Systèmes Informatiques 1

Chapitre 1 : Introduction

Section 1.1 : Introduction

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)

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.
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)***
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
```

Chapitre 2 : Langage C

Section 2.1 : Le langage C

Le langage C cest un langage rapide qui compose la plus part des systèmes d'exploitations actuels.

Le langage machine : langage binaire pour le processeur.
Le langage assembleur est converti en langage machine grâce à un assebleur. Ce langage est le plus proche du processeur.
Chaque famille de processeur possède un language d'assemblage qui lui est propre

Préprocesseur

Au moment de la compilation, le compilateur va exécuter les directives préprocesseur.

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

String

Le langage C n'integre pas d'office les boolean et les strings. En C, les strings sont des tableaux de caractères dont le dernier élément contient la valeur '\0'.

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

Constructons Syntaxiques

```
if (COND){}else{}
while(COND){} //if cond false do nothing
do {} while(COND); if cond is false then
for(INIT;COND;INCR){}
```

```
#include<stdio.h>

int main ( intarg c , char * argv [ ] ){
    printf ( "Hello , %s!\n" , NAME) ;// affiche sur la sortie standard le
    // \n c'est pour le retour à la ligne le %s c'est pour la variable
    return 0;//Un programme retourne toujours une valeur (en C : return ou exit ).
}
```

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

```
sizeof(DATA_TYPE)
```

Les **nombres signés** sont représentés sous la forme : Signe (négatif si = 1) - Nombre

Standard IEEE 754

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 tableau en mémoire si on veut l'utiliser par la suite

```
#define N 10
float matrice[N][N]
```

Caractères et chaines de caractères

ASCII : 7 et 8 bits
Les lettres étant des char stockés sous la forme d'entiers, on peut donc effectuer des manipulations numériques avec des char.
L'UNICODE lui, gère **toutes** les langues connues sur terre.
Une chaine de caractères se termine toujours par un '\0' (equivalent à 0)

Les Pointeurs

Un pointeur est défini comme étant une variable contenant l'adresse d'une autre variable.

```
&var // adresse à laquelle une variable est stockée
sizeof(i) // taille en bytes d'une variable
*ptr // récupère la valeur à l'adresse du pointeur
```

Les structures

En C, contrairement au java on n'encapsule pas les données dans des classes avec des méthodes propres.

```
struct coord{
    int x;
```

```
    int y;
    int z;
}
struct coord test = {1,2,3}; // crée une instance
test.x = 1 //l'opérateur point accède directement àla variable x de l'instance test.
(*ptr).x // accède à l'élèm x du ptr
ptr->x //idem supp
```

Les anciens compilateurs ne permettaient pas de posséder des tableaux de taille variables à l'intérieure de structures.

On peut redéfinir des noms de structures :

```
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*

Les fonctions

En C les fonctions peuvent être utilisés en argument.

Manipulation bits

```
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
```

Declarations

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

---

Caractères et chaines de caractères

ASCII - 7 bits mais représenté sur 8 bits.
ISO-8859 8bits
Unicode Permet de représenter tous les caractères connus de toutes les langues. il utilise plus de 8 bits par caractères

En C chaque chaine de caractère se termine avec '\0'
La taille d'une chaine de caractères est donc de 'chaine' + '\0' = 'taille de la chaine' + 1

Variables

```
int x // initialise le x comme un entier
x = 2 // affecte 2 à la variable 2
printf("%p\n",&x); //afficher une variable en mémoire dans un print:
```

Pointeurs

En C la mémoire est gérée avec des pointeurs.

Déclaration d'un pointeur :

```
int i = 5;
int *ptr = &i ;
```

Récupérer la valeur stockée à l'adresse du pointeur :

```
int j = * ptr ;
printf ( "%d\n" , j ) ; // affichera 5
```

Modifier la valeur stockée à l'adresse du pointeur :

```
* ptr = 10;
printf ( "%d == %d\n" , * ptr, i ) ; // affichera 10
== 10
```

Structures

Definition d'une structure

```
struct student {
int matricule ;
char prenom [ 2 0 ] ;
char nom [ 3 0 ] ;
} ;
```

Declaration + Initialization

```
struct student linus = { 1 , " Linus " , " Torvalds " } ;
struct student richard = { .matricule = 2, .prenom = " Richard " , . nom = " Stallman " } ;
struct student evil ;
evil . noma = 3 ;
evil . prenom = "Bill" ;
evil . nom = " Gates " ;
```

> *typedef //permet de redéfinir le nom des types de données.*
>
> "

Les éléments d'une structure peuvent être accédés via l'opérateur '.' (point)

Les éléments référencés par un pointeur vers une structure peuvent être accédés via l'opérateur '->'

Déclaration

Indique au compilateur le type (en cas de variables) ou les arguments et le type de la valeur de retour (en cas de fonctions). Toutes fonctions ou variables doivent être déclarées avant d'être utilisées.

```
int timestwo( *n);
unsigned long my long ;
```

Définition

Le corps de la fonction spécifié dans la déclaration, ou en cas d'une variable son initialisation.

```
it times two(int *n) {
return((*n)+(*n)) ;
}
my long = 5 ;
```

Tableaux

```
int tab[3]; //puis allocation par la suite
int tab2[3] = {1,2,3};
int tab[]= {1,2,3};
```

Attention : on ne peut pas récuérer la taille d'un tableau avec la variable tab.length.

Pointeurs

Attention : quand on place un pointeur comme argument d'une fonction on ne peut la modifier.

```
&var // permets e récupérer la variable
*ptr = &var // on déclare un pointeur
```

```
int times_two ( i nt *n) {
return((*n)+(*n));
}
int timestwo (int *n) {
*n=(*n ) +(*n ) ;
return ( *n ) ;
}
```

Structures

```
struct student{
int matricule;
char prenom[20];
char nom[30];
}
```

```
struct student bob ={42,"Bob","Truc"}
```

Les éléments d'une structure de référence par un **(*e).nom** ou **e->nom**

modificationdes type d'élément

```
typedef int entier;
```

Manipulation des bits

```
~a // négation  : inverse tous les bits
```

> *Bit shift : Action de déplaçer les bits vers la gauche ou la droite.*
> *plus optimisé pour "effecturer des opérations"*
>
> "

Cours 3

Partie des variables

» Variable globales
» Variable locales

Main

```
int main (int argc, char *argv[])
```

Constantes et enum

```
const int vie = 42; //défini une variable qui ne peut changer
typedef enum {lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche};
```

Organisation des processus en mémoire

OS - arg, envp - Stack & Heap - données non-initialisées - données initialisées - code(texte) - OS

Alocations dynamiques en mémoire

Stack : allouées en automatiquement en mémoire (functions)
Heap : alouées manuellement en mémoire (variables)

brk & sbrk

“#include

”

```
int brk(void*addr);
void*sbrk(intptr_t incrément);
```

ces deux éléments permettent de modifier la taille du data segment size mais ils ne sont que très peu utilisées

Calloc

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

la fonction calloc alloue de la mémoire préalablement initialisée à 0.

Malloc

```
void *malloc(size_t size); // base
char *ptr = malloc(15*size(char));//exemple
```

Attention : cette fonction ne fait que libérer de l’espace en mémoire, il ne reinitialise pas les données à 0.

Realloc (BONUS)

```
...
```

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****