Introduction à la Programmation (sous Linux)

Basile Starynkévitch - starynkevitch.net/Basile
basile@starynkevitch.net mobile: +33 6 8501 2359

MontPelibre (Montpellier)

été 2020

Les opinions me sont personnelles

git 71ab9965929b963b https://montpellibre.fr/spip.php?article4875
Suivre les hyperliens donnés ici.

Licence

Le code LaTeX (`lualatex`, tutoriel sur www.tuteurs.ens.fr/logiciels/latex/) de cette présentation (construite avec le script `./build.sh`) est en
`github.com/bstarynk/intropprog-montpellibre-2020/`

Plan

Introduction

Couches logicielles
- généralités sur le logiciel
- architecture des microprocesseurs
- rôles et traits du noyau Linux
- fichiers et processus

Outils et langages des développeurs Linux
- outillage
- langages et paradigmes de programmation
- bases de données
- aspects algorithmiques

Exercice de programmation

Sommaire

Introduction

Couches logicielles

Outils et langages des développeurs Linux

Exercice de programmation

Introduction

les puissances, les bases, les nombres, les logarithmes

Les **puissances de dix**: $10^1 = 10$, $10^2 = 10 \times 10 = 100$,
$10^3 = 10 \times 10 \times 10 = 1000$, etc.... Par convention $10^0 = 1$.
Réciproque: **logarithme décimal**: $log_{10}1000 = 3$ (car $10^3 = 10000$)

Les **puissances de deux**: $2^1 = 2$, $2^2 = 2 \times 2 = 4$, $2^3 = 2 \times 2 \times 2 = 8$,
$2^4 = 2 \times 2 \times 2 \times 2 = 16$, etc.... Par convention $2^0 = 1$.
Réciproque: **logarithme binaire**: $log_2 16 = 4$ (car $16 = 2^4$) et $log_2 4 = 2$

Notation décimale (base dix) et binaire (base deux) des nombres entiers:
cent vingt trois, c'est **en base dix**, $123 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$
douze, c'est **en base deux**, $1100 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$

Introduction

Qu'est ce que l'information

un bit

Quantité "élémentaire" d'information. Le jeu de pile ou face transmet approximativement **un bit**, car il y a $2 = 2^1$ possibilités

Remarques:

- on a fait **abstraction** des autres possibilités (la pièce perdue dans le caniveau ...)
- on a fait une **simplification** et une **modélisation** de la réalité.
- on a évidemment $log_2\ 2 = 1$ car $2 = 2^1$

Mais l'*abstraction*, la *simplification*, la *modélisation* sont **au cœur de l'activité de programmation**.

Introduction

Combien de bits transmis au jeu de dés?

un dé a 6 faces, donc plus de 2 et moins de 3 bits transmis, puisque $4 = 2^2 < 6 < 2^3 = 8$

"informatiquement" on a transmis log_2 6 bits, donc $\approx 2,58496$ bits

Q: *combien de bits (environ) pour le jeu de la roulette?* (36 cas)

Introduction

Que faire avec un bit

Représenter toutes choses à deux possibilités

- valeur de vérité en logique : **vrai** ou **faux**
- comparaison ($<$ ou $>$) entre deux grandeurs (longueur, tension électrique, etc...)
- chiffre binaire
- signe $+$ ou $-$

Distinction entre chiffre et nombre

Introduction

Comment représenter *physiquement* un bit

Utiliser, ou simplifier, par un **phénomène physique à deux états**

- interrupteur marche/arrêt (donc tension électrique: $\approx 0V$ vs $1V$ à $3V$)
- pendule mécanique (à gauche ou à droite), horlogerie (Babbage)
- onde sonore
- magnétisation (tambour, disque dur)
- tubes à vide (ENIAC), transistors, circuits intégrés
- etc...

NB. Certaines technologies (mémoire Flash, SSD, Clefs USB) représentent deux bits par 4 états. Mais le matériel est imparfait.

Introduction

Notation décimale, binaire, octale, hexadécimale des nombres entiers

- *cent vingt trois* $= 123$ en décimal car $1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$
- *dix* $= 1010_b$ ou `0b1010` en binaire (binary digit = bit) car $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
- *douze* $= 14_o$ ou `014` ou `0o14` en octal car $1 \times 8^1 + 4 \times 8^0$
- *vingt* $= 14_h$ ou `0x14` en hexadécimal car $1 \times 16^1 + 4 \times 16^0$. Les chiffres après 9 sont A ($= 10$), B, C, D, E, F ($= 15$).
  Donc `0x1a` est vingt-six.

Remarque: un chiffre octal correspond à 3 bits (par exemple $5_o = 101_2$); un chiffre hexadécimal correspond à 4 bits (par exemple $11 = B = 1011_2$).

Introduction

Octets

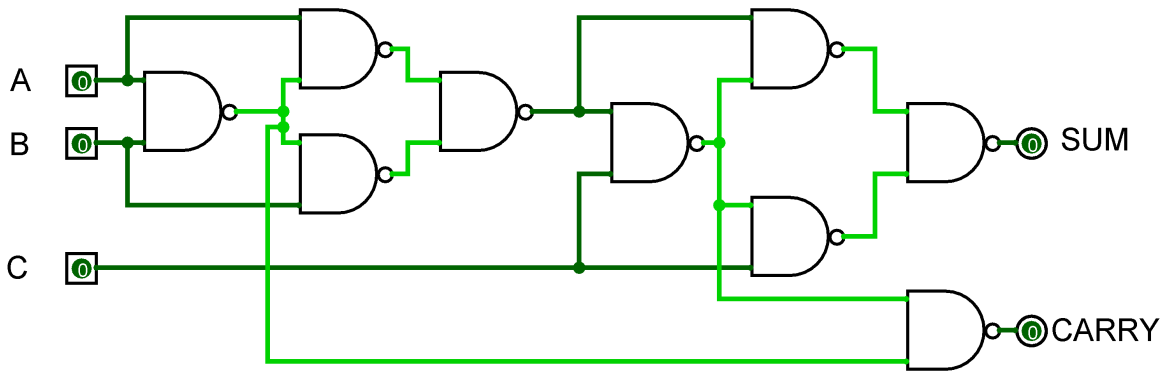
Un **octet** (anglais: *byte*) contient huit bits.

- petit entier entre 0 et $2^8 - 1$ donc 255
- entier signé entre -128 (-2^7) et +127 ($2^7 - 1$)
- codage de caractères (lettres, chiffres, ponctuation)
  en.wikipedia.org/wiki/ASCII
  `A` codé 65, `C` codé 67, `Z` codé 90, `a` codé 97, `?` codé 63
- codage universel Unicode UTF-8 sur un à quatre octets
  `en.wikipedia.org/wiki/UTF-8` et `utf8everywhere.org`.
  Par exemple ° codé sur deux octets `0xc2 0xb0`

Introduction

Circuits logiques combinatoires

Voir www.positron-libre.com/cours/electronique/logique-combinatoire/

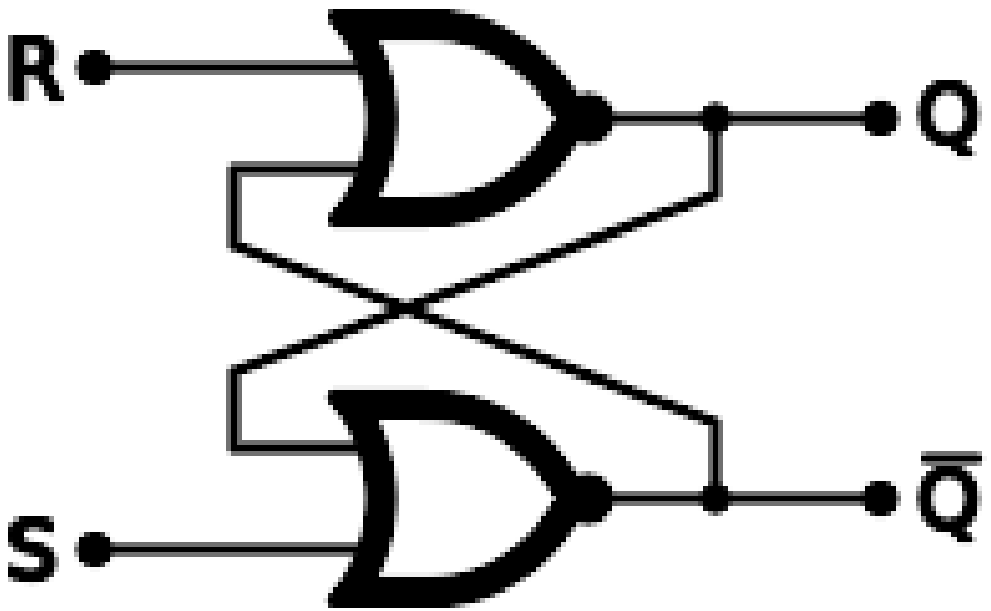


Additionneur un bit avec retenue à base de portes **NAND** (et-non)
Les portes **NAND** ou **NOR** sont *universelles* et simples à réaliser (quelques transistors).

Introduction

Circuits logiques circulaires

La bascule flip-flop (verrou RS) à deux portes **NOR** (ou-non) têtes-bêches mémorise un bit.



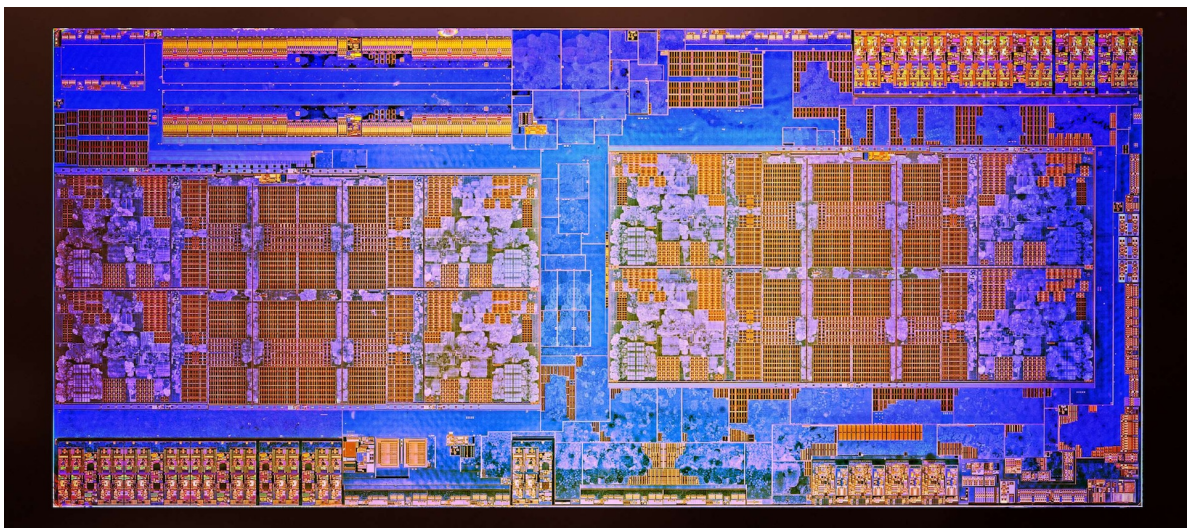
Voir www.paturage.be/electro/inforauto/portes/bascule.html

Introduction

Combinaisons de portes logiques

En combinant beaucoup (millions) de portes logiques on produit les microprocesseurs actuels (milliard de transistors, AMD ThreadRipper Ryzen)

`cdn.arstechnica.net/wp-content/uploads/2017/02/ryzen-die.jpg`



Introduction

Ordres de grandeurs économiques en 2020

- investissement pour la R&D, la fabrication, l'usine pour un microprocesseur haut de gamme: dix milliards de US$
- nombre de transistors par puce: milliard (sur quelques centimètres carrés)
- prix de vente (après test): environ mille dollars / pièce (milliers)
- taux de rendement: quelques pourcents, la plupart des puces sont défaillantes
- taux de panne par porte : moins d'une défaillance par heure
- consommation énergetique du numérique: quelques pourcents de la production électrique française
- prix de vente d'un ordinateur: centaines d'euros.
- plusieurs "ordinateurs" par habitant (microcontrôleurs, informatique embarquée, téléphones portables, datacenters)
- productivité d'un développeur: quelques dizaines de milliers de ligne de code source par an.

Introduction

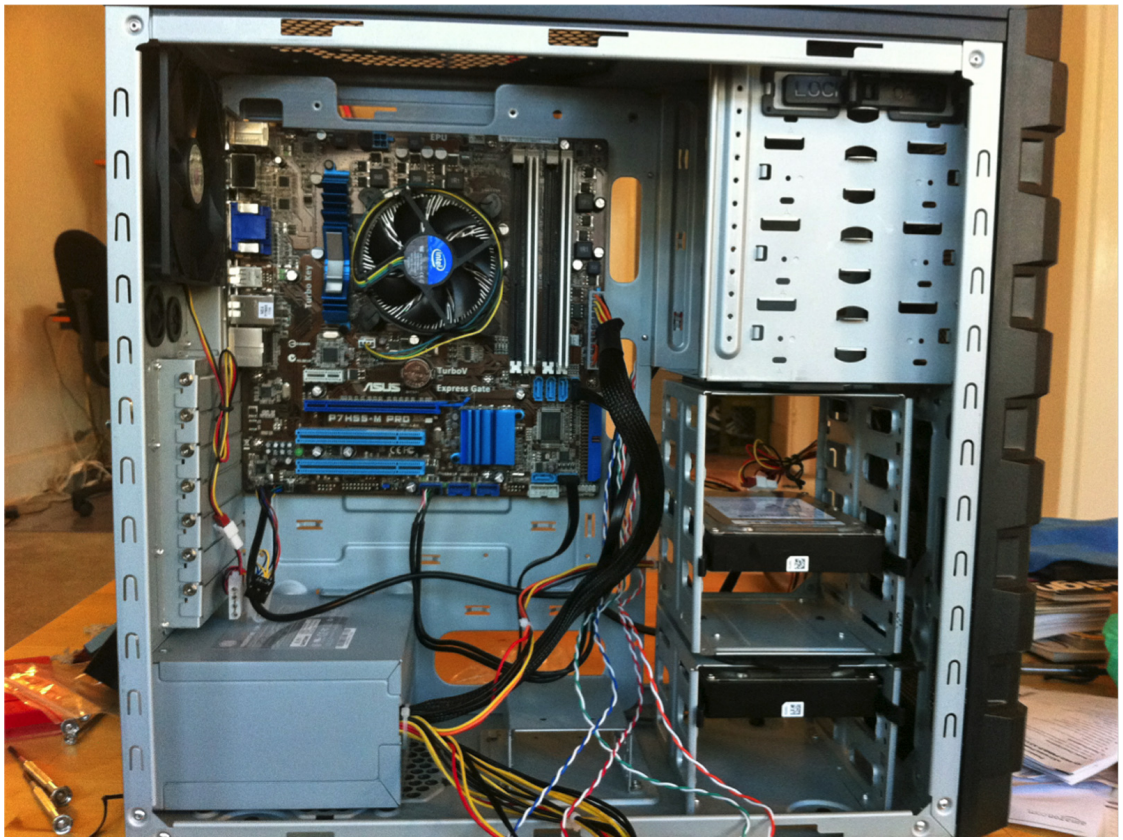
Ordres de grandeurs technologiques en 2020

- 8 cœurs de processeur à 3GHz
- mémoire cache: 16 Mo (accès 10ns, bande passante 0,5To/s)
- taille mémoire RAM d'un ordinateur : 32 Goctets
- temps d'accès RAM: 200 ns
- bande passante RAM: centaine Mo/seconde
- taille disque SSD : un demi téra-octet
- puissance thermique dissipée en charge: 250W
- temps d'accès SSD: 50 microsecondes pour un bloc de 4Ko.

Introduction

Vue interne d'un ordinateur de bureau en 2020

images.app.goo.gl/BMQxP7c4Zcyrh3XD7



Sommaire

Introduction

Couches logicielles
- généralités sur le logiciel
- architecture des microprocesseurs
- rôles et traits du noyau Linux
- fichiers et processus

Outils et langages des développeurs Linux

Exercice de programmation

Couches logicielles

Les différents types de logiciels

- logiciel et langage de description des circuits (VHDL, SystemC)
- firmware - interne à la souris (langage machine 8051), au disque dur, à la carte réseau
- logiciel de démarrage (UEFI, BIOS): charge le système d'exploitation
- noyau (par exemple Linux) du système d'exploitation, gère les fichiers et les processus
- utilitaires systèmes (gestion du réseau, ...)
- interface graphique ou navigateur Web
- bibliothèques logicielles standard (Qt, `libc`)
- outils logiciels pour le développeur (compilateur)
- logiciels applicatifs (bureautique) ou métiers (aéronautique)
- scripts utilisateurs et fichiers de configuration
- commandes, schémas de bases de données, langage d'imprimante (PDF, ...)

Couches logicielles

Aspects légaux (code pénal)

En France

article 323-1 et suivants du Code Pénal

Le fait d'accéder ou de se maintenir, frauduleusement, dans tout ou partie d'un système de traitement automatisé de données est puni de deux ans d'emprisonnement et de 60 000 € d'amende.
Lorsqu'il en est résulté soit la suppression ou la modification de données contenues dans le système, soit une altération du fonctionnement de ce système, la peine est de trois ans d'emprisonnement et de 100 000 € d'amende.
Lorsque les infractions prévues aux deux premiers alinéas ont été commises à l'encontre d'un système de traitement automatisé de données à caractère personnel mis en œuvre par l'Etat, la peine est portée à cinq ans d'emprisonnement et à 150 000 € d'amende.

Couches logicielles

Aspects légaux (RGPD)

En Europe, obligation d'avoir le consentement éclairé et préalable pour tout traitement de données à caractère personnel. **Règlement Général sur la Protection des Données**

Voir `donnees-rgpd.fr/definitions/rgpd-pour-les-nuls`, `www.laquadrature.net` et `april.org` pour en savoir plus.

Les pénalités sont dissuasives

Couches logicielles

Aspects légaux (copyright et logiciels libres, brevets logiciels, vente liée)

copyrights logiciels et logiciels libres: Voir `april.org` et `aful.org` et `eff.org` et `fsf.org` et `opensource.org` et `adullact.org` et `montpellibre.fr` et `cecill.info` *etc...* (ou votre juriste préféré) pour en savoir plus.

Vente liée matériel + logiciel : voir `non.aux.racketiciels.info` et `bons-constructeurs-ordinateurs.info`

NB. **Ne pas inventer sa propre licence logicielle libre** sans assistance juridique: le droit est aussi complexe que l'informatique!

PS. Attention aux incompatibilités de licences logicielles entre composants logiciels

Couches logicielles

Aspects techniques et sociaux

Code binaire ou **code machine**
Il est executé par l'ordinateur. Totalement illisible. **Toujours généré** par d'autres logiciels.

Code source
Il est écrit par l'informaticien et partagé avec d'autres développeurs.
Contient plein de choses pour les humains, inutiles aux ordinateurs.

Code intermédiaire
Des logiciels transforment le code source en code binaire, en passant par des representations intermédiaires. Dont le **code objet** ("object code") et les bibliothèques partagées ("shared libraries").

Méta-programmation
Beaucoup de logiciels manipulent, analysent, génèrent, gèrent du code.
Méta-programmer, c'est écrire un programme qui manipule le code d'autres programmes.

logiciel et architecture des microprocesseurs

Jeu d'instructions machine, registres, conventions d'appel

application binary interface. Document (parfois, 800 pages) décrivant le comportement souhaité d'un microprocesseur et son **jeu d'instructions machine** ("instruction set architecture")

registres d'un processeur. Mémoire ultra-rapide sur la puce. Souvent, une à trois douzaines de mots de 16, 32 ou 64 bits

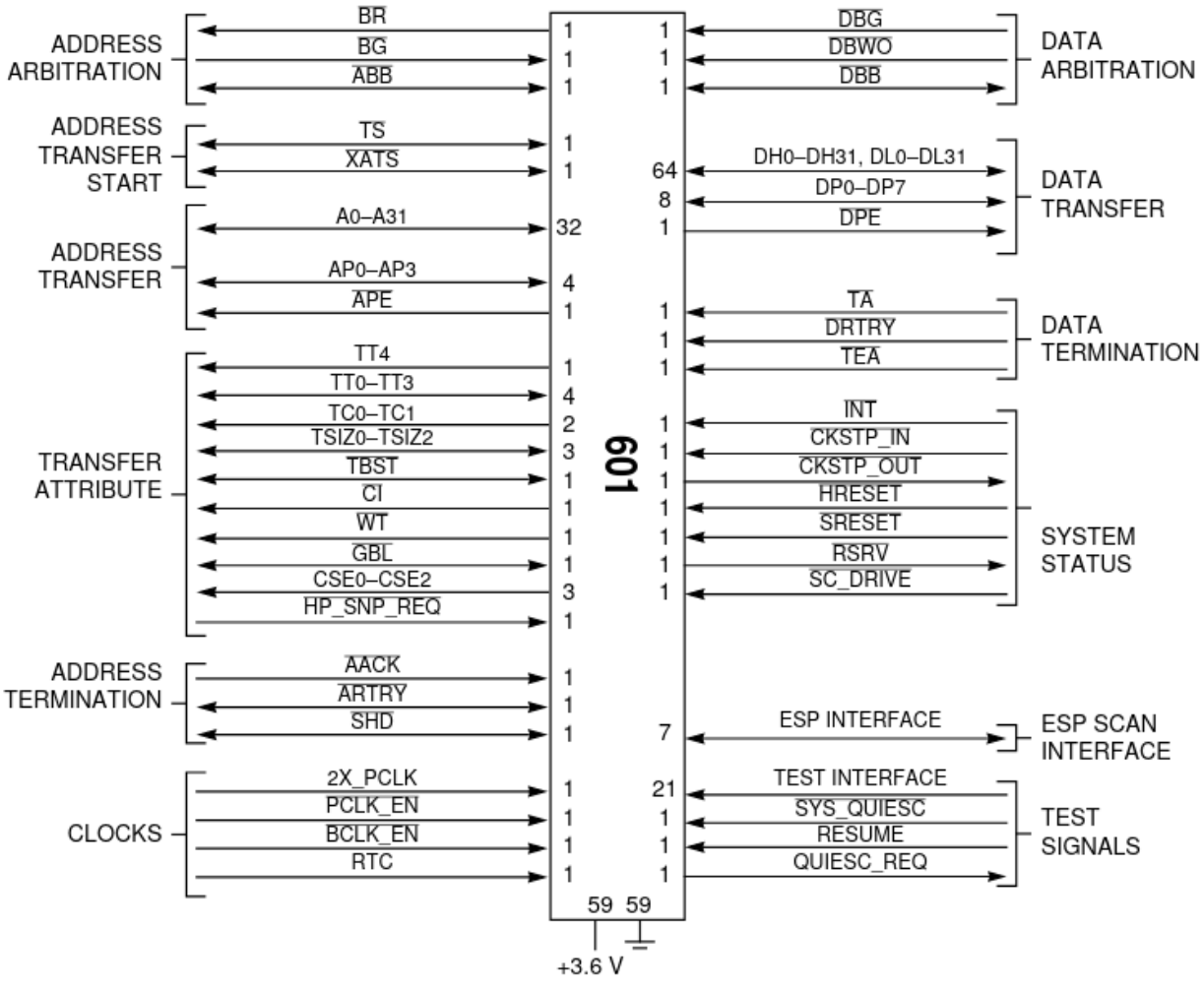
conventions d'appel. Règles d'usage des registres, pour des sous-programmes différents puissent être assemblés ensemble.

exemples: `riscv.org` (cartes graphiques, microcontroleurs), `x86-64` (PCs actuels), ARM (tablettes, téléphones, automobile), PowerPC

Le x86 (tous nos PCs) est *très complexe* pour des raisons historiques de compatibilité ascendante. Une vue pédagogique simplifiée est le Y86 (fictif).

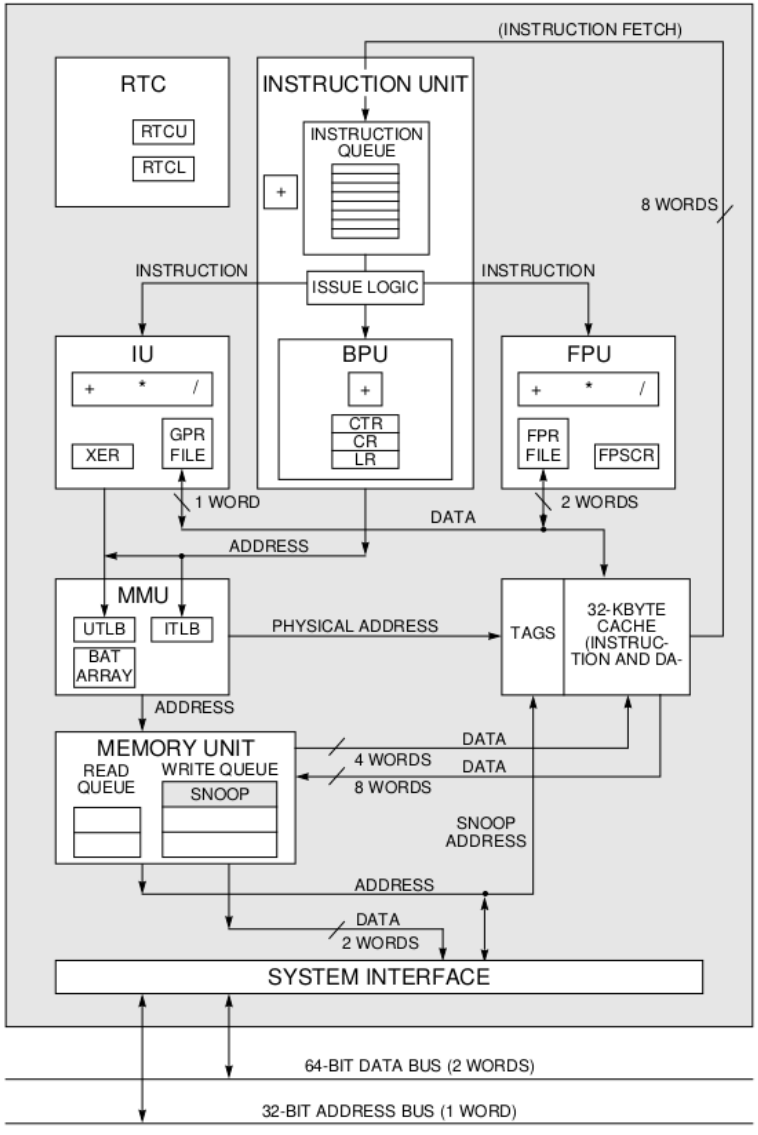
logiciel et architecture des microprocesseurs

Exemple: PowerPC NXP 601 : broches et signaux



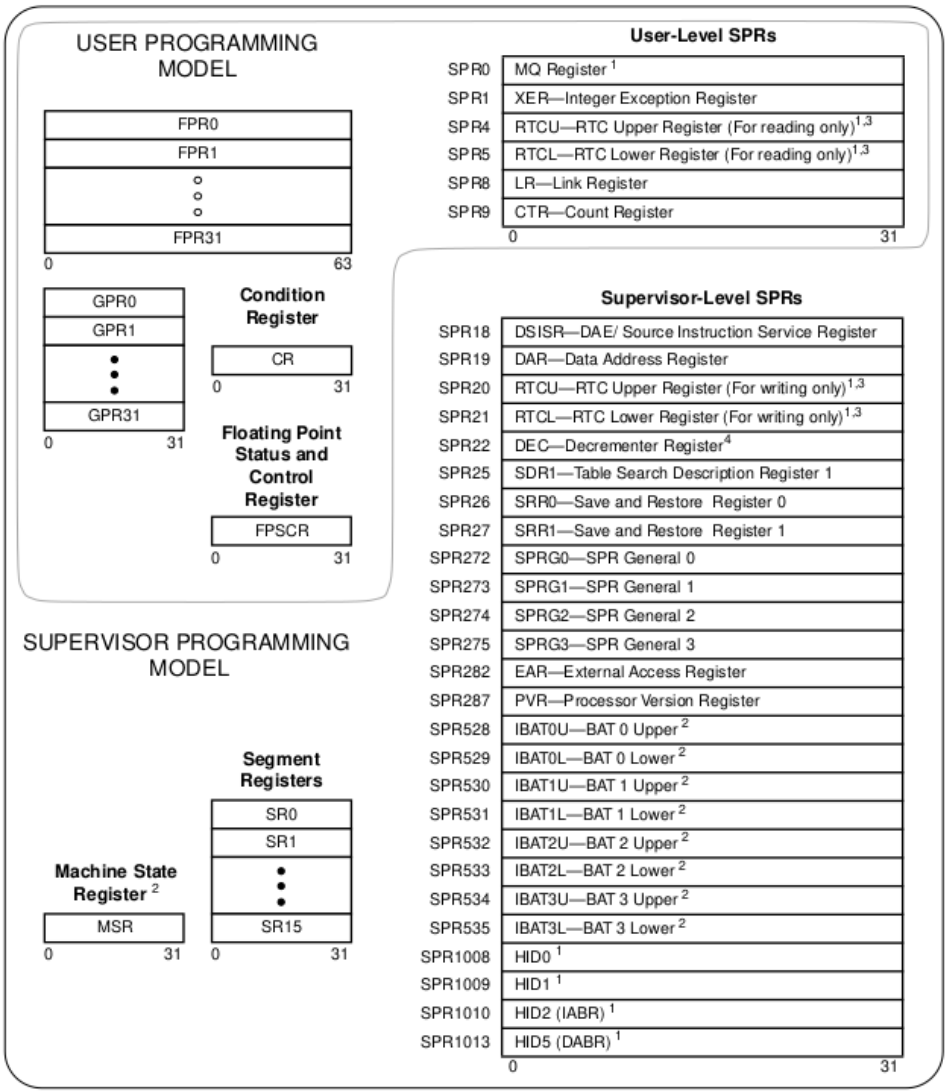
logiciel et architecture des microprocesseurs

Exemple: PowerPC NXP 601 : microarchitecture



logiciel et architecture des microprocesseurs

Exemple: Architecture PowerPC601 : registres complets



[1] 601-only registers. These registers are not necessarily supported by other PowerPC processors.
[2] These registers may be implemented differently on other PowerPC processors. The PowerPC architecture defines two sets of

logiciel et architecture des microprocesseurs

Exemple: Architecture PowerPC : registres utilisateurs

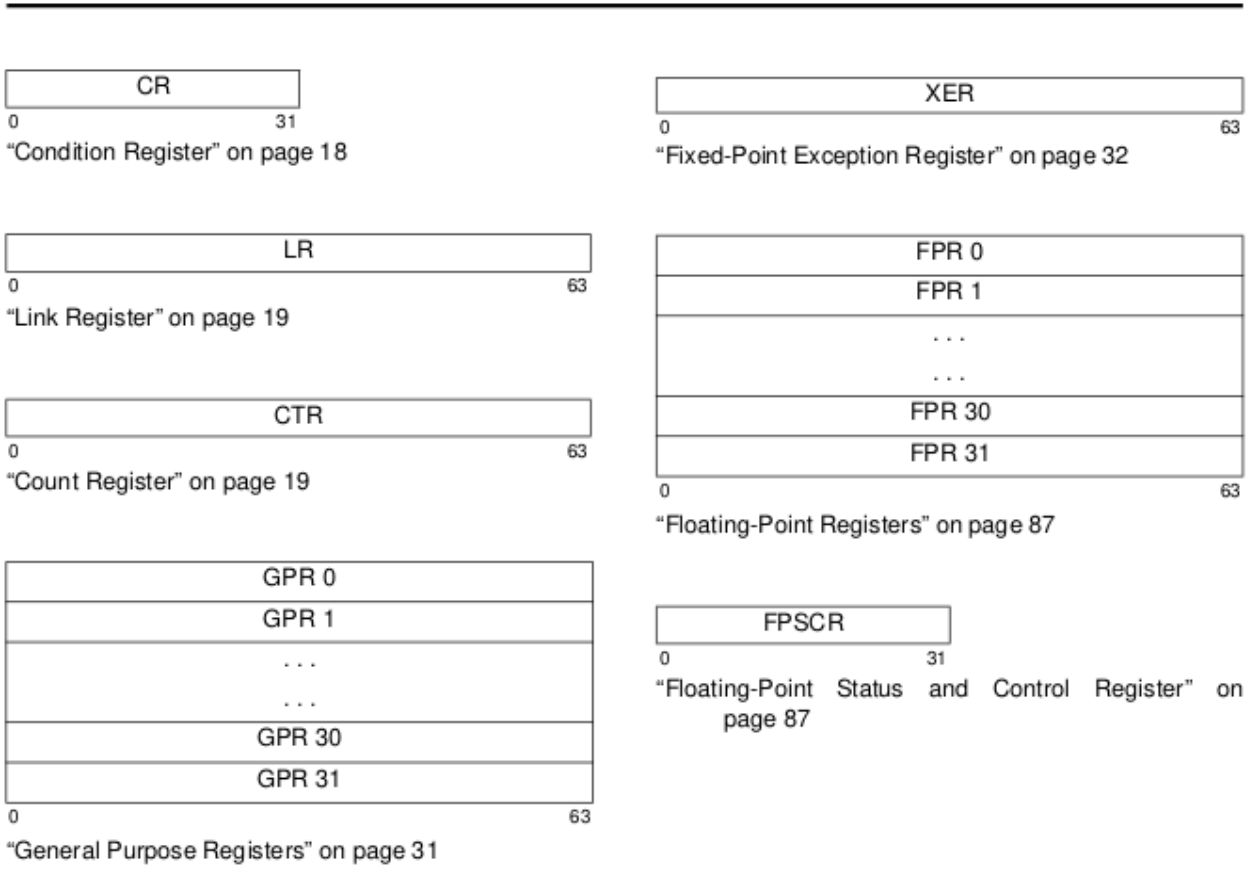
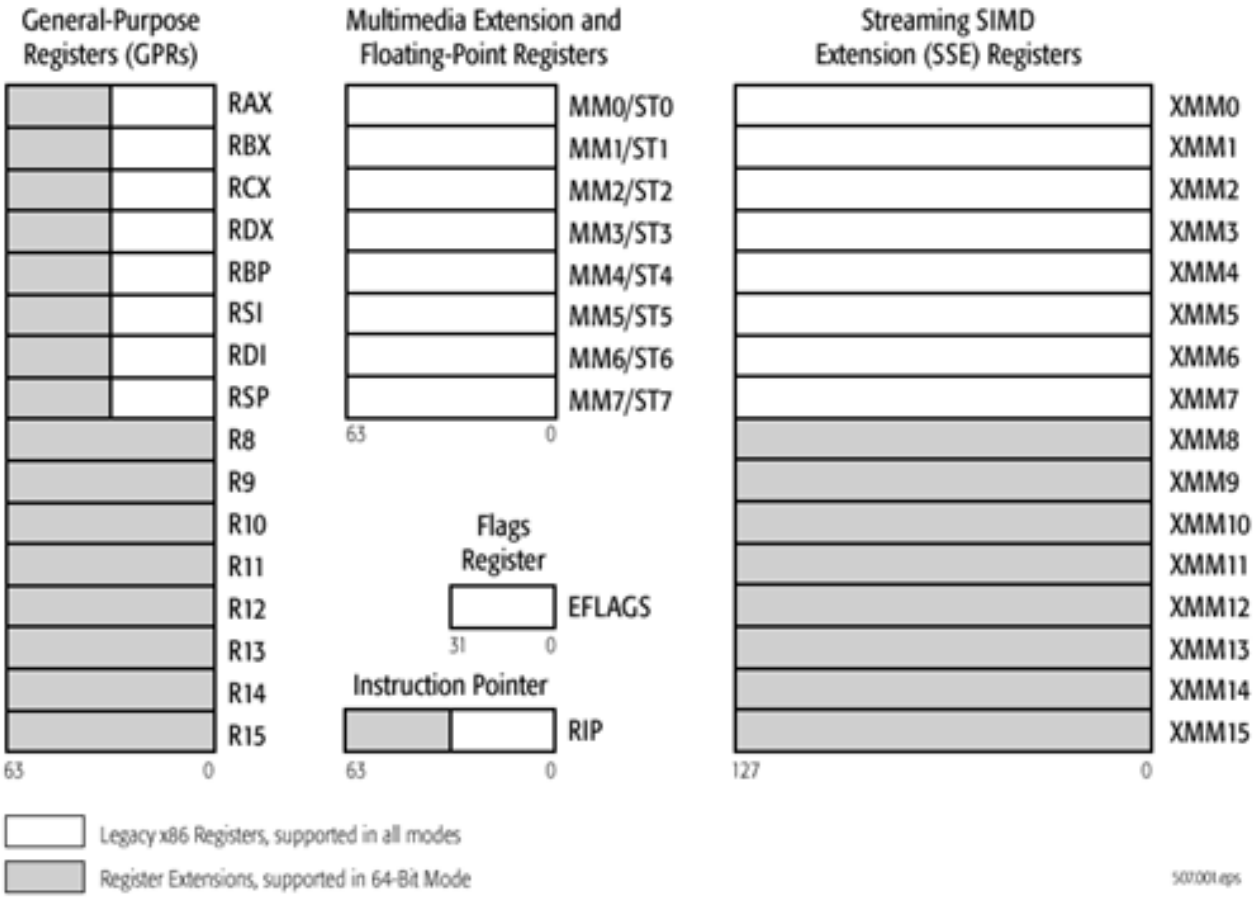


Figure 2. Power PC user register set

avec `GPR0` toujours à 0

logiciel et architecture des microprocesseurs

Exemple: Architecture x86-64 : registres utilisateurs



où `RSP` est le pointeur de pile (“stack pointer”)

logiciel et architecture des microprocesseurs

Exemple: Architecture PowerPC : registre de conditions

2.2.4 Condition Register (CR)

The condition register (CR) is a 32-bit register that reflects the result of certain operations and provides a mechanism for testing and branching. The bits in the CR are grouped into eight 4-bit fields, CR0–CR7, as shown in Figure 2-5.

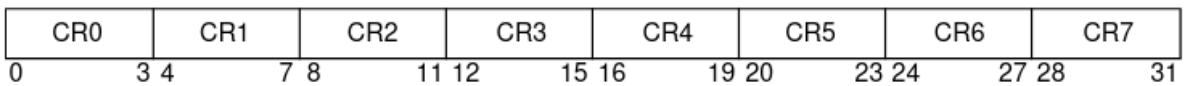


Figure 2-5. Condition Register (CR)

The CR fields can be set in one of the following ways:

- Specified fields of the CR can be set by a move instruction (**mtcrf**, or **mcrfs**) to the CR from a GPR.
- Specified fields of the CR can be moved from one CR*x* field to another with the **mcrf** instruction.
- A specified field of the CR can be set by a move instruction (**mcrxr**) to the CR from the XER.
- Condition register logical instructions can be used to perform logical operations on specified bits in the condition register.
- CR0 can be the implicit result of an integer operation.
- CR1 can be the implicit result of a floating-point operation.
- A specified CR field can be the explicit result of either an integer or floating-point compare instruction.

Branch instructions are provided to test individual CR bits. The condition register is cleared by hard reset.

logiciel et architecture des microprocesseurs

Exemple: Architecture PowerPC : registre d'état machine (MSR)

2.3.1 Machine State Register (MSR)

The machine state register (MSR), shown in Figure 2-11, is a 32-bit register that defines the state of the processor. When an exception occurs, MSR bits, as described in Table 2-9, are altered as determined by the exception. The MSR can also be modified by the **mtmsr**, **sc**, and **rfi** instructions. It can be read by the **mfmsr** instruction. Note that in 64-bit PowerPC implementations, the MSR is a 64-bit register.

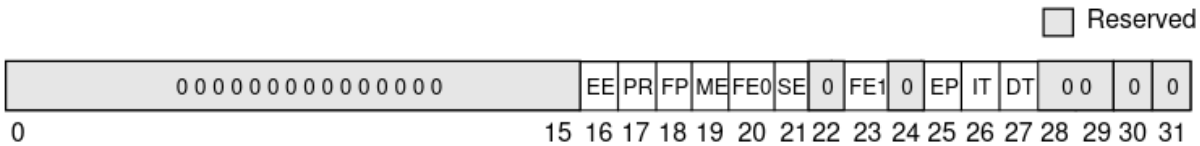


Figure 2-11. Machine State Register (MSR)

Table 2-9 shows the bit definitions for the MSR.

Voir
www.nxp.com/files-static/32bit/doc/user_guide/MPC601UM.pdf

logiciel et architecture des microprocesseurs

Exemple: Architecture PowerPC : rôles des bits du registre d'état machine (MSR)

Table 2-9. Machine State Register Bit Settings

Bit(s)	Name	Description
0–15	—	Reserved*
16	EE	External exception enable 0 While the bit is cleared the processor delays recognition of external interrupts and decrementer exception conditions. 1 The processor is enabled to take an external interrupt or the decrementer exception.
17	PR	Privilege level 0 The processor can execute both user- and supervisor-level instructions. 1 The processor can only execute user-level instructions.
18	FP	Floating-point available 0 The processor prevents dispatch of floating-point instructions, including floating-point loads, stores, and moves. 1 The processor can execute floating-point instructions, and can take floating-point enabled exception type program exceptions.
19	ME	Machine check enable 0 A checkstop is taken, unless either HID0[CE] or HID0[EM] is cleared (disabled), in which case the machine check exception is taken. 1 Machine check exceptions are enabled. In the 601, this bit is set after a hard reset, although the PowerPC architecture specifies that this bit is cleared.
20	FE0	Floating-point exception mode 0 (See Table 2-10).
21	SE	Single-step trace enable 0 The processor executes instructions normally. 1 The processor generates a single-step trace exception upon the successful execution of the next instruction. In the 601 this is implemented as a run-mode exception; the PowerPC architecture defines this as a trace exception. When this bit is set, the processor dispatches instructions in strict program order. Successful execution means the instruction caused no other exception. Single-step tracing may not be present on all implementations.

logiciel et architecture des microprocesseurs

Deux modes de fonctionnement des microprocesseurs

- **mode superviseur**: toutes les instructions machines sont permises, y compris les plus dangereuses (entrée/sorties physiques, configuration de l'espace d'addressage, modification de la fréquence ou de l'énergie consommée). Pour **le code du noyau** supposé **fiable et correct**.
- **mode utilisateur**: les instructions machines dangereuses (E/S, mémoire virtuelle, ...) sont interdites et ne sont pas executées. Pour **le code applicatif** : quand il plante, le système (noyau) prend la main.

Les **interruptions** font passer du mode utilisateur au mode superviseur. Elles sont **internes** (instruction illicite en mode utilisateur, division par 0, défaut de page, appel système...) ou **externes** (temporisation, surchauffe, paquet Ethernet reçu, fin de lecture d'un bloc du disque, ...)

logiciel et architecture des microprocesseurs

Exemple: Architecture PowerPC : formats d'instructions machine

1.7.1 I-Form

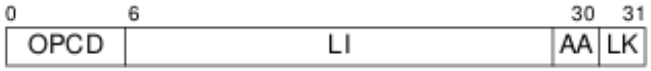


Figure 3. I instruction format

1.7.2 B-Form

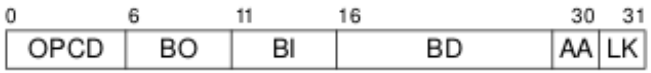


Figure 4. B instruction format

1.7.3 SC-Form

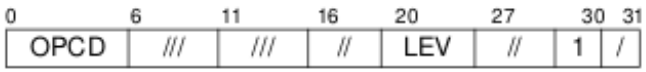


Figure 5. SC instruction format

1.7.4 D-Form

0	6	11	16	31
OPCD	RT	RA	D	
OPCD	RT	RA	SI	
OPCD	RS	RA	D	
OPCD	RS	RA	UI	
OPCD	BF / L	RA	SI	
OPCD	BF / L	RA	UI	
OPCD	TO	RA	SI	
OPCD	FRT	RA	D	
OPCD	FRS	RA	D	

Figure 6. D instruction format

1.7.5 DS-FORM

0	6	11	16	30 31
OPCD	RT	RA	DS	XO
OPCD	RS	RA	DS	XO

Figure 7. DS instruction format

logiciel et architecture des microprocesseurs

Exemple: Architecture PowerPC : appel système

System Call SC-form

sc LEV

[POWER mnemonic: svca]

17	///	///	//	LEV	//	1	/
0	6	11	16	20	27	30	31

This instruction calls the system to perform a service. A complete description of this instruction can be found in Book III, *PowerPC Operating Environment Architecture*.

The use of the LEV field is described in Book III. The LEV values greater than 1 are reserved, and bits 0:5 of the LEV field (instruction bits 20:25) are treated as a reserved field.

When control is returned to the program that executed the *System Call* instruction, the contents of the registers will depend on the register conventions used by the program providing the system service.

This instruction is context synchronizing (see Book III, *PowerPC Operating Environment Architecture*).

Special Registers Altered:

Dependent on the system service

logiciel et architecture des microprocesseurs

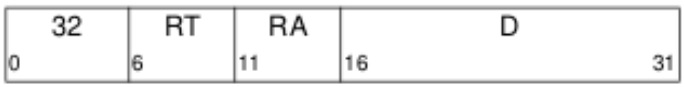
Exemple: Architecture PowerPC : chargement d'un registre

Version 2.02

Load Word and Zero D-form

lwz RT,D(RA)

[POWER mnemonic: l]




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + EXTS(D)
RT ← 32 0 || MEM(EA, 4)
```

Let the effective address (EA) be the sum (RA|0)+ D. The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

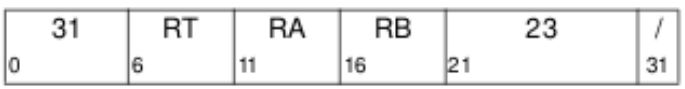
Special Registers Altered:

None

Load Word and Zero Indexed X-form

lwzx RT,RA,RB

[POWER mnemonic: lx]




```
if RA = 0 then b ← 0
else          b ← (RA)
EA ← b + (RB)
RT ← 32 0 || MEM(EA, 4)
```

Let the effective address (EA) be the sum (RA|0)+ (RB). The word in storage addressed by EA is loaded into $RT_{32:63}$. $RT_{0:31}$ are set to 0.

Special Registers Altered:

None

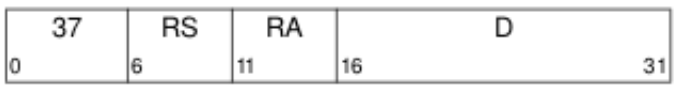
logiciel et architecture des microprocesseurs

Exemple: Architecture PowerPC : écriture d'un registre

Store Word with Update D-form

stwu RS,D(RA)

[POWER mnemonic: stu]




```
EA ← (RA) + EXTS(D)
MEM(EA, 4) ← (RS)32:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ D. $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

EA is placed into register RA.

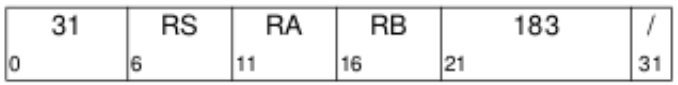
If RA=0, the instruction form is invalid.

Special Registers Altered:
None

Store Word with Update Indexed X-form

stwux RS,RA,RB

[POWER mnemonic: stux]




```
EA ← (RA) + (RB)
MEM(EA, 4) ← (RS)32:63
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (RB). $(RS)_{32:63}$ are stored into the word in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:
None

logiciel et architecture des microprocesseurs

Exemple: Architecture PowerPC : multiplication

Multiply Low Doubleword XO-form

mulld	RT,RA,RB	(OE=0 Rc=0)
mulld.	RT,RA,RB	(OE=0 Rc=1)
mulldo	RT,RA,RB	(OE=1 Rc=0)
mulldo.	RT,RA,RB	(OE=1 Rc=1)

31	RT	RA	RB	OE	233	Rc
0	6	11	16	21	22	31

$prod_{0:127} \leftarrow (RA) \times (RB)$
$RT \leftarrow prod_{64:127}$

The 64-bit operands are (RA) and (RB). The low-order 64 bits of the 128-bit product of the operands are placed into register RT.

If OE=1 then OV is set to 1 if the product cannot be represented in 64 bits.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:

CR0	(if Rc=1)
SO OV	(if OE=1)

Programming Note

The XO-form *Multiply* instructions may execute faster on some implementations if RB contains the operand having the smaller absolute value.

Multiply Low Word XO-form

mullw	RT,RA,RB	(OE=0 Rc=0)
mullw.	RT,RA,RB	(OE=0 Rc=1)
mullwo	RT,RA,RB	(OE=1 Rc=0)
mullwo.	RT,RA,RB	(OE=1 Rc=1)

[POWER mnemonics: muls, muls., mulso, mulso.]

31	RT	RA	RB	OE	235	Rc
0	6	11	16	21	22	31

$RT \leftarrow (RA)_{32:63} \times (RB)_{32:63}$

The 32-bit operands are the low-order 32 bits of RA and of RB. The 64-bit product of the operands is placed into register RT.

If OE=1 then OV is set to 1 if the product cannot be represented in 32 bits.

Both operands and the product are interpreted as signed integers.

Special Registers Altered:

CR0	(if Rc=1)
SO OV	(if OE=1)

Programming Note

For ***mulli*** and ***mullw***, the low-order 32 bits of the product are the correct 32-bit product for 32-bit mode.

logiciel et architecture des microprocesseurs

Exemple: Architecture PowerPC : comparaison

Compare Immediate D-form

cmpi BF,L,RA,SI

11	BF	/	L	RA	SI
0	6	9	10	11	16 31

```
if L = 0 then a ← EXTS((RA)32:63)
         else a ← (RA)
if      a < EXTS(SI) then c ← 0b100
else if a > EXTS(SI) then c ← 0b010
else                      c ← 0b001
CR4×BF:4×BF+3 ← c || XERSO
```

The contents of register RA ($(RA)_{32:63}$ sign-extended to 64 bits if L=0) are compared with the sign-extended value of the SI field, treating the operands as signed integers. The result of the comparison is placed into CR field BF.

Special Registers Altered:

CR field BF

Extended Mnemonics:

Examples of extended mnemonics for Compare Immediate:

Extended:		*Equivalent to:*	
cmpdi	Rx,value	cmpi	0,1,Rx,value
cmpwi	cr3,Rx,value	cmpi	3,0,Rx,value

Compare X-form

cmp BF,L,RA,RB

31	BF	/	L	RA	RB	0	/
0	6	9	10	11	16	21	31

```
if L = 0 then a ← EXTS((RA)32:63)
                b ← EXTS((RB)32:63)
           else a ← (RA)
                b ← (RB)
if      a < b then c ← 0b100
else if a > b then c ← 0b010
else               c ← 0b001
CR4×BF:4×BF+3 ← c || XERSO
```

The contents of register RA ($(RA)_{32:63}$ if L=0) are compared with the contents of register RB ($(RB)_{32:63}$ if L=0), treating the operands as signed integers. The result of the comparison is placed into CR field BF.

Special Registers Altered:

CR field BF

Extended Mnemonics:

Examples of extended mnemonics for Compare:

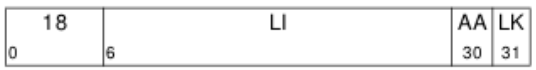
Extended:		*Equivalent to:*	
cmpd	Rx,Ry	cmp	0,1,Rx,Ry
cmpw	cr3,Rx,Ry	cmp	3,0,Rx,Ry

logiciel et architecture des microprocesseurs

Exemple: Architecture PowerPC : branchement

Branch I-form

b	target_addr	(AA=0 LK=0)
ba	target_addr	(AA=1 LK=0)
bl	target_addr	(AA=0 LK=1)
bla	target_addr	(AA=1 LK=1)




```
if AA then NIA ←iea EXTS(LI || 0b00)
else        NIA ←iea CIA + EXTS(LI || 0b00)
if LK then LR ←iea CIA + 4
```

target_addr specifies the branch target address.

If AA=0 then the branch target address is the sum of LI || 0b00 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If AA=1 then the branch target address is the value LI || 0b00 sign-extended, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

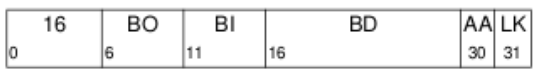
If LK=1 then the effective address of the instruction following the Branch instruction is placed into the Link Register.

Special Registers Altered:

LR (if LK=1)

Branch Conditional B-form

bc	BO,BI,target_addr	(AA=0 LK=0)
bca	BO,BI,target_addr	(AA=1 LK=0)
bcl	BO,BI,target_addr	(AA=0 LK=1)
bcla	BO,BI,target_addr	(AA=1 LK=1)




```
if (64-bit mode)
  then M ← 0
  else M ← 32
if ¬BO2 then CTR ← CTR - 1
ctr_ok ← BO2 | ((CTRM:63 ≠ 0) ⊕ BO3)
cond_ok ← BO0 | (CRBI = BO1)
if ctr_ok & cond_ok then
  if AA then NIA ←iea EXTS(BD || 0b00)
  else       NIA ←iea CIA + EXTS(BD || 0b00)
if LK then LR ←iea CIA + 4
```

The BI field specifies the Condition Register bit to be tested. The BO field is used to resolve the branch as described in Figure 21. *target_addr* specifies the branch target address.

If AA=0 then the branch target address is the sum of BD || 0b00 sign-extended and the address of this instruction, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If AA=1 then the branch target address is the value BD || 0b00 sign-extended, with the high-order 32 bits of the branch target address set to 0 in 32-bit mode.

If LK=1 then the effective address of the instruction following the *Branch* instruction is placed into the Link Register.

Le noyau Linux

abstraction et services fournis par le noyau Linux aux applications

- Linux est **multi-tâches**, **multi-utilisateurs**
- les **fichiers** et **répertoires**
- les **processus** (un programme en train de tourner) et les threads (filaments) - leur **isolation**
- l'**exécution d'un fichier exécutable**
- la **mémoire virtuelle**
- la **communication entre processus**
- la **gestion des périphériques** (écran, clavier, souris, clef USB, disque, ...)
- le **réseau** (Internet) et la **communication entre ordinateurs** (Ethernet, Wifi, ...)

Le noyau Linux fournit les appels systèmes listés en syscalls(2)

Le noyau Linux

La "philosophie d'Unix"

Unix philosophy (wikipedia)

The Unix philosophy, originated by Ken Thompson, is a set of cultural norms and philosophical approaches to minimalist, modular software development. It is based on the experience of leading developers of the Unix operating system. Early Unix developers were important in **bringing the concepts of modularity and reusability into software engineering practice**, spawning a "software tools" movement. Over time, the leading developers of Unix (and programs that ran on it) established a set of cultural norms for developing software; these norms became as important and influential as the technology of Unix itself; this has been termed the "Unix philosophy."

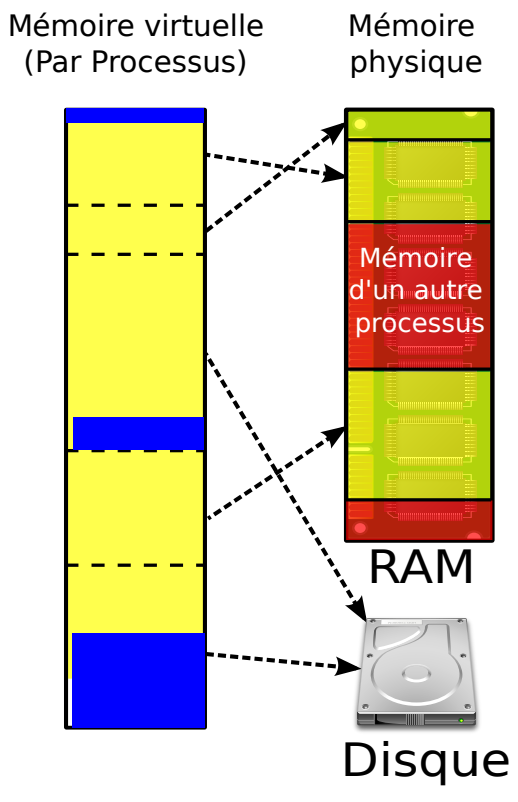
The Unix philosophy emphasizes building simple, short, clear, modular, and extensible code that can be easily maintained and repurposed by developers other than its creators. **The Unix philosophy favors composability as opposed to monolithic design**.

Accepter de mal faire quand ça marche souvent, pour faciliter le travail des collègues....

Le noyau Linux

mémoire virtuelle

Les addresses (par exemple sur 32 bits) dans les registres ne correspondent pas **directement** à la mémoire physique DRAM.
Le noyau du système d'exploitation gère la mémoire virtuelle (Wikipedia)

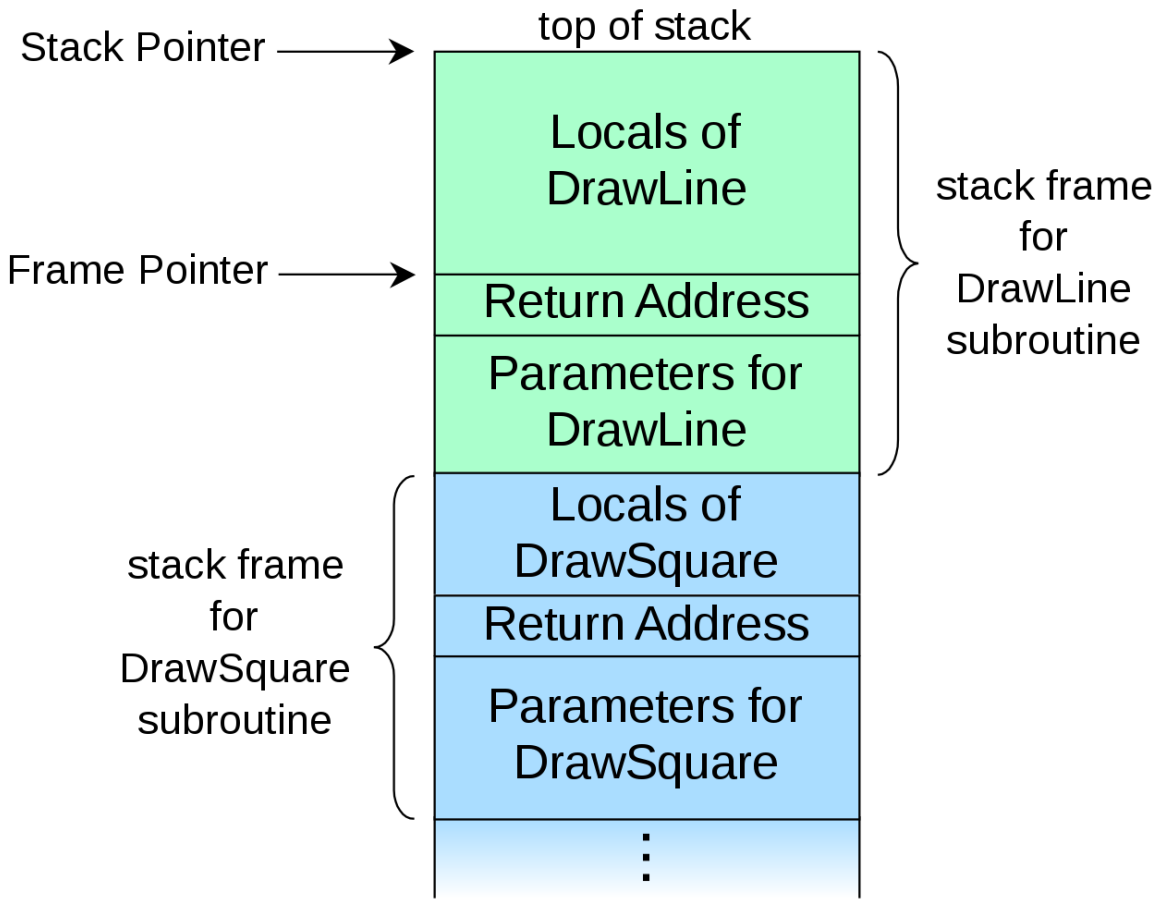


- l'**espace d'adressage** d'un processus est géré par **pages** de 4Ko
- une page pourrait être (pour son processus):
  1. interdite d'accès (en bleu)
  2. partagée avec d'autres processus
  3. projetée (en lecture, en écriture) sur un bout de fichier
  4. avec/sans du code machine exécutable
  5. à lecture seule
  6. à lecture et écriture

Le noyau Linux

pile d'exécution (call stack)

wikipedia pile d'exécution



Le noyau Linux

pagination de la mémoire virtuelle

Commande Linux pmap(1) utilisant le système de pseudo-fichiers `/proc/`, voir proc(5) et (pour cyber-sécurité) Address space layout randomization
Essayez `pmap $$` puis `/bin/cat /proc/self/maps` dans un terminal de commandes
Ou bien avec `sash`, le stand-alone shell

```
pmap 41424
41424: sash
0000000000400000 1088K r-x-- sash
0000000000710000 28K rw--- sash
0000000000717000 12K rw--- [ anon ]
000000000217e000 140K rw--- [ anon ]
00007ffc09812000 132K rw--- [ pile ]
00007ffc0989b000 12K r---- [ anon ]
00007ffc0989e000 4K r-x-- [ anon ]
ffffffffff600000 4K --x-- [ anon ]
total 1420K
```

Le noyau Linux

systèmes de fichiers - abstraction

Les fichiers et répertoires "n'existent pas" et sont **une abstraction** fournie par le noyau du système d'exploitation. **Les fichiers sont des séquences d'octets** (souvent une dizaine, parfois des milliards) **avec des méta-données**.

Conventions de nommage et d'accès (simplifiées)

- conventions des fichiers sous Linux documentées dans `hier(7)`
  - `/bin/` et `/sbin/`: executables essentiels.
  - `/usr/bin/` et `/usr/sbin/`: executables usuels.
  - `/etc/` : fichiers de configuration du système. Par exemple `/etc/passwd` décrit les utilisateurs (et parfois leur mot de passe), voir passwd(5). `/etc/X11/xorg.conf` configure le gestionnaire de fenêtres graphiques X.Org.
  - `/home/` : contient les répertoires domestiques ("home directory") de chaque utilisateur. Par exemple `/home/basile/`
  - `/dev/` : contient les pseudo-fichiers correspondant aux périphériques ("device", par exemple `/dev/cdrom` ou `/dev/zero` ou `/dev/random`).
- les **répertoires**, les **liens symboliques** sont des fichiers.

Le noyau Linux

systèmes de fichiers - conseils de nommage

Voir `path_resolution(7)` pour "comment ça marche"

- éviter les caractères bizarres, par exemple espace, point-virgule ou tabulation
- commencer et terminer un nom de fichier par une lettre ou chiffre ou blanc-souligné _
- les fichiers dont le nom commencent par un point . sont "cachés"
- les chemins . et .. désignent le répertoire courant ou parent
- l'extension (après un point) désigne par convention le type de contenu: `.html` pour du web, `.pdf` pour un document PDF, `.c` pour du code C, `.s` pour du code assembleur, `.so` pour une bibliothèque partagée ou un greffon (voir dlopen(3)), **etc...**

importance de suivre les conventions

Le noyau Linux

types de fichiers (vue statique)

Un **fichier** peut être:

- un **fichier ordinaire** ("plain file") octets) - par exemple un document LibreOffice ou un executable
- un **répertoire** ("directory") associant des noms à des fichiers
- un **lien symbolique** symlink(7)
- les liens durs ("hard links") traduisent le fait qu'un même fichier peut avoir plusieurs noms
- un **canal nommé** fifo(7)

Un fichier peut ne pas avoir de nom ou en avoir plusieurs. Le plus souvent, un fichier a un nom.
Utiliser les commandes ls(1), stat(1), find(1), df(1), du(1) pour interroger un système Linux sur ses fichiers, et dans un programme les appels systèmes stat(2) et les fonctions opendir(3), readdir(3), nftw(3) pour interroger un répertoire.

Le noyau Linux

systèmes de fichiers et montage

Un **système de fichiers** ("file system") regroupe une sous-arborescence de fichiers dans un même disque ou partition

Il existe plusieurs types de systèmes de fichiers (Ext3, Ext4, XFS, ...)
ext4(5), xfs(5)

Il existe des systèmes de fichiers distants (NFS, CIFS) nfs(5)

Linux sait **monter un système de fichiers** sur un répertoire mount(8) fstab(5)

Sur les supercalculateurs: milliards de fichiers, système de fichiers géants (peta-octets) distribués

Le noyau Linux

contrôle et droits d'accès, propriétaires et groupes

Les fichiers comme les processus ont chacun :

- **un propriétaire** ou utilisateur (désigné par un numéro, "owner user id")
- **un groupe** (désigné par un numéro, "group id")
- **des droits d'accès** pour le propriétaire, le groupe, les autres

L'utilisateur 0 est spécial: **root** et "peut tout faire"

La correspondance entre numéro d'utilisateur et son nom est décrite par le fichier `/etc/passwd` (voir passwd(5)) accessible par les fonctions getpwuid(3) et suivantes.
La correspondance entre numéro de groupe et son nom est décrite par le fichier `/etc/group` (voir group(5)) accessible par les fonctions getgrgid(3) et suivantes.

Fichiers et processus

méta-données des fichiers

Voir inode(7) pour les détails, et stat(2) pour les interroger programmatiquement.

- taille du fichier (en octets) et taille occupée en blocs
- en option date de création (en secondes par rapport à l'Epoch, début de l'année 1970)
- date de dernière modification ou changement (en secondes)
- date de dernier accès (en secondes ou mieux)
- système de fichiers et périphérique le contenant
- type de fichier (ordinaire, lien symbolique, FIFO, répertoire, etc...)
- compteur de liens (ou de noms)
- utilisateur propriétaire (user id) et groupe propriétaire (group id)
- droit d'accès (`rwx`) lecture, écriture, exécution pour l'utilisateur
- droit d'accès (`rwx`) lecture, écriture, exécution pour le groupe
- droit d'accès (`rwx`) lecture, écriture, exécution pour les autres

Fichiers et processus

exemple de méta-données d'un fichier

Exemple avec la commande stat(1) utilisant stat(2) :
`stat intro-programmation-sous-Linux.tex` produit :

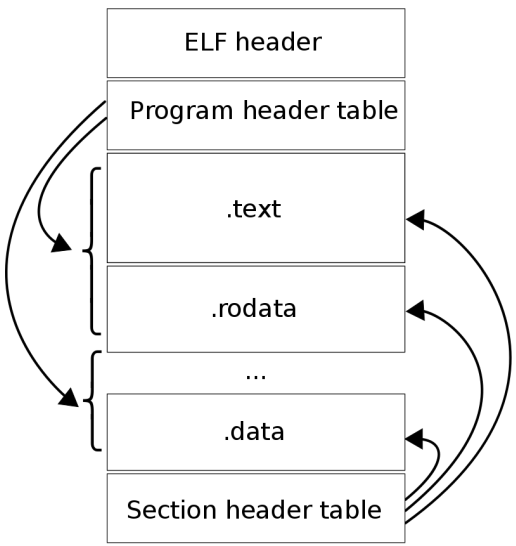
```
Fichier : intro-programmation-sous-Linux.tex
Taille : 33520 Blocs : 72 Blocs d'E/S : 4096 fichier
Périphérique: 802h/2050d Inœud: 918078 Liens: 1
Accès: (0644/-rw-r--r--) UID: (12752/basilest) GID: ( 4200/basilegr)
Accès: 2020-08-02 12:40:26.275083457 +0200
Modif.: 2020-08-02 12:40:20.153093527 +0200
Changt: 2020-08-02 12:40:20.153093527 +0200
Créé: -
```

Fichiers et processus

les fichiers exécutables

Le noyau Linux peut exécuter dans un processus avec l'appel système execve(2) (re-initialisant l'espace d'adressage et les registres de ce processus) :

- un fichier de script (lisible et executable, débutant par le shebang `#!`)
- un exécutable binaire au format elf(5) (voir wikipage Executable and Linkable Format) avec des "octets magiques" (*magic number*):



Similitude voulue avec l'espace d'adressage :

- *certaines sections* correspondent à des *segments* de l'espace d'adressage
- d'autres sections ne sont pas projetées dans la mémoire virtuelle.
  Par exemple: *informations de déboguage* au format DWARF ou table des symboles.

Fichiers et processus

Analyser ou modifier le contenu d'un fichier

De nombreuses commandes existent. On peut en créer d'autres...

- dump en octal od(1) ou en hexadécimal hd(1)
- lignes de tête head(1) ou de queue tail(1)
- concatener plusieurs fichiers cat(1)
- recherche grep(1)
- édition par flots sed(1)
- copie cp(1) , suppression rm(1), renommage mv(1)
- sur un fichier exécutable ELF :
  - commande nm(1) pour la table des noms (symboles)
  - commande readelf(1) pour extraire des informations
  - commande objdump(1) pour afficher des informations
  - commande ldd(1) pour lister les dépendances

Fichiers et processus

Les processus

Chaque processus est un programme -fichier exécutable- en train de tourner -en cours d'exécution-. Méta-données descriptives décrites dans credentials(7) et accessibles via proc(5), notamment `/proc/self/` pour le processus courant

- commandes ps(1), pstree(1) et top(1) pour les lister (via le pseudo-système de fichiers `/proc/`)
- un même exécutable (ou script) peut être exécuté par *plusieurs* processus en parallèle (exécution concurrente, en particulier l'interprète de commande bash(1) qui lance des processus à partir de commandes)
- **chaque processus a son propre espace d'adressage** virtuel modifiable par execve(2), mmap(2), munmap(2), sbrk(2), mprotect(2)
- **chaque processus a son identifiant unique numérique** (pid, "process id") accessible par getpid(2)
- **chaque processus est créé par son processus père** (sauf `/sbin/init`, de pid 1, démarré automagiquement par le noyau) dont le pid est renvoyé par getppid(2). Il peut survivre à son père.

Fichiers et processus

création et terminaison des processus

Création d'un processus: par **l'appel système fork(2)**, qui peut échouer, et quand il réussit créer par "clonage" un processus fils quasi identique au père, et **a deux résultats** : 0 dans le fils, et le pid du fils dans le père.

Terminaison d'un processus:

- **terminaison normale** explicite par `_exit(2)` ou `exit(3)` ou **fin normale du programme**
- **terminaison prématurée** en erreur par un signal (voir signal(7)) forcé par **un plantage** (ou abus de resources; cf setrlimit(2)) du programme de ce processus (division par 0, viol de l'espace d'adressage, dépassement de temps alloué, débordement d'espace disque, ...)
- **terminaison forcée** par un signal (voir signal(7)) **émis par un processus tueur** via l'appel système `kill(2)` ou la commande `kill(1)`. Les droits d'accès s'appliquent.

Fichiers et processus

attente et observation dynamique d'un processus

Attente d'un processus: l'appel système `wait(2)` attend un fils quelconque. Et `waitpid(2)` attend un fils donné (ou un groupe de fils). Un shell comme `bash(1)` lance et attend plein de fils (un par commande lancée).

Plusieurs manières d'**observer un processus** donné:

- l'appel système de bas niveau `ptrace(2)`
- la commande `strace(1)` liste les appels systèmes, et `ltrace(1)` liste les appels de fonctions de bibliothèque.
- le débogueur `gdb(1)` profite des informations de déboguage
- le chronomètre `time(1)` mesure les temps passés (voir time(7) pour les détails)
- l'utilitaire `perf(1)` analyse les performances

Fichiers et processus

table de descripteurs de fichiers

Chaque processus Linux a sa propre table de descripteurs de fichiers ouverts ("open file descriptor table", moins d'une dizaine à des milliers d'entrées), voir stdio(3). Conventionnellement souvent au moins:

0. **entrée standard** `STDIN_FILENO`, en C le canal `stdin`
1. **sortie standard** `STDOUT_FILENO`, en C le canal `stdout`
2. **message d'erreurs standard** `STDERR_FILENO`, en C le canal `stderr`

Certains processus Linux (notamment les logiciels serveurs Web) peuvent avoir des dizaines de milliers de descripteurs de fichiers (une par connection TCP/IP). On peut en limiter le nombre par setrlimit(2) et ces limites sont héritées par le processus fils créé par fork(2). Utiliser `/proc/self/fd/` pour consulter cette table comme un pseudo-répertoire.

Fichiers et processus

modification de la table de descripteurs de fichiers

Les appels systèmes suivants modifient cette table (copiée par fork(2)) :

- open(2) pour **ouvrir un fichier** existant, donné par son chemin (absolu, par exemple `/etc/fstab` ou relatif comme `../rps_manifest.json` ou `doc/garbage-collection.md`) et creat(2) pour **créer un nouveau fichier**
- close(2) pour **fermer un descripteur de fichiers**
- dup2(2) pour **dupliquer un descripteur** de fichiers
- pipe(2) pour **créer les deux bouts d'un tube de communication** (cf pipe(7))
- socket(2), accept(2), connect(2) pour **les connections réseau** (par exemple en tcp(7))
- **etc...**

deuxième jour

****

Introduction

Couches logicielles
- généralités sur le logiciel
- architecture des microprocesseurs
- rôles et traits du noyau Linux
- fichiers et processus

Outils et langages des développeurs Linux
- outillage
- langages et paradigmes de programmation
- bases de données
- aspects algorithmiques

Exercice de programmation

Fichiers et processus

chemins relatif et absolu d'un fichier pour un processus

Voir path_resolution(7)

Si un chemin ou **nom de fichier commence par** /, il est **absolu**

Autrement il est **relatif au répertoire courant**

Dans un chemin, les **répertoires sont séparés** par /

Le **répertoire parent** (parent directory) est .. et le **répertoire courant** est .

Donc `/home/../tmp/foo` est compris comme `/tmp/foo`

Voir fonction realpath(3) et commande realpath(1)

Fichiers et processus

contexte d'exécution d'un processus: répertoire courant, etc...

Voir credentials(7), namespaces(7), capabilities(7) pid_namespaces(7)
Chaque processus a (cf fork(2)):

- **son répertoire courant** (current working directory) (cf getcwd(2) et chdir(2) pour le changer)
- **son espace d'adressage** dont la pile d'appel et l'environnement
- **sa table de descripteur de fichiers**
- **etc...** (par exemple, traitement des signaux, limites de resources, son répertoire racine chroot(2))

Fichiers et processus

démarrage d'un programme dans un processus par `execve`

le *seul* moyen de démarrer un programme Linux, c'est l'appel système **execve(2)** qui "initialise" le processus courant pour un nouveau programme, ou les fonctions qui l'utilisent: exec(3)
Cet execve(2) prend comme arguments:

- le chemin du fichier exécutable (ELF ou script) à exécuter `/bin/ls`
- le tableau des arguments (chaînes de caractères comme `ls` `-l` `mon-fichier.c` par exemple)
- l'environnement (tableau de chaînes de caractères comme `HOME=/home/basile` `PATH=.:/bin:/usr/bin`...)

Le tableau d'arguments est connu du `main` du nouveau programme exécuté.
L'environnement aussi (voir environ(7) et getenv(3), etc...). Les arguments et les environnements sont dans la pile d'appel du nouvel espace d'addressage.
Donc **execve(2) est coûteux** (milliseconde parfois).

Fichiers et processus

mon environnement comme exemple

La commande printenv(1) me donne (extrait *incomplet*...) :

```
SSH_AUTH_SOCK=/tmp/ssh-AokrAslPJO7B/agent.3033
USER=basilest
HOME=/home/basilest
PWD=/home/basilest/introprog-montpellibre-2020
LANGUAGE=fr_FR:en
LANG=fr_FR.UTF-8
LC_IDENTIFICATION=fr_FR.UTF-8
COLORTERM=truecolor
TERM=xterm-256color
LC_NUMERIC=fr_FR.UTF-8
LC_MEASUREMENT=fr_FR.UTF-8
LC_TIME=fr_FR.UTF-8
LC_PAPER=fr_FR.UTF-8
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/bin
LC_MONETARY=fr_FR.UTF-8
SHELL=/bin/zsh
LC_TELEPHONE=fr_FR.UTF-8
XAUTHORITY=/home/basilest/.Xauthority
LC_NAME=fr_FR.UTF-8
DISPLAY=:0.0
LC_ADDRESS=fr_FR.UTF-8
PS1=%m %3~ %T %#
```

Fichiers et processus

contexte d'exécution d'un processus: conventions et combinaisons en pipeline de programmes

Les descripteurs de fichiers usuels (entrée standard et sortie standard) permettent conventionnellement de **combiner des programmes simples** (par exemple ls(1) listant des fichiers, grep(1) filtrant des motifs textuels, wc(1) comptant les mots ou les lignes) **en un pipeline** ou tube de commandes, séparées par un pipe | (voir aussi pipe(7)).
La sortie standard de la commande de gauche est reliée à l'entrée de la commande de droite.

```
ls -1 | grep powerpc | wc -l
```

va compter les fichiers dont le nom contient `powerpc` donc afficher `14` sur la sortie standard.

Chaque "pipe" contient un tampon de données géré par le noyau (avec **synchronisation des processus** communiquants).

Fichiers et processus

conventions d'écriture des programmes

Il est de bon ton de suivre les usages ci-dessus:

- **favoriser les échanges entre programmes** par des formats ouverts, **documentés**, familiers et **textuels** (voir aussi JSON, XML, YAML et les protocoles de communication comme HTTP ou SMTP ou CUPS ...)
- préférer ou permettre la **lecture des données d'entrée sur l'entrée standard** (souvent notée `-` ou `/dev/stdin` dans les arguments du programme), et l'**écriture des résultats de sortie sur la sortie standard** (souvent passée comme `-` ou `/dev/stdout` en argument de programme)
- **un programme qui fonctionne bien reste silencieux** et se termine en faisant un `exit(EXIT_SUCCESS)` où `EXIT_SUCCESS` est 0.
- en cas d'erreur ou anomalie, produire un message d'erreur sur la sortie d'erreur standard
- **un programme échoue** avec au moins un `exit(EXIT_FAILURE)` où `EXIT_FAILURE` est 1, ou tout autre code d'erreur *documenté* > 0 et < 127
- **tous les programmes acceptent les arguments** `--help` (donnant un message d'aide) et `--version` (donnant leur version)
- s'appuyer sur et **s'inspirer des commandes existantes** donc avec des noms d'option *en anglais*

Fichiers et processus

pour en savoir plus

- les pages de `man` en `man7.org/linux/man-pages/`
- pour **les appels systèmes de Linux**, voir syscalls(2) et y suivre les hyperliens
- pour **les fonctions en C**, voir intro(3)
- pour les cas d'erreur errno(3). Essayez de les traiter.
- sur les systèmes d'exploitation, *Operating Systems - Three Easy Pieces* et OSDEV
- livre en ligne *Advanced Linux Programming*
- **exemples de code open source** sur `github.com` et `gitlab.com` **etc...**.
- **forums** en ligne `stackoverflow.com`, developpez.com et sites `debian.org` et `ubuntu.com` et `framalibre.org` et *mailing lists*
- le World Wide Web tout entier, notamment `www.gnu.org`, `savannah.gnu.org`, `kernel.org`, `kernelnewbies.org`
- **le logiciel libre *n'est pas* gratuit** (faits par des esclaves) donc **contribuez y** (au moins par de bons rapports de bogue)

Fichiers et processus

le langage de commande "Unix shell"

- c'est **un langage standardisé** par POSIX (portable operating system interface)
- il y a plusieurs interprètes de commandes en logiciel libre: GNU `bash` ou `zsh` ou es ou `sash` et il faut **lire leur documentation**
- variables indiquées par un signe `$`, exemple `echo $PATH`
- répertoire domestique de l'utilisateur noté `~`
- changement de répertoire courant par `cd` (qui ne peut pas être un programme) par exemple `cd ~/intoprog-montpellibre-2020`
- le shell remplace par expansion (voir glob(7) ...) un mot par plusieurs, selon les fichiers existants. Par exemple:
  - `ls a*.c` liste tous les nom de fichiers commençant par `a` et se terminant par `.c`
  - `rm [a-h]*.o` supprime (par l'appel système unlink(2)) tous les fichiers (conventionnellement des fichiers objets) dont un nom commence par une lettre `a`, `b`, ... `h` et se terminant par `.o`
- la commande echo(1) affiche ses arguments
- la commande file(1) analyse le contenu d'un fichier

Sommaire

Introduction

Couches logicielles

Outils et langages des développeurs Linux
- outillage
- langages et paradigmes de programmation
- bases de données
- aspects algorithmiques

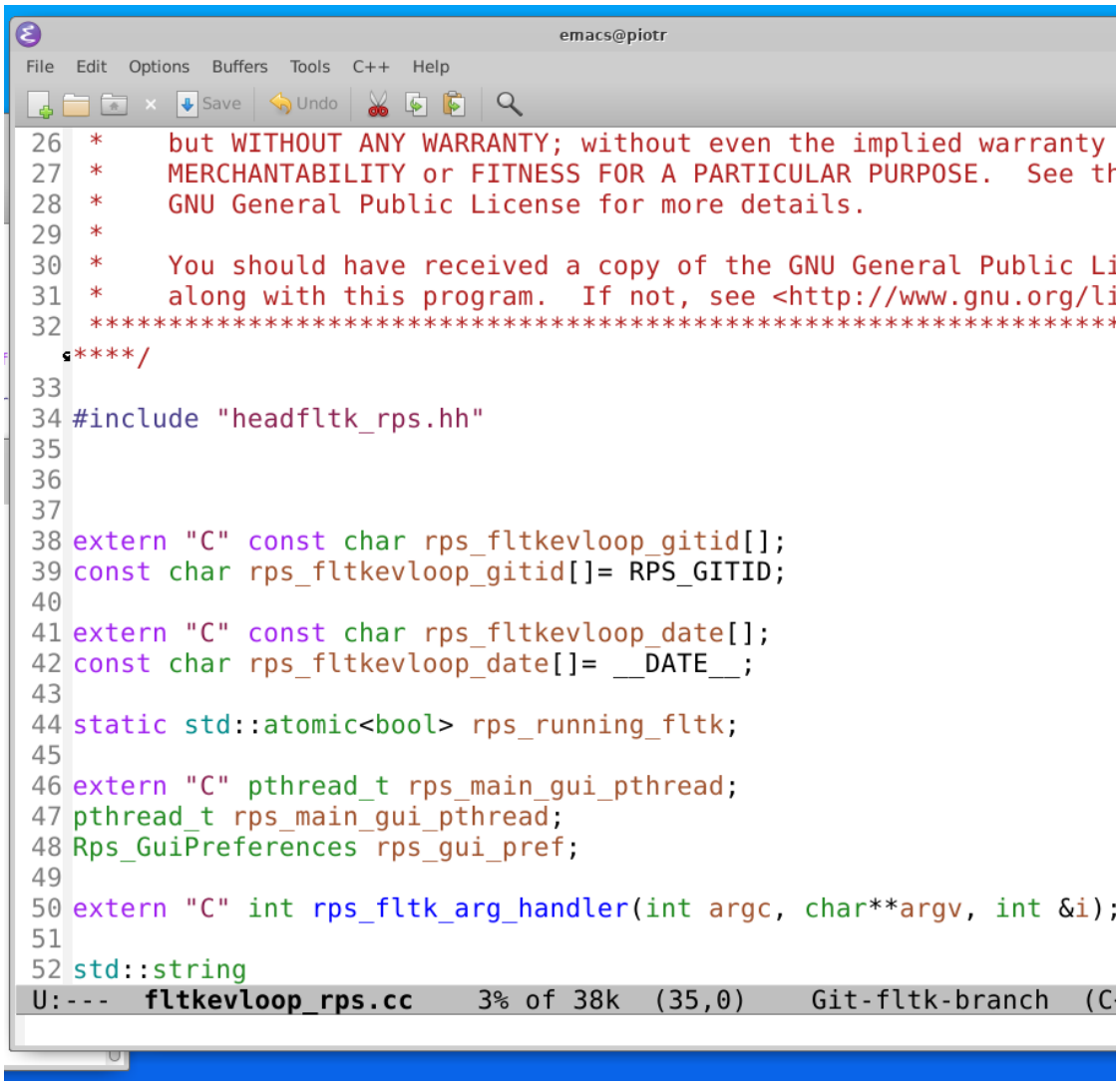
Exercice de programmation

Outils des développeurs Linux

- **éditeur de code source** comme GNU `emacs`, `vim`, `gedit` pour créer et modifier les fichiers de code sources
- **logiciel de gestion de versions** comme `git` pour gérer efficacement différentes versions d'un code source
- **forges logicielles** par exemple `framagit`
- **générateurs de code** comme GNU `bison` (génération d'analyse syntaxique) ou SWIG (génération de code glue)
- **moteur de production** par exemple GNU `make` ou `ninja` coordonnant des outils plus élémentaires.
- **interpréteurs** (exécutant du code source) comme `bash(1)` ou awk ou GNU Guile
- **compilateurs** (transformant du code source) comme GCC ou Ocaml
- **formatteurs de documents** comme LaTeX ou Lout et **générateurs de documentation** comme Doxygen
- assembleurs et éditeurs de liens notamment GNU `binutils`
- **bibliothèques logicielles** comme Qt ou Ncurses ou GMPLIB etc...
- **débogueur** comme `gdb(1)` ou Valgrind

Outils des développeurs Linux

emacs sur du code source C++ de `refpersys.org`



Langages des développeurs Linux

la factorielle comme exemple

La factorielle de n noté $n!$ est le nombre de permutations sur n objets. On a $n! = 1 \times 2 \times \ldots \times (n-1) \times n$ et la formule de récurrence $n! = n \times (n-1)!$ dès que $n > 1$ et bien sûr $1! = 1$

La factorielle compte le nombre de chemins circulaires entre villes. Pour cinq villes (Annecy, Berlin, Casablanca, Dijon, Erlangen) il y a $5! = 120$ circuits possibles. Pour treize villes, il y en a $13! = 6227020800$ (plus de six milliards)

Lire aussi sur l'**explosion combinatoire** et le problème du voyageur de commerce (donc l'optimisation des déplacements d'un livreur).

Langages des développeurs Linux

coloriage de la factorielle

Le code source contient (dans les exemples ci-dessous) :

- des **mots-clés** du langage de programmation, en rouge pour les mots-clés.
  (certains langages de programmation -PL/1 ou Common Lisp- n'ont théoriquement pas de mots-clés *réservés*, mais pragmatiquement ils en ont)
- des **commentaires** pour les dévelopeurs humains, en vert pour les commentaires
- des **définitions** de nouveaux noms, en vert olive pour les noms nouvellement définis
- les **litéraux** (chaînes, nombres) pourraient mériter une couleur ("syntax coloring")

En 2020, **un bon éditeur de code source va coloriser votre code**.

Langages des développeurs Linux

factorielle en Ocaml

En langage Ocaml, peut-être dans un fichier `fact.ml` :

```
(*fonction factorielle en Ocaml, récursivement*)
let rec fact n =
      if n > 1 then n * fact (n - 1)
      else 1
;;
```

variante codée itérativement avec un ou deux sites d'appel de fonction en récursion terminale en brun:

```
(*factorielle en Ocaml, itérativement*)
let fact n =
      let rec factloop n p =
         if n > 0
         then factloop (n - 1) (n * p)
         else p
      in factloop n 1
;;
```

Langages des développeurs Linux

factorielle en *C* - récursivement

En langage C, voir le livre *Modern C*, des cours comme *Apprenez à programmer en C*, des sites tels que `cppreference.com` (pour C et C++), le standard *C11* en n1570 (sur 700 pages); dans un fichier source `fact.c` :

Forme récursive:

```
/*fonction factorielle en C*/
extern long fact(int); //déclaration
//variante récursive
long fact(int n) {
  //retour avec expression conditionnelle
    return
       (n > 1) ?
       n*fact(n-1) :
       1;
}
```

Langages des développeurs Linux

factorielle en *C* - boucle itérative

```
/*fonction factorielle en C*/
extern long fact(int); //déclaration
//variante itérative en boucle
long fact(int n) {
    long res = 1;
    while (n > 0) {
        res = res * n;
        n = n - 1;
    };
    return res;
}
```

Notez les instructions d'**affectation** ("assignment statements", car mathématiquement $n = n - 1$ est impossible) qui décrémente le compteur `n`. On pourrait écrire `n--;` au lieu de `n = n-1;` (et si besoin `p++;` au lieu de `p=p+1;`)

Langages des développeurs Linux

factorielle en *C* - itération avec saut (années 1980)

```
/*fonction factorielle en C*/
extern long fact(int); //déclaration
//variante itérative avec saut
long fact(int n) {
   long res = 1;
  etiq:
   if (n <= 0)   return res;
    res = res * n;
    n = n - 1;
   goto etiq; //saut en arrière!
}
```

Mais les sauts par goto rendent le code source illisible (code spaghetti), fameux papier *Go To Statement Considered Harmful* (Dijkstra, 1968)
Les deux variantes itératives pourraient être compilées ver le même code machine!

Langages des développeurs Linux

factorielle en *Scheme*

Scheme (1975) est un **très beau langage de programmation minimaliste** inspiré de Lisp (1958). **Tout est expression**, et toute expression composite se code **entre parenthèses** *toujours significatives* : (*opérateur* $opérande_1$ ... $opérande_n$). Les noms permis en Scheme sont bizarres: *letrec-syntax* est un seul nom (un quasi- mot-clé introduisant temporairement vos propres nouveaux mots-clés).

GNU Guile est une implémentation de Scheme défini dans R5RS (en seulement 50 pages). Bigloo en est une autre (compilateur de Scheme vers C). Voir ce tutoriel d'initiation

```
;; fonction factorielle en Scheme, récursive
(define (fact n)
  (if (< n 0) 1
   (* n (fact (- n 1)))))
```

À lire absoluement le fameux et fabuleux SICP (*"structure and interpretation of computer programs"*) **en ligne** (au MIT, centaines de pages)

Langages des développeurs Linux

d'une fonction à un programme

Les exemples de code précédents ne font rien et sont inutiles car incomplets. Il manque une spécification et le code glue pour appeler les fonctions définies.

On veut donc écrire un programme (exécutable) `impr-fact` qui imprime la factorielle de plusieurs arguments du programme. Par exemple `impr-fact 5 10` doit calculer et imprimer 5! et 10! donc afficher 120 suivi de 3628800

Le code source d'intendance sera plus gros en taille que le code source de la factorielle.

Langages des développeurs Linux

intendance de la factorielle en Ocaml

Lire la documentation d'Ocaml puis y utiliser le module `Arg` pour le traitement des arguments du programme, et la fonction standard `int_of_string` pour convertir une chaîne en un entier.
On utilise un style de programmation fonctionnelle car notre fonction `traiter_arg` est une valeur (lire sur Wikipedia à propos des fermetures) passée à `Arg.parse`.... Code non testé dans `intendance.ml` !

```
let imprime_fact n = Printf.printf "%d\n" (fact n) in
let traiter_arg a = imprime_fact (int_of_string a) in
Arg.parse [] traiter_arg
    "imprime la factorielle de chaque argument";;
```

On compilerait avec la commande
`ocamlopt fact.ml intendance.ml -g -o impr-fact`

Langages des développeurs Linux

intendance de la factorielle en *C*

Il faudrait lire le chapitre parsing program arguments pour fournir le traitement convenable d' un `--help` et `--version`. Sans ça (voir atoi(3) et printf(3)) dans un fichier `intendance.c` contenant :

```
/* fichier intendance.c */
#include <stdio.h> /* pour printf */
#include <stdlib.h> /* pour atoi */

extern long fact (int);
int main(int argc, char**argv) {
  for (int numarg = 1; numarg<argc; numarg++) {
    int n = atoi(argv[numarg]);
    printf("%ld\n", fact(n));
  }
  exit(EXIT_SUCCESS);
}
/* fin du fichier intendance.c */
```

Langages des développeurs Linux

compilation de la factorielle en *C*

Il faudrait avoir un `Makefile` (où la tabulation est significative); lire la documentation de GNU `make` et lancer `make -p` pour en comprendre les règles prédéfinies.

En attendant, on peut compiler et relier (par le "linker" `ld` de GNU *binutils*) les deux unités de compilation ("translation units") `fact.c` et `intendance.c` avec gcc(1) par la commande:
`gcc -Wall -Wextra -O -g fact.c intendance.c -o impr-fact`
(qui lance des processus `cc1`, `as`, `ld`).

Les options `-Wall -Wextra` demande les très utiles avertissements ("warnings") du compilateur, et l'option `-g` les informations de déboguage au format DWARF au profit du débogueur gdb(1). L'option `-O` (ou `-O2` ou `-O3` ou `-Os`) déclenche les optimisations du compilateur (code généré plus rapide, temps de compilation plus grand). L'ordre des arguments à gcc(1) est important.

Langages des développeurs Linux

examen du code assembleur produit par gcc

La commande `gcc -O -fverbose-asm -S fact.c` produit le fichier assembleur `fact.s` obtenu de `fact.c` qu'on peut ensuite examiner avec un éditeur tel que `emacs`.

On y devine les conventions d'appel (calling convention) du x86-64 sous Linux, documentées dans wikipedia x86 calling conventions et surtout System V Application Binary Interface AMD64 Architecture Processor Supplement

On peut obtenir (par exemple avec `-fdump-tree-all`...) les nombreuses **représentations intermédiaires du compilateur** GCC notamment GIMPLE

Langages des développeurs Linux

pièges de la programmation en *C* sous Linux

Ils sont nombreux, lire attentivement la documentation de GCC:

- l'**allocation de la mémoire**
- les **dépassements de tampon** ("buffer overflow")
- les **fuites de mémoire** - utiliser `valgrind` et *Address Sanitizer*
- les **erreur de segmentation** et autres atteintes à l'espace d'addressage
- le traitements des signaux (voir signal(7) et signal-safety(7) etc...)
- les appels systèmes qui échouent (voir aussi errno(3) et perror(3))
- les **comportements indéfinis** : lire *What Every C Programmer Should Know About Undefined Behavior*

Voir aussi (en 2020) mon brouillon de rapport sur Bismon.
Compiler avec `gcc -Wall -Wextra -g` et utiliser en 2020 un GCC récent (au moins GCC 10), peut-être avec votre greffon de GCC.
Envisager l'utilisation d'outils comme Frama-C ou Clang static analyzer.

Langages des développeurs Linux

avantages de la programmation en *C* sous Linux

C est un "langage assembleur portable". Nombreux avantages aussi:

- **foultitude d'exemples**; penser à **la métaprogrammation** (vos générateurs de code C spécialisés)
- qualité des compilateurs (essayer GCC **et** Clang et lire leur documentation) et leurs facultés d' optimisations
- qualité et quantité des bibliothèques: Program Library HowTo
- bibliothèques partagées *How To Write Shared Libraries* donc vos greffons grâce à dlopen(3) et dlsym(3)
- débogueur GDB avec:
  - points d'arrêts breakpoints
  - arrêt sur changement de mémoire watchpoints
  - scriptable en Guile et en Python
- **génération de code à la volée**, par exemple avec libgccjit

Langages des développeurs Linux

Programmation en *C++*, *Rust*, *OpenCL* sous Linux

C++ (lire Programming – Principles and Practice Using C++ puis le site C++ reference puis la norme C++11 n3337) est un langage difficile à apprendre, peut-être obsolescent, mais compatible avec C.

- language complexe et difficile à apprendre
- bibliothèque standard riche, notamment par ses containers
- pleins de bibliothèques sous Linux, dont Qt, FLTK (interface graphique), Wt (web), Vmime (courriel), Poco (internet et réseau), SFML (jeu et multimédia)
- de gros logiciels libres sont codés en C++: LibreOffice (bureautique), Mozilla Firefox (navigateur Web)

Rust est à préférer à C++ pour de nouveaux développements dont le code est proche du matériel ou doit être efficace ou robuste.
OpenCL ressemble au C mais tourne vectoriellement sur GPU (cartes graphiques). Pour le calcul vectoriel ou matriciel intensif.

Langages des développeurs Linux

Programmation en *Ocaml*, *Haskell*, *Go*, *Common Lisp* etc... sous Linux

- OCAML est fonctionnel, fortement typé, vivant. Utilisé dans le démonstrateur de théorèmes / assistant de preuves Coq. Faiblesse en 2020: support du multi-threading insuffisant.
- Haskell est fonctionnel, puissant et "mathématique" car **déclaratif**, paresseux et efficace.
- Go pourrait remplacer Python (quasi mort, moche) ou PHP (dinosaure, enterré).
- Common Lisp est très puissant, ressemble à *Scheme*, homo-iconique donc favorisant la métaprogrammation. SBCL est une excellente implémentation.
- les ancêtres **Python, TCL, shells, Perl, PHP** ne devraient être considérés que pour les vieux logiciels libres. **à proscrire** sauf si on est obligé.

Langages des développeurs Linux

paradigmes de programmation

différents **paradigmes de programmation**:

- programmation impérative (C, Rust, Ocaml)
- programmation orientée objet (C++, Java, JavaScript, Go, Ocaml, Lua, Io...) SmallTalk, *Metaclasses are First Class: the ObjVLisp model*
- programmation fonctionnelle (Ocaml, Scheme, Lisp, Haskell) et λ-calcul
- programmation logique avec des prédicats (Prolog....) et programmation par contraintes (ECLiPSe...)
- programmation concurrente -multicœeur- Pthreads en C, *Go* et π-calcul
- Programmation événementielle, notamment les serveurs et les interfaces graphiques, avec une boucle d'événements centrée sur poll(2) ou mieux. Voir Ocsigen pour coder des applications Web.

Langages des développeurs Linux

pour en savoir plus sur les langages

- analyse lexicale, syntaxe des langages de programmation et sa notation en Extended Backus-Naur Form (EBNF), analyse syntaxique, abstract syntax tree
- sémantique des langages de programmation, sémantique dénotationnelle et théorie des types, inférence de types
- survol des langages: *Scott: Programming Language Pragmatics*
- techniques de compilation: Dragon Book - indispensables à connaître, car elles sont partout (i.e. recursive descent parsing JSON ou XML ou HTML ou HTTP). Générateurs d'analyseurs syntaxiques: GNU BISON et ANTLR
- gestion de la mémoire : ramasse-miettes ("garbage collector" ou "glaneur de cellules" donc Garbage Collection Handbook
- l'excellent livre de Christian Queinnec : *Principes d'implantation de Scheme et Lisp*

Langage des développeurs Linux

les bases de données

Il faut faire attention à bien stocker les données sur le disque et à la sauvegarder.

Une base de données organise et traite les données applicatives en rapport avec un problème. Elles sont nécessaires car à l'arrêt d'un PC Linux l'information en RAM est perdue, et quand on veut traiter une grosse quantité de données (gigaoctets)
Les systèmes de gestion de base de données (SGBD) sont de plusieurs sortes:

- base de données relationnelles : serveur PostGreSQL, bibliothèque Sqlite utilisant le langage Structured Query Language (SQL)
- base de données destructurées ou indexées : serveur REDIS, bibliothèque GDBM etc
- base de données "document" comme MongoDB ou Cassandra etc...
- requêtes et transactions vers une base de données

Penser à la sauvegarde!

Langage des développeurs Linux

bases de données relationnelles

On représente dans quelques tables des relations n-aires. On fait attention à la normalisation de la base de données pour éviter les incohérences et les redondances. Une table peut avoir des dizaines de colonnes et des millions de rangées.
Voir ce cours et tutoriel sur SQL
On crée une table par une requête SQL décrivant les colonnes telle que:

```
CREATE TABLE table_etudiants (
    id_etudiant SERIAL,
    nom_etudiant VARCHAR(40) NOT NULL,
    prenom_etudiant VARCHAR(20),
    naissance_etudiant DATE
);
```

On la remplit par des milliers de requêtes d'insertion telles que:

```
INSERT INTO table_etudiants
    (nom_etudiant, prenom_etudiant, naissance_etudiant)
VALUES ("Dupont", "Jean", "1995-10-03");
```

Langage des développeurs Linux

bases de données relationnelles (interrogation)

On peut ensuite interroger la base, par exemple

```
SELECT id_etudiant, prenom_etudiant FROM table_etudiants
    WHERE nom_etudiant = "Dupont" ;
```

pour obtenir l'identifiant et le prénom de tous les étudiants qui s'appellent `Dupont`

Pour des raisons d'efficacité, il faut **créer des index de table** et des **requêtes préparées** ("prepared statements")

On peut faire des jointures entre tables (composer les relations entre tables) et c'est toute la puissance du modèle relationnel.

Langage des développeurs Linux

aspects algorithmiques

Le livre ***Introduction to Algorithms*** **est à lire absoluement**

- les types abstraits et les structures de données
- la recherche dichotomique - deviner en 20 questions oui/non un mot dans un dictionnaire
- les arbres binaires de recherche notamment arbre bicolore rouge-noir
- les table de hachage
- etc.

Sommaire

Introduction

Couches logicielles

Outils et langages des développeurs Linux

Exercice de programmation

Exercice de programmation

problème posé

Exemples: compter le nombre de mots dans plusieurs fichiers, et de plus en calculer la fréquence