

# Notes LINFO1252

Quentin Bodart

Q1 2024-2025

## Contents

<b>1</b>	<b>CM 1</b>	<b>3</b>
1.1	Le système informatique et le rôle du système d'exploitation . . . .	3
1.1.1	Fondamentaux . . . . .	3
1.1.2	Architecture de von Neumann . . . . .	3
1.1.3	Fonctionnement d'un système informatique . . . . .	4
1.1.4	Traitement d'une interruption . . . . .	4
1.1.5	Accès direct à la mémoire . . . . .	4
1.1.6	Système informatique complet . . . . .	5
1.1.7	Rôle du système d'exploitation . . . . .	5
1.1.8	Virtualisation . . . . .	5
1.1.9	Séparation entre mécanisme et politique . . . . .	5
1.1.10	Modes d'exécution . . . . .	6
1.1.11	Appel système . . . . .	6
1.2	Utilisation de la ligne de commande . . . . .	6
1.2.1	Utilitaires UNIX . . . . .	6
1.2.2	Shell / Interpréteur de commandes . . . . .	7
1.2.3	Flux et redirections . . . . .	7
1.2.4	Scripts . . . . .	8
<b>2</b>	<b>CM2 : langage C et compléments</b>	<b>10</b>
2.1	Le langage C . . . . .	10
2.1.1	Préprocesseur . . . . .	10
2.1.2	headers . . . . .	10
2.1.3	Intégration avec le shell . . . . .	11
2.1.4	Affichage formaté . . . . .	11
2.2	Types de données . . . . .	12
2.2.1	Nombres signés et flottants . . . . .	12
2.2.2	Caractères . . . . .	13
2.2.3	Chaînes de caractères . . . . .	13
2.2.4	Pointeurs . . . . .	14
2.2.5	Pointeurs vers une fonction . . . . .	14
2.2.6	Manipulation de bits . . . . .	14

<b>3</b>	<b>CM3 : Gestion de la mémoire</b>	<b>15</b>
3.1	Organisation d'un programme Linux en mémoire . . . . .	15
3.2	Gestion de la mémoire dynamique . . . . .	16
3.2.1	Objectif d'un algorithme de gestion de mémoire dynamique .	16
3.2.2	Implémentation de l'algorithme . . . . .	16
<b>4</b>	<b>CM4 : Structure des ordinateurs</b>	<b>18</b>
4.1	Organisation d'un système informatique . . . . .	18
4.1.1	Mémoire du processeur . . . . .	18

## Objectifs du cours

- utiliser et comprendre les systèmes informatiques (i.p. GNU/Linux)
- utiliser les services fournis par les SE (systèmes d'exploitation)
- design et mise en oeuvre d'un SE

## 1 CM 1

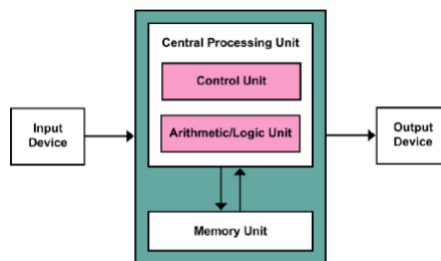
### 1.1 Le système informatique et le rôle du système d'exploitation

Syllabus : <https://sites.uclouvain.be/SystInfo/notes/Theorie/intro.html>

#### 1.1.1 Fondamentaux

- Composants :
  - CPU / Processeur
  - Mémoire Principale (RAM)
  - Dispositifs d'entrée/sortie (y.c. de stockage)
- Fonctionnement d'un CPU
  - Lire / écrire en mémoire vers / depuis des registres
  - Opérations (calculs, comparaisons) sur ces registres
- Jeux d'instructions
  - x86\_64 (PC, anciens Mac)
  - ARM A64 (Raspberry PI, iPhone, nouveaux Mac M1-3)

#### 1.1.2 Architecture de von Neumann

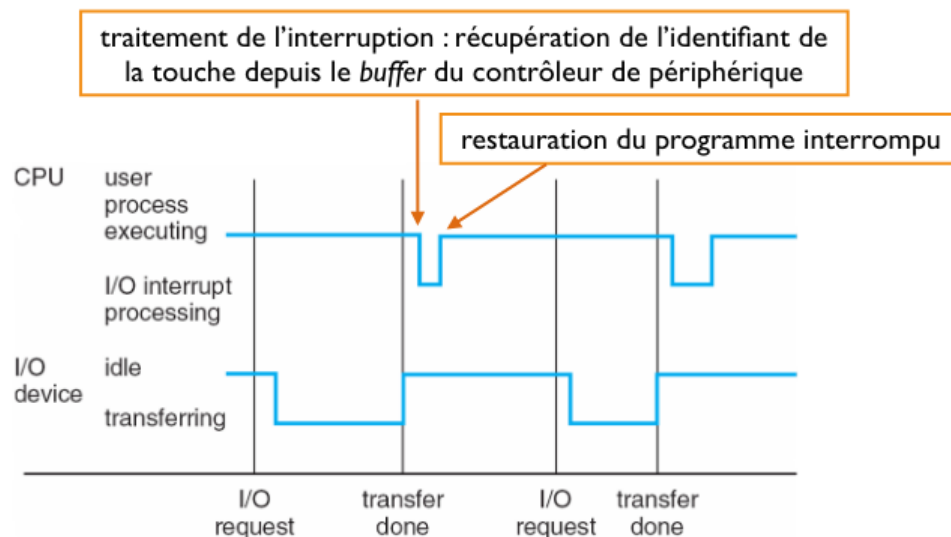


### 1.1.3 Fonctionnement d'un système informatique

- La représentation des données se fait en **binaire**.
- Les opérations d'entrée-sortie se déroulent de manière concurrente.
- Il y a des contrôleurs distincts contrôlant chacun un type de périphérique.
- Chaque contrôleur possède une mémoire dédiée (buffer)
- Le processeur doit déplacer des données depuis/vers la mémoire principale depuis/vers ces buffers dédiés
- Le processeur suit un "fil" continu d'instructions
- Le contrôleur de périphérique annonce au processeur la fin d'une opération d'entrée/sortie en générant une interruption (signal électrique à destination du processeur)

### 1.1.4 Traitement d'une interruption

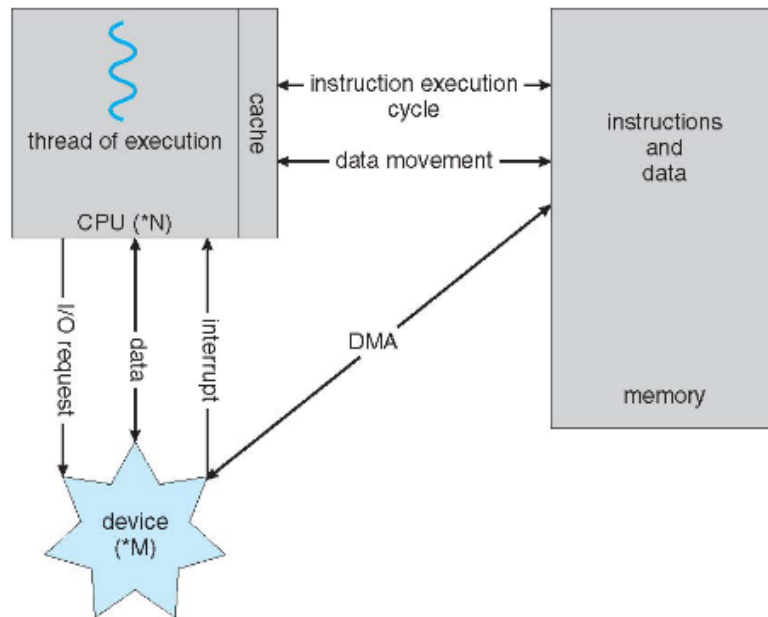
Le processeur interrompt le fil d'exécution d'instructions courant et transfère le contrôle du processeur à une routine de traitement. Cette même routine détermine la source de l'interruption, puis restaure l'état du processeur et reprend le processus :



### 1.1.5 Accès direct à la mémoire

Direct Memory Access (DMA) désigne le fait de ne pas faire une interruption à chaque octet lu depuis un disque dur. Une interruption est tout de même faite à la fin du transfert d'un **bloc**.

### 1.1.6 Système informatique complet



### 1.1.7 Rôle du système d'exploitation

Programmer directement au-dessus du matériel, gérer les interruption, etc... serait une trop grosse tâche pour le programmeur.

3 rôles principaux

- Rendre l'utilisation et le développement d'applications plus simple et plus universel (portable d'une machine à une autre)
- Permettre une utilisation plus efficace des ressources
- Assurer l'intégrité des données et des programmes entre eux (e.g., un programme crash mais pas le système)

### 1.1.8 Virtualisation

Le système d'exploitation **virtualise** les ressources matérielles afin de fonctionner de la même manière sur des systèmes avec des ressources et composants fort différents. Chaque SE doit trouver un compromis entre abstraction et efficacité !

### 1.1.9 Séparation entre mécanisme et politique

- Un mécanisme permet le partage de temps

- Une politique arbitre entre les processus pouvant s'exécuter et le(s) processeur(s) disponibles

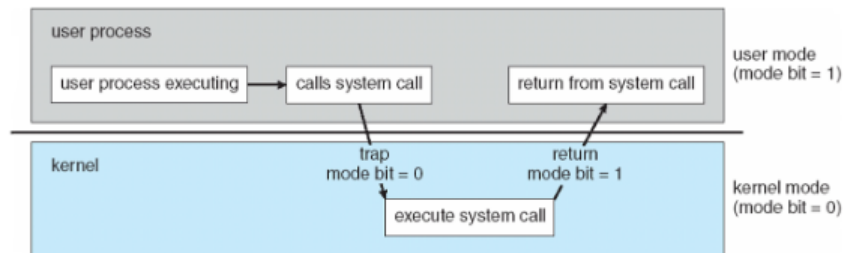
On peut définir des politiques d'ordonnancement différentes selon les contextes, mais sur la base du même mécanisme.

#### 1.1.10 Modes d'exécution

- mode utilisateur : programme utilisant les abstractions fournies par le SE ; certaines instructions sont interdites, comme par exemple:
  - Accès à une zone mémoire non-autorisée (SegFault)
  - De manière générale, toutes les instructions permettant de changer la configuration matérielle du système, comme la configuration ou la désactivation des interruptions
- mode protégé : utilisé par le noyau du SE, toutes les instructions sont autorisées
- L'utilisation de fonctionnalités du SE par un processus utilisateur nécessite de passer d'un mode à l'autre : **appel système**

#### 1.1.11 Appel système

- Un appel système permet à un processus utilisateur d'invoquer une fonctionnalité du SE
- Le processeur interrompt le processus, passe en mode protégé, et branche vers le point d'entrée unique du noyau :



## 1.2 Utilisation de la ligne de commande

**Syllabus :** <https://sites.uclouvain.be/SystInfo/notes/Theorie/shell/shell.html>

### 1.2.1 Utilitaires UNIX

La philosophie lors de la création des utilitaires UNIX était de créer des outils les plus simples possible, donc d'avoir une seule tâche par outil :

## Quelques utilitaires standard

Utilitaire	Fonction
<code>cat</code>	lire/afficher le contenu d'un fichier ex : <code>cat fichier.txt</code>
<code>echo</code>	afficher une chaîne de caractères passée en argument, ex : <code>echo "Bonjour Monde"</code>
<code>head / tail</code>	affiche le début resp. la fin d'un fichier ex : <code>tail errors.log</code>
<code>wc</code>	compte le nombre de caractères / de lignes d'un fichier. ex : <code>wc -l students.dat</code>
<code>sort</code>	trie un fichier. ex : <code>sort -n -r scores.txt</code>
<code>uniq</code>	extraire les lignes uniques ou dupliquées d'un fichier <b>trié</b> fourni en argument. ex : <code>uniq -d students.dat</code>

Afin d'en savoir plus sur

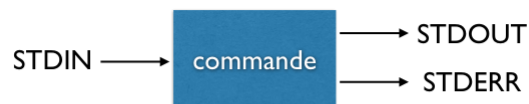
une commande, il suffit d'utiliser l'utilitaire 'man'.

### 1.2.2 Shell / Interpréteur de commandes

Rend possible l'interaction avec le SE. Il en existe plusieurs, mais le principal est 'bash'. Il est toujours complémentaire à une **interface graphique**.

### 1.2.3 Flux et redirections

#### Flux standards et redirections



- 3 flux standards (1 entrée, 2 sorties)
- Redirections permettent de combiner des commandes en utilisant des fichiers intermédiaires

<code>&lt; file</code>	redirige le contenu de file vers STDIN
<code>&gt; file</code>	redirige STDOUT vers file (si le fichier n'existe pas, il est créé, si il existe déjà son contenu est écrasé)
<code>&gt;&gt; file</code>	redirige STDOUT vers file (si le fichier n'existe pas, il est créé, si il existe déjà le contenu est ajouté à la suite du fichier)
<code>2&gt;&amp;1</code>	redirige STDERR vers STDOUT

Exemple de redirections :

```
$ echo "Un petit texte" | wc -c
15
$ echo "bbbb ccc" >> file.txt
$ echo "aaaaa bbbbb" >> file.txt
$ echo "bbbb ccc" >> file.txt
$ cat file.txt
bbbb ccc
aaaaa bbbbb
bbbb ccc
$ cat file.txt | sort | uniq
aaaaa bbbbb
bbbb ccc
```

### 1.2.4 Scripts

Un système UNIX peut exécuter du langage machine ou des **langages interprétés**

Un script commence par convention par les symboles `#!/`, référant à l'interpréteur, ici `bin/bash` :

```
#!/bin/bash
echo "Hello, world"
```

Ils peuvent aussi contenir des **variables**:

```
#!/bin/bash
PROG="LINFO"
COURS=1252
# concaténation
echo $PROG$COURS
```

# : commentaire

accès à la valeur de la variable avec `$`

⚠ une variable non déclarée vaut "" (chaîne vide)

⚠ ambiguïté possible ?

milieu = "mi"; echo do\$milieuno  
utiliser des `{ }` pour délimiter le nom de la variable :  
echo do\${milieu}no

et des **conditionnelles** :



## Conditionnelles (if)

⚠ espace important (source de bug commune)

```
#!/bin/bash
# Vérifie si les deux nombres passés en arguments sont égaux
if [ $# -ne 2 ]; then
    echo "Erreur, deux arguments sont nécessaires" > /dev/stderr
    exit 2
fi
if { $1 -eq $2 }; then
    echo "Nombres égaux"
else
    echo "Nombres différents"
fi
exit 0
```

⚠ ; ou passage à la ligne (pas les 2)

`$1 -eq $2` est vraie lorsque les deux variables `$1` et `$2` contiennent le même nombre.  
`$1 -lt $2` est vraie lorsque la valeur de la variable `$1` est numériquement strictement inférieure à celle de la variable `$2`  
`$1 -ge $2` est vraie lorsque la valeur de la variable `$1` est numériquement supérieure ou égale à celle de la variable `$2`  
`$s = $e` est vraie lorsque la chaîne de caractères contenue dans la variable `$s` est égale à celle qui est contenue dans la variable `$e`  
`= $e` est vraie lorsque la chaîne de caractères contenue dans la variable `$s` est vide

et des **boucles for**:

```
#!/bin/bash
# exemple for.sh
students="Julie Maxime Hakim"
for s in $students; do
    l=`wc -l TP1-$s.txt | cut -d' ' -f1`
    echo "Bonjour $s, ton compte rendu de TP comporte $l lignes."
done
```

- `s` prend successivement les valeurs présentes dans la liste d'entrée `$students`
- ``command`` permet d'assigner la sortie d'une commande à une variable
  - que fait la commande composée ci-dessus ?
- boucles `while` et `until` :  
principe similaire, avec une condition d'arrêt booléenne

## 2 CM2 : langage C et compléments

### 2.1 Le langage C

Jusqu'en 1970, les OS étaient codés en assembleur, ce qui était très chronophage et difficile à maintenir, d'où la nécessité d'un langage plus haut niveau, mais toujours proche de la machine : le C. Son objectif était de concevoir un langage :

- Haut niveau (structuré)
- Proche du matériel (accès direct à la mémoire et traduction direct au processeur)
- Efficace (non-interprété et sans machine virtuelle)
- Portable

#### 2.1.1 Préprocesseur

Le code C est tout d'abord transformé par un préprocesseur avant d'être compilé en langage machine par le compilateur.

Les directives telles que "`#include`" ou "`#define`" y sont remplacées par du texte spécifique à la commande.

C'est seulement après ça que le code est compilé en code machine.

#### 2.1.2 headers

Les directives `#include` du préprocesseur permettent d'importer des définitions de fonctions, de constantes, etc. d'une librairie

Séparation entre fichiers headers (`.h`) et sources (`.c`) :

- Le header contient la définition des fonctions, suffisante pour savoir comment générer le code pour les appeler
- Le code source contient la mise en œuvre de ces fonctions
- `#include "monheader.h"` pour inclure un header du dossier courant (du même projet)
- `#include <header.h>` pour inclure un header standard

### 2.1.3 Intégration avec le shell

15

### Intégration avec le shell

- Point d'entrée : `main`

```

//=====
* cmdline.c
*
* Programme affichant ses arguments
* sur la sortie standard
*
//=====
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int i;
    printf("Ce programme a %d argument(s)\n", argc);
    for (i = 0; i < argc; i++)
        printf("argument[%d] : %s\n", i, argv[i]);
    return EXIT_SUCCESS;
}

```

nombre d'arguments

arguments

`argv[0]` : nom du programme exécuté par l'utilisateur :

```

Ce programme a 5 argument(s)
argument[0] : ./cmdline
argument[1] : 1
argument[2] : -list
argument[3] : abcddef
argument[4] : llnfol252

```

valeur de retour (#? pour la lire en bash)  
`EXIT_FAILURE` en cas d'échec

UCLouvain École polytechnique de Louvain Systèmes Informatiques — E. Rivière

Le tableau d'argument donné à `main` se termine toujours par `NULL`.

### 2.1.4 Affichage formaté

Pour afficher un String formaté en C, on peut utiliser la fonction `printf`:

16

### Affichage formaté avec `printf`

- En Java
  - `System.out.println("Bonjour " + student.getName() + " !");`
- En C avec `printf` : utilisation de codes de formats, remplacés par une version formatée de la valeur de la variable (ou constante) utilisée :

**Input:** `printf("Color %s, Number %d, Float %.2f", "red", 123456, 3.14);`

**Output:** Color red, Number 123456, Float 3.14

credit : L. Surachet, CC BY-SA 3.0, Wikimedia

UCLouvain École polytechnique de Louvain Systèmes Informatiques — E. Rivière

Modificateurs (format specifiers):

<code>%c</code>	char
<code>%d / %i</code>	signed int
<code>%e / %E</code>	scientific notation
<code>%f</code>	float
<code>%ld / %li</code>	long
<code>%lf</code>	double
<code>%Lf</code>	long double

## 2.2 Types de données

- int, long : variable entière
- char : caractère
- double, float : nombre en virgule flottante
- booléens : définis dans stdbool.h, true ou false
- String : pas d'objet String en C, seulement une chaîne de char → longueur de la chaîne inconnue !
- Structures de contrôle : while, for, if, ...

La taille occupée par un type de donnée peut être récupérée via **sizeof()**.

### 2.2.1 Nombres signés et flottants

Les entiers, en C, sont signés par défaut :

22

### Entiers signés

- Représentation en complément à 2
- Exemple : +3 sur 4 bits est 0011
  - Inversion des bits : 0011 => 1100
  - Ajout de 1 : 1100 => 1101
  - -3 est donc 1101
- Il y a toujours une valeur négative sans son inverse (e.g. -128 mais pas +128)

Type	Explication
short	Nombre entier signé représenté sur au moins 16 bits
int	Nombre entier signé représenté sur au moins 16 bits
long	Nombre entier signé représenté sur au moins 32 bits
long long	Nombre entier signé représenté sur au moins 64 bits

Decimal value	Two's-complement Representation
0	0000 0000
1	0000 0001
2	0000 0010
126	0111 1110
127	0111 1111
-128	1000 0000
-127	1000 0001
-126	1000 0010
-2	1111 1110
-1	1111 1111

UCLouvain École polytechnique de Louvain      Systèmes Informatiques — E. Rivière      credit : Wikipedia

Via le mot-clé "unsigned", on peut "dé-signer" un type de donnée :

21

### Entiers non signés

- Types :

Type	Explication
unsigned short	Nombre entier non signé représenté sur au moins 16 bits
unsigned int	Nombre entier non signé représenté sur au moins 16 bits
unsigned long	Nombre entier non signé représenté sur au moins 32 bits
unsigned long long	Nombre entier non signé représenté sur au moins 64 bits

- on obtient le nombre d'octets utilisés par un type donné avec `sizeof (type)` ; e.g. `sizeof (int)` est souvent = à 4 (32 bits)
- Définitions dans `stdint.h`

UCLouvain École polytechnique de Louvain      Systèmes Informatiques — E. Rivière

## Représentation de nombres flottants:

# Nombres flottants

- Réel impossibles à représenter de façon complète ( $1/3$ ,  $\pi$ ,  $e$ , ...)
- Approximation avec une représentation flottante, selon un format standardisé (IEEE-754) :

sign (1 bit)      exponent (8 bits)      fraction (23 bits)

00111110010000000000000000000000 = 0.15625

31 30      23 22      (bit index)      0

- simple précision (float) : séquence de 32 bits
- double précision (double) : séquence de 64 bits.

```
#define FLT_MIN 1.17549435e-38F
#define FLT_MAX 3.40282347e+38F
#define DBL_MIN 2.2250738585072014e-308
#define DBL_MAX 1.7976931348623157e+308
```

défini dans float.h

credit : Wikipedia

### 2.2.2 Caractères

Les caractères en C sont représentés par un nombre, en accord avec la norme ASCII.

Deux types de caractères :

- 0x0..... : permet de représenter les caractères anglais
- 0x1..... : permet de représenter les caractères accentués

26

# Code ASCII (7 bits)

HEX	DEC	CHR	CTRL
01	1	NUL	^
02	2	SOH	^
03	3	STX	^
04	4	ETX	^
05	5	END	^
06	6	ACK	^
07	7	DEL	^
08	8	BS	^
09	9	HT	^
0A	10	LF	^
0B	11	VT	^
0C	12	FF	^
0D	13	CR	^
0E	14	SO	^
0F	15	SI	^
10	16	SL	^
11	17	DC1	^
12	18	DC2	^
13	19	DC3	^
14	20	DC4	^
15	21	NAK	^
16	22	SYN	^
17	23	ETB	^
18	24	CAN	^
19	25	EM	^
1A	26	ESC	^
1B	27	SUB	^
1C	28	FS	^
1D	29	GS	^
1E	30	RS	^
1F	31	US	^

HEX	DEC	CHR
20	32	SP
21	33	!
22	34	"
23	35	#
24	36	\$
25	37	%
26	38	&
27	39	'
28	40	(
29	41	)
2A	42	*
2B	43	+
2C	44	,
2D	45	-
2E	46	.
2F	47	/
30	48	0
31	49	1
32	50	2
33	51	3
34	52	4
35	53	5
36	54	6
37	55	7
38	56	8
39	57	9
3A	58	:
3B	59	;
3C	60	<
3D	61	=
3E	62	>
3F	63	?

HEX	DEC	CHR
40	64	@
41	65	A
42	66	B
43	67	C
44	68	D
45	69	E
46	70	F
47	71	G
48	72	H
49	73	I
4A	74	J
4B	75	K
4C	76	L
4D	77	M
4E	78	N
4F	79	O
50	80	P
51	81	Q
52	82	R
53	83	S
54	84	T
55	85	U
56	86	V
57	87	W
58	88	X
59	89	Y
5A	90	Z
5B	91	[
5C	92	\
5D	93	]
5E	94	^
5F	95	_

HEX	DEC	CHR
60	96	`
61	97	a
62	98	b
63	99	c
64	100	d
65	101	e
66	102	f
67	103	g
68	104	h
69	105	i
6A	106	j
6B	107	k
6C	108	l
6D	109	m
6E	110	n
6F	111	o
70	112	p
71	113	q
72	114	r
73	115	s
74	116	t
75	117	u
76	118	v
77	119	w
78	120	x
79	121	y
7A	122	z
7B	123	{
7C	124	
7D	125	}
7E	126	~
7F	127	DEL

UCLouvain  
 Institut polytechnique de Louvain

Systèmes Informatiques – E. Rivière

### 2.2.3 Chaînes de caractères

En C, une chaîne de caractères est stockée sous la forme d'un tableau.

Le dernier élément du tableau contient le caractère "\0".

## 2.2.4 Pointeurs

Un **pointeur** est une variable qui contient l'adresse à laquelle est stockée un autre élément (variable, fonction, etc.)

La mémoire peut être vue comme une suite de "cases" (octets) avec des adresses consécutives (index de chaque octet).

- On peut créer un pointeur en utilisant le caractère "\*" et récupérer une adresse en utilisant le caractère "&" .
- Le caractère "\*" peut aussi être utilisée sur un pointeur pour récupérer sa valeur (déréférencement)
- En pratique, `char[]` et `char*` sont équivalents.
- Les pointeurs permettent de passer une variable par référence à une fonction.

## 2.2.5 Pointeurs vers une fonction

Une fonction contient une séquence d'instructions, commençant à l'adresse où est stockée sa première instruction.

On peut stocker l'adresse d'une fonction dans un **pointeur de fonction**:

Déclaration : `type (*ptr)([type_arg]+)`

Exemple :

### Exemple d'utilisation d'un pointeur vers une fonction

```
void qsort(void *base, size_t nel, size_t width,
int (*compar)(const void *, const void *));
```

Argument	Rôle
<code>void *base</code>	Début d'une zone mémoire à trier
<code>size_t nel</code>	Nombre d'éléments à trier
<code>size_t width</code>	Taille de chaque élément
<code>int (*compar)(const void *, const void *)</code>	Fonction de comparaison qui prend deux arguments : retourne un <code>int &lt; 0</code> si premier élément précède (e.g. plus petit) le 2ème ; ou un nombre positif ou nul sinon

Les pointeurs sont définis comme `void *` (pointeur sans type) pour rendre la fonction générique — c'est pourquoi la taille des éléments doit être fournie !

Le mot clé `const` indique au compilateur que la fonction n'a pas l'autorisation de modifier la donnée référencée par ce pointeur (c'est une mesure de prudence)

UCLouvain  
Systèmes Informatiques — E. Rivière

## 2.2.6 Manipulation de bits

4 opérateurs binaires :

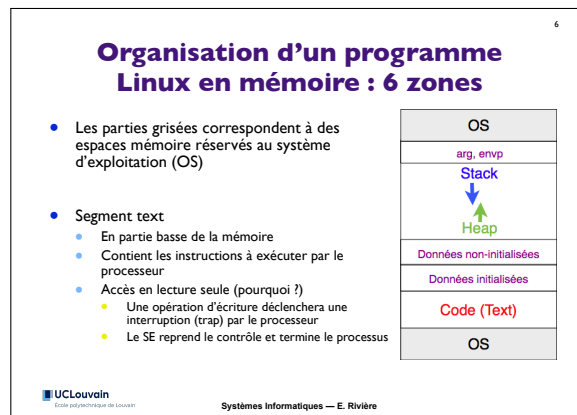
- tilde : inversion  $\rightarrow$  `~00000000 = 11111111`
- & : ET  $\rightarrow$  `11111010 & 01011111 = 01011010`
- | : OU  $\rightarrow$  `11111010 | 01011111 = 11111111`
- ^ : OU exclusif  $\rightarrow$  `11111010 ^ 01011111 = 10100101`

## 3 CM3 : Gestion de la mémoire

### 3.1 Organisation d'un programme Linux en mémoire

Les différentes parties de la mémoire de Linux sont appelés **Segments**.  
Ceux qui nous intéressent:

- Segment 1 : **code**  
Il contient les instructions à exécuter par le processeur, directement extraites des codes.
- Segment 2 : **données initialisées**  
Contient les données initialisées par les codes du segment 1
- Segment 3 : **données non-initialisées**  
Contient les données non-initialisées par le codes du segment 1.  
Elles sont souvent initialisées à 0 par défaut, en attente d'être transférées dans le segment 2.
- Segment 4 : **Heap et Stack**
  - Heap : Permet aux programmes d'y réserver des zones mémoire (c.f. malloc, free, valgrind)
  - Stack : Permet de stocker les variables locales et les appels de fonctions, ainsi que de récupérer les valeurs de retour.
- Segment 5 : **Arguments et variables d'environnement**



## 3.2 Gestion de la mémoire dynamique

malloc() et free() ne sont pas mises en oeuvre dans le noyau du système d'exploitation, ce sont simplement des **fonctions de la librairie standard utilisant des appels système : brk() et sbrk()**.

malloc() est aligné sur un **facteur d'alignement** : elle retourne un nombre entier d'octets directement supérieur au nombre d'octets demandés. (padding)

Exemple : Sous Linux, ce facteur est de **16 octets**. Si l'on demande 37 octets, le malloc va en retourner 64.

### 3.2.1 Objectif d'un algorithme de gestion de mémoire dynamique

Les objectifs de cet algorithme sont les suivants :

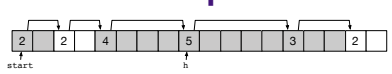
- **Minimiser le temps d'exécution** et maximiser sa stabilité
- **Utiliser efficacement la mémoire** (minimiser la fragmentation):  
Deux types de fragmentation:
  - Fragmentation interne : espace perdu par le padding
  - Fragmentation externe : espace perdu par l'allocation et désallocation de la mémoire
- **Bonnes propriétés de localité spatiale** :  
Alloue les variables proches les unes des autres afin d'accélérer l'accès du cache du processeur

### 3.2.2 Implémentation de l'algorithme

Une des implémentations pourrait utiliser des métadonnées :


48

### Liste implicite



- Chaque bloc commence par un mot de métadonnées (header)
  - Taille du bloc (en octets, #ints pour la figure)
  - Type de bloc {libre, alloué} dans le bit de poids faible
- Parcours de l'ensemble des blocs pour trouver un bloc libre

```
h = start;
while (h < end &&
      ((*(h & 0x1) != 0 || // fin de liste ?
        *(h & 0x1) == 0 || // déjà alloué
        *(h & 0x1) == 0)) // trou trop petit
      h = h + (*h & ~0x1); // progresse vers le prochain bloc
```

  
Université catholique de Louvain

Systèmes Informatiques — E. Rivière




Cependant, aucune des implémentations ne peut satisfaire les 3 conditions parfaitement. Par exemple :

55

### Politiques de placement

- Est-il sage de choisir le premier bloc libre de taille suffisante rencontré lors du parcours ? (Politique "first fit")
  - 🚀 Rapide
  - 😬 Entraîne de la fragmentation
  - 😬 Localité faible, empirant au fur et à mesure de l'exécution du programme
- Politique "next fit" : comme "first fit" mais démarre la parcours depuis le dernier bloc alloué
  - 😊 Meilleures propriétés de localité
  - 🚧 Fragmentation très élevée
- Politique "best fit" : parcours intégral de la liste pour trouver le bloc libre le plus petit possible pouvant accueillir le nouveau bloc
  - 😊 Optimal en termes de fragmentation
  - 😬 Localité médiocre
  - 🚧 Coût d'exécution élevé

 UCLouvain  
Université catholique de Louvain

Systèmes Informatiques — E. Rivière

## 4 CM4 : Structure des ordinateurs

### 4.1 Organisation d'un système informatique

- La mémoire est adressée par **bytes**.
- Les adresses sont généralement écrites sur 32 ou 64 bits. (64 bits permettent environ  $1,8e19$  valeurs différentes)
- Les instructions sont aussi codées en binaire, elles aussi de taille variable

#### 4.1.1 Mémoire du processeur

Il en existe deux types :

- SRAM : généralement très chère et rapide, présente dans les processeurs modernes (cache)
- DRAM : beaucoup moins chère, mais bien plus grande capacité et modulaire

Redondant, il est chiant ce cours...