Systèmes d'Exploitation (HLIN303)

Michel Meynard

UM

Univ. Montpellier

Table des matières

Plan

Introduction I

Définition d'un SE (*Operating System*)

Couche logicielle offrant une interface entre la machine matérielle et les utilisateurs.

Objectifs

- convivialité de l'interface (GUI/CUI)
- clarté et généricité des concepts (arborescence de répertoires et fichiers, droits des utilisateurs, ...)
- efficacité de l'allocation des ressources en temps et en espace

Introduction II

Services

- multiprogrammé (ou multi-tâches) préemptif (isolement des processus)
- multi-utilisateurs (authentification)
- pilotage des périphériques toujours plus nombreux
- fonctionnalités réseaux (partage de ressources distantes)
- communications réseaux (protocoles Internet)
- personnalisable selon l'utilisation (développeur, multimédia, SGBD, applications de bureau, . . .)

Introduction III

Pricipaux OS

Microsoft Windows pricipal système sur ordi. personnels

les Unix GNU/LInux, BSD, Unix propriétaires (AIX d'IBM, Solaris de Sun, HP-UX, ...)

Mac OS X, iOS des dérivés d'Unix sur les produits Apple

Android, Google Chrome OS des dérivés basés sur un noyau Linux

Mainframes VM, MVS, OS/400 d'IBM, GCOS de Bull

Historique I

- A l'origine : machine énorme programmée depuis une console, beaucoup d'opérations manuelles
- développement de périphériques d'E/S (cartes 80 col., imprimantes, bandes magnétiques), développement logiciel : assembleur, chargeur, éditeur de liens, librairies, pilotes de périphérique
- langages évolués (compilés) ; exemple de job : exécution d'un prg Fortran
  - montage manuel de la bande magnétique contenant le compilateur F
  - lecture du prg depuis le lecteur de cartes 80 col.
  - production du code assembleur sur une bande
  - montage de la bande contenant l'assembleur
  - assemblage puis édition de lien produisant le binaire sur une bande
  - chargement et exécution du prog.

Historique II

Remarques

- beaucoup d'interventions manuelles !
- sous-utilisation de l'UC
- machine à 2 millions de dollars réservée par créneaux d'1h !

solution 1

- regroupement (batch) des opérations de même type
- seuls les opérateurs manipulent la console et les périph.
- en cas d'erreur, dump mémoire et registres fourni au programmeur

Historique III

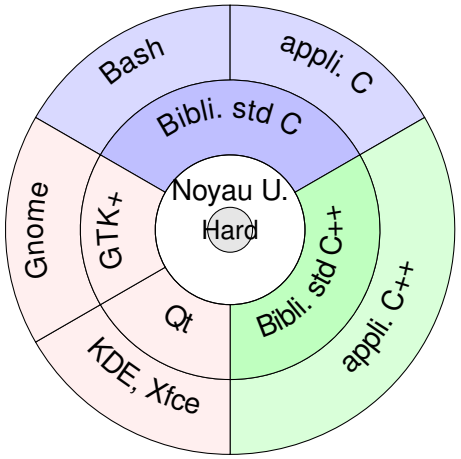
solution 2

- moniteur résidant en mémoire séquençant les jobs
- cartes de ctrl ajoutées spécifiant la tâche à accomplir
- définition d'un langage de commande des jobs (JCL) ancêtre des shell

solution 3

- améliorer le moniteur pour en faire un SE multiprogrammé !
- stocker le SE sur disque dur et l'amorcer (bootstrap) depuis un moniteur résidant en ROM (le BIOS)

Architecture en couche d'Unix



- une appli. C utilise la bibliothèque standard C (`printf, stat`, ...) (man 3)
- une appli C++ utilise la bibli std C++ (insertion dans un flot `<<`, les classes `vector, thread`, ...)
- d'autres bibliothèques existent (GUI)
- toute appli peut utiliser les appels noyau (man 2) : `fork, pipe, ...`

Les langages I

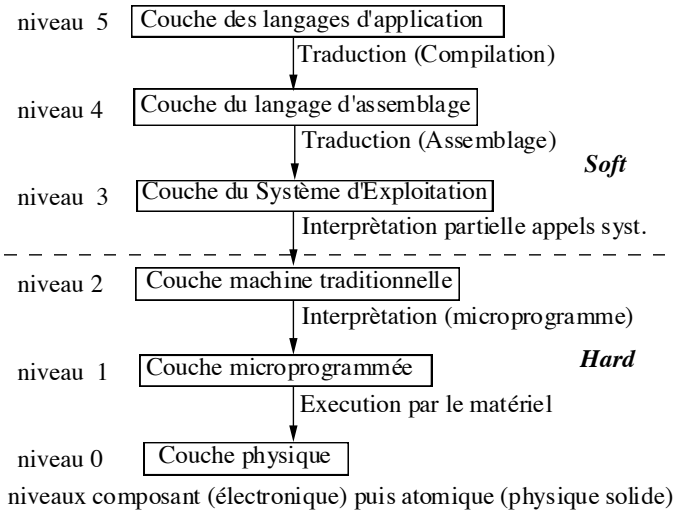
Le jeu d'instructions du processeur est limité et primitif. On construit donc au-dessus, une série de couches logicielles permettant à l'homme un dialogue plus aisé.

Interprétation Vs compilation

Les programmes en Li sont :

- soit traduits (compilés) en L_{i-1} ou L_{i-2} ou ... L_1,
- soit interprétés par un interpréteur tournant en L_{i-1} ou L_{i-2} ou ... L_1

Couches et langages I



Couches et langages II

Description des couches

0. portes logiques, circuits combinatoires, à mémoire (électronique)
1. instruction machine (code binaire) interprétée par son microprogramme
2. suite d'instructions machines du jeu d'instructions
3. niveau 2 + ensemble des services offerts par le S.E. (appels systèmes)
4. langage d'assemblage symbolique traduit en 3 par le programme assembleur
5. langages évolués (de haut niveau) traduits en 3 par compilateurs ou alors interprétés par des programmes de niveau 3

Matériel et Logiciel I

Hardware

Le matériel est l'ensemble des composants mécaniques et électroniques de la machine : processeur(s), mémoires, périphériques, bus de liaison, alimentation...

Software

Le logiciel est l'ensemble des programmes, de quelque niveau que ce soit, exécutables par un niveau de l'ordinateur. Un programme est un mot d'un langage. Le logiciel est immatériel même s'il peut être stocké physiquement sur des supports mémoires.

Matériel et Logiciel II

Matériel et Logiciel sont conceptuellement équivalents

- Toute opération effectuée par logiciel peut l'être directement par matériel et toute instruction exécutée par matériel peut être simulée par logiciel
- Le choix est facteur du coût de réalisation, de la vitesse d'exécution, de la fiabilité requise, de l'évolution prévue (maintenance), du délai de réalisation du matériel
- Dans un langage donné, le programmeur communique avec une machine virtuelle sans se soucier des niveaux inférieurs.

Matériel et Logiciel III

Exemples de répartition matériel/logiciel

- premiers ordinateurs : multiplication, division, manip. de chaînes, commutation de processus ... par logiciel : actuellement descendus au niveau matériel
- à l'inverse, l'apparition des processeurs micro-programmés à fait remonter d'un niveau les instructions machines
- les processeurs RISC à jeu d'instructions réduit ont également favorisé la migration vers le haut
- machines spécialisées (Lisp, bases de données)
- Conception Assistée par Ordinateur : prototypage de circuits électroniques par logiciel
- développement de logiciels destinés à une machine matérielle inexistante par simulation (contrainte économique fondamentale)

Objectifs du cours

- Comprendre le processus de compilation des programmes (C sous Unix)
- posséder les bases indispensables de la représentation des données en machines afin de comprendre l'utilité de structure de données efficaces
- développer des algorithmes simples puis les traduire en un langage de programmation (C)
- distinguer les appels systèmes Unix des fonctions de la bibliothèque C
- appréhender les Entrées/Sorties généralisées et leur lien avec un Système de Gestion de Fichier
- maîtriser la gestion des fichiers et des flots C sous Unix

Plan

Compilation des programmes (C sous Unix) I

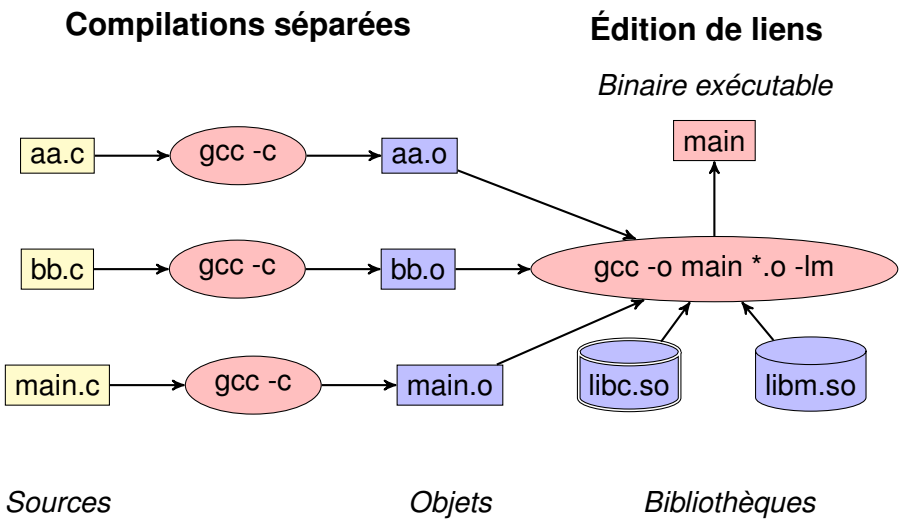
- Unix développé en C donc interface naturelle avec le SE
- Compilation vs interprétation : maîtriser les phases
  - Prétraitement des directives de compilation (`#include`, `#define`, `#ifdef`, `...`) de chaque fichier
  - Analyse lexicale et syntaxique (parse error ou syntax error)
  - Analyse sémantique (correspondance de type, déclaration préalable des objets, ...)
  - Compilation proprement dite du source C en source écrit en langage d'assemblage
  - Assemblage en fichier objet .o
  - Edition des liens des objets entre eux et avec la ou les bibliothèques pour réaliser le **fichier binaire exécutable**
- Cette succession est souvent réalisée à l'aide d'une unique commande : `gcc monprog.c -o monprog`

Compilation des programmes (C sous Unix) II

- la commande `gcc` supporte ces principales options :

-c	Compiler et assembler seulement (compile)
-o xxx	Renommage du fichier de sortie (output)
-lm	Utilisation de la librairie mathématique `libm.a` ou `.so`
-Wall	Voir tous les avertissements (Warning all)
-g	Ajoute les informations de débogage nécessaires à `gdb`
-E	Que le prétraitement
-S	Compiler sans assembler
-std=c99	Permet les déf de var dans les for, les //, (c11 pour le standard C 2011)
-static	pour l'édition de lien statique

Processus de compilation



Structure d'un programme C I

```
Cours$ cat argv.c
#include <stdio.h>

int main(int argc, char *argv[], char *env[]) {
  printf("Nombre d'arguments : %i\n\nListe des \
arguments :\n",argc);
  for (int i=0;i<argc;i++){
    printf("%s\n",argv[i]);
  }
  return 0;
}
Cours$ gcc -o argv argv.c -Wall
Cours$ argv un 2 34.5
Nombre d'arguments : 4
```


Structure d'un programme C II

```
Liste des arguments :
argv
un
2
34.5
Cours$ echo $?
0
```


Les fonctions I

Déclaration d'une fonction

```
char* itoa(int, char *);
```

- le type de retour de la fonction (`char*`)
- le nom de la fonction (`itoa`)
- la liste des types des paramètres (éventuellement accompagnés des noms de paramètres formels)
- un `;` indispensable
- plusieurs déclarations identiques de la même fonction sont possibles (inclusions multiples du même fichier d'en-tête)
- le type `void` permet de déclarer une fonction sans résultat ou sans paramètre

Les fonctions II

Définition d'une fonction

```
char* itoa(int i, char *s){ ...}
```

- le ; est remplacé par un **bloc** d'instructions
- les noms des paramètres formels sont indispensables
- pas de surcharge : une unique définition dans tout le programme
- la fonction `main()` est l'unique point d'entrée du programme. C'est une fonction comme les autres (elle peut être appelée récursivement)
- la déclaration d'une fonction doit **précéder** son appel, mais sa définition peut être absente (dans un autre fichier objet ou dans une bibliothèque)

Un exemple complet : fact.c I

```
#include <stdio.h>
#include <stdlib.h>

unsigned int fact(unsigned int);

int main(int argc, char* argv[], char* env[]){
  if(argc!=2){
    fprintf(stderr, "Syntaxe incorrecte : %s \
fichier.txt\n", argv[0]);
    return 1;
  }
  int n=atoi(argv[1]);
  if (n<0){
    fprintf(stderr, "L'argument doit être un entier \
```


Un exemple complet : fact.c II

```
positif !\n");
    return 2;
  }
  printf("%d!=%d\n",n,fact(n));
  exit(0);  /* ou return */
}
unsigned int fact(unsigned int i){
  if (i<=1)
    return 1;
  else
    return i*fact(i-1);
}
```


Un exemple complet : fact.c III

Quelques exécutions

```
Cours$ fact
Syntaxe incorrecte : fact fichier.txt
Cours$ fact -65
L'argument doit être un entier positif !
Cours$ echo $?
2
Cours$ fact 12
12!=479001600
Cours$ fact toto
0!=1
Cours$ echo $?
0
```

Passage des paramètres I

- En C, le seul mode de passage des paramètres à une fonction est le passage par **copie** (aussi appelé par valeur) : une copie du paramètre réel (d'appel) est placé sur la pile et c'est cette copie qui est ensuite utilisée par la fonction appelée
- il ne peut donc pas y avoir de modification par l'appelée sur le paramètre réel
- le passage d'un paramètre de type pointeur permet à l'appelée de modifier la zone pointée mais pas le pointeur lui-même
- le passage d'un tableau à une fonction est similaire au passage du pointeur sur la première case de ce tableau : par conséquent, le contenu du tableau pourra être modifié

Passage des paramètres II

Exemple passageparam.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void modifieur(int i, char* s, float t[2]){
  i=i+1;
  s[0]='M'; s=NULL;
  t[0]=0.0; t++;
  return;
}
int main(int argc, char* argv[], char* env[]){
  int n=5;
  char* ch=malloc(strlen("bonjour")+1);
  // les chaînes littérales sont const!
```


Passage des paramètres III

```
  strcpy(ch,"bonjour");
  float compl[2]={1.1,2.2};
  printf("AVANT : n=%d ; ch=%s ; compl={%f,%f} ; \
&ch=%p ; &compl=%p\n",n,ch,compl[0],compl[1],&ch,&compl);
  modifieur(n,ch,compl);
  printf("APRES : n=%d ; ch=%s ; compl={%f,%f} ; \
&ch=%p ; &compl=%p\n",n,ch,compl[0],compl[1],&ch,&compl);
  return 0;
}
```

Exécution

```
AVANT : n=5 ; ch=bonjour ; compl={1.100000,2.200000} ;
&ch=0x7fff53fd7a68 ; &compl=0x7fff53fd7a90
APRES : n=5 ; ch=Monjour ; compl={0.000000,2.200000} ;
&ch=0x7fff53fd7a68 ; &compl=0x7fff53fd7a90
```


Les variables C : propriétés I

- en C, toute variable est **typée**, ce qui lui donne une taille (sizeof) et un codage (voir représentation des données)
- une variable est **située** dans un des deux segments suivant : la pile, le segment de données statique. Un objet dynamique est situé dans le tas, il n'est accessible que par un pointeur
- la **durée de vie** d'une variable ou d'un objet dyn. est liée à sa localisation :
  - pile : durée de vie de la fonction dans laquelle elle a été définie
  - tas : depuis le `malloc()` jusqu'au `free()`
  - statique : durée de vie du processus
- la **portée** d'une variable est la zone du programme où elle peut être utilisée. Une variable définie en dehors de toute fonction a une **portée globale** à toute l'application (sauf si `static` qui limite au fichier). Une variable définie dans une fonction ou dans un bloc a une **portée locale** au bloc.

Les variables C : propriétés II

- la **résolution** de portée d'un nom de variable consiste à remonter les blocs englobants pour retrouver la définition de variable la plus proche
- une variable globale peut être déclarée en la faisant précéder du mot-clé `extern` : `extern int g;`. Toute variable ne peut être définie qu'une fois

Exemple portee.c

```
#include <stdio.h>
float g=10.2;
int main(int argc, char* argv[], char* env[]){
  {
    char g='A';
    for(int g=1;g<5;g++){
      printf("g=%d;",g);
```


Les variables C : propriétés III

```
    }
    printf("\ng=%c\n",g);
  }
  printf("g=%f\n",g);
  return 0;
}
```

Exécution

```
Cours$ portee
g=1;g=2;g=3;g=4;
g=A
g=10.200000
```


Les types C I

Un type de données utilise un système de codage et a une taille (`sizeof()`) qui est un multiple de l'octet. Le codage des types est fixé dans la norme du langage tandis que leur taille dépendent parfois des architectures de machines (32 bits, 64 bits, ...).

- **char** entier signé en complément à 2 sur 1 octet. Il permet de représenter les caractères ASCII (7 bits), les octets lus dans des fichiers, les cases des chaînes de caractères. Il existe aussi `signed char` et `unsigned char`
- **int** entier signé en complément à 2 de taille dépendant de la machine (souvent 4 octets). Le type `unsigned int` est de même taille mais codé en RBNS

Les types C II

- **short, long** entier court (long) codé en complément à 2 dont la taille dépend de l'architecture. Par exemple (Mac OS X i5) : int(4), short(2), long(8), long long(8). Les types non signés correspondants sont possibles.
- **C99** cette norme définit les types de taille fixée : `int8_t`, `uint8_t, int16_t, uint32_t`. Elle définit également des types rapides de taille minimale comme : `uint_fast64_t` (en-tête `stdint.h`)
- **float** nombre flottant IEEE-754 avec des tailles non fixées : `float(4), double(8), long double(16)`

Les types C III

pointeur un pointeur est une adresse mémoire (entier non signé) sur un objet d'un certain type. Le type `char *` n'est pas le même que `int *` même s'ils ont la même taille. Le type `void *` est un pointeur générique (sur n'importe quoi). Le pointeur `NULL` vaut 0 et pointe sur une adrs mémoire interdite ! L'arithmétique des pointeurs est basée sur une unité égale à la taille du type pointé : incrémenter un pointeur sur `char` avance de 1 alors que sur un `int*` l'incrémentation avance de `sizeof(int)` (4)

Les types C IV

tableau séquence d'objets de même type e.g. `int t[4]`. La taille d'un tableau n'est pas définie dans le tableau : il faut soit la conserver dans une autre variable (`argc` est la taille d'`argv`), soit positionner un objet terminateur à la fin de la séquence (`'\0'` en fin de chaîne, `NULL` en fin d'`env`). Depuis c99, la taille d'un tableau local peut être initialisé à l'exécution. La taille d'un tableau(`sizeof()`) est la taille d'un objet multiplié par le nombre de cases. Le nom du tableau peut être vu comme un pointeur constant adressant la première case. L'opérateur d'indexation ([exp]) peut être appliqué à un nom de tableau comme à un pointeur pour référencer une case.

Les types C V

struct séquence hétérogène d'objets e.g. :
`struct cell{int val;struct cell *suiv;} tete;`
Les champs de la `struct` sont référencés grâce à la notation pointée sur la variable `tete` : `(tete.suiv)->val` est la seconde valeur de la liste. **Il faut allouer (malloc) de la mémoire aux structures de données dynamiques.**

union un champ parmi plusieurs possibles :
`union dyn{int i;float f;char *s;} x;` x est une variable qui peut contenir un entier, un flottant ou une chaine. La taille d'une union est la taille de son plus grand composant.

typedef permet de définir un nouveau type : `typedef exptype nom;`

Un exemple complet I

liste.h

```
/** @file liste.h
 * @brief en-tête des fonctions de manipulation de liste
 * @author Michel Meynard*/
#ifndef _LISTE
#define _LISTE
/** @typedef liste
 * @brief le type liste est un pointeur sur cell. */
typedef struct cell* liste;
/** @typedef cell
 * @brief une cellule composée d'un entier et d'une liste. */
typedef struct cell {
  int val; /**< élément proprement dit */
  liste suiv; /**< pointeur sur cellule suivante */
} cell;
/** Crèe une liste d'entier vide.
```

Un exemple complet II

```
 * @return une liste vide (NULL)
 */
liste creerListe();
/** Teste si une liste est vide.
 * @param l la liste à tester
 * @return 0 si non vide, 1 sinon
 */
int vide(liste l);
/** Retourne le premier entier de la liste sans le retirer.
 * @param l la liste
 * @return le premier entier de l
 * @warning non défini si liste vide
 */
int premier(liste l);
/** Retourne la liste l sans son premier élément (sans le désallouer
 * sans modifier l).
 * @param l la liste
```


Un exemple complet III

```
 * @return la suite de la liste
 * @warning non défini si liste vide
 */
liste suite(liste l);
/** Retourne la liste l à laquelle on a ajouté un nouveau
 * premier élément.
 * (sans modifier l)
 * @param i l'entier à ajouter en premier
 * @param l la liste
 * @return la nouvelle liste*/
liste ajDeb(int i, liste l);
/** Teste si un entier fait partie d'une liste.
 * @param i l'entier recherché
 * @param l la liste à tester
 * @return 1 si i est dans l, 0 sinon
 */
int dansListe(int i, liste l);
```


Un exemple complet IV

```
/** Vide une liste en désallouant toutes ses cellules.
 * @param pl un pointeur sur la liste à vider
 * @warning effets de bord sur des listes qui partageraient des cellu
 */
void vider(liste *pl);
#endif /* _LISTE */
```

liste.c

```
#include<stdlib.h>
#include "liste.h"
liste creerListe(){return NULL;}      // creer une liste vide
int vide(liste l){return (l==NULL);}  // teste si vide
int premier(liste l){return l->val;}  // non defini si liste vide
liste suite(liste l){return l->suiv;} // non defini si liste vide
liste ajDeb(int i, liste l){       // ajoute i au début de l
  liste nouv=(liste) malloc(sizeof(cell));
```


Un exemple complet V

```
  nouv->val=i;
  nouv->suiv=l;
  return nouv;
}
int dansListe(int i, liste l){ // vrai si i dans l
  return !vide(l) && (
      i==premier(l) ||
      dansListe(i,suite(l))
      );
}
void vider(liste *pl){ // pl est un pointeur sur la liste
// vide recursivement une liste (attention aux listes qui pointa
  if (vide(*pl)) return;   // vidage d'une liste par desalloc de
  else {
    vider(&((*pl)->suiv)); // appel recursif
    free((liste)*pl); // desalloue, ne modifie pas
    (*pl)=creerListe();
```

Un exemple complet VI

```
    return;
  }
}
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "liste.h"
int main(int argc, char *argv[]){
  if(argc<2){
    fprintf(stderr, "Un argument entier S.V.P. !\n");
    return 1;
  }
  liste prems=ajDeb(2,ajDeb(3,ajDeb(5,ajDeb(7,ajDeb(11,\
ajDeb(13,creerListe()))))));
  if(dansListe(atoi(argv[1]),prems)){
```


Un exemple complet VII

```
    printf("%d est un nombre premier < 17 !\n", atoi(argv[1]));
  }else{
    printf("%d n'est pas un nombre premier < 17 !\n", atoi(argv[
  }
  vider(&prems);
  return 0;
}
```

Compilation puis exécution

```
gcc -o main liste.c main.c -std=c99 -Wall
$ main 13
13 est un nombre premier < 17 !
$ main 6
6 n'est pas un nombre premier < 17 !
```


Plan

Plan

Les Unités I

- La technologie passée et actuelle a consacrée les circuits mémoires (électroniques et magnétiques) permettant de stocker des données sous forme binaire
- des chercheurs ont étudié et continuent d'étudier des circuits ternaires et même décimaux
- **bit** : abréviation de binary digit, le bit constitue la plus petite unité d'information et vaut soit 0, soit 1
- bits stockés dans des **mots de n bits** numérotés de la façon suivante :

b_{n-1}	b_{n-2}	...	b_2	b_1	b_0
1	0	...	1	1	0

- On regroupe ces bits par paquets de n qu'on appelle des quartets (n=4), des octets (n=8) *byte*, ou plus généralement des mots de n bits *word*

Les Unités II

- Poids fort et faible : la longueur des mots étant la plupart du temps paire (n=2p), on parle de demi-mot de poids fort (ou le plus significatif) pour les p bits de gauche et de demi-mot de poids faible (ou le moins significatif) pour les p bits de droite

Exemple : mot de 16 bits

b_{15}	b_{14}	...	b_8	b_7	b_6	...	b_0
1	0	...	1	1	0	...	0
octet le plus significatif				octet le moins significatif			
Most Signifant Byte				Least Significant Byte			

Les Unités III

- Unités multiples : nouvelles normes internationales (1999)

1 Kilo-octet = 10^3 octets	1 Kibi-octet = 2^{10} = 1024 octets (1 Kio)
1 Méga-octet = 10^6 octets	1 Mébi-octet = 2^{20}=1 048 576 (1 Mio)
1 Giga-octet = 10^9 octets	1 Gibi-octet = 2^{30} noté 1 Gio
1 Téra-octet = 10^{12} octets	1 Tébi-octet = 2^{40} octets (1 Tio)

Plan

Représentation en base 2 I

- Un mot de n bits permet de représenter 2^n codes différents. En base 2, ces 2^n configurations sont associées aux entiers positifs x compris dans l'intervalle $[0, 2^n - 1]$ de la façon suivante :

$$x = b_{n-1} * 2^{n-1} + b_{n-2} * 2^{n-2} + \ldots + b_1 * 2 + b_0$$

- un quartet permet de représenter l'intervalle [0, 15], un octet [0, 255], un mot de 16 bits [0, 65535].

Représentation en base 2 II

- Cette convention sera notée Représentation Binaire Non Signée (RBNS)

00...0	0
00...001	1
0000 0111	7 (4+2+1)
0110 0000	96 (64+32)
1111 1110	254 (128+64+32+16+8+4+2)
0000 0001 0000 0001	257 (256+1)
100...00	2^{n-1}
111...11	$2^n - 1$

Représentation en base 2^p

Plus compact en base 2^p : découper le mot $b_{n-1}b_{n-2}...b_0$ en tranches de p bits à partir de la droite (la dernière tranche est complétée par des 0 à gauche si n n'est pas multiple de p). Chacune des tranches obtenues est la représentation en base 2 d'un chiffre de x représenté en base 2^p.
p=3 (représentation **octale**) ou p=4 (représentation **hexadécimale**).
En représentation hexadécimale, les valeurs 10 à 15 sont représentées par les symboles A à F. On préfixe le nombre octal par 0, le nombre hexa par `0x`.

Exemples

```
x=200 et n=8
en binaire :     11 001 000 (128+64+8) : 200
en octal :        3   1   0 (3*64+8)   : 0310
en hexadécimal :     C   8 (12*16+8)   : 0xC8
```


Opérations

Addition binaire sur n bits Ajouter successivement de droite à gauche les bits de même poids des deux mots ainsi que la retenue éventuelle de l'addition précédente. En RBNS, la dernière retenue ou report (**carry**), représente le coefficient de poids 2^n et est donc synonyme de **dépassement de capacité**. Cet indicateur de Carry (Carry Flag) est situé dans le registre d'état du processeur.

exemple sur 8 bits

```
    1           1   1   1   1
    1   1   1   0   0   1   0   1          0xE5
+   1   0   0   0   1   0   1   1      +   0x8B
---------------------------------      --------
(1) 0   1   1   1   0   0   0   0          0x170      368 > 255
```

Les autres opérations dépendent de la représentation des entiers négatifs.

Le complément à 1 (C1)

Les entiers positifs sont en RBNS. Les entiers négatifs -|x| sont obtenus par inversion des bits de la RBNS de |x|. Le bit de poids n-1 indique le signe (0 positif, 1 négatif).
Intervalle de définition : $[-2^{n-1}+1, 2^{n-1}-1]$

Exemples sur un octet

```
3     0000 0011     -3    1111 1100
127   0111 1111     -127  1000 0000
0     0000 0000     0     1111 1111
```

Inconvénients :

- 2 représentations distinctes de 0 ;
- opérations arithmétiques peu aisées : 3 + -3 = 0 (1111 1111) mais 4 + -3 = 0 (00...0) !

Le second problème est résolu si l'on ajoute 1 lorsqu'on additionne un positif et un négatif : 3+1+ -3=0 (00...0) et 4 +1+ -3=1 (00...01)
D'où l'idée de la représentation en Complément à 2.

Le complément à 2 (C2) I

Les entiers positifs sont en RBNS tandis que les négatifs sont obtenus par C1+1. Le bit de poids n-1 indique le signe (0 positif, 1 négatif). Une autre façon d'obtenir le C2 d'un entier relatif x consiste à écrire la RBNS de la somme de x et de 2^n.
Intervalle de définition : $[-2^{n-1}, 2^{n-1}-1]$
Exemples sur un octet [-128, +127] :

```
3     0000 0011     -3    1111 1101
127   0111 1111     -127  1000 0001
0     0000 0000     -128  1000 0000
```

Inconvénient :

- Intervalle des négatifs non symétrique des positifs ;
- Le C2 de -128 est -128 !

Le complément à 2 (C2) II

Avantage fondamental du C2

L'addition binaire fonctionne correctement : l'addition de deux entiers en C2 donne le bon résultat en C2
-3+3=0 : 1111 1101+0000 0011=(1) 0000 0000
-3 + -3 = -6 : 1111 1101+1111 1101=(1) 1111 1010

Remarquons ici que le positionnement du Carry à 1 n'indique pas un dépassement de capacité !
Dépassement de capacité en C2 : l'Overflow Flag (OF).

```
127+127=-2   : 0111 1111 + 0111 1111=(0) 1111 1110
-128+-128=0  : 1000 0000 + 1000 0000=(1) 0000 0000
-127+-128=1  : 1000 0001 + 1000 0000=(1) 0000 0001
```

$Overflow = Retenue_{n-1} \oplus Retenue_{n-2}$

L'excédent à $2^{n-1}-1$

Tout nombre x est représenté par $x + 2^{n-1} - 1$ en RBNS. Attention, le bit de signe est inversé (0 négatif, 1 positif).
Intervalle de définition : $[-2^{n-1}+1, 2^{n-1}]$
Exemples sur un octet :

```
3     1000 0010     -3    0111 1100
128   1111 1111     -127  0000 0000
0     0111 1111
```

Avantage :

- représentation uniforme des entiers relatifs ;

Inconvénients :

- représentation des positifs différente de la RBNS ;
- opérations arithmétiques à adapter !

Opérations en RBNS et C2

Le C2 étant la représentation la plus utilisée (`int`), nous allons étudier les opérations arithmétiques en RBNS (`unsigned int`) et en C2.

Addition en RBNS et en C2, l'addition binaire (ADD) donne un résultat cohérent s'il n'y a pas dépassement de capacité (CF en RBNS, OF en C2).

Soustraction La soustraction x-y peut être réalisée par inversion du signe de y (NEG) puis addition avec x (ADD). L'instruction de soustraction SUB est généralement câblée par le matériel.

Multiplication et Division La multiplication x*y peut être réalisée par y additions successives de x tandis que la division peut être obtenue par soustractions successives et incrémentation d'un compteur tant que le dividende est supérieur à 0 (pas efficace $O(2^n)$).
Cependant, la plupart des processeurs fournissent des instructions MUL et DIV efficaces en $O(n)$ par décalage et addition.
Exemples : $13 * 12 = 13 * 2^3 + 13 * 2^2 = 13 << 3 + 13 << 2$
$126/16 = 126/2^4 = 1216 >> 4$

Codage DCB

Décimal Codé Binaire DCB ce codage utilise un quartet pour chaque chiffre décimal. Les quartets de 0xA à 0xF ne sont donc pas utilisés. Chaque octet permet donc de stocker 100 combinaisons différentes représentant les entiers de 0 à 99.
Le codage DCB des nombres à virgule nécessite de coder :

- le signe ;
- la position de la virgule ;
- les quartets de chiffres.

Inconvénients

- format de longueur variable ;
- taille mémoire utilisée importante ;
- opérations arithmétique lentes : ajustements nécessaires ;
- décalage des nombres nécessaires avant opérations pour faire coincider la virgule.

Avantage Résultats absolument corrects : pas d'erreurs de troncatures ou de précision d'où son utilisation en comptabilité

Plan

Virgule flottante

notation scientifique en virgule flottante : $x = m * b^e$

- m est la mantisse,
- b la base,
- e l'exposant.

Exemple : pi = $0,0314159 * 10^2 = 31,4159 * 10^{-1} = 3,14159 * 10^0$

Représentation **normalisée** : positionner un seul chiffre différent de 0 de la mantisse à gauche de la virgule . On obtient ainsi : $b^0 \leq m < b^1$.

Exemple de mantisse normalisée : $pi = 3,14159 * 10^0$

En binaire normalisé
Exemple : $7,25_{10} = 111,01_2 = 1,1101 * 2^2$
4+2+1+0,25=(1+0,5+0,25+0,0625)*4

Virgule Flottante binaire

Remarques :

- $2^0 = 1 \leq m < 2^1 = 2$
- Les puissances négatives de 2 sont : 0,5 ; 0,25 ; 0,125 ; 0,0625 ; 0,03125 ; 0,015625 ; 0,0078125 ; ...
- La plupart des nombres à partie décimale finie n'ont pas de représentation binaire finie : (0,1 ; 0,2 ; ...).
- Par contre, tous les nombres finis en virgule flottante en base 2 s'expriment de façon finie en décimal car 10 est multiple de 2.
- Réfléchir à la représentation en base 3 ...
- Cette représentation binaire en virgule flottante, quel que soit le nombre de bits de mantisse et d'exposant, ne fait qu'**approcher** la plupart des nombres décimaux.

Algorithme de conversion de la partie décimale On applique à la partie décimale des multiplications successives par 2, et on range, à chaque itération, la partie entière du produit dans le résultat.

Virgule Flottante en machine

Exemples de conversion 0,375*2=**0**,75*2=**1**,5 ; 0,5*2=**1**,0 soit 0,011
0,23*2=0,46*2=0,92*2=1,84*2=1,68*2=1,36*2=0,72*2=1,44*2=0,88 ...
0,23 sur 8 bits de mantisse : 0,00111010

Standardisation

- portabilité entre machines, langages ;
- reproductibilité des calculs ;
- communication de données via les réseaux ;
- représentation de nombres spéciaux (∞ NaN, ...) ;
- procédures d'arrondi ;

Norme IEEE-754 flottants en simple précision sur 32 bits (float)

- signe : 1 bit (0 : +, 1 : -) ;
- exposant : 8 bits en excédent 127 [-127, 128] ;
- mantisse : 23 bits en RBNS ; normalisé sans représentation du 1 de gauche ! La mantisse est arrondie !

Virgule Flottante simple précision

4 octets ordonnés : signe, exposant, mantisse.

Valeur décimale d'un float : $(-1)^s * 2^{e-127} * 1, m$

Exemples 33,0 = +100001,0 = $+1,000012^5$ représenté par : 0 1000 0100 0000 100... c'est-à-dire : 0x 42 04 00 00
-5,25 = -101,01 = $-1,01012^2$ représenté par : 1 1000 0001 0101 000... c'est-à-dire : 0x C0 A8 00 00

Nombres spéciaux

- 0 : e=0x0 et m=0x0 (s donne le signe) ;
- infini : e=0xFF et m=0x0 (s donne le signe) ;
- NaN (Not a Number) : e=0xFF et m qcq ;
- dénormalisés : e=0x0 et $m \neq$0x0 ; dans ce cas, il n'y a plus de bit caché : très petits nombres.

Intervalle : $]-2^{128}, 2^{128}[= [-3.4 * 10^{38}, 3.4 * 10^{38}]$ avec 7 chiffres décimaux **significatifs** (variant d'une unité)

Virgule Flottante (fin)

Remarques

- Il existe, entre deux nombres représentables, un intervalle de réels non exprimables. La taille de ces intervalles (pas) croît avec la valeur absolue.
- Ainsi l'**erreur relative** due à l'arrondi de représentation est approximativement la même pour les petits nombres et les grands !
- Le nombre de chiffres décimaux significatifs varie en fonction du nombre de bits de mantisse, tandis que l'étendue de l'intervalle représenté varie avec le nombre de bits d'exposant :

double précision sur 64 bits (double)

- 1 bit de signe ;
- 11 bits d'exposant ;
- 52 bits de mantisse (16 chiffres significatifs) ;

Plan

Représentation des caractères

Symboles
- alphabétiques,
- numériques,
- de ponctuation et autres (semi-graphiques, ctrl)

Utilisation
- entrées/sorties pour stockage et communication ;
- représentation interne des données des programmes ;

Code ou jeu de car. : Ensemble de caractères associés aux mots binaires les représentant. Historiquement, les codes avaient une taille fixe (7 ou 8 ou 16 bits).
- ASCII (7) : alphabet anglais ;
- ISO 8859-1 ou ISO Latin-1 (8) : code national français (é, à, ...) ;
- UniCode (16 puis 32) : codage universel (mandarin, cyrillique, ...) ;
- UTF8 : codage de longueur variable d'UniCode : 1 caractère codé sur 1 à 4 octets.

ASCII

American Standard Code for Information Interchange Ce très universel code à 7 bits fournit 128 caractères (0..127) divisé en 2 parties :
- 32 caractères de fonction (de contrôle) permettant de commander les périphériques (0..31) ;
- 96 caractères imprimables (32..127).

Codes de contrôle importants

```
0 Null   9 Horizontal Tabulation    10 Line Feed
13 Carriage Return
```

Codes imprimables importants

```
0x20 Espace            0x30-0x39 '0'-'9'
0x41-0x5A 'A'-'Z'      0x61-0x7A 'a'-'z'
```


Code ASCII

Hexa	MSD	0	1	2	3	4	5	6	7
LSD	Bin.	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	espace	0	@	P	`	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	-	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O		o	DEL

ISO 8859-1 et utf-8

MSD\ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö		Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö		ø	ù	ú	û	ü	ý	þ	ÿ

TABLE: ISO Latin-1

Représentation binaire UTF-8	Signification
0xxxxxxx	1 octet codant 1 à 7 bits
0011 0000	0x30='0' caractère zéro
110xxxxx 10xxxxxx	2 octets codant 8 à 11 bits
1100 0011 1010 1001	0xC3A9 caractère 'é'
1110xxxx 10xxxxxx 10xxxxxx	3 octets codant 12 à 16 bits
1110 0010 1000 0010 1010 1100	0xE282AC caractère euro €
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	4 octets codant 17 à 21 bits

TABLE: utf-8

Caractères UTF-8 et char C

char C

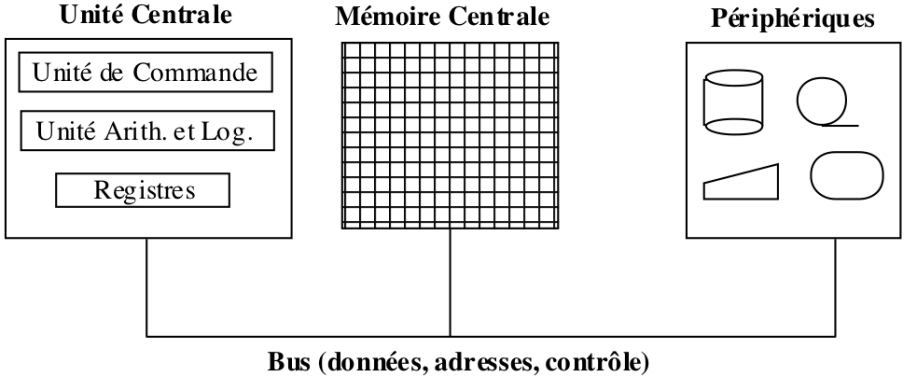
Un `char` C est l'équivalent d'un `byte` Java : un octet. Les caractères UTF-8 sont codés sur plusieurs octets (multi-byte).

- une chaîne littérale contenant des caractères accentués a une taille (`strlen`) supérieure au nombre de ses lettres
- la bibliothèque standard C permet de manipuler des caractères larges `wchar_t`
- l'entête `wchar.h` contient les déclarations des fonctions utiles telles que `wint_t fgetwc(FILE *stream);`
- cependant, la taille du type `wchar_t` dépend du compilateur (minimum 8 bits !) et les traitements ne sont donc pas portables !

Plan

Modèle de Von Neumann

Le modèle d'architecture de la plupart des ordinateurs actuels provient d'un travail effectué par John Von Neumann en 1946.



Modèle de programmation

- code = séquence d'instructions en MC
- données stockées en MC

Plan

L'Unité Centrale (UC) I

- Egalement appelé microprocesseur, processeur, CPU (*Central Processing Unit*), l'UC exécute séquentiellement les instructions stockées en Mémoire Centrale
- Le traitement d'une instruction se décompose en 3 temps : chargement, décodage, exécution
- L'Unité de commande (*control unit*) ordonnance l'exécution des instructions
- L'UAL (*Arithmetical and Logical Unit*) réalise les opérations telles que l'addition, la rotation, la conjonction, . . . sur des paramètres et résultats entiers stockés dans des registres (mots mémoires dans l'UC) ou en MC
- L'Unité Flottante (*Floating Point Unit*) réalise les opérations sur les nombres flottants

L'Unité Centrale (UC) II

- Les registres de l'UC sont répartis en 2 catégories :
  spécialisés destinés à une tâche pariculière :
  - Le Compteur Ordinal (CO) (*Instruction Pointer IP*, *Program Counter PC*) pointe sur la prochaine instruction à exécuter
  - le Registre Instruction (RI) contient l'instruction en cours d'exécution
  - le registre d'état (*Status Register, Flags, Program Status Word PSW*) contient un certain nombre d'indicateurs (ou drapeaux ou bits) permettant de connaître et de contrôler l'état du processeur
  - Le pointeur de pile (*Stack Pointer*) permet de mémoriser l'adresse en MC du sommet de pile (structure de données *Last In First Out LIFO* indispensable pour les appels procéduraux)

L'Unité Centrale (UC) III

de travail utilisés pour mémoriser les paramètres des fonctions, les variables sur lesquels on réalise des opérations :
- Des registres d'adresse (index ou bases) permettent de stocker les adresses des données en mémoire centrale. Le *Base Pointer BP* permet à une instance de fonction de mémoriser la base de son cadre de pile ; *Source Index, Destination Index* sont deux registres pointeurs sur des zones de MC contenant chacune un tableau sur lesquels on opère des copies, comparaisons, recherches . . .
- Des registes de données contenant des valeurs (*AL, AX, EAX, EBX, ECX, . . .*)

Algorithme de l'unité de commande I

répéter

1. charger dans RI l'instruction stockée en MC à l'adresse pointée par le CO
2. CO :=CO+taille(instruction en RI)
3. décoder (RI) en micro-instructions
4. (localiser en mémoire les données de l'instruction)
5. (charger les données)
6. exécuter l'instruction (suite de micro-instructions)
7. (stocker les résultats mémoires)

jusqu'à l'infini

Algorithme de l'unité de commande II

- Lors du démarrage de la machine, CO est initialisé à l'adresse mémoire 0 où se trouve le moniteur (Grub, lilo) en mémoire morte qui tente de charger l'amorce "boot-strap" du système d'exploitation
- Remarquons que cet algorithme peut parfaitement être simulé par un logiciel (interprèteur) qui permettra de tester des processeurs matériels avant même qu'il en soit sorti un prototype, ou bien de simuler une machine X sur une machine Y (émulation)

L'Unité Arith. et Log. I

opérations arithmétiques addition, soustraction, C2, incrémentation, décrémentation, multiplication, division, décalages arithmétiques (multiplication ou division par 2n)

opérations logiques et, ou, xor, non, rotations et décalages

Remarques

- Selon le processeur, certaines de ces opérations sont présentes ou non
- De plus, les opérations arithmétiques existent parfois pour plusieurs types de nombres (RBNS, C2, DCB, virgule flottante) ou bien des opérations d'ajustement permettent de les réaliser
- FPU spécialisée pour les opérations en virgule flottante

L'Unité Arith. et Log. II

- Les opérations arithmétiques et logiques positionnent certains indicateurs d'état du registre PSW. C'est en testant ces indicateurs que des branchements conditionnels peuvent être exécutés vers certaines parties de programme
- Pour accélérer les calculs, on a intérêt à utiliser les registres de travail comme paramètres des procédures, notamment l'accumulateur quand il existe

Plan

La Mémoire Centrale (MC) I

La mémoire centrale de l'ordinateur est constituée d'un ensemble ordonné de 2^m cellules (cases), chaque cellule contenant un mot de n bits. Ces cases permettent de conserver instructions, données, adresses.

Accès à la MC

La MC est une mémoire électronique et l'on accède en temps constant à n'importe laquelle de ses cellules au moyen de son adresse comprise dans l'intervalle $[0, 2^m - 1]$.

Les deux types d'accès à la MC par le processeur sont :

- la lecture qui transfère sur le bus de données, le mot contenu dans la cellule dont l'adresse est située sur le bus d'adresse
- l'écriture qui transfère dans la cellule dont l'adresse est sur le bus d'adresse, le mot contenu sur le bus de données.

La Mémoire Centrale (MC) II

- La taille n des cellules mémoires ainsi que la taille m de l'espace d'adressage sont des caractéristiques fondamentales de la machine
- Le mot de n bits est la plus petite unité d'information transférable entre la MC et les autres composant
- Généralement, les cellules contiennent des mots de 8, 16, 32 ou 64 bits

Contenu/adresse (valeur/nom) I

- Attention à ne jamais confondre le contenu d'une cellule, mot de n bits, et l'adresse de celle-ci, mot de m bits même lorsque n=m
- Il n'y a aucun moyen physique de distinguer une adresse stockée dans un case d'un entier stocké dans la case suivante : ces sont des mots binaires
- C'est le binaire exécutable qui distingue les contenus selon l'endroit où le compilateur les a placés !
- Parfois, le bus de données a une taille multiple de n ce qui permet la lecture ou l'écriture de plusieurs mots consécutifs en mémoire. Par exemple, les microprocesseurs x86-64 permettent des échanges de mots de 64 bits, soit 8 cases consécutives d'un octet

Contenu/adresse (valeur/nom) II

Exemple de représentation MC

Adresse (hexa)	Contenu (binaire)	Contenu (hexa)
00	0101 0011	53
01	1111 1010	FA
02	1000 0000	80
...	...	...
20	0010 0000	20
...	...	...
$2^m - 1$	0001 1111	1F

Parfois, une autre représentation graphique de l'espace mémoire est utilisé, en inversant l'ordre des adresses : adresses de poids faible en bas, adresses fortes en haut.

RAM et ROM I

Random Access Memory (RAM)

- La RAM est un type de mémoire électronique volatile et réinscriptible
- Elle est aussi nommée mémoire vive et plusieurs technologies permettent d'en construire différents sous-types : statique (SRAM), dynamique (DRAM) car nécessite des rafraîchissements
- La RAM constitue la majeure partie de l'espace mémoire
- DDR SDRAM : Double Data Rate Synchronous Dynamic RAM est une RAM dynamique (condensateur) qui a un pipeline interne permettant de synchroniser les opérations R/W
- Les caches mémoire et les registres de l'UC sont réalisés en SRAM qui est plus rapide que les DRAM mais qui est plus chère

RAM et ROM II

Read Only Memory (ROM)

- La ROM est un type de mémoire électronique non volatile et non réinscriptible
- Elle est aussi nommée mémoire morte et plusieurs technologies permettent d'en construire différents sous-types (ROM, PROM, EPROM, EEPROM (Flash), ...)
- La ROM constitue une faible partie de l'espace mémoire puisqu'elle ne contient que le moniteur réalisant le chargement du système d'exploitation et les Entrées/Sorties de plus bas niveau (Basic Input Output System BIOS)
- Sur Mac, le moniteur contient également les routines graphiques de base
- C'est toujours sur une adresse ROM que le Compteur Ordinal pointe lors du démarrage machine.

Plan

Les périphériques I

Les périphériques, ou organes d'Entrée/Sortie (E/S) Input/Output (I/O), permettent à l'ordinateur de **communiquer** avec l'homme ou d'autres machines, et de **mémoriser** massivement les données ou programmes dans des fichiers. La caractéristique essentielle des périphériques est leur **lenteur** : Processeur cadencé en Giga-Hertz : instruction éxécutée chaque nano-seconde (10^{-9} s) ; Disque dur de temps d'accès entre 10 et 20 ms (10^{-3} s) : rapport de 10^7 ! Clavier avec frappe à 10 octets par seconde : rapport de 10^8 ! Disque électronique (SSD) temps d'accès 10^{-4} s : rapport de 10^5

Les périphériques II

- Communication : échange d'informations avec l'homme à travers des terminaux de communication homme/machine : clavier, écran, souris, imprimante, synthétiseur (vocal), table à digitaliser, scanner, crayon optique, lecteur de codes-barres, lecteur de cartes magnétiques, terminaux, consoles . . . Il communique avec d'autres machines par l'intermédiaire de réseaux locaux ou longue distance
- Mémorisation de masse ou mémorisation secondaire :
  - non volatilité et réinscriptibilité
  - faible prix de l'octet stocké
  - lenteur d'accès et modes d'accès (séquentiel, séquentiel indexé, aléatoire, . . .)
  - forte densité
  - parfois amovibilité
  - Mean Time Between Failures plus important car organes mécaniques donc stratégie de sauvegarde

Support de la mémorisation de masse I

Disques électroniques (SSD)

- *solid-state drive* à base de mémoire électronique (Flash)
- plus résistant (MTBF), meilleur débit, plus faible consommation que les disques magnétiques
- beaucoup de config. avec un disque SSD pour l'OS (150 Gio) et un gros disque dur pour les données (plusieurs Tio)

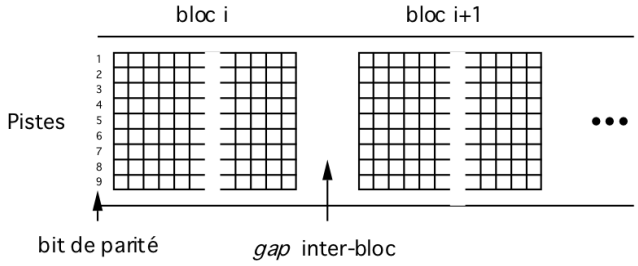
Supports Optiques

- Historiquement, les cartes 80 colonnes . . .
- Les rubans perforés (Machines Outils à Commande Numérique)
- CD-RW, DVD-RW permettent la sauvegarde de données à moindre coût

Support de la mémorisation de masse II

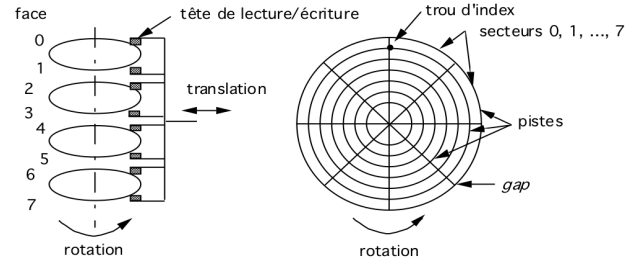
Supports Magnétiques : Bandes magnétiques : supports historiques à accès **séquentiel** particulièrement utilisés dans la sauvegarde (streamers)

Schéma d'une bande magnétique



Support de la mémorisation de masse III

Schéma d'un disque dur magnétique



- Les disques durs (*Hard Drive*) constituent les mémoires de masse les plus répandues
- fixes ou amovibles
- capacités jusqu'à 6 To, temps d'accès autour de 10 ms, transfert 200 Mo/s

Support de la mémorisation de masse IV

- Composé de faces, de pistes concentriques, de secteurs "soft sectored", la densité des disques est souvent caractérisée par le nombre de "tracks per inch" (tpi)
- Un cylindre est constitué d'un ensemble de pistes de même diamètre
- Un contrôleur de disque (carte) est chargé de transférer les informations entre un ou plusieurs secteurs et la MC. Pour cela, il faut lui fournir : le sens du transfert (R/W), l'adresse de début en MC (Tampon), la taille du transfert, la liste des adresses secteurs (n° face, n° cylindre, n° secteur)
- La plus petite unité de transfert physique est 1 secteur
- Sur les disques récents, le nombre de secteurs par piste est variable

Disque Dur : Temps d'Accès Moyen et Taux de transfert I

- Pour accéder à un secteur donné, le contrôleur doit commencer par translater les bras mobiles portes-têtes sur le bon cylindre, puis attendre que le bon secteur passe sous la tête sélectionnée pour démarrer le transfert. Le temps d'accès moyen caractérise la somme de ces deux délais moyens
- Le volume est le produit du nombre de faces par le nombre de pistes par face par les nombre de secteur par piste par la taille d'un secteur
- Le taux de transfert est le nombre d'octets tansférés à la seconde, une fois que le temps d'accès moyen a permis de se placer sur le bon secteur en supposant que les secteurs du fichier à transférer sont contigüs

Disque Dur : Temps d'Accès Moyen et Taux de transfert II

Exemple : TAmoyen et transfert d'1 secteur

disque dur 16 faces, 16 Kibi cylindres, 256 secteurs/piste de 4 Kio, tournant à 7200 tours/mn, ayant une vitesse de translation de 2 m/s et une distance entre la première et la dernière piste de 5 cm
TAmoyen = (5 cm/2)/2 m/s + (1/(7200/60 t/s))/2 = 12,5 ms+4,2 ms=16,7 ms
Transfert d'1 secteur = (1/(7200/60 t/s))/256 = 32 10^{-3} ms
Taux de transfert = (7200/60 t/s) * 256 * 4 Kio = 120 Mo/s
Volume = $16 * 16 * 2^8 * 256 * 4$ Kio = 4 Tio

Pilote, contrôleur d'E/S et IT I

- A l'origine, l'UC gérait les périphériques en leur envoyant une requête puis en attendant leur réponse. Cette attente active était supportable en environnement monoprogrammé
- Actuellement, l'UC délègue la gestion des E/S aux processeurs situés sur les cartes contrôleur (disque, graphique, . . .)
- La communication avec un périphérique donné est réalisée par le pilote (driver) qui est un module logiciel du SE :
  - la requête d'un processus est transmise au pilote concerné qui la traduit en programme contrôleur puis qu'il envoie à la carte contrôleur
  - le processus courant "s'endort" et l'UC exécute un processus "prêt"
  - le contrôleur exécute l'E/S
  - le contrôleur prévient l'UC de la fin de l'E/S grâce au mécanisme matériel d'interruption

Pilote, contrôleur d'E/S et IT II

  - l'UC traite l'interruption en désactivant le processus en cours d'exécution puis "réveille" le processus endormi qui peut reprendre son exécution
- Grâce à ce fonctionnement, l'UC ne perd pas son temps à des tâches subalternes
- Généralement, plusieurs niveaux d'interruption plus ou moins prioritaires sont admis par l'UC
- Les E/S sont dites bloquantes au sens où le processus reste bloqué tant que la lecture ou l'écriture n'est pas réalisée

Plan

Les Bus de données, d'adresse et de contrôle I

- Le bus de données est constitué d'un ensemble de lignes bidirectionnelles sur lesquelles transitent les bits des données lues ou écrites par le processeur, par exemple (Data 0-31) sur un processeur 32 bits
- Le bus d'adresses est constitué d'un ensemble de lignes unidirectionnelles sur lesquelles le processeur inscrit les bits formant l'adresse désirée, par exemple (Ad 0-35) avec 64 Go adressable en MC. Remarquons que les processeurs d'E/S écrivent également sur le bus d'adresse (synchronisation)
- Le bus de contrôle est constitué d'un ensemble de lignes permettant au processeur de signaler certains événements et d'en recevoir d'autres. On trouve fréquemment des lignes représentant les signaux suivants :
  - Vcc et GROUND : tensions de référence

Les Bus de données, d'adresse et de contrôle II

- Reset : réinitialisation de l'UC
- $R/\overline{W}$: indique le sens du transfert vers la MC
- $MEM/\overline{IO}$: adresse mémoire ou E/S

- Technologiquement, les technologies de bus évoluent rapidement (Vesa Local Bus, ISA, PCI, PCI Express, ATA, SATA, SCSI, . . .)
- Actuellement, d'autres types d'architecture (5e génération, machines systoliques, grid computing) utilisant massivement le parallélisme permettent d'améliorer notablement la vitesse des calculs
- On peut conjecturer que dans l'avenir, d'autres paradigmes de programmation spécifiques à certaines applications induiront de nouvelles architectures

Plan

Hiérarchie mémoire et Cache I

Classiquement, il existe 3 niveaux de mémoire ordonnés par vitesse d'accès et prix décroissant et par taille croissante :

- Registres
- Mémoire Centrale
- Mémoire Secondaire

Afin d'accélérer les échanges, on peut augmenter le nombre des niveaux de mémoire en introduisant des caches de Mémoire Centrale (ou antémémoire) entre registres et MC, et/ou des caches de Mémoire Secondaire entre MC et disque dur.

Hiérarchie mémoire et Cache II

Fonctionnement

- Un processus demande à lire une information (donnée ou instruction) : si le cache possède l'information, l'opération est réalisée par l'UC depuis le cache, sinon, l'unité de cache récupère l'info. depuis la MC puis réalise l'op.
- En cas d'écriture, si la zone écrite est dans le cache, l'UC écrit sur le cache plutôt qu'en MC.
- Dans le cas où la zone accédée n'est pas dans le cache, l'Unité de cache doit procéder à une désallocation dans le cache d'une autre zone peu utilisée (*Least Recently Used, Lest Frequently Used*) puis effectuer cette opération dans la nouvelle zone allouée
- Le principe de **séquentialité** des instructions et des structures de données permet d'optimiser l'allocation du cache avec des segments contigus de MC

Hiérarchie mémoire et Cache III

Exemple de Cache MC réalisé en SRAM

- Niveau 1 (L1) : séparé en 2 caches (instructions, données), situé dans le processeur, communique avec L2
- Niveau 2 (L2) : unique (instructions et données) situé dans le processeur
- Niveau 3 (L3) : existe parfois sur certaines cartes mères

Hiérarchie mémoire et Cache IV

Cache Disque

De quelques Méga-octets, ce cache réalisé en DRAM ou en Flash est géré par le processeur du contrôleur disque. Il ne doit pas être confondu avec les tampons systèmes stockés en mémoire centrale (100 Mio). Intérêts de ce cache :

- Lecture en avant (arrière) du cylindre
- Synchronisation avec l'interface E/S (IDE, SATA, ...)
- Mise en attente des commandes (SCSI, SATA)

Pipeline I

Technique de conception de processeur avec plusieurs petites unités de commande placées en série et dédiées à la réalisation d'une tâche spécifique. Plusieurs instructions se chevauchent à l'intérieur même du processeur
Par exemple, décomposition simple d'une instruction en 5 étapes :

- Fetch : chargement de l'instruction depuis la MC
- Decode : décodage en micro-instructions
- Load : chargement éventuel d'une donnée de MC
- Exec : exécution de l'instruction
- Write Back : écriture éventuelle du résultat en MC

Soit la séquence d'instruction : i1, i2, i3, ...

sans pipeline temps →

i1F i1D i1L i1E i1W i2F i2D i2L i2E i2W i3F ...

Pipeline II

avec pipeline à 5 étages temps →

```
i1F i1D i1L i1E i1W
    i2F i2D i2L i2E I2W
        i3F i3D i3L I3E I3W
            i4F i4D i4L I4E I4W
```

- Chaque étage du pipeline travaille "à la chaîne" en répétant la même tâche sur la série d'instructions qui arrive
- Si la séquence est respectée, et s'il n'y a pas de conflit, le débit d'instructions (*throughput*) est multiplié par le nombre d'étages
- Intel Core i7 possède 14 étages

SIMD, DMA, Bus Mastering I

- *Single Instruction Multiple Data* désigne un ensemble d'instructions vectorielles permettant des opérations scientifiques ou multimédia. Par exemple, l'AMD 64 possède 8 registres 128 bits et des instructions spécifiques utilisables pour le streaming, l'encodage audio ou vidéo, le calcul scientifique. Multiple IMD est l'amélioration de SIMD avec plusieurs processeurs (ou coeurs)
- L'accès direct mémoire ou DMA (*Direct Memory Access*) est un procédé informatique où des données circulant de ou vers un périphérique (port de communication, disque dur) sont transférées directement par un contrôleur adapté vers la mémoire centrale de la machine, sans intervention du microprocesseur si ce n'est pour initier et conclure le transfert. La conclusion du transfert ou la disponibilité du périphérique peuvent être signalés par interruption

SIMD, DMA, Bus Mastering II

- La technique de *Bus Mastering* (contrôle de bus) permet à n'importe quel contrôleur d'E/S de demander et de prendre le contrôle du bus : le maître peut alors communiquer avec n'importe lequel des autres contrôleurs sans passer par l'UC. Cette technique implémentée dans le bus PCI permet à n'importe quel contrôleur de réaliser un DMA. Si l'UC a besoin d'accéder à la mémoire, elle devra attendre de récupérer la maîtrise du bus (pas de temps réel).
  Le *chipset* des PCs utilise la technique de *bus mastering* en étant l'interface entre l'UC, la MC et les bus plus ou moins rapides des périphériques (PCI Express, PCI, USB, . . .)

Architecture Multi-coeurs I

Le processeur possède plusieurs coeurs possédant chacun :

- UAL et FPU
- Unité de commande à pipeline
- registres

Chaque coeur peut posséder un cache dédié (L1), et l'ensemble des coeurs partagent un cache partagé (L2). Chaque coeur est destiné à exécuter un *thread*.

Plan

Plan

La couche Machine I

- Elle constitue le niveau le plus bas auquel l'utilisateur a accès
- Les instructions machines sont codées en binaire soit dans le fichier binaire exécutable, soit en MC dans le segment de code
- Comme chaque instruction est en correspondance avec une instruction en langage d'assemblage (mnémoniques) on utilise souvent ce dernier pour étudier cette couche.

La couche Machine II

Architecture x86-64

- x86-64 : version 64-bit de l'architecture x86 et de son jeu d'instruction
- 2^{64} octets de mémoire virtuelle et de mémoire physique
- registres de travail d'une taille de 64 bits
- le code x86-64 est rétro-compatible avec le code x86 : les anciennes applications peuvent donc s'exécuter
- elles ont intérêt à être reprogrammées afin de bénéficier de meilleures performances
- créée par AMD (AMD64) puis reprise par Intel et VIA
- différents noms (x64, Intel 64) persistent

La couche Machine III

Caractéristiques de x86-64

- registres 64 bits décomposables (AL, AH, AX, EAX, RAX)
- 16 registres : rax, rbx, rcx, rdx, rbp, rsp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15
- Espace adrs : 16 exbibytes
- Mémoire virtuelle non segmentée sur 64 bits
- SIMD avec registres 128 bits
- (petit boutiste) Little endian : les octets de poids forts sont situés dans les adresses mémoires les plus grandes (little en premier). Les PowerPC, MIPS, ARM sont bi-endian. Le protocole TCP/IP est big endian !

Format des instructions machines I

Une instruction est composée de plusieurs champs :

- 1 champ obligatoire : le **code opération** désigne le type d'instruction. Sa longueur est souvent variable afin d'optimiser la place et la vitesse utilisée par les instructions les plus fréquentes (Huffman)
- 0 à 2 champs optionnels : les **opérandes** désignent des données immédiates ou stockées dans des registres ou en MC. Le type de désignation de ces données est nommé **mode d'adressage**

Exemple :

Code Opération	Opérande1	Opérande2	Commentaire
MOV	AX	BX	; AX = BX
NEG	AX		; AX = C2(AX)
JC	ETIQ		; if CF !=0 goto ETIQ

Format des instructions machines II

- La taille d'une instruction est un multiple d'octets
- Plus le jeu d'instruction est grand, plus la taille du code opération augmente (RISC versus CISC)
- La taille et le codage des opérandes dépend de leur mode d'adressage

Plan

Modes d'adressage I

Types de donnée représentés par les opérandes :

- donnée **immédiate** stockée dans l'instruction machine
- donnée dans un **registre** de l'UC
- donnée située à une **adresse** en MC

Remarques :

- Parfois, un opérande est implicite (pas désigné dans l'instruction) ; le Z80 a au maximum un opérande explicite (et A comme opérande implicite).
- Source et Destination : Lorsque 2 opérandes interviennent dans un transfert ou une opération arithmétique, l'un est source et l'autre destination de l'instruction. L'ordre d'apparition varie suivant le type d'UC et le langage d'assemblage :
  IBM 370, x86 : ADD DST, SRC ;DST = DST+SRC
  PDP-11, 68000 : ADD SRC, DST ;DST = DST+SRC

Adressage immédiat I

- La valeur de la donnée est stockée dans l'instruction
- Cette valeur est donc copiée de la MC vers l'UC lors de la phase de chargement (fetch) de l'instruction
- Avantage : pas d'accès supplémentaire à la MC
- Inconvénient : taille limitée de l'opérande

Exemples :

- (Z80) ADD A, <n> ; Code Op. 8 bits, n sur 8 bits en C2
- (Z80) LD <Reg>, <n> ; Code Op 5 b., Reg 3 b., n sur 8 bits en C2
- (x86-64) MOV AX, -28 ; -28 codé sur 16 bits en C2

Adressage registre I

- La valeur de la donnée est stockée dans un registre de l'UC. La désignation du registre peut être explicite ou implicite (A sur Z80)
- Avantage : accès rapide versus MC
- Inconvénient : taille limitée de l'opérande et nombre limité de registres
- Taille de l'opérande dépend du nombre de registres : $log_2(nbregistres)$

Exemples :

- (Z80) ADD A, <n> ; A implicite dans l'instruction machine
- (x86-64) MOV AX, BX ; les deux opérandes sont des registres

Adressage direct I

- La valeur de la donnée est stockée à une adresse en MC
- C'est cette adresse qui est représentée dans l'instruction et la donnée est chargée pendant la phase de *load* (si lecture)
- Avantage : taille quelconque de l'opérande
- Inconvénient : accès mémoire supplémentaire, taille importante de l'instruction
- Selon la gestion de la mémoire, plusieurs adressages directs peuvent coexister ou pas

Exemples :

- (x86-64) MOV MAVAR, 123 ; MAVAR est codé par son adresse dans le segment de donnée statique (write back)

Adressage direct II

- (8086) MOV AL, <dis> ; déplacement intra-segment (court) : transfère dans le registre AL, l'octet situé à l'adresse dis dans le segment de données : dis est codé sur 16 bits, Data Segment est implicite
- (x86-64) JMP ETIQ ; saut direct à l'adresse ETIQ
- (8086) ADD BX, <aa> ; adresse absolue aa= S,D : ajoute à BX, le mot de 16 bits situé à l'adresse D dans le segment S ; D et S sont codés sur 16 bits.
- L'IBM 370 n'a pas de mode d'adressage direct, tandis que le 68000 permet l'adressage direct court (16 bits) et long (32 bits).

Adressage indirect I

- La valeur de la donnée est stockée à une adresse m en MC
- Cette adresse m est stockée dans un registre d'adrs r ou à une adresse m'
- C'est r ou m' qui est codé dans l'instruction (m' est appelé un pointeur)
- L'adressage indirect par registre est présent dans la totalité des UC
- Par contre, l'adressage indirect par mémoire est peu fréquent (vecteur d'interruption)
- Il peut être simulé par un adressage direct dans un registre suivi d'un adressage indirect par registre.
- Avantage : taille quelconque de l'opérande

Adressage indirect II

- Inconvénient : accès mémoire supplémentaire (load, write back), taille importante de l'instruction (sauf si registre)

Exemples :

- (Z80) ADD A, (HL) ; adressage indirect seulement par registre HL ; codage de l'instruction sur 8 bits (code op.) : A et HL sont désignés implicitement
- (x86) MOV AL, [BX] ; adressage indirect par registre pointeur BX
- (asm GNU) `movl -4(%ebp), %eax` ; [ebp-4] dans eax (source to dest)

Adressage indexé (ou basé) I

- Objectif : accéder à des données situées à des adresses successives en MC (tableau, struct, instances)
- Adressage indexé par registre : le registre d'index est chargé avec l'adresse de début de la zone de données, puis dans l'instruction, le déplacement relatif est codé : `MOV AX, [BP+4]`
- Certaines instructions exécutent automatiquement l'incrémentation ou la décrémentation de leurs registres d'index afin de réaliser des transferts ou d'autres opérations sur des séquences (chaînes de caractères) en itérant
- Avantage : taille importante de la zone adressable (256 octets si déplacement sur 8 bits)
- Inconvénient : taille importante de l'instruction (codage du déplacement)

Adressage indexé (ou basé) II

Exemples :

Z80 indexation par IX et IY
LD (IX+<dépl>), <reg> ; adressage indexé par IX codage de l'instruction : code op. sur 12 bits, IX sur 1 bit, reg sur 3 bits, dépl sur 8 bits.

8086 MOVSB (MOVe String Byte) permet de transférer l'octet en (SI) vers (DI) puis d'incrémenter ou décrémenter SI et DI. Remarquons que l'indexation sans déplacement équivaut à l'indirection.

Remarques I

- Dans une instruction, lorsque deux opérandes sont utilisés, deux modes d'adressages interviennent
- Certains modes sont incompatibles avec d'autres : souvent un seul adressage mémoire par instruction (direct ou indirect ou indexé)
- L'adressage basé est un synonyme d'adressage indexé
- Toute indirection à n niveaux peut être simulée dès lors qu'on possède une instruction fournissant l'indirection

Plan

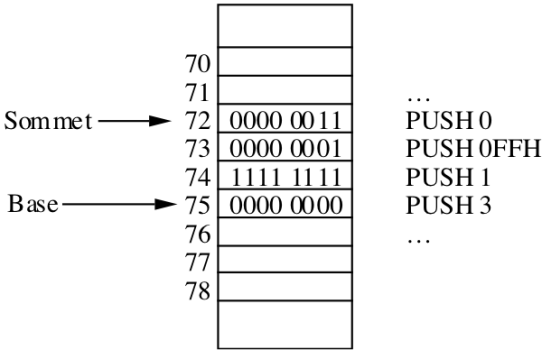
Adressage par pile I

- La pile d'exécution de l'UC est constituée d'une zone de la MC dans laquelle sont transférés des mots selon une stratégie Dernier Entré Premier Sorti (Last In First Out LIFO)
- Le premier élément entré dans la pile est placé à la **base** de la pile et le dernier élément entré se situe au **sommet** de la pile
- Des registres spécialisés (**SP** Stack pointer, **BP** Base Pointer) pointant sur le sommet et la base d'un **bloc de pile** (stack frame)
- Des instructions spécialisées de manipulation de pile : `PUSH <n ou reg>` pour empiler ; `POP <reg>` pour dépiler le sommet de pile
- Selon l'UC, la pile remonte vers les adresses faibles ou bien descend vers les adresses fortes et la taille des mots varie

Adressage par pile II

- L'utilisation de la pile est **indispensable** car elle permet l'appel procédural (fonctions, méthodes, ...)
- sa cohérence nécessite égalité du nombre d'empilements et de dépilements (programmeur en assembleur ou compilateur)
- Lors d'une récursivité infinie, un débordement de pile (stack overflow) survient

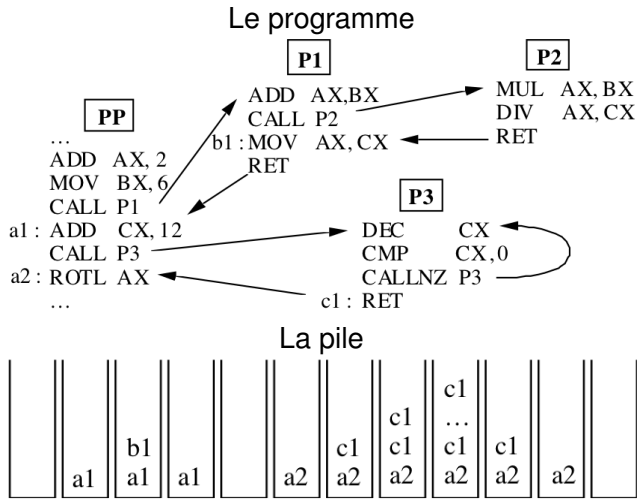
Adressage par pile III



L'appel procédural I

- Procédures : concision, modularité, réutilisation ; historiquement sous-routines utilisant un registre de retour !
- Le programme principal (PP ou `main`) fait appel (`CALL P1`) à une procédure P1 qui exécute sa séquence d'instructions puis rend la main en retournant (`RET`) à l'instruction du PP qui suit l'appel
- Le `CALL` réalise un push du compteur ordinal avant de `JMP`er au début de P1 ; Le `RET` fait un pop dans le compteur ordinal
- Cette rupture de séquence avec retour doit également pouvoir être réalisée dans n'importe quelle procédure vers n'importe laquelle y compris le `main`

Exemple I



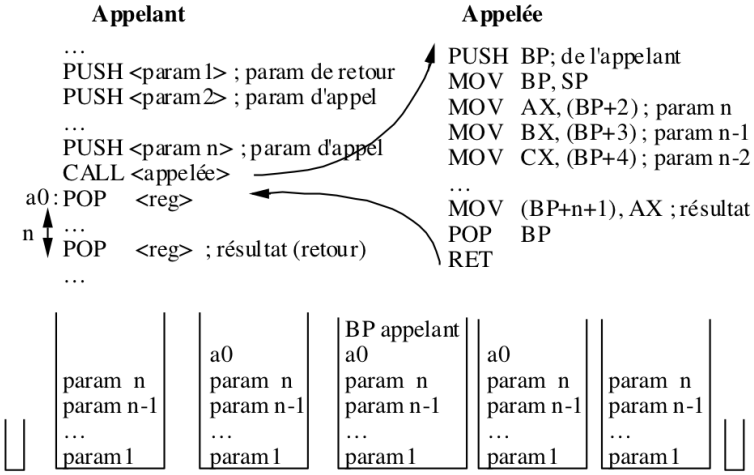
Paramètres et Variables locales I

- Les langages de programmation évolués (Java, C, . . .) permettent le passage de paramètres **données** et/ou **résultats** entre appelant et appelé
- L'utilisation des registres de l'UC est la méthode la plus efficace mais ne suffit pas lorsque le nombre et la complexité des paramètres augmente
- Un compilateur doit fournir une gestion générique des paramètres quel que soit leur nombre et leur mode de passage
- La pile est utilisée dans l'appelante, avant l'appel (CALL) : le compilateur génère des instructions d'empilement (PUSHs) des paramètres d'appel et de retour
- Dans l'appelante, juste après le CALL, il génère le même nombre de dépilements afin de nettoyer la pile

Paramètres et Variables locales II

- Au début de l'appelée, l'initialisation de son **bloc de pile** consiste à affecter au registre de base (BP) la valeur du sommet de pile. Par la suite, les références aux paramètres sont effectuées via ce registre [BP+0..n]
- Durant son exécution, une instance de procédure ne doit en aucun cas modifier son registre de base de pile au risque de ne plus retrouver ses paramètres
- Ce registre doit donc être sauvegardé (dans la pile) en début de toute procédure et restaurée en fin de toute procédure (`PUSH BP ... POP BP`)

Exemple de passage de paramètres I



Variables locales ou automatiques I

- Les variables **locales à une procédure** sont créées à l'activation de la procédure et détruites lors de son retour (durée de vie)
- D'autre part, leur visibilité est réduite aux instructions de la procédure.
- L'implémentation de l'espace dédié aux variables locales est réalisée dans la pile d'exécution au dessus du `BP` de l'appelante
- L'espace pour ces variables est réservé mais **non initialisé**
- Ces variables locales seront ensuite accédées via des adressages [BP-i] générés par le compilateur
- En C, les paramètres sont passés toujours **par valeur**, un nom de tableau étant l'adresse de sa première case

Un exemple complet I

- Une fonction récursive simple n'utilisant que des paramètres et variables locales codés sur un mot machine
- La fonction `mult` réalise la multiplication de 2 entiers positifs par additions successives

```
entierpositif mult(entierpositif x, entierpositif y)
entierpositif i ; // var locale
si x=0 alors
  retourne 0
sinon
  début
    i=y
    retourne mult(x-1,i) + y
  fin

main()  afficher(mult(2,5))
```

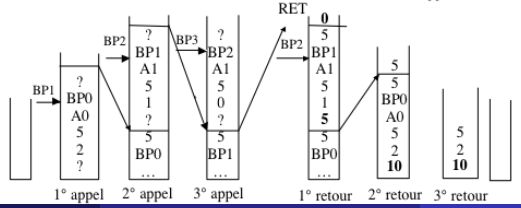
Un exemple complet II

Appelant

```
…
PUSH <multret> ; param de retour
PUSH  x ; param d'appel 2
PUSH  y ; param d'appel 5
CALL  mult
A0: POP   DX
POP   DX
POP   AX  ; résultat (retour)
CALL  AFFICHEAX
…
```

Mult

```
PUSH  BP          ; de l'appelant
MOV   BP, SP
PUSH  DX          ; i qcq en (BP-1)
CMP   (BP+3), 0   ; x=0 ?
JE    ZERO
MOV   (BP-1),(BP+2)   ; i:=y
MOV   AX, (BP+3)      ; x
DEC   AX              ; x-1
PUSH  DX          ; retour qcq
PUSH  AX          ; appel x-1
PUSH  (BP-1)      ; appel i
CALL  mult        ; récursif
A1: POP   DX   ; dépile
POP   DX   ; dépile appel
POP   AX   ; résultat (retour)
ADD   AX, (BP+2)   ; résult + y
MOV   (BP+4), AX   ; range résultat
JMP   FIN
ZERO : MOV (BP+4),0 ; range rés.=0
FIN :   POP DX ; var locale i
POP   BP          ; BP appelant
RET
```



Conclusion I

- L'exemple précedent illustre le danger de croissance de la pile lors d'appels récursifs mal programmés
- Remarquons que la dérécursivation évidente de mult peut être réalisée par le programmeur mais souvent aussi par le compilateur
- L'utilisation de la **trace de pile** en débogage permet de savoir où est située l'erreur dans le programme
- Autres utilisations de la pile : automates à pile en analyse syntaxique (parsing), parcours d'arbre (préfixe, infixe, postfixe), ...

Plan

Plan

Les Fichiers : définitions I

conceptuelle Un fichier est une collection organisée d'informations de même nature regroupées en vue de leur conservation et de leur utilisation dans un Système d'Information (agenda, catalogue de produits, répertoire téléphonique, ...)

logique C'est une collection ordonnée d'articles (enregistrement logique, item, "record"), chaque article étant composés de champs (attributs, rubriques, zones, "fields"). Chaque champ est défini par un nom unique et un domaine de valeurs. Remarque : Selon les SE, la longueur, le nombre, la structure des champs est fixe ou variable. Lorsque l'article est réduit à un octet, le fichier est qualifié de **non structuré**

Les Fichiers : définitions II

physique Un fichier est stocké dans une liste de blocs (enregistrement physique, granule, unité d'allocation, "block", "cluster") situés en mémoire secondaire. Les articles d'un même fichier peuvent être groupés sur un même bloc (Facteur de groupage ou de Blocage (FB) = nb d'articles/bloc) mais on peut aussi avoir la situation inverse : une taille d'article nécessitant plusieurs blocs. En aucun cas, un article de taille inférieure à la taille d'un bloc n'est partitionné sur plusieurs blocs : lecture 1 article = 1 E/S utile. Les blocs de MS sont alloués à un fichier selon différentes méthodes liées au Système de Fichier (NTFS, e4fs, VFAT, ...).

Opérations et modes d'accès I

création création et initialisation du noeud descripteur (i-node, File Control Block, Data Control Block) contenant taille, date modif., créateur, adrs bloc(s), ...

destruction désallocation des blocs occupés et suppression du noeud descripteur (sans effacement des données sur les blocs)

ouverture réservation de tampons d'E/S en MC pour le transfert des blocs ; l'ouverture est souvent associée à un mode d'accès indiquant la ou les opérations réalisables par la suite (RDONLY, WRONLY, APPEND, RDWR, ...)

fermeture recopie des tampons MC vers MS (sauvegarde) puis désallocation des tampons

lecture consultation d'un ou plusieurs articles

écriture insertion ou suppression d'un article

Opérations et modes d'accès II

déplacement déplacement sur un article

droits d'accès un SE multi-utilisateurs doit toujours vérifier les droits de l'utilisateur lors de l'ouverture d'un fichier

La suite de ce chapitre détaille la façon dont ces principes généreaux sont implémentés dans le noyau Unix, et plus généralement encore dans la bibliothèque standard du langage C qui est portée sur tous les systèmes d'exploitation.

Plan

Création de fichier I

Unix définit dans son noyau des fonctions (man 2 ...) de base d'accès à des fichiers **non structurés** permettant l'accès **séquentiel** ainsi que le déplacement à une position quelconque (si le support le permet).
Création de fichier : ouverture avec des paramètres spécifiques

```
int open(char *path, O_WRONLY|O_CREAT
         |O_TRUNC, int droits)
```

- path : nom ou chemin d'accès au fichier ("../Monrep/toto.txt")
- droits : droits d'accès qui seront masqués par `umask` (0644)
- return : un descripteur entier positif ou -1 si erreur

Création de fichier II

Par exemple

```
int f=open("../toto/ficessai", O_WRONLY
          |O_CREAT|O_TRUNC, 0640);
```

- crée un fichier vide s'il n'existait pas sinon le vide
- la tête de lecture/écriture est à 0 (début du fichier)
- si le fichier existait déjà, il conserve ses anciens droits d'accès

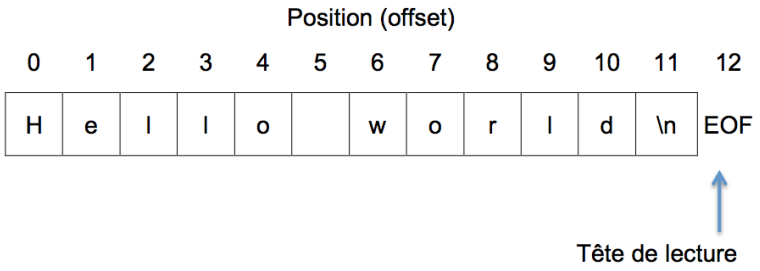
Ecriture dans un fichier

```
int write(int desc, char *buf, int nboctets)
```

- desc : le descripteur retourné par `open`
- buf : la chaîne de caractères qu'on veut écrire
- nboctets : le nombre d'octets qu'on tente d'écrire
- return : le nb d'octets écrits dans le fichier et dont a avancé la tête de lecture, -1 si erreur
- surcharge : les octets écrits écrasent ceux qui existaient auparavant ; si on est en fin de fichier, ce dernier est allongé automatiquement

Un exemple complet

```
int f=open("essai.txt", O_WRONLY|O_CREAT|O_TRUNC, 0640);
if(f==-1){
  fprintf(stderr, "Impossible de créer le fichier !\n");
  exit(1);
}
char *s="Hello world\n";
write(f,s,strlen(s)); close(f);
```



Ouverture, fermeture

```
int open(char *path, int flags[, int droits])
```

- **path** nom ou chemin d'accès au fichier
- **flags** `O_RDONLY` xor `O_WRONLY` xor `O_RDWR`, `O_APPEND`, `O_TRUNC`
- **droits** droits d'accès qui seront masqués par `umask` (0644)
- **return** un descripteur entier positif ou -1 si erreur

ouvre un fichier selon un mode (R|W|RW) et positionne le pointeur en début (fin si `O_APPEND`) de fichier

```
int close(int desc)
```

ferme le fichier (désalloue les tampons systèmes)

Lecture

```
int read(int desc, char *buf, int nboctets)
```

- **desc** le descripteur retourné par `open`
- **buf** la chaîne de caractères **allouée** dans laquelle vont être stockés les octets lus
- **nboctets** le nombre d'octets qu'on tente de lire
- **return** le nb d'octets lus dans le fichier et dont a avancé la tête de lecture, 0 si fin de fichier, -1 si erreur

Déplacement de la tête de lecture/écriture (accès Direct)

```
off_t lseek(int fd, off_t offset, int whence)
```

- **fd** le descripteur retourné par `open`
- **offset** le déplacement à effectuer (entier signé long)
- **whence** `SEEK_SET` xor `SEEK_CUR` xor `SEEK_END`, position à partir de laquelle se déplacer (début, courante, fin)
- **return** nouvelle position courante ou -1 si erreur

Autres appels systèmes I

int dup(int desc) duplication de descripteur (redirection d'E/S) : d=creat("ficredir", 0666) ; close(1) ; dup(d) ;

access teste les droits d'accès

link, symlink création de lien dur ou symbolique

unlink suppression d'un lien et possiblement du fichier

stat retourne le contenu du i-noeud d'un fichier (lstat, fstat)

chdir change répertoire courant

chown, chmod change propriétaire, droits d'accès

mkdir, rmdir création, suppression de répertoire

Plan

Bibliothèque versus appels systèmes

Il est préférable d'utiliser les fonctions de la bibliothèque standard C (lorsqu'elles existent) plutôt que d'utiliser les appels noyaux Unix de bas niveau pour les raisons suivantes :

- portabilité des programmes sur différents systèmes d'exploitation ;
- optimisation du nombre d'E/S grâce au tamponnement ;
- facilité d'utilisation notamment E/S formattées ;
- programmation de plus haut niveau donc réutilisabilité et maintenance favorisée ;

Flot (stream)

Le type `FILE` permettant de manipuler les fichiers par des `FILE *` souvent nommés flots « stream ». Ce qui suit est décrit dans le manuel Unix section 3 (man 3) mais est indépendant de ce système d'exploitation.

- les E/S sont tamponnées dans des tampons utilisateurs : en écriture, le vidage du tampon dans le fichier est réalisé par char de synchro `'\n'`, par appel à `fflush`, ou lorsque le tampon est plein
- 3 macros : `stdin, stdout, stderr` pour descripteurs 0, 1, 2
- constante EOF (-1) : entier retourné lorsque fin de fichier

Liste des fonctions de la bibliothèque standard C I

```
FILE *fopen(char *path, char *mode)
```

path chemin d'accès au fichier

mode
- "r" lecture
- "r+" lecture et écriture
- "w" création ou troncature du fichier ouvert en écriture seulement (droits 0666 en création)
- "w+" création ou troncature du fichier ouvert enlecture/écriture
- "a" ouverture en écriture seulement, création si nécessaire, position en fin de fichier
- "a+" ouverture en lecture/écriture, création si nécessaire, écritures en fin de fichier, lecture en début

Liste des fonctions de la bibliothèque standard C II

```
int fclose(FILE* f)
```

vidage du tampon sur le fichier

```
int [f]getc(FILE *g); int [f]putc(char c,FILE *g);
int getchar(); int putchar(c)
```

lit/écrit un caractère du fichier g ; sans f : stdin, stdout

```
int [f]gets(char *ch, int n, FILE *f)
```

lit une chaîne : min(nb char jusqu'à `'\n'`, n-1) char + `'\0'` sont copiés dans ch. Attention, gets remplace le `'\n'` par `'\0'` mais pas fgets !

```
int [f]puts(char *ch, FILE *f)
```


Liste des fonctions de la bibliothèque standard C III

copie ch dans f, sauf le `'\0'` final

```
int fseek(FILE *f, long depl, int base)
```

déplace le pointeur de fichier ; base==0 (1,2) : déplacement depuis le début (courant, fin) de f

```
int feof(FILE* f)
```

retourne vrai (!=0) si le pointeur est en EOF

```
int fflush(FILE *f)
```

vidage du tampon en écriture vers le fichier

```
int fpurge(FILE *f)
```


Liste des fonctions de la bibliothèque standard C IV

RAZ des tampons en écriture et en lecture sans utilisation du contenu !

```
int [f]printf(FILE *f, format, listeVals)
```

printf pour stdout ; écriture formatée : `%s` chaîne, `%c` car, `%d` entier décimal, `%x` hexa `%10.2f` pour convertir un flottant en chaîne de 10 char dont 2 décimales

```
int [f]scanf(FILE *f, format, listePtrs)
```

scanf pour stdin ; lecture selon un certain format dans un fichier

```
int sprintf(char *ch, format, liste_vals)
```

écriture formatée dans une chaîne.

Liste des fonctions de la bibliothèque standard C V

```
int sscanf(char *ch, format, liste_ptrs)
```

lecture selon un certain format dans une chaîne

```
int strlen(char *s)
```

longueur de s

```
char *strcat(char *dst, *src);
char *strncat(char *dst, *src, int n)
```

concaténation (bornée par n) ; penser à l'allocation

```
strcmp(char *s1,*s2);   strncmp(char *s1, *s2, int n)
```

comparaison (bornée par n) ; 0 si égales

Liste des fonctions de la bibliothèque standard C VI

```
char *strcpy(char* dst, *src);
char *strncpy(dst,src,n)
```

copie (bornée par n)

```
char*strchr(char *s,char c)
```

recherche du 1er c dans s

```
char *strpbrk(char *s1, char *s2)
```

recherche d'un char de s2 dans s1

```
char *strstr(char *meule, const char *aiguille);
```

recherche d'une sous-chaîne (facteur)

Plan

Conclusion

Il est absolument indispensable de comprendre les concepts de base des machines informatiques et des systèmes d'exploitation afin :

- de manipuler différents systèmes d'exploitation en comprenant leurs différences et leurs ressemblances
- d'évaluer correctement les résultats numériques fournis par les machines (précision)
- de programmer intelligemment les algorithmes dont on a minimisé la complexité (des E/S fréquentes peuvent ruiner un algorithme d'une complexité inférieure à un autre)
- de pouvoir optimiser les parties de programme les plus utilisées en les réécrivant en langage de bas niveau (C)
- d'écrire des compilateurs ou des interpréteurs performants même si ceux-ci sont écrits en langage de haut niveau
- de se préparer à la programmation concurrente
- d'oser utiliser des systèmes d'exploitation dont le code source est connu !