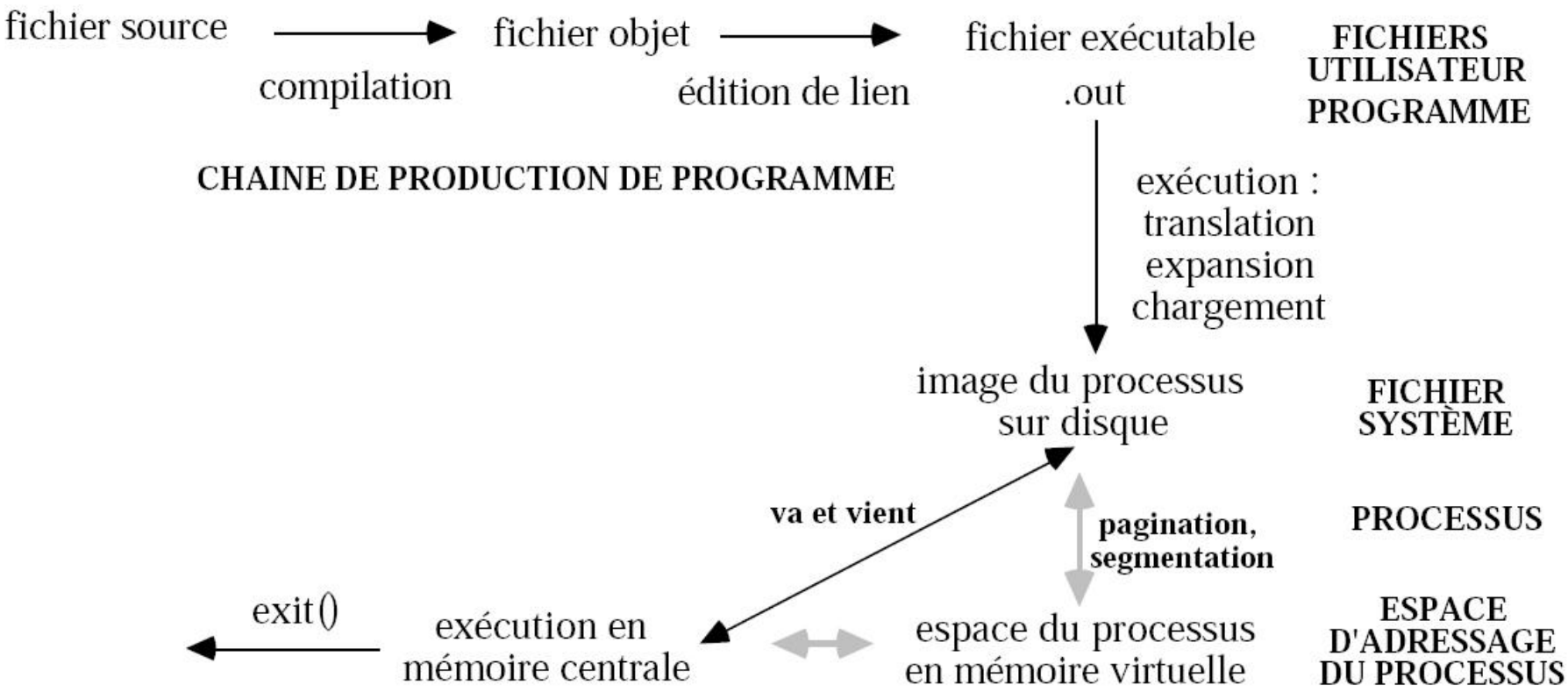
Cours Système
Miage de l'université Paris Ouest
Agilité des systèmes d'information et e-business

J.-F. Pradat-Peyre
Université Paris X - Nanterre - UFR SEGMI

Septembre 2010

2 : Du programme au processus

Du programme au processus



Traduction de programmes : Interprétation

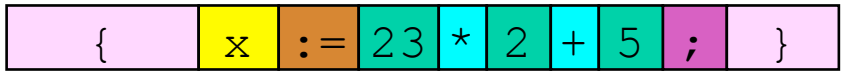
- Langage de haut niveau chaque instruction (ou bloc d'instruction) est lue par un programme qui
  - « comprend » (lecture et sémantique)
  - Execute cette instruction lui même.
- Analogie avec interprète d'une langue en une autre.
- Exemple : vba, matlab, SAS, python, bash
- Avantage : Portabilité le programme tourne de la même manière sur toutes les plateformes qui ont un interpréteur.
- Interpréteur est lui dépendant de la machine

Traduction de programmes (3)

- Compilation = traduction
- Analogie traduction d'un texte écris en un texte écris en une autre langue.
- Différentes phases sont enchaînées
  - Analyse lexicale : trouver les lexems
  - Analyse syntaxique : vérifier la syntaxe et comprendre les suites de lexems
  - Analyse sémantique : comprendre les valeurs et données manipulées
  - Optimisation :
  - Génération de code

Analyse lexicale

- module source
  - suite de caractères,
  - représentation d ’une suite de symboles
- Retrouver les symboles à partir des caractères
  - exemple: `{ x := 23*2 + 5 ; }`
  - retrouver le regroupement:



- Coder les symboles dans une représentation interne
- Implique une structure lexicale non ambiguë

Analyse syntaxique

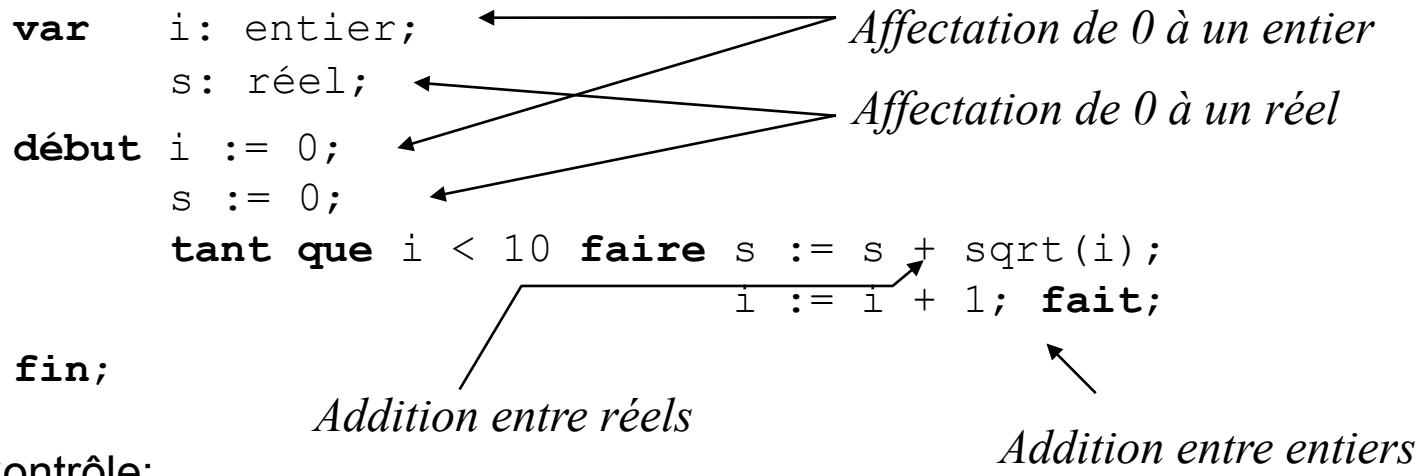
- Vérifier la concordance de la suite de symboles avec la structure du langage
- *règles de production*
  - règles de construction d'une suite de symboles du langage
- *analyse syntaxique*
  - retrouver les règles utilisées par le programmeur
- Syntaxe
  - non ambiguë
  - indépendante du contexte
  - doit faciliter cette reconstruction

Analyse sémantique (1)

- trouver les objets manipulés par le programme
  - désignés par des identificateurs
- trouver les propriétés d'un objet
  - Comment?
    - ✓ déduites implicitement de l'utilisation
    - ✓ à partir de déclarations explicites
  - son *type* => sémantique des opérations, combinaison d'opérations machine
  - sa *durée de vie* => quand doit-il être créé ou détruit
  - sa *taille* => nombre d'emplacements mémoire
  - son *adresse* => désignation dans les instructions machine

Analyse sémantique (2)

- trouver les actions du programme sur ces objets



- Contrôle:
  - déclarations présentes, utiles et compatibles
  - signification des expressions, etc...
- Problème: la sémantique est-elle celle désirée => unicité

Génération de code et optimisation

- Génération
  - construction du programme machine équivalent
- Optimisation
  - améliorer l'efficacité du résultat produit (*et non le programme lui-même*)
  - compenser la perte d'efficacité due à la programmation de haut niveau
  - mesure de la qualité d'un compilateur
- Exemples
  - expressions constantes évaluées à la compilation
  - élimination du code inaccessible => compilation conditionnelle
  - sortir des boucles les instructions ayant même résultat
  - remplacer les calculs liés aux variables de boucles par d'autres + essentiels
    => progression d'adresses au lieu d'indice de tableau

Détails sur les modules

- Format ELF = “Executable and Linking Format”
- L'édition de liens et les bibliothèques
- Le chargement

Exemple de format de module : la structure ELF

- Format remplaçant l'ancien format « a.out » d'Unix sous BSD et Linux
- ELF = "Executable and Linking Format"
- Définit le format des fichiers binaires exécutables : 4 types
  - Fichiers « relogeables » ou « translatables » :
    - ✓ produits par les compilateurs
    - ✓ doivent être traités ensuite par le linker
  - Exécutable
    - ✓ Les translations d'adresse ont été faites et les symboles résolus (exceptés ceux résolus à l'exécution)
  - Les modules partagés (shared library)
  - Les fichier « Core »

Les deux faces d'un fichier ELF

Vue coté compilateurs

Vue coté chargeur

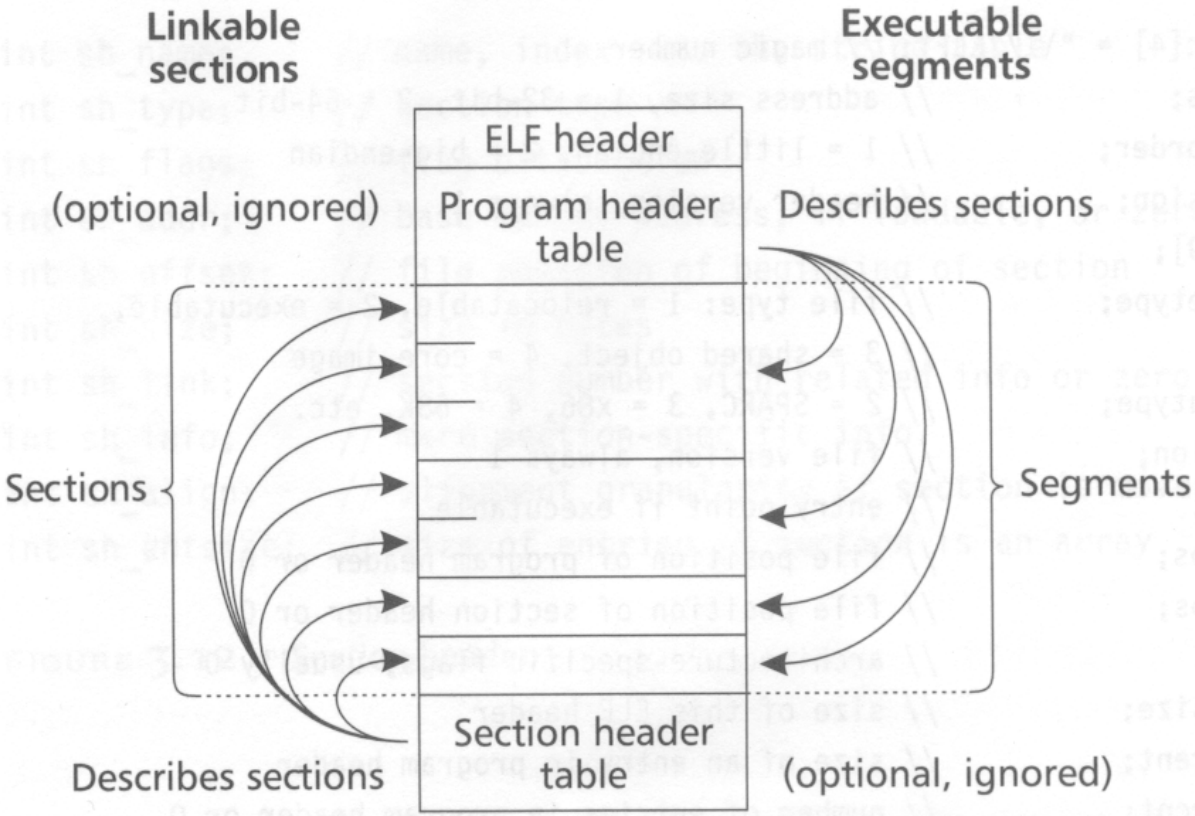


FIGURE 3.10 • Two views of an ELF file.

Notion de module translatable

- Appelés souvent fichier objets
  - Représentent une suite d'instructions en langage machine
- Emplacements disjoints des modules à l'exécution
- En général => pouvoir mettre les modules n'importe où
  - sections différentes: code, données constantes, variables
  - indiquer la nature de la section
  - indiquer les emplacements contenant des adresses relatives à une section

  => ajouter à chacun de ces emplacements l'adresse de début de la section correspondante

Format d'un fichier programme translatable (ELF)

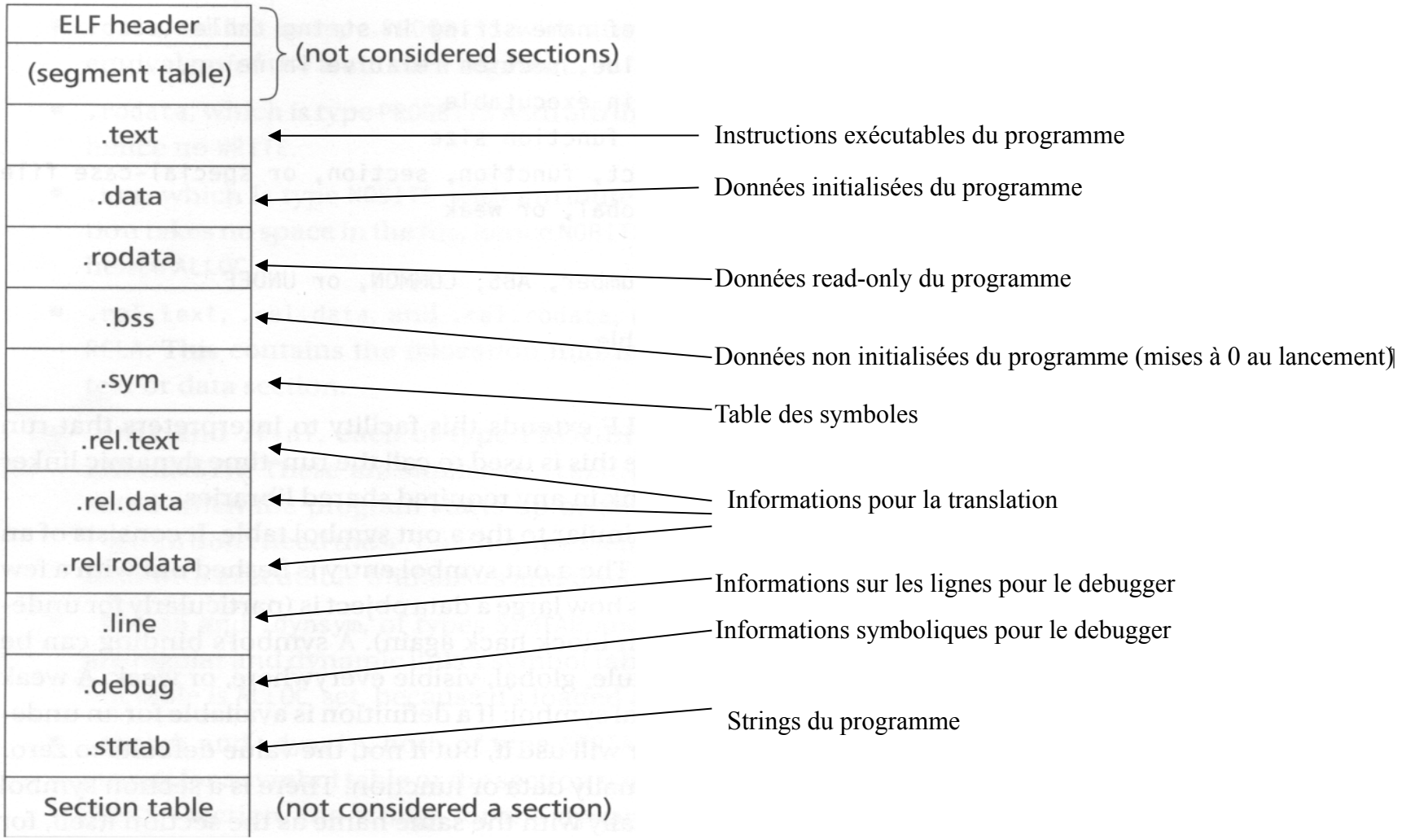


Table des symboles

Les informations permettant d'utiliser les liens sont regroupées dans une table des symboles

```
int name;      // position of name string in string table
int value;     // symbol value, section relative in reloc,
               // absolute in executable
int size;      // object or function size
char type:4;   // data object, function, section, or special-case file
char bind:4;   // local, global, or weak
char other;    // spare
short sect;    // section number, ABS, COMMON, or UNDEF
```

ELF symbol table.

Édition d'une liste de modules

- Trois étapes
  - placement des sections provenant des modules
    - ✓ regroupement selon leur nature (protection):
      - code -> exécution
      - constantes -> lecture
      - variables -> lecture/écriture

    => définition des adresses d'implantations des sections

    ad (section n+1) := ad (section n) + taille (section n)

  - construction de la table des liens utilisables (transformation adresse)
  - construction du programme
    - ✓ translation du module
    - ✓ résolution des liens à satisfaire

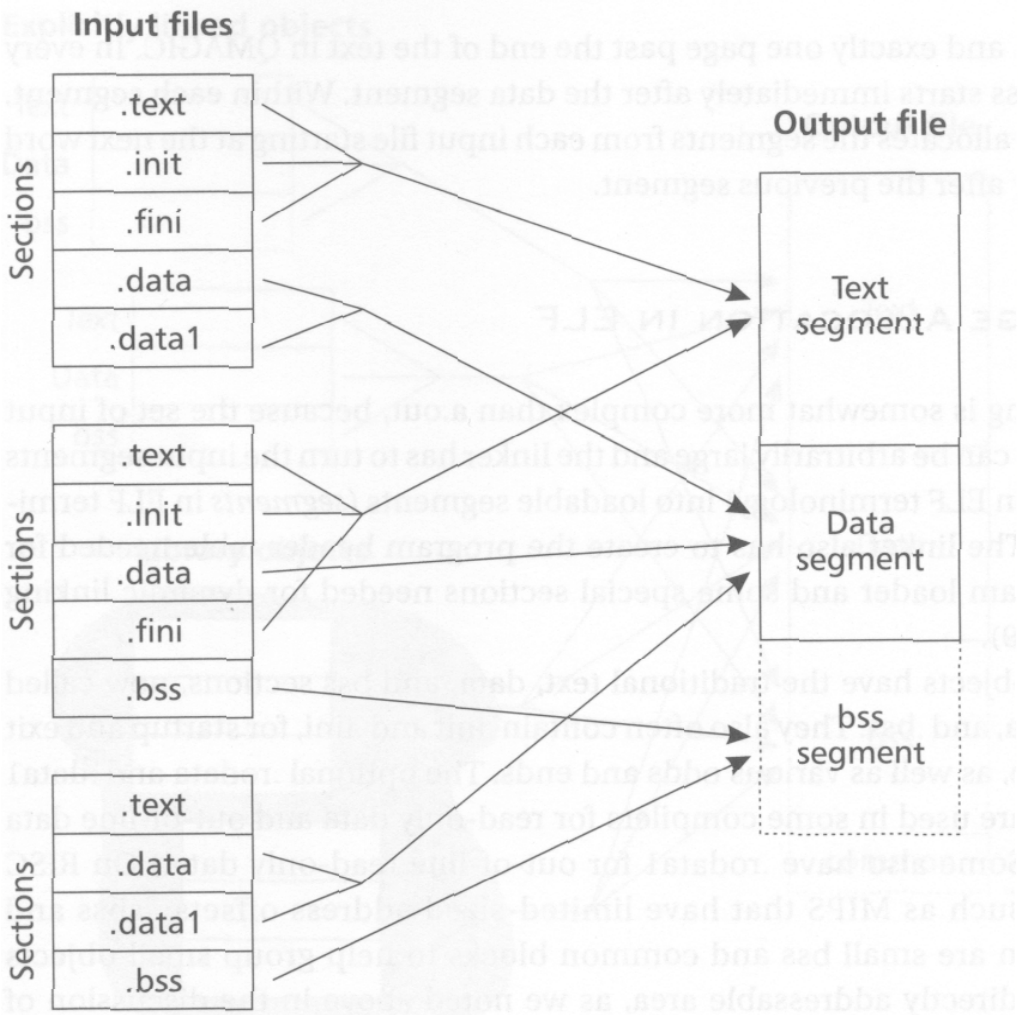
      si pas de lien utilisable pour un lien à satisfaire => non résolu
      anomalie mais non erreur

fusion

Se fait en dur : les bibliothèques et modules sont inclus dans l'executable

Edition de fichiers au format ELF

Fichiers relogeables



Fichier exécutable

Chargement

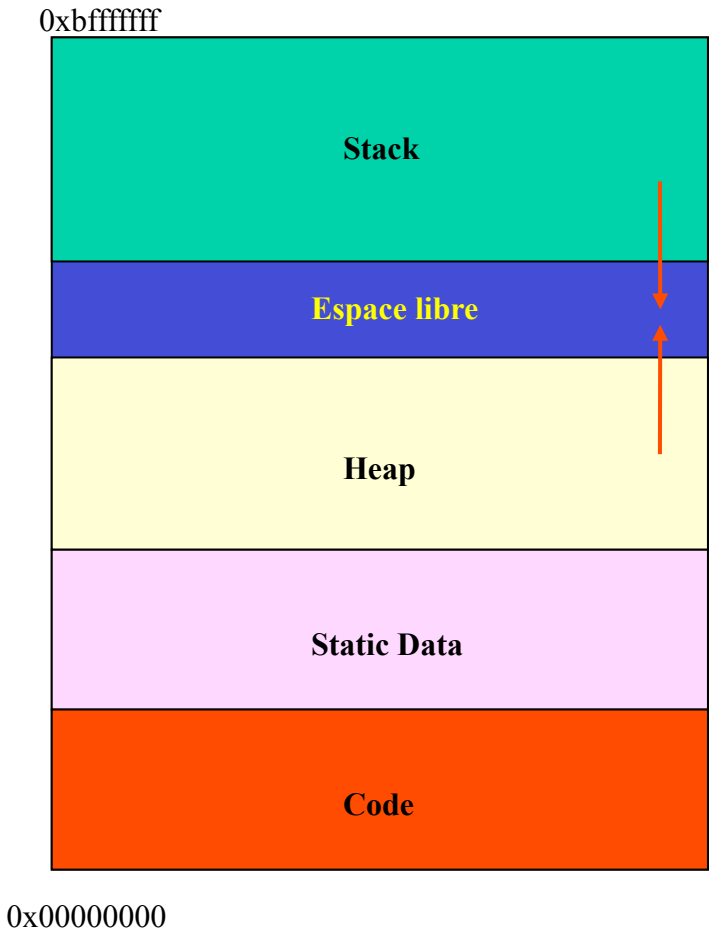
- Édition de liens => programme exécutable
- Chargement = mise en mémoire et lancement
  - lecture en emplacement fixe => trop restrictive
  - avec translation => mise en mémoire n'importe où
    - ✓ informations de translation (traducteurs -> éditeur de liens)
  - adresse de lancement = adresse 1ère instruction à exécuter
    - ✓ traducteur reconnaît le programme principal et fournit l'adresse
    - ✓ éditeur de lien la transforme en adresse dans programme
  - environnement du programme exécutable => à construire
    - ✓ programme autonome = machine nue
    - ✓ programme sur machine abstraite = construire la machine

faire l'édition de liens dynamique, partage de sous programmes, ...

Fichiers dll sous windows .so sous linux

Chargement en mémoire d'un exécutable

- Un processus est créé
- Quatre zones mémoire (segments) sont construites en mémoire virtuelle à partir des données du fichier (segments text, data et bss)
  - Segment de code (appelé aussi text)
  - Segment des données statiques initialisées + non initialisées (bss)
  - Segment pile (Stack): appel de fonctions, variables locales, …
  - Segment tas (Heap) : malloc(), …
- Les références externes sont résolues
- Le compteur ordinal du processus est fixé à l'adresse de la première instruction



Regroupements de modules : Bibliothèque de modules (1)

- bibliothèque de calcul
  - sous programme standard: sinus, cosinus, racine_carrée, etc.
  - sous programmes liés à un langage
- bibliothèque d'interface système
  - sous programmes exécutant les appels systèmes
  - préparent les paramètres, et exécutent l'appel
  - appel système traité différemment d'un appel de sous programme:
    - ✓ contrôles incontournables
    - ✓ pas d'établissement de liaison par l'éditeur de liens (indépendance)
    - ✓ instruction particulière permettant le changement de mode
- bibliothèque d'utilisateur
  - sous programmes pour des besoins spécifiques

Regroupements de modules : Bibliothèque de modules (3)

- En c (usuellement)
- On écrit un ensemble de fonctions relatives dans un fichier
  - Ex : velo.c
  - On peut lister l'ensemble des fonctions dans un header « velo.h »
- On compile pour en faire un module (gcc -c)
- On peut archiver avec la commande ar sous linux
- On déclare ces fonctions dans le fichier qui va les utiliser en les faisant précéder du mot clef `extern` qui signifie définies dans un autre fichier

Commandes Unix relatives à la compilation

- Gcc : compilation, options très usuelles :
  - -c option pour créer un fichier objet :
    - gcc -c prog.c
  - -o détermine le nom de l'executable :
    - gcc -o c+bokeapointoutnon
  - -W option relative aux avertissement :
    - gcc -o Wall progbugge
  - -O2 option relative à l'optimisation de code :
    - gcc -O2 prog.c
  - -g pour obtenir des informations lors du debuggage
    - gcc -g prog.c
- Gcc : compilation, édition de liens
  - -static pour forcer un linkage static
  - -l nom(s) des librairies à lier
  - -L chemin des librairies
  - -I (i majuscule) chemin des header
    - gcc -l math.so -L /usr/local -I /home/moi/headers tonprog.c

Commandes Unix relatives à la compilation et au chargement

- ldd : affiche les bibliothèques partagées nécessaires à un exécutable
  - ex: ldd prog
- nm : affiche la table des symboles d'un fichier objet ou d'une bibliothèque
- ar: permet de créer des bibliothèques (archives)
- strace : permet de « suivre » le les appels systèmes faits par le pocessus correspondant à la commande ou le programme passé en paramètre (et les signaux reçus)
- cpp permet de connaître les effets des directives du préprocesseur
- Pour profiler un code (connaître les répartitions relatives des temps dans les fonctions)
  - On compile avec l'option -pg
  - On lance l'executable on obtient un fichier gmon.out
  - On interprète ce fichier avec gprof
  - Autre utilitaire GUI Kcachegrind on obtient les traces avec callgrind
- Pour connaître utilisation de la mémoire on utilise valgrind

Makefile

- Un makefile est un fichier qui permet de sérialiser une série de commandes
- Si on a 1 fichier de fonctions aide.c et un programme principal main.c
  - Pour compiler `gcc -c aide.c` puis `gcc main.c -o LeProg2lamortkitue aide.o`
  - Mais `main.c` dépend de `aide.o`
- Un makefile est une suite de cibles (pour lesquelles on précises les dépendances)

```
CFLAGS=-g -O2 -Wall
#Cible 1
Programme : main.c aide.o
    gcc $(CFLAGS) main.c aide.o -o LeProg2lamortkitue

aide.o : aide.c
    gcc -c $(CFLAGS) aide.c

clean :
    rm aide.o
```

Conclusions

- translation => permettre l'exécution d'un module n'importe où
- lien => information permettant à un module d'accéder à un autre
- édition de liens => 2 phases
  - construction de la table des liens
  - remplacement des liens à satisfaire et construction du programme
- bibliothèque => moyen de compléter une liste de modules
  - sans structure -> ordre des modules est important
  - avec structure -> accès par lien utilisable
- édition de liens dynamique => partager les bibliothèques
- recouvrement => diminuer l'encombrement total du programme
- références croisées => savoir *qui utilise quoi*
- chargement => mise en mémoire, machine abstraite, lancement