CRO: C langage and programming Roots

tanguy.risset@insa-lyon.fr
Lab CITI, INSA de Lyon
Version du January 28, 2019

Tanguy Risset

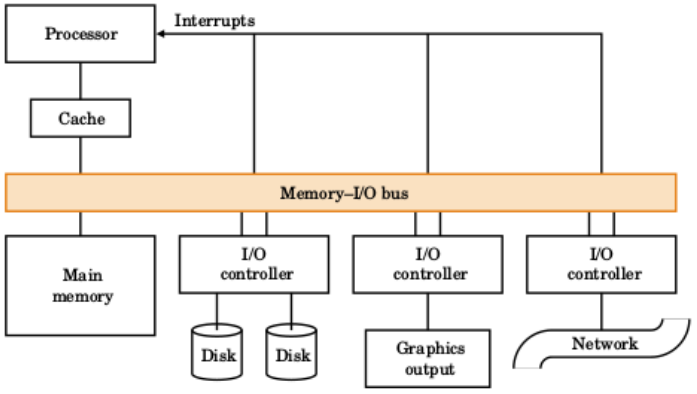
January 28, 2019

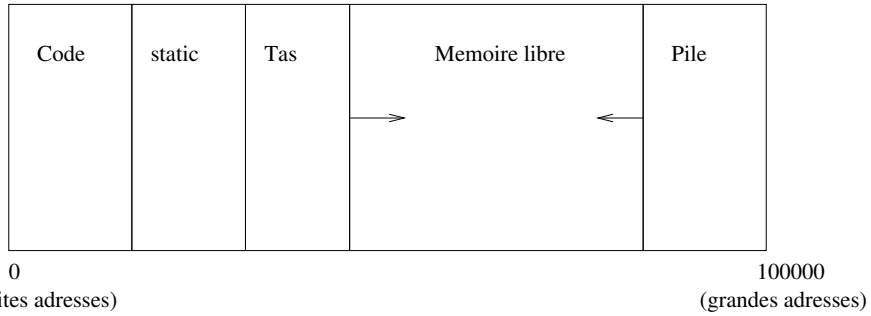
Table of Contents

Rappels sur l'architecture d'un ordinateur

- Un ordinateur de bureau est composé (au moins):
  - D'un processeur
  - D'une mémoire (dite *vive*: rapide et non rémanente)
  - D'un espace de stockage (disque dur: lent, rémanent)
  - De périphériques d'entrée/sortie (écran, claviers, etc.)
- Principe du processeur programmable:
  - Le processeur lit un programme en mémoire (programme *exécutable*, dépendant du type de processeur).
  - En fonction de ce programme
    - Il lit ou écrit des données en mémoire à une certaine *adresse mémoire* (nombre entier sur 32 bits)
    - Il effectue des calculs entre ces données

Rappels d'architecture



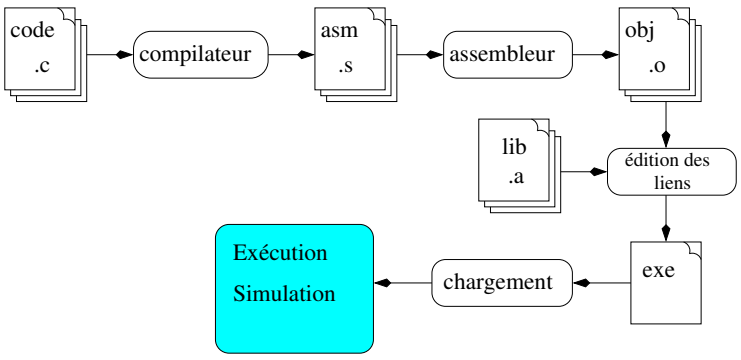


Architecture vue du programmeur

- Les systèmes modernes permettent
  - D'exécuter plusieurs programmes indépendant en parallèle (processus)
  - D'accéder à un espace mémoire plus grand que la mémoire physique disponible (mémoire virtuelle)
- Pour le programmeur: tout cela est transparent
  - Un seul programme s'exécute avec une mémoire très grande disponible
- La mémoire vue du processeur contient:
  - Le code à exécuter
  - Les données statiques (taille connue à la compilation)
  - Les données dynamiques (taille connues à l''exécution: le tas, et l'espace necessaire à l'exécution elle-meme: la pile)
- Le programmeur lui ne voit que les données (statiques et dynamiques)

Processus de compilation

- le processus complet va traduire un programme C en code exécutable (le chargement et l'exécution auront lieu plus tard).



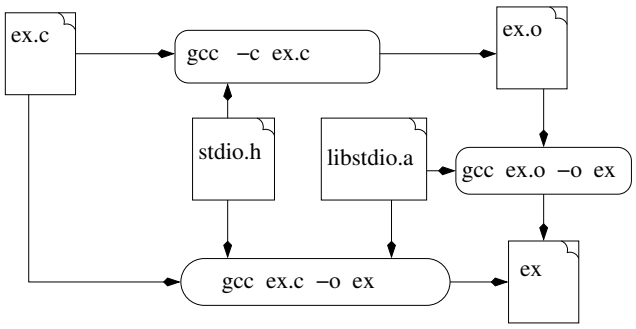
- On nomme souvent *compilation* l'ensemble compilateur+assembleur
- Le compilateur gcc inclut aussi un assembleur et un éditeur de lien (accessibles par des options)

Votre processus de compilation

- Le programmeur:
  - Écrit le programme C: ici le dans le fichier `ex.c`
  - Compile vers un programme objet `ex.o`
  - Fait l'édition de lien pour générer l'executable `ex`

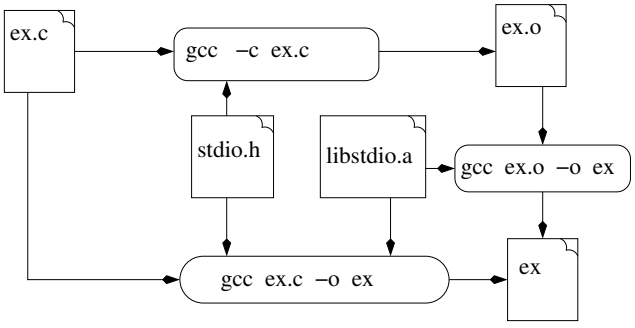
contenu de `ex.c`

```
#include <stdio.h>

int main()
{
  printf("hello World\n");

  return(0);
}
```



Dénomination à retenir

- Le programme C que vous écrivez est le *programme source* (`ex.c`)
- Le programme source est compilé en un *programme objet* par le compilateur (`ex.o`)
- Le programme source inclut un fichier d'en-tete (`stdio.h`) pour vérifier les noms des fonctions externes utilisées (`printf`)
- le *programme executable* est construit par l'éditeur de liens grâce aux programmes objets et aux bibliothèques.
  - Les *bibliothèques* (*library* en anglais) sont des codes assembleurs (i.e. déjà compilés) de fonctions fréquements utilisées. Ici la bibliothèque utilisée est `libstdio.a` qui contient le code de la fonction `printf`



Outils pour le développement logiciel

- Méthodes
  - Réfléchir avant d'agir !
  - règles de développement (programmation agile, programmation par test)
  - Language objet: C++, Java
- Outils
  - Commentaire en Doxygen
  - Gestionnaire de compilation : `make`
  - Debugger : `gbd, ddd, ....`
  - Environnements de développement (intègrent tout): `eclipse, VisualC++`

Table of Contents

Makefile et la commande: `make`

- `make` est un outils d'aide à la compilation, indispensable au développement sous Linux.
- L'équivalent de cet utilitaire est intégré dans tous les environnements de développement, quel que soit le langage de programmation utilisé (Eclipse, Visual Studio etc.)
- Nous expliquons ici les principe de base permettant l'utilisation d'un `Makefile` (fichier de configuration de l'utilitaire `make`).
- Si l'on souhaite réaliser un projet C de taille importante (et surtout portable sur tout type de machine), il est nécessaire d'utiliser des outils de génération de Makefile.
- Les *autotools* ont beaucoup été utilisés (`autoconf`, `automake`
- Aujourd'hui la préférence semble aller à `cmake`

Make

- À l'aide d'un fichier de description, l'utilitaire `make` crée une suite de commandes qui seront exécutées par le shell d'unix.
- Utilisé principalement pour le développement logiciel mais peut aussi servir à de nombreux projets: production de gros documents, mise en place d'expérimentations etc...
- Le principal avantage est de ne pas tout recompiler lorsque l'on a changé un seul fichier.
- Lorsque l'on tape `make`, l'utilitaire recherche dans l'ordre un fichier `makefile` puis un fichier `Makefile`, on peut lui indiquer d'utiliser un autre fichier avec l'option `-f`:
  `make -f monFich.mk`

Makefile: Cibles et dépendances

- Dans un fichier `Makefile`, une *cible* est un objet qui va être produit (par exemple un exécutable) par une règle de production.
- On produit une cible particulière à partir de fichiers particuliers, ces fichiers sont les *dépendances* de la cible.
- Ces dépendances peuvent être à leur tour des cibles d'autres règles de production.
- Exemple (Attention, le caractère `tab` est nécessaire avant les règles de production):

```
all: main

main:  main.o
       gcc main.o -o main

main.o: main.c type.h
      gcc -c main.c -o main.o
```


Makefile: définitions de variables

- Les variables (ou macros) de `make` ont un comportement similaire aux macros de C définies par `#define`.
- La syntaxe de la définition de variable:
  `VAR1=quelquechose etencorequelquechose`
- On utilise la variable en mettant `$(VAR1)` ou `${VAR1}`
- Exemple typique d'utilisation:

```
TARGET_BIN = ../bin
OBJ        = main.o utils.o truc.o

main:  $(OBJ)
       gcc -o main $(OBJ)
       mv main $(TARGET_BIN)
```

- la commande `make main` exécute:

```
gcc -o main main.o utils.o truc.o
mv main ../bin
```

Makefile: Variables prédéfinies

- Un certain nombre de variables sont fournies par défaut par `make`
- `SHELL` indique sous quel shell est exécuté la commande (en général `sh`
- `CC` indique le compilateur C utilisé par le système.
- `CFLAGS` indique les options par défaut du compilateur.
- `LDFLAGS` indique les options par défaut de l'éditeur de lien.
- On peut visualiser l'ensemble des réglages par défaut de `make` avec l'option `-p`

Makefile: substitution et variables dynamique

- Si on a défini une variable:
  `SRC = main.c utils.c truc.c`
- Alors l'expression `${SRC:.c=.o}` vaudra:
  `main.o utils.o truc.o`
- Certaines variables sont positionnées dynamiquement lors de l'évaluation d'une règle, par exemple `$@` est le nom de la cible de la règle:

```
main: main.o utils.o truc.o
	${CC}  -o $@  main.o utils.o truc.o
```

- `$<` est le nom de la première dépendance sélectionnée

Makefile: règles implicites

- La compilation de programme C suit toujours le même schéma, par exemple: `gcc -c monprog.c -o monprog.o`
- On peut définir des règles par défaut pour les cibles ayant un suffixe particulier. Pour cela on indique à make quels sont les suffixes intéressants et on spécifie la règle implicite. Par exemple

```
.SUFFIXES : .o .c
.c.o :
	${CC} ${CFLAGS} -c $< -o $@
```

- On spécifie alors uniquement les dépendances pour chaque fichier .o

```
.SUFFIXES : .o .c
.c.o :
	${CC} ${CFLAGS} -c $< -o $@
main.o: main.c type.h
```


Makefile: Exemple

```
CC      =gcc
EXE     =ex1 ex1.1 ex1.2 ex1.3 ex2.1 ex3.1

SRC=$(EXE:=.c)
OBJ=$(SRC:.c=.o)

all: $(EXE)

tar:
	make clean
	cd ..;tar -cvf TD1_corr.tar TD1/ex*.c TD1/test*.in TD

.SUFFIXES : .o .c
.c.o :
	${CC} ${CFLAGS} -c $< -o $@
clean:
```

Makefile: Exemple

Résultat de l'exécution de la commande `make`

```
gcc      ex1.1.c   -o ex1.1
gcc      ex1.2.c   -o ex1.2
gcc      ex1.3.c   -o ex1.3
gcc      ex2.1.c   -o ex2.1
gcc      ex3.1.c   -o ex3.1
```


Plus d'info sur les Makefile

- Un tutorial en Français (developpez.com): `https://gl.developpez.com/tutoriel/outil/makefile/`
- Un autre (Jean-Claude Iehl) `http://perso.univ-lyon1.fr/jean-claude.iehl/Public/educ/Makefile.html`
- Celui de l'enswiki de l'Ensimga (connaissez vous ce site?): `https://ensiwiki.ensimag.fr/index.php?title=Makefile`