Cours 3
L'exécution des programmes

1. Le chargement du processus en mémoire

Une des fonctions du système est de **charger** en mémoire les programmes et de leur "passer la main". Un programme en train de s'exécuter en mémoire s'appelle un **processus**. Les opérations à faire sont :

- ¬ Trouver un emplacement libre suffisant en mémoire et l'attribuer au processus.
- ¬ Copier les données du programme disque vers l'emplacement choisi pour le processus
- ¬ Copier les informations système propres au processus (terminal, environnement,..)
- ¬ Enregistrer les informations relatives au processus créé dans une table des processus en cours d'exécution
- ¬ Initialiser les registres pour lancer l'exécution du processus

Cette fonction est réalisée par un appel système. En C, la forme la plus simple pour lancer un processus est la fonction system().

Exemple : on a dans le code C

```
system("cp /home/users/TPINFO/sys1");
```

cp est un programme sur le disque. Le système va lui attribuer un espace mémoire, le copier en mémoire, initialiser stdin, stdout et stderr pour ce processus, recopier la chaîne "cp /home/users/TPINFO/sys1" dans l'espace mémoire du processus, créer un tableau de pointeurs vers les mots de cette chaîne (ici : "cp" et "/home/users/TPINFO/sys1"), placer le nombre de mots (ici : 2) et l'adresse du tableau sur la pile, appeler la fonction main().
Quand on écrit main(int argc, char* argv[]), on récupère le 2 dans argc et le tableau dans argv. Quand on écrit main(int argc, char* argv[], char* arge[]), on récupère en plus un tableau de chînes de caractères appelées **variables d'environnement** (ex: le path). Ces variables seront vues bientôt.

Sur le disque (dans le programme exécutable) il y a :

- Le code
- Les chaînes de caractère (quand on écrit printf("bonjour"); il faut noter le bonjour quelque part en dehors du code)
- Les données globales initialisées (quand on écrit en tète de fichier int tab[3]={0, 1, 2}; il faut noter les valeurs initiales 0, 1, 2 dans le programme)

Dans la mémoire, le processus prend plus de place. Son espace mémoire comporte :

- Une zone de code. Sa taille utile est connue à l'avance.
- Une zone de **données initialisées** (y compris les chaînes de caractère). Sa taille utile est connue à l'avance.
- Une zone de **données non initialisées** (si l'on déclare int tab[1000]; il faut réserver dans la mémoire 4000 octets pour mémoriser le tableau.. Sur le disque, il suffisait de noter que l'on avait besoin de cet espace). Sa taille utile est connue à l'avance, sauf pour l'allocation dynamique (voir 2).
- Une zone de pile. Sa taille utile n'est pas connue à l'avance.

2. Classes de variables et mémoire occupée par le processus

A. Les classes de variables sont définies par leur **portée** et leur **durée de vie**. Les règles pour décider à quelle classe appartient une variable dépendent du langage, celles données ici sont celles du C.

Portée d'une variable : c'est la partie du code source dans laquelle cette variable est connue (accessible).
Si la variable est déclarée dans un bloc (variable **locale**), sa portée va de la déclaration à la fin du bloc.
Si la variable est déclarée en dehors de tout bloc (variable **globale**) sans modificateur ou avec le modificateur **extern**, sa portée inclut de la déclaration à la fin du fichier et tous les autres fichiers du programme (elle est exportée).
Si la variable est déclarée en dehors de tout bloc (variable **globale**) avec le modificateur **static**, sa portée inclut de la déclaration à la fin du fichier (elle n'est pas exportée).

Durée de vie d'une variable : c'est la période de l'exécution du programme pendant laquelle l'espace mémoire qui a été alloué à la variable lui est réservé.
Variable **automatique** : son espace mémoire est libéré dès qu'elle n'est plus accessible. L'espace mémoire pour les variables automatiques est alloué sur la pile.
Variable **statique** : son espace mémoire lui est réservé jusqu'à la fin de l'exécution. L'espace mémoire pour les variables statiques est alloué dans la zone des données (initialisées ou non, selon le cas).

Quelques règles de bon sens :
- ¬ Une variable locale peut être automatique (sans modificateur) ou statique (avec le modificateur **static**)
- ¬ Une variable globale est nécessairement statique (elle est accessible jusqu'à la fin de l'exécution). Le modificateur **static** ne change que sa portée.
- ¬ Quand une variable globale a une portée sur tout le programme, un seul des fichiers qui l'utilisent lui réserve de l'espace mémoire. C'est celui qui la déclare sans modificateur. Les autres fichiers ont besoin de connaître son type, mais ne réservent pas d'espace. C'est ce que veut dire le modificateur **extern**.

Les fonctions en C :
Ce sont des variables contenant du code, mais avec moins de possibilités. Toutes les fonctions sont obligatoirement statiques et globales. Toutes les fonctions sont exportables, sauf si elles ont le modificateur static. Une fonction qui est utilisée dans un fichier où elle n'est pas définie est considérée comme externe par le compilateur. Enfin, une fonction qui est utilisée dans un fichier où elle n'est ni définie, ni déclarée est supposée renvoyer un entier (quand le compilateur ne considère pas cela comme une erreur).

B. Exemple:

```
/* Fichier mymain.c        */
extern int tab[];  /* Ne pas attribuer d'espace à tab. A l'édition de lien, chercher parmi les autres fichiers
                      lequel lui a attribué de l'espace */
int i, j=0;          /* réserver de l'espace pour i en zone de données non initialisées, pour j en zone de
                      données initialisées. i et j sont accessibles dans tout le programme. */
int code(int c);   /* code est définie dans un autre fichier. Elle renvoie un entier.)

/* Fichier util.c*/
int tab[50]; /* Réserve un espace pour 50 entiers non initialisés. tab est accessible dans tout le
                      programme */
static int aux[29]={1234, 515,…} ; /* Réserve un espace pour 29 entiers et l'initialise. aux est accessible
                      dans tout util.c */
int code(int c){ /* code est accessible dans tout le programme. c n'est accessible qu'à l'intérieur de
                      code. */
      static int num ; /* num n'est accessible qu'à l'intérieur de code. Elle est allouée en zone statique
                           initialisée, et initialisée à 0. A chaque appel de code, on retrouve la valeur de
                           num en fin de l'appel précédent. */
      int temp ;        /*  temp n'est accessible qu'à l'intérieur de code. Elle est allouée sur la pile à
                           chaque appel de code, et libérée quand code retourne */
      temp = (num x num) % 29 ;
      num ++ ;          /*  donc a chaque appel de code, num contient le nombre d'appels à code
                           effectués auparavant */
      return(aux[temp] ^ c) ; /* calcule le code */
}
```

C. L'allocation dynamique
Il arive que le programmeur ait besoin d'espace en zone de données pour des variables dont la portée n'est pas locale, mais dont la durée de vie est inférieure à celle du programme ou dont la taille ne peut être estimée à l'avance. En C, l'appel malloc() alloue un bloc d'espace libre en zone de données et renvoie son adresse (pointeur). L'appel free() désalloue ce bloc et le rend disponible pour autre chose. (note : en java, new alloue un bloc ; la désallocation est automatique). A l'heure actuelle, la plupart des systèmes ont la capacité d'agrandir la zone de données pour faire de l'allocation dynamique.

3. Erreurs : détection, protection

Pour mettre au point un programme, il faut savoir reconnaître à quelle étape du cycle programme l'erreur se produit et ce qu'elle veut dire, pour ensuite retrouver sa cause. Le message ne dit jamais quelle est l'erreur de

programmation. Il dit seulement à quel endroit le processus de traduction est bloqué, ou l'exécution déclenche un mécanisme de sécurité.

A. Erreurs à la compilation : ce sont soit des erreurs de syntaxe (le compilateur ne sait pas quoi faire de ce qui est écrit, ex : a =+ 3;) , soit des incohérences (ex: *int c ; char c ;* ou bien *char f(void); f(5);* ou f n'attend pas d'argument). Quand il n'y a pas d'erreur à la compilation, on peut générer les fichiers objets.

B. Erreurs à l'édition de liens : l'éditeur de lien ne peut pas résoudre les références externes. Par exemple : plusieurs fichiers déclarent en variable globale *extern int ref ;* mais aucun ne déclare *int ref ;* Ou bien un fichier appelle la fonction *round() ;* mais aucun ne la définit. Ou encore le fichier qui appelle *round()* ne l'a pas déclarée (elle est donc supposée renvoyer un entier), mais elle est définie *float round(float f) {..}* dans un autre fichier (ce peut être le cas quand on oublie un #include en appelant une fonction de bibliothèque).

C. Erreurs à l'exécution : on arrive à générer un exécutable, mais quand on le lance il sort avec une erreur système, ou il ne termine pas, ou il donne des résultats imprévisibles. Ce sont les erreurs les plus difficiles à corriger. En général, elles sont dues soit à une erreur d'algorithmique (boucle infinie, débordement au delà des bornes d'un tableau), soit à une insuffisance d'espace mémoire disponible (demande d'allocation dynamique au delà de la mémoire disponible, débordement de pile par excès d'appels de fonctions emboités).