Architecture des ordinateurs (M1 bio-info)

Alexandre Guitton (alexandre.guitton@uca.fr)

UNIVERSITÉ Clermont Auvergne

Préambule

- Enseignement « architecture des ordinateurs »
  - Elément constitutif de l'UE1 « Introduction à la bioinformatique et aux analyses intégratives »
  - Durée : 6h (incluant un contrôle continu d'1h en dernière séance)
- Contrôle des connaissances
  - Un contrôle continu écrit d'une heure en dernière séance
  - Représente 1/3 de la note de l'UE (avec 1/3 en « bash » et 1/3 en « projet de bioanalyse »)

Objectifs de l'UE

- Comment fonctionne un ordinateur ?
  - Comprendre le fonctionnement d'un ordinateur permet de comprendre comment le programmer, quelles sont ses capacités, et quelles sont ses limites
  - => cf chapitre 1 (« architecture d'un ordinateur »)
- Qu'est-ce qu'un programme parallèle ? Quelles sont les notions théoriques qui permettent de faire un programme parallèle ?
  - => cf chapitre 2 (« programmation parallèle »)

Chapitre 1 – Architecture d’un ordinateur

- Circuits logiques
- Circuits séquentiels
- Carte mère
- Programmation d’un processeur
- Processeurs graphiques (GPU)
- Limites d’un ordinateur

1.1 – Circuits logiques

- L’informatique a une base binaire (0 = pas de courant, 1 = courant)
- Le binaire va servir à stocker (=> cf 1.1.1 stockage)
- Le binaire va servir à faire des calculs (=> cf 1.1.2 traitement)

1.1.1 – Stockage

- Les données sont (toutes) représentées avec du binaire
  - Les entiers positifs
  - Les entiers négatifs
  - Les nombres réels
  - Les caractères
  - Les chaînes de caractères
  - Les images (=> hors programme)
  - Le son (=> hors programme)
  - Les fichiers DOCX, ODT, PDF, JPG, MP3, etc. (=> hors programme)

Stockage des entiers positifs

- Tout entier positif n peut s'écrire sous la forme suivante :
  - $a_0.2^0 + a_1.2^1 + a_2.2^2 + \ldots + a_k.2^k$, avec a_i dans {0,1}
  - La représentation binaire de n est alors $a_k \ldots a_2 a_1 a_0$, sur k bits
- Par exemple, prenons n=19
  - 19 s'écrit $16 + 2 + 1 = 2^4 + 2^1 + 2^0$, soit $1.2^4 + 0.2^3 + 0.2^2 + 1.2^1 + 1.2^0$
  - Donc 19 (en base 10) s'écrit 10011 (en base 2)
- Avec k bits, on ne peut stocker que des entiers de 0 à 2^k-1
  - Il faut donc bien choisir k en fonction des entiers manipulés

exemple

Stockage des entiers négatifs

- Quatre méthodes sont habituelles :
  - Utiliser un bit de signe (0=positif ou 1=négatif)
  - Faire le complément logique (CL) : on inverse tous les bits pour les nombres négatifs
    - Simplification des calculs par rapport à l'utilisation d'un bit de signe
  - Faire le complément arithmétique (CA) : on inverse tous les bits et on ajoute 1 pour les nombres négatifs
    - Simplification des calculs par rapport à l'utilisation du complément logique (car on ne réintègre pas la dernière retenue)
  - Utiliser un biais (donc un décalage constant)

exemple CA + CL

Stockage des nombres réels

- Problème : il faut pouvoir représenter une virgule
  - avoir une virgule à une position fixe ne permet pas d'avoir une précision qui s'adapte au calcul
- Solution : Format IEEE 854
  - Tout nombre est représenté sous la forme +/- $m.2^e$, avec la mantisse m telle que 0,5<=m<1 et l'exposant e qui est un entier (proche de la notation scientifique)
  - m a un bit caché, et e est biaisé (car il peut être négatif)
  - Simple précision (1 signe, 23 mantisse, 8 exposant avec biais de -126) ou double précision (1 signe, 52 mantisse, 11 exposant avec biais de -1022)

exemples

Stockage des caractères

- Les caractères sont encodés comme des nombres
- L'encodage le plus simple est le code ASCII *(American Standard Code for Information Interchange)*
- Problématique de l'encodage (nombre de caractères pouvant être représentés, taille de l'encodage, etc.)

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

https://www.fil.univ-lille1.fr/~wegrzyno/portail/Info/Doc/HTML/seq7_codage_caracteres.html

Stockage des chaînes de caractères

- Méthode « Pascal » : une chaîne de caractères commence par sa longueur (sur une taille connue à l'avance), puis par la suite des caractères
- Méthode « C » : une chaîne de caractères est terminée par un caractère spécial (généralement le caractère de valeur 0)
- Problématique de l'encodage

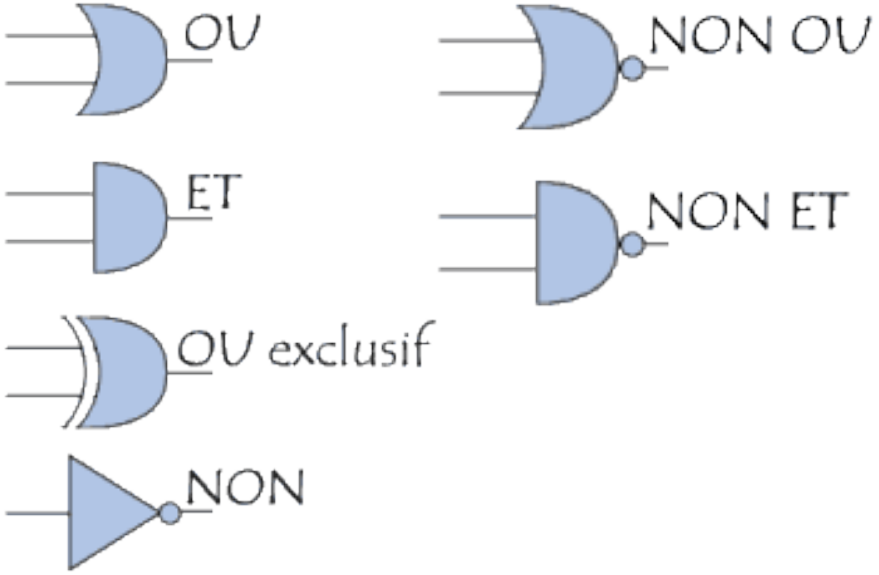
1.1.2 - Traitement

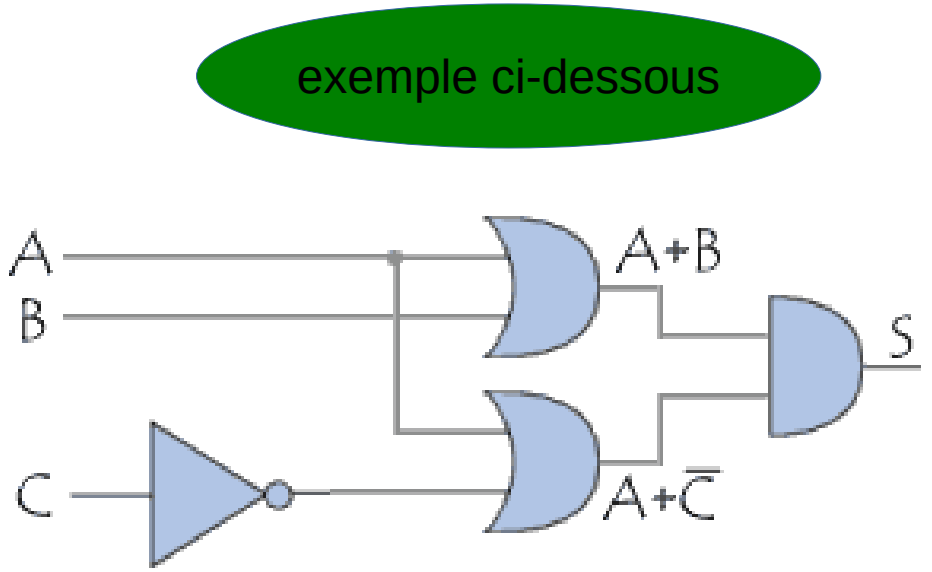
- Un circuit logique est un circuit électronique simple permettant de réaliser des opérations binaires
  - Il est simple car il n'intègre pas de notion de temps ou de mémoire (contrairement aux circuits séquentiels)
  - Le circuit électronique est simplifié, et ignore l'essentiel des problématiques physiques
- Un circuit logique calcule des sorties (binaires) en fonction d'entrées (binaires)
  - Exemple : une lumière est allumée (1) ou éteinte (0) en fonction de la position de plusieurs interrupteurs qui se trouvent dans la pièce (0 / 1 pour chacun)

Opérations de base

- Les manipulations des signaux se font avec des fonctions logiques (d'un point de vue mathématique) ou des portes logiques (d'un point de vue physique)
- Il faut connaître les fonctions suivantes : NOT, OR, AND, XOR, NOR, NAND, etc.
  - Quand on énumère les sorties pour toutes les entrées possibles, on parle de table de vérité

Portes logiques usuelles et exemple de circuit logique





https://www.commentcamarche.net/contents/649-circuits-logiques

Fonctions logiques minimales

- Théorème : On peut écrire toute fonction logique avec une combinaison de NOT, AND et OR
- Théorème : On peut écrire toute fonction logique avec une combinaison de NOT et AND (ou bien de NOT et OR)
- Théorème : On peut écrire toute fonction logique avec une combinaison de NAND (ou bien de NOR)

Gros circuits logiques

- On peut réaliser de « gros » circuits logiques qui font des opérations complexes
- Exemples :
  - Une addition
  - Une multiplication
  - Une traduction d'un code vers un autre code
  - Etc.

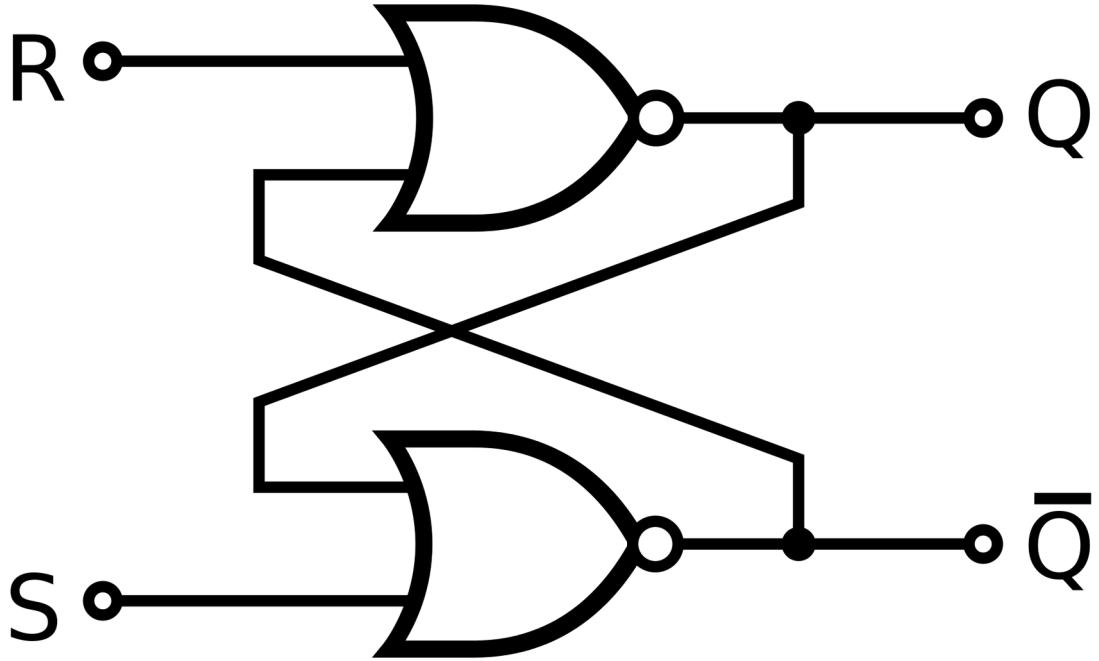
1.2 – Circuits séquentiels

- Un circuit séquentiel est un circuit électronique complexe permettant de réaliser des opérations binaires
  - Il est complexe car il intègre une notion de temps (et/ou de mémoire, cf après)
  - Le temps est cadencé par une horloge qui fonctionne à une certaine fréquence (en Hz)
  - On réalise une opération à chaque fois que l'horloge fournit un tic

Bascule RS (1/2)

- La bascule RS est un circuit logique qui :
  - Permet de sauvegarder un bit d'information (appelé Q)
  - Possède deux opérations : mettre à 0 le bit stocké (opération R pour *Reset*), et mettre à 1 le bit stocké (opération S pour *Set*), et une valeur accessible (Q)
- La bascule RS fonctionne sur le modèle suivant :
  - Il y a trois entrées : Q_{ancien}, R et S
  - Il y a une sortie : $Q_{nouveau}$
  - On peut montrer que $Q_{nouveau}=R+(S+Q_{ancien})$
  - Donc $Q_{nouveau}=R \text{ NOR } (S \text{ NOR } Q_{ancien})$

Bascule RS (2/2)



exemple

https://fr.wikibooks.org/wiki/Fonctionnement_d%27un_ordinateur/Les_circuits_s%C3%A9quentiels

Vers le circuit (séquentiel) d'un processeur

- Grâce aux circuits séquentiels, on peut à présent construire un circuit qui :
  - Utilise de la mémoire (via la bascule RS)
  - Choisit un calcul à effectuer en fonction de certains bits d'entrée (les autres bits d'entrée servant à fournir les opérandes du calcul)
  - Exécute un calcul (potentiellement différent) à chaque tic d'horloge
- On peut donc construire un processeur !
  - Il faudra lister un (petit) ensemble de calculs que le processeur peut réaliser
  - Pour chaque calcul possible, il faudra donner la valeur des bits d'entrée « d'opération » (hors opérandes) : c'est appelé l'opcode (pour *operation code*)

exemple

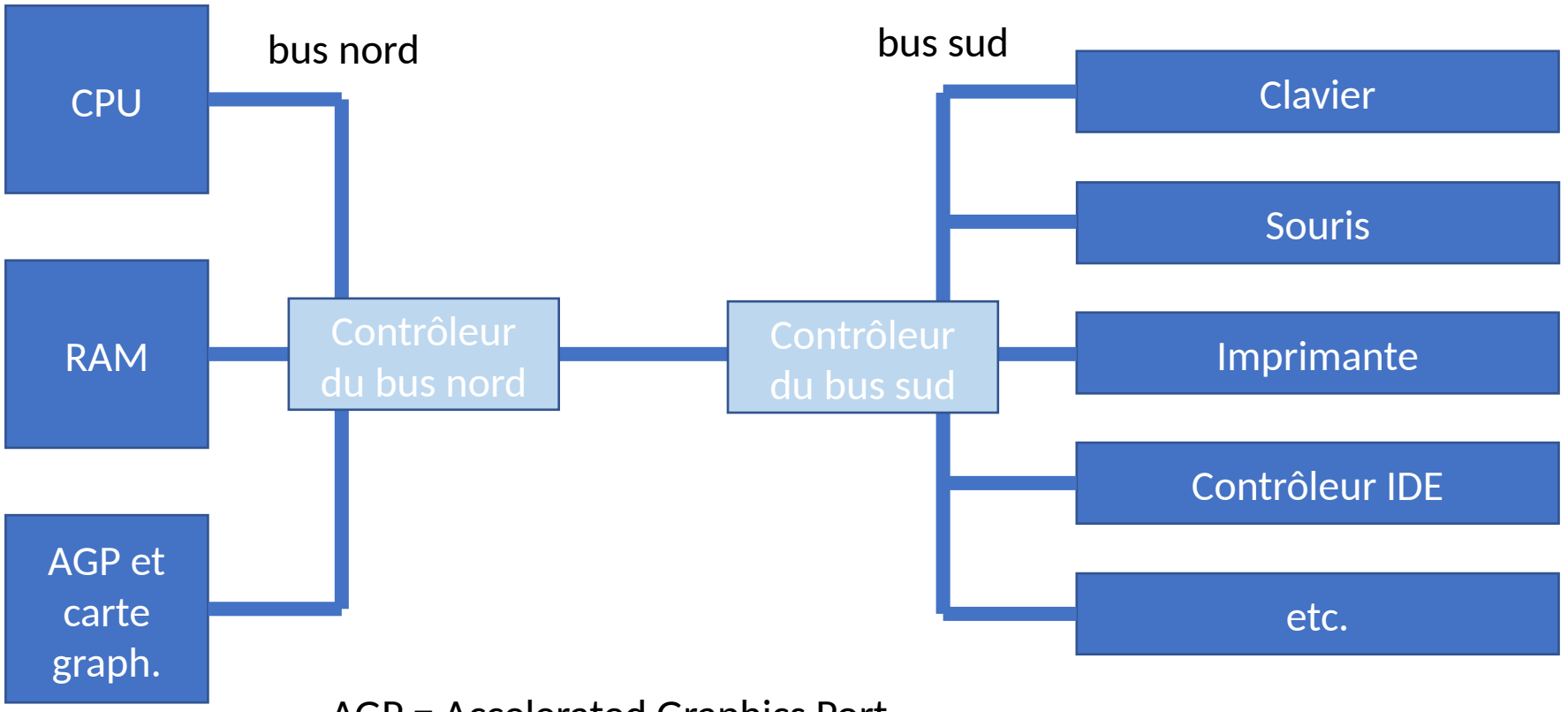
Vers le circuit d’un ordinateur

- Architecture de von Neumann
  - Modèle d’ordinateur dans lequel la mémoire est (unique et) séparée du processeur
  - Les composants (lecteur de disque, écran, imprimante, etc.) sont des circuits séquentiels
  - Les composants communiquent entre eux via des bus

1.3 – Carte mère

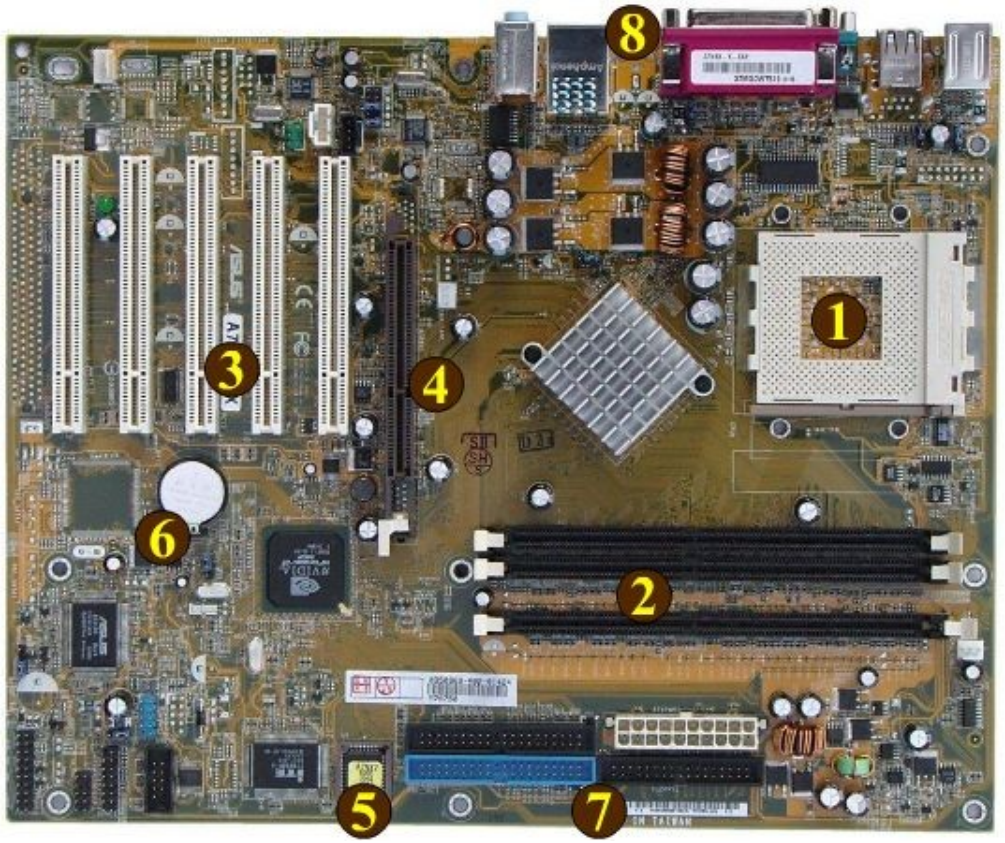
- Ordinateur
  - carte mère (processeur, mémoire vive, etc.)
  - périphériques externes (écran, clavier, souris, lecteurs de disques durs, lecteur DVD, imprimantes, clés USB, appareils photos, etc.)
- Carte mère
  - Processeur (CPU = *Central Processing Unit*)
  - Mémoire (RAM = *Random Access Memory*, avec une partie de ROM = *Read Only Memory*)
  - Pile CMOS pour sauvegarder quelques paramètres, dont l'heure (CMOS = une technologie de pile)
  - Ports (pour connecter les périphériques)
  - Bus (pour interconnecter les composants) et contrôleur (pour gérer les accès concurrents sur le bus)

Schéma de la carte mère



AGP = Accelerated Graphics Port
IDE = Integrated Drive Electronics

Photos de cartes mères (1/5)



http://www.lopr.net/ccm.php?n=1

1 : socket (pour le processeur)

2 : RAM

3 : ports PCI *(Peripheral Component Interconnect)*

4 : port AGP

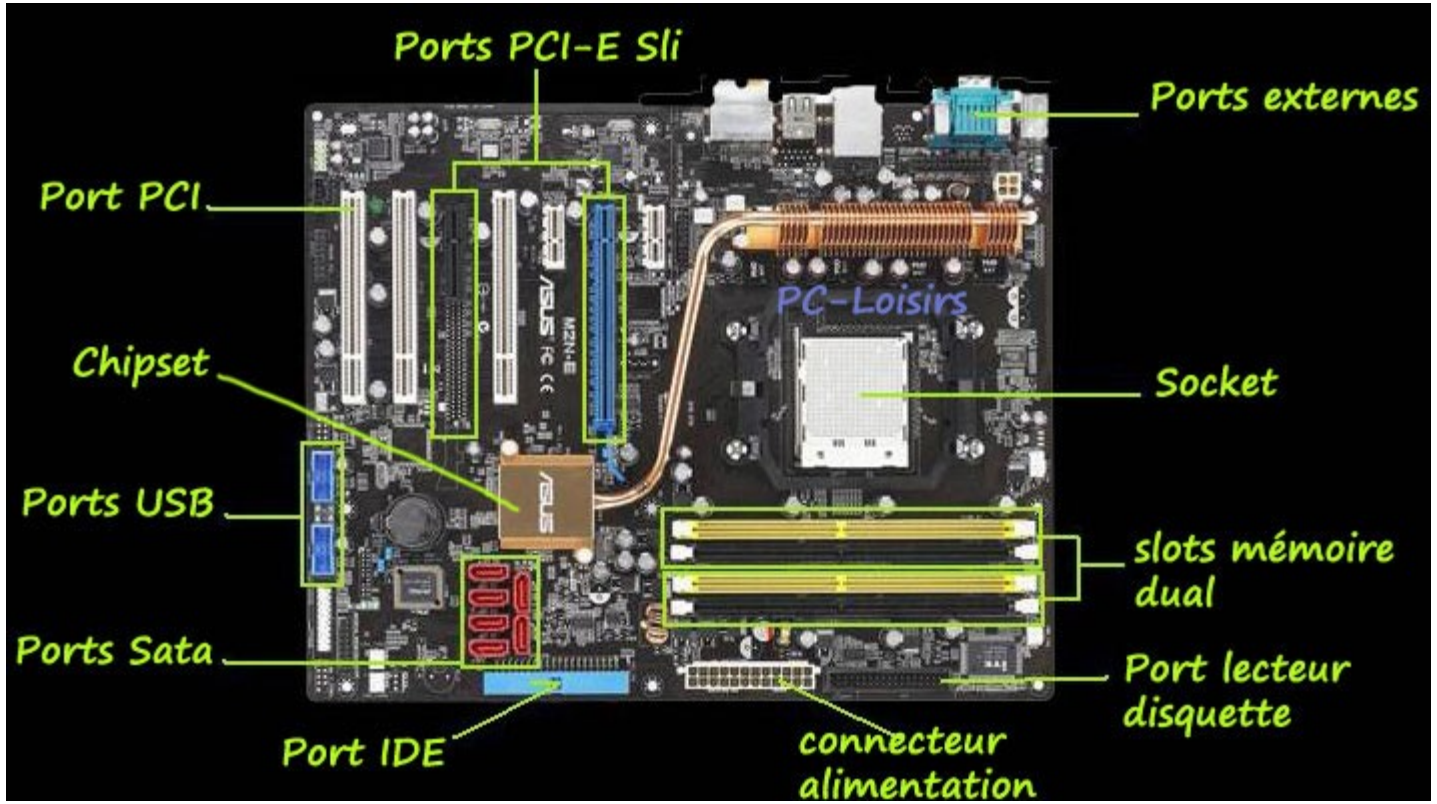
5 : puce BIOS *(Basic Input Output System)*

6 : pile CMOS

7 (de haut-gauche à bas-droite) : IDE + alimentation + IDE + lecteur disquettes

8 : ports externes

Photos de cartes mères (2/5)



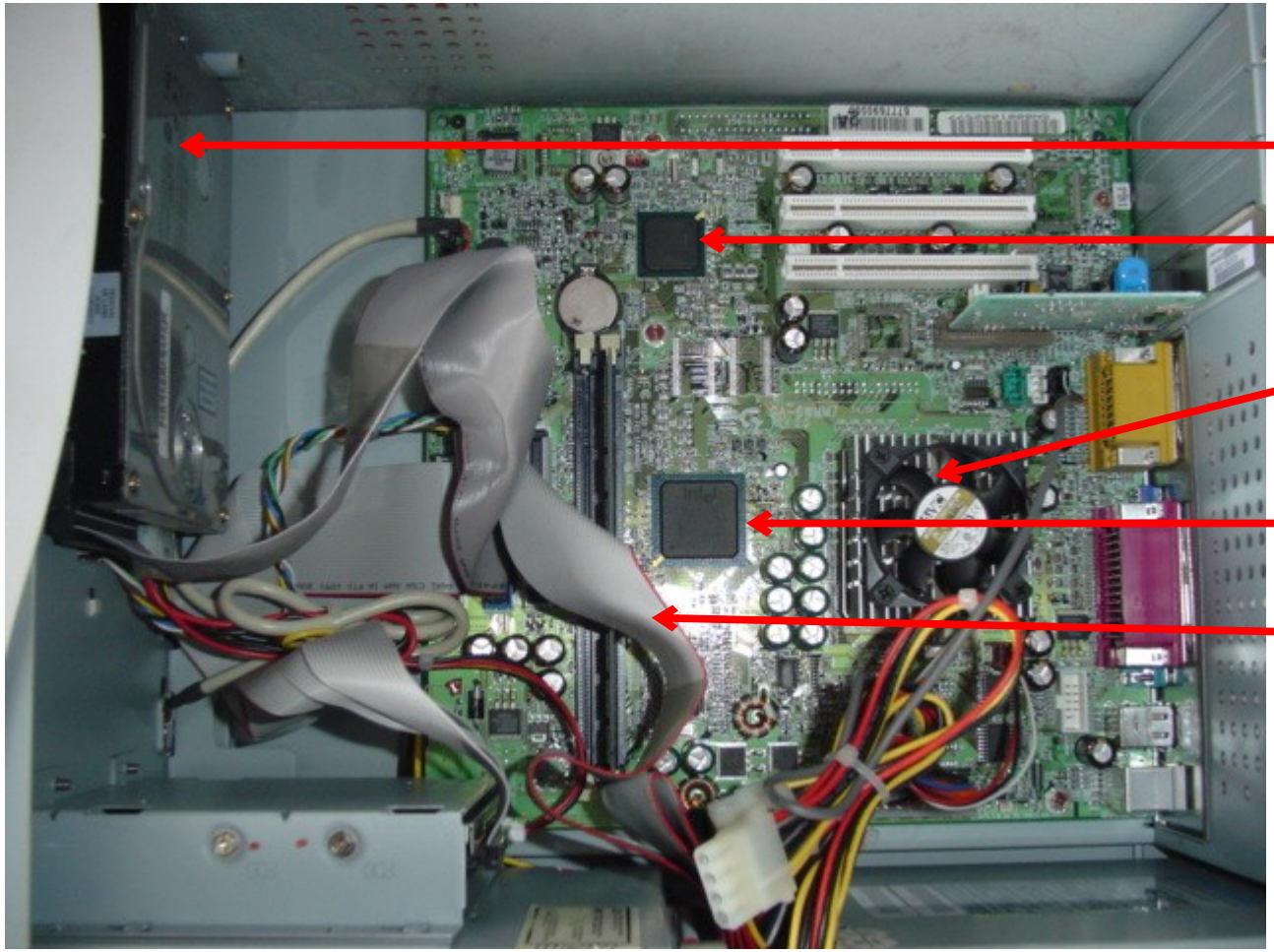
http://www.pc-driver.net/uploads/image/carte-mere-driver-bios.jpg

PCI-E SLI (*PCI Express – Scalable Link Interface*)

USB *(Universal Serial Bus)*

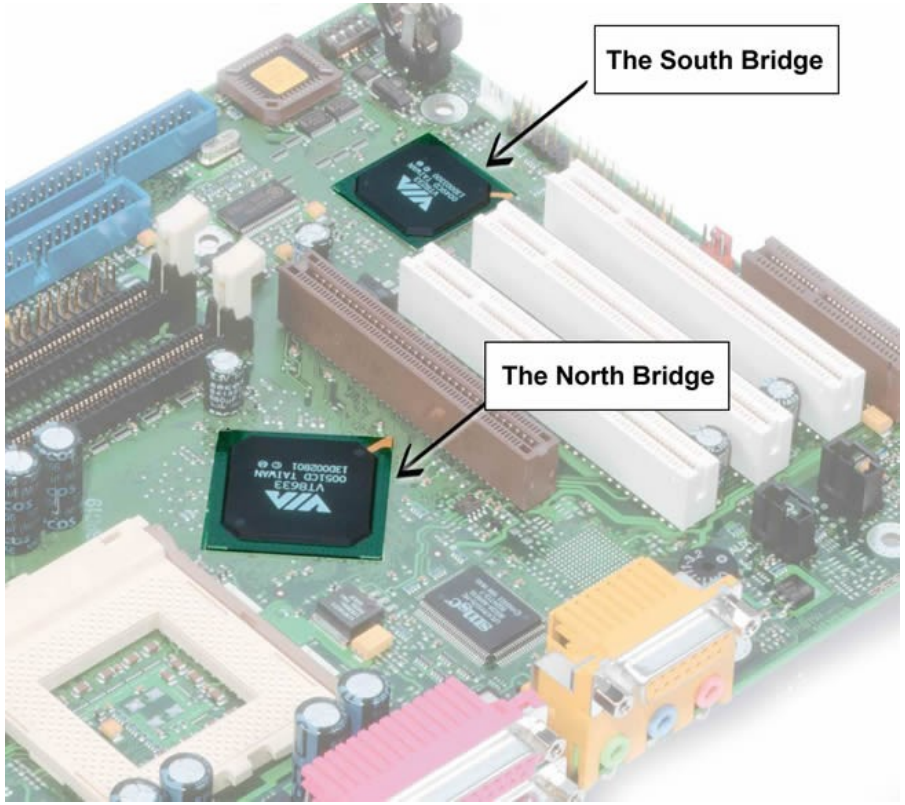
SATA *(Serial Advanced Technology Attachment)*, pour les lecteurs

Photos de cartes mères (3/5)



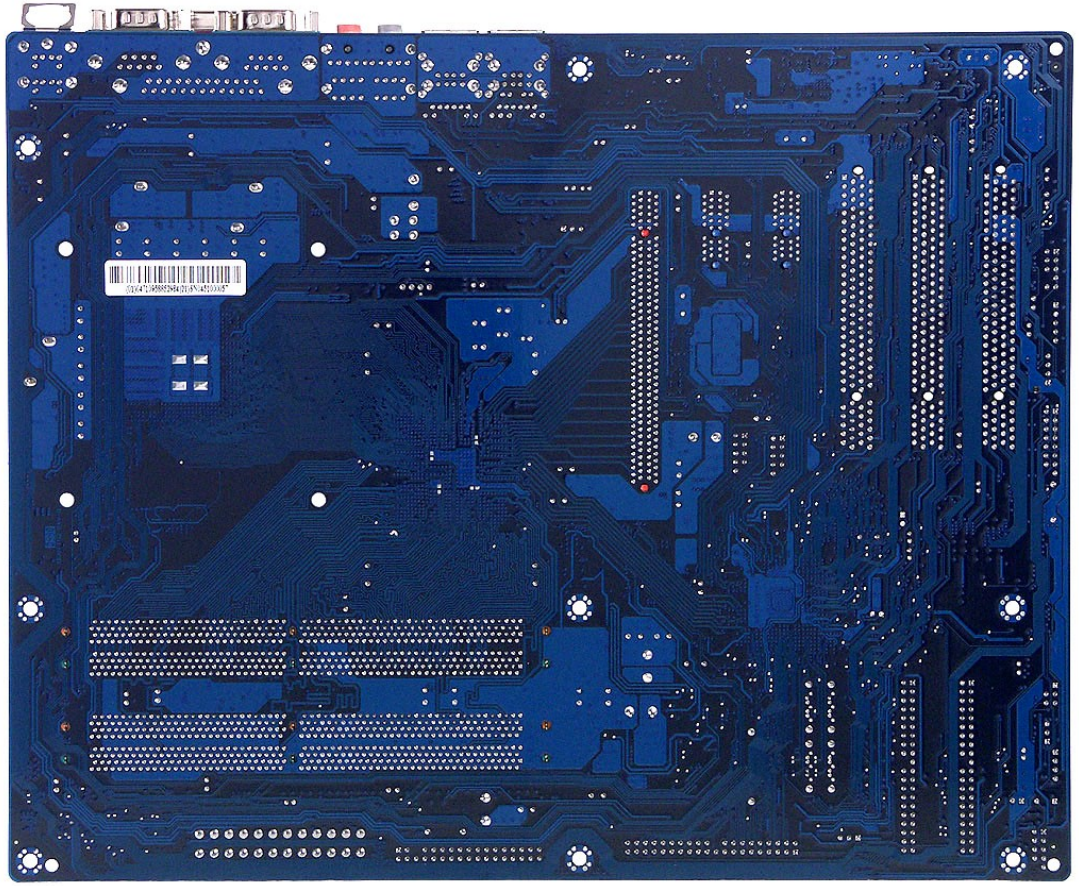
http://www.rcassin.no/index.php?id=4532889

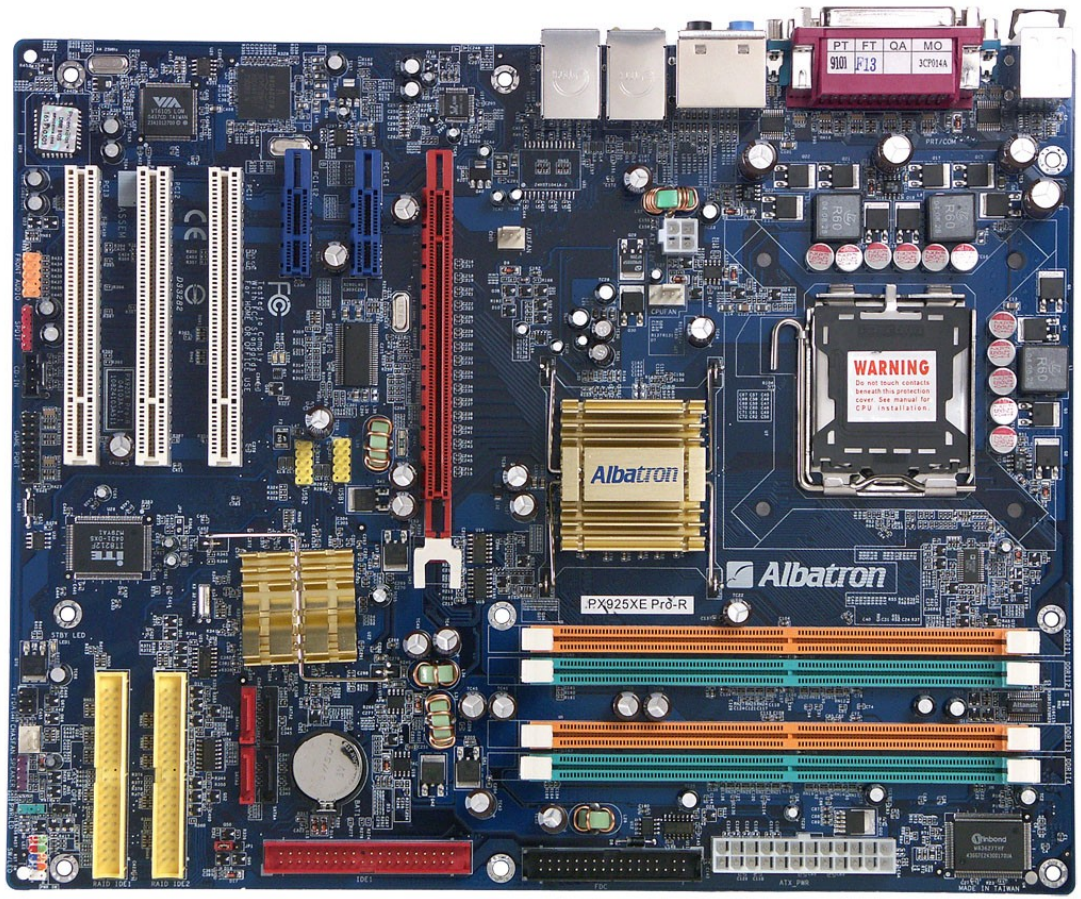
Photos de cartes mères (4/5)



http://www.karbosguide.com/books/pcarchitecture/chapter26.htm

Photos de cartes mères (5/5)





http://ixbtlabs.com/articles2/mainboard/albatron-px925xe-pro-r-i925xe.html

(début du cours #2)

Mémoires

- La mémoire principale est la RAM
  - Elle est très volumineuse (plusieurs Go en général) mais à accès lent
- Le processeur utilise une mémoire ultra-rapide, appelée « registres »
  - Un processeur n'a que quelques registres (une vingtaine)
  - Ces registres servent à faire des calculs temporaires
- La carte mère intègre une petite mémoire rapide appelée « cache »
  - Le cache sert à accéder rapidement à des données fréquemment utilisées
  - Il peut y avoir plusieurs niveaux de mémoire cache (L1, L2, L3)
  - Les niveaux inférieurs sont petits et rapides, les niveaux supérieurs sont plus grands et plus lents
  - La position physique des mémoires caches varie selon leur niveau
  - Sur les architectures multi-processeurs ou multi-cœurs, le cache L3 peut être partagé

Exemple de caractéristique technique d’un portable

Caractéristiques techniques : Portable HP 15-N082SF 15,6" Tactile
Type de produit : Portable – Notebook
Ecran : 15,6 "
Poids en kg : 2,2 Kg
Dimensions (l x p x h) en mm : 385 x 258 x 22
Ecran tactile : Oui
Processeur : Intel Core i7-4500U
Vitesse du processeur : 1,8 Ghz
Système d'exploitation : Windows 8 64 Bits
Mémoire cache externe : 4 Mo
Mémoire RAM : 8 Go
Type de mémoire RAM : DDR3
Type de disque dur : SATA
Capacité du disque dur : 1 To
Vitesse de rotation du disque dur : 5400 tours/min

Lecteur / Graveur : Graveur DVD±RW SuperMulti avec support double couche
Lecteur de cartes mémoire SD
Carte graphique : NVIDIA GeForce GT 740M
Mémoire vidéo dédiée : 2048 Mo
Webcam intégrée : Oui
Audio : DTS Sound+ avec 2 haut-parleurs
Carte réseau Ethernet : 10/100 Mbps
Communication sans fil : 802.11b/g/n
Connecteurs : 1 port HDMI ; 1 sortie casque ; 1 entrée microphone ; 1 port USB 2.0 ; 2 ports USB 3.0 ; 1 RJ45
Dispositif de pointage : TouchPad
Pavé numérique : Oui

http://www.fnac.com/Portable-HP-15-N082SF-15-6-Tactile/a6424902/w-4#ficheDt

1.4 – Programmation d'un processeur

- Un processeur exécute des instructions très simples, qui sont codées au niveau matériel
  - Chaque instruction correspond à un code (appelé « opcode »)
  - Les instructions peuvent manipuler des constantes, des registres, ou de la mémoire
- Exemple d'instructions
  - Mettre la valeur 17 dans le registre EAX : MOV EAX, 17
    - B8 11 00 00 00 (Intel x86)
  - Soustraire EAX du registre ECX : SUB ECX, EAX
    - 2B C8 (Intel x86)
  - Mettre la valeur de la case mémoire EAX dans ECX : MOV ECX, [EAX]
    - 8B 08 (Intel x86)
  - Etc.

Langage d’assemblage

- Un processeur ne comprend que le langage binaire et le langage assembleur
  - Langage binaire : B8 11 00 00 00 8B 08 2B C8 …
  - Langage assembleur : MOV EAX, 17 ; MOV ECX, [EAX] ; SUB ECX, EAX ; …
- … mais il est très fastidieux de programmer ainsi
- On utilise donc d’autres langages
  - Des langages compilés (C, C++, Java, etc.) qui nécessitent de traduire les instructions du langage en assembleur, via un compilateur
  - Des langages interprétés (Python, etc.) qui utilisent une machine virtuelle pour traduire les instructions du langage en assembleur

Optimisations – exemple 1

- Dans les langages compilés, le compilateur a aussi le rôle d'optimiser le programme généré
- Exemple 1 :
  - Algorithme : a = x+3; b = x+2;
  - Traduction lente :
    MOV EAX, [x]
    ADD EAX, 3
    MOV [a], EAX
    MOV EAX, [x]
    ADD EAX, 2
    MOV [b], EAX
  - Traduction plus rapide :
    MOV EAX, [x]
    MOV EBX, EAX
    ADD EBX, 3
    MOV [a], EBX
    ADD EAX, 2
    MOV [b], EAX

Optimisations – exemple 2

- Exemple 2 :
  - Algorithme : x = 5*x
  - Traduction lente :
    MOV EAX, [x]
    MUL EAX, 5
    MOV [x], EAX

IMUL = *Integer Multiplication* = multiplication
SHL = *Shift Left* = décalage à gauche des bits

- Traduction plus rapide :
  MOV EAX, [x]
  MOV EBX, EAX
  SHL EBX, 2
  ADD EAX, EBX
  MOV [x], EAX

Optimisations – exemple 3

- Exemple 3 *(loop unrolling)* :
  - Algorithme : j = 0 ; pour i de 1 à n faire ; j = j+i ; fin pour
  - Traduction lente (simplifiée) : répéter n fois la boucle
  - Traduction plus rapide :
    j = 0 ; pour i de 1 à n par pas de 2 faire ; j = j+2*i+1 ; fin pour
  - Traduction encore plus rapide : j = n*(n-1)/2

- Il existe de nombreuses techniques d'optimisations de code qui modifient peu l'algorithme…
  - Ces techniques peuvent simplifier les calculs, souvent en les factorisant
  - Ou bien elles profitent des accès mémoires rapides grâce aux caches
- … et quelques techniques qui le modifient beaucoup

Complément : *pipeline* d'instructions

- Les processeurs modernes utilisent un *pipeline* d'instructions pour accélérer l'exécution
  - L'exécution de chaque instruction est découpée en stages, par exemple : *instruction fetch*, *instruction decode*, *execute*, *memory access*, *register write back*
  - Certains processeurs ont des *pipelines* beaucoup plus longs (il y a 20 stages pour le Pentium 4 d'Intel, ou 31 stages pour les Xeon)
  - A chaque tic d'horloge, le stage passe au suivant pour chaque instruction dans le *pipeline*
- Objectif : accélérer les traitements en découpant les instructions complexes en stages simples (donc courts)
  - Mais permet aussi de réduire les coûts (moins de portes logiques + meilleur usage des circuits car tous les stages sont utilisés)

Complément : exemple d'un *pipeline* d'instructions

Temps	t_1	t_2	t_3	t_4	t_5	t_6	t_7
Stage 1	Instruct. 1	Instruct. 2	Instruct. 3	Instruct. 4	Instruct. 5	Instruct. 6	Instruct. 7
Stage 2		Instruct. 1	Instruct. 2	Instruct. 3	Instruct. 4	Instruct. 5	Instruct. 6
Stage 3			Instruct. 1	Instruct. 2	Instruct. 3	Instruct. 4	Instruct. 5
Stage 4				Instruct. 1	Instruct. 2	Instruct. 3	Instruct. 4
Stage 5					Instruct. 1	Instruct. 2	Instruct. 3

Complément : quelques problèmes avec les *pipelines* d'instructions

- Les *pipelines* d'instructions peuvent causer des problèmes
- Problème de dépendance :
  - Si le calcul de l'instruction N+1 dépend du résultat de l'instruction N, il faut introduire un temps d'attente pour permettre au résultat de l'instruction N d'être sauvegardé
  - Dans ce cas, le processeur introduit alors une bulle dans le pipeline pour retarder l'exécution de l'instruction N+1. La bulle suit ensuite son parcours dans le *pipeline* jusqu'à la sortie
- Problème de prédiction des branchements :
  - En présence de branchement conditionnel, le processeur prédit les instructions qui s'exécuteront
  - En cas d'échec de prédiction, le pipeline doit être vidé

Optimisations plus complexes

- Il existe des optimisations plus complexes utilisant le *pipeline* d'instructions
  - Les processeurs peuvent faire de la prédiction de branchement, pour essayer d'anticiper les prochaines instructions à exécuter (=> hors programme)
  - Les processeurs peuvent aussi réordonnancer les instructions pour les exécuter dans un autre ordre que l'ordre donné par le programme (compilateur) : on parle alors d'exécution *out-of-order* (=> hors programme)

1.5 – Processeurs graphiques (GPU)

- Un GPU *(Graphics Processing Unit)* est un processeur dédié aux calculs graphiques
  - Il réalise de nombreux calculs, et notamment : trigonométrie (pour la 3D : calcul des faces cachées, lumière, texture, etc.) et traitement d'images (ex : anticrénelage, droites, rotations, zooms, etc.)
  - Les GPU réalisent de nombreux calculs en parallèle sur des données différentes

1.6 – Limites d’un ordinateur

- Dans ce chapitre, nous avons vu comment était construit un processeur, et comment pouvait être exécuté un algorithme
- Limites d’un ordinateur
  - Les nombres ont une capacité et une précision limitées
  - Plus les calculs sont complexes, plus ils sont lents
  - Les accès mémoire sont lents
  - La mémoire a une taille limitée

Limites du calcul

- Limites du calcul
  - Les instructions s'exécutent séquentiellement (= une par une, dans l'ordre)
- Pour accélérer les calculs, il faut donc :
  - Soit réduire la durée de chaque calcul, ce qui nécessite une optimisation du schéma du circuit du processeur ou une accélération de l'horloge, mais pose des problèmes de distance, de température, de proximité trop forte de câbles électriques, etc.
  - Soit changer l'algorithme
  - Soit exécuter plusieurs instructions en même temps (= parallélisme)

Chapitre 2 – Programmation parallèle

- Généralités
- Processus et *threads*
- Mécanismes de communications entre processus et threads
- Mécanismes de synchronisation
- Introduction sur OpenMP et MPI

2.1 - Généralités

- Le système d'exploitation est le programme qui permet de gérer l'ordinateur et ses ressources (dont la mémoire, les programmes installés, les périphériques, les programmes en cours d'exécution, etc.)
  - Exemple : Windows, GNU / Linux, Unix, MAC OS, etc.
  - Un système d'exploitation est responsable : de la gestion des processus, de la gestion de la mémoire (vive), de la gestion des fichiers, de la gestion des périphériques, et de l'interface homme-machine (dans une moindre mesure)

Parallélisation d’un traitement long

- Pour accélérer un traitement long, une solution est de le paralléliser, c’est-à-dire de le faire exécuter par plusieurs processeurs
  - Plus il y a de processeurs pour exécuter un traitement, plus le traitement est rapide… jusqu’à une certaine limite => cf loi d’Amdhal
  - Mais, il n’est pas si facile de paralléliser un traitement entre plusieurs processeurs : ces processeurs doivent se coordonner, doivent communiquer entre eux des résultats partiels, doivent éviter des inter-blocages, etc.

Exemple de programmes parallèles

Calcul de la moyenne (séquentiel)

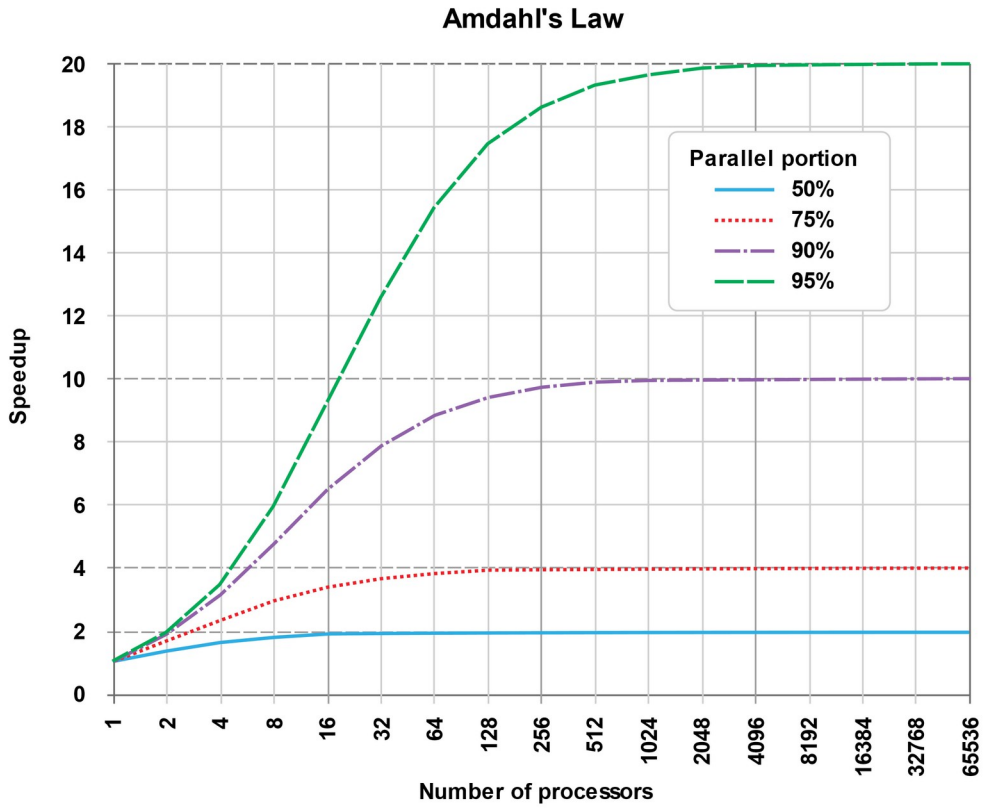
- P1
  - somme 0
  - pour i de 1 à 1000 faire
    - somme somme + t[i]
  - retourner somme/1000

Calcul de la moyenne (parallèle)

- P1
  - somme1 0
  - pour i de 1 à 500 faire
    - somme1 somme1 + t[i]
  - envoyer somme1 à P2
- P2
  - somme2 0
  - pour i de 501 à 1000 faire
    - somme2 somme2 + t[i]
  - récupérer somme1 de P1
  - retourner (somme1+somme2)/1000

Loi d'Amdhal

- La loi d'Amdhal est une formule qui estime le gain de paralléliser un algorithme
  - En fonction du nombre de processeurs s
  - En fonction de la partie de l'algorithme que l'on peut paralléliser p (car on ne peut pas tout paralléliser)
  - La formule est 1/(1-p+p/s)



https://upload.wikimedia.org/wikipedia/commons/e/ea/AmdahlsLaw.svg

exemple

Démonstration

- Soit T le temps de calcul (avec 1 processeur)
  - Avec 1 processeur, on peut réécrire T = (1-p).T+p.T, où p.T est le temps parallélisable, et (1-p).T est le temps non parallélisable
  - Avec s processeurs, (1-p).T ne change pas, mais le temps parallélisé devient p.T/s, que l'on note T(s)
  - L'accélération, calculée comme un ratio, est T/T(S)=T/((1-p).T+p.T/s)=1/(1-p+p/s)

Difficultés autour de la programmation parallèle

- Comment paralléliser un programme séquentiel ?
  - Il faut parfois changer l'algorithme
  - Exemple sur la moyenne :
    - Faut-il séparer les calculs de 1 à 500 et de 501 à 1000, ou les nombres pairs pour l'un et les nombres impairs pour l'autre ?
    - Faut-il donner plus de calculs à P1 car P2 travaille plus ensuite pour agréger ?
- Comment les processus doivent-ils communiquer entre eux ?

2.2 – Processus et *threads*

- Le parallélisme est implémenté de plusieurs manières sur un ordinateur :
  - Au travers de processus (« lourds », car consommant beaucoup de ressources pour le processeur)
  - Au travers de *threads* (« légers », car consommant peu de ressources pour le processeur)

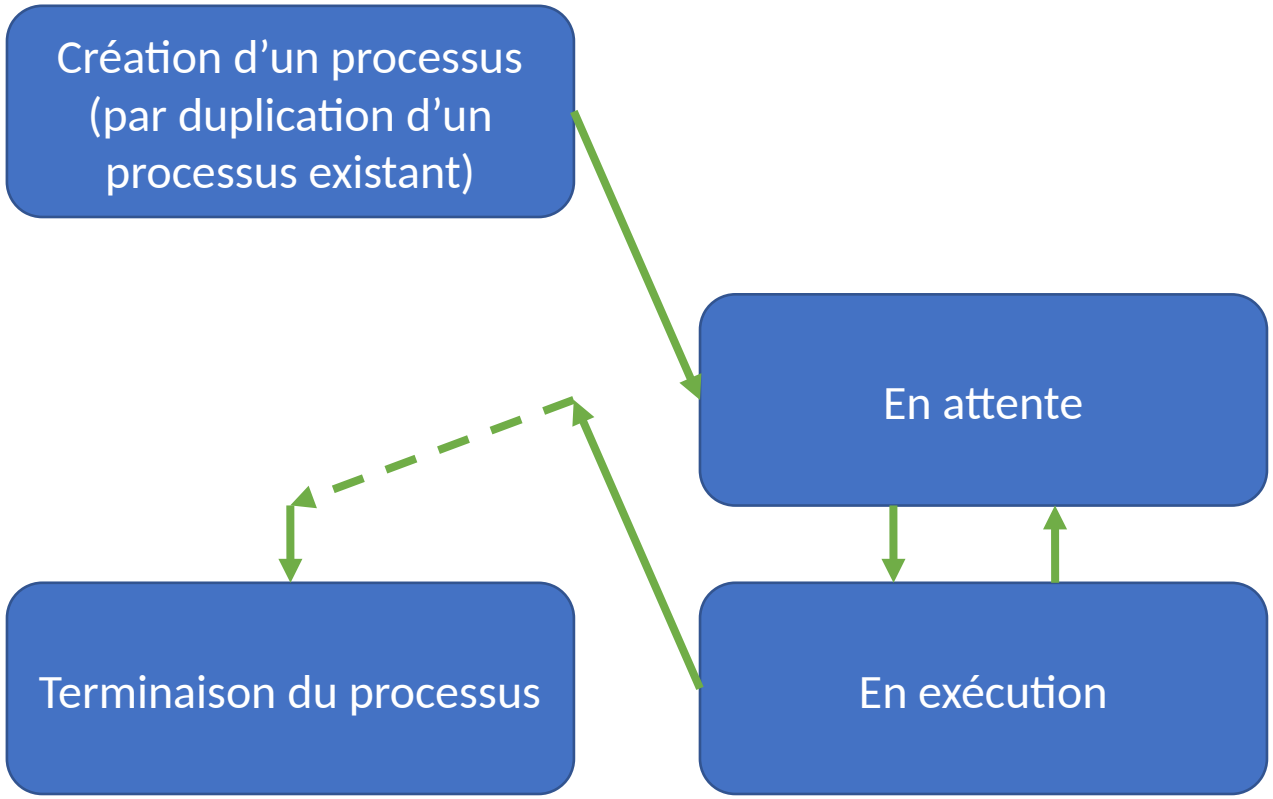
Processus

- Un processus est un programme en cours d'exécution
- Le système d'exploitation donne à chaque processus :
  - Une vision simplifiée du système => le processus a une vision indépendante du matériel, et les accès aux ressources sont homogénéisés
  - Des ressources propres
    - Le processus croit qu'il est seul à utiliser la mémoire (grâce à un mécanisme de mémoire virtuelle)
    - Le processus a sa propre copie des descripteurs de fichiers ouverts
    - Ainsi, chaque processus peut gérer ses propres ressources sans risquer d'interférence
  - Quelques ressources partagées (exemple : l'écran, le clavier, les fichiers, etc.)
  - Du temps d'exécution

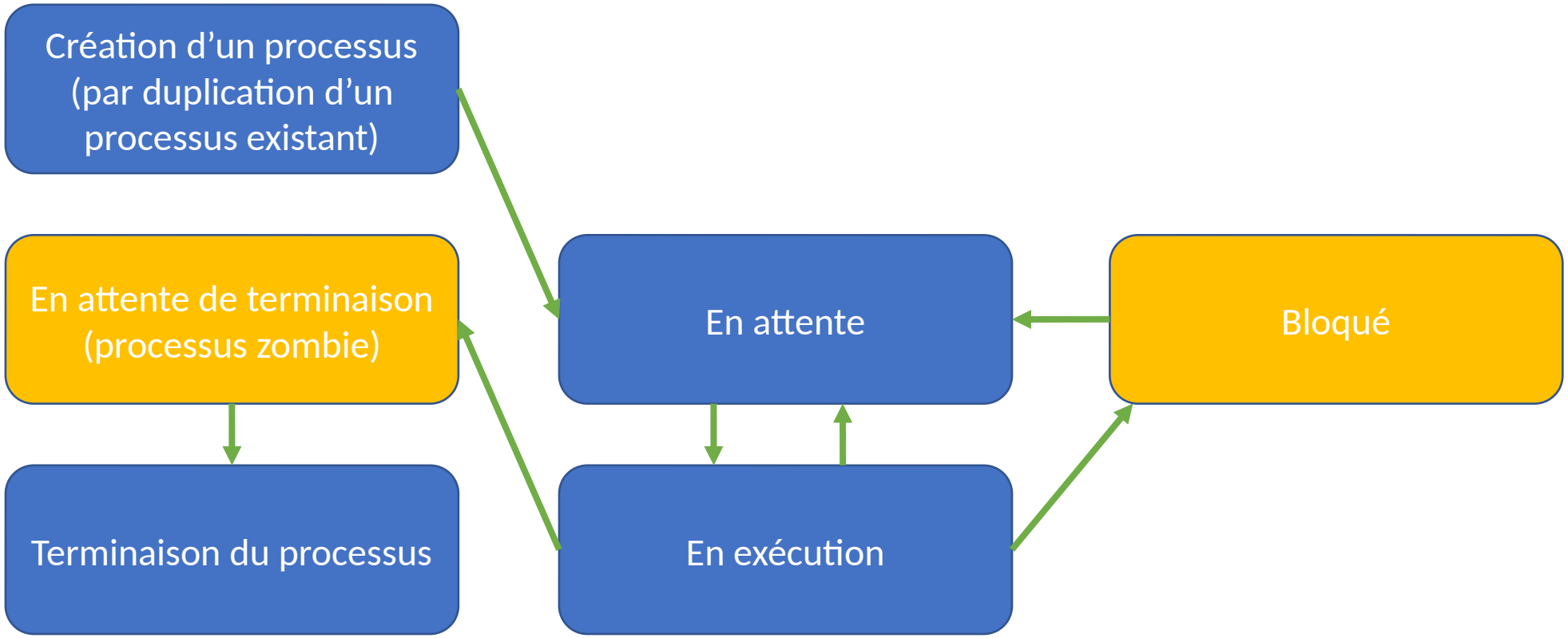
Multi-processus sur mono-processeur

- Les systèmes d'exploitation sont (presque) tous multi-processus, y compris s'il n'y a qu'un seul processeur sur l'ordinateur : ils peuvent exécuter plusieurs processus « en même temps »
  - Pour donner une impression de parallélisme, le processeur exécute les processus les uns après les autres pendant une courte durée de temps
  - Le basculement d'exécution d'un processus à un autre s'appelle la commutation. Ce basculement est très fréquent, et consomme des ressources (car il faut sauvegarder le contexte de l'ancien processus et charger le contexte du nouveau processus), c'est ce qui en fait un mécanisme dit « lourd »
  - L'ordre d'exécution des processus est défini par l'ordonnanceur

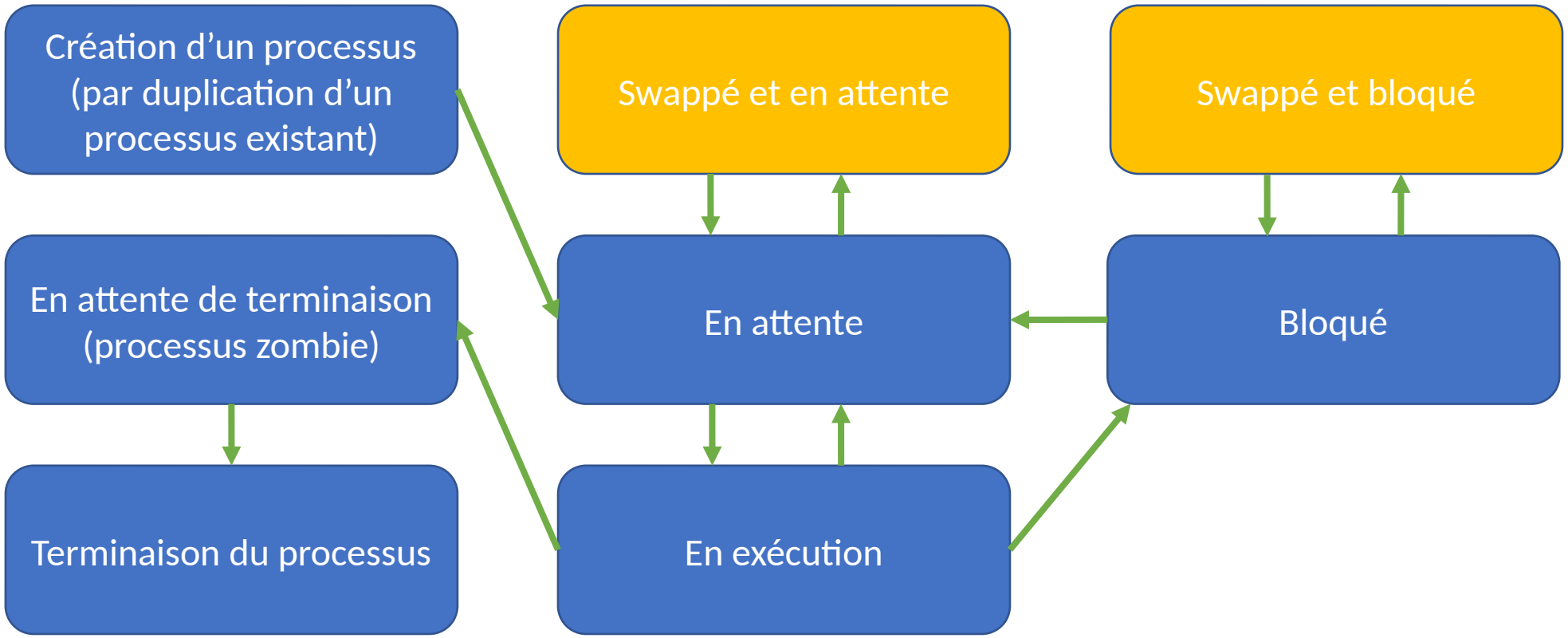
Cycle de vie d’un processus



Cycle de vie d’un processus



Cycle de vie d’un processus



Threads

- Constat : la gestion des processus est lourde pour le système
  - Il y a beaucoup de changements de contexte
  - L'indépendance des ressources propres à chaque processus rend les communications entre processus coûteuses
- Un *thread* (ou fil d'exécution) est un sous-processus
  - Il possède la même mémoire que le processus
  - Mais il possède son pointeur d'exécution et sa pile d'appels : dit autrement, un *thread* peut exécuter deux parties différentes d'un même programme, et avoir ses propres variables locales

Risques liés au parallélisme

- La programmation parallèle n'est pas simple :
  - Si on utilise plusieurs processus pour un même programme, il faut trouver une solution pour échanger des informations (=> cf partie 2.3 sur les mécanismes de communication) de manière fiable
  - Si on utilise plusieurs *threads* pour un même programme, il faut fiabiliser les accès à la mémoire partagée (variables globales et variables allouées dynamiquement), aux fichiers partagés (si un *thread* ferme un fichier, le fichier est fermé dans les autres *threads*)

Risque – accès concurrent à la mémoire

- Si deux *threads* (ou processus) accèdent à une partie commune de la mémoire sans protection explicite, il peut y avoir des comportements incohérents
- Exemple :
  - Thread 1 écrit 111111 dans la mémoire / Thread 2 écrit 222222 dans ma mémoire
  - Si Thread 1 est exécuté avant Thread 2, la mémoire contiendra 222222 au final
  - Si Thread 2 est exécuté avant Thread 1, la mémoire contiendra 111111 au final
  - Si l'exécution de Thread 1 et Thread 2 est entrelacée, la mémoire peut contenir des valeurs incohérentes, par exemple 112221
- Exemple :
  - Thread 1 écrit « Bonjour\n », Thread 2 écrit « Aurevoir\n »
  - Un résultat incohérent peut être l'affichage de « BonAurejour\n » puis « voir\n »

2.3 – Mécanismes de synchronisation

(début du cours #3)

- Opérations atomiques et sections critiques
- Sémaphores
- Exemples

Opérations atomiques et section critique

- Une opération atomique est une opération qui ne peut pas être interrompue (le changement de *thread*/processus ne peut pas se faire pendant cette opération)
  - Les instructions assembleur (sans *pipeline*) sont atomiques (exemple : écriture d'une valeur constante en mémoire… selon la compilation)
  - Sous Linux, les appels systèmes sont atomiques (exemple : ouvrir un fichier)
- Une section critique est une partie d'un programme dans laquelle il ne faut jamais plus d'un seul processus/*thread* en cours d'exécution

Implémentation des sections critiques

- Il existe plusieurs mécanismes pour implémenter des sections critiques
  - Les sémaphores (cf après)
  - Les verrous (pour les fichiers notamment)
  - Les moniteurs
  - Etc.
- Ces mécanismes peuvent néanmoins causer des effets de bord :
  - Problème de famine : un processus/*thread* n'obtient jamais de temps d'exécution
  - Problème d'interblocage *(deadlock)* : deux processus/*threads* sont bloqués mutuellement, chacun en attente de l'autre

Sémaphores

- Mécanisme proposé par Dijkstra
- Variable partagée par plusieurs processus, avec des opérations atomiques permettant de réaliser l'exclusion mutuelle pour un accès à une section critique
- Opérations
  - Initialiser
  - P (*proberen*=tester) / attendre / prendre
  - V (*verhogen*=incrémenter) / signaler / libérer

Sémaphores – détail des opérations

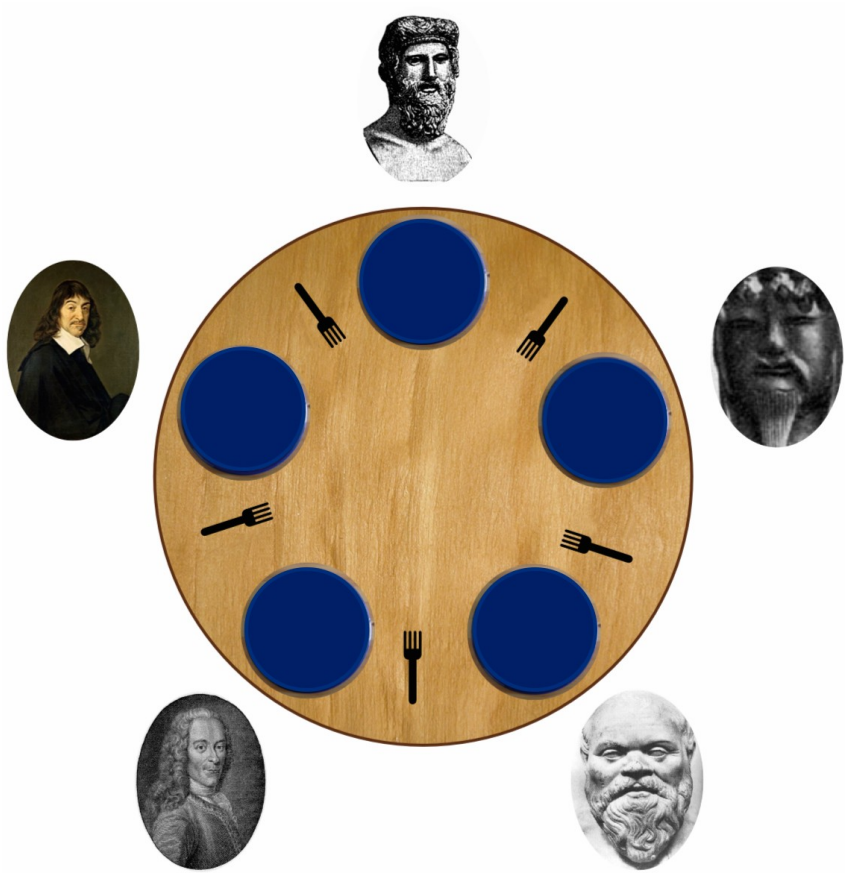
- Initialiser (atomique)
  - Affecte une valeur initiale à la variable partagée
- Prendre (atomique)
  - si la variable est négative ou nulle alors
    - bloquer le processus
    - ajouter le processus à la liste des processus bloqués
  - décrémenter la variable
- Libérer (atomique)
  - incrémenter la variable
  - si la liste des processus bloqués n'est pas vide alors
    - débloquer le premier processus de la liste

Remarques sur les sémaphores

- Une exclusion mutuelle peut être faite en initialisant le sémaphore à 1, et en entourant la section critique de wait(s) et de signal(s) pour tous les processus
- Un ordonnancement entre deux processus (P1 avant P2), aussi appelé barrière de synchronisation, peut être fait en initialisant la variable à 0, en faisant signal(s) dans P1, et en faisant wait(s) dans P2
- Erreurs fréquentes
  - Interblocage
  - Demander un sémaphore déjà obtenu par le processus en cours
  - Oublier de libérer un sémaphore bloqué

Exemple : le dîner des philosophes

- Description :



[Multithreading: Dining Philosophers Problem - Austin G. Walters (austingwalters.com)]

Exemple : le dîner des philosophes

- Solutions :
  - Avec sémaphores
    - Solution de Dijkstra, qui a inventé le problème
  - Avec un contrôleur centralisé
  - Avec parité
    - Les philosophes « pairs » commencent par prendre la fourchette de gauche, ceux « impairs » commencent par la droite

Exemple : le barbier endormi

- Description :

Sleeping Barber | MyCareerwise

Exemple : le barbier endormi

- Solution :
  - La solution utilise trois sémaphores : un pour les clients (qui compte les clients) qui démarre à 0, un pour le barbier (qui détermine s'il est occupé ou non) qui démarre à 0, et un pour l'exclusion mutuelle qui démarre à 1
  - Algorithme barbier( )
    - tantque vrai faire
      - wait(customer)
      - wait(mutex)
      - waiting--
      - signal(barber)
      - signal(mutex)
      - couper-cheveux( )

- Algorithme client( )
  - wait(mutex)
  - si (waiting<CHAIRS) alors
    - waiting++
    - signal(customer)
    - signal(mutex)
    - wait(barber)
    - se-faire-couper-cheveux( )
  - sinon
    - signal(mutex)

2.4 – Mécanismes de communication entre processus et threads

- Mécanismes spécifiques aux threads
  - Mémoire (allouée) nativement partagée => attention
  - Les descripteurs de fichier sont partagés => attention
- Mécanismes communs aux processus et aux threads
  - Segment de mémoire partagée
  - Fichiers
  - Sockets (= sortes de fichiers utilisés pour les communications réseaux)
  - Messages entre processus (sur un même processeur)
  - Signaux entre processus (sur un même processeur)
- Attention à la protection de toutes ces communications

2.5 – Complément : Introduction sur OpenMP et MPI

(début du cours #4)

- OpenMP et MPI sont des bibliothèques permettant de faire des programmes parallèles
  - OpenMP *(Open Multi Processing)* est utilisé sur des machines à mémoire partagée
  - MPI *(Message Passing Interface)* est utilisé sur des machines à mémoire distribuée, pour échanger des messages
  - Un même programme peut utiliser OpenMP et MPI

Complément : OpenMP

- Une application OpenMP est exécutée dans un processus unique, par plusieurs *threads*
  - Une tâche maître gère des tâches exécutables en parallèle
  - Les tâches exécutables en parallèle sont soit des tâches différentes, soit des tâches identiques (boucles)
  - Le nombre de tâches parallèles peut varier pendant le programme (clause NUM_THREADS)
- Compilation en C++ : -fopenmp

Complément : Exemple avec OpenMP

- Calcul de la somme des 10 premiers entiers :

```
#include <iostream>
using namespace std;
int main(int argc, char ** argv) {
        int sum = 0;
#pragma omp parallel for reduction(+: sum)
        for (int i=0; i<10; i++) {
        sum = sum+i;
        }
        cout « Total is » << sum << endl;
        return 0;
}
```

- Clauses et fonctions utiles :
  - #ifdef _OPENMP
  - #pragma omp parallel for reduction(+: sum)
  - #pragma omp parallel for num_threads(3)
  - #pragma omp critical
  - omp_get_thread_num()
  - omp_get_num_threads()

Complément : MPI

- MPI est défini comme un standard de communication, avec plusieurs implémentations
  - OpenMPI est une implémentation libre et *open source*
- Compilation en C : avec mpicc
- Exécution : avec mpirun –np X ./executable

Complément : Exemple avec OpenMPI (1/3)

https://www.geeksforgeeks.org/sum-of-an-array-using-mpi/ (19 may 2021)

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define N 10
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int tmpArray[N];
```

```
int main(int argc, char* argv[]) {
    int pid, np, el_per_p; // el_per_p = éléments à gérer par processus
    int n_el_received;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(
      MPI_COMM_WORLD, &pid); // <= numéro du processus
    MPI_Comm_size(
      MPI_COMM_WORLD, &np); // <= nombre de processus
```

Complément : Exemple avec OpenMPI (2/3)

```
if (pid==0) { // processus maître
   int index, i;
   el_per_p = N / np;
   if (np > 1) { // s'il y a plusieurs processus…
      for (i = 1; i < np - 1; i++) {
         index = i * el_per_p; // chaque processus reçoit…
         MPI_Send(&el_per_p,1,MPI_INT,i,0,MPI_COMM_WORLD);
         MPI_Send(&a[index],el_per_p,MPI_INT,i,
           0,MPI_COMM_WORLD); // … le nombre et le tableau (partiel)
      }
      index = i * el_per_p; // cas particulier du dernier processus
      int el_left = n - index;
      MPI_Send(&el_left,1,MPI_INT,i, 0,MPI_COMM_WORLD);
      MPI_Send(&a[index],elements_left,MPI_INT,
        i,0,MPI_COMM_WORLD);
   }
```

```
      // calcul effectué par le processus maître (sur le début du tableau)
      int sum = 0;
      for (i = 0; i < el_per_p; i++) {
         sum += a[i];
      }
      int tmp;
      for (i = 1; i < np; i++) { // agrégation des résultats des autres processus
         MPI_Recv(&tmp,1,MPI_INT,MPI_ANY_SOURCE,
           0,MPI_COMM_WORLD,&status);
         int sender = status.MPI_SOURCE;
         sum += tmp;
      }
      printf("Sum of array is : %d\n", sum); // résultat final
   } // fin du if (pid==0)
```

Complément : Exemple avec OpenMPI (3/3)

```
else { // processus esclaves
    MPI_Recv(&n_el_received,1,MPI_INT,
     0,0,MPI_COMM_WORLD,&status);
    MPI_Recv(&tmpArray,n_el_received,
     MPI_INT,0,0,MPI_COMM_WORLD,
     &status);
    int partial_sum = 0;
    for (int i=0; i<n_el_recieved; i++) {
        partial_sum += tmpArray[i];
    }
    MPI_Send(&partial_sum,1,MPI_INT,
     0,0,MPI_COMM_WORLD);
} // fin du else
    MPI_Finalize();
    return 0;
}
```