Systèmes Informatiques 1

Chapitre 1 : Introduction

Section 1.1 : Introduction

1.1.1 Composition d'un système informatique

Le système informatique le plus simple est **un processeur** (*cpu*) et **une mémoire**. Ce prosesseur est capable de : *lire et écrire* des l'informations en mémoire et de *réaliser des calculs*.

Systèmes Unix

Unix est un nom générique donné à une famille de systèmes d'exploitations. On y retrouve MacOS, FreeBSD et Linux qui sont les plus utilisés.

Un système unix est composé de trois grands types de logiciels:

1. **Le noyau du système d'exploitation** : chargé au démarage de la machine, il se charge de toutes les interactions entre les logiciels et le matériel.

2. **Les librairies** :Nombreuses, elles facilitent l'écriture et le développement d'applications

3. **Les programmes utilitaires** : utilisés pour résoudre une série de problèmes
   API signifie *Application Programming Interface*

   /**usr** : utilitaires et librairies installées sur le système
   /**bin** et /**sbin** : utilitaires de base nécéssaire à l'admin. syst.
   /**tmp** : fichiers temporaires (effacé au redémarrage)
   /**etc** : configuration du système
   /**home** : répertoire personnel des utilisateurs
   /**dev** : fichiers spéciaux
   /**root** : administrateur système

En Unix une application est composée de un ou plusieurs processus.
Un processus : ensemble cohérent d'instructions qui utilisent une partie de la mémoire et sont exécutées sur un processeur.

Les processus peuvent utiliser des ressources en mémoire. lorsque le processus va se terminer, il va libérer ces ressources et retourner un entier au processus parent (0 si Ok sinon autre chose)

Shell

Généralement appelée console ou terminal. Un **shell** est un programme qui à été spécialement conçu pour faciliter l'utilisation d'un système Unix via le clavier. Sa puissance vient de sa capacité à écrire des commandes enchaînées : <,>,>>,|
Une **pipe** ;) est une redirection de la sortie standartd d'un programme vers l'entrée std d'un autre sans passer apr un fichier intermédiaire.
cat : affiche le contenu d'un fichier sur la sortie Standard
sort : trie les lignes d'un fichier
mv : utilitaire pour renommer ou déplacer un fichier ou dossier.
head et **tail** : extrait le debut et la fin d'un fichier.
wc : compte le nombre de (lignes, mots, caractères).
sort : trie le fichier par ordre alphabétique
uniq : retire les doublons *(Attention : fichier trié au préalable)*
tar : permets de regrouper des fichiers dans une archive (fonctionne souvent avec gzip)
gzip/**gunzip** : compression / décompression d'un fichier .gz
cp : copie un fichier ou dossier (-r pour les dossiers).
rm : efface un fichier ou dossier.
mkdir : crée un répertoire.
rmdir : efface un répertoire vide.
cd : change le répertoire courant.
pwd : affiche le répertoire courant.
./prog : est utilisé pour lancer le programme prog.
grep : utilitaire permettant d'extraire d'un fichier les lignes qui contiennent ou non une chaine de caractère passée en argument.
echo "Blabla" >> file.txt copie le texte Blabla dans le fichier file.txt.
man :lire les pages de manuel d'un système Unix.

Un script bash commence par

```
#!/bin/bash
$# # nombre d args
$1 # arg1 ..
$@ # liste des args
if [Cond]; then ...fi
exit 0
```

```
$i-eq $j # vrai si les deux variables sont différentes
$i -eq $j # vrai si les deux variables sont équivalentes
$s = $t #vrai quand les deux chaines de caractères sont équivalentes
$i -lt $j # vrai si i est strictement inférieure à j
$i -ge $j # vrai si i est suppérieure inférieure à j
-z $s #vrai si la variable est vide
```

Chapitre 2 : Langage C

Section 2.1 : Le langage C

Un **langage** permets d'écire des programmes qui seront *compilés* pour être *exécutables* par le processeur (binaires = language Machine). L'**assembleur** quand à lui est un langage proche de la machine pour être le plus rapide possible. (Chaque famille de processeur à son language d'assemblage qui lui est propre. On converti le **langage d'assemblage** en langage Machine via un *assembleur*).
Le C (inventé dans les années 70), pour le système Unix, est rapide et permets d'interagir directement avec le matériel.

2.1.1 Programme de base

``` C
#include

int main ( intarg c , char * argv [ ] ){
printf ( "Hello World!") ;// affiche sur la sortie standard le message
return 0; //Un programme retourne toujours une valeur (en C : return ou exit ).
}
```

```
#### 2.1.2 Compilation
La compilation du fichier hello.c en un exécutable s'effectue sur un système Unix c
```

gcc -Wall -o hello hello.c

```
l'argument *-Wall* affiche tous les warnings
l'argument *-o hello* donne le nom de sortie de l'exécutable


*** update 21/02/18

#### 2.1.3 Préprocesseur
Au moment de la compilation, le compilateur va exécuter les directives préprocesseu
* **#define** : permets la définitionn de substitution et est fréquemment utilisé p
* **#include** : macro permettant l'inclusion de fichiers.
  * *< stdio.h >* : librairie de fonctions standard permettant d'interragir avec le
  * *< stdlib.h >* : fonction et constantes de la librairie standard
```

C
#include <...> //ajoute les librairies au moment de la compilation
#define ZERO 0 // replace ZERO par 0 au moment de la compilation

```
#### 2.1.4 Strings
Les strings sont des tableaux de caractères. en C, ils se terminent par la valeur ''
```

C
char string[10];
string[0]='j';
string[1]='a';
string[2]='v';
string[3]='a';
string[4]='\0';
printf("le string est : %s \n", string);
// %s fait référence à la variable string qui est un string
// %d fait référence à une variable de type int
// %c fait référence à une variable de type char

```
#### 2.1.5 Constructons Syntaxiques
```

C
if (COND){}else{}
while(COND){} //if cond false do nothing
do {} while(COND); //if cond is false then don't repeat
for(INIT;COND;INCR){}

```
#### 2.1.6 Les arguments d'un programme
un programme peut (comme en java) retourner un int (cfr infra) ou void (rien)
```

C
#include
#include
int main(int argc, char *argv[]){
int i;

for(i=0;i<argc;i++){
printf("argument[%d] : %s\n", i, argv[i]);
}
return EXIT_SUCESS;
}

```
#### 2.1.7 Le Manuel
accessible via la commande man
1. Utilitaire disponible pour tous les utilisateurs
2. Appels systèmes en C
3. Fonctions de la librairie
4. Fichiers spéciaux
5. Formats de fichiers et conventions pour certains types de fichiers
6. Jeux
7. Utilitaires de manipulation de fichiers textes
8. Commandes et procédure de gestion du système

### Section 2.2 : Types de données
Les types de données et leur représentation en mémoire
- decimal : 123
- Binaire : *0b* + 1111011
- Octal : *0* + 173
- Hexadécimal :*0x* + 7B

On peut obtenir la taille en mémoire d'un type de données avec
```

c
sizeof(DATA_TYPE)

```
#### 2.2.1 Les nombres entiers
Les **nombres signés** sont représentés sous la forme : Signe (négatif si = 1) - Nor
Les **nombres non-signés** sont représentés sous la forme binaire std 5 = 2^2 + 2^0

#### 2.2.2 IEEE 754
Le **Standard IEEE 754** est une représentation des nombres réels sous forme : Signe
il existe la single et la double précision (respectivement  32 et 64 bits)

#### 2.2.3 Les tableaux
Dans les anciennes version du langage langage C, les tableaux étaient de taille fixe
**Attention** pas de Tab.length en C  il faut donc prévoir de garder la taille du ta
```

C
#define N 10 // taille du tableau
float vecteur[N]
float matrice[N][N]

int tab[3] = {1,2,3};

```
*attention pas de buffer overflow car pas d'exception**

#### 2.2.4 Caractères et chaines de caractères
Le langage C n'integre pas d'office les boolean et les strings. En C, les strings s
```

C
char string[20]= "text";
printf("%s \n",string);

```
Les lettres étant des char (ASCII de 7 ou 8 bits) stockés sous la forme d'entiers,

| Encodage | Particularité |
| ------------- |: -------------: |
| ASCII | caractère Anglais |
| ISO8859 | Latin avec Accents |
| UNICODE | Tous les caractères dans toutes les langues |

Il n'existe pas de mécanisme d'exception en C. Celà pose des problèmes sécurité & d
Une chaine de caractères se termine toujours par un '\0' ( equivalent à 0)

#### 2.2.5 Les Pointeurs
En C, le programmeur peut interragir directement avec la mémoire où les données qu'

La **mémoire** est une zone qui est définie et accessible via son adresse.
Un **pointeur** est une variable contenant l'adresse d'une autre variable.
```

c
&var // adresse à laquelle une variable est stockée
var // variable en mémoire
*ptr // récupère la valeur à l'adresse du pointeur

```
-----
En C, contrairement au java. Il n'y a pas de garbage collection qui retire de la mé
----
#### 2.2.6 Les structures
En C, contrairement au java (et autres langages orientés objet) on ne peut pas crée

Une **structure** est une combinaison de différents types de données simples ou str
Dans les premières version du langage, les structures avaient une taille fixe
```

c
struct NOM-STRUCTURE
int VARIABLE1;
int VARIABLE2;
}
struct NOM-STRUCTURE NOM-INSTANCE = {1,2}; // crée une instance
NOM-INSTANCE.VARIABLE1= 2 // accède directement à la variable en mémoire
(* ptr).x // accède à l'élèm x du ptr
ptr->x //idem supp

```
#### 2.2.7 Les Alias
On peut redéfinir des noms de structures :
```

c
typedef int ENTIER; //redéfini int par entier

```
On peut les utiliser pour  :
- la portabilité de l'app
- diminuer la taille des identifiants
- redéfinir des pointeurs (attention car un ptr reste un ptr )

#### 2.2.8 Les fonctions
Les fonctions sont des découpes simples de tâches complexes.
On peut définir une fonction comme suit :
```

c
type_de_retour nom_fonction(type_var_1 nom_var_1, type_var_2 nom_var_2){…} // syntaxe Générale
void hello(){…} //fonction sans arguments et sans valeur de retour
int hola(int age){…} //fonction avec argument qui retourne un int
int main(int argc, char *argv[]){…}// fonction main

```
La fonction main est la fonction principale du programme. elle est obligatoire

**Déclaration** : Indique au compilateur le type des arguments et le type de la val

**Définition** : Le corps de la fonction est spécifiée  dans la déclaration ou dans

En C les fonctions et les pointeurs peuvent être utilisés en argument.


#### 2.2.9 Manipulation bits
```

c
r = ~a; //négation bit à bit
r = a & b; // conjonction bit à bit
r = a | b; // disjonction bit à bit
r = a ^ b; // xor bit à bit
a = n >> B // décale bits n de B bits

```
### Section 2.3 : Declarations
Les variables sont définies par leur portée. **La portée d'une variable** peut-être

Les variables globales sont des variables accessibles de partout dans le programme.

Pour définir des constantes on peut :
```

#define M_PI 3.14159265 //préprocesseur
const double pi=3.14159265 // constante

```
Dans les premières versions de C on devait définir les variables au début de chaque

### Section 2.4 : Unions et énumérations
#### 2.4.1 énumérations
**enum** est utilisé pour définir un type de données énumérées. càd un nombre fixe d
```

C
typedef enum{
monday, tuesday, wednesday, thursday, friday, saturnday, sunday
}day;
day jour = monday;

```
#### 2.4.2 Unions

**union** permet de réserver une zone en mémoire pour stocker plusieurs types de va
```

C
union u_t{
int i;

char c;
}u;
u.i = 12; // si cette variable contient un int, elle ne peut plus contenir de char sans supprimer la valeur du int

```
Attention, union est diffférent d'une *Struct* qui pourrait contenir un *int* et un

### Section 2.5 : L'organisation de la mémoire

La mémoire peut-être divisée en six zones principales :

![](https://raw.githubusercontent.com/Twan0u/Sinf12BA/master/LSINF1252-Syst%C3%A8me

#### 2.5.1 Le segment text
Contient toutes les instruction qui sont exécutées par le microprocesseur. (uniqueme

#### 2.5.2 Le segment des données initialisées
Contient l'ensemble des données et chaînes de caractères qui sont utilisées dans le

#### 2.5.3 Le segment des données non-initialisées
Contient les valeurs des variables non-globales

#### 2.5.4 Le tas (ou heap)
C'est dans une des 2 zones dans laquelle un programme peur obtenir de la mémoire su

##### Malloc
En pratique, on utilise malloc(3) pour allouer de la mémoire et free(3) pour la lib
aloué manuellement en mémoire (variables). Attention l'oubli de ces libérations mém

Malloc, contrairement à calloc ne réinitialise pas la zone mémoire libérée.
```

C
#include
string= (char *) malloc(length * sizeof(char))//malloc retourne un ptr void qu'il faut caster ensuite
free(string);

```
##### Calloc
```

c
void *calloc(size_t num_element, size_t size); // base
char *ptr = calloc(15,size(char));//exemple
```

## 2.5.5 La pile (ou stack)
```

Cette zone est très importante, elle stocke :

- l'ensemble des variables locales
- les variables de retour de toutes les fonction qui sont appelées
- Les arguments placés aux fonctions

Cette zone est gérée comme une pile.

Les arguments et variables d'environnement

argc : nombre d'arguments
char* argv[] : les arguments
argv[0]: nom du programme exécuté

Les **variables d'environnement** sont toutes les variables permettant d'accéder à certaines informations de l'environnement qui lance le programme. (ex : path , lang, shell , home, ...).

Cours S4

Dram : condensateur - gourmand en énergie, 50ns
Sram : consomation en continue => production de chaleur , 1ns
Dram : GB
Sram : MB
Posibilité d'avoir les deux avantages ?
on doit mettre dans la sram les données en cours d'utilisation (aussi appelé Cache)

Mémoire

Code-Données-Heap-Stack

Principe de localité :

- spatiale : si on edite un élément, il est courrant d'accéder à une variable proche en mémoire
- Si on a accédé à l'adresse X à l'instant t, il est commun d'accéder à la même adresse X à l'instant t+1

En pratique, on a une hierarchie de mémoire caches. on sépare la cache insruction de la cache données.

il existe une cache dans le processeur.

Cours S5

Cours S6

livelock = le processeur tourne mais rien n'est exécuté

Cours S7

voir slide

Cours S8

ar -> archive

2 formes de lib
-stat
on incl les librairies manuellement dans le makefile
-dynamiques
il est inutile de sauvegarder des librairies dans chaque exécutable si la librairie est présente dans tous les fichiers
référence vers une librairie en mémoire
+efficace : mémoire
Attentoion que elle soit bien présente sur le système

- Matériel
- Kernel(drivers[abstraction du matériel], interruptions, …)
- processus système
- Applications

Image sur les slides

appels systèmes : abstraction pour intéragir avec le noyau
dnas la section 2 du manuel

- getpid : n° du processus système
- read : lire des fichiers
- kill : tuer le processus
- brk : mémoire (utilisé par malloc)

appel système :

1. Appeler le kernel
2. Quel appel système ?
3. Passer les arguments
4. Exécuter appel système
5. Retourner le résultat
6. Retour au processus

fork : copie presque identique en mémoire (pid !=) copie ses data (contexte != stack et heap …)

père waitpid()
execve("hello"): remplace le programme par un exécutable . suicide par execve (pid =)
ensuite il fait appel à exit
préviens le père qu'il a fini****