



**Programmation
système en langage C
sous LINUX**

DRAFT



Seconde édition
(en cours de rédaction)

Philippe PINCHON

(cc) 2016 - Ce document est mise à disposition selon les termes de la
[Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage à l'Identique 3.0 non transposé.](#)

Introduction



- Déroulé du cours
- Tour de table

Déroulé du cours

(1/5)

- ◆ Module 1 – Langage C, outils & système Linux
 - ◆ Rappels sur le langage C
 - ◆ Outils pour développer
 - ◆ Système d'exploitation Linux
- ◆ Module 2 – Processus, threads et ordonnancement
 - ◆ Notion de processus
 - ◆ Threads
 - ◆ Ordonnancement

Déroulé du cours

(2/5)

- ◆ Module 3 – Système de gestion de fichiers
 - ◆ Notions générales
 - ◆ Le système de gestion de fichiers virtuel VFS
 - ◆ Le système de fichiers /proc
- ◆ Module 4 – Gestion des Entrées/Sorties
 - ◆ Principes généraux
 - ◆ E/S Linux

Déroulé du cours

(3/5)

- ◆ Module 5 – Gestion de la mémoire centrale
 - ◆ Les mécanismes de pagination et de mémoire virtuelle
 - ◆ La gestion de la mémoire centrale sous Linux
- ◆ Module 6 – Gestion des signaux
 - ◆ Présentation générale
 - ◆ Aspect du traitement des signaux par le noyau
 - ◆ Programmation des signaux
 - ◆ Signaux temps réel

Déroulé du cours

(4/5)

- ❖ Module 7 – Communications entre processus
 - ❖ Communication par tubes
 - ❖ IPC (inter Processus Communication)



Tour de table

- ◆ Nom et prénom
- ◆ Fonctions au sein de votre entreprise
- ◆ Connaissances et compétences du sujet abordé
- ◆ Attentes

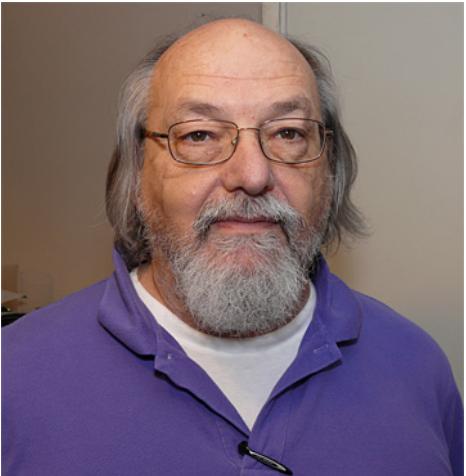
Module 1

Langage C, outils & système Linux

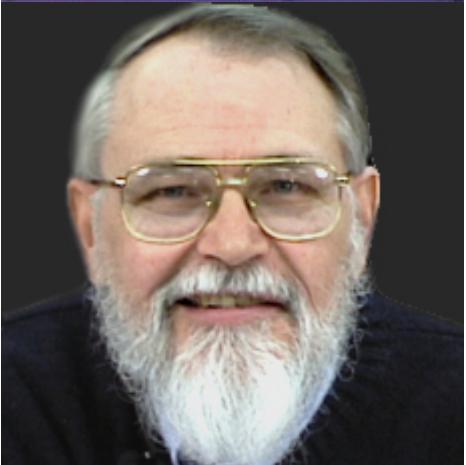
Module 1 - Outils & langage C

- ◆ Langage C
 - ◆ Rappels
- ◆ Outils pour développer
 - ◆ GNU Compiler Collection (GCC)
 - ◆ Compilation avec GCC
 - ◆ 4 étapes de la compilation
 - ◆ Débogage
- ◆ Système d'exploitation
 - ◆ Introduction
 - ◆ Structure générale
- ◆ Linux
 - ◆ Architecture
 - ◆ Modes d'exécutions
 - ◆ Commutations de contexte
 - ◆ Gestion des interruptions matérielles et logicielles

L A N G A G E C



Kenneth Lane Thompson (1943)
aka Ken Thompson



Brian Kernighan (1942)

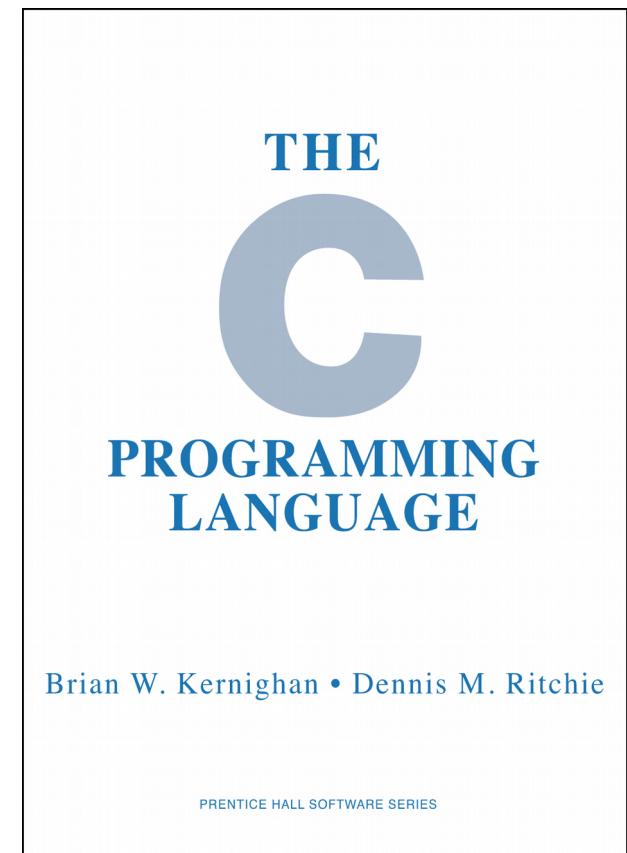


Dennis MacAlistair Ritchie (1941 - 2011)
aka Denis Ritchie

Langage C

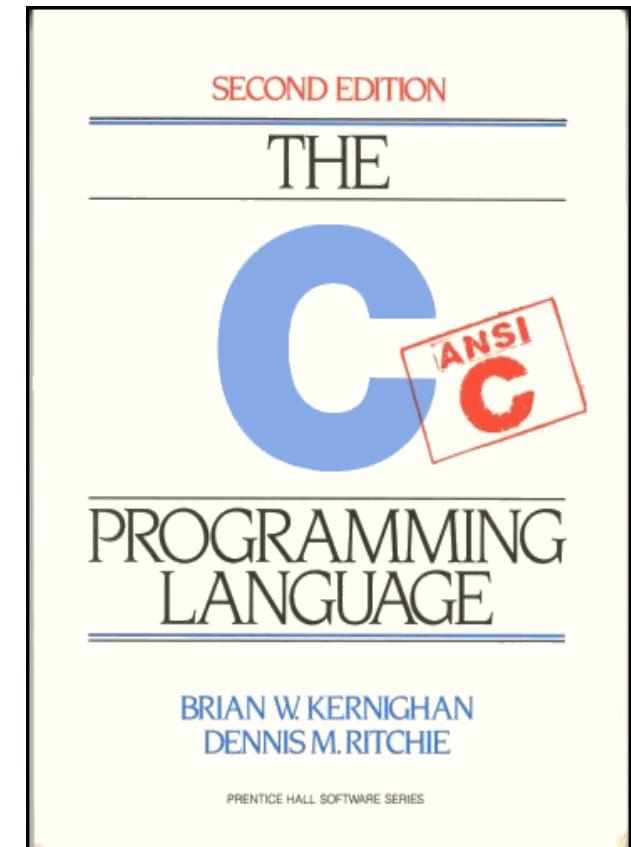
C traditionnel ou K&R C

- Le langage C, conçu en 1972 par Dennis Richie et Ken Thompson (Bell Labs), est un langage procédural afin de développer le système UNIX sur un DEC PDP-11.
- En 1978, Brian Kernighan et Dennis Richie publient la définition classique du C dans le livre « The C Programming language » connu sous le nom « C traditionnel » ou « K&R C ».



Langage C ANSI C

- ◆ De plus en plus populaire dans les années 80, plusieurs groupes mirent sur le marché des compilateurs C comportant des extensions particulières.
- ◆ En 1983, l'ANSI décida de normaliser le langage. Le projet s'acheva en 1989 : « ANSI C ».
- ◆ Celle-ci fut reprise telle quelle par l'ISO en 1990 : « C90 » ou « C ISO » (ISO/CEI 9899:1990).



Langage C

C89 ou C94/95

- ♦ Entre 1994 et 1996, le groupe de travail de l'ISO (ISO/IEC JTC1/SC22/WG14) a publié 2 correctifs et 1 amendement à C90 :
 - ♦ ISO/IEC 9899/COR1:1994 Technical Corrigendum 1,
 - ♦ ISO/IEC 9899/AMD1:1995 Intégrité de C,
 - ♦ ISO/IEC 9899/COR1:1996 Technical Corrigendum 2.
- ♦ Ces changements assez modestes sont parfois appelés C89 avec amendement 1, ou C94/C95.

Langage C

Mots réservés C89

auto	Déclare 1 variable locale non-statique qui est implicitement non statique.
break	Sort d'une boucle ou saute certains cas non utiles d'un programme.
case	Vérifie l'expression envoyé la directive switch.
char	Type de données dont la valeur s'étend de -128 à 127.
const	Déclare une valeur constante.
continue	Recommence depuis le début d'une boucle sans la terminer.
default	Appelle les instructions à effectuer par défaut dans une directive switch.
do	Met en place une boucle de type do...while.
double	Type de données dont la valeur s'étend de -1.7*10^-308 à 1.7*10^308.
else	Traite les instructions qui lui sont rattachées si la condition n'est pas validée du if.
enum	Crée une énumération.
extern	Déclare des variables globales ou des fx définies dans d'autres fx du programme.
float	Type de variables pour des nombres à virgule flottante en simple précision. Plage de valeurs : -3.4*10^-38 à 3.4*10^38
for	Boucle
goto	Permet au programme de continuer son exécution à partir d'un label invoqué.

Langage C

Mots réservés C89

if	Instruction conditionnelle
int	Type de variable pour des valeurs entières comprises entre -32 768 et 32 767.
long	Type de variable pour des valeurs entières comprises entre -2 147 483 648 et 2 147 483 647.
register	Signale au compilateur que la variable sera stockée dans le registre du processeur et non dans la mémoire vive.
return	Termine une fonction avec ou on un renvoi d'une valeur.
short	Type de variable pour des valeurs entières comprises entre -32 768 et 32 767.
signed	Une variable peut être négative ou positive. Par défaut les variables sont signed.
sizeof	Retourne la taille en octet occupée par une variable dans la mémoire vive.
static	Déclare une variable globale dont la portée ne sera que dans le fichier où elle est déclarée.
struct	Regroupe plusieurs variables dans une structure.
switch	Fait plusieurs tests de valeurs sur le contenu d'une même variable.
typedef	Renommer un type de variable pour une meilleure compréhension du code.
union	Rassemble plusieurs variables (un peu comme avec les structures). Cependant, on ne peut en utiliser qu'une seule. La taille de l'union en mémoire vaut la taille en mémoire du type qui prend le plus de place.

Langage C

Mots réservés C89

unsigned	Signale au compilateur que la variable ne sera pas négative. Sa valeur sera donc comprise entre 0 et la taille maximale du type qui lui est associé.
void	Déclare qu'une fonction ne renverra aucune valeur ou ne prendra aucun arguments.
volatile	Déclare une variable qui pourra être modifiée par des éléments externes au programme (comme le système d'exploitation).
while	Boucle. Les instructions sont traitées tant qu'une condition est vérifiée.

Langage C

C99

- En 1999, une nouvelle évolution du langage est normalisée par l'ISO : C99 (ISO/CEI 9899:1999). Il ajoute 5 mots réservés :

_Bool	Type de variable booléenne pouvant être 0 ou 1 (défini dans stdbool.h).
_Complex	Type de variables d'un nombre complexe. (défini dans complex.h).
_Imaginary	Type de variables d'un nombre imaginaire (défini dans complex.h).
inline	Indique qu'il faut que le compilateur élargisse la fonction en ligne au moment de l'appel de la fonction ou d'un membre de fonctions
restrict	Indique une déclaration de fonction ou de définition n'envoyant pas un type de format pointeur. Le compilateur retournera un objet n'étant pas un alias mais avec tous les pointeurs des autres.

Langage C

C11

- ◆ L'ISO ratifie en 2011 la norme C11 (ISO/IEC 9899:2011) et ajoute 7 mots réservés :

_Alignas	Le spécificateur de alignas (défini dans stdalign.h) ne peut être utilisé lors de la déclaration des objets qui ne sont pas des champs de bits et n'ont pas la classe de stockage de registre. Il ne peut pas être utilisé dans les déclarations de paramètres de fonctions et ne peut être utilisé dans un typedef.
_Alignof	Récupère l'alignement (défini dans stdalign.h).
_Atomic	(Défini dans stdatomic.h).
_Generic	--
_Noreturn	(Défini dans stdnoreturn.h).
_Static_assert	(Défini dans assert.h).
_Thread_local	(Défini dans threads.h)

Langage C (7/32)

◆ Instructions du préprocesseur :

#include <stdio.h> #include « /prj/src/monfic.h »	Permet d'inclure le contenu d'un fichier. <> stipule l'inclusion d'un fichier header standard. « » stipule l'inclusion d'un fichier header personnel.
#define	Définition d'une constante ou d'une macro.
#pragma	Défini depuis C89. Directive entièrement dépendante de l'implantation.
#if #endif	Si la condition est vraie alors compiler le code de cette condition. Fin de la condition.
#ifdef #ifndef	Vérifie si une constante a été définie. Vérifie si une constante n'a pas été définie.
#elif	Défini depuis C89. Sinon si la condition est vraie alors compiler le code de cette condition.
#else	Sinon compiler ce code.
#undef	Supprime la directive spécifiée
#line	Permet de changer le numéro de ligne et le nom du fichier courant.
#error	Arrête la compilation et affiche le message d'erreur passé en argument.

Langage C

Exemple d'utilisation des instructions define et include

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

```
#define puts(s) printf("Macro : %s\n", (s))
```

```
int main(void) {
    char s[] = "Salut :-)";
    puts(s);                                /* Macro */
    (puts)(s);                               /* Fonction */
    return EXIT_SUCCESS;
}
```

Langage C

Exemple d'utilisation des instructions define et include

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

```
#define puts(s) printf("Macro : %s\n", (s))
```

```
int main(void) {
    char s[] = "Salut :-)";
    puts(s);                                /* Macro */
    (puts)(s);                               /* Fonction */
    return EXIT_SUCCESS;
}
```

Langage C (8/32)

◆ Les types entiers :

Types entiers, en ordre croissant			
Type	Taille	Magnitude <i>signed</i> (magnitude <i>minimale</i> exigée par le standard ³)	Magnitude <i>unsigned</i> (magnitude <i>minimale</i> exigée par le standard ³)
<code>char</code> , <code>unsigned char</code> , <code>signed char</code> (C89)	≥ 8 bits	-127 à 127	0 à 255 (0xFF en hexadécimal noté avec le préfixe <code>0x</code> de la syntaxe de C)
<code>short</code> (identique à <code>signed short</code>), <code>unsigned short</code>	≥ 16 bits	-32 767 à +32 767	0 à 65 535 (0xFFFF)
<code>int</code> (identique à <code>signed int</code>), <code>unsigned int</code>	≥ 16 bits (taille d'un mot machine)	-32 767 à +32 767	0 à 65 535 (0xFFFF)
<code>long</code> (identique à <code>signed long</code>), <code>unsigned long</code>	≥ 32 bits	-2 147 483 647 à +2 147 483 647	0 à 4 294 967 295 (0xFFFFFFFF)
<code>long long</code> (identique à <code>signed long long</code>), <code>unsigned long long</code> (C99)	≥ 64 bits	-9 223 372 036 854 775 807 à +9 223 372 036 854 775 807	0 à 18 446 744 073 709 551 615 (0xFFFFFFFFFFFFFFFF)

◆ Le type *enum* est un type énuméré

`enum jour {lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche};`

Langage C (9/32)

- ◆ Les types de nombres à virgule flottante :

Types décimaux, en ordre croissant

Type	Précision	Magnitude
float	≥ 6 chiffres décimaux	environ 10^{-37} à 10^{+37}
double	≥ 10 chiffres décimaux	environ 10^{-37} à 10^{+37}
long double	≥ 10 chiffres décimaux	environ 10^{-37} à 10^{+37}
long double (C89)	≥ 10 chiffres décimaux	

- ◆ C99 a ajouté pour représenter les nombres complexes associés :

float complex

double complex

long double complex

Langage C (10/32)

- ◆ Le type booléen
 - ◆ Les versions antérieures à C99 ne proposent pas de type booléen. Il fallait le définir :

```
enum boolean {false, true};  
typedef enum boolean bool;
```

Ou :

```
typedef enum boolean {false, true} bool;
```
 - ◆ C99 ajoute le type _Bool

```
_Bool true = 1;
```

Langage C

(11/32)

- ◆ Les unions :

```
union id
{
    int entier;
    double flottant;
    char lettre;
};
```

L'accès aux champs :

```
union id var;
var.entier = 42;
```

Langage C

(12/32)

◆ Les structures

- ◆ Une structure est un regroupement de variables (champs) dans un même bloc.
- ◆ Pour définir une structure, il faut utiliser le mot réservé « struct » suivi du nom de la structure. Les variables sont ensuite être déclarées dans un bloc qui se finit par un point-virgule :

```
struct personne
{
    int age;
    char *nom;
};
```

Langage C

(13/32)

- ◆ Les structures (suite)

L'accès aux champs :

```
int main()
{
    struct personne p;

    p.nom = "Albert";
    p.age = 46;
}
```

Langage C

(14/32)

- ◆ Les pointeurs
 - ◆ Un pointeur a pour valeur l'adresse d'un objet C d'un type donné (un pointeur est typé). Ainsi, un pointeur contenant l'adresse d'un entier sera de type pointeur vers entier.
 - ◆ L'opérateur & permet de connaître l'adresse d'une variable, on dira aussi la référence. Toute déclaration de variable occupe un certain espace dans la mémoire de l'ordinateur. La référence permet de savoir où cet emplacement se trouve.

```
int i;  
printf("%p\n", &i);
```

Langage C

(15/32)

- ♦ Les pointeurs (suite)
 - ♦ Pouvoir récupérer l'adresse n'a d'intérêt que si on peut manipuler l'objet pointé. Il faut donc déclarer un pointeur. Pour cela, on utilise l'étoile (*) entre le type et le nom de la variable pour indiquer qu'il s'agit d'un pointeur.

```
int a = 2;  
char b;
```

```
int *p1;  
char *p2;
```

```
p1 = &a;  
p2 = &b;
```

Langage C (16/32)

- ◆ Les tableaux [...]
 - ◆ Un tableau qui contient 8 éléments de type char :
`char Tableau [8]`

Définition du tableau	Taille du tableau (en octets)
<code>char Tableau1[12]</code>	$1 * 12 = 12$
<code>int Tableau2[10]</code>	$2 * 10 = 20$
<code>float Tableau3[8]</code>	$4 * 8 = 32$
<code>double Tableau4[15]</code>	$8 * 15 = 120$

L'opérateur `sizeof()` permet de retourner la taille de l'élément qui lui est passé en argument

Langage C

(17/32)

- ◆ Les fonctions (...)

- ◆ Des fonctions sont stockées dans des bibliothèques, tout ce dont nous avons besoin de connaître pour les utiliser : printf, scanf, strlen...
- ◆ On peut créer aussi ses propres fonctions :

```
Type Nom_fx (type_param1 nom_param1, type_param2 nom param2...)  
{  
    contenu de la fonction...  
}
```

- ◆ La dernière instruction de la fonction est l'instruction return si cette dernière doit retourner un résultat. Le type du résultat doit correspondre à celui déclaré comme type de retour de la fonction.

Langage C (18/32)

◆ Les commentaires

- ◆ Pour les versions antérieures au C99, les commentaires commence par « /* » et se termine par « */ ». Tout ce qui est compris entre ces 2 symboles est considéré comme commentaire, saut de ligne compris :

```
/* Ceci est  
un commentaire  
multi-lignes */
```

- ◆ C99 a repris les commentaires de fin de ligne du C++, introduits par deux barres obliques et se terminant avec la ligne :

```
// Commentaire jusqu'à la fin de la ligne
```

Langage C

(19/32)

- ♦ Structures de contrôle : les tests
 - ♦ if (expression) instruction
 - ♦ if (expression) instruction else instruction
 - ♦ condition ? expression_si_vrai : expression_si_faux ;
 - ♦ switch (expression) instruction avec case et default dans l'instruction

Langage C

(20/32)

- ♦ Structures de contrôle : les itérations
 - ♦ Boucle for :
for (initialisation ; condition ; itération)
bloc
 - ♦ Boucle while :
while (condition)
bloc
 - ♦ Boucle do... while :
do
bloc
while (condition);

Langage C

(21/32)

- ♦ Structures de contrôle : les itérations (suite)
 - ♦ L'instruction « break » permet de sortir immédiatement d'une boucle for, while ou do...while.
 - ♦ L'instruction « continue » permet de recommencer la boucle depuis le début du bloc.

Langage C

(22/32)

- ◆ Structures de contrôle : Saut inconditionnel
 - ◆ L'instruction « goto » permet de continuer l'exécution du programme à un autre endroit, dans la même fonction :
`goto label;`
 - ◆ Où label est un identificateur quelconque. Cet identificateur devra être défini quelque part dans la même fonction, avec la syntaxe suivante :
`label:`

Langage C (23/32)

- ♦ La bibliothèque standard C ISO
 - ♦ Elle consiste en 24 en-têtes (fichiers headers « .h ») qui peuvent être inclus dans un projet de programmeur avec une simple directive. Chaque en-tête contient des prototypes de fonctions, des définitions de types et de macros.

Langage C (24/32)

- ◆ La bibliothèque standard C ISO (suite)
 - ◆ <assert.h>
Contient la macro assert utilisée pour aider à détecter des incohérences de données et d'autres types de bogues dans les versions de débogage d'un programme.
 - ◆ <complex.h>
Pour manipuler les nombres complexes (C99).
 - ◆ <ctype.h>
Fonctions utilisées pour classifier rapidement les caractères ou pour convertir entre majuscules et minuscules de manière indépendante du système de codage des caractères.

Langage C (25/32)

- ◆ La bibliothèque standard C ISO (suite)
 - ◆ <errno.h>
Ensemble des codes d'erreurs renvoyés par les fonctions de la bibliothèque std via la variable errno.
 - ◆ <fenv.h>
Contrôle l'environnement en virgule flottante (C99).
 - ◆ <float.h>
Contient des constantes qui spécifient les propriétés des nombres en virgule flottante qui dépendent de l'implémentation telles que la différence minimale entre deux nombres en virgule flottante différents (xxx_EPSILON), le nombre maximum de chiffres de précision (xxx_DIG) et l'intervalle des nombres pouvant être représentés (xxx_MIN, xxx_MAX).

Langage C (26/32)

- ◆ La bibliothèque standard C ISO (suite)
 - ◆ <inttypes.h>
Pour des conversions précises entre types entiers (C99).
 - ◆ <iso646.h>
Pour programmer avec le jeu de caractères ISO 646 (introduit par Amd.1).
 - ◆ <limits.h>
Contient des constantes qui spécifient les propriétés des types entiers qui dépendent de l'implémentation, comme les intervalles des nombres pouvant être représentés (xxx_MIN, xxx_MAX).

Langage C (27/32)

- ◆ La bibliothèque standard C ISO (suite)
 - ◆ <locale.h>
Pour s'adapter aux différentes conventions culturelles.
 - ◆ <math.h>
Pour calculer des fonctions mathématiques courantes.
C99 a ajouté de nombreuses fonctions mathématiques, en particulier pour converger avec la norme CEI 559 dite aussi IEEE 754.
 - ◆ <setjmp.h>
Pour exécuter des instructions goto non locales (sortes d'exceptions).

Langage C (28/32)

- ◆ La bibliothèque standard C ISO (suite)
 - ◆ <signal.h>
Pour contrôler les signaux (conditions exceptionnelles demandant un traitement immédiat, par exemple signal de l'utilisateur).
 - ◆ <stdarg.h>
Pour créer des fonctions avec un nombre variable d'arguments.
 - ◆ <stdbool.h> Pour avoir une sorte de type booléen (C99).
 - ◆ <stddef.h>
Définit plusieurs types et macros utiles, comme NULL.

Langage C (29/32)

- ◆ La bibliothèque standard C ISO (suite)
 - ◆ <stdint.h>
Définit divers types d'entiers, c'est un sous-ensemble de inttypes.h (introduit par C99).
 - ◆ <stdio.h>
Fournit les capacités centrales d'entrée/sortie du langage C, comme la fonction printf.
 - ◆ <stdlib.h>
Pour exécuter diverses opérations dont la conversion, la génération de nombres pseudo-aléatoires, l'allocation de mémoire, le contrôle de processus, la gestion de l'environnement et des signaux, la recherche et le tri.

Langage C (30/32)

- ◆ La bibliothèque standard C ISO (suite)
 - ◆ <string.h>
Pour manipuler les chaînes de caractères.
 - ◆ <tgmath.h>
Pour des opérations mathématiques sur des types génériques (C99).
 - ◆ <time.h>
Pour convertir entre différents formats de date et d'heure.
 - ◆ <wchar.h>
Pour manipuler les caractères larges nécessaire pour supporter un grand nombre de langues et Unicode (Amd.1).

Langage C (31/32)

- ◆ La bibliothèque standard C ISO (suite)
 - ◆ <wctype.h>
Pour classifier les caractères larges (introduit par Amd.1).

Langage C (32/32)

- ♦ Les bibliothèques externes
 - ♦ De nombreuses bibliothèques ont été créées :
libjpeg,
libpng,
expat...
 - ♦ Pour connaître les fonctions dans une bibliothèque :
objdump -T /usr/lib/x86_64-linux-gnu/libjpeg.so.62
 - ♦ Ils sont précédés du mot « Base »

Outils pour développer



GNU Compiler Collection (1/4)

- ◆ GCC est un logiciel libre créé par le projet GNU qui comporte un ensemble de compilateurs :
 - ◆ Langage C (gcc),
 - ◆ Langage C++ (g++),
 - ◆ Objective-C (Gobjc),
 - ◆ Objective-C++ (Gobjc++),
 - ◆ Java (GCJ),
 - ◆ Ada (GNAT)
 - ◆ Fortran (Gfortran).
 - ◆ Go (gccgo).

GNU Compiler Collection (2/4)

- ♦ Installation de GCC (Distribution Debian)

- ♦ Dans une session root :

```
# apt-get install build-essential
```

- ♦ Le paquet build-essential contient :

- ♦ libc6-dev (Bibliothèque standard C),
 - ♦ gcc (Compilateur C),
 - ♦ g++ (Compilateur C++),
 - ♦ make,
 - ♦ dpkg-dev (Debian package development tools).



GNU Compiler Collection (3/4)

- ◆ Installation de GCC (Distribution Red Hat)
 - ◆ Dans une session root :

```
# yum group install "Development Tools"
```
- ◆ Les paquets du groupe développement sont :
 - ◆ autoconf, automake, binutils, bison
 - ◆ flex, gcc, gcc-c++, gettext, libtool ,
 - ◆ make, patch, pkgconfig
 - ◆ redhat-rpm-config, rpm-build, rpm-sign



GNU Compiler Collection (4/4)

- ◆ Pour identifier la version du compilateur :

```
gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/4.8/lto-wrapper
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 4.8.2-19ubuntu1' --with-bugurl=file:///usr/share/doc/gcc-4.8/README.Bugs --enable-languages=c,c++ +,java,go,d,fortran,objc,obj-c++ --prefix=/usr --program-suffix=-4.8 --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --with-gxx-include-dir=/usr/include/c++/4.8 --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --enable-gnu-unique-object --disable-libmudflap --enable-plugin --with-system-zlib --disable-browser-plugin --enable-java-awt=gtk --enable-gtk-cairo --with-java-home=/usr/lib/jvm/java-1.5.0-gcj-4.8-amd64/jre --enable-java-home --with-jvm-root-dir=/usr/lib/jvm/java-1.5.0-gcj-4.8-amd64 --with-jvm-jar-dir=/usr/lib/jvm-exports/java-1.5.0-gcj-4.8-amd64 --with-arch-directory=amd64 --with-ecj-jar=/usr/share/java/eclipse-ecj.jar --enable-objc-gc --enable-multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --with-tune=generic --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
```

Thread model: posix
gcc version 4.8.2 (Ubuntu 4.8.2-19ubuntu1)

Compiler avec gcc

- ◆ Lorsque gcc compile un fichier, il produit par défaut en sortie un exécutable nommé *a.out*.

```
gcc compil.c
```

```
ls  
a.out      compil.c
```

- ◆ Le fichier s'exécute ainsi :

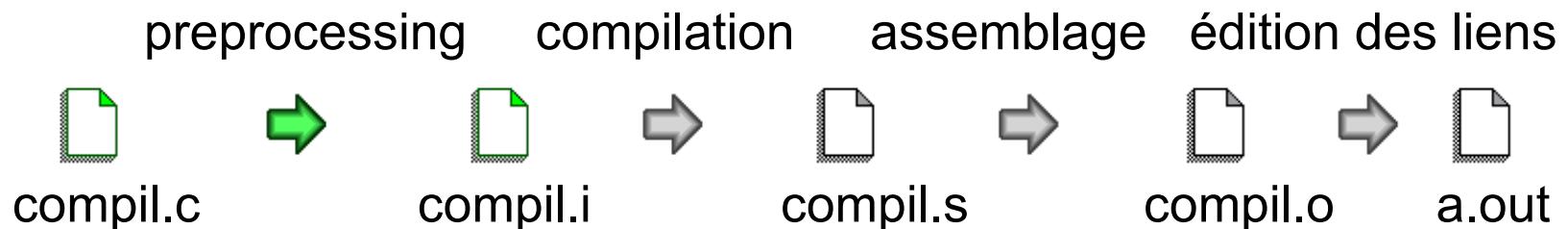
```
./a.out
```

- ◆ Pour nommer l'exécutable, il faut utiliser l'option **-o** (output)

```
gcc compil.c -o appli
```

Étapes de la compilation (1/2)

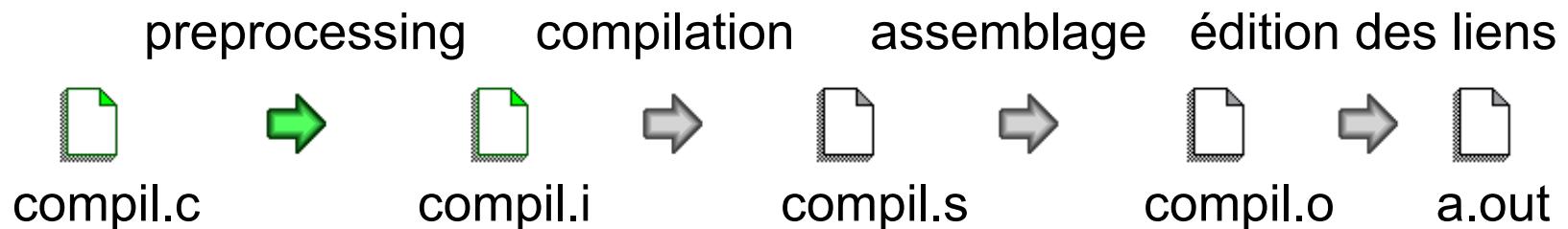
- Le compilateur gcc effectue 4 étapes pour produire un exécutable :



1. Passage au pré-processeur (preprocessing)
2. Compilation en langage assembleur (compiling)
3. Conversion du langage assembleur en code machine (assembling)
4. Édition des liens (linking)

Étapes de la compilation (2/2)

- Le compilateur gcc effectue 4 étapes pour produire un exécutable :



- Les fichiers intermédiaires ne sont pas présents dans le dossier une fois la compilation terminée. Seuls les fichiers sources *.c et l'exécutable apparaissent.

1ère étape de la compilation preprocessing (1/4)

- ◆ L'option -E interrompt la compilation après la 1ère étape :

```
gcc -E compil.c > compil.i
```

```
ls  
compil.c      compil.i
```

- ◆ Le pré-processeur réalise plusieurs opérations de substitution sur le code source :
 - ◆ Suppression des commentaires // et /* */
 - ◆ Inclusion (#include) des fichiers d'en-têtes *.h dans le fichier source .c
 - ◆ Traitement des directives de compilation (#define, #ifdef, #pragma...)

1ère étape de la compilation preprocessing (2/5)

- Le contenu du fichier compil.i est un fichier texte en langage C lisible :

```
# 1 "compil.c"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2

...
int main(void)
{
    clrscr();

    gotoxy(10,15);
    printf("%s\n","Processus de compilation...");
    return (0);
}
```

1ère étape de la compilation preprocessing (3/5)

- ♦ Et en cas d'erreurs de syntaxe ?
 - ♦ gcc affiche les erreurs à la console et il interrompt le traitement. Il ne passera pas à l'étape suivante.

```
gcc -E compil.c > compil.i
compil.c:20:12: warning: missing terminating " character [enabled by default]
    printf("\033[2J\033[1;1H);
               ^

```

1ère étape de la compilation preprocessing (4/5)

- ♦ Et en cas d'avertissemens (warnings) ?
 - ♦ Contrairement aux erreurs les avertissements (warnings) ne sont pas bloquants.
 - ♦ Ne s'affichant pas automatiquement sur la console, il faut utiliser l'option *-Wall*.

```
#gcc -Wall err_compil.c

err_compil.c: In function ‘clrscr’:
err_compil.c:18:5: warning: implicit declaration of function ‘print’ [-Wimplicit-function-declaration]
    print("\033[2J\033[1;1H");
           ^
/tmp/ccgU1zgC.o: dans la fonction « clrscr »:
err_compil.c:(.text+0xf): référence indéfinie vers « print »
collect2: error: ld returned 1 exit status
```

1ère étape de la compilation preprocessing (4/5)

- ◆ En cas d'avertissemens (warnings) ou d'erreurs?
 - ◆ -W
Supprime tous les avertissemens.
 - ◆ -W
GCC est plus exigeant quant aux warnings.
 - ◆ -Wall
GCC est encore plus exigeant. Mais, attention !
Avec cette option, le compilateur peut trouver des erreurs dans des bibliothèques externes que l'on utilise...
 - ◆ -Werror
Tous les warnings deviennent des erreurs. Seul le code « parfait » compilera !

2ème étape de la compilation compiling

- Le code C est transformé en assembleur en utilisant l'option -S de gcc :

```
gcc -S compil.i
```

- Un fichier *compil.s* est créé :

```
.file  "compil.c"
       .section      .rodata
.LC0:
       .string  "\033[2J\033[1;1H"
       .text
       .globl  clrscr
       .type   clrscr, @function
clrscr:
.LFB2:
       .cfi_startproc
       pushq  %rbp
       .cfi_def_cfa_offset 16
       .cfi_offset 6, -1
...
```

3ème étape de la compilation assembling (1/4)

- Le code assembleur qui est lisible est transformé en code machine binaire avec l'option -c de gcc :

```
gcc -c compil.s
```

- Ou en utilisant la commande as (the portable GNU assembler) :

```
as -o compil.o compil.s
```

3ème étape de la compilation assembling (2/4)

- Le fichier objet *compil.o* obtenu est binaire et ne peut pas donc être directement édité ou lu par un humain. Il faut utiliser la commande *od* :

```
od -x compil.o
```

```
0000000 457f 464c 0102 0001 0000 0000 0000 0000
0000020 0001 003e 0001 0000 0000 0000 0000 0000
0000040 0000 0000 0000 0000 00b8 0000 0000 0000
0000060 0000 0000 0040 0000 0000 0040 0009 0006
0000100 4700 4343 203a 5528 7562 746e 2075 2e34
0000120 2e38 2d34 7532 7562 746e 3175 317e 2e34
0000140 3430 312e 2029 2e34 2e38 0034 2e00 7973
0000160 746d 6261 2e00 7473 7472 6261 2e00 6873
0000200 7473 7472 6261 2e00 6574 7478 2e00 6164
0000220 6174 2e00 7362 0073 632e 6d6f 656d 746e
0000240 2e00 6f6e 6574 472e 554e 732d 6174 6b63
```

...

3ème étape de la compilation assembling (3/4)

- Le fichier objet peut être lu aussi par *objdump* :

```
objdump -t compil1.o
```

```
compil1.o:    format de fichier elf64-x86-64
```

SYMBOL TABLE:

```
0000000000000000 I  df *ABS* 0000000000000000 compil1.c
0000000000000000 I  d .text 0000000000000000 .text
0000000000000000 I  d .data 0000000000000000 .data
0000000000000000 I  d .bss 0000000000000000 .bss
0000000000000000 I  d .rodata 0000000000000000 .rodata
0000000000000000 I  d .note.GNU-stack 0000000000000000 .note.GNU-stack
0000000000000000 I  d .eh_frame 0000000000000000 .eh_frame
0000000000000000 I  d .comment 0000000000000000 .comment
0000000000000000 g  F .text 000000000000015 clrscr
0000000000000000      *UND* 0000000000000000 printf
000000000000000015 g  F .text 0000000000000027 gotoxy
...
```

3ème étape de la compilation assembling (4/4)

- ◆ Assemblage de fichiers sources :

```
gcc -c compil1.c compil2.c
```

- ◆ Résultat :

```
compil1.c
compil1.o
compil2.c
compil2.o
```

4ème étape de la compilation linking

- Le fichier *compil.o* produit par l'étape 3 est incomplet, il ne contient pas le code des fonctions telles que *printf()*. Ce code est dans une bibliothèque.
- L'édition des liens va réunir le fichier objet et les fonctions contenues dans les bibliothèques, pour produire le programme complet e.i. l'exécutable *a.out* :

```
gcc compil1.o compil2.o
```

ou un nom de fichier spécifié avec *-o* :

```
gcc -o compil compil1.o compil2.o
```

Autres options pour compiler (1/2)

- ◆ -pipe
gcc ne génère pas de fichiers temporaires entre chaque étape. Il utilise des tubes comme les « | » en shell.
- ◆ -v
Mode verbeux.
- ◆ -I
Spécifie le ou les répertoire(s) des fichiers headers « .h ».

Autres options pour compiler (2/2)

- ◆ **-Ox**

gcc optimise votre code. Il va modifier le code source de façon à ce que le programme ait le même résultat par le chemin le plus court.

x peut varier entre 1 et 6. Plus le degré est fort, plus l'optimisation est bonne. La compilation risque d'être longue et gourmande en mémoire !

- ◆ **-Os**

gcc organisera votre code de manière à ce qu'il soit le plus court possible.

TP : Compilation (1/4)

- ◆ Créez un dossier 00-hello-ansi-c et écrivez le code hello-ansi.c :
 - Inclure les fichiers headers nécessaires.
 - Fonction clrscr()
 - effacer l'écran
 - Fonction gotoxy(colonne,ligne)
 - positionner le curseur sur l'écran
 - Fonction main
 - effacer l'écran
 - placer le curseur sur la colonne 5 et ligne 10
 - avec la fonction printf, afficher le message suivant :
 - « Hello world! »
 - Terminer la fonction avec le code retour EXIT_SUCCESS
- ◆ Compiler le fichier hello-ansi.c. Le nom de l'exécutable doit se nommer hello

TP : Compilation

Réponse (1/4)

```
// File: hello-ansi.c
// Compilation : gcc -o hello hello-ansi.c

#include <stdio.h>                                // pour la fonction printf()
#include <stdlib.h>                               // pour le code retour EXIT_SUCCESS

void clrscr(void) {
    printf("\033[2J\033[1;1H");
}

void gotoxy(int col, int lig) {
    printf("\x1b[%d;%df", lig, col);
}

int main(void) {
    clrscr();
    gotoxy(10,15);
    printf("%s\n", "Hello world!");
    return (EXIT_SUCCESS);
}
```

TP : Compilation (2/4)

- ◆ Supprimer l'exécutable hello

Effectuez la compilation pas à pas :

1) Étape preprocessing

Quel est le nom du fichier obtenu ?

Lire le fichier obtenu. Que s'est-il produit ?

2) Étape compiling

Quel est le nom du fichier obtenu ?

Lire le fichier obtenu. Que s'est-il produit ?

3) Étape Assembling

Quel est le nom du fichier obtenu ?

Lire le fichier obtenu. Que s'est-il produit ?

4) Étape linking

Le nom du fichier de sortie doit être : hello

Tester son exécution

TP : Compilation Réponse (2/4)

- ◆ Supprimer l'exécutable hello : rm ./hello

Effectuez la compilation pas à pas :

1) Étape preprocessing : gcc -E hello-ansi.c > hello.i

Le nom du fichier obtenu est hello.i

Le pré-processeur réalise plusieurs opérations de substitution sur le code source

2) Étape compiling : gcc -S hello.i

Le nom du fichier obtenu est hello.s

Le code C est transformé en assembleur

3) Étape Assembling : gcc -c hello.s

Ou bien : as -o hello.o hello.s

Le nom du fichier obtenu est hello.o

Le code assembleur est transformé en code machine (binaire)

4) Étape linking : gcc -o hello hello.o

L'édition réunit le(s) fichier(s) objet(s) et les fonctions contenues dans les bibliothèques pour produire le programme complet : hello

TP : Compilation (3/4)

- ◆ Effacer tous les fichiers produits lors de la précédente compilation
- ◆ Effectuer maintenant la compilation avec l'option qui demande au compilateur d'afficher tous les messages de prévention (warnings)
- ◆ Quels sont les fichiers produits ?

TP : Compilation Réponse (3/4)

- ◆ Effacer tous les fichiers produits lors de la précédente compilation
`rm hello.[ios] && rm hello`
- ◆ Effectuer maintenant la compilation avec l'option qui demande au compilateur d'afficher tous les messages de prévention (warnings)
`gcc -Wall -o hello hello.c`
- ◆ Quels sont les fichiers produits ?
`hello`

Les fichiers intermédiaires ne sont pas présents dans le dossier une fois la compilation terminée. Seuls les fichiers sources *.c et l'exécutable apparaissent.

TP - Code divisé en plusieurs fichiers sources

(4/4)

- ◆ Créer un fichier ficmain.c qui contiendra la fonction main().
Cette dernière devra :
 - ◆ Appeler la fonction clrscr() pour effacer l'écran,
 - ◆ Appeler la fonction gotoxy() pour positionner le curseur colonne 10 et ligne 15 sur l'écran,
 - ◆ Appeler la fonction print_hello() pour afficher le texte « Hello world! »
 - ◆ Retourner le code retour EXIT_SUCCESS
- ◆ Les 3 fonctions doivent être écrites dans le fichier fic02.c.
- ◆ Les prototypes sont déclarées dans le fichier fx.h
- ◆ Compiler l'ensemble.
Le fichier exécutable s'appelle hello

TP - Code divisé en plusieurs fichiers sources

Réponse (4/4) – Fichier 1 sur 3

```
// File name: ficmain.c
// Compilation: gcc -o hello fic02.c ficmain.c

#include <stdlib.h>
#include "fx.h"

int main (void)
{
    clrscr();

    gotoxy(10,15);
    print_hello();

    return (EXIT_SUCCESS);
}
```

TP - Code divisé en plusieurs fichiers sources

Réponse (4/4) – Fichier 2 sur 3

```
// File name : fx.h

#ifndef H_HELLO
#define H_HELLO
    void print_hello(void);
#endif

#ifndef H_CLRSCR
#define H_CLRSCR
    void clrscr(void);
#endif

#ifndef H_GOTOXY
#define H_GOTOXY
    void gotoxy(int col, int lig);
#endif
```

TP - Code divisé en plusieurs fichiers sources

Réponse (4/4) – Fichier 3 sur 3

```
// File name : fic02.c

#include <stdio.h>

void clrscr(void) {
    printf("\033[2J\033[1;1H");
}

void gotoxy(int col, int lig) {
    printf("\x1b[%d;%df",lig,col);
}

void print_hello(void) {
    puts("Hello world!");
}
```

Bibliothèques statiques (1/2)

- ◆ Le code des fonctions d'une librairie statique est introduit dans le fichier exécutable.
- ◆ Avantage : le fichier exécutable n'est plus dépendant des librairies installées dans le système.
- ◆ Inconvénient : si on modifie la librairie, il faudra recompiler l'exécutable pour obtenir un nouveau programme utilisant la nouvelle librairie.

Bibliothèques statiques (2/3)

- ◆ L'extension des fichiers bibliothèques statiques est « .a »
- ◆ Créer une librairie :

```
gcc -c fic01.c fic02.c
mkdir libs
ar cr libs/libfic.a fic01.o fic02.o
ar r libs/libfic.a fic03.o
```

- ◆ Afficher le contenu de la librairie :

```
ar t libfic.a
nm -s libs/libfic.a
nm -s --defined-only libs/libfic.a
```

- ◆ Remplacer ou ajouter un fichier objet

```
ar r libs/libfic.a fic03.o
```

Bibliothèques statiques (3/3)

- ◆ Ajouter un fichier objet

```
ar a libs/libfic.a fic03.o
```

- ◆ Détruire un fichier objet

```
ar d libs/libfic.a fic03.o
```

- ◆ Compiler

```
gcc -o ficmain ficmain.c ./libs/libfic.a
```

TP : Créer une bibliothèque statique (1/2)

- ◆ Créer un fichier fic01.c qui contient la fonction clrscr()
Créer un fichier fic02.c qui contient la fonction gotoxy()
Créer un fichier ficmain.c qui contient la fonction main()
- ◆ Effectuer l'assembling des 2 fichiers fic01.c et fic02.c
- ◆ Créer 1 répertoire ./libs
Ajouter les 2 fichiers objets dans la librairie libfic.a
- ◆ Compiler le fichier ficmain.c avec la librairie libfic.a

TP : Créer une bibliothèque statique

Réponse (1/2)

- ◆ Créer un fichier fic01.c qui contient la fonction clrscr()
Créer un fichier fic02.c qui contient la fonction gotoxy()
Créer un fichier ficmain.c qui contient la fonction main()
- ◆ Effectuer l'assembling des 2 fichiers fic01.c et fic02.c

```
gcc -c fic01.c fic02.c
```
- ◆ Créer 1 répertoire ./libs
Ajouter les 2 fichiers objets dans la librairie libfic.a

```
mkdir libs  
ar rcs libs/libfic.a fic01.o fic02.o
```
- ◆ Compiler le fichier ficmain.c avec la librairie libfic.a

```
gcc -o ficmain ficmain.c ./libs/libfic.a
```

Bibliothèques dynamiques (1/5)

- ◆ L'extension d'une bibliothèque dynamique est .so (bibliothèque partagée ou shared object) suivie, le plus souvent, du numéro de sa version.
- ◆ Pendant l'édition des liens, une portion de code est insérée et sera utilisée lorsque l'application sera exécutée pour lancer l'éditeur de liens dynamique.
- ◆ Le code des fonctions n'est plus dupliqué dans chaque exécutable. Il est présent uniquement dans le fichier bibliothèque partagé.

Bibliothèques dynamiques (2/5)

- ◆ L'éditeur de liens dynamique localise les bibliothèques partagées et les charge en mémoire.
- ◆ Le fichier exécutable occupe donc moins d'espace disque, et moins d'espace mémoire.
- ◆ Quand plusieurs applications utilisent les mêmes fonctions de bibliothèques partagées, elles sont exécutées en même temps, le code des fonctions est présent qu'une seule fois dans la mémoire.

Bibliothèques dynamiques (3/5)

- ◆ Compiler le code source avec les options suivantes :
 - ◆ -fPIC pour que le code ne soit pas « re-logeable » (Position Independent Code).
 - ◆ -shared pour créer, lors de la compilation, la bibliothèque partagée.

```
mkdir ./lib
gcc -fPIC -shared -Wl,-soname,libfic.so.1 -o lib/libfic.so.1.0 fic01.c fic02.c
cd lib
ln -sf libfic.so.1.0 libfic.so
```

- ◆ Les numéros 1 et 0 correspondent respectivement aux numéros majeur et mineur de version de la bibliothèque.

Bibliothèques dynamiques (4/5)

- ◆ L'option -WI enregistre dans l'en-tête de la bibliothèque son nom officiel incluant le numéro majeur de version et on transmet la valeur -soname.
- ◆ L'application cherchera libfic.so et non libfic.so.1. il faut donc faire un lien symbolique.
- ◆ Dans /etc/ld.so.conf.d créez 1 fichier pmylibs.conf et ajoutez le chemin du répertoire où se situe votre bibliothèque. Actualisez le cache avec ldconfig :

```
ldconfig  
ldconfig -v |grep libfic
```

Bibliothèques dynamiques (5/5)

- ◆ Pour compiler votre application :

```
gcc -L./lib ficmain.c -o ficmain -lfic
```

- ◆ `export LD_librarie_PATH=`pwd`` est identique à l'option `-L` de `gcc`.
- ◆ `ldd` permet de connaître les bibliothèques partagées d'une application :

```
ldd ficmain
    linux-vdso.so.1 => (0x00007ffd139b9000)
    libfic.so.1 => /home/phil/Workspace/Dev/C/00-Compilation/3-
compil/lib/libfic.so.1 (0x00007fc49bed5000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fc49bb10000)
    /lib64/ld-linux-x86-64.so.2 (0x0000556493de3000)
```

TP : Créer une bibliothèque dynamique (1/1)

- ◆ Copier dans un nouveau dossier fic01.c fic02.c et ficmain.c
- ◆ Créer un sous-dossier libs
Faites l'assembling des fichiers fic01.c et fic02.c
Ajouter les dans la bibliothèque libfic.so
- ◆ Assurez-vous que la bibliothèque soit utilisable dans le système
- ◆ Compiler le fichier ficmain.c en utilisant la bibliothèque libfic.so

TP : Créer une bibliothèque dynamique

Réponse (1/1)

- ◆ Copier dans un nouveau dossier fic01.c fic02.c et ficmain.c
- ◆ Créer un sous-dossier libs
Faites l'assembling des fichiers fic01.c et fic02.c
Ajouter les dans la bibliothèque libfic.so
 - mkdir ./libs
 - gcc -fPIC -shared -Wl,-soname,libfic.so.1 -o libs/libfic.so.1.0 fic01.c fic02.c
- ◆ Assurez-vous que la bibliothèque soit utilisable dans le système
 - cd libs
 - ln -sf libfic.so.1.0 libfic.soDans /etc/ld.so.conf.d créer 1 fichier mylibs.conf pour ajouter le chemin du dossier libs et actualiser avec ldconfig. Vérifier avec : ldconfig -v | grep libfic
- ◆ Compiler le fichier ficmain.c en utilisant la bibliothèque libfic.so
 - gcc -L./libs ficmain.c -o ficmain -lfic

Débogage GNU Debugger – GDB (1/)

- ◆ GNU DeBugger est un programme qui permet de débugger un programme écrit en langages C et C++ principalement. Il permet donc de traquer les bugs se trouvant dans le code source.
- ◆ Comment l'installer ?
 - ◆ GDB est disponible dans la plupart des dépôts (apt-get ou yum...)
 - ◆ Il peut être téléchargé directement depuis le site officiel : <https://sourceware.org/gdb/>
Ensuite, il faudra le compiler puis l'installer.

Débogage GNU Debugger – GDB (2/)

- ◆ GDB fonctionne sur le principe d'une invite de commandes.
- ◆ Pour démarrer une session, il faut lancer GDB en lui passant éventuellement des paramètres.
- ◆ Exemples :

```
gdb ficmain.c -q
gdb ficmain dumpfile
```

- ◆ Avec dumpfile, le système a enregistré dans un fichier une copie de ce qui se trouvait en mémoire au moment du plantage (core dumped).
- ◆ L'option -silent (-quiet ou -q) permet de ne pas afficher les quelques informations légales au lancement.
- ◆ La commande h ou help pour invoquer l'aide :

```
(gdb) help
```

Débogage GNU Debugger – GDB

(3/)

- ◆ L'option -g de gcc permet de compiler avec les informations de débogage :

```
gcc -g -o fuite_mem fuite_mem.c
```

- ◆ Il suffit de lancer gdb avec le nom du programme :

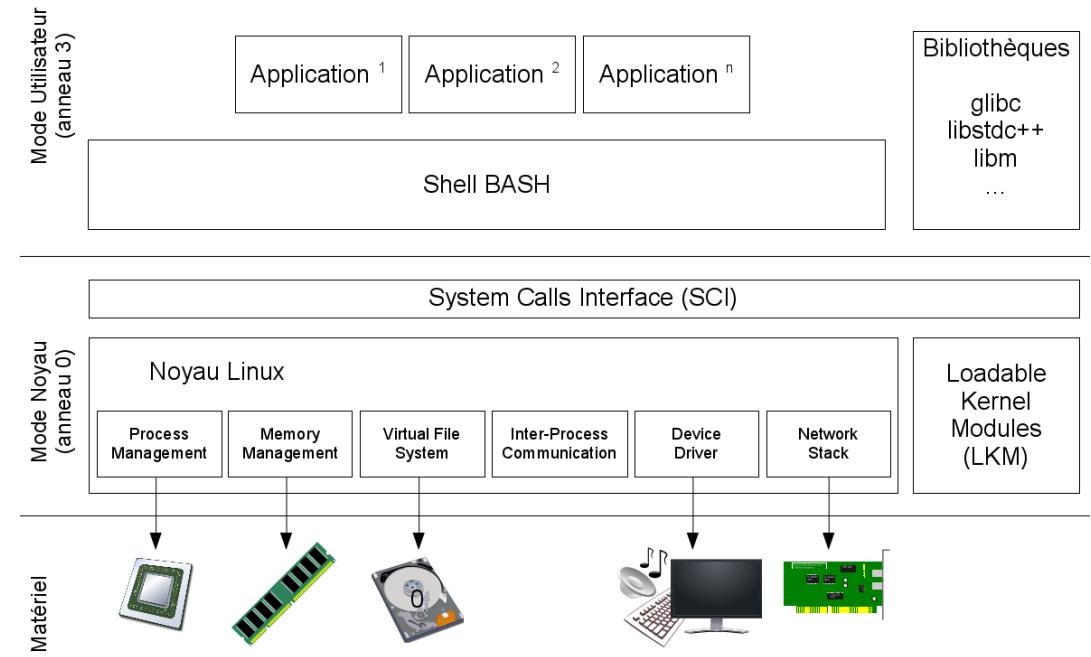
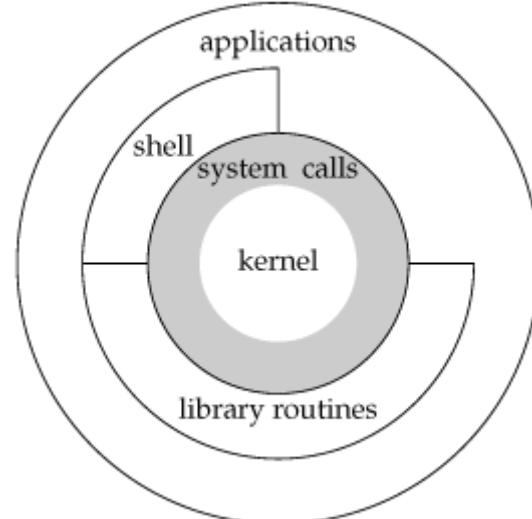
```
gdb fuite_mem
```

- ◆ < A FINIR...>

DataDisplay Debugger (DDD)

- ◆ Non écrit
- ◆

Système d'exploitation



Système d'exploitation

Introduction (1/3)

- ◆ Un OS est un ensemble de programmes qui effectue l'interface entre la matériel de la machine et les utilisateurs.
- ◆ Le système a 2 objectifs :
 - ◆ Prendre en charge la gestion de plus en plus complexe des ressources et partager celles-ci.
 - ◆ Construire au-dessus du matériel une machine virtuelle plus facile d'emploi et plus conviviale.

Système d'exploitation

Introduction (2/3)

- ◆ Le rôle du système dans un environnement multi-programmé est de gérer le partage de la machine physique et des ressources matérielles entre les différents programmes.
- ◆ Cette gestion doit assurer l'équité d'accès aux ressources matérielles et assurer que les accès des programmes à ces ressources s'effectuent correctement e.i. que les opérations réalisées par les programmes sont licites pour la cohérence des ressources. C'est la « protection des ressources ».

Système d'exploitation

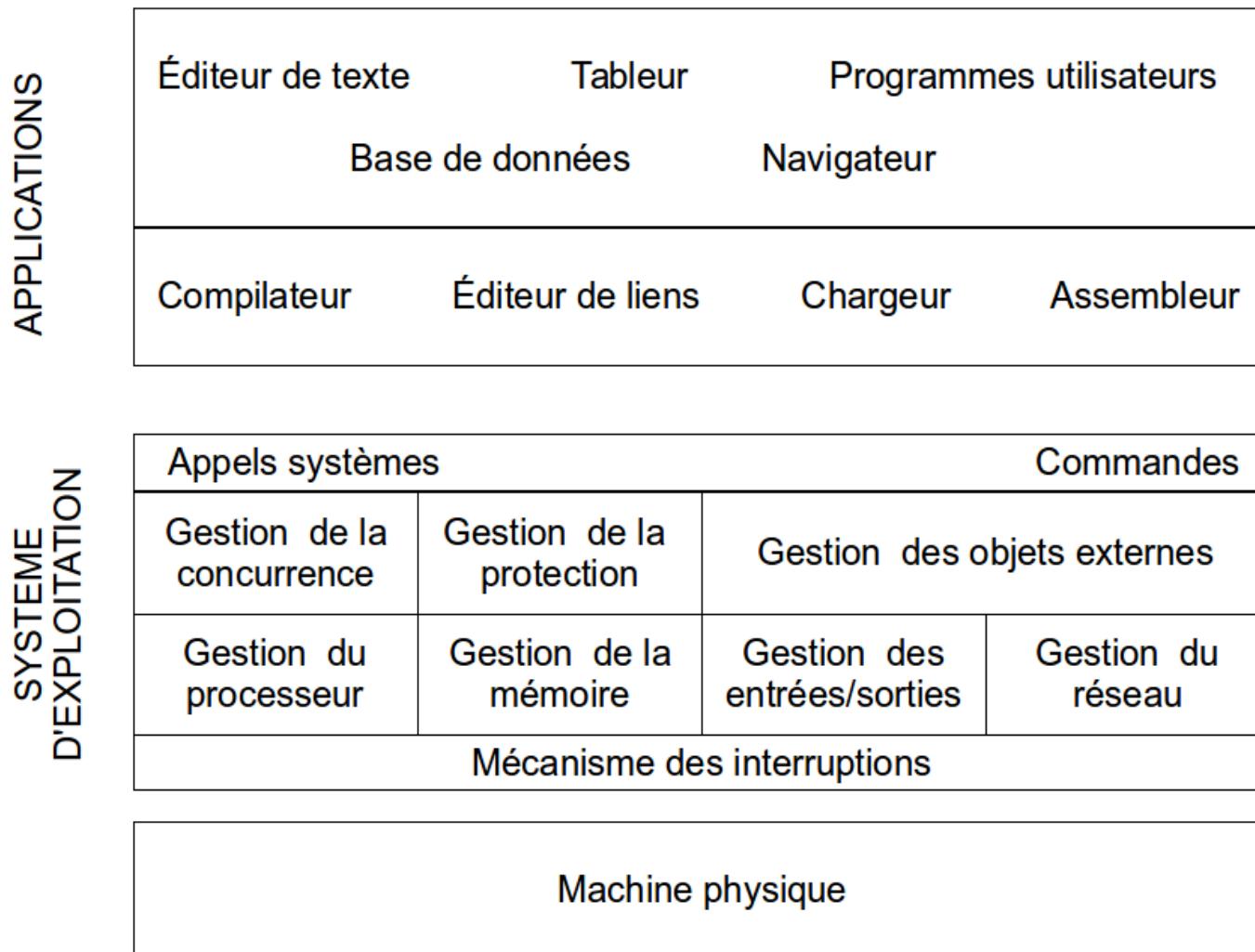
Introduction (3/3)

- ♦ Les systèmes multi-programmés peuvent être classés selon les buts et les services offerts :
 - ♦ Les systèmes à traitements par lots
 - ♦ Les systèmes multi-utilisateurs interactifs
 - ♦ Les systèmes temps réels

Système d'exploitation

Structure générale (1/8)

- L'OS est découpé en plusieurs fonctionnalités :



Système d'exploitation

Structure générale (2/8)

- ♦ L'OS est découpé en plusieurs fonctionnalités :
 - ♦ La gestion du processeur
Le système doit gérer l'allocation du processeur aux différents programmes pouvant s'exécuter. L'allocation s'effectue par le biais d'un algorithme d'ordonnancement qui planifie l'exécution des programmes. Une exécution de programme est appelée « processus ».

Nous allons aborder ce sujet dans le module 2.

Système d'exploitation

Structure générale (3/8)

- ♦ L'OS est découpé en plusieurs fonctionnalités :
 - ♦ La gestion des objets externes
La mémoire centrale est une mémoire volatile.
Toutes les données devant être conservées au-delà l'arrêt de la machine doivent être stockées sur une mémoire de masse non volatile.
La gestion de l'allocation des mémoires de masse ainsi que l'accès aux données stockées s'appuient sur la notion de « fichiers » et de « système de gestion de fichiers (SGF) ».

Nous allons aborder ce sujet dans le module 3.

Système d'exploitation

Structure générale (4/8)

- ♦ L'OS est découpé en plusieurs fonctionnalités :
 - ♦ La gestion des entrées/sorties
Le système doit gérer l'accès aux périphériques e.i. la liaison entre les appels de haut niveau des programmes utilisateurs (exemple : getchar()) et les opérations de bas niveau de l'unité d'échange (clavier) responsable du périphérique.
C'est le pilote de E/S (driver) qui assure la correspondance.

Nous allons aborder ce sujet dans le module 4.

Système d'exploitation

Structure générale (5/8)

- ♦ L'OS est découpé en plusieurs fonctionnalités :
 - ♦ La gestion de la mémoire
Le système doit gérer l'allocation de la mémoire centrale entre les différents programmes. Il doit trouver un espace libre suffisant en mémoire centrale pour que le chargeur puisse y placer un programme à exécuter en s'appuyant sur les mécanismes matériels sous-jacents de segmentation et de pagination.
La mémoire physique étant souvent trop petite pour contenir la totalité des programmes, la gestion de la mémoire se fait selon le principe de la mémoire virtuelle.

Nous allons aborder ce sujet dans le module 5.

Système d'exploitation

Structure générale (6/8)

- ♦ L'OS est découpé en plusieurs fonctionnalités :
 - ♦ La gestion de la concurrence
Puisque plusieurs programmes coexistent en mémoire centrale, ceux-ci peuvent vouloir communiquer pour échanger des données.
Il faut aussi synchroniser l'accès aux données partagées pour maintenir leur cohérence.

Nous allons aborder ce sujet dans le module 6, 7 et 8.

Système d'exploitation

Structure générale (7/8)

- ♦ L'OS est découpé en plusieurs fonctionnalités :
 - ♦ La gestion de la protection
Le système doit fournir des mécanismes garantissant que ses ressources (CPU, mémoire, fichiers) ne peuvent être utilisée que par les programmes auxquels les droits nécessaires ont été accordés. Il faut notamment protéger le système et la machine des programmes utilisateurs (mode utilisateur et superviseur).

Système d'exploitation

Structure générale (8/8)

- ♦ L'OS est découpé en plusieurs fonctionnalités :
 - ♦ L'accès réseau
Des exécutions de programmes placés sur des machines distinctes doivent pouvoir échanger des données.
Le système fournit des outils permettant aux applications distantes de dialoguer à travers une couche de protocoles réseau telle que TCP/IP.

Nous allons aborder ce sujet dans le module 9.

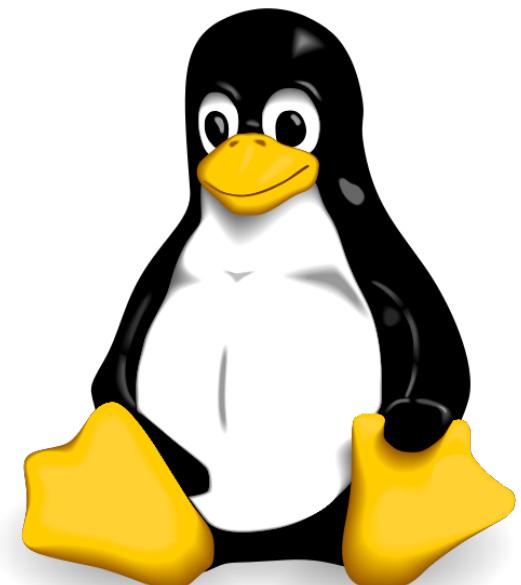
Système GNU/Linux



Richard Matthew Stallman (1953)
aka rms



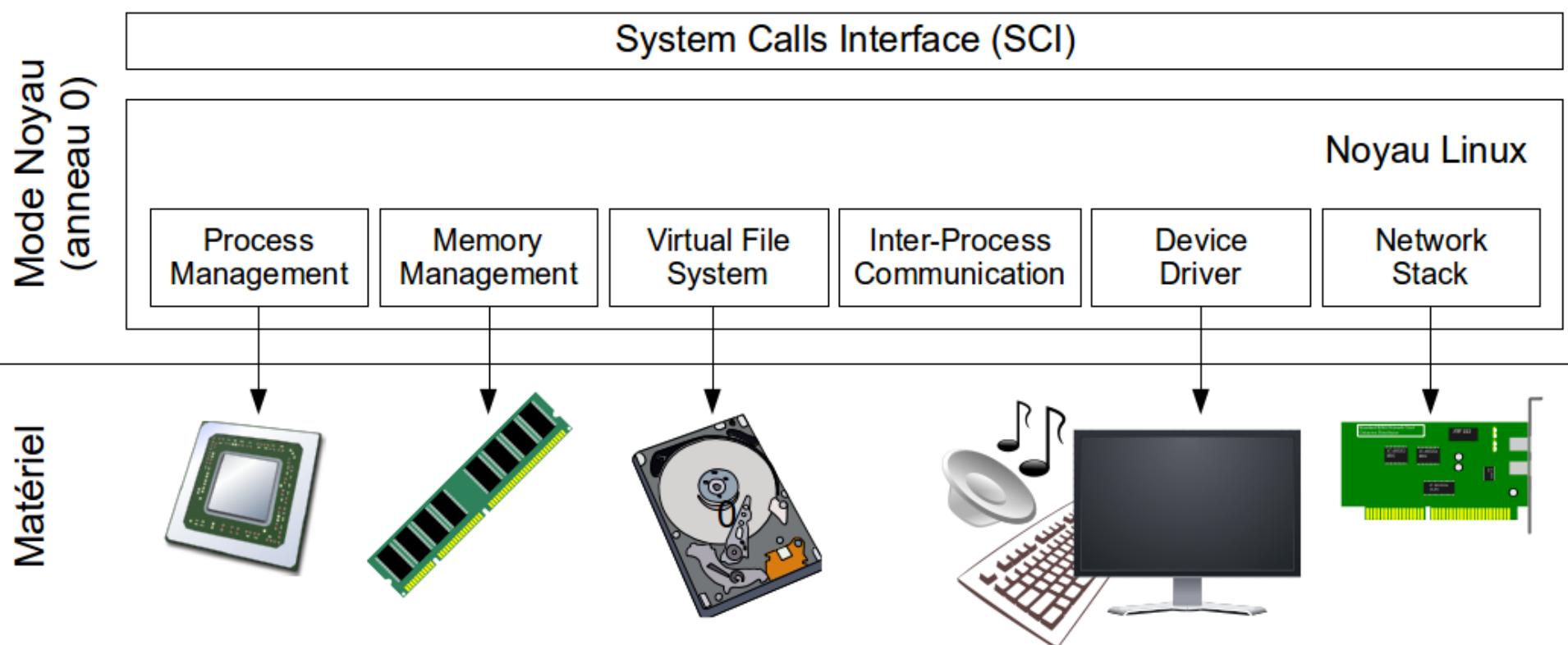
Linus Torvalds (1969)



Linux

Architecture (1/5)

- Le système Linux, structuré comme un noyau monolithique, est composé de 6 principaux sous-systèmes :



Linux

Architecture (2/5)

- ♦ Les 6 principaux sous-systèmes du noyau :
 - ♦ Gestion des processus (Process management - PM) qui est chargé de répartir de façon équitable les accès au processeur entre toutes les applications actives.
 - ♦ Gestion de la mémoire (Memory Management - MM) qui est chargé d'affecter à chaque programme une zone mémoire qui ne doit pas être lue ou modifiée par un autre processus.

Linux

Architecture (3/5)

- ♦ Les 6 principaux sous-systèmes du noyau :
 - ♦ Gestion des fichiers de haut niveau (Virtual File System - VFS) qui garantit une gestion correcte des fichiers et un contrôle des droits d'accès (ACL - Access Control List). Pour limiter la complexité liée aux nombreux systèmes de fichiers existants, il utilise des appels systèmes identiques quel que soit le système de fichiers choisi. Le noyau Linux détourne les appels standards vers les appels spécifiques au système de fichiers.

Linux

Architecture (4/5)

- ♦ Les 6 principaux sous-systèmes du noyau :
 - ♦ L'Inter-Processus Communication (IPC) permet à des applications de communiquer entre elles du fait qu'un processus ne peut accéder qu'à la zone mémoire qui lui a été allouée.
 - ♦ Le Device Driver (DD) gère les ressources matérielles avec des pilotes de périphériques (device drivers) et fournit aux programmes une interface uniforme pour l'accès à ces ressources.

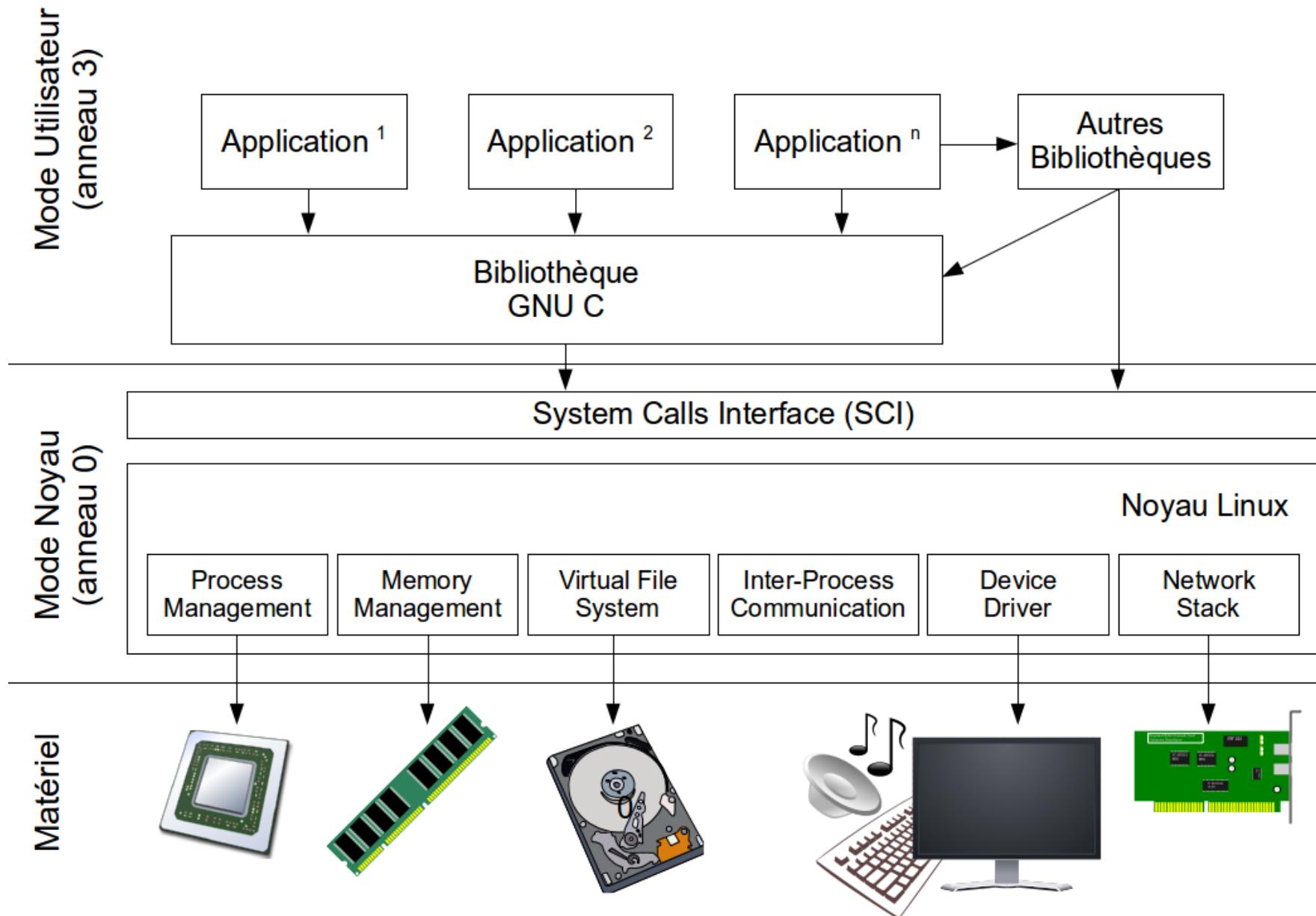
Linux

Architecture (5/5)

- ♦ Les 6 principaux sous-systèmes du noyau :
 - ◆ Le Network Stack (NET) permet de se connecter à d'autres systèmes à travers un réseau informatique. De nombreux périphériques matériels sont supportés. Plusieurs protocoles réseaux peuvent être utilisés comme IPX/SPX, TCP/IP version 4 ou 6...
- ♦ L'architecture du système Linux peut être ajustée autour du noyau grâce au « Modules chargeables de noyau » (Loadable Kernel Modules - LKM).

Linux

Modes d'exécutions (1/4)



Linux

Modes d'exécutions (2/4)

- Un processus s'exécute par défaut selon un « mode utilisateur » (user mode) dans lequel les actions effectuées par le programme sont volontairement restreintes dans le but de protéger la machine contre des actions malencontreuses parfois faites par le programmeur.
- Le système d'exploitation s'exécute dans un mode superviseur (kernel mode) pour lequel aucune restriction de droits n'existe.
- Le codage de ces 2 modes est réalisé au niveau du processeur dans le registre d'état de celui-ci.

Linux

Modes d'exécutions (3/4)

- ◆ Le noyau Linux est un noyau non préemptible e.i. qu'un processus passé en mode superviseur pour exécuter une fonction du noyau ne peut pas être arbitrairement suspendu et remplacé par un autre processus.
- ◆ En mode superviseur, un processus libère toujours de lui-même le processeur.

Linux

Modes d'exécutions (4/4)

- ◆ Puisqu'un processus exécute soit son propre code en « mode utilisateur » (user mode), soit le code du système en « mode superviseur » (kernel mode), 2 piles d'exécution lui ont été associées dans le système Linux :
 - ◆ 1 pile utilisateur utilisée en « mode utilisateur »,
 - ◆ 1 pile noyau utilisée en « mode superviseur »

Linux

Commutations de contexte (1/2)

- ◆ Lorsqu'un processus utilisateur demande l'exécution d'un routine du système par le biais d'un appel système, ce processus quitte son mode courant d'exécution (mode utilisateur) pour passer en mode d'exécution système (mode superviseur). Ce passage constitue une « commutation de contexte ».
- ◆ Une opération de sauvegarde du contexte utilisateur. Un contexte noyau est alors chargé.
- ◆ Dès que l'exécution de la fonction système est achevée, le processus repasse du « mode superviseur » au « mode utilisateur » avec une restauration du contexte utilisateur sauvegardé.

Linux

Commutations de contexte (2/2)

- ◆ 3 causes de commutations de contexte :
 - ◆ Quand un processus utilisateur appelle une fonction du système.
 - ◆ L'exécution par le processus utilisateur d'une opération illicite (division par 0, instruction machine interdite, violation mémoire...).
C'est la « trappe » ou « l'exception ».
L'exécution du processus utilisateur est alors arrêtée.
 - ◆ La prise en compte d'une interruption par le matériel (IRQ) et le système. Le processus est alors stoppé et l'exécution de la routine d'interruption associée à l'interruption survenue est exécutée en « mode superviseur ».

Linux

Gestion des interruptions matérielles et logicielles (1/2)

- ◆ Trappes et appels systèmes sont parfois qualifiés « d'interruptions logicielles » ou synchrones pour les opposer aux interruptions matérielles (IRQ).
- ◆ Sous le système Linux, chaque interruption matérielle ou logicielle est identifiée par un entier de 8 bits appelé « vecteur d'interruption » :
 - ◆ Valeurs de 0 à 31 correspondent aux interruptions non masquables et aux exceptions,
 - ◆ De 32 à 47 sont affectées aux interruptions masquables levées par les périphériques (IRQ).
 - ◆ De 48 à 255 peuvent être utilisées pour identifier d'autres types de trappes. L'entrée 128 (0x80) est utilisée pour implanter les appels systèmes.

Linux

Gestion des interruptions matérielles et logicielles (2/2)

- ◆ Nous avons donc une table de 256 entrées appelées « table des vecteurs d'interruptions (idt_table) placée en mémoire lors du boot de la machine.
- ◆ Chaque entrée de la table contient l'adresse en mémoire centrale du gestionnaire chargé de la prise en compte de l'interruption.
- ◆ Ainsi, l'unité de contrôle du processeur avant de commencer l'exécution d'une nouvelle instruction machine, vérifie si une interruption ne lui a pas été délivrée.

Questions (1/2)

- ◆ Quel est le rôle d'un OS ?
 - Compiler un programme et construire un exécutable.
 - Gérer les accès aux périphériques.
 - Partager la machine physique entre les différents programmes et bâtir une machine virtuelle plus accessible à l'utilisateur.
- ◆ Quels sont les étapes pour construire un exécutable ?

Questions (1/2)

- ◆ Quel est le rôle d'un OS ?
 - Compiler un programme et construire un exécutable.
 - Gérer les accès aux périphériques.
 - Partager la machine physique entre les différents programmes et bâtir une machine virtuelle plus accessible à l'utilisateur.
- ◆ Quels sont les étapes pour construire un exécutable ?
4 étapes :
 - Passage au pré-processeur (preprocessing)
 - Compilation en langage assembleur (compiling)
 - Conversion du langage assembleur en code machine (assembling)
 - Édition des liens (linking)

Questions (2/2)

- ◆ Une interruption est un signal dont le rôle est de :
 - permettre de séquencer l'exécution des programmes.
 - obliger le processeur à interrompre son traitement en cours pour un événement survenu sur la machine.
- ◆ Une commutation de contexte intervient :
 - à chaque changement de mode d'exécution.
 - sur l'occurrence d'une interruption.
 - lorsque le programme commet une erreur grave.
- ◆ Le mode superviseur :
 - est le mode d'exécution d'un programme utilisateur.
 - est un planificateur de tâches.
 - est le mode d'exécution d'un programme système.

Questions (2/2)

- ◆ Une interruption est un signal dont le rôle est de :
 - permettre de séquencer l'exécution des programmes.
 - obliger le processeur à interrompre son traitement en cours pour un événement survenu sur la machine.
- ◆ Une commutation de contexte intervient :
 - à chaque changement de mode d'exécution.
 - sur l'occurrence d'une interruption.
 - lorsque le programme commet une erreur grave.
- ◆ Le mode superviseur :
 - est le mode d'exécution d'un programme utilisateur.
 - est un planificateur de tâches.
 - est le mode d'exécution d'un programme système.

Module 2

Processus, threads et ordonnancement

Module 2 - Processus, threads et ordonnancement

- ◆ Notion de Processus
- ◆ Threads
- ◆ Ordonnancement

Notion de processus

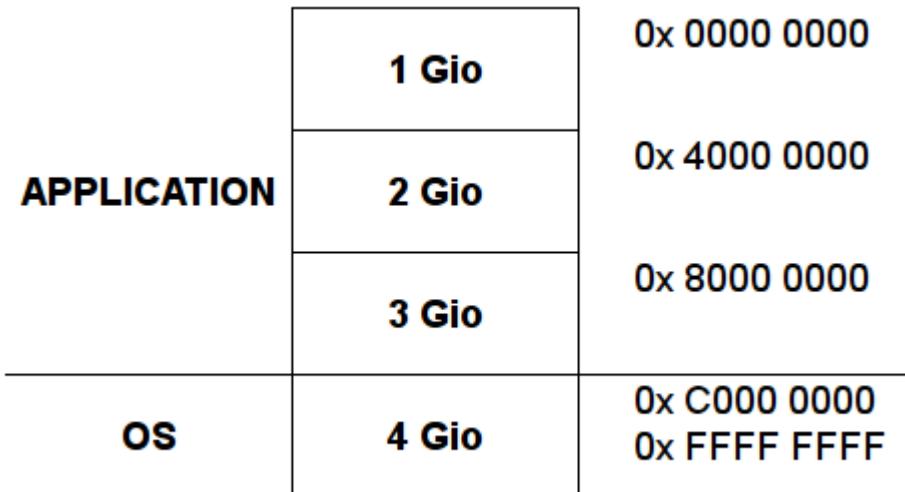
Rappel (1/3)

- ◆ Un processus est :
 - ◆ Un programme en cours d'exécution auquel est associé un environnement processeur (compteur ordinal, registre d'état, registres généraux) et un environnement mémoire appelés contexte du processus.
 - ◆ L'instance dynamique d'un programme et incarne le fil d'exécution de celui-ci. Il évolue dans un espace d'adressage protégé.

Notion de processus

Rappel (2/3)

- ♦ Un processus 32 bits ou 64 bits occupe un espace d'adressage virtuel de 4 Gio :



- ♦ La totalité des 4 Gio d'un processus 64 bits est affectée à l'espace utilisateur.

Notion de processus

Rappel (3/3)

- ♦ L'espace de l'adressage virtuel est subdivisé en régions :

Mémoire Utilisateur	Région De Code	Code des bibliothèques, des fonctions, de la fonction main() Code de la bibliothèque standard du C
	Région des Données Statiques	Variables (globales, statiques) pendant toute la durée du processus
	Région des Données Dynamiques	Contient les variables créés puis détruites plusieurs fois pendant la durée de vie du processus. L'adresse varie à chaque création.
	Région Des Arguments	Contient les arguments lors du lancement du processus
Mémoire Système		

Notion de processus

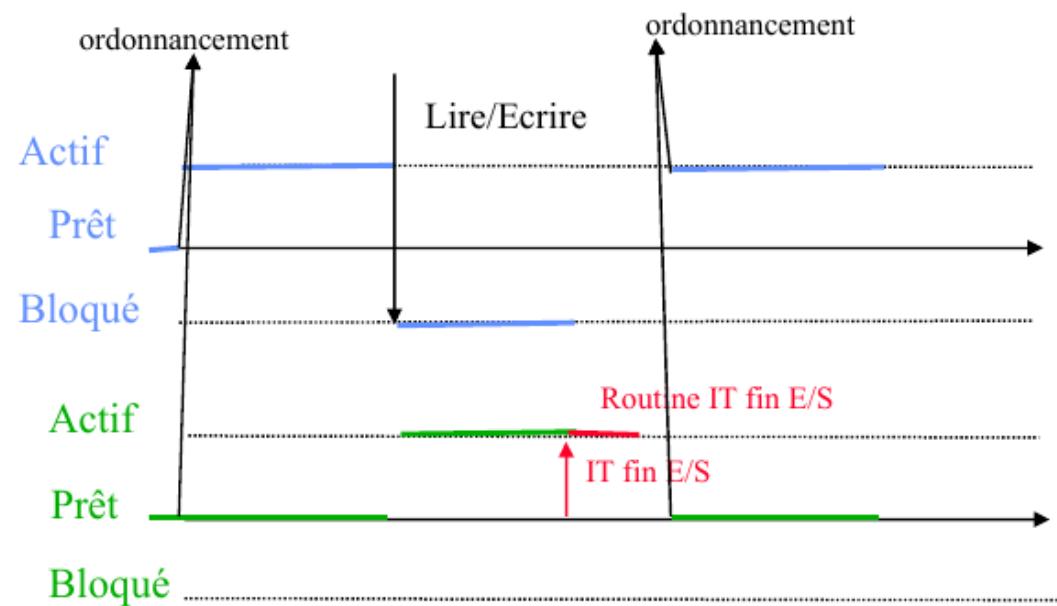
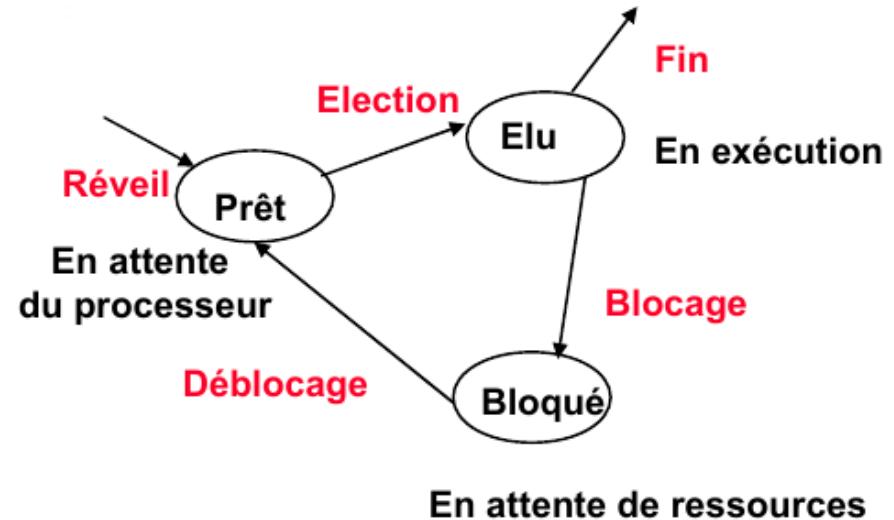
PID – Processus IDentifier

- ◆ Un processus se voit attribuer un PID par le système.
- ◆ La plus grande valeur du PID est 32768 (/proc/sys/kernel/pid_max).
- ◆ Lorsqu'un processus reçoit la valeur 32768, il prend alors le plus petit PID libéré par un processus mort.

Notions de processus

États des processus (1/2)

identificateur processus
état du processus
contexte pour reprise (registres et pointeurs, piles,..)
compteur instructions
pointeurs sur file d' attente et priorité(ordonnancement)
informations mémoire (limites et tables pages/segments)
informations de comptabilisation et sur les E/S, périphériques alloués, fichiers ouverts,..



Notions de processus

États des processus (2/2)

- ◆ **TASK_RUNNING**
État prêt ou élu.
- ◆ **TASK_INTERRUPTIBLE**
État bloqué.
- ◆ **TASK_UNINTERRUPTIBLE**
État bloqué. La réception d'un signal peut réveiller le processus.
- ◆ **TASK_ZOMBIE**
Exécution terminé mais le PCB du processus existe toujours du fait que son père n'a pas pris en compte sa terminaison.
- ◆ **TASK_STOPPED**
Suspendu par l'utilisateur via la réception d'un signal SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU.

Notions de processus

Bloc de contrôle de processus (PCB)

identificateur processus
état du processus
contexte pour reprise (registres et pointeurs, piles,...)
compteur instructions
pointeurs sur file d' attente et priorité(ordonnancement)
informations mémoire (limites et tables pages/segments)
informations de comptabilisation et sur les E/S, périphériques alloués, fichiers ouverts,..

- Le chargeur a pour rôle de monter en mémoire centrale le code et les données du programme à exécuter.
- En même temps que ce chargement, Linux crée une structure de description du processus associé au programme exécutable : le PCB (Process Control Block)
- Le PCB permet de sauvegarder et de restaurer le contexte mémoire et le contexte processeur pendant les opérations de commutations de contexte.

Notions de processus

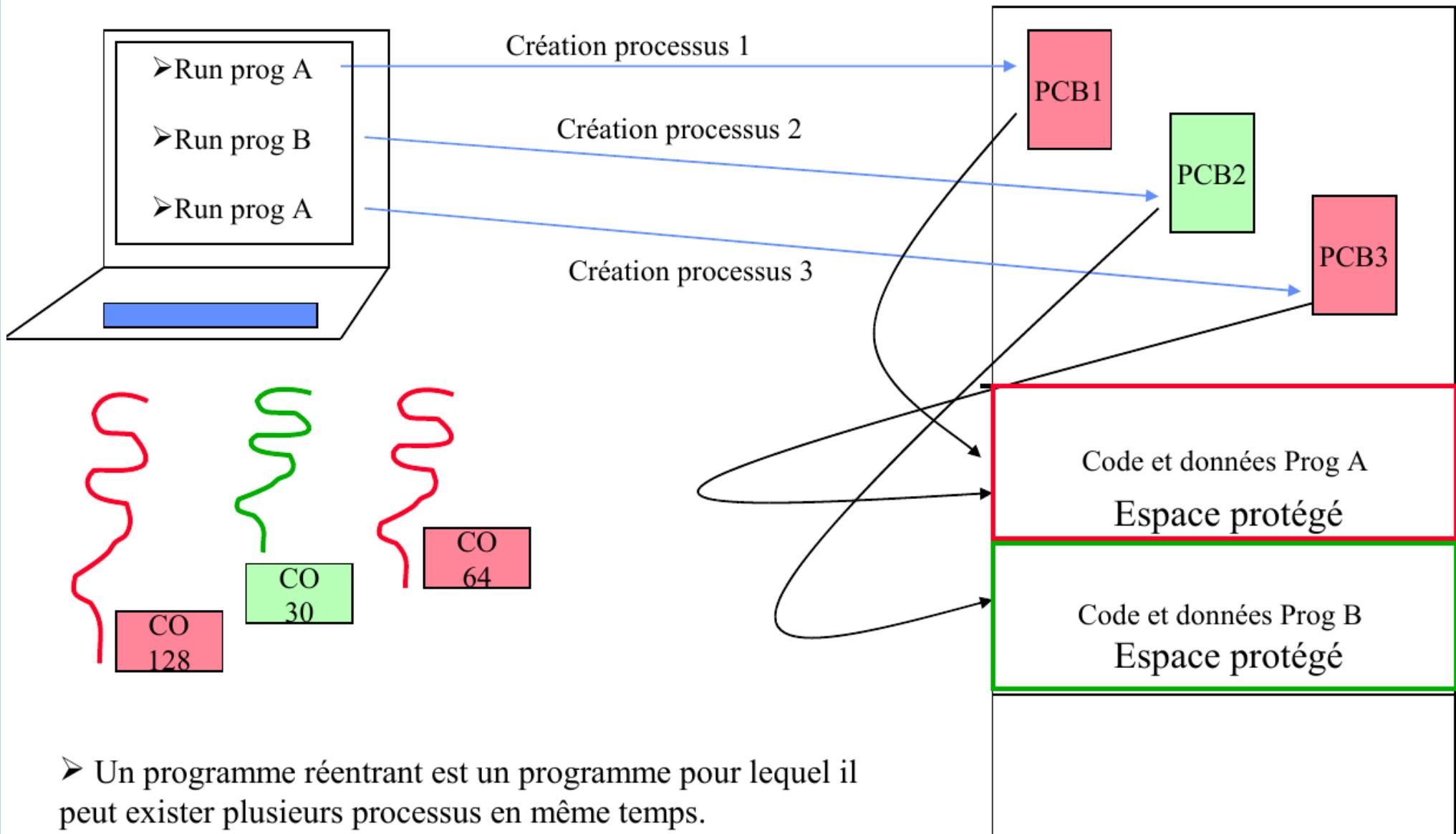
PCB – struct task_struct

- Chaque processus est représenté par un PCB alloué dynamiquement par le système au moment de la création du processus. Structure task_struct définie dans le fichier /usr/src/linux-headers-\$(uname -r)/include/linux/sched.h

```
struct task_struct {  
    volatile long      state;           /* -1 unrunnable, 0 runnable, >0 stopped */  
    long               counter;  
    long               priority;  
    unsigned          long signal;  
    unsigned          long blocked;     /* bitmap of masked signals */  
    unsigned          long flags;       /* per process flags, defined below */  
    int                errno;  
    long               debugreg[8];      /* Hardware debugging registers */  
    struct exec_domain *exec_domain;  
/* various fields */  
    struct linux_binfmt *binfmt;  
    struct task_struct *next_task, *prev_task;  
    struct task_struct *next_run, *prev_run;
```

Notions de processus

Programme ré-entrant



Notion de processus

Linux (1/7)

- ◆ Tout processus (père) peut créer un autre processus (fils). Primitive *fork()* :

```
#include <unistd.h>
pid_t fork(void) ;
```

- ◆ En cas d'échec la fonction *fork()* retourne -1 et la variable *errno* est positionnée avec les valeurs suivantes :
 - ◆ EAGAIN
Le nombre maximal de processus a été atteint pour l'utilisateur courant ou pour le système.
 - ◆ ENOMEM
Le noyau n'a pas pu allouer assez de mémoire pour créer un nouveau processus.

Notion de processus

Linux (2/7)

- ◆ Les primitives suivantes permettent à un processus respectivement de connaître la valeur de son PID, du PPID, l'identifiant de l'utilisateur qui l'a créé et le numéro du groupe auquel il appartient :

```
#include <unistd.h>
pid_t getpid(void) ;      //pid du processus appelant
pid_t getppid(void) ;     //ppid du processus appelant
uid_t getuid(void) ;      //uid du processus appelant
gid_t getgid(void) ;      //gid du processus appelant
```

Notion de processus

Linux (3/7)

- ◆ Un processus termine normalement son exécution en achevant l'exécution du code qui lui est associé :

```
#include <stdlib.h>
void exit(int status) ;
```

- ◆ La valeur de status est entre 0 et 255.

Notion de processus

Linux (4/7)

- ◆ Le père peut se mettre en attente de la mort de l'un de ses fils :

```
#include <sys/wait.h>
pid_t wait (int *status);
```

- ◆ L'exécution du processus père est suspendue jusqu'à ce qu'un processus fils se termine. Si un processus fils est déjà dans l'état zombie au moment de l'appel, la fonction retourne immédiatement le résultat (le pid du fils terminé et le code retour de celui-ci dans la variable status) :

```
#include <sys/wait.h>
pid_t waitpid (pid_t pid, int *status, int options);
```

Notion de processus

Linux (5/7)

- ◆ Si la variable pid est strictement positive alors le système suspend le processus père jusqu'à ce qu'un processus fils, dont la valeur du PID est égal au pid, se termine.
- ◆ Si pid est nul alors le système suspend le processus père jusqu'à la mort de n'importe quel fils appartenant au même groupe que le père.
- ◆ Si pid vaut -1 alors le système suspend le processus père jusqu'à la mort de n'importe quel de ses fils.
- ◆ Si pid est strictement inférieur à -1 alors le système suspend le processus père jusqu'à la mort de n'importe quel de ses fils dont le numéro de groupe est égal à pid.

Notion de processus Linux (6/7)

- ◆ L'interprétation de la variable *status* se fait à l'aide de macros définies dans <sys/wait.h> :

WIFEXITED(status)

est vrai si le fils s'est terminé par un appel à la primitive *exit()*

WEXITSTATUS(status)

récupère le code passé par le fils au moment de la terminaison.

WIFSIGNALED(status)

permet de savoir que le fils s'est terminé à cause d'un signal

WIFSTOPPED(status)

indique que le fils est stoppé temporairement

Notion de processus Linux (7/7)

- La variable option peut prendre la valeur WNOHANG qui provoque un retour immédiat de la primitive si aucun fils n'est encore terminé.
- La primitive renvoie -1 si échec La variable *errno* peut prendre les valeurs suivantes :

ECHILD

processus spécifié par pid n'existe pas.

EFAULT

variable *status* contient une adresse invalide.

EINTR

Interruption de l'appel système par un signal.

Notion de processus

Les primitives de recouvrement

- ◆ La « famille exec » est un ensemble de 6 primitives (`execl`, `execlp`, `execle`, `execv`, `execvp`, `execve`) qui permet à un processus de charger en mémoire un nouveau code exécutable.
- ◆ Des données peuvent être passées au nouveau code via les arguments de la primitive `exec` qui les récupère dans le tableau `argv[]`.
- ◆ Pour rappel, la forme générale d'un programme principale en langage C est :

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

Notion de processus - primitives exec (juste un petit rappel)

- Pour rappel, la forme générale d'un programme principale en langage C est :

int main(int argc, char *argv[], char *arge[]) ;

- argc est le nombre total de paramètres transmis au programme.
- argv est un tableau comprenant les différents paramètres passés au programme.
- arge est une liste de pointeurs permettant d'accéder à l'environnement d'exécution du processus.

Notion de processus

la « famille exec »

- ◆ Les prototypes des 6 primitives exec :

```
#include <unistd.h>
extern char **environ;

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

- ◆ La sortie des fonctions exec() n'intervient que si une erreur s'est produite. La valeur de retour est -1, et errno contient le code d'erreur.

TP : Création d'un processus fils (1/4)

Créer le fichier 01a_perefils.c

Effectuer la compilation

Déterminer les fichiers d'en-tête (headers) à inclure

*Fonction main() sans passage d'arguments et avec un code retour
Appel de la fonction fork()*

SI le retour de la fonction fork() = 0 alors

Afficher "Je suis le fils, mon pid est x. Mon ppid est x

SINON

Afficher "Je suis le père, mon pid est x. Le pid de mon fils est x

Fin de SI

Valeur du code retour = 0

Fin de la fonction main()

TP : Création d'un processus fils

1ère réponse (1/4)

```
// 01a_perefils.c
// gcc perefils.c -o perefils

#include <stdio.h>
#include <unistd.h>

// Fonction main
int main(void)
{
    pid_t pid = fork();

    if(pid == 0)
    {
        printf("Je suis le fils, mon pid est %d. Mon ppid est %d\n", getpid(), getppid());
    }
    else
    {
        printf("Je suis le père, mon pid est %d. Le pid de mon fils est %d\n", getpid(), pid);
    }

    return 0;
}
```

TP : Création d'un processus fils

2ème réponse (1/4)

```
// 01a_perefils.c
// gcc perefils.c -o perefils

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

// Fonction main
int main(void)
{
    pid_t pid = fork();
    int status ;
    if(pid == 0)
    {
        printf("Je suis le fils, mon pid est %d. Mon ppid est %d\n", getpid(), getppid());
    }
    else
    {
        printf("Je suis le père, mon pid est %d. Le pid de mon fils est %d\n", getpid(), pid);
    }
    wait(&status) ;
    return 0;
}
```

TP : Création d'un processus fils (2/4)

Créer un fichier 01b_perefils.c à partir de 01a_perefisl.c

Effectuer la compilation

Déterminer les fichiers d'en-tête (headers) à inclure

Fonction main() sans passage d'arguments et avec un code retour

Appel de la fonction fork()

SI le retour de la fonction fork() = 0 ALORS

Afficher "Je suis le fils, mon pid est x. Mon ppid est x

SINON SI retour de la fonction fork() = -1

Afficher "Je suis le père, mon pid est x. Le pid de mon fils est x

SINON

erreur et code retour 1

Fin de SI

Synchronisation avec le père

Valeur du code retour = 0

Fin de la fonction main()

TP : Création d'un processus fils

Réponse (2/4)

```
// 01b_perefils.c

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

// Fonction main
int main(void)
{
    pid_t pid = fork();
    int status;

    if(pid == 0)
    {
        printf("Je suis le fils, mon pid est %d. Mon ppid est %d\n", getpid(), getppid());
    }
    else if(pid != -1)
    {
        printf("Je suis le père, mon pid est %d. Le pid de mon fils est %d\n", getpid(), pid);
    }
    else
    {
        perror("erreur");
        return 1;
    }
    waitpid(-1, &status, 0);
    return 0;
}
```

TP : Primitives de recouvrement (3/4)

Créer le fichier 03a_exec.c et effectuer la compilation

Déterminer les fichiers d'en-tête (headers) à inclure

Fonction main sans passage d'arguments et avec un code retour

Appel de la fonction fork()

SI le retour de la fonction fork() = -1 ALORS

Afficher « Erreur de création de processus »

SINON SI retour de la fonction fork() = 0 ALORS

Afficher « je suis le fils, mon PID est x, mon PPID (PID de mon père) est x »

Recouvrement par le processus ls -l / avec la primitives execp

SINON

Afficher « Je suis le père, mon PID est x »

Afficher « Le PID de mon fils est x »

Attendre le fils

Fin de SI

Fin de la fonction main()

TP : Primitives de recouvrement

Réponse (3/4)

```
// 03a_exec.c
```

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void)
{
    pid_t pid = fork();
    int status;

    if(pid == -1)
    {
        printf("Erreur création de processus");
    }
    else if(pid == 0)
    {
        printf("je suis le fils, mon PID est %d, mon PPID (PID de mon père) est %d\n", getpid(), getppid());
        execvp("ls", "ls", "-l", "/", NULL);
    }
    else
    {
        printf("Je suis le père, mon PID est %d\n", getpid());
        printf("Le PID de mon fils est %d\n", pid);
        wait(&status);
        return(0);
    }
}
```

TP : Primitives de recouvrement (4/4) - Partie 1

Créer le fichier 03b_exec.c et le compiler.

Déterminer les fichiers d'en-tête (headers) à inclure

Fonction main sans passage d'arguments et avec un code retour

Déclaration des variables nécessaires

Message "Donnez la valeur de i : " et saisie de l'entier i

Message "Donnez la valeur de j : " et saisie de l'entier j

Appel de la fonction fork()

SI le retour de la fonction fork() = 0 ALORS

convertir i en chaîne de caractères de 11 éléments dans la variable conv1

convertir j en chaîne de caractères de 11 éléments dans la variable conv2

recouvrement par le processus 03b_calculer avec les arguments conv1 et conv2

SINON

Attendre le fils

FiN DE SI

Fin de la fonction main()

TP : Primitives de recouvrement

Réponse (4/4) - Partie 1

```
//03b_exec.c
// Compilation : gcc 03b_exec.c -o 03b_exec

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

void main(void)
{
    pid_t pid;
    int i,j,status;
    char conv1[10], conv2[10];

    printf("Donnez la valeur de i : ");
    scanf("%d",&i);
    printf("Donnez la valeur de j : ");
    scanf("%d",&j);

    pid=fork();

    if(pid == 0)
    {
        sprintf(conv1, "%d", i);
        sprintf(conv2, "%d", j);
        execl("./calculer", "Calcul", conv1, conv2, NULL);
    }
    else
    {
        wait(&status);
    }
}
```

TP : Primitives de recouvrement (4/4) - Partie 2

Créer le fichier 03b_calculer.c et le compiler.

Déterminer les fichiers d'en-tête (headers) à inclure

Fonction main avec le passage d'arguments et sans code retour

Déclaration des variables nécessaires

SI le nombre d'arguments n'est pas correct ALORS

Afficher « Erreur »

Sortir du programme

FIN DE SI

*Faire l'addition des 2 arguments et stocker le résultat dans la variable sum
(Attention ! Ne pas oublier de convertir les types...)*

Afficher « La somme est x »

Fin de la fonction main()

TP : Primitives de recouvrement

Réponse (4/4) - Partie 2

```
//03b_calculer.c
// Compilation : gcc 03b_calculer.c -o calculer

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

void main(int argc, char *argv[])
{
    int sum;

    if(argc != 3)
    {
        printf("Erreur !\n");
        exit(1);
    }

    sum = atoi(argv[1]) + atoi(argv[2]);
    printf("La somme est %d\n",sum);
    exit(0);
}
```

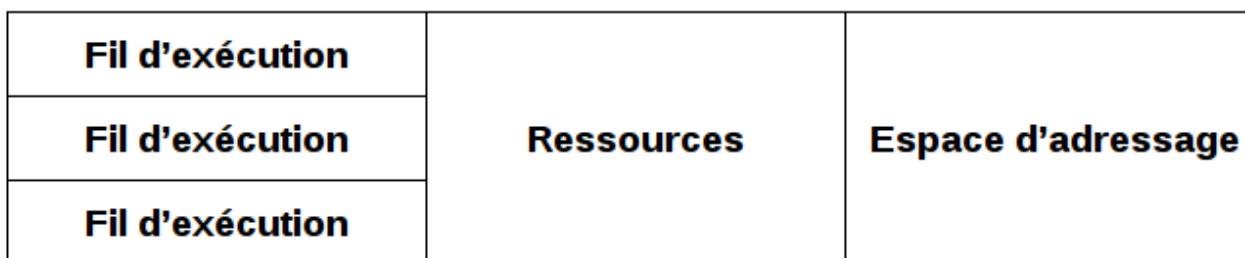
Threads (processus légers, fils d'exécution)

- ◆ Un processus est constitué d'un espace d'adressage avec un seul fil d'exécution.
- ◆ L'extension du modèle processus consiste à admettre plusieurs fils d'exécution indépendants dans un même espace d'adressage.
- ◆ Les threads sont caractérisés par une valeur de compteur ordinal propre et une pile d'exécution privée.
- ◆ L'entité contenant les différents fils d'exécutions est appelée processus.

Note : Dans un CPU, le compteur ordinal est le registre qui contient l'adresse mémoire de l'instruction en cours d'exécution. Une fois l'instruction chargée, il est automatiquement incrémenté pour pointer l'instruction suivante.

Threads (processus légers, fils d'exécution)

- ◆ Un processus est constitué d'un espace d'adressage avec un seul fil d'exécution.



Processus à threads

- ◆ Avantages :
La notion de processus léger est un allègement des opérations de commutations de contexte.
L'opération de création d'un nouveau fil d'exécution est aussi allégé puisqu'elle nécessite plus la duplication complète de l'espace d'adressage du processus père.

Threads (processus légers, fils d'exécution)

- ◆ Cependant...
Puisque les threads au sein d'un même processus partage le même espace d'adressage, il s'ensuit des problèmes de partage de ressources plus importants.
- ◆ Les threads peuvent être implémentés à 2 niveaux différents :
 - ◆ Niveau utilisateur
 - ◆ Niveau noyau

Threads niveau utilisateur

- ◆ Les threads implémentés dans l'espace utilisateur, le noyau Linux les ignore. Il ordonne des processus classiques composés d'un seul fil d'exécution.
- ◆ Une bibliothèque système (POSIX Threads ou appelés pthreads) gère l'interface avec le noyau en prenant en charge la gestion des threads en les cachant à ce dernier. Elle est donc responsable de commuter les threads au sein d'un même processus.
- ◆ Avantage. Cette approche allège les commutations entre threads d'un même processus puisque celles-ci s'effectuent au niveau utilisateur sans passer par le mode noyau.
- ◆ Inconvénient. Au sein d'un même processus, un thread peut monopoliser le processeur pour lui seul. Un thread bloqué au sein d'un processus (suite à une demande de ressource ne pouvant être immédiatement satisfaite) bloque l'ensemble des threads de ce processus.

Threads niveau noyau

- ◆ Le noyau Linux connaît l'existence des threads au sein d'un processus. Il attribue le processeur à chacun des threads de manière indépendante.
- ◆ Chaque commutation de contexte est géré par le noyau. Ceci a un coût.
- ◆ En revanche, cette approche permet d'éviter le blocage de tout un processus lorsque l'un des threads est bloqué.

Primitives de gestion des threads (1/4)

- ◆ Elles sont définies dans la bibliothèque Linux Threads créée par Xavier Leroy (INRIA) et sont conformes à POSIX.
- ◆ Linux Threads étant indépendante de la GlibC, il faut inclure le fichier d'en-tête pthread.h dans les fichiers sources.
- ◆ L'utilisation de l'option -lpthread lors de l'édition des liens est nécessaire pour spécifier l'utilisation de cette bibliothèque.
- ◆ Chaque thread est identifié de manière unique au sein d'un processus par un type pthread_t.
- ◆ La primitive pthread_self() permet à un thread de connaître son propre identifiant.

Primitives de gestion des threads (2/4)

- ◆ `pthread_create()` permet de créer un thread au sein d'un processus. Elle retourne 0 si succès sinon une valeur négative :

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
                  void *(*start_routine)(void *), void *arg);
```

- ◆ `*thread` est l'identificateur du nouveau fil d'exécution.
- ◆ Le thread est attaché à l'exécution de la routine (`*start_routine`).
- ◆ `*attr` correspond aux attributs associés au thread. Valeur `NULL` pour hériter des attributs standards.
- ◆ `*arg` correspond à un argument passé au thread pour l'exécution de la fonction (`*start_routine`)

Primitives de gestion des threads (3/4)

- ◆ `pthread_exit(void *ret)` met fin au thread et retourne le paramètre `ret`.
- ◆ Cette valeur peut être récupérée par un autre thread qui fait un appel à la primitive :

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- ◆ La valeur `thread` correspond à l'identifiant du thread attendu.
- ◆ `value_ptr` correspond à la valeur `ret` renvoyée lors de la terminaison.
- ◆ Le processus qui fait appel à cette primitive est suspendu jusqu'à ce que le thread attendu soit terminé.
- ◆ Le paramètre `ret` contient alors la valeur passée par le thread au moment de l'exécution de la primitive `pthread_exit()`.

attributs d'un thread

(4/4)

- ◆ Chaque thread est doté des attributs suivants :
 - ◆ L'adresse de départ et la taille de la pile,
 - ◆ La politique d'ordonnancement,
 - ◆ La priorité,
 - ◆ Son attachement ou bien son détachement.
Un thread détaché se termine immédiatement sans pouvoir être pris en compte par la primitive `pthread_join()`.
- ◆ Lors de la création d'un nouveau thread par un appel de `pthread_create()`, les attributs de celui-ci sont fixés par l'argument `pthread_attr_t *attr`. Si les attributs par défaut sont suffisants, la valeur de ce paramètre est `NULL`. Sinon, il faut initialiser une structure de type `pthread_attr_t` en invoquant la primitive `pthread_attr_init()`...

TP : Création de threads exécutant la fonction f_thread()

Créer le fichier 04a_threads.c

Effectuer la compilation

Déterminer les fichiers d'en-tête (headers) à inclure

Définir les attributs par défaut des threads à NULL

Déclarer les variables nécessaires

Fonction f_thread(entier i) sans retour

Afficher « je suis le Xème pthread d'identité PID, pthread_self() »

Fin de la fonction f_thread()

Fonction main() sans passage d'arguments et avec un code retour

Pour i=0, i<3, i++

Appel de pthread_create()

Fin du Pour

Afficher « Je suis le thread initial PID, pthread_self() »

Appel de la fonction pthread_join()

Valeur du code retour = 0

Fin de la fonction main()

Réponse : Cr ation de threads ex cutant la fonction f_thread()

```
// 04a_threads.c
// Compilation : gcc 04a_threads.c -o 04a_threads -lpthread

#include <stdio.h>
#include </usr/include/pthread.h>

#define pthread_attr_default NULL

pthread_t pthread_id[3];

void f_thread(int i) {
    printf("je suis le %d me pthread d'identit  %d.%d\n",i, getpid(), pthread_self());
}

int main(void) {
    int i;

    for(i=0;i<3;i++)
        pthread_create(&pthread_id[i], pthread_attr_default, (void *)f_thread, &i);

    printf("Je suis le thread initial %d.%d\n",getpid(),pthread_self());
    pthread_join(*pthread_id,NULL);

    return(0);
}
```

TP : Création de threads exécutant la fx print_message()

Créer le fichier 04b_threads.c et effectuer la compilation

Déterminer les fichiers d'en-tête (headers) à inclure

Déclarer les variables nécessaires

Fonction print_message(message) sans retour

Afficher la variable message

Fin de la fonction print_message()

Fonction main() sans passage d'arguments et avec un code retour

Déclarer message1= « Thread 1 »

Déclarer message2= « Thread 2 »

Créer 1 thread pour lancer la fx print_message avec argument message1

Créer 1 thread pour lancer la fx print_message avec argument message2

Appel de pthread_join pour chaque thread

Afficher la valeur retour de chaque thread

Valeur du code retour = 0

Fin de la fonction main()

Réponse : Cr ation de threads ex cutant la fx print_message()

```
// 04b_threads.c
// Compilation : gcc 04b_threads.c -o 04b_threads -lpthread

#include <stdio.h>
#include <stdlib.h>
#include </usr/include/pthread.h>

void *print_message( void *ptr ){
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}

int main(void){
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int iret1, iret2;

    iret1 = pthread_create( &thread1, NULL, print_message, (void*) message1);
    iret2 = pthread_create( &thread2, NULL, print_message, (void*) message2);

    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2);

    return 0;
}
```

TP : Partage d'une variable entre threads d'un même processus

Créer le fichier 05a_threads.c

Effectuer la compilation

Déterminer les fichiers d'en-tête (headers) à inclure

Déclarer la variable i de type entier

Fonction somme() sans arguments et sortie

i+=10 et afficher « Fils et la valeur de i »

i+=20 et afficher « Fils et la valeur de i »

Fin de la fonction somme()

Fonction main() sans passage d'arguments et avec un code retour

Déclaration de variables nécessaires

Créer 1 thread pour lancer la fx somme

i+=1000 et afficher « Père et la valeur de i »

i+=2000 et afficher « Père et la valeur de i »

Appel de la fonction pthread_join()

Valeur du code retour = 0

Fin de la fonction main()

Reponse : Partage d'une variable entre threads d'un même processus

```
// 05a_threads.c
// Compilation : gcc 05a_threads.c -o 05a_threads -lpthread

#include <stdio.h>
#include <stdlib.h>
#include </usr/include/pthread.h>

int i;

void somme(void) {
    i = i+10;
    printf("Fils %d\n",i);
    i = i+20;
    printf("Fils %d\n",i);
}

int main(void) {
    pthread_t num_thread;
    i = 0;

    if(pthread_create(&num_thread, NULL, (void *(*)(())somme, NULL) == -1)
        perror("Problème pthread_create\n");
    i = i+1000;
    printf("Proc. %d\n", i);
    i = i+2000;
    printf("Proc. %d\n", i);
    pthread_join(num_thread, NULL);
    Return 0;
}
```

TP : Héritages de variable entre processus lourds

Créer le fichier 05b_proc.c et effectuer la compilation

Déterminer les fichiers d'en-tête (headers) à inclure

Déclarer la variable i de type entier

Fonction somme() sans arguments et sortie

i+=10 et afficher « Fils et la valeur de i »

i+=20 et afficher « Fils et la valeur de i »

Fin de la fonction print_message()

Fonction main() sans passage d'arguments et avec un code retour

Déclaration de variables nécessaires

Appel de la fonction fork()

Si retour de la fonction fork() = 0 ALORS

Appel de la fonction somme()

SINON

i+=1000 et afficher « Père et la valeur de i »

i+=2000 et afficher « Père et la valeur de i »

Attendre le fils

FIN DE SI

Valeur du code retour = 0

Fin de la fonction main()

Réponse : Héritages de variable entre processus lourds

```
// 05b_proc.c
// Compilation : gcc 05b_proc.c -o
// 05b_proc
#include <stdio.h>

int i;

void somme(void) {
    i += 10;
    printf("Proc. fils %d\n", i);
    i += 20;
    printf("Proc. fils %d\n", i);
}
```

```
int main(void) {
    int pid;
    i=0;
    pid=fork();
    if(pid == 0) {
        somme();
    }
    else {
        i=i+1000;
        printf("Proc. père : %d\n", i);
        i=i+2000;
        printf("Proc. père : %d\n", i);
        wait();
    }
    return 0;
}
```

TP : Processus zombie

Créer le fichier 06_zombie.c

Effectuer la compilation

Déterminer les fichiers d'en-tête (headers) à inclure

Fonction main() sans passage d'arguments et avec un code retour

Déclaration de variables nécessaires

Appel de la fonction fork()

Si retour de la fonction fork() = 0 ALORS

Afficher « Nous sommes dans le processus parent. Attendre une minute... »

Endormir le processus pour une durée de 1 minute

SINON

Afficher « Nous sommes dans le processus fils. Sortie immédiate. »

Sortir sans code retour

FIN DE SI

Valeur du code retour = 0

Fin de la fonction main()

Réponse : Processus zombie

```
// 06_zombie.c

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    pid_t child_pid;
    child_pid = fork();

    if (child_pid > 0) {
        printf("Nous sommes dans le processus parent. Attente d'une minute...\n");
        sleep(60);
    }
    else
    {
        printf("Nous sommes dans le processus fils. Sortie immédiate.\n");
        exit();
    }

    Return(0);
}
```

Ordonnancement (1/3)

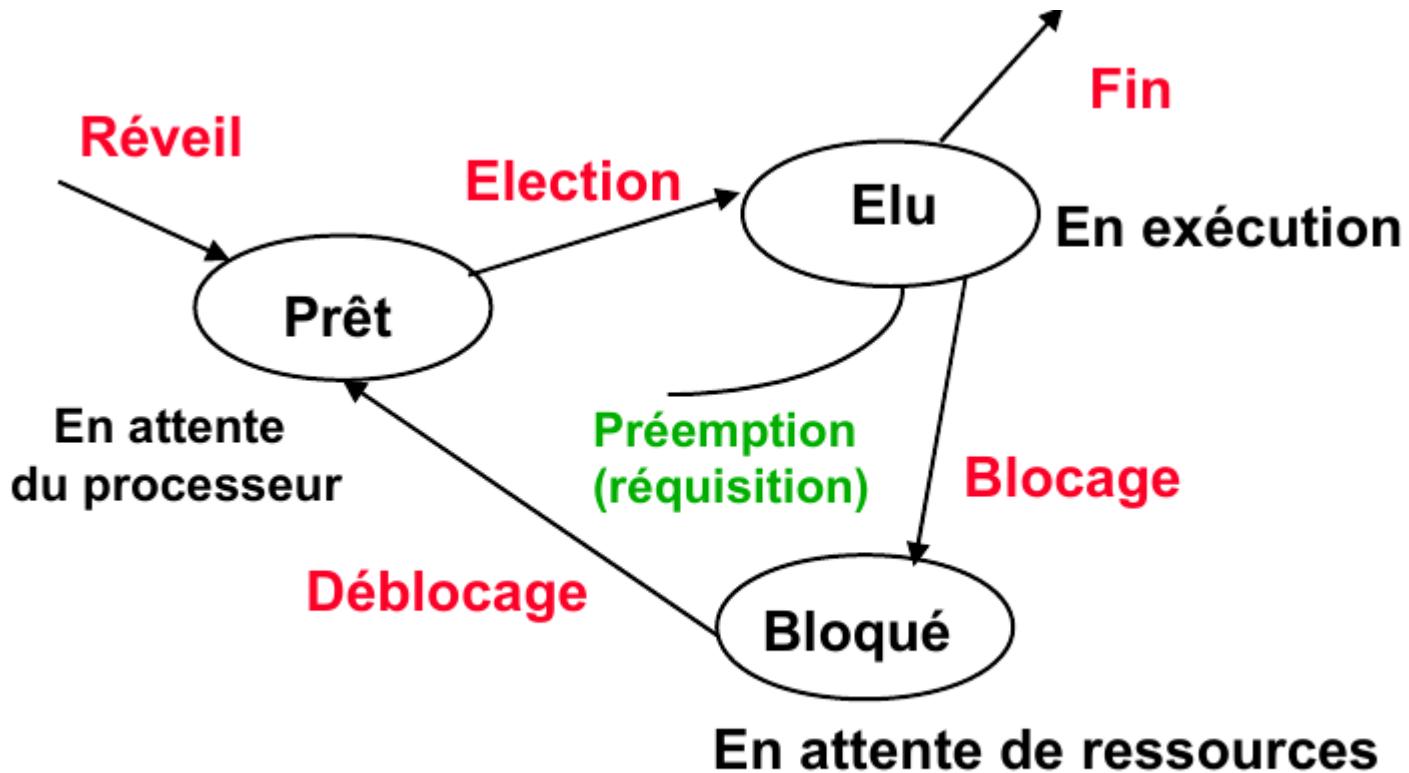
- ◆ Le rôle de l'ordonnancement est de gérer le partage entre le processeur et les processus qui sont dans l'état prêt.
- ◆ L'attribution du processeur à un processus prêt constitue l'opération d'élection. Le processus passe alors dans l'état élu.
- ◆ La politique d'ordonnancement mise en œuvre par l'OS peut ou non autoriser l'opération réquisition du processeur (le processeur est retiré au processus élu et ce dernier passe dans l'état prêt). Cette réquisition porte le nom de préemption.

Ordonnancement (2/3)

- ◆ Si l'opération de réquisition est autorisé ou non, l'ordonnancement est qualifié d'ordonnancement préemptif ou non préemptif.
- ◆ Si non préemptif,
la transition de l'état élu vers l'état prêt est interdite. Un processus quitte le processeur s'il a terminé son exécution ou s'il se bloque.
- ◆ Si préemptif,
la transition de l'état élu vers l'état prêt est autorisée. Un processus quitte le processeur s'il a terminé son exécution, s'il se bloque ou si le processeur est réquisitionné.

Ordonnancement (3/3)

- ♦ En résumé :



Principaux algorithmes d'ordonnancement

- ❖ Les politiques d'ordonnancement les plus courantes :
 - ◆ Politique premier arrivé, premier servi
Les processus sont élus selon l'ordre de leur arrivé. Il n'y a pas de réquisition e.i. il s'exécute jusqu'à ce qu'il soit terminé ou jusqu'à ce qu'il se bloque de lui-même.
 - ◆ Politique par priorité
Chaque processus possède une priorité. Le processus élu est le processus prêt de plus forte priorité.
 - ◆ Politique par tourniquet (Round Robin)
Le temps est découpé en tranches (quantums de temps). Lorsqu'un processus est élu, il s'exécute au plus dans un quantum de temps.
Si il n'a pas terminé son exécution, il est préempté et réintègre la file de processus prêts.

L'ordonnancement sous Linux

- ◆ Linux implémente 3 politiques d'ordonnancement conformément à la norme POSIX 1003.4.
- ◆ Les politiques SCHED_FIFO et SCHED_RR sont destinées aux processus « temps réel ». Ils sont toujours plus prioritaires que les processus classiques
- ◆ Quand à la politique SCHED_OTHER, elle est destinée aux processus classiques.
- ◆ La politique d'ordonnancement appliquée à un processus est codée dans le champ *policy* de son PCB.
- ◆ L'ordonnancement réalisé par le noyau Linux est découpé en périodes. Au début de chaque période, le système calcule le quantum de temps attribués à chaque processus.

L'ordonnancement des processus temps réel

- ◆ Ils sont qualifiés par une priorité fixe (valeur entre 1 et 99) défini par le paramètre `rt_priority` du PCB et sont ordonnancé soit par `SCHED_FIFO` ou soit par `SCHED_RR`.
- ◆ `SCHED_FIFO` est une politique préemptive « Premier arrivé, premier servi » entre processus de même priorité.
- ◆ `SCHED_RR` est une politique Round Robin (tourniquet) à quantum de temps (paramètre `counter` dans le PCB) entre processus de même priorité.

L'ordonnancement des processus classiques (1/2)

- ◆ SCHED_OTHER est destiné au processus classiques
- ◆ Il sont qualifiés par une priorité dynamique qui varie en fonction de l'utilisation faite par le processus des ressources de la machine et notamment du processeur.
- ◆ Cette dynamique représente le quantum alloué au processus.
- ◆ Un processus devient moins prioritaire que les processus qui ne sont pas encore exécutés. Ce principe est appelé « extinction de priorité »
- ◆ Un processus fils hérite lors de sa création du quantum de base de son père et de la moitié du nombre de ticks horloge restant de son père.

L'ordonnancement des processus classiques (2/2)

- ◆ D'autres politiques existent :
 - ◆ SCHED_BATCH pour une exécution de style traitement par lot des processus.
 - ◆ SCHED_IDLE pour l'exécution de tâches de très faible priorité en arrière-plan.

Les primitives liées à l'ordonnancement (1/7)

- ◆ Les prototypes sont déclarés dans sched.h
- ◆ `int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);`
Modifie la politique d'ordonnancement policy du processus PID. La valeur de policy peut être SCHED_FIFO, SCHED_RR, SCHED_OTHER. Si PID est NULL alors le processus courant est concerné.
- ◆ `int sched_getscheduler(pid_t pid);`
Obtient la politique d'ordonnancement associé au processus PID. Si PID est NULL alors le processus courant est concerné.

Les primitives liées à l'ordonnancement (2/7)

- ◆ `int sched_setparam(pid_t pid, const struct sched_param *param);`
Modifie les paramètres d'ordonnancement du processus PID.
La structure `sched_param` contient un seul champ correspondant à la priorité statique du processus. Si PID est NULL alors le processus courant est concerné.

```
struct sched_param {  
    ...  
    int sched_priority;  
    ...  
};
```

- ◆ `int sched_getparam(pid_t pid, struct sched_param *param);`

Les primitives liées à l'ordonnancement (3/7)

- ◆ `int sched_getparam(pid_t pid, struct sched_param *param);`
Obtient les paramètres d'ordonnancement du processus PID.
Si PID est NULL alors le processus courant est concerné.
- ◆ En cas d'échec, ces primitives renvoient la valeur -1.
La variable *errno* peut prendre les valeurs suivantes
 - ◆ EFAULT
paramètres contenant une adresse non valide
 - ◆ EINVAL
param ou *policy* contiennent une valeur non valide
 - ◆ EPERM
Processus n'ayant pas les droits nécessaires pour modifier ses paramètres.
 - ◆ ESRCH
Processus spécifié par PID inexistant.

Les primitives liées à l'ordonnancement (4/7)

- ◆ `int sched_get_priority_max(int policy);`
`int sched_get_priority_min(int policy);`
permettent de connaître les valeurs des priorités statiques minimale et maximale associées à une politique d'ordonnancement.
- ◆ `int sched_rr_get_interval(pid_t pid, struct timespec * tp);`
Obtient l'intervalle SCHED_RR pour le processus indiqué
La structure timespec est déclarée comme suit :

```
struct timespec {  
    time_t tv_sec; /* secondes */  
    long tv_nsec; /* nanosecondes */  
};
```

Les primitives liées à l'ordonnancement (5/7)

- ◆ Prototype déclaré dans <unistd.h>

```
int nice(int inc);
```

Modifie la politesse du processus

Les primitives liées à l'ordonnancement (6/7)

- ◆ Prototypes déclarés dans <sys/time.h> et <sys/resource.h>

int getpriority(int which, int who);

int setpriority(int which, int who, int prio);

Permet de lire ou d'écrire la priorité d'ordonnancement du processus.

- ◆ La valeur de which :

- ◆ **PRIO_PROCESS.**

who définit le PID du processus concerné.

- ◆ **PRIO_PGRP.**

who définit le PGID du groupe de processus concerné

- ◆ **PRI_USER.**

who définit l'identifiant de l'utilisateur dont tous les processus sont concernés.

Les primitives liées à l'ordonnancement (7/7)

- En cas d'échec, ces primitives retournent la valeur -1. Errno peut prendre les valeurs suivantes :
 - EINVAL
Which contient une valeur non valide
 - EACCESS
Le processus n'a pas les droits nécessaires pour augmenter la priorité d'autres processus
 - ESRCH
Aucun processus ne correspond aux paramètres de who et de which

Questions (1/1)

- ◆ Un processus est :
 - un programme exécutable.
 - une instance d'un programme exécutable.
 - un contexte processeur.
- ◆ Un processus zombie est un processus qui :
 - a perdu son père.
 - a terminé son exécution en erreur.
 - a terminé son exécution et attend la prise en compte de cette fin par son père.
- ◆ En quoi consiste l'ordonnancement préemptif ?

Questions (1/1)

- ◆ Un processus est :
 - un programme exécutable.
 - une instance d'un programme exécutable.
 - un contexte processeur.
- ◆ Un processus zombie est un processus qui :
 - a perdu son père.
 - a terminé son exécution en erreur.
 - a terminé son exécution et attend la prise en compte de cette fin par son père.
- ◆ En quoi consiste l'ordonnancement préemptif ?
La transition de l'état élu vers l'état prêt est autorisée. Un processus quitte le processeur s'il a terminé son exécution, s'il se bloque ou si le processeur est réquisitionné.

TP : Ordonnancement

Créer le fichier 07_ordo.c et effectuer la compilation

Déterminer les fichiers d'en-tête (headers) à inclure

Fonction main() sans passage d'arguments et avec un code retour

Définir les variables et leurs types respectifs

Appel de la fonction fork()

Si le valeur de retour de la fonction fork() vaut 0 Alors

Appel de la fonction getpid()

Afficher "Je suis le fils. Ma priorité d'ordonnancement est X"

Obtenir la politique d'ordonnancement actuel

Si la politique est égale à SCHED_RR Alors

afficher "Ma politique d'ordonnancement est SCHED_RR"

Si la politique est égale à == SCHED_FIFO Alors

afficher "Ma politique d'ordonnancement est SCHED_FIFO"

Si la politique est égale à == SCHED_OTHER Alors

afficher "Ma politique d'ordonnancement est SCHED_OTHER"

Définir la priorité d'ordonnancement à 10

Définir la politique d'ordonnancement à SCHED_FIFO

Si problème de définition de la politique d'ordonnancement alors

afficher "Problème setscheduler"

Obtenir la priorité actuelle d'ordonnancement

Afficher "Ma priorité d'ordonnancement est X"

politique = sched_getscheduler(pid);

Si la politique est SCHED_RR alors

afficher "Ma nouvelle politique d'ordonnancement est SCHED_RR"

Si la politique est SCHED_FIFO alors

afficher "Ma nouvelle politique d'ordonnancement est SCHED_FIFO"

Si la politique est SCHED_OTHER alors

afficher "Ma nouvelle politique d'ordonnancement est SCHED_OTHER"

Sinon

afficher "Je suis le père : Priorité min X et max X de la politique SCHED_FIFO"

afficher "Priorité min X et max X de la politique SCHED_RR"

Obtenir les valeurs du quantum de la politique

Afficher "Les valeurs du quantum de la politique SCHED_RR, X secondes, X nanosecondes"

Attendre le fils

Fin de si

Fin de la fonction main()

Réponse : Ordonnancement

```

#include <stdio.h>
#include <sched.h>
#include <sys/wait.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/resource.h>

int main(void)
{
    pid_t ret, pid;
    int politique, status, priorite;
    struct timespec quantum;
    struct sched_param param;

    ret = fork();
    if (ret == 0)
        {
            pid = getpid();
            printf("je suis le fils : Ma priorité d'ordonnancement est %d\n", getpriority(PRIO_PROCESS,pid));

            politique = sched_getscheduler(pid);
            if (politique == SCHED_RR)
                printf("Ma politique d'ordonnancement est SCHED_RR\n");
            if (politique == SCHED_FIFO)
                printf("Ma politique d'ordonnancement est SCHED_FIFO\n");
            if (politique == SCHED_OTHER)
                printf("Ma politique d'ordonnancement est SCHED_OTHER\n");

            param.sched_priority = 10;
            setpriority(PRIO_PROCESS, pid, 10);
            if (sched_setscheduler(pid, SCHED_FIFO, &param) == -1)
                perror("Problème setscheduler");

            priorite = getpriority(PRIO_PROCESS, pid);
            printf("Ma priorité d'ordonnancement est %d\n", priorite);
            politique = sched_getscheduler(pid);
            if (politique == SCHED_RR)
                printf("Ma nouvelle politique d'ordonnancement est SCHED_RR\n");
            if (politique == SCHED_FIFO)
                printf("Ma nouvelle politique d'ordonnancement est SCHED_FIFO\n");
            if (politique == SCHED_OTHER)
                printf("Ma nouvelle politique d'ordonnancement est SCHED_OTHER\n"); }

        Else { printf("Je suis le père : Priorité min %d et max %d de la politique SCHED_FIFO\n",
                    sched_get_priority_min(SCHED_FIFO),      sched_get_priority_max(SCHED_FIFO)) ;
        printf("Priorité min %d et max %d de la politique SCHED_RR\n", sched_get_priority_min(SCHED_RR),
               sched_get_priority_max(SCHED_RR));
        sched_rr_get_interval(0, &quantum);
        printf("Voici les valeurs du quantum de la politique SCHED_RR, %d secondes, %ld nanosecondes\n",
               (int) quantum.tv_sec, quantum.tv_nsec);
        wait(&status); }
}

```

Module 3

Système de gestion de fichiers

Module 3 - Système de gestion de fichiers

- ◆ Notions générales
- ◆ Le système de gestion de fichiers virtuel VFS
- ◆ Le système de fichiers /proc

Notions générales

Rappel

- ◆ La mémoire centrale de l'ordinateur est volatile.
- ◆ Les programmes et les données stockés dans la mémoire centrale ont besoin d'être conservés lorsque l'alimentation électrique de l'ordinateur est interrompue.
- ◆ Le système de gestion de fichiers assure la conservation des données sur un support de masse non volatile.

Le système d'exploitation offre à l'utilisateur une unité de stockage indépendante des propriétés physiques des supports : le fichier.

Notions générales

Rappel

- ◆ Le concept de fichier recouvre 2 niveaux :
 - ◆ Le fichier logique
qui représente l'ensemble des données incluses dans le fichier telles qu'elles sont vues par l'utilisateur.
 - ◆ Le fichier physique
qui représente le fichier tel qu'il est alloué physiquement sur le support de masse.
- ◆ L'OS gère ces 2 niveaux et assure la correspondance entre eux en utilisant une structure de répertoires (dossiers).

Notions générales

Extfs

- ♦ L'extended file system est le 1^{er} système de fichiers créé en avril 1992 pour GNU/Linux par Rémy Card pour surmonter certaines limitations du système de fichiers Minix (Minix FS) créé par Andrew Tanenbaum.
- ♦ Minix a de nombreuses limitations :
 - ♦ la partition de 64 Mo,
 - ♦ les noms de fichiers de 14 caractères au maximum,
 - ♦ un seul horodatage,
 - ♦ Les répertoires comportent seulement 16 entrées,
 - ♦ Etc...

Notions générales ext2 ou xiafs ?

- ◆ Développé par Frank Xia, Xiafs est un système de fichiers basé sur Minix FS pour GNU/Linux.
- ◆ Ext2 et Xiafs ont été inclus dans la version du noyau Linux 0.99.15 en décembre 1993. Leur but était d'offrir de bonnes performances et des limitations raisonnables.
- ◆ Xiafs était plus puissant et plus stable que ext2 mais avait quelques limitations comme les fichiers limités à 64 Mo ou les partitions limitées à 2 Go
- ◆ Xiafs est aujourd'hui obsolète.

Notions générales

ext2

- ◆ Créé par Rémy Card, le 2nd extended file system (ext2 ou ext2fs) est basé sur l'extended file system (extfs) et a été influencé par le FFS (Berkeley Fast File System).
- ◆ Caractéristiques :
 - ◆ Taille maximale de fichier 16 Gio – 2 Tio
 - ◆ Nombre maximal de fichiers 10^{18}
 - ◆ Taille maximale du nom de fichiers 255 octets
 - ◆ Taille maximale de volume 2 Tio – 32 Tio
 - ◆ Non journalisé

Notions générales

FS Journalisé

- Un journal est la partie d'un système de fichiers journalisé qui trace les opérations d'écriture tant qu'elles ne sont pas terminées et cela en vue de garantir l'intégrité des données en cas d'arrêt brutal.
- L'intérêt est de pouvoir plus facilement et plus rapidement récupérer les données en cas d'arrêt brutal du système d'exploitation (coupure d'alimentation, plantage du système....) alors que les partitions n'ont pas été correctement synchronisées et démontées.

Notions générales

FS Journalisé

- ♦ Sans un tel fichier journal, un outil de récupération de données après un arrêt brutal doit parcourir l'intégralité du système de fichier pour vérifier sa cohérence. Lorsque la taille du système de fichiers est importante, cela peut durer très longtemps... pour un résultat parfois moins efficace (possibilité de pertes de données).

Notions générales

Ext3fs

- ◆ Développé par Stephen Tweedie, ext3fs est une évolution de ext2fs. Une extension a été ajoutée permettant de journaliser le système de fichiers en février 1999.
- ◆ ext3 passe dans la branche 2.4.15 du noyau Linux en novembre 2001.
- ◆ Un système qui ne connaît que l'ext2 est parfaitement capable de lire et d'écrire de l'ext3 mais il n'y aura pas alors de journalisation.

Notions générales

Ext3fs

- ◆ Limitations :
 - ◆ Taille maximale de fichier 16 Gio – 2 Tio
 - ◆ Nombre maximal de fichiers variable*
 - ◆ Taille maximale du nom de fichiers 255 caractères
 - ◆ Taille maximale de volume 2 Tio – 32 Tio

* Le nombre maximum d'inodes est défini à l'installation du système de fichiers. Si V est la taille du volume en octets, alors le nombre maximum par défaut d'inodes est donné par $V/2^{13}$, et le minimum par $V/2^{23}$.

Notions générales

Différence entre ext2 et ext3

- ◆ La différence entre les 2 systèmes réside dans l'adjonction d'une zone journal et la suppression des données rendant la récupération de celles-ci impossible sur le système de fichier ext3. Il suffit de cocher une option dans son noyau et de le recompiler pour bénéficier du support de ext3.
- ◆ Pour convertir une partition ext2 en ext3 :
tune2fs -j /dev/sda2

Notions générales

Ext4fs

- ◆ Ext4 garde une compatibilité avec son prédecesseur ext3 et est considéré par ses propres concepteurs comme une étape intermédiaire en attendant une nouvelle génération de FS tel que Btrfs.
- ◆ A titre expérimental, il est inclus dans le noyau Linux 2.6.19 (29 novembre 2006).
- ◆ A compter de la version 2.6.28 du noyau, le système est considéré comme stable.

Notions générales

Ext4fs

- ◆ La fonctionnalité majeure de ext4 est l'allocation par extent qui permettent la pré-allocation d'une zone contiguë pour un fichier afin de minimiser la fragmentation.
- ◆ L'option extent est active par défaut depuis le noyau Linux 2.6.23. Auparavant, elle devait être explicitement indiquée lors du montage de la partition :

mount /dev/sda1 /mnt/point -t ext4dev -o extents

Notions générales

Ext4fs

- ◆ Limitations :
 - ◆ Taille maximale de fichier 16 Tio
 - ◆ Nombre maximal de fichiers 4 milliards
 - ◆ Taille maximale du nom de fichiers 256 octets
 - ◆ Taille maximale de volume
(limité à 16To par e2fsprogs) 1 Eio

Systèmes de fichiers

xfs

- ◆ XFS est un FS 64 bits journalisé de haute performance créé par SGI pour son système d'exploitation IRIX.
- ◆ En mai 2000, SGI place XFS sous la licence GPL.
- ◆ XFS est inclus par défaut avec les versions du noyau Linux 2.5 et 2.6
- ◆ XFS devient le FS par défaut de Redhat 7

Systèmes de fichiers

xfs

♦ Limitations :	
◆ Taille maximale de volume	8 Eio - 1 octet
◆ Taille maximale de fichier	8 Eio - 1 octet
◆ Nombre de fichiers max.	2^{64}
◆ Taille maximale du nom de fichiers	255 caractères
◆ Caractères autorisés dans le nom du fichier	Tous sauf NULL et /

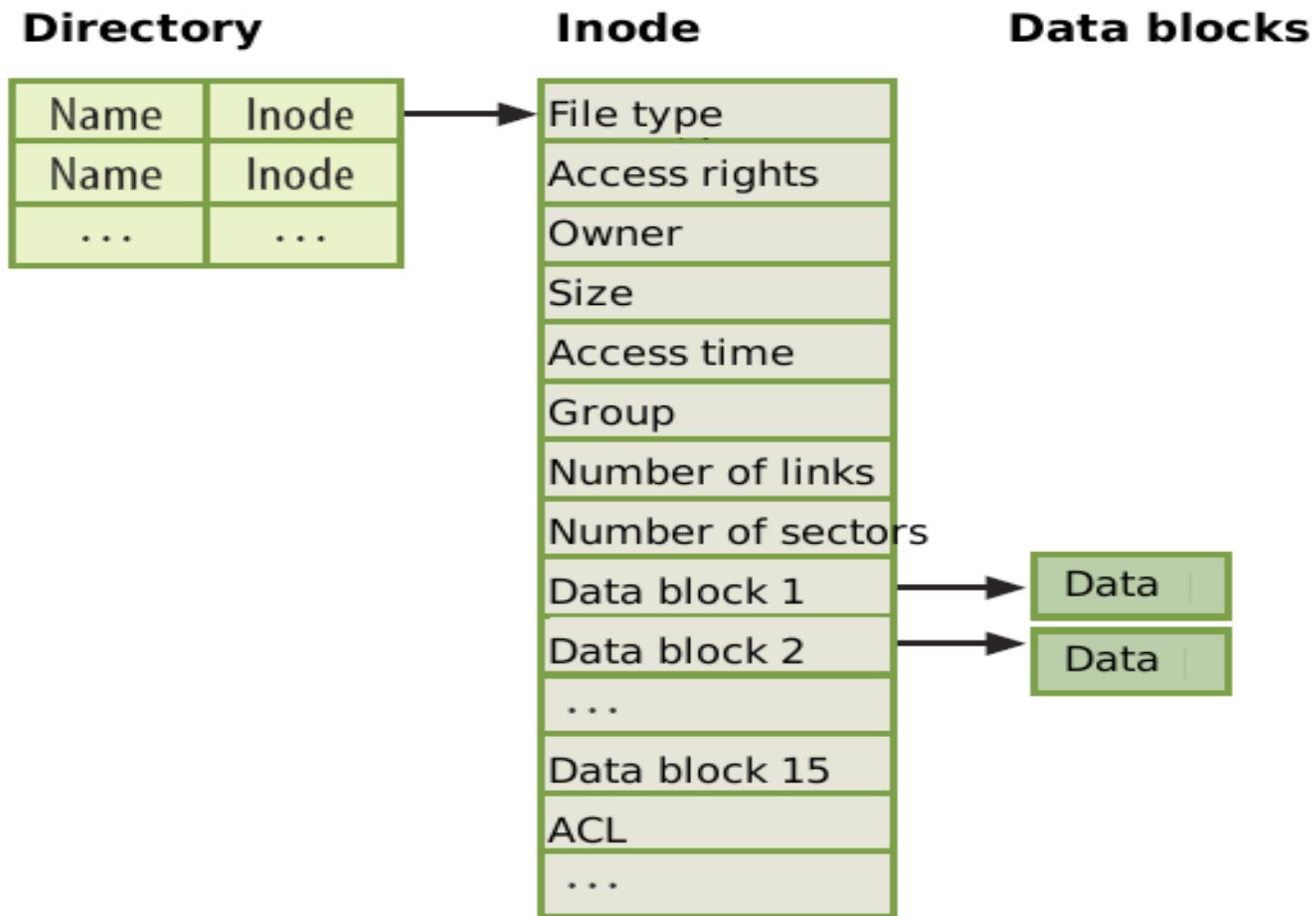
Notions générales

Inodes

- ◆ Les inodes (« index » et « node ») sont des structures de données contenant des informations concernant les fichiers stockés.
- ◆ A chaque fichier correspond un numéro d'inode dans le système de fichiers dans lequel il réside, unique au périphérique sur lequel il est situé.
- ◆ Les inodes contiennent aussi des informations concernant le fichier tel que son créateur, son propriétaire, son type d'accès...

Notions générales

Inodes



Notions générales

Inodes

- ♦ Exemple, nous utilisons la commande *stat* :

stat \$(which ls)

Fichier : «/bin/ls»

Taille : 105840 Blocs : 208 ES blocs : 4096 fichier

Device : 801h/2049d Inode : 654158 Liens : 1

Accès : (0755/-rwxr-xr-x) UID : (0/ root) GID : (0/ root)

Accès : 2012-05-15 18:42:23.990740309 +0200

Modi. : 2012-04-01 05:09:12.000000000 +0200

Chgt : 2012-05-12 09:53:43.561717644 +0200

Créé : -

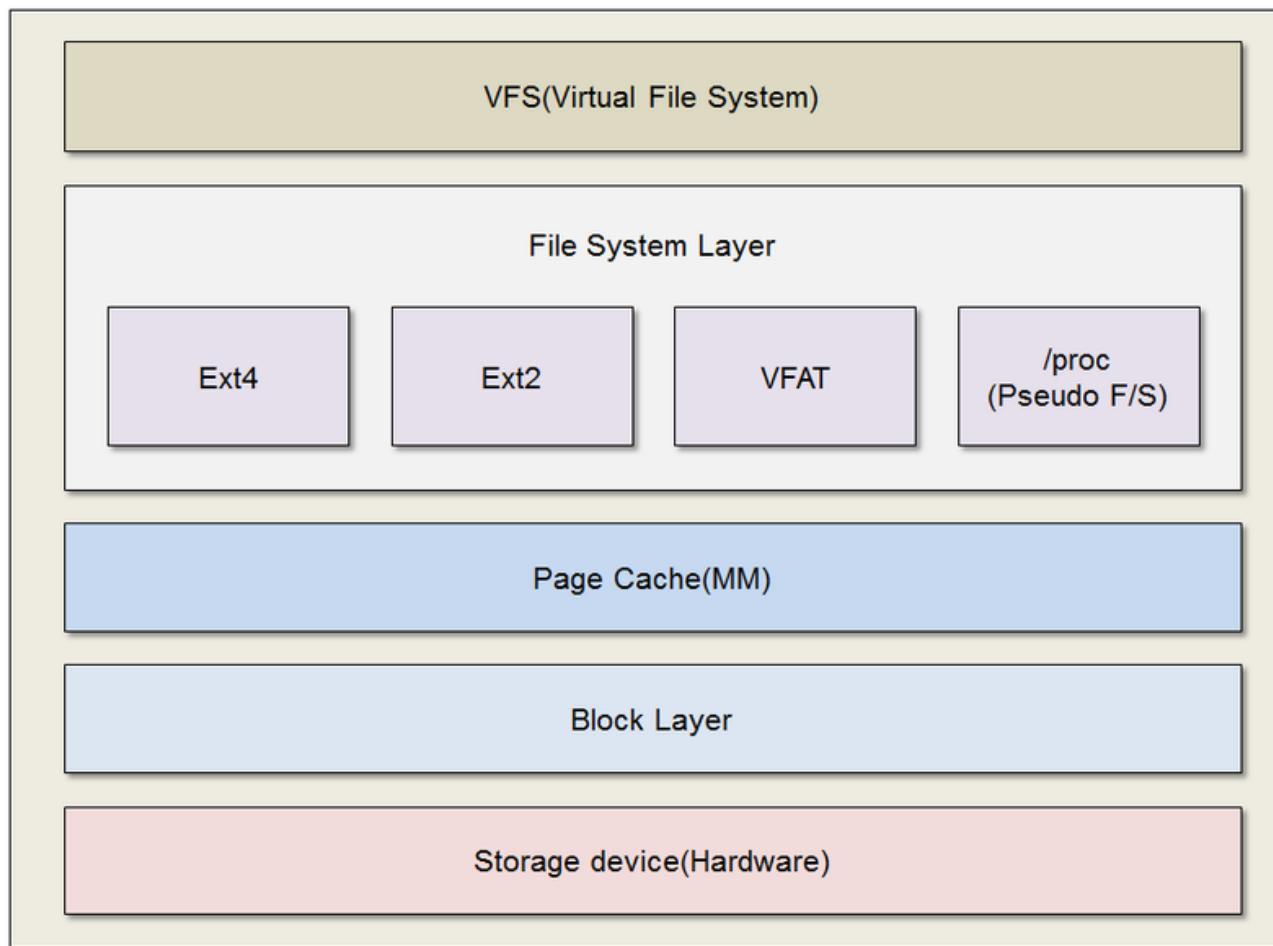
- ♦ *Stat* est le nom d'un appel système et d'une commande faisant partie du paquetage Coreutils ou GNU Core Utilities. Cet outil permet d'obtenir des informations sur des fichiers ou répertoires.

VFS

- ♦ Linux supporte d'autres systèmes de fichiers que le FS natif tels que ZFS, BTRFS...
- ♦ Pour que l'accès à ces différents systèmes de fichiers soit transparent et uniforme pour l'utilisateur, une couche logicielle appelé VFS (Virtual File Systems) est insérée dans le noyau Linux.

VFS

- Les structures du VFS créent un modèle de fichiers pour tous FS qui est très proche du modèle de fichier natif de Linux.



VFS

- ◆ La structure du VFS tourne autour de 4 objets :
 - ◆ Objet système de gestion de fichiers (superblock)
(struct super_block définie dans fs.h)
 - ◆ Objet inode
(struct inode définie dans fs.h)
 - ◆ Objet fichier (file)
(struct file définie dans fs.h)
 - ◆ Objet nom de fichier (dentry)
- ◆ VFS maintient 2 caches :
 - ◆ Le dcache pour conserver les dentry les plus récemment utilisés
 - ◆ Le buffer cache pour conserver les blocs disque les plus récemment utilisés

VFS

Les primitives

- ◆ Ouverture d'un fichier :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
```

int open(const char *ref, int mode_ouverture, mode_t mode);

*ref spécifie le chemin du fichier

mode_ouverture est une combinaison de constante (ou binaire) : O_RDONLY, O_CREAT, O_APPEND...

mode n'est positionné que si la création du fichier a été demandé (O_CREAT). Il permet de spécifier les droits d'accès associés au fichier : S_IRUSR, S_IWUSR...

VFS

Les primitives

- ◆ Fermeture d'un fichier :

```
#include <unistd.h>
```

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

desc correspond au descripteur de fichier obtenu lors de l'ouverture du fichier.

VFS

Les primitives

- ◆ Lecture dans un fichier :

```
#include <unistd.h>
```

```
ssize_t read(int desc, void *ptr_buf, size_t nb_octets);
```

read lit dans le fichier correspondant au descripteur *desc*, dans le buffer de réception pointé par *ptr_buf*, un nombre de *nb_octets* octets.

- ◆ Écriture dans un fichier :

```
#include <unistd.h>
```

```
ssize_t write(int desc, void *ptr_buf, size_t nb_octets);
```

write place dans le fichier correspondant au descripteur *desc*, le contenu du buffer pointé par *ptr_buf*, un nombre de *nb_octets* octets.

VFS - Les primitives

```
*****  
/*      Exemple de manipulation d'un fichier:      */  
/*      création, positionnement, fermeture      */  
*****  
  
#include <stdio.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include <unistd.h>  
main()  
{  
    struct eleve {  
        char nom[10];  
        int note;  
    };  
    int fd, i, ret;  
    struct eleve un_eleve;  
    fd = open ("/home/delacroix/fichnotes", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);  
    i = 0;  
    while (i<4)  
    {  
        printf ("Donnez le nom de l'élève \n"); scanf ("%s", un_eleve.nom);  
        printf ("Donnez la note de l'élève \n"); scanf ("%d", &un_eleve.note);  
        write (fd, &un_eleve, sizeof(un_eleve));  
        i = i + 1;  
    }  
    ret = lseek(fd,0,SEEK_SET);  
    printf ("la nouvelle position est %d\n", ret);  
    i = 0;  
    while(i<4)  
    {  
        read (fd, &un_eleve, sizeof(un_eleve));  
        printf ("le nom et la note de l'élève sont %s, %d\n", un_eleve.nom,  
        un_eleve.note);  
        i = i + 1;  
    }  
    close(fd);  
}
```

- En programmation, un fichier logique est un type de donnée sur lequel peuvent être appliquées des opérations spécifiques. C'est un ensemble d'enregistrements, désigné par un nom, Accessible selon différentes méthodes d'accès

Liaison via le SGF avec le fichier physique
Représentation du fichier interne au programme

Accès séquentiel

Accès direct

Rupture de la Liaison avec le fichier physique

VFS

Les primitives

- ◆ Attributs des fichiers :

```
#include <sys/types.h>
#include <sys/stat.h>
```

```
int stat(const char *ref, struct stat *infos);
int fstat(const int desc, struct stat *infos);
```

Les 2 fonctions retournent les attributs associés à un fichier désigné par son nom ou bien par son descripteur dans la structure struct stat *infos (sys/stat.h).

VFS

Les primitives

- ◆ Modification des droits d'accès d'un fichier :

```
#include <sys/stat.h>
```

```
int chmod(const char *ref, mode_t mode);  
int fchmod(const int desc, mode_t mode);
```

- ◆ Test des droits d'accès d'un fichier :

```
#include <unistd.h>
```

```
int access(char *ref, int access) ;
```

*ref correspond au nom du fichier et access est une constante qui représente le droit à tester : F_OK, R_OK, W_OK, X_OK. La fonction retourne 0 si le fichier est accessible.

VFS

Les primitives

- ◆ Destruction de liens physiques

```
#include <unistd.h>
```

```
int link(const char *ref1, const char *ref2);  
int unlink(const char *ref);
```

- ◆ Modification du nom de fichier

```
#include <unistd.h>
```

```
int rename(const char *ref_a_nom, const char *ref_n_nom) ;
```

- ◆ Modification du propriétaire d'un fichier :

```
#include <unistd.h>
```

```
int chown(const char *ref, uid_t id_user, gid_t id_grp);  
int fchown(int desc, uid_t id_user, gid_t id_grp);
```

VFS

Les primitives

- ◆ Changer de répertoire racine

```
#include <unistd.h>
int chroot(const char *ref);
```

- ◆ Changer de répertoire courant

```
#include <unistd.h>
int chdir(const char *ref);
int fchdir(int desc);
```

- ◆ Connaître le répertoire courant

```
#include <unistd.h>
char *getcwd(char *buf, size_t taille);
```

VFS

Les primitives

- ◆ Création d'un répertoire :

```
#include <sys/type.h>
#include <fnctl.h>
#include <unistd.h>
```

```
int mkdir(const char *ref, mode_t mode);
```

- ◆ Suppression d'un répertoire :

```
#include <unistd.h>
```

```
int rmdir(const char *ref);
```

VFS

Les primitives

- ◆ Exploration d'un répertoire :

```
#include <sys/type.h>
#include <dirent.h>
#include <unistd.h>
```

```
DIR *opendir(const char *ref) ;
struct dirent *readdir (DIR *rep) ;
int closedir(DIR *rep) ;
```

- ◆ Liens symboliques

```
#include <unistd.h>

int symlink(const char *target, const char *linkpath);
int readlink(const char *target, char *buf, size_t taille);
```

VFS

Les primitives

- ◆ Montage et démontage de partitions

```
#include <sys/mount.h>
```

```
int mount(const char *source, const char *target,  
         const char *filesystemtype, unsigned long mountflags,  
         const void *data);
```

```
int umount(const char *target);
```

```
int umount2(const char *target, int flags);
```

TP : Manipulation de fichiers création, positionnement et fermeture

Créer le fichier 01_manipfic.c

Effectuer la compilation

Inclure les fichiers d'en-tête (headers)

Fonction clrscr()

effacer l'écran

Fin de fonction clrscr()

Fonction main() sans d'arguments

et avec un code retour

Déclaration de variables

Déclaration des structures nécessaires

Créer le fichier ./notes.dat

*Si erreur alors afficher « Problème
fonction xxxx »*

Appel de la fonction clrscr()

Dans une boucle

Saisir 4 élèves (noms et notes)

Écrire la saisie dans ./notes.dat

Fin de boucle

*Positionner le curseur au début
du fichier*

SI erreur de positionnement

*ALORS Afficher « Problème
fonction xxxx »*

Afficher nouvelle position est x

Dans une boucle

Lire tout le contenu du fichier

Fin de boucle

Fermer le fichier ./notes.dat

Fin de la fonction main()

Réponse : Manipulation de fichiers création, positionnement et fermeture

```
// 01_manipfic.c
// Compilation : gcc 01_manipfic.c -o 01_manipfic
```

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

void clrscr(void) {
    printf("\033[2J\033[1;1H");
}

int main(void) {
    struct elev {
        char nom[10];
        int note;
    };

    int fd,i,ret;
    struct elev un_eleve;

    fd=open("./notes.dat", O_RDWR|O_CREAT,S_IRUSR|S_IWUSR);

    if(fd== -1)
        perror("Problème fonction OPEN");
}
```

```
Clrscr();
i=0;
while(i<4) {
    printf("Nom élève : "); scanf("%s",
un_eleve.nom);
    printf("Notes : "); scanf("%d",
&un_eleve.note);
    write(fd,&un_eleve, sizeof(un_eleve));
    i++;
}
ret=lseek(fd,0,SEEK_SET);
if(ret== -1)
    perror("Problème fonction LSEEK");
printf("\n\n\nLa nouvelle position est
%d.\n\n", ret);
i=0;
while(i<4) {
    read(fd,&un_eleve,sizeof(un_eleve));
    printf("Nom : %s\nNote : %d\n",
un_eleve.nom, un_eleve.note);
    i++;
}
close(fd);
}
```

TP : Manipulation de fichiers

Créer le fichier 02a_manipdir.c et effectuer la compilation

Inclure les fichiers d'en-tête (headers)

Fonction clrscr()

effacer l'écran

Fin de fonction clrscr()

Fonction main() sans d'arguments et avec un code retour

Déclaration de variables

Déclaration des structures nécessaires

Appel de la fonction clrscr()

Ouvrir le répertoire courant

Dans une boucle

Afficher le contenu du répertoire courant

Fin de boucle

Fermer le répertoire

Fin de la fonction main()

Réponse : Manipulation de fichiers

```
// 02a_manipdir.c
```

```
// Compilation : gcc 02a_manipdir.c -o 02a_manipdir
```

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

void clrscr(void) {
    printf("\033[2J\033[1;1H");
}
```

```
int main(void) {
    struct dirent *lecture;
    DIR *rep;

    clrscr();

    rep = opendir(".");
    while ((lecture = readdir(rep))) {
        printf("%s\n", lecture->d_name);
    }
    closedir(rep);
}
```

TP : Manipulation de répertoires

Créer le fichier 02b_manipdir.c et effectuer la compilation

Inclure les fichiers d'en-tête (headers)

Fonction clrscr()

effacer l'écran

Fin de fonction clrscr()

Fonction main() sans d'arguments

et avec un code retour

Déclaration de variables

Déclaration des structures nécessaires

Appel de la fonction clrscr()

Ouvrir le répertoire courant

Afficher « Répertoire courant x »

Dans une boucle lire le contenu du répertoire

Afficher « Numéro inode : x et Nom de fichier x

Fin de boucle

Fermer le répertoire

Fin de la fonction main()

Réponse : Manipulation de répertoires

```
// 02b_manipdir.c
```

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
#include <unistd.h>
#include <limits.h>
#include <stdlib.h>

void clrscr(void) {
    printf("\033[2J\033[1;1H");
}

int main(void) {
    char buf[ UCHAR_MAX ];
    struct dirent *lecture;
    DIR *rep;

    clrscr();

    rep = opendir(".");
}
```

```
if (getcwd (buf, UCHAR_MAX) == NULL) {
    return EXIT_FAILURE;
}

printf("Répertoire courant %s\n", buf);
lecture=readdir(rep);
while (lecture != NULL)  {
    printf("Inode : %d et Nom de fichier %s\n",
(int)lecture->d_ino, lecture->d_name);
    lecture=readdir(rep);
}

closedir(rep);
return EXIT_SUCCESS;
}
```

/proc

- Le pseudo système de fichier procfs (process file system) est dynamiquement mis à jour par le noyau. Il est monté pour qu'il soit accessible sur le dossier /proc. Malgré qu'il soit non-vide, il affiche une taille de 0 kio :

```
# ls -ld /proc  
dr-xr-xr-x 110 root root 0 juil. 28 16:14 /proc
```

/proc

- ◆ Le concept de taille de fichier n'a pas de sens pour procfs puisqu'il occupe uniquement une quantité limitée de mémoire vive.
- ◆ Le dossier /proc, supporté par VFS, est utilisé pour accéder aux informations du noyau qui est en cours d'exécution. Vous pouvez ainsi consulter en temps réel des informations sur le matériel, le système, le réseau, les processus, etc... Il est possible de changer la configuration du noyau en écrivant dans certains fichiers stockés dans /proc/sys (voir sysctl).

Questions (1/1)

- ◆ Quel est le rôle de VFS ?
 - Permettre l'accès à des données distantes.
 - Uniformiser l'accès à de multiples systèmes de gestion de fichiers.
 - Gérer les accès disque
- ◆ Une inode est :
 - un descripteur de processus.
 - un descripteur de partition.
 - un descripteur de fichier.
- ◆ Le buffer cache est :
 - un mécanisme de cache de processeur.
 - un mécanisme de cache système qui mémorise les pages mémoires les plus récemment accédées.
 - un mécanisme de cache qui mémorise les blocs disques les plus récemment accédés.

Réponses (1/1)

- ◆ Quel est le rôle de VFS ?
 - Permettre l'accès à des données distantes.
 - Uniformiser l'accès à de multiples systèmes de gestion de fichiers.
 - Gérer les accès disque
- ◆ Une inode est :
 - un descripteur de processus.
 - un descripteur de partition.
 - un descripteur de fichier.
- ◆ Le buffer cache est :
 - un mécanisme de cache de processeur.
 - un mécanisme de cache système qui mémorise les pages mémoires les plus récemment accédées.
 - un mécanisme de cache qui mémorise les blocs disques les plus récemment accédés.

Module 4

Gestion des Entrées/Sorties

Module 4 - Gestion des E/S

- ◆ Principes généraux
- ◆ Entrées/Sorties Linux

Principes généraux

- ◆ Le processeur dialogue avec l'extérieur par le biais de périphérique (clavier, écran, imprimante, disque dur...)
- ◆ La multitude de périphériques est grande. Les caractéristiques techniques de ces différents périphériques sont diverses.
- ◆ La gestion des périphériques s'effectue à 2 niveaux :
 - ◆ Matériel avec le dispositif de l'unité d'échange ou le contrôleur d'E/S.
 - ◆ OS avec le pilote d'E/S

Principes généraux

Niveau matériel – unité d'échange

- ◆ Le processeur s'interface avec le périphérique par le biais de l'unité d'échange ou contrôleur d'E/S.
- ◆ L'unité d'échange offre une interface standard d'E/S au processeur. Elle adapte les caractéristiques électriques et logiques des signaux émis ou reçus par les périphériques à celle des signaux véhiculés par le bus.
- ◆ L'unité d'échange est constituée par un ensemble de registres adressables par le processeur (registre d'état, registre de commandes, registre de données...).

Principes généraux

Niveau système – pilote de périphérique (1/)

- ◆ Au niveau système, la réalisation des opérations E/S est prise en charge par un pilote de périphérique ou driver.
- ◆ Le pilote maintient des structures de données qui décrivent l'état des unités d'échange associés à un ensemble de fonctions pour agir sur les périphériques :
 - ◆ Open. Permet d'initialiser le périphérique et les structures de données associées.
 - ◆ Close. Rompt la liaison avec le périphérique.
 - ◆ Read, write. Effectue des lectures ou écritures de données.
 - ◆ Schedule. Permet d'ordonnancer des requêtes d'E/S de façon à optimiser les temps de réalisation des opérations.
 - ◆ Interrupt. Constitue le gestionnaire d'interruption associé au périphérique. Elle est exécutée lorsque celui-ci lève une interruption à destination du processeur.

Principes généraux

Niveau système – pilote de périphérique (2/)

- ◆ Protocoles d'E/S

- ◆ Mode par scrutation (ou attente active)

Le pilote scrute en permanence l'unité d'échange pour savoir si elle est prête à recevoir/délivrer une nouvelle donnée. Le processeur durant l'opération d'E/S est totalement occupé par l'exécution du pilote

Algorithme du déroulé est le suivant :

```
programme pilote : {  
    lire registre_d'état ;  
    while (nb_caractères > 0) {  
        while (périphérique_occupé) {  
            lire registre_d'état ; }  
        Ecrire un caractère dans le registre de données ;  
        nb_caractères=nb_caractères – 1 ; }  
}
```

Principes généraux

Niveau système – pilote de périphérique (2/)

- ◆ Protocoles d'E/S (suite)

- ◆ Mode avec interruption

L'unité d'échange utilise le mécanisme des interruptions pour signaler qu'elle est prête. Un gestionnaire d'interruption *interrupt()* est associé au pilote.

A la réception d'une interruption, le programme en cours d'exécution est arrêté au profit de ce gestionnaire.

Ce mode de prise en compte de l'opération d'E/S libère le processeur sauf pendant les périodes où le processeur est attribué au programme de gestion de l'interruption et au pilote.

Principes généraux

Niveau système – pilote de périphérique (2/)

- ◆ Protocoles d'E/S (suite)
 - ◆ Mode avec interruption

Algorithme :

```
programme pilote : {  
    lire registre_d'état ;  
    while (périphérique_occupé) {  
        lire registre_d'état ;  
        Ecrire un caractère dans le registre de données ;  
        nb_caractères=nb_caractères – 1 ;  
        enable_it_imprimante ; }  
}  
  
gestionnaire_interruption() : {  
    disable_it_imprimante ;  
    if (nb_caractères > 0) {  
        Ecrire le caractère suivant dans le registre de données ;  
        nb_caractères=nb_caractères – 1 ; }  
    enable_it_imprimante ; }  
}
```

Principes généraux

Niveau système – pilote de périphérique (2/)

- ◆ Protocoles d'E/S (suite)

- ◆ Mode avec DMA (Direct Memory Access)

C'est un composant matériel permettant d'effectuer des échanges entre mémoire centrale et unité d'échange sans utilisation du processeur central. Il comprend :

- ◆ Un registre d'adressage qui reçoit l'adresse du premier caractère à transférer
 - ◆ Un registre de comptage qui reçoit le nombre de caractères à transférer
 - ◆ Un registre de commande qui reçoit le type d'opération à effectuer (lecture ou écriture)
 - ◆ Une zone de tampon pour stocker les données
 - ◆ Un composant actif de type processeur qui exécute un transfert sans l'utilisation du processeur central.

Principes généraux

Niveau système – pilote de périphérique (2/)

- ◆ Protocoles d'E/S (suite)
 - ◆ Mode avec DMA (Direct Memory Access)

Algorithme :

```
programme pilote : {  
    registre_adresse_DMA = adresse du 1er caractère en mémoire centrale ;  
    registre_comptage_DMA = nb_caractères ;  
    registre_commande_DMA = écriture ;  
    enable_it_imprimante ;  
}
```

```
gestionnaire_interruption() : {  
    disable_it_imprimante ;  
    vérifier que le transfert s'est bien passé ;  
}
```

Principes généraux

Niveau système – pilote de périphérique (2/)

- ◆ Protocoles d'E/S (suite)

- ◆ L'ordonnancement des requêtes

Le pilote peut comporter une fonction schedule. Son rôle est d'ordonner les requêtes à destination du périphérique qu'il contrôle.

Par exemple, l'accès à un secteur du disque dur.

- ◆ Les principaux algorithmes d'ordonnancement sont :
 - ◆ FCFS (First Come, First Served),
 - ◆ SSTF (Shortest Seek Time First),
 - ◆ SCAN (ascenseur ou balayage)

Entrées/Sorties Linux (1/7)

- ♦ Linux gère les E/S au travers de fichiers spéciaux. Par exemple, un appel système write peut être utilisé dans un fichier régulier ou lancer une impression en écrivant dans le fichier spécial /dev/lp0.
- ♦ Ils sont de 2 types :
 - ♦ Fichiers de type bloc (b)
Périphériques structurés en bloc pour lesquels l'accès direct est possible (disques, CDROM...). L'accès s'effectue au travers du *buffer cache*.
 - ♦ Fichiers de type caractères (c)
Périphériques sans structure sur lesquels les données sont accessibles de façon séquentielle, octet par octet (imprimante, terminal, carte son...).

Entrées/Sorties Linux (2/7)

- ◆ Les fichiers spéciaux sont identifiés par 3 attributs :
 - ◆ Son type b ou c
 - ◆ Son numéro majeur (compris entre 1 et 255) qui identifie le pilote de périphérique
 - ◆ Son numéro mineur qui identifie le périphérique physique géré par le pilote
- ◆ Les primitives pour l'enregistrement des pilotes :
`register_chrdev(unsigned int major, const char *name,
 struct file_operations *fops) ;`
`register_blkdev(unsigned int major, const char *name,
 struct file_operations *fops) ;`

Entrées/Sorties Linux (3/7)

- ◆ La primitive pour créer un fichier spécial :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
```

```
int mknod(const char *ref, mode_t mode, dev_t dev) ;
```

ref spécifie le nom du fichier à créer
mode donne les permissions
dev définit le type de fichier à créer :

S_IFREG	: fichier régulier (-)
S_IFCHR	: fichier de type caractère (c)
S_IFBLK	: fichier de type bloc (b)
S_IFIFO	: fichier tube nommé (p)

Entrées/Sorties Linux (4/7)

- ◆ Pour gérer les numéros de périphériques :

```
#define _BSD_SOURCE  
#include <sys/types.h>
```

```
dev_t makedev(int maj, int min);  
unsigned int major(dev_t dev);  
unsigned int minor(dev_t dev);
```

Entrées/Sorties Linux (5/7)

- ◆ Pour modifier les paramètres d'un périphérique :

```
#include <sys.ioctl.h>
```

```
int ioctl(int fd, int cmd, char *arg) ;
```

L'opération désignée par *cmd* avec l'argument *arg* est réalisé sur le périphérique dont le descripteur est *fd*.

Entrées/Sorties Linux (6/7)

- ❖ Multiplexage des E/S :

```
#include <sys/types.h>
```

```
int select(int nb_desc, fd_set *ens_lire, fd_set *ens_ecrire,  
          fd_set *ens_exceptions, struct timeval *delai) ;
```

ens_lire correspond à l'ensemble des descripteurs sur lesquels le processus attend des données à lire

ens_ecrire correspond à l'ensemble des descripteurs sur lesquels le processus attend des données à écrire

ens_exceptions correspond à l'ensemble des descripteurs sur lesquels le processus attend l'arrivée de conditions exceptionnelles (rarement utilisé).

Entrées/Sorties Linux (7/7)

delay correspond à la définition d'un temps d'attente maximal. Si aucune temporisation n'est définie, ce paramètre est NULL.

```
struct timeval {  
    time_t tv_sec ;          //secondes  
    time_t tv_usec ;         //microsecondes  
} ;
```

nb_desc correspond au numéro du plus grand descripteur dans l'ensemble auquel est ajouté 1.

TP : Endormir un processus en attente d'1 caractère sur STDIN

Créer le fichier XXXX.c et effectuer la compilation

Include les fichiers d'en-tête (headers)

Fonction main() sans d'arguments et sans retour

Déclaration de variables

Déclaration des structures nécessaires

Fin de la fonction main()

Réponse : Endormir un processus en attente d'un caractère sur STDIN

```
// 01a_select.c
```

```
#include <sys/time.h>
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

void main(void)
{
    fd_set readset;
    struct timeval duree;
    char c;
    int fd, retour;

    Fd=0;
    FD_ZERO(&readset);
    FD_SET(fd, &readset);

    duree.tv_sec=10;
    duree.tv_usec=0;

    retour=select(1, &readset, NULL, NULL, &duree);
```

```
        if(retour ==0)
            printf("Temporisation finie. Pas de caractères reçus !\n");
        else if (FD_ISSET(fd, &readset))
        {
            read(fd, &c, sizeof(c));
            printf("Caractère lu : %c\n", c);
        }
    }
```

Module 5

Gestion de la mémoire centrale

Module 5 - Gestion de la mémoire centrale

- ◆ Les mécanismes de pagination et de mémoire virtuelle
- ◆ La gestion de la mémoire centrale sous Linux

Les mécanismes de pagination et de mémoire virtuelle

- ◆ Rappel sur la mémoire physique :
 - ◆ Elle est constituée d'un ensemble de mots mémoire contigus désignés chacun par une adresse physique.
 - ◆ Un mot mémoire est formé par 1 octet (8 bits), 2 octets (16 bits), 4 octets (32 bits), 8 octets (64 bits)...
 - ◆ La mémoire centrale est composée par de la mémoire type DRAM qui est volatile et accessible en lecture et écriture.
 - ◆ Le processeur accède aux mots de la mémoire centrale par le biais de 2 registres : RAD qui contient l'adresse du mot à lire ou à écrire et RDO qui contient la donnée à lire ou à écrire.

Les mécanismes de pagination et de mémoire virtuelle

- ◆ L'espace d'adressage d'un processus :
 - ◆ Il est constitué de l'ensemble d'adresses auxquelles le processus a accès au cours d'exécution. Il est au moins constitué de 3 parties : le code, les données du processus, la pile d'exécution.
 - ◆ L'espace d'adressage d'un processus constitue une notion totalement indépendante de la mémoire physique. Il est appelé « espace d'adresses logiques ou virtuelles ».
- ◆ Pour exécuter un programme, il doit être chargé dans la mémoire physique. Ce qui implique de mapper l'espace d'adressage du processus sur l'espace de la mémoire physique de la mémoire centrale.

Les mécanismes de pagination et de mémoire virtuelle

- ◆ Pagination de la mémoire centrale :
 - ◆ L'espace d'adressage du programme est découpé en morceaux linéaires de même taille appelés « pages ».
 - ◆ L'espace de la mémoire physique est aussi découpé en morceaux linéaires de même taille appelés « cases » ou « cadres de page ». La taille d'une case est égale à la taille d'une page. Elle est défini par le matériel selon le système d'exploitation (étant une puissance de 2). La valeur varie entre 512 et 8192 octets.
 - ◆ Charger un programme en mémoire centrale consiste à placer les pages dans n'importe quelle case disponible. Le système maintient une « table des cases ».

Module 6

Gestion des signaux

Module 6 - Gestion des signaux

- ◆ Présentation générale
- ◆ Aspect du traitement des signaux par le noyau
- ◆ Programmation des signaux
- ◆ Signaux temps réel

Présentation générale

- ◆ Un signal est un message envoyé par le noyau à un processus ou à un groupe de processus pour indiquer l'occurrence d'un événement survenu au niveau du système.

Aspect du traitement des signaux par le noyau

- ♦ Les mécanismes de traitement des signaux par le noyau :
 - ♦ Envoi d'un signal à un processus
 - ♦ Prise en compte du signal par le processus et l'action qui en résulte
 - ♦ Interaction de certains appels systèmes avec les signaux

Programmation des signaux

- ◆ Primitive pour envoyer un signal

```
#include <signal.h>
```

```
int kill(pid_t pid, int num_sig)
```

Module 7

Communications entre processus

Module 7 – Communications entre processus

- ◆ Communication par tubes
- ◆ IPC (inter Processus Communication)

Communication par tubes

- ◆ Un tube est un tuyau dans lequel un processus peut écrire des données qu'un autre processus peut lire.
- ◆ La communication est unidirectionnelle dans le tube. Une fois que le sens d'utilisation du tube est choisi, il ne peut plus être changé.
- ◆ Linux propose 2 types de communication par tubes :
 - ◆ Tubes anonymes
 - ◆ Tubes nommés



A close-up photograph of a person's hand holding a plain white rectangular card. The hand is positioned with the fingers supporting the back of the card, and the thumb is visible on the left side. The card contains the following text in large, bold, blue sans-serif font:

**Programmation
système en langage C
sous LINUX**

Fin de session

DRAFT