶ mot clé `extern` optionnel : la signature (ou prototype) suffit

Fichier un.c

```
int x;
extern double y;
static int z;
int f(int i) {... }
static int g(int i) {... }
/* x: écriture
   y: lecture
   z: écriture
   f: accessible
   g: accessible */
```

Fichier deux.c

```
extern int x;
double y;

/* x: lecture
   y: écriture
   z: non-référençable
   f: invisible
   g: non-référençable */
```

Fichier trois.c

```
extern double y;
int f(int i);

/* x: invisible
   y: lecture
   z: non-référençable
   f: accessible
   g: non-référençable */
```

Exemples

Module A

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

```
void p2(void);  
int main() {  
    p2();  
    return 0;  
}
```

Module B

```
static double f=4.5;
```

```
#include <stdio.h>  
  
char main;  
  
void p2() {  
    printf("0x%x\n",  
          main);  
}
```

Que se passe-t-il ?

Pas de problème

f.B static \Rightarrow invisible d'ailleurs

Ca marche (et affiche 0x55).

Link sans soucis : main.B est faible, il est écrasé

Execution : ref(main.B) = def(main.A), et 0x55 est l'adresse du main() à gauche

Exercice : pourquoi chaque programme doit contenir une fonction main() ?

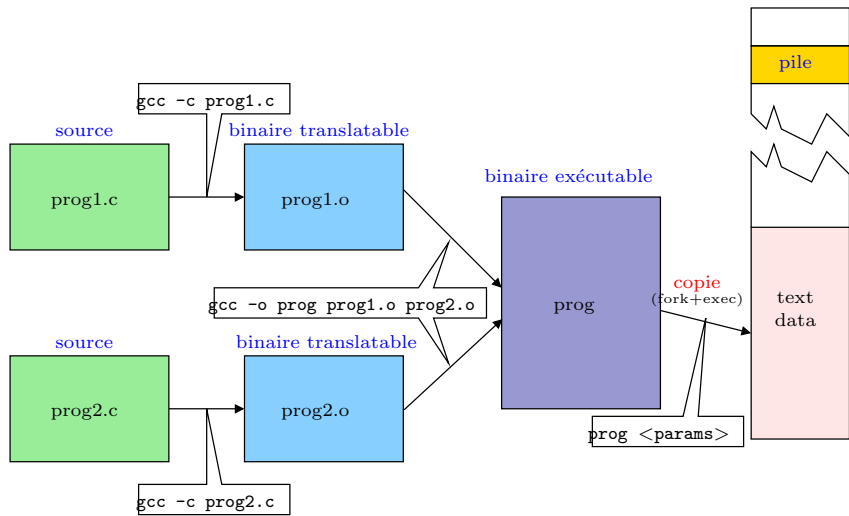
car par convention, exec() passe le contrôle à une fonction de ce nom

Exercice : que se passe-t-il quand main() fait return ou exit()

return : la fonction qui a invoqué main() reprend le contrôle. Elle appelle _exit()

exit : _exit() aussi ; libc reprend le contrôle, termine le processus et prévient l'OS

Cycle de vie d'un programme : résumé



Quatrième chapitre

Exécution des programmes

- Schémas d'exécution
- Interprétation
 - Principe de fonctionnement d'un interpréteur
 - Exemple d'interpréteur : le shell
- **Compilation et édition de liens**
 - Principes de la compilation
 - Liaison, éditeur de liens et définitions multiples en C
 - Bibliothèques statiques et dynamiques
- Conclusion

Bibliothèques

- ▶ Bibliothèque = recueil de fonctions utilisées dans divers programmes
- ▶ Ne pas réinventer la roue ; réutilisation du code
- ▶ Exemples : `atoi`, `printf`, `malloc`, `random`, `strcmp`, `sin`, `cos`, `floor`, etc.

Divers moyens de rendre ce service :

- ▶ Réalisation par le compilateur :
 - ▶ Compilateur Pascal remplace appels fonctions standards par code correspondant
 - ☹ Changer le compilateur pour changer ce code
- ▶ Approche par binaire translatable :
 - ▶ Le code de toutes les fonctions standards placé dans un `/usr/lib/libc.o`
 - ☹ Plusieurs Mo à chaque fois, même si inutilisé
- ▶ Approche par collections de binaires transposables :
 - ▶ Code chaque fonction dans un binaire translatable différent
 - ☹ `gcc toto.c /usr/lib/printf.o /usr/lib/scanf.o ...` (un peu lourd)
- ▶ Approche par archives de binaires transposables (**bibliothèques statiques**) :
 - ▶ Concaténation de tous les `.o` dans un fichier, et l'éditeur de lien choisi

Bibliothèques statiques

- ▶ `/usr/lib/libc.a` : bibliothèque standard du langage C (`printf`, `strcpy`, etc.)
- ▶ `/usr/lib/libm.a` : bibliothèque mathématique (`cos`, `sin`, `floor`, etc.)

Utilisation :

- ▶ Passer bibliothèques utilisées à l'éditeur de lien (`libc.a` ajoutée par défaut)
`gcc -o prog prog.c /usr/lib/libm.a`
- ▶ L'éditeur de liens ne copie que les «fonctions» effectivement utilisées
- ▶ **Notation abrégée** : `-l<nom>` équivalent à `/usr/lib/lib<nom>.a`
Exemple : `gcc -o prog prog.c -lsocket` (inclut `/usr/lib/libsocket.a`)
- ▶ **Chemin de recherche** : Variable `LIBPATH`, ajout avec `gcc -L<rep>`

Format et création

- ▶ Archives de binaires translatables
- ▶ Manipulées par `ar(1)`
`ar rcs libamoi.a fonct1.o fonct2.o fonct3.o`
- ▶ **Remarque** : historiquement, `tar` est une version particulière de `ar`

Dépendances entre bibliothèques statiques

- ▶ Difficulté lorsqu'une bibliothèque utilise une autre bibliothèque
- ▶ L'ordre des `-ltoto` lors de l'appel à l'éditeur de liens devient important car :
 - ▶ Éditeur de liens ne copie que les objets nécessaires, pas toute l'archive
 - ▶ Il ne parcourt ses arguments qu'une seule fois
- ▶ Règle : déclaration de fonction doit être placée **après** son premier usage

Exercice : quelle(s) commande(s) fonctionne(nt)/échoue(nt) ? Pourquoi ?

`prog.c` utilise `machin()` de `libmachin.a`, et `machin()` utilise `truc()` de `libtruc.a`

- ▶ `gcc -o prog prog.c -ltruc -lmachin` : référence à `truc()` indéfinie
- ▶ `gcc -o prog prog.c -lmachin -ltruc` : fonctionne

Exercice : que faire en cas de dépendances circulaires ?

faire figurer une bibliothèque deux fois pour casser le cycle

Bibliothèques dynamiques

- ▶ Défauts des bibliothèques statiques :
 - ▶ Code dupliqué dans chaque processus les utilisant
 - ▶ Liaison à la compilation (nouvelle version de bibliothèque ⇒ recompilation)
- ▶ Solution : **Bibliothèques dynamiques** :
Partage du code entre applications et chargement dynamique
- ▶ Exemples : Windows : **DLL** (dynamically linkable library) ; Unix : **.so** (shared object)
- ▶ Bibliothèques partagées à plus d'un titre :
sur disque (un seul .so) et en mémoire (physique, du moins)
- ▶ Chargement dynamique :
 - ☹ Impose une édition de lien au lancement (ldd(1) montre les dépendances)
 - 😊 Mise à jour des bibliothèques simplifiée (mais attention au DLHell)
Versionner les bibliothèques (et même les symboles) n'est pas trivial
 - 😊 Mécanisme de base pour les greffons (*plugins*)
 - 😊 Même possible d'exécuter du code au chargement/déchargement!

```
void chargement(void) __attribute__((constructor)) {...}  
void alafin(void) __attribute__((destructor)) {...}
```


Bibliothèques dynamiques

Construction

- ▶ `gcc -shared -fPIC -o libpart.so fonct1.c fonct2.c fonct3.c`
- ▶ `-fPIC` (compil.) : demande code repositionnable (Position Independent Code)
- ▶ `-shared` (éd. liens) : demande objet dynamique

Outils (voir les pages de man)

- ▶ `ar` : construire des archives/bibliothèques
- ▶ `strings` : voir les chaînes affichables
- ▶ `strip` : supprimer la table des symboles
- ▶ `nm` : lister les symboles
- ▶ `size` : taille de chaque section d'un fichier elf
- ▶ `readelf` : afficher la structure complète et tous les entêtes d'un elf (`nm+size`)
- ▶ `objdump` : tout montrer (même le code désassemblé)
- ▶ `ldd` : les bibliothèques dynamiques dont l'objet dépend, et où elles sont prises

Chargement dynamique de greffons sous linux

Greffon (*plugin*) :

- ▶ Bout de code chargeable dans une application pour l'améliorer
- Choisir l'implémentation d'une fonctionnalité donnée (affichage, encodage)
- Ajouter des fonctionnalités (mais bien plus dur, nécessite API dynamique)

Compiler du code utilisant des greffons :

- ▶ `gcc -rdynamic -ldl autres options`

Exercice : Pourquoi ce `-ldl` ?

Pour charger une bibliothèque nommée dl (dynamic library)

Interface de programmation sous UNIX modernes (Linux et Mac OSX)

```
#include <dlfcn.h>
void *dlopen(const char *filename, int flag); /* charge un greffon */
void *dlsym(void *handle, const char *symbole); /* retourne pointeur vers «symbole» */
int dlclose(void *handle); /* ferme un greffon */
char *dLError(void); /* affiche la cause de la dernière erreur rencontrée */
```

Exemple de greffon sous linux

chargeur.c

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <dlfcn.h>
int main(int argc, char * argv[]) {
    char path[256], *erreur = NULL;
    int (*mon_main)();
    void *module;
    /* Charge le module */
    module = dlopen(argv[1], RTLD_NOW);
    if (!module) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    /* Retrouve la fonction d'entrée */
    mon_main = dlsym(module, "mon_main");
    erreur = dlerror();
    if (erreur != NULL) {
        perror(erreur);
        exit(1);
    }
    /* Appelle cette fonction */
    (*mon_main)();
    /* Ferme tout */
    dlclose(module);
    return 0;
}
```

module_un.c

```
int mon_main() {
    printf("Je suis le module UN.\n");
}
```

module_deux.c

```
int mon_main() {
    printf("Je suis le module DEUX.\n");
}
```

```
$ gcc -shared -fPIC -o un.so module_un.c
$ gcc -shared -fPIC -o deux.so module_deux.c
$ gcc -o chargeur chargeur.c -rdynamic -ldl
$ ./chargeur ./un.so
Je suis le module UN.
$ ./chargeur ./deux.so
Je suis le module DEUX.
$ ldd chargeur
libld.so.2 => /lib/i686/cmox/libld.so.2 (0xb7f7e70)
libc.so.6 => /lib/i686/cmox/libc.so.6 (0xb7e7e70)
/lib/ld-linux.so.2 (0xb7fd1000)
$ nm chargeur | grep mon_main
$
```

Bibliothèques dynamiques et portabilité

Chaque système a sa propre interface

- ▶ Solaris, Linux, MacOSX et BSD : dlopen (celle qu'on a vu, par SUN)
- ▶ HP-UX : shl_load
- ▶ Win{16,32,64} : LoadLibrary
- ▶ BeOS : load_add_on

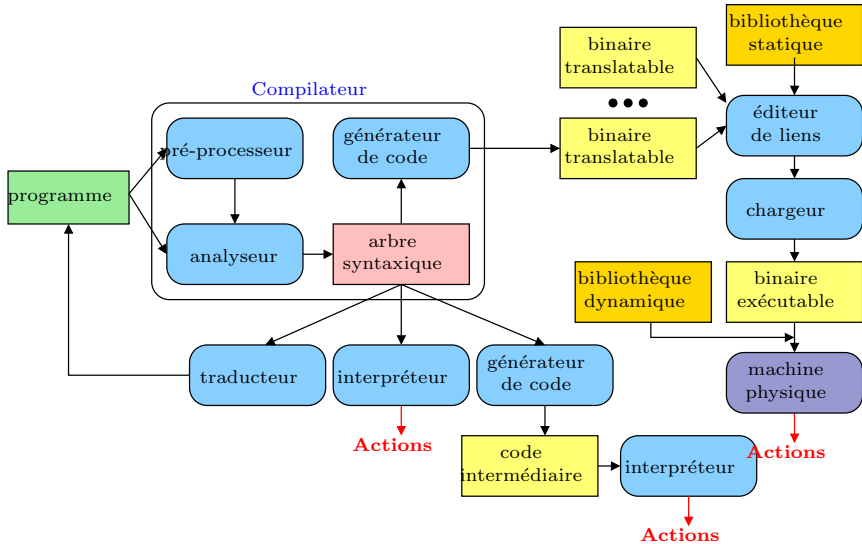
libtool offre une interface portable

- ▶ nommé libltdl (LibTool Dynamic Loading library)
- ▶ Il permet également de compiler une bibliothèque dynamique de façon portable
- ▶ Associé à autoconf (usage peu aisé ; compilation depuis UNIX principalement)

Interface (fonctions de base : similaires à dlopen)

```
#include <ltdl.h>
lt_dlhandle lt_dlopen(const char *filename);
lt_ptr_t lt_dlsym(lt_dlhandle handle, const char *name);
int lt_dlclose(lt_dlhandle handle);
const char *lt_dlerror(void);
int lt_dlopen_init(void);
int lt_dlexit(void);
```

Schéma général d'exécution de programmes



Résumé du quatrième chapitre

Schémas d'exécution. Problème :

- ▶ Interprétation : peu efficace, mais pratique
- ▶ Compilation : efficace, mais lors de la mise au point : cycle complet à chaque modif
- ▶ Autres approches : machine virtuelle (Java)

Bibliothèques et liaison. Pourquoi :

- ▶ Éditeur de lien : assembler les "modules"
- ▶ Définitions multiples en C
- ▶ Bibliothèques statiques
 - ▶ Pourquoi : collection de fonctions
 - ▶ Comment : archive ar
 - ▶ Avantages et défaut : simple, mais duplique le code dans les binaires
- ▶ Bibliothèques dynamiques
 - ▶ Pourquoi : partage du code entre applications
 - ▶ Comment : édition de liens au lancement