

Programmation système II

Exécutables et compilations

DUT 1^{re} année

Université de Marne La vallée

Exécutables

Processus

Compilations

Fichier d'en-tête

Préprocesseur

Makefile

Qu'est ce qu'un exécutable ?

- ▶ Un fichier
 - ▶ Ayant des droits d'exécution
 - ▶ Contenant du code machine ou des données à interpréter
 - ▶ 2 premiers octets (Magic Number) du fichier permettent de savoir comment traiter le contenu

Quel fichier est exécuté ?

- ▶ Fichier dont le chemin d'accès est donné en console
 - ▶ `/bin/ls` (chemin absolu)
 - ▶ `./ls` dans le répertoire `/bin` (chemin relatif)

- ▶ **Pourtant on peut taper `ls` sans préciser le chemin ?**

- ▶ Fichiers trouvés dans une liste de répertoire

- ▶ Définis dans la variable d'environnement **PATH**

```
nicolas@lancelot:~$ echo $PATH
/usr/lib/lightdm/lightdm:/usr/local/sbin:
/usr/local/bin:/usr/sbin:/usr/bin:/sbin:
/bin:/usr/games:/opt/real/RealPlayer
```

- ▶ La commande UNIX `which` permet de récupérer le chemin des exécutables

```
nicolas@lancelot:~$ which ls
/bin/ls
```

Il faut avoir les droits d'exécution sur l'ensemble des répertoires du chemin.

Exécution

- ▶ Le système associe un processus au programme
 - ▶ Un programme peut être associé à plusieurs processus (commande `fork`, un programme peut créer des processus enfant (PID, PPID)).
- ▶ Un processus est associé à :
 - ▶ Une identité
 - ▶ Un espace mémoire (virtuel) propre au processus
 - ▶ Des droits dépendants de l'utilisateur et du propriétaire du fichier
 - ▶ Diverses autres choses

Comment produire un fichier exécutable ?

- ▶ Écrire en binaire !!!
- ▶ Écrire en assembleur et générer le binaire :
 - ▶ non portable
 - ▶ difficile à écrire
 - ▶ difficile à maintenir
- ▶ Utilisation d'un langage haut niveau et d'un compilateur vers l'architecture cible

Un programme C

"hello world" en langage C

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("hello _world\n");
6
7     return 0;
8 }
```

Un programme assembleur

"hello world" pour assembleur x86, sous Linux, écrit pour l'assembleur NASM

```
section .data
    helloMsg:      db 'hello world!',10
    helloSize:     equ $-helloMsg
section .text
    global _start
_start:
    mov eax,4      ; Appel système "write" (sys_write)
    mov ebx,1      ; Description du flux, 1 pour STDOUT
    mov ecx, helloMsg ; Adresse de la chaîne a afficher
    mov edx, helloSize ; Taille de la chaîne
    int 80h        ; Exécution de l'appel système
                  ; Sortie du programme
    mov eax,1      ; Appel système "exit"
    mov ebx,0      ; Code de retour
    int 80h
```

Binaire: langage machine

"hello world" pour une architecture x86

```
10111010 00010000 00000001 10110100
00001001 11001101 00100001 00110000
11100100 11001101 00010110 10111000
00000000 01001100 11001101 00100001
01001000 01100101 01101100 01101100
01101111 00100000 01010111 01101111
01110010 01101100 01100100 00100001
00100100
```

Soit en hexadécimal :

```
BA 10 01 B4 09 CD 21 30 E4 CD 16 B8 00 4C CD 21 48 65 6C
6C 6F 20 57 6F 72 6C 64 21 24
```

Compilation : version courte

Compilateur

hello.c $\longrightarrow$ gcc $\longrightarrow$ hello

```
nicolas@lancelot:~/test$ ls
hello.c
nicolas@lancelot:~/test$ gcc -o hello hello.c
nicolas@lancelot:~/test$ ls
hello  hello.c
nicolas@lancelot:~/test$ ./hello
Hello world
nicolas@lancelot:~/test$
```

Compilation : version longue

```
$ cpp hello.c hello.i #preprocesseur
```

```
$ cc1 -s hello.i #compilateur
```

```
$ as hello.s -o hello.o #assembleur
```

```
$ ld -o hello hello.o /usr/lib/crt1.o \  
/usr/lib/crti.o /usr/lib/crtn.o -lc \  
-dynamic-linker /lib/ld-linux.so.2 # Edition de lien
```

Pourquoi la ligne d'édition de liens est si longue ?

```
$ ld -o hello hello.o /usr/lib/crt1.o \  
/usr/lib/crti.o /usr/lib/crtn.o -lc \  
-dynamic-linker /lib/ld-linux.so.2 # Edition de lien
```

- ▶ Elle précise le nom de l'exécutable à créer et le nom du fichier objet contenant mon programme ;
- ▶ elle précise le nom d'autres fichiers objets contenant des fonctions nécessaires à la construction du programme exécutable ;
- ▶ elle précise le nom d'une bibliothèque (library) contenant des fonctions de base (printf(), etc...) ;
- ▶ elle dit que ces fonctions doivent être liées dynamiquement à l'exécution.

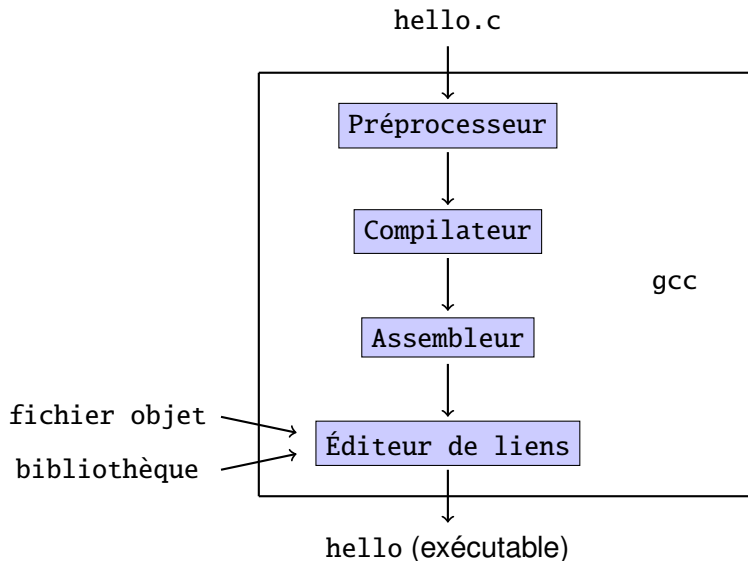
Informations sur les exécutables...

Les commandes `file` et `nm` donnent des informations sur les exécutables.

```
$ test>file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.24,
BuildID[sha1]=0x783a72742005bb021a7353a687547932b1ca5bea,
not stripped
$ nm hello
08049f28 d _DYNAMIC
08049ff4 d _GLOBAL_OFFSET_TABLE_
080484bc R _IO_stdin_used
          w _Jv_RegisterClasses

...
08048320 T _start
0804a014 b completed.6159
0804a00c W data_start
0804a018 b dtor_idx.6161
080483b0 t frame_dummy
080483d4 T main
          U puts@@GLIBC_2.0
```

Compilation : version longue



Un programme avec fonction extérieure

```
1 int main(void) {  
2     printf("Racine de 4 = %d\n", sqrt(4));  
3     return 0;  
4 }
```

```
lancelot-~/test>make test  
cc      test.c  -o test  
... Plein de warning ...  
lancelot-~/test>./test  
Racine de 4 = 0
```

Le compilateur ne possède pas suffisamment d'informations pour une utilisation correcte de `sqrt`.

Un programme avec fonction extérieure

- ▶ Les types de `printf` et `sqrt` sont déduits de leur utilisation.
- ▶ Il faut rajouter les déclarations de `printf` et `sqrt`.
 - ▶ à la main

```
extern int printf (char * format, ...);
extern double sqrt (double v);
```
 - ▶ En utilisant les fichiers `.h` d'en-tête (header) associés à `printf` et `sqrt`.

```
#include <stdio.h>
#include <math.h>
```
- ▶ Il faut ajouter des options de compilation pour vérifier que toutes les fonctions externes sont bien déclarées et bien utilisées.
`-Wall -ansi`

Un programme avec fonction extérieure

Voici un fichier de compilation makefile minimal pour donner des indications à make.

1 CFLAGS=-ansi -Wall

```
lancelot-~/test>make test
cc -ansi -Wall    test.c    -o test
test.c: In function 'main':
test.c:2:3: attention : implicit declaration of function 'printf'
test.c:2:3: attention : incompatible implicit declaration
test.c:2:3: attention : implicit declaration of function 'sqrt'
test.c:2:33: attention : incompatible implicit declaration
test.c:2:3: attention : format '%d' expects argument of type 'int',
               but argument 2 has type 'double'

lancelot-~/test>./test
Racine de 4 = 0
```

Les flags précisent les actions requises pour éviter les erreurs.
Ici printf et sqrt ne sont pas des fonctions connues.

Éliminer tous les attentions!

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main(void){
5     printf("Racine de 4 = %f\n", sqrt(4));
6     return 0;
7 }
```

```
lancelot-~/test>make test
cc -ansi -Wall    test.c    -o test
lancelot-~/test>./test
Racine de 4 = 2.000000
```

Les flags ne produisent aucun "Warning" et le résultat est maintenant cohérent.

Comment sont ajoutées les déclarations du fichier d'en-tête ?

Par le préprocesseur `cpp` !

Le préprocesseur gère les commandes commençant par le caractère `#` généralement placé en haut des fichiers sources ou dans les fichiers d'en-tête (en `.h`).

- ▶ `#include <foo.h>` recherche le fichier d'en-tête de la bibliothèque `foo` et recopie le contenu du fichier avant d'envoyer au compilateur.
- ▶ `#include "foo.h"` recherche le fichier `"foo.h"` et recopie le contenu du fichier avant d'envoyer au compilateur.
- ▶ `#define FOO BAR` recherche toutes les occurrences de `FOO` et les remplace par `BAR`. C'est équivalent à exécuter la commande `sed "s/FOO/BAR/g" fichier_source.c` du `bash` UNIX.

Où sont trouvés les fichiers d'en-tête ?

- ▶ Dans un ensemble de répertoires prédéfinis.

```
nicolas@lancelot:~$ cpp --verbose
```

```
...
```

la recherche pour `#include <...>` débute ici:

```
/usr/lib/gcc/i686-linux-gnu/4.6/include
```

```
/usr/local/include
```

```
/usr/lib/gcc/i686-linux-gnu/4.6/include-fixed
```

```
/usr/include/i386-linux-gnu
```

```
/usr/include
```

Fin de la liste de recherche.

- ▶ Dans des répertoires précisés avec l'option `-I ()`
- ▶ Si le nom de fichier à inclure est entre guillemets ("). Le fichier est également recherché dans le répertoire courant

Que contient un fichier d'en-tête ?

- ▶ **Jamais de code exécutable!**
- ▶ Des demandes d'inclusions d'autres fichiers .h
`#include <module.h>`
- ▶ Des déclarations de prototypes de fonctions externes
`extern int fclose (FILE * stream);`
- ▶ Des déclarations de variables globales externes
`extern struct _IO_FILE *stdin;`
- ▶ Des définitions de constantes ou de macros
`#define EOF (-1)`
`#define getc(_fp) _IO_getc (_fp)`
- ▶ Des définitions de types
`typedef struct _IO_FILE FILE`

Attention aux macros...

Les définitions de constantes posent rarement des problèmes.

```
1 #include <stdio.h>
2 #define PI 3.14
3
4 int main(void){
5     float r = 5.0;
6
7     printf("Perimetre du cercle : %f\n", 2*r*PI);
8     printf("Aire du cercle : %f\n", r*r*PI);
9     return 0;
10 }
```

```
nicolas@lancelot:~/test$ make test
cc      test.c      -o test
nicolas@lancelot:~/test$ ./test
Perimetre du cercle : 31.400000
Aire du cercle : 78.500000
```

Attention aux macros...

Les macros à paramètres présentent parfois des pièges.

```
1 #include <stdio.h>
2 #define CARRE(x) x*x
3
4 int main(void){
5
6     printf("Carre de 10 : %d\n", CARRE(10));
7     printf("Carre de 3+7 : %d\n", CARRE(3+7));
8     return 0;
9 }
```

```
nicolas@lancelot:~/test$ make test
```

```
cc      test.c  -o test
```

```
nicolas@lancelot:~/test$ ./test
```

```
Carre de 10 : 100
```

```
Carre de 3+7 : 31
```

Attention aux macros...

Le préprocesseur fait le travail suivant :

```
cpp test.c
```

```
...
```

```
int main(void){
```

```
    printf("Carre de 10 : %d \n", 10*10);
```

```
    printf("Carre de 3+7 : %d \n", 3+7*3+7);
```

```
    return 0;
```

```
}
```

La bonne macro est donc ici :

```
1 #define CARRE(x) (x)*(x)
```

Cas pratique

Soit le fichier source test.c

```
1 #include <stdio.h>
2
3 int main(void){
4     printf("La somme de 4 et 7 est %d\n", somme(4,7));
5     return 0;
6 }
7
8 int somme(int a, int b){
9     return a+b;
10 }
```

```
nicolas@lancelot:~/test$ gcc -o test test.c -Wall -ansi
test.c: In function 'main':
test.c:5:5: attention : implicit declaration of function 'somme'
```

Le compilateur nous informe que la fonction somme n'est pas connue lors de sa première rencontre.

Cas pratique

Première Solution :

```
1 #include <stdio.h>
2
3 int somme(int a, int b){
4     return a+b;
5 }
6
7 int main(void){
8     printf("La somme de 4 et 7 est %d\n", somme(4,7));
9     return 0;
10 }
```

On fait en sorte que les fonctions soit définies avant leur utilisation. C'est facile si on a qu'un seul fichier dans notre projet et qu'il existe un ordre d'apparition relativement clair des fonctionnalités.

Cas pratique

Seconde Solution :

```
1 #include <stdio.h>
2
3 int somme(int a, int b);
4
5 int main(void){
6     printf("La somme de 4 et 7 est %d\n", somme(4,7));
7     return 0;
8 }
9
10 int somme(int a, int b){
11     return a+b;
12 }
```

On introduit en début de fichier (après les `#include` et `#define`) les prototypes des fonctions à venir.

Cas pratique

Encore mieux :

On utilise un fichier d'en-tête : test.h

```
1 #include <stdio.h>
2
3 int somme(int a, int b);
```

Puis un fichier code:

```
1 #include "test.h"
2
3 int main(void){
4     printf("La somme de 4 et 7 est %d\n", somme(4,7));
5     return 0;
6 }
7
8 int somme(int a, int b){
9     return a+b;
10 }
```

Makefile

Comment simplifier la compilation d'un projet de manière à ce que l'utilisateur final n'est pas à connaître les règles de compilation ?

On constitue un makefile :

```
CFLAGS = -Wall -ansi
```

```
test: test.c test.h  
    gcc -o test test.c test.h $(CFLAGS)
```

```
clean:  
    rm test
```

Un makefile se place à la racine d'un projet. Il contient les règles nécessaires à la compilation des sources pour obtenir les exécutables.

Utilisation du Makefile

L'utilitaire make d'UNIX se charge d'aller chercher les règles de compilations pour construire le projet.

```
nicolas@lancelot:~/test$ ls
makefile  test.c  test.h
nicolas@lancelot:~/test$ make
gcc -o test test.c test.h -Wall -ansi
nicolas@lancelot:~/test$ ls
makefile  test  test.c  test.h
nicolas@lancelot:~/test$ make clean
rm test
nicolas@lancelot:~/test$ ls
makefile  test.c  test.h
```

Utilisation du Makefile

L'utilitaire `make` gère à son appel ce qui mérite d'être recompilé! Lorsque la date de dernière modification des sources est postérieure à l'âge des exécutables correspondant, il recompile les parties concernées.

```
nicolas@lancelot:~/test$ make
gcc -o test test.c test.h -Wall -ansi
nicolas@lancelot:~/test$ make
make: << test >> est à jour.
nicolas@lancelot:~/test$ touch test.h
nicolas@lancelot:~/test$ make
gcc -o test test.c test.h -Wall -ansi
nicolas@lancelot:~/test$ make
make: << test >> est à jour.
```