

Systèmes d'Exploitation (HAI303I)

Michel Meynard

UM

Univ. Montpellier

Table des matières II

- 8 Système de Gestion des fichiers
- 9 Communications basiques entre Processus (signaux et tubes)
- 10 Thread

Table des matières I

- 1 Introduction
- 2 Développement en C sous Unix
- 3 Représentation de l'information
- 4 Structure des ordinateurs
- 5 La couche Machine
- 6 Gestion des Fichiers et des Entrées/Sorties
- 7 Gestion des processus

Plan

- 1 Introduction

Introduction I

Définition d'un SE (*Operating System*)

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

Objectifs

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

Introduction III

Principaux OS

Microsoft Windows principal système sur ordi. personnels

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

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

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

Serveurs z/VM, z/OS, z/TPF d'IBM, Oracle Solaris, MacOS Server, Windows Server

Introduction II

Services

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

Historique I

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

Historique II

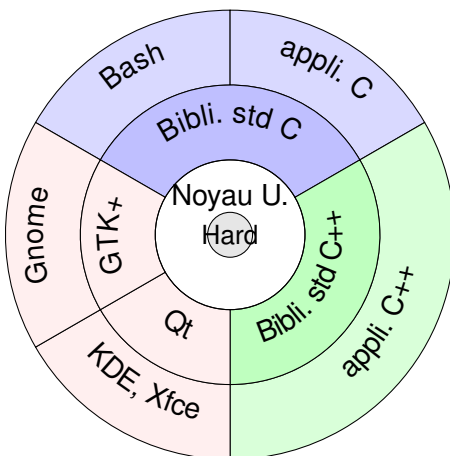
Remarques

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

solution 1

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

Architecture en couche d'Unix



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

Historique III

solution 2

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

solution 3

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

Les langages I

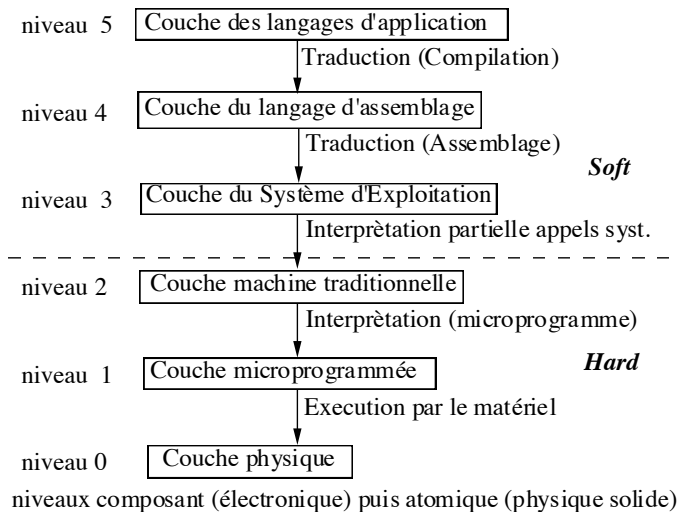
Le jeu d'instructions du processeur est limité et primitif. On construit donc au-dessus, une série de couches logicielles permettant à l'homme un dialogue plus aisé.

Interprétation Vs compilation

Les programmes en L_i sont :

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

Couches et langages I



Matériel et Logiciel I

Hardware

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

Software

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

Couches et langages II

Description des couches

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

Matériel et Logiciel II

Matériel et Logiciel sont conceptuellement équivalents

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

Matériel et Logiciel III

Exemples de répartition matériel/logiciel

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

Plan

2 Développement en C sous Unix

Objectifs du cours

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

Plan

- 2 Développement en C sous Unix
 - Compilations et édition de lien
 - Structure d'un programme C : les fonctions
 - Variables et types
 - Prétraitement, assembleur et objet
 - L'édition de liens statique et dynamique

Compilation des programmes (C sous Unix) I

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

Compilation des programmes (C sous Unix) II

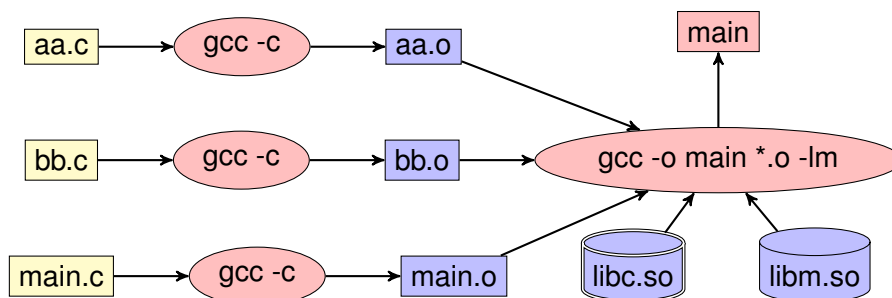
- la commande `gcc` supporte ces principales options :
 - c Compiler et assembler seulement (compile)
 - o xxx Renommage du fichier de sortie (output)
 - lm Utilisation de la librairie mathématique `libm.a` ou `.so`
 - Wall Voir tous les avertissements (Warning all)
 - g Ajoute les informations de débogage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

Processus de compilation

Compilations séparées

Édition de liens

Binaire exécutable



Sources

Objets

Bibliothèques

Plan

- 2 Développement en C sous Unix
 - Compilations et édition de lien
 - Structure d'un programme C : les fonctions
 - Variables et types
 - Prétraitement, assembleur et objet
 - L'édition de liens statique et dynamique

Structure d'un programme C I

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

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

Structure d'un programme C II

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

Les fonctions I

Déclaration d'une fonction

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

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

Les fonctions II

Définition d'une fonction

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

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

Un exemple complet : fact.c I

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

unsigned int fact(unsigned int);

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

Un exemple complet : fact.c II

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

Un exemple complet : fact.c III

Quelques exécutions

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

Passage des paramètres I

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

Passage des paramètres II

Exemple `passageparam.c`

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

Plan

2 Développement en C sous Unix

- Compilations et édition de lien
- Structure d'un programme C : les fonctions
- Variables et types
- Prétraitement, assembleur et objet
- L'édition de liens statique et dynamique

Passage des paramètres III

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

Exécution

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

Les variables C : propriétés I

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

Les variables C : propriétés II

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

Exemple `portee.c`

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

Les types C I

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

char entier signé en complément à 2 sur 1 octet. Il permet de représenter les caractères ASCII (7 bits), les octets lus dans des fichiers, les cases des chaînes de caractères. Il existe aussi `signed char` et `unsigned char`

int entier signé en complément à 2 de taille dépendant de la machine (souvent 4 octets). Le type `unsigned int` est de même taille mais codé en RBNS

Les variables C : propriétés III

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

Exécution

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

Les types C II

short, long entier court (long) codé en complément à 2 dont la taille dépend de l'architecture. Par exemple (Mac OS X i5) : `int(4)`, `short(2)`, `long(8)`, `long long(8)`. Les types non signés correspondants sont possibles.

C99 cette norme définit les types de taille fixée : `int8_t`, `uint8_t`, `int16_t`, `uint16_t`, `int32_t`, `uint32_t`. Elle définit également des types rapides de taille minimale comme : `uint_fast64_t` (en-tête `stdint.h`)

float nombre flottant IEEE-754 avec des tailles non fixées : `float(4)`, `double(8)`, `long double(16)`

Les types C III

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

Les types C V

struct séquence hétérogène d'objets e.g. :

```
struct cell{int val; struct cell *suiv;} tete;
```

Les champs de la struct sont référencés grâce à la notation pointée sur la variable `tete` : `(tete.suiv)->val` est la seconde valeur de la liste. **Il faut allouer (malloc) de la mémoire aux structures de données dynamiques.**

union un champ parmi plusieurs possibles :

```
union dyn{int i; float f; char *s;} x; x est une variable qui peut contenir un entier, un flottant ou une chaîne.
```

La taille d'une union est la taille de son plus grand composant.

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

Les types C IV

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

Un exemple complet I

liste.h

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

Un exemple complet II

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

Un exemple complet III

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

Un exemple complet IV

```
/** Vide une liste en désallouant toutes ses cellules.
 * @param pl un pointeur sur la liste à vider
 * @warning effets de bord sur des listes qui partageraient des cellules
 */
```

```
void vider(liste *pl);
#endif /* _LISTE */
```

liste.c

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

Un exemple complet V

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

Un exemple complet VI

```
    return;
}
}
```

```
main.c
```

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

Plan

2 Développement en C sous Unix

- Compilations et édition de lien
- Structure d'un programme C : les fonctions
- Variables et types
- **Prétraitement, assembleur et objet**
- L'édition de liens statique et dynamique

Un exemple complet VII

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

Compilation puis exécution

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

Le prétraitement (C PreProcessing) I

Uniquement des substitutions **textuelles** dépendant des directives de compilation qui commencent toutes par #

- include** <stdio.h> ou "mesfons.h" : inclusion de fichiers d'entêtes standards ou propres au projet;
- define** MACRO expr : partout où le nom MACRO apparaîtra par la suite, il sera remplacé par l'expression expr;
- define** MAX(i,j) (i>j?i:j) : la macro-fonction MAX sera remplacée par l'expression conditionnelle où i et j seront remplacés par les paramètres réels de la macro;
- ifndef** _MM_H ... #elif ... #endif : inclusion conditionnée par la définition préalable de la macro _MM_H; les deux premières lignes de stdio.h sont :
#ifndef _STDIO_H_ #define _STDIO_H_;

Le prétraitement (C PreProcessing) II

`if expression` : pour des expressions plus complexes,
`#error`, `#warning` ...

- pour obtenir le résultat du prétraitement : `gcc -E main.c`
- pour indiquer un répertoire personnel où se trouvent des fichiers d'entête : `gcc -I./MesEntete main.c`
- erreurs possibles : fichier d'entête introuvable (différence entre " " et <>);
- dans les macro-définitions, penser à entourer l'expression de parenthèses afin d'éviter des problèmes de priorité;
- toujours encadrer ses fichiers d'en-têtes par des `#ifndef` ... afin d'éviter de multiples définitions !

La compilation : du C à l'objet I

Traduction d'un langage évolué en langage d'assemblage (texte) puis en format objet (binaire).

- 1 analyse lexicale (tokenisation);
- 2 analyse syntaxique selon la grammaire du langage C (parse ou syntax error);
- 3 analyse sémantique : correspondance de types, correspondance du nombre de paramètres, ... (importance de l'option `-Wall` de gcc);
- 4 optimisation ...
- 5 génération du code : `gcc -S ->` assembleur, `gcc -c ->` objet

Le Langage d'assemblage I

fact.c : un exemple de prog. C simple

```
#include <stdio.h>
#include <stdlib.h>
int fact(int n){
    if (n<=1)
        return 1;
    else
        return n*fact(n-1);
}
int main(int argc, char** argv, char** arge){
    if(argc!=2){
        printf("Syntaxe incorrecte : %s 8 \n", argv[0]);
        return 1;
    } else {
        int n=atoi(argv[1]);
        printf("%d ! = %d\n",n,fact(n)); ...
    }
}
```

Le Langage d'assemblage II

fact.s obtenu par gcc -S fact.c

```
; %rr registre; $1 valeur immédiate 1; mov src dest;
.file "fact.c"
.text ; début du segment de code
.globl _fact ; nom _fact est global (externe)
.def _fact; .scl 2; .type 32; .endef
_fact: ; début de la proc fact
    pushl %ebp ; sauve registre base de pile ebp
    movl %esp, %ebp ; affecte ebp jusqu'a la fin de la proc
    subl $8, %esp ; reserve 8 octets sur la pile
    cmpl $1, 8(%ebp) ; if (n<=1) : ebp+8 pointe sur le n de
l'appelant
    jg L2 ; if (n>1) goto L2
    movl $1, -4(%ebp) ; sinon (return 1;) ebp-4 valeur de retou
```

Le Langage d'assemblage III

```

    jmp L1    ; aller en fin commune de la proc
L2:
    movl 8(%ebp), %eax ; eax=n
    decl %eax    ; eax-- (n-1)
    movl %eax, (%esp) ; empiler n-1 avant l'appel récursif
    call _fact    ; appel réc.
    imull 8(%ebp), %eax ; n*fact(n-1)    (eax contient le résultat de :
    movl %eax, -4(%ebp) ;                ; +4    adrs de retour
                                        ; +8    n de l'appelant ...
L1: ; fin commune
    movl -4(%ebp), %eax ; ebp-4 dans eax (résultat de la fon)
    leave
    ret ; return de la proc

; pile de fact avant l'appel récursif à fact(n-1)
; -8 esp-> n-1
; -4    résultat temporaire de la fonction fact
; ebp -> ebp de l'appelant

```

Le Langage d'assemblage IV

```

; +4    adrs de retour
; +8    n de l'appelant ...

```

Le format objet .o

C'est un format binaire donc non lisible par l'homme. Il existe quelques commandes utiles de GNU Binutils :

```

$ file segfault.o
segfault.o: ELF 64-bit LSB relocatable, x86-64, version 1
$ nm segfault.o
0000000000000000 T main
                  U putchar
$ readelf -a segfault.o
...

```

La commande `nm` permet de connaître les types des symboles :

- T (Text=Code),
- U (Undefined)
- D (Data initialisée)

Plan

- 2 Développement en C sous Unix
 - Compilations et édition de lien
 - Structure d'un programme C : les fonctions
 - Variables et types
 - Prétraitement, assembleur et objet
 - L'édition de liens statique et dynamique

L'édition de liens dynamique

Cette phase permet de **résoudre** les références non définies des fichiers objets entre eux et/ou avec des bibliothèques. Une bibliothèque est constituée d'un ensemble de fonctions au format objet. Son extension est : `.so` (Shared Object) sous linux, `.dll` sous Windows, `.dylib` sous MacOS.

```
$ cd /lib/x86_64-linux-gnu/
$ readelf -s libc.so.6 | wc -l
2225      # nb de symboles de la lib. C standard
$ readelf -s libc.so.6 | grep atoi
1643: 00000000000039f50      21 FUNC      GLOBAL DEFAULT ...
$ readelf -s libm.so.6 | wc -l
427       # nb de symboles de la lib. math
$ readelf -s libm.so.6 | grep sqrt
117: 00000000000034590      38 FUNC      WEAK      DEFAULT ...
```

Les dépendances de l'édition de liens dynamique

L'édition de liens réalisée lors de la compilation ne résout pas les liens vers les fonctions de bibliothèques dynamiques mais insère des appels systèmes pour réaliser la liaison à l'exécution (après vérification de l'existence de ces librairies). La commande `ldd` permet de lister l'ensemble des bibliothèques partagées requises par un exécutable.

```
$ ldd mm
linux-vdso.so.1 => (0x00007fff8d073000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f191...
/lib64/ld-linux-x86-64.so.2 (0x00007f191bc66000)
```

Fabriquer et utiliser une librairie dynamique

```
$ gcc -fPIC -c *.c
$ gcc -shared -Wl,-soname,libtruc.so.1 -o libtruc.so.1.0 *.o
$ ln -s libtruc.so.1.0 libtruc.so.1
$ ln -s libtruc.so.1 libtruc.so
$ export LD_LIBRARY_PATH=`pwd`:LD_LIBRARY_PATH
```

- `-fPIC` : position independant code
- `-shared` : librairie partagée
- `-Wl` : fait passer l'option `-soname libtruc.so.1` à l'éditeur de lien
- `-soname` : nom interne de la librairie inscrit par ld (Link eDit)
- `ln` : pour les références avec des versions différentes

Pour compiler mon programme en utilisant ma librairie `truc` :

```
gcc -o monprog monprog.c -ltruc; ./monprog
```

L'édition de liens statique

Cette phase permet de **résoudre** complètement les références non définies des fichiers objets entre eux et avec les bibliothèques en construisant un binaire exécutable **autonome**. Une bibliothèque statique est constituée d'un ensemble de **fichiers** au format objet. Son extension est : `.a`.

```
$ cd /usr/lib/x86_64-linux-gnu
$ ar -t libc.a | grep stdio
stdio.o
xdr_stdio.o
$ ar -t libc.a | wc -l
1561      # nb de fichiers objets contenus
$ nm libc.a | grep " T " | wc -l
2375      # nb de fonctions dans le segment de code (Text)
$ nm libc.a | grep fprintf
fprintf.o:
0000000000000000 T fprintf
...
```


Fabriquer et utiliser une librairie statique

Pour construire une librairie statique :

```
ar rs libmabib.a a.o b.o
```

- r : Remplace les .o
- s : maj la table des Symboles

Pour utiliser mabib :

```
gcc -static -o monprog monprog.o -lmabib -L.; ./monprog
```

- -static : édition statique
- -lx : librairie statique libx.a
- -L. : chercher dans .

Plan

3 Représentation de l'information

- Unités (bits, octets, ...)
- Représentation des entiers
- Nombres flottants
- Caractères

Plan

3 Représentation de l'information

Les Unités I

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

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

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

Les Unités II

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

Exemple : mot de 16 bits

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

Plan

3 Représentation de l'information

- Unités (bits, octets, ...)
- Représentation des entiers
- Nombres flottants
- Caractères

Les Unités III

- Unités multiples : nouvelles normes internationales (1999)

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

Représentation en base 2 I

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

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

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

Représentation en base 2 II

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

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

Représentation en base 2^p

Plus compact en base 2^p : découper le mot $b_{n-1}b_{n-2}...b_0$ en tranches de p bits à partir de la droite (la dernière tranche est complétée par des 0 à gauche si n n'est pas multiple de p). Chacune des tranches obtenues est la représentation en base 2 d'un chiffre de x représenté en base 2^p .

p=3 (représentation **octale**) ou p=4 (représentation **hexadécimale**).

En représentation hexadécimale, les valeurs 10 à 15 sont représentées par les symboles A à F. On préfixe le nombre octal par 0, le nombre hexa par 0x.

Exemples

x=200 et n=8

en binaire : 11 001 000 (128+64+8) : 200

en octal : 3 1 0 (3*64+8) : 0310

en hexadécimal : C 8 (12*16+8) : 0xC8

Opérations

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

exemple sur 8 bits

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

Les autres opérations dépendent de la représentation des entiers négatifs.

Le complément à 1 (C1)

Les entiers positifs sont en RBNS. Les entiers négatifs $-|x|$ sont obtenus par inversion des bits de la RBNS de $|x|$. Le bit de poids n-1 indique le signe (0 positif, 1 négatif).

Intervalle de définition : $[-2^{n-1} + 1, 2^{n-1} - 1]$

Exemples sur un octet

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

Inconvénients :

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

Le second problème est résolu si l'on ajoute 1 lorsqu'on additionne un positif et un négatif : $3+1+ -3=0$ (00...0) et $4+1+ -3=1$ (00...01)

D'où l'idée de la représentation en Complément à 2.

Le complément à 2 (C2) I

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

Intervalle de définition : $[-2^{n-1}, 2^{n-1} - 1]$

Exemples sur un octet $[-128, +127]$:

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

Inconvénient :

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

L'excédent à $2^{n-1} - 1$

Tout nombre x est représenté par $x + 2^{n-1} - 1$ en RBNS. Attention, le bit de signe est inversé (0 négatif, 1 positif).

Intervalle de définition : $[-2^{n-1} + 1, 2^{n-1}]$

Exemples sur un octet :

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

Avantage :

- représentation uniforme des entiers relatifs ;

Inconvénients :

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

Le complément à 2 (C2) II

Avantage fondamental du C2

L'addition binaire fonctionne correctement : l'addition de deux entiers en C2 donne le bon résultat en C2

$-3+3=0$: 1111 1101+0000 0011=(1) 0000 0000

$-3 + -3 = -6$: 1111 1101+1111 1101=(1) 1111 1010

Remarquons ici que le positionnement du Carry à 1 n'indique pas un dépassement de capacité !

Dépassement de capacité en C2 : l'Overflow Flag (OF).

$127+127=-2$: 0111 1111 + 0111 1111=(0) 1111 1110

$-128+-128=0$: 1000 0000 + 1000 0000=(1) 0000 0000

$-127+-128=1$: 1000 0001 + 1000 0000=(1) 0000 0001

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

Opérations en RBNS et C2

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

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

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

Multiplication et Division La multiplication $x*y$ peut être réalisée par y additions successives de x tandis que la division peut être obtenue par soustractions successives et incrémentation d'un compteur tant que le dividende est supérieur à 0 (pas efficace $O(2^n)$).

Cependant, la plupart des processeurs fournissent des instructions MUL et DIV efficaces en $O(n)$ par décalage et addition.

Exemples : $13 * 12 = 13 * 2^3 + 13 * 2^2 = 13 \ll 3 + 13 \ll 2$

$126/16 = 126/2^4 = 1216 \gg 4$

Codage DCB

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

Le codage DCB des nombres à virgule nécessite de coder :

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

Inconvénients

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

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

Plan

3 Représentation de l'information

- Unités (bits, octets, ...)
- Représentation des entiers
- Nombres flottants
- Caractères

Virgule flottante

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

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

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

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

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

En binaire normalisé

Exemple : $7,25_{10} = 111,01_2 = 1,1101 * 2^2$

$4+2+1+0,25=(1+0,5+0,25+0,0625)*4$

Virgule Flottante binaire

Remarques :

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

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

Virgule Flottante en machine

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

Standardisation

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

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

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

Virgule Flottante (fin)

Remarques

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

double précision sur 64 bits (double)

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

Virgule Flottante simple précision

4 octets ordonnés : signe, exposant, mantisse.

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

Exemples $33,0 = +10001,0 = +1,00012^5$ représenté par : 0 1000 1000 0000 100... c'est-à-dire : 0x 42 04 00 00
 $-5,25 = -101,01 = -1,01012^2$ représenté par : 1 1000 0001 0101 000... c'est-à-dire : 0x C0 A8 00 00

Nombres spéciaux

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

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

Plan

3 Représentation de l'information

- Unités (bits, octets, ...)
- Représentation des entiers
- Nombres flottants
- Caractères

Représentation des caractères

Symboles

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

Utilisation

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

Code ou jeu de car. : Ensemble de caractères associés aux mots binaires les représentant. Historiquement, les codes avaient une taille fixe (7 ou 8 ou 16 bits).

- ASCII (7) : alphabet anglais ;
- ISO 8859-1 ou ISO Latin-1 (8) : code national français (é, à, ...);
- UniCode (16 puis 32) : codage universel (mandarin, cyrillique, ...);
- UTF8 : codage de longueur variable d'UniCode : 1 caractère codé sur 1 à 4 octets.

ASCII

American Standard Code for Information Interchange Ce très universel code à 7 bits fournit 128 caractères (0..127) divisé en 2 parties :

- 32 caractères de fonction (de contrôle) permettant de commander les périphériques (0..31);
- 96 caractères imprimables (32..127).

Codes de contrôle importants

0 Null 9 Horizontal Tabulation 10 Line Feed
13 Carriage Return

Codes imprimables importants

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

Code ASCII

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

ISO 8859-1 et utf-8

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

TABLE – ISO Latin-1

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

TABLE – utf-8

Caractères UTF-8 et char C

char C

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

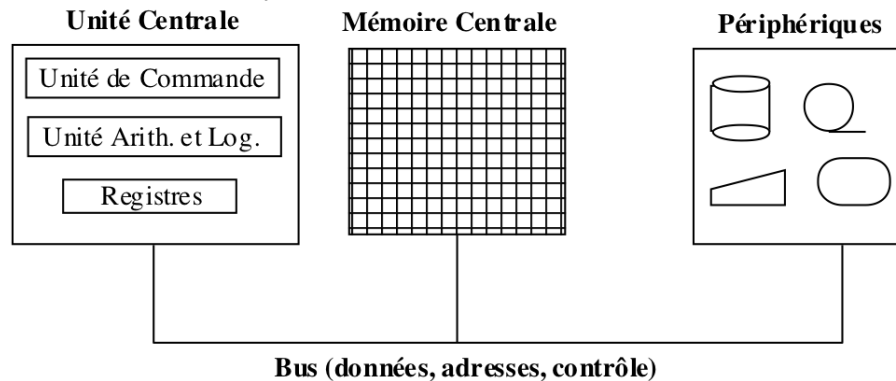
- une chaîne littérale contenant des caractères accentués a une taille (`strlen`) supérieure au nombre de ses lettres
- la bibliothèque standard C permet de manipuler des caractères larges `wchar_t`
- l'entête `wchar.h` contient les déclarations des fonctions utiles telles que `wint_t fgetwc(FILE *stream);`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

Plan

4 Structure des ordinateurs

Modèle de Von Neumann

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



Modèle de programmation

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

Plan

4 Structure des ordinateurs

- L'Unité Centrale
 - La Mémoire Centrale (MC)
 - Les périphériques
 - Les Bus de données, d'adresse et de contrôle
 - Améliorer les performances

L'Unité Centrale (UC) I

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

L'Unité Centrale (UC) III

de travail utilisés pour mémoriser les paramètres des fonctions, les variables sur lesquels on réalise des opérations :

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

L'Unité Centrale (UC) II

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

Algorithme de l'unité de commande I

répéter

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

jusqu'à l'infini

Algorithme de l'unité de commande II

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

L'Unité Arith. et Log. II

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

L'Unité Arith. et Log. I

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

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

Remarques

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

Plan

- 4 **Structure des ordinateurs**
 - L'Unité Centrale
 - La Mémoire Centrale (MC)
 - Les périphériques
 - Les Bus de données, d'adresse et de contrôle
 - Améliorer les performances

La Mémoire Centrale (MC) I

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

Accès à la MC

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

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

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

Contenu/adresse (valeur/nom) I

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

La Mémoire Centrale (MC) II

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

Contenu/adresse (valeur/nom) II

Exemple de représentation MC

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

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

RAM et ROM I

Random Access Memory (RAM)

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

Plan

4 Structure des ordinateurs

- L'Unité Centrale
- La Mémoire Centrale (MC)
- Les périphériques
- Les Bus de données, d'adresse et de contrôle
- Améliorer les performances

RAM et ROM II

Read Only Memory (ROM)

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

Les périphériques I

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

Les périphériques II

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

Support de la mémorisation de masse I

Disques électroniques (SSD)

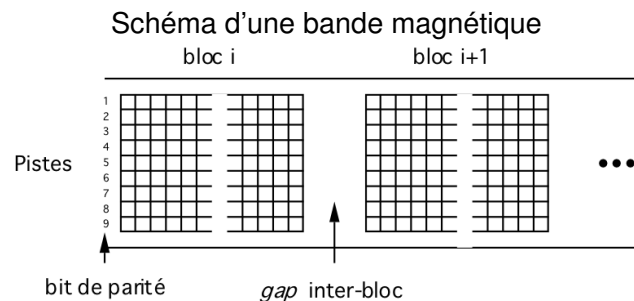
- *solid-state drive* à base de mémoire électronique (Flash)
- plus résistant (MTBF), meilleur débit, plus faible consommation que les disques magnétiques
- beaucoup de config. avec un disque SSD pour l'OS (150 Gio) et un gros disque dur pour les données (plusieurs Tio)

Supports Optiques

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

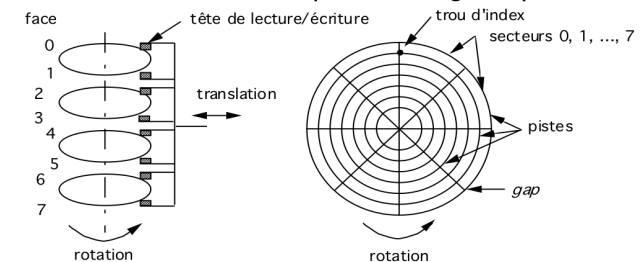
Support de la mémorisation de masse II

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



Support de la mémorisation de masse III

Schéma d'un disque dur magnétique



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

Support de la mémorisation de masse IV

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

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

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

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

Exemple : TAmoyen et transfert d'1 secteur

disque dur 16 faces, 16 Kibi cylindres, 256 secteurs/piste de 4 Kio, tournant à 7200 tours/mn, ayant une vitesse de translation de 2 m/s et une distance entre la première et la dernière piste de 5 cm

$T_{\text{Amoyen}} = (5 \text{ cm}/2)/2 \text{ m/s} + (1/(7200/60 \text{ t/s}))/2 = 12,5 \text{ ms} + 4,2 \text{ ms} = 16,7 \text{ ms}$

Transfert d'1 secteur = $(1/(7200/60 \text{ t/s}))/256 = 32 \cdot 10^{-3} \text{ ms}$

Taux de transfert = $(7200/60 \text{ t/s}) * 256 * 4 \text{ Kio} = 120 \text{ Mo/s}$

Volume = $16 * 16 * 2^8 * 256 * 4 \text{ Kio} = 4 \text{ Tio}$

Pilote, contrôleur d'E/S et IT I

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

Pilote, contrôleur d'E/S et IT II

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

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

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

Plan

4 Structure des ordinateurs

- L'Unité Centrale
- La Mémoire Centrale (MC)
- Les périphériques
- Les Bus de données, d'adresse et de contrôle
- Améliorer les performances

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

- Reset : réinitialisation de l'UC
- R/\overline{W} : indique le sens du transfert vers la MC
- MEM/\overline{IO} : adresse mémoire ou E/S
- Technologiquement, les technologies de bus évoluent rapidement (Vesa Local Bus, ISA, PCI, PCI Express, ATA, SATA, SCSI, ...)
- Actuellement, d'autres types d'architecture (5^e génération, machines systoliques, grid computing) utilisant massivement le parallélisme permettent d'améliorer notablement la vitesse des calculs
- On peut conjecturer que dans l'avenir, d'autres paradigmes de programmation spécifiques à certaines applications induiront de nouvelles architectures

Plan

4 Structure des ordinateurs

- L'Unité Centrale
- La Mémoire Centrale (MC)
- Les périphériques
- Les Bus de données, d'adresse et de contrôle
- Améliorer les performances

Hierarchie mémoire et Cache I

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

- Registres
- Mémoire Centrale
- Mémoire Secondaire

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

Hierarchie mémoire et Cache II

Fonctionnement

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

Hierarchie mémoire et Cache III

Exemple de Cache MC réalisé en SRAM

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

Hierarchie mémoire et Cache IV

Cache Disque

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

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

Pipeline II

avec pipeline à 5 étages temps →

```

i1F i1D i1L i1E i1W
  i2F i2D i2L i2E i2W
    i3F i3D i3L i3E i3W
      i4F i4D i4L i4E i4W

```

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

Pipeline I

Technique de conception de processeur avec plusieurs petites unités de commande placées en série et dédiées à la réalisation d'une tâche spécifique. Plusieurs instructions se chevauchent à l'intérieur même du processeur

Par exemple, décomposition simple d'une instruction en 5 étapes :

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

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

sans pipeline temps →

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

SIMD, DMA, Bus Mastering I

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

SIMD, DMA, Bus Mastering II

- La technique de *Bus Mastering* (contrôle de bus) permet à n'importe quel contrôleur d'E/S de demander et de prendre le contrôle du bus : le maître peut alors communiquer avec n'importe lequel des autres contrôleurs sans passer par l'UC. Cette technique implémentée dans le bus PCI permet à n'importe quel contrôleur de réaliser un DMA. Si l'UC a besoin d'accéder à la mémoire, elle devra attendre de récupérer la maîtrise du bus (pas de temps réel).

Le *chipset* des PCs utilise la technique de *bus mastering* en étant l'interface entre l'UC, la MC et les bus plus ou moins rapides des périphériques (PCI Express, PCI, USB, ...)

Plan

5 La couche Machine

Architecture Multi-coeurs I

Le processeur possède plusieurs coeurs possédant chacun :

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

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

Plan

5 La couche Machine

- Introduction
- Modes d'adressage
- La pile (stack)

La couche Machine I

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

La couche Machine III

Caractéristiques de x86-64

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

La couche Machine II

Architecture x86-64

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

Format des instructions machines I

Une instruction est composée de plusieurs champs :

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

Exemple :

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

Format des instructions machines II

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

Modes d'adressage I

Types de donnée représentés par les opérandes :

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

Remarques :

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

Plan

- 5 La couche Machine
 - Introduction
 - Modes d'adressage
 - La pile (stack)

Adressage immédiat I

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

Exemples :

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

Adressage registre I

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

Exemples :

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

Adressage direct II

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

Adressage direct I

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

Exemples :

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

Adressage indirect I

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

Adressage indirect II

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

Exemples :

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

Adressage indexé (ou basé) II

Exemples :

Z80 indexation par IX et IY

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

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

Adressage indexé (ou basé) I

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

Remarques I

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

Plan

5 La couche Machine

- Introduction
- Modes d'adressage
- La pile (stack)

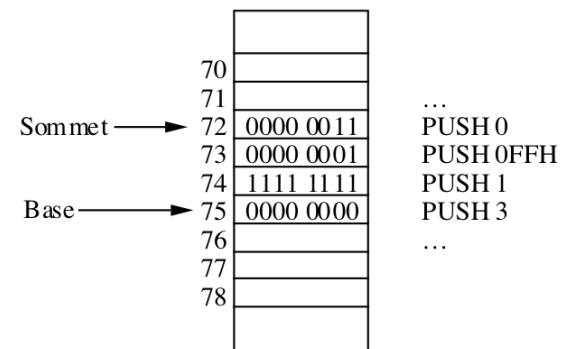
Adressage par pile II

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

Adressage par pile I

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

Adressage par pile III



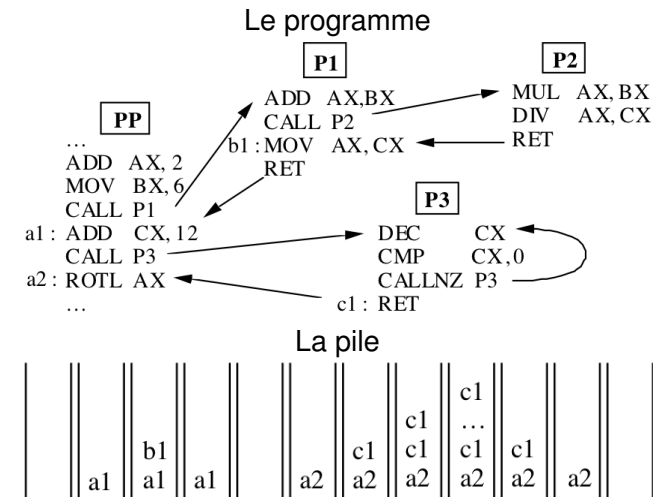
L'appel procédural I

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

Paramètres et Variables locales I

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

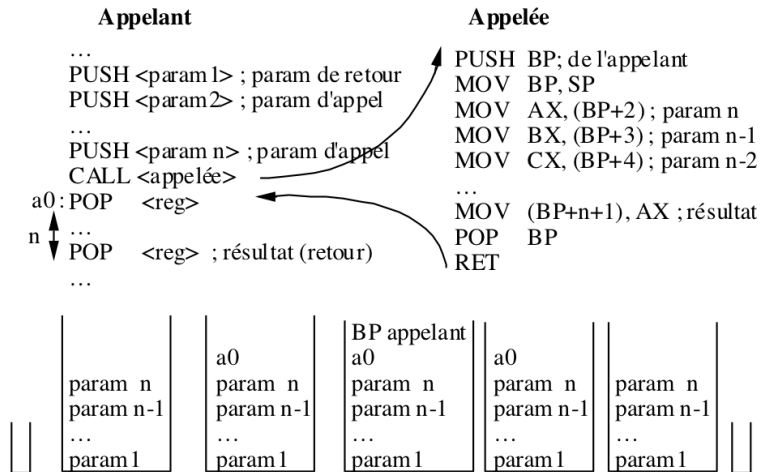
Example I



Paramètres et Variables locales II

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

Exemple de passage de paramètres I



Variables locales ou automatiques I

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

Un exemple complet I

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

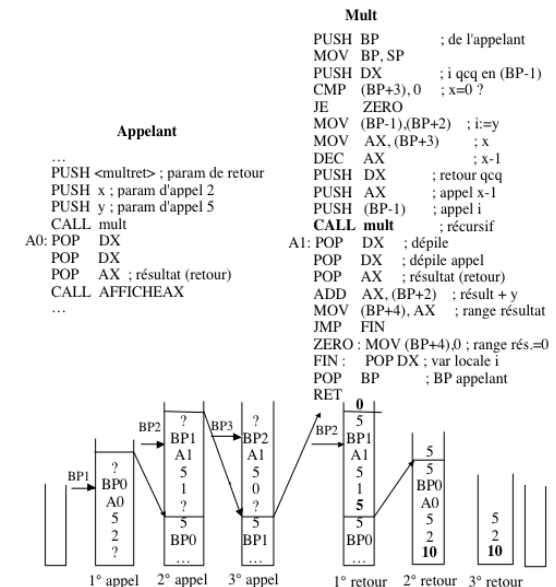
```

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

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

```

Un exemple complet II



Conclusion I

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

Plan

6 Gestion des Fichiers et des Entrées/Sorties

- Périphériques et entrées/sorties
 - Types de Périphériques
 - Quelques Détails
 - Fichiers et opérations sur les fichiers
 - Fichier séquentiel et flot sous Unix
 - Résumé des Appels systèmes UNIX
 - Résumé des fonctions de la bibliothèque d'E/S standard du C

Plan

6 Gestion des Fichiers et des Entrées/Sorties

Rôle et Organisation

La composante du système s'occupant de la *Gestion des Entrées-Sorties (I/O management)* est chargée de la communication avec les périphériques.

L'interface qu'offre le système d'exploitation pour l'accès aux périphériques cherche à uniformiser, à **banaliser** l'accès, c'est-à-dire à rendre la syntaxe de l'appel système indépendant du périphérique.

Quand on écrit `read(descripteur, donnée, taille)`, on ne dit rien du périphérique concerné (disque, clavier, ...).

Le système d'exploitation fait la liaison entre cette demande et le périphérique grâce à la demande d'ouverture.

Du logiciel au matériel on trouve la liaison entre appels système, pilotes et contrôleurs.

Pilote, Contrôleur, etc.

Plusieurs couches interviennent dans la communication avec le périphérique :

- ① L'*appel système* lui-même détermine en fonction de l'entrée-sortie demandée le type de support (faut-il transférer un bloc ? une ligne ?, ...) et par conséquent le périphérique concerné.
- ② Le *pilote*, logiciel du système d'exploitation, établit la liaison entre le système d'exploitation et les actions à demander au matériel du périphérique.
- ③ Le *contrôleur* est une partie intégrante du matériel indépendante de l'UC. Son rôle est de recevoir les demandes du pilote et de les réaliser.

On rappelle que les données sont transférées entre la mémoire et les périphériques par l'intermédiaire de *bus*, ensemble matériel de fils **et** un protocole. Le protocole gère l'exclusivité d'accès et permet à plusieurs périphériques d'utiliser le même bus.

Exemple - suite

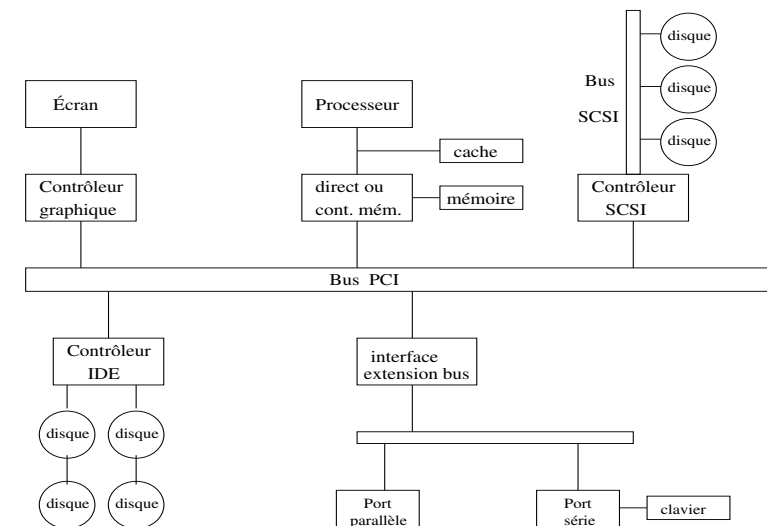
- Le gérant d'interruption réveille le processus demandeur qui exécute le pilote.
- Le pilote recopie une partie du tampon système vers l'espace utilisateur (adresse de la donnée).
- Puis, le pilote retourne un résultat à l'appel noyau permettant au processus de repasser en mode utilisateur.

Exemple

Un processus demande une lecture sur un descripteur de fichier.

- l'appel système passe en mode noyau et vérifie si on peut satisfaire directement, sans accès au périphérique, la demande. Si oui, la demande est satisfaite et le retour au mode utilisateur du processus immédiat. Sinon, on détermine de quel périphérique il s'agit, ici disque, puis la demande est expédiée au pilote concerné.
- Le pilote calcule et convertit la demande en adresse de disque et de secteur, puis invoque le contrôleur. Le processus est alors bloqué en attendant l'interruption, sauf cas spécifique d'attente active.
- Le contrôleur reçoit un numéro de secteur et une instruction (lecture, écriture) ; il recopie le contenu du secteur dans un tampon système en MC. Puis, il lance une interruption de fin d'E/S.

Exemple - fin : Structure Classique



Plan

6 Gestion des Fichiers et des Entrées/Sorties

- Périphériques et entrées/sorties
- Types de Périphériques
- Quelques Détails
- Fichiers et opérations sur les fichiers
- Fichier séquentiel et flot sous Unix
- Résumé des Appels systèmes UNIX
- Résumé des fonctions de la bibliothèque d'E/S standard du C

Plan

6 Gestion des Fichiers et des Entrées/Sorties

- Périphériques et entrées/sorties
- Types de Périphériques
- Quelques Détails
- Fichiers et opérations sur les fichiers
- Fichier séquentiel et flot sous Unix
- Résumé des Appels systèmes UNIX
- Résumé des fonctions de la bibliothèque d'E/S standard du C

Bloc ou Caractère

La distinction majeure en termes de types de périphériques provient de l'unité de transfert :

- **caractère** pour les périphériques dont l'unité est un caractère ou une suite de caractères de taille non déterminée à l'avance ; par exemple le clavier.
- **bloc** pour les périphériques dont l'unité de transfert est un ensemble de taille fixe par catégorie de périphérique ; disque par exemple.

Sous Unix, l'ensemble des périphériques est représenté dans le répertoire `/dev`. On peut y voir que les types de fichiers sont représentés par les lettres **b** pour bloc et **c** pour caractère.

Constatons qu'on vient d'enrichir les types de fichiers connus : on avait jusque là les fichiers réguliers (`-`), les répertoires (`d`), les liens symboliques (`l`), les tubes (`p`) et on ajoute les deux nouveaux.

Contrôleur

Un contrôleur

- reçoit une commande, lire ou écrire, et une donnée dans un tampon local,
- il met en route le matériel si besoin (contrôle de la rotation du disque par exemple), réalise la commande, vérifie qu'elle est faite (test de relecture après écriture),
- avertit qu'elle est faite (interruption).

Il existe des contrôleurs très simples, contrôleurs de hauts-parleurs par exemple et très complexes, contrôleurs disque SCSI, qui disposent de leur propre processeur et sont capables de gérer plusieurs disques simultanément.

Pilote

Un pilote transcrit la demande de l'utilisateur en demande compréhensible par un contrôleur déterminé.

De plus en plus fréquemment, il n'est plus nécessaire de recompiler le noyau du système d'exploitation lorsqu'un nouveau pilote est mise en place pour un nouveau matériel.

Il reste parfois nécessaire de *retoucher* la configuration, ou d'ajouter le nouveau pilote s'il était absent.

Ceci est dû à la création de pilotes génériques permettant de faire la liaison dynamiquement entre un pilote et le système.

Extension : On peut gérer sous forme de pseudo-périphériques des matériels divers : la mémoire, le noyau du système, disques virtuels. . .

Modification du Comportement

On peut modifier ce comportement bloquant canonique :

- L'appel système *fcntl()* permet de modifier le comportement habituel de tout fichier, en particulier de basculer les comportements bloquant et non-bloquant.
- Les fonctions *tcgetattr()* et *tcsetattr()* permettent de modifier **tous** les paramètres de gestion du clavier.

Attention : passer au comportement non-canonique, implique que toutes les touches saisies sont validées immédiatement, sans possibilité de correction ! Par exemple, la touche d'effacement devient un caractère normal (0x08).

Exemple - Configuration du Clavier

Le fonctionnement classique, pour toutes les lectures au clavier, consiste à rentrer une suite de caractères et la valider par la touche *entrée*.

Ce fonctionnement est dit **bloquant, canonique**.

bloquant car l'instruction de lecture ne sera débloquée que lorsque la donnée sera disponible en mémoire,

canonique car la chaîne saisie est validée par *entrée*, et avant cette validation, on peut effacer, revenir sur ce qui est déjà frappé autant que nécessaire.

Le mode **canonique** est spécifique du clavier alors que le mode **bloquant** s'applique à tout fichier.

Plan

6 Gestion des Fichiers et des Entrées/Sorties

- Périphériques et entrées/sorties
- Types de Périphériques
- Quelques Détails
- Fichiers et opérations sur les fichiers
- Fichier séquentiel et flot sous Unix
- Résumé des Appels systèmes UNIX
- Résumé des fonctions de la bibliothèque d'E/S standard du C

Les Fichiers : définitions I

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

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

Les Fichiers : définitions II

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

Opérations et modes d'accès I

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

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

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

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

lecture consultation d'un ou plusieurs articles

écriture insertion ou suppression d'un article

Opérations et modes d'accès II

déplacement déplacement sur un article

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

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

Plan

6 Gestion des Fichiers et des Entrées/Sorties

- Périphériques et entrées/sorties
- Types de Périphériques
- Quelques Détails
- Fichiers et opérations sur les fichiers
- Fichier séquentiel et flot sous Unix
- Résumé des Appels systèmes UNIX
- Résumé des fonctions de la bibliothèque d'E/S standard du C

Fichier séquentiel (2) et flot(3)

- Un fichier (séquentiel) est une abstraction constitué d'une séquence de caractères muni d'une tête de lecture-écriture auto-incrémentée après chaque opération :
 - de lecture (read) ;
 - et/ou d'écriture (write) ;
- Un flot est une abstraction de la **bibliothèque** standard englobant un fichier séquentiel. Un flot utilise des tampons (*buffer*) pour ses lectures et écritures, ce qui le rend plus efficace.

fichiers et flots ouverts

Tout processus lancé depuis un `bash` démarre avec 3 flots (et fichiers) déjà ouverts :

- `stdin`, (0) le flot (fichier) d'entrée standard ;
- `stdout`, (1) le flot (fichier) de sortie standard ;
- `stderr`, (2) le flot (fichier) de sortie d'erreur standard ;

Habituellement, ils sont associés aux périphériques suivants :

0 le clavier ;

1,2 l'écran (la fenêtre du terminal en mode graphique) ;

Ces trois flots peuvent être redirigés vers des fichiers de l'arborescence depuis le `bash` :

```
monprog < ./ficin.txt > ./ficout.txt 2> ./ficerr.txt
```

Ouverture d'un fichier

Un fichier doit être ouvert ((f)open) sur un fichier de l'arborescence afin que le système réserve des tampons en MC :

```
int open(char* path, int mode, int droits)
```

- `path` : désignation du fichier ou du périphérique ;
- `mode` :
`O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_APPEND`, `O_CREAT` ;
- `droits` : seulement pour la création de fichier ;
- `résultat` : entier **descripteur** du fichier ou -1 si erreur ;

Un **pointeur courant** est positionné en début de flot lors de l'ouverture (sauf pour `APPEND`), et il est automatiquement incrémenté au fur et à mesure des lectures ou écritures. Lors d'une lecture en fin de flot, le résultat indique une "fin de fichier" (End Of File).

Algorithmique

Afin de généraliser notre discours à d'autres systèmes qu'Unix, on utilisera une notation algorithmique afin de décrire des mécanismes systèmes puis on traduira en C pour Unix.

comptage des octets d'un fichier

Données : chemin chaîne désignant le fichier

Résultat : entier nombre d'octets

Fonction `compte(chemin)` : entier ;

entier `nb=0`, `fd`;

`fd=ouvrir(chemin, LECTURE)`;

tant que `EOF != lireCar(fd)` **faire**

└ `nb++`;

`fermer(fd)`;

return `nb`;

Traduction en C avec des appels systèmes : compte.c

```
int compte(char *chemin) {
    int nb=0, fd;
    fd=open(chemin, O_RDONLY);
    if (fd<0) {
        fprintf(stderr, "Impossible d'ouvrir le fichier %s
        ↪ !\n", chemin);
        exit(EXIT_FAILURE);
    }
    char c;
    while(1==read(fd, &c, 1)) {
        nb++;
    }
    close(fd);
    return nb;
}
```

Traduction en C avec des appels systèmes : compte.c

```
int main(int n, char *argv[], char *env[]) {
    if (n!=2) {
        fprintf(stderr, "Syntaxe : %s chemin !\n", argv[0]);
        return EXIT_FAILURE;
    }
    printf("%s contient %d octets
    ↪ !\n", argv[1], compte(argv[1]));
    return EXIT_SUCCESS;
}
```

Traduction en C avec des fonctions de bib. : compte2.c

```
int compte(char *chemin) {
    int nb=0;
    FILE *f=fopen(chemin, "r");
    if (f==NULL) {
        fprintf(stderr, "Impossible d'ouvrir le fichier %s
        ↪ !\n",
        chemin);
        exit(EXIT_FAILURE);
    }
    int c;
    while(EOF!=(c=fgetc(f))) {
        nb++;
    }
    fclose(f);
}
```


Traduction en C avec des fonctions de bib. : compte2.c II

```
return nb;
}
```

Création de fichier I

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

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

path nom ou chemin d'accès au fichier ("../Monrep/toto.txt")

droits droits d'accès qui seront masqués par `umask` (022)

return un descripteur entier positif ou -1 si erreur

Plan

6 Gestion des Fichiers et des Entrées/Sorties

- Périphériques et entrées/sorties
- Types de Périphériques
- Quelques Détails
- Fichiers et opérations sur les fichiers
- Fichier séquentiel et flot sous Unix
- **Résumé des Appels systèmes UNIX**
- Résumé des fonctions de la bibliothèque d'E/S standard du C

Création de fichier II

Par exemple

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

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

Ecriture dans un fichier

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

desc le descripteur retourné par `open`

buf la chaîne de caractères qu'on veut écrire

nboctets le nombre d'octets qu'on tente d'écrire

return le nb d'octets écrits dans le fichier et dont a avancé la tête de lecture, -1 si erreur

surcharge les octets écrits écrasent ceux qui existaient auparavant ; si on est en fin de fichier, ce dernier est allongé automatiquement

Ouverture, fermeture

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

path nom ou chemin d'accès au fichier

flags `O_RDONLY` xor `O_WRONLY` xor `O_RDWR`, `O_APPEND`, `O_TRUNC`

droits droits d'accès qui seront masqués par `umask` (0644)

return un descripteur entier positif ou -1 si erreur

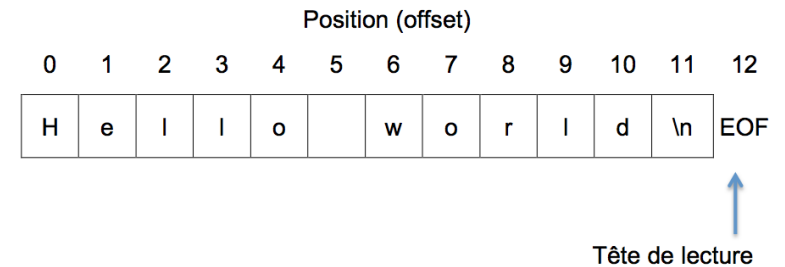
ouvre un fichier selon un mode (R|W|RW) et positionne le pointeur en début (fin si `O_APPEND`) de fichier

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

ferme le fichier (désalloue les tampons systèmes)

Un exemple complet

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



Lecture

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

desc le descripteur retourné par `open`

buf la chaîne de caractères **allouée** dans laquelle vont être stockés les octets lus

nboctets le nombre d'octets qu'on tente de lire

return le nb d'octets lus dans le fichier et dont a avancé la tête de lecture, 0 si fin de fichier, -1 si erreur

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

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

fd le descripteur retourné par `open`

offset le déplacement à effectuer (entier signé long)

whence `SEEK_SET` xor `SEEK_CUR` xor `SEEK_END`, position à partir de laquelle se déplacer (début, courante, fin)

return nouvelle position courante ou -1 si erreur

Plan

6 Gestion des Fichiers et des Entrées/Sorties

- Périphériques et entrées/sorties
- Types de Périphériques
- Quelques Détails
- Fichiers et opérations sur les fichiers
- Fichier séquentiel et flot sous Unix
- Résumé des Appels systèmes UNIX
- Résumé des fonctions de la bibliothèque d'E/S standard du C

Autres appels systèmes I

int dup(int desc) duplication de descripteur (redirection d'E/S) :

```
d=creat("ficredir", 0666); close(1); dup(d);
```

access teste les droits d'accès

link, symlink création de lien dur ou symbolique

unlink suppression d'un lien et possiblement du fichier

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

chdir change répertoire courant

chown, chmod change propriétaire, droits d'accès

mkdir, rmdir création, suppression de répertoire

Bibliothèque versus appels systèmes

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

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

Flot (stream)

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

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

Liste des fonctions de la bibliothèque standard C II

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

vidage du tampon sur le fichier

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

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

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

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

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

Liste des fonctions de la bibliothèque standard C I

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

`path` chemin d'accès au fichier

`mode`

"r" lecture

"r+" lecture et écriture

"w" création ou troncature du fichier ouvert en écriture seulement (droits 0666 en création)

"w+" création ou troncature du fichier ouvert en lecture/écriture

"a" ouverture en écriture seulement, création si nécessaire, position en fin de fichier

"a+" ouverture en lecture/écriture, création si nécessaire, écritures en fin de fichier, lecture en début

Liste des fonctions de la bibliothèque standard C III

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

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

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

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

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

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

vidage du tampon en écriture vers le fichier

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

Liste des fonctions de la bibliothèque standard C IV

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

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

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

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

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

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

écriture formatée dans une chaîne.

Liste des fonctions de la bibliothèque standard C VI

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

copie (bornée par n)

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

recherche du 1er c dans s

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

recherche d'un char de s2 dans s1

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

recherche d'une sous-chaîne (facteur)

Liste des fonctions de la bibliothèque standard C V

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

lecture selon un certain format dans une chaîne

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

longueur de s

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

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

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

comparaison (bornée par n) ; 0 si égales

Plan

7 Gestion des processus

Plan

7 Gestion des processus

- Qu'est-ce qu'un processus
- Vie des processus
- Changement de contexte
- Scénario de vie de processus
- Observation des processus
- Génération de processus
- Recouvrement de processus

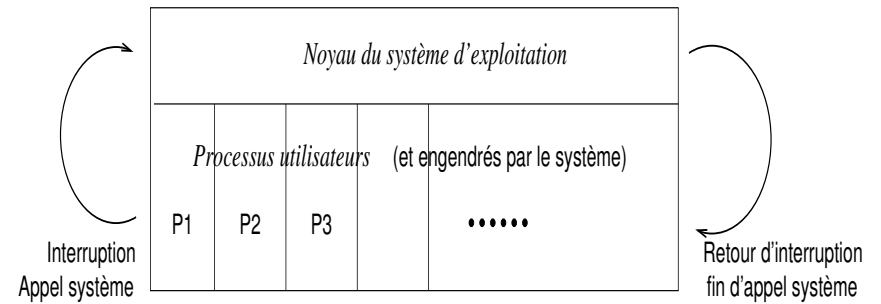
Table des processus

La table des processus contient, par processus, un ensemble d'informations relatives à chaque composante du système. Un tout petit extrait :

| G. processus | G. mémoire | G. fichiers |
|--|--------------------------------|--|
| CO, PP, PSW Temps UC identités état | ptrs segments gest. signaux | descripteurs masque répertoire travail |

Attention : Le système gère beaucoup d'autres tables : table des fichiers ouverts, de l'occupation mémoire, des utilisateurs connectés, des files d'attente, etc.

schéma général système



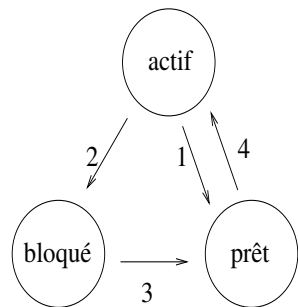
Plan

7 Gestion des processus

- Qu'est-ce qu'un processus
- Vie des processus
- Changement de contexte
- Scénario de vie de processus
- Observation des processus
- Génération de processus
- Recouvrement de processus

États d'un processus

On commence par les états de base :



actif tient la ressource UC

prêt seule la ressource UC lui manque

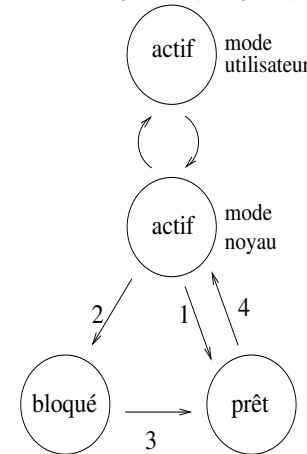
bloqué manque au moins une autre ressource

Important : le passage à l'état actif ne peut se faire que par l'état prêt.

Exercice : citer un exemple pour chaque cas de changement d'état.

Un peu plus

On complète (un peu) ces états de base :



Déjà vu : les appels système, les gérants d'interruption font passer du mode utilisateur au mode noyau ; les retours de ces appels font le passage inverse. Compléments plus loin.

En dehors des changements entre modes *actif utilisateur* et *actif noyau*, tous les autres changements d'état ne peuvent se produire qu'en mode noyau. Heureusement. . .

question : pourquoi ?

Plan

7 Gestion des processus

- Qu'est-ce qu'un processus
- Vie des processus
- **Changement de contexte**
- Scénario de vie de processus
- Observation des processus
- Génération de processus
- Recouvrement de processus

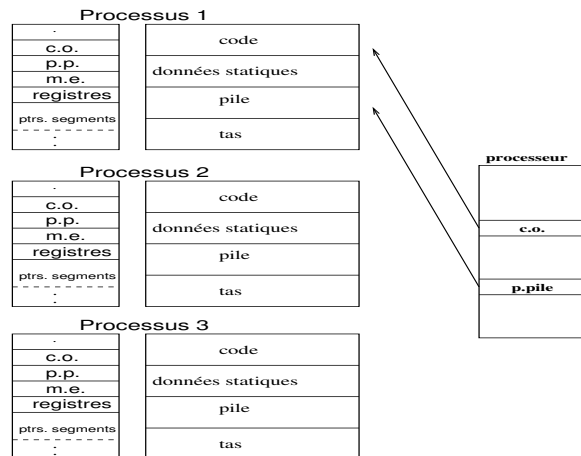
Principe du changement de contexte

- **le système s'exécute dans le contexte du processus actif ;**
- on reconnaît le processus actif car c'est le seul processus vers qui pointent le CO et le pointeur de pile -SP- du processeur (système mono-processeur) ; il est aussi désigné par l'état *actif* dans la table des processus ;
- si ce processus doit s'interrompre temporairement, il faut sauvegarder tout les éléments qui risquent de disparaître et les restituer lorsqu'il pourra continuer.

On dit que le système effectue un *changement de contexte*, ou un *basculement de contexte* (*context switch*).

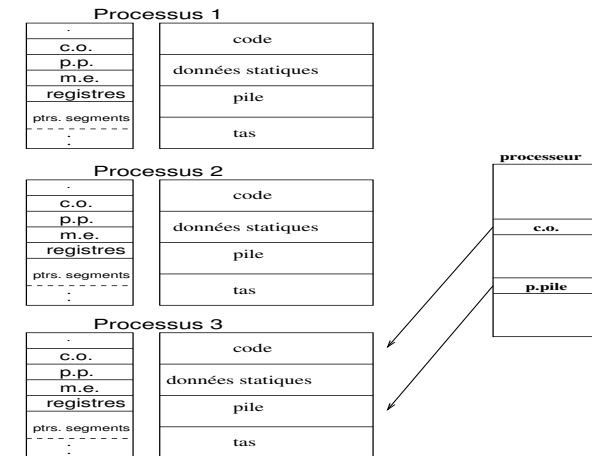
Ce changement se produit lorsque le processus actif passe à l'état bloqué ou prêt et qu'un autre processus devient actif.

Un processus actif



Un autre processus actif

Sauvegarde du contexte de P1, restauration du contexte de P3.



Réalisation des changements de contextes

Lorsque le processus actif passe au mode noyau, il y aura exécution d'une fonction du noyau : soit un gérant d'interruption, soit la fonction appelée directement par ce processus. la fonction du noyau appelée s'exécute dans le contexte du processus actif courant. Cette exécution va invariablement finir par l'appel à l'**ordonnanceur** (*scheduler*).

Déroulement d'un appel noyau

On peut décrire le déroulement d'un appel noyau :

- ① passage du processus actif en mode noyau ;
- ② exécution de l'appel noyau (appelé aussi *routine*) ;
- ③ cet appel modifie l'état du processus actif, sauvegarde son contexte ;
- ④ appel de l'ordonnanceur qui procède à l'élection ; **remarque** : la fonction noyau appelée n'est **pas terminée** ;
- ⑤ l'ordonnanceur restaure le contexte du processus élu et le marque *actif* ;
- ⑥ fin de l'ordonnanceur (*return*) dans le contexte du nouvel élu ;
- ⑦ fin de l'appel ayant provoqué précédemment la suspension de l'élu ;
- ⑧ passage de l'élu en mode utilisateur et suite de son exécution.

Interrogation orale

Questions :

- ❶ Expliquer pourquoi la fonction noyau appelée n'est pas terminée ;
- ❷ Pour tous les processus en attente, c'est-à-dire tous sauf le processus actif, quel est l'état de leur pile ? quelle est la dernière fonction empilée ?
- ❸ Dans quel mode d'exécution se trouvent tous les processus en attente ?

Schéma algorithmique ordonnanceur

tant que pas de processus élu faire

consultar table processus ;

chosir celui de plus haut priorité parmi les prêts;

si pas d'élu alors

attendre ;

//jusqu'à nouvelle interruption (processeur à l'état *latent*)

marquer ce processus actif ;

basculer le contexte ;

return ;

//le processus continue son exécution

Rôle de l'ordonnanceur

Règles à respecter :

- élire parmi les processus **prêts** celui qui deviendra actif ;
- l'élu est celui de plus haute priorité compte tenu de la **politique** d'allocation de l'UC du système (vaste programme...) ;
- effectuer un changement de contexte : sauvegarder celui du processus courant et restituer celui de l'élu.

Plan

- ❶ Gestion des processus
 - Qu'est-ce qu'un processus
 - Vie des processus
 - Changement de contexte
 - **Scénario de vie de processus**
 - Observation des processus
 - Génération de processus
 - Recouvrement de processus

Scénario - Étape Initiale

On suppose trois processus, P1, P2 et P3, tels que P1 est *actif*, P2 et P3 sont dans l'état *prêt*

P1 possède donc la ressource UC. On suppose qu'il ne consomme pas entièrement son quantum de temps, car il fait une demande de lecture d'une donnée sur disque.

Les étapes suivantes vont se dérouler :

- ❶ P1 passe en mode privilégié en faisant l'appel système *read()* ;
- ❷ l'exécution de *read()* va commencer, puis lancer la demande de lecture physique qui sera prise en charge par une entité extérieure dépendant du périphérique concerné (contrôleur disque, contrôleur clavier, ...);
- ❸ l'état de P1 sera passé à *bloqué* et son contexte sauvegardé ;
- ❹ enfin, *read* va appeler l'ordonnanceur.

Scénario - Étape 2

Choix possibles pour l'ordonnanceur : P2 ou P3 ; on suppose que c'est P2 qui est élu. Les étapes suivantes sont :

- ❶ l'état de P2 est passé à *actif* ; on rappelle qu'il est en mode privilégié d'exécution ;
- ❷ le contexte de P2 est restauré ;
- ❸ l'horloge programmable allouant les quantum de temps est réinitialisée à la valeur fixée dans le système ;
- ❹ fin de l'ordonnanceur : le CO est restitué à partir de la pile de P2 (adresse de retour) ;
- ❺ fin de l'appel noyau ou de l'interruption qui avait provoqué l'arrêt précédent de P2 et P2 repasse alors en mode utilisateur.

Suite Étape 2

- ❻ suite de l'exécution de P2 ; on suppose que P2 ne fait aucune opération d'entrée-sortie et qu'aucun événement ne vient le perturber ; P2 consomme ainsi entièrement son quantum de temps ;
- ❼ il y a interruption d'horloge ;
- ❽ passage en mode noyau et exécution du gérant d'interruption d'horloge qui passe P2 à l'état *prêt* ;
- ❾ sauvegarde du contexte de P2 par le gérant ;
- ❿ fin du gérant (presque) : appel ordonnanceur.

Remarque : On dit dans cette situation que P2 a été *préempté* et que le système d'exploitation qui opère fait de la *préemption*.

Scénario - Étape 3

Choix possibles pour l'ordonnanceur : P2 ou P3 ; on suppose P3 élu ; **rappel rapide** : P3 passe à l'état actif, son contexte est restauré, il est en mode noyau et l'horloge est réinitialisée. Suite du scénario :

- ❶ P3 est en cours d'exécution ; on suppose que la lecture demandée par P1 est (enfin) prête ; alors,
- ❷ P3 est interrompu par une interruption disque ;
- ❸ il y a passage en mode noyau et exécution du gérant d'interruption disque ; la donnée lue est donc disponible en mémoire, dans un espace tampon du système ; d'autres situations sont possibles ici, selon la gestion des transferts entre disques et mémoire centrale, mais le principe de l'interruption reste ;
- ❹ P1 est passé à l'état *prêt* (on dit que P1 est *réveillé*) ;
- ❺ P3 est **aussi** passé à l'état *prêt* ;
- ❻ après la sauvegarde du contexte de P3, appel de l'ordonnanceur.

Remarques

Noter l'exécution de la routine d'interruption disque sur le compte et dans le contexte de P3 qui n'est pas concerné et se voit interrompu et délogé.

Noter aussi l'instant où se produit l'interruption lors d'une entrée-sortie :

en **entrée** lorsque la donnée (le secteur lu, la ligne entrée par l'utilisateur, le clic souris) est disponible en mémoire, en **sortie** lorsque la donnée est transférée de l'espace du processus vers le tampon système (on peut modifier son contenu dans l'espace du processus)

Suite Étape 4

- ④ passage en mode noyau et réalisation de *exit()* ; un ensemble d'opérations de nettoyage est lancé : appel de destructeurs éventuels, fermeture des fichiers encore ouverts, opérations comptables, restitution de l'espace mémoire occupé par P1, signalement de sa fin à ses descendants (voir plus loin, la descendance des processus), nettoyage de l'entrée P1 dans la table des processus, ...
- ⑤ appel ordonnanceur : P2 et P3 sont prêts.

Scénario - Étape 4

Choix possibles pour l'ordonnanceur : P1, P2 ou P3.

On suppose que P1 est élu ; pour sa mise en place, voir le rappel rapide ci-avant. Déroulement de la suite :

- ① *read()* continue son exécution pour P1 et amène le contenu du tampon disque dans la mémoire de P1 ; **exemple** : si P1 a fait *read(monfich,&erlude,sizeof(int))* la donnée *erlude* sera remplie à partir du tampon disque ;
- ② fin d'exécution de *read()* entraînant le passage de P1 en mode utilisateur ;
- ③ on suppose que P1 continue en faisant quelques instructions, puis se termine ; il fait donc appel à *exit()*, qui est un appel noyau ;

D'autres soucis ?

Il est temps d'ajouter quelques nouveautés : des problèmes non encore traités.

- Comment se fait la génération des processus ? Voir paragraphe suivant.

ou des questions :

- Que se passe-t-il s'il n'y a aucun processus prêt ? Voir *état latent* du processeur dans la bibliographie,
- Quel est le lien entre les appels *kill()* et *exit()* ? Voir la communication entre processus plus loin.

Plan

7 Gestion des processus

- Qu'est-ce qu'un processus
- Vie des processus
- Changement de contexte
- Scénario de vie de processus
- **Observation des processus**
- Génération de processus
- Recouvrement de processus

commande ps II

- TTY est le terminal d'attachement du pus : ici deux onglets d'une même application Terminal présents dans /dev/pts/ (pseudo terminaux)
- STATE état du pus :
 - S sleep interruptible (attente d'un événement)
 - s leader de Session
 - + groupe des pus d'avant-plan
 - Run désigne les pus éligibles (prêts) ou l'élú.
- TIME indique la durée passée sur le processeur en mode noyau et (+) en mode utilisateur

D'autres options permettent de visualiser d'autres colonnes :

commande ps I

Cette commande liste les processus selon différentes syntaxes d'options (BSD, standard).

```
$ ps f
  PID TTY          STAT       TIME COMMAND
 19701 pts/1        Ss           0:00  bash
 20173 pts/1        S+           0:00  \_ man ps
 20183 pts/1        S+           0:00  \_ less
 17752 pts/0        Ss           0:00  bash
 20245 pts/0        R+           0:00  \_ ps f
```

- f (forest) représente la relation parent/enfant entre processus
- PID est l'identifiant de processus

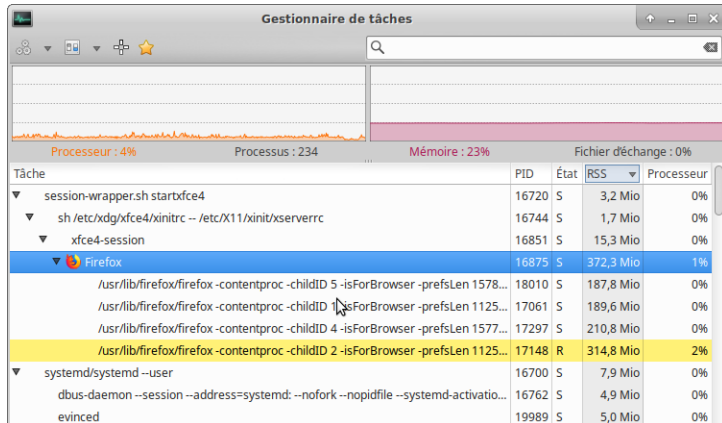
commande ps III

```
$ ps fux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
mmeynard 19449   0.0   0.0  14120   3176 ?        S    10:25   0:00 | \_ /bin/bash /n
mmeynard 19450   0.8   3.5 2144256 289652 ?        Sl   10:25   1:04 | \_ \_ ./thunde
mmeynard 16942   0.0   0.1 173336 14784 ?        S    09:52   0:00 \_ /usr/lib/x86_64-
mmeynard 16947   0.0   0.2 331292 18844 ?        Sl   09:52   0:00 \_ /usr/lib/x86_64-
mmeynard 16985   0.0   0.3 343600 25772 ?        Sl   09:52   0:00 \_ /usr/lib/x86_64-
mmeynard 16989   0.0   0.4 692892 34008 ?        Sl   09:52   0:08 \_ /usr/lib/x86_64-
mmeynard 17011   0.0   0.3 329712 24844 ?        Sl   09:52   0:00 \_ /usr/lib/x86_64-
mmeynard 17748   0.1   0.5 646612 41264 ?        Sl   09:53   0:10 \_ /usr/bin/xfce4-t
mmeynard 17752   0.0   0.0  14596   3824 pts/0    Ss   09:53   0:00 \_ bash
mmeynard 20437   0.0   0.9 404980 79280 pts/0    Sl   11:09   0:03 | \_ \_ emacs
mmeynard 23313   0.0   0.0  30596   3296 pts/0    R+   12:29   0:00 | \_ \_ ps fux
mmeynard 20856   0.0   0.0  14596   3812 pts/2    Ss+  11:09   0:00 \_ bash
```

- RSS Resident Set Size mémoire physique utilisée en Kio
- VSZ Virtual memory SiZe (Kio)
- SI muLti-thread

Autres commandes

- top affiche la liste ordonnée des pds selon le pourcentage d'utilisation du processeur
- des gestionnaires de tâches existent pour chaque distribution qui permettent de visualiser les processus



| Tâche | PID | État | RSS | Processeur |
|--|-------|------|-----------|------------|
| session-wrapper.sh startxfce4 | 16720 | S | 3,2 Mio | 0% |
| sh /etc/xdg/xfce4/xinitrc -- /etc/X11/xinit/xserverrc | 16744 | S | 1,7 Mio | 0% |
| xfce4-session | 16851 | S | 15,3 Mio | 0% |
| Firefox | 16875 | S | 372,3 Mio | 1% |
| /usr/lib/firefox/firefox -contentproc -childID 5 -isForBrowser -prefsLen 1578... | 18010 | S | 187,8 Mio | 0% |
| /usr/lib/firefox/firefox -contentproc -childID 1 -isForBrowser -prefsLen 1125... | 17061 | S | 189,6 Mio | 0% |
| /usr/lib/firefox/firefox -contentproc -childID 4 -isForBrowser -prefsLen 1577... | 17297 | S | 210,8 Mio | 0% |
| /usr/lib/firefox/firefox -contentproc -childID 2 -isForBrowser -prefsLen 1125... | 17148 | R | 314,8 Mio | 2% |
| systemd/systemd --user | 16700 | S | 7,9 Mio | 0% |
| dbus-daemon --session --address=systemd: --nofork --nopidfile --systemd-activatio... | 16762 | S | 4,9 Mio | 0% |
| evince | 19989 | S | 5,0 Mio | 0% |

Plan

7 Gestion des processus

- Qu'est-ce qu'un processus
- Vie des processus
- Changement de contexte
- Scénario de vie de processus
- Observation des processus
- Génération de processus
- Recouvrement de processus

Génération de processus

Objectif : obtenir un nouveau processus à l'état *prêt*. Il faut :

- vérifier l'existence de l'exécutable,
- réserver un élément dans la table des processus,
- réserver l'espace nécessaire en mémoire,
- charger le code et données statiques dans les segments correspondants,
- initialiser les divers éléments des tables du système,
- mettre en place les fichiers ouverts par défaut,
- initialiser le contexte (compteur ordinal, pointeur pile en particulier).

Important : noter que c'est forcément un processus (le processus actif) qui demande cette création !

Sous Unix

Sous Unix, deux phases distinctes :

- mise en place d'un clone, par une copie de l'ensemble des segments du processus demandeur,
- mise en place de nouveaux segments de code et données statiques, réinitialisation de la pile et du tas, **si nécessaire**.

Le clone est réalisé par l'appel noyau *fork()* ; le remplacement des segments par *execve()* (cet appel est décliné en plusieurs variantes).

Exemple : on lance dans une fenêtre de l'interprète de langage de commande (le *shell*) **belotte**.

Pour le réaliser, l'interprète se duplique d'abord. Il y a donc un deuxième processus interprète et dans ce deuxième il y a appel à *execve()* afin de charger le code de **belotte** à la place du celui de l'interprète.

Principe de fonctionnement de *fork()*

- Créer une copie des segments de l'appelant ;
- chacun des deux processus aura donc le même code exécutable et continuera son exécution indépendamment de l'autre ;
- permettre au parent de reconnaître l'enfant créé parmi tous ceux qu'il a créés en lui restituant le numéro du nouvellement créé.

On peut noter que l'enfant aura un moyen de reconnaître son générateur ; en effet, si un parent peut avoir plusieurs enfants, un enfant ne peut avoir qu'un seul parent. Quoique, en cas de décès prématuré du générateur...

Exemple de *fork()* : forksimple.c

```
int main(){
    pid_t pid;
    switch(pid = fork()){
        case -1:{ // echec du fork
            printf("Probleme : echec du fork") ;
            break ;
        }
        case 0:{ // c'est le descendant
            printf("du descendant : valeur de retour de fork() : %d\n", pid);
            printf("du descendant : je suis %d de parent %d \n", getpid(),getppid()) ;
            break ;
        }
        default:{ // c'est le parent
            printf("du parent : valeur de retour de fork() : %d\n", pid);
            printf("du parent : je suis %d de parent %d \n",getpid(), getppid());
            break ;
        }
    }
    printf("Qui suis-je ? : %d\n",getpid());
}
```

Schéma algorithmique fork()

```
//résultat : dans parent : numéro de l'enfant ; dans enfant : 0
si (ressources système non disponibles) alors
    └ retourner erreur ; exit(0) ;
créer nouvel élément dans table processus ;
obtenir nouveau numéro processus ;
marquer état de ce processus en cours création ;
initialiser table processus[enfant] ;
copier segments de l'appelant dans l'espace mémoire du nouveau ;
incrémenter décompte fichiers ouverts ;
marquer état enfant prêt ;
si (processus en cours est le parent) alors
    └ retourner numéro enfant ;
sinon
    └ retourner 0 ;
```

Exécution

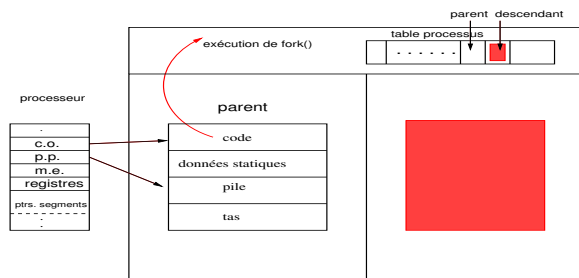
```
Exemple$ gcc -g -Wall -o forksimple forksimple.c
Exemple$ forksimple
du parent : valeur de retour de fork() : 27470
du parent : je suis 27469 de parent 77441
Qui suis-je ? : 27469
du descendant : valeur de retour de fork() : 0
du descendant : je suis 27470 de parent 1
Qui suis-je ? : 27470
```

Questions

- Qui est le pus 77441 ?
- Pourquoi, dans **cette** exécution, le descendant a comme parent le pus 1 (init) ?

Déroulement

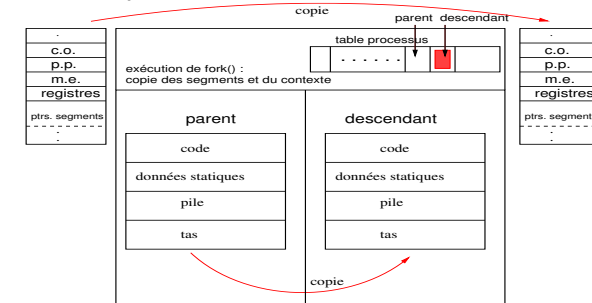
Le processus exécute ce code ; il y a appel noyau : *fork()*.



Il y a vérification de disponibilité des ressources : dans la table des processus, dans l'espace mémoire, etc, puis réservation d'espace pour l'enfant.

Suite et fin Déroulement

Cet appel fait une copie :



Avant la fin de *fork()* deux processus existent et tous les deux vont exécuter la fin de l'appel noyau, chacun dans son contexte. La pile de chacun contient un résultat différent de l'exécution de *fork()*. Donc chacun déroulera un cas différent de l'exécution du même code.

Questions et Exercices

- ❶ En reprenant l'exemple précédent, où vont avoir lieu les affichages respectifs ?
- ❷ Dans le schéma précédent représentant la suite et fin de déroulement, dans quelle partie de la mémoire sont localisées le contexte d'exécution de chaque processus ?
- ❸ Peut-on prévoir l'ordre dans lequel les deux processus vont s'exécuter ? Justifier. Autrement dit, dans l'exemple peut-on dire dans quel ordre s'afficheront les lignes écrites par les processus ?
- ❹ Donner deux exemples dans lesquels on aura un résultat négatif à *fork()*.

Plan

- 7 Gestion des processus
 - Qu'est-ce qu'un processus
 - Vie des processus
 - Changement de contexte
 - Scénario de vie de processus
 - Observation des processus
 - Génération de processus
 - Recouvrement de processus

Unix : Recouvrement de Processus

Le recouvrement consiste à demander dans un processus l'exécution d'un autre code exécutable que celui en cours d'exécution.

Principe :

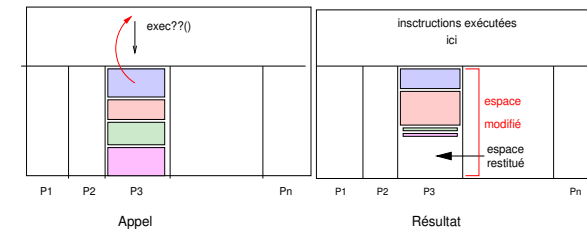
- vérifier l'existence et l'accessibilité (droits) du fichier exécutable ;
- écraser son propre segment de code par le nouvel exécutable ;
- passer éventuellement à ce code des paramètres d'exécution ;
- générer un nouveau segment de données statiques ;
- vider le segment de pile ;
- vider le tas ;
- faire quelques modifications dans la table des processus (espace mémoire alloué, compteur ordinal, etc).

Déroulement

Difficulté : se rendre compte qu'on fait de l'auto-destruction sans risque, car

- les instructions exécutées sont dans le noyau ; on ne risque pas de les écraser ;
- la partie écrasée est dans une autre partie de l'espace mémoire et les données ne sont pas utiles ;

C'est donc une copie disque → mémoire qui est faite, avec une réinitialisation ou rechargement des segments de données.



Recouvrement - Syntaxe

Il y a un et un seul appel noyau, `execve()`, décliné en plusieurs formes "confortables", faciles à appréhender.

Deux formes simples :

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

int execl(const char *path, const char *arg, ...)
int execlp(const char *file, const char *arg, ...)
```

Exemples

Exemple 1 : Le processus 123 exécute un programme x contenant l'instruction suivante :

```
int i=execl("/auto_home/jdupont/bin/monprog",
"monprog", "toto", "456", NULL);
```

- L'exécution du processus 123 va provoquer, si les ressources sont disponibles, le recouvrement par l'exécutable `monprog` de l'exécutable x ;
- `monprog` recevra les paramètres indiqués, `monprog` en position 0, `toto` en position 1, `456` en position 2

Exemples - suite

Exemple 2 : si on modifie la ligne d'appel de la façon suivante : `int i=execlp("monprog", "monprog", "toto", "456", NULL);`

- le fichier exécutable `monprog` va être recherché dans l'ensemble des répertoires cités dans la variable d'environnement `PATH`;
- si le fichier n'est pas trouvé, `execlp` rend un résultat négatif et le programme `x` continue;
- noter que seul un défaut de terminaison de `exec()` engendre un résultat retourné; autrement dit, pas de résultat positif à attendre.

Exercices

- 1 Trouver deux cas d'erreurs possibles.
- 2 Est-ce qu'un nombre de paramètres incohérent entre ceux passés et ceux attendus par l'exécutable est un cas d'erreur de `exec()` ?
- 3 Comment faire pour que ses propres programmes soient pris en compte, par `execlp()` (deux solutions) ?

Unix : Suite recouvrement

L'appel noyau :

```
int execve(const char *path, char *const argv[],
char *const envp[])
```

Rapprocher cette syntaxe de celle de `main()`. Il est possible de passer un environnement.

Autres formes :

Voir le manuel pour les détails, `int execlp()`, `int execlv()`, `int execlvp()`

Résumé :

- **l** liste explicite des paramètres,
- **v** liste des paramètres selon pointeur (`char *const argv[]`)
- **p** chemin exécutable selon `PATH`,
- **e** environnement à passer selon pointeur (`char *const envp[]`)

Plan

- 8 Système de Gestion des fichiers

Plan

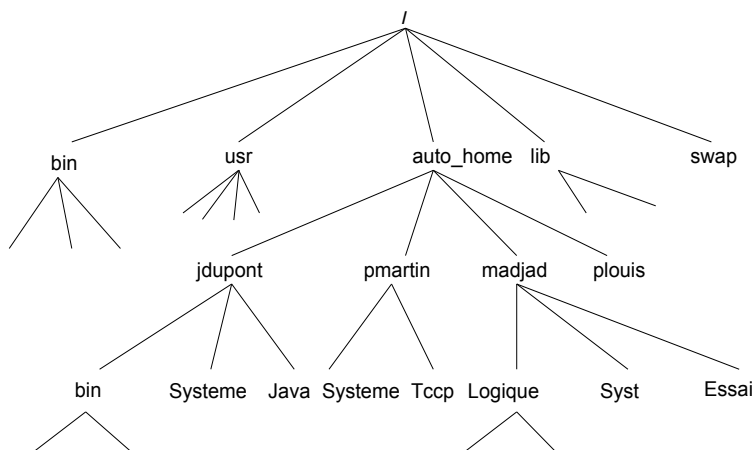
8 Système de Gestion des fichiers

- Introduction
- Systèmes de Fichiers
- Table des inodes
- Droits Unix
- Répertoires
- Liens durs
- Appels système du SGF
- Stockage des données des Fichiers
- Lien Symbolique
- Traitement des Fichiers Ouverts
- Cohérence des Partitions
- Retour sur les Droits

Objectifs

- Gérer l'espace disque, répondre aux demandes d'allocations et de libération d'espace ;
- Donner à l'utilisateur une vision **arborescente** simple ;
- Utiliser efficacement l'espace disque en place et en temps ;
- Assurer la sécurité des données grâce à un système de droits ;
- Gérer efficacement petits et gros fichiers ;
- Permettre l'utilisation de différents systèmes de fichiers (FAT32, NTFS, E4FS, NFS ...) greffés dans l'arborescence.

Vision utilisateur



L'utilisateur face au système

L'utilisateur veut des fichiers :

- accessibles grâce à un nom (au moins 1),
- organisés de sorte à les retrouver "facilement",
- sur lesquels il a le droit de propriété absolu et le droit de laisser faire certaines actions aux autres utilisateurs,
- copiables, déplaçables, renommables ...

Problèmes et réponses du système Unix

Le système doit gérer un certains nombres de problèmes :

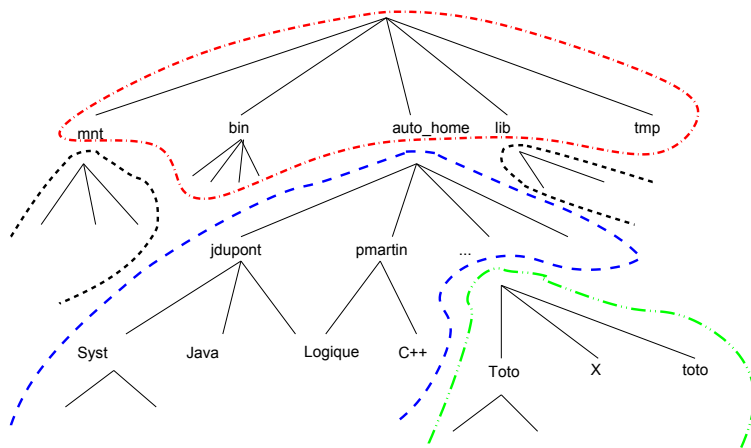
- concurrence des utilisateurs sur un même espace disque
- allocation désallocation des blocs et gestion du chaînage des blocs constituant un fichier
- minimisation de l'espace d'administration des fichiers (table des inode, table des blocs dispo., ...)
- solution générale des OS : une ou plusieurs arborescences de répertoires et de fichiers
- identification interne des fichiers par un numéro d'inode
- gestion des droits simple rwx
- gestion de systèmes de fichiers non Unix

Plan

8 Système de Gestion des fichiers

- Introduction
- Systèmes de Fichiers
- Table des inodes
- Droits Unix
- Répertoires
- Liens durs
- Appels système du SGF
- Stockage des données des Fichiers
- Lien Symbolique
- Traitement des Fichiers Ouverts
- Cohérence des Partitions
- Retour sur les Droits

Partitions et sous-arbres



Partition et Système de fichier

- un disque dur est généralement partitionné en plusieurs partitions
- au début du disque dur, résident le *Master Boot Record* et la table des partitions
- chaque partition est formatée selon un type de système de fichier
- différents *File System* existent : e4fs (Linux), NTFS (Windows), FAT32 (DOS Windows), HFS (Apple), ...
- chaque partition peut posséder un secteur de boot (secondaire) dans le cas où la partition est bootable
- l'unité d'allocation dans un système de fichier est le **bloc**

Structure d'un système de fichier Unix

Un système de fichier linux (e3fs ou e4fs) se compose de plusieurs parties :

| | |
|----------------|--------------------|
| gestion | Table des inodes |
| 3% | Table d'allocation |
| 97% données | Blocs de fichiers |

- Espace de gestion**
- éventuellement secteur de lancement (*bootstrap*).
 - une table des *inodes* (ou *i-nœuds*) : métadonnées des fichiers et localisation des blocs
 - une table d'allocation permettant de connaître les blocs libres

Espace des données contient les contenus des fichiers et quelques blocs d'indirection pointés par des inoeuds

Table des inodes

| num. | type | droits | liens | prop. | grp | taille | dates | pointeurs |
|------|------|--------|-------|-------|-----|--------|-------|-----------|
| | | | | | | | | |

num. numéro d'inode

type type de l'inode : fichier régulier (-), répertoire (d), ...

droits droits habituels rwx pour utilisateur, groupe, autres

liens compteur de liens durs ou nb de sous-répertoires

prop. identité (numéro) du propriétaire

grp identité du groupe

taille taille du fichier en nombre d'octets

pointeurs numéros des blocs dans l'espace des données

dates création, modification, accès

Plan

8 Système de Gestion des fichiers

- Introduction
- Systèmes de Fichiers
- **Table des inodes**
- Droits Unix
- Répertoires
- Liens durs
- Appels système du SGF
- Stockage des données des Fichiers
- Lien Symbolique
- Traitement des Fichiers Ouverts
- Cohérence des Partitions
- Retour sur les Droits

Exemple de représentation des Fichiers et Répertoires

| num. | type | droits | liens | prop. ; grp | taille | dates | pointeurs |
|------|------|-----------|-------|-------------|--------|-------|-----------|
| 1450 | - | rwXr-Xr-X | 1 | 470 ; 47001 | 125 | * | ... |
| 795 | d | rwXr-X-- | 2 | 470 ; 47001 | 2048 | * | ... |
| 2 | d | rwXr-Xr-X | 2 | 0 ; 0 | 2048 | * | ... |

Répertoires

| /home | | / (racine) | |
|-------|---------|------------|------|
| num. | nom | num. | nom |
| 2 | •• | 2 | •• |
| 795 | • | 795 | home |
| 1450 | hello.c | 7654 | usr |

Plan

8 Système de Gestion des fichiers

- Introduction
- Systèmes de Fichiers
- Table des inodes
- Droits Unix
- Répertoires
- Liens durs
- Appels système du SGF
- Stockage des données des Fichiers
- Lien Symbolique
- Traitement des Fichiers Ouverts
- Cohérence des Partitions
- Retour sur les Droits

Un système simple de droits

- Axiome : tout accès à un fichier doit être réalisé après ouverture (*open*) d'un **chemin** (path) absolu ou relatif
- l'ouverture vérifie l'accessibilité de l'utilisateur du processus à chaque répertoire traversé et au fichier final
- 3 types de droits : r(ead), w(rite), x(eXecute)
- 3 catégories d'utilisateur : u(owner), g(roup), o(ther)
- tout répertoire étant un fichier, les droits signifient :
 - r lecture du contenu du répertoire (ls)
 - w écriture dans le répertoire c-à-d création (*creat*), suppression (*rm*), renommage (*mv*) de fichier
 - x traversée du répertoire : un fichier lisible par tous, situé dans un répertoire non traversable, ne pourra être lu
- rappel : à tout processus est associé un répertoire courant (*getcwd()*), un utilisateur (*getuid()*) et un groupe (*getgid()*)

Plan

8 Système de Gestion des fichiers

- Introduction
- Systèmes de Fichiers
- Table des inodes
- Droits Unix
- Répertoires
- Liens durs
- Appels système du SGF
- Stockage des données des Fichiers
- Lien Symbolique
- Traitement des Fichiers Ouverts
- Cohérence des Partitions
- Retour sur les Droits

Structure d'un Répertoire

- Un répertoire contient une liste de couples (entrées) $n^o \text{ inode} \Leftrightarrow \text{nom}$ où chaque nom est unique
- la **longueur des noms des fichiers** est variable, avec une limite administrable, fixée par le système, par exemple 256 octets
- les éléments autres que le *numéro d'inode* et le *nom*, inclus dans un répertoire, permettent la **gestion de la liste**, par exemple la longueur de l'élément courant
- tout répertoire contient au moins deux entrées **•** et **••** afin de permettre la navigation relative au répertoire courant
- la taille d'un répertoire est gérée autrement que celle d'un fichier (multiple de la taille d'un bloc)
- les opérations sur les répertoires sont soit des appels systèmes (*mkdir()*, *rmdir()*) soit des fonctions de bibliothèque (*opendir()*, *readdir()*, *closedir()*)
- on ne peut admettre qu'un utilisateur puisse modifier directement (*open*, *write*) un répertoire ; le SF pourrait être corrompu.

Plan

8 Système de Gestion des fichiers

- Introduction
- Systèmes de Fichiers
- Table des inodes
- Droits Unix
- Répertoires
- **Liens durs**
- Appels système du SGF
- Stockage des données des Fichiers
- Lien Symbolique
- Traitement des Fichiers Ouverts
- Cohérence des Partitions
- Retour sur les Droits

Exemple de liens durs (*hard link*)

| num. | type | droits | liens | prop. | taille | dates | pointeurs |
|------|------|------------------------|----------|------------|--------|-------|-----------|
| 1450 | - | <code>rwXr-xr-x</code> | 2 | 470; 47001 | 125 | ... | 8003475 |

Répertoires

| tp5 | | ProjetX | |
|------|----------|---------|----------|
| num. | nom | num. | nom |
| 1450 | commun.h | 1450 | commun.h |

On mémorise dans l'inode le **nombre** de liens !

Notion de lien dur (*hard link*)

- chaque entrée dans un répertoire référence un élément de la table des inodes, c'est à dire un fichier
- plusieurs entrées du même répertoire ou dans différents répertoires peuvent référencer le même inode : `ln hello.c hello2.c`
- les deux noms de fichiers sont des liens durs équivalents qui pointent sur le même contenu et les mêmes métadonnées (inode)
- l'accès en écriture sur cet unique fichier peut être réalisé via l'un ou l'autre
- l'usage historique des liens était le partage de fichiers communs à un groupe de développeurs :
`ln commun.h ~jdupont/commun.h`

Lien Dur

Un *lien dur* ou *physique* est différent d'un *lien symbolique* ou *raccourci* qu'on étudiera plus tard.

Attention : on peut dire que toute référence à un inode représentant un fichier est un *lien dur* ! En effet, une fois l'opération effectuée, on ne peut établir un ordre de précedence parmi les références.

Caractéristiques :

- Un lien dur est forcément dans une **même partition** (SF) ; on dit qu'un lien dur ne peut *traverser* les SFs.
- On ne peut créer un lien dur sur un **répertoire** : cela supprimerait la structure arborescente (graphe avec circuits)
- Noter qu'il y a un seul inode, donc un seul propriétaire, un seul contenu, un seul ensemble de droits, ... **MAIS** des utilisateurs différents devant appartenir au groupe du fichier afin de pouvoir lire/modifier le contenu

Problèmes Induits par les Liens Durs

Une correction à faire : La suppression d'une référence (nom de fichier, lien dur) ne supprimera pas forcément l'inode et tout le contenu du fichier.

Principe de la solution :

- à chaque création d'un lien incrémenter le nombre de liens dans la table des inodes
- le décrémenter à chaque suppression
- ne supprimer l'inode et le contenu (désallocation des blocs) que lorsque le nombre de liens est nul

Remarque : la commande de suppression de fichier (`rm`) est implémentée par l'appel système `unlink()` qui supprime une entrée de répertoire et peut avoir un effet de bord

Exemple de Suppression

| num. | type | droits | liens | prop. | taille | dates | pointeurs |
|------|------|------------------------|-------|-------------|--------|-------|-----------|
| 1450 | - | <code>rwXr-xr-x</code> | 2 | 470 ; 47001 | 125 | * | 8003475 |

Répertoires

| tp5 | | replique | |
|------|----------|----------|----------|
| num. | nom | num. | nom |
| 1450 | commun.h | 1450 | commun.h |

Algorithme de Suppression

Algorithme 2 : supprimer(fichier)

si (*droits de traversée jusqu'au répertoire contenant acquis et droits d'écriture dans répertoire contenant*) **alors**

 nombreDeLiens - - ;

 effacer entrée dans répertoire ;

si (*nombreDeLiens == 0*) **alors**

 restituer espace désigné par pointeurs ;

 effacer entrée dans table-inodes ;

sinon

 afficher suppression impossible

Avantages et Manques

Avantages

- une seule version du fichier, donc mises à jour cohérentes
- permet d'assurer la compatibilité entre emplacements différents prévus pour un fichier, par exemple entre versions d'un système d'exploitation (fichiers dans `/bin`, `/usr/bin`, ...)

Manques

- la limite au SF est très réductrice
- pas de référence à un répertoire
- les éditeurs de texte tels `emacs` enregistrent en renommant l'ancienne version en `commun.h~` et créent un nouveau fichier (inode) `commun.h`

Remarque : la donnée *liens* dans la table des inodes est utilisée et a un sens différent pour les répertoires.

Plan

8 Système de Gestion des fichiers

- Introduction
- Systèmes de Fichiers
- Table des inodes
- Droits Unix
- Répertoires
- Liens durs
- Appels système du SGF
- Stockage des données des Fichiers
- Lien Symbolique
- Traitement des Fichiers Ouverts
- Cohérence des Partitions
- Retour sur les Droits

Appels Système du SGF

On établit une liste non exhaustive des appels systèmes permettant d'accéder ou manipuler soit des *fichiers*, soit des *répertoires*, soit des *inodes*.

Accès à des fichiers

| | |
|--------------------------|---|
| open(), close(), creat() | ouv., ferm. création |
| read(), write() | lect., écriture |
| lseek() | déplacement de la tête |
| unlink() | supprimer un lien dur (entrée de rép.) |
| link() | créer un lien dur (entrée de rép.) |
| rename() | renommer et déplacer un lien à l'intérieur du même SF (répertoires aussi) |

Exemples : la commande `rm` fait appel à `unlink()`, la commande `ln`, peut faire appel à `link()`, selon les options passées.

Appels Système pour Répertoires

Noter que les commandes Unix utilisent forcément ces appels.

| | |
|-----------------------|--|
| rename() | renommer, déplacer |
| mkdir(), rmdir() | création, suppression |
| opendir(), closedir() | ouverture, fermeture |
| readdir() | lecture d'une entrée (n^o inode, nom) |
| chdir() | changer le répertoire courant du pus |

Remarque : pas de `writedir()` : pourquoi ?

Appels Système pour Inode

Noter que les commandes Unix utilisent forcément ces appels.

| | |
|----------------------------|-------------------------------|
| stat(), lstat(), fstat() | accès à un inode |
| chmod(), fchmod() | modification droits |
| access() | vérifier droits fichier |
| touch(), utime(), utimes() | accès et modification dates |
| chown(), chgrp() | modifier propriétaire, groupe |

Accès à l'Inode

À partir du nom ou d'un descripteur sur un fichier ouvert, on peut accéder à l'inode, mais pas aux adresses de blocs (pointeurs dans la table des inodes)... Syntaxe :

NAME

stat, fstat, lstat - get file status

SYNOPSIS

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

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

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

```
int stat(const char *file_name, struct stat *buf);
```

```
int fstat(int filedes, struct stat *buf);
```

```
int lstat(const char *file_name, struct stat *buf);
```

Structure Récupérée

```
struct stat{
    dev_t st_dev; /* device */
    ino_t st_ino; /* inode */
    umode_t st_mode; /* protection */
    nlink_t st_nlink; /* number of hard links */
    uid_t st_uid; /* user ID of owner */
    gid_t st_gid; /* group ID of owner */
    dev_t st_rdev; /* device type (if inode device) */
    off_t st_size; /* total size, in bytes */
    unsigned long st_blksize; /* blocksz for filesystem I/O */
    unsigned long st_blocks; /* number of blocks allocated */
    time_t st_atime; /* time of last access */
    time_t st_mtime; /* time of last modification */
    time_t st_ctime; /* time of last change */
}
```

Exemple d'Accès à l'Inode

On constate qu'on peut récupérer toutes les informations présentées précédemment dans la table des inodes.

Reste à connaître les quelques masques nécessaires pour extraire une information consistant en une suite de bits de longueur différente de 32 ou 8.

```
struct stat etat;
int i = stat ("toto.txt", &etat);
if (i == 0){
    printf("le fichier toto.txt a pour inode : %d ", \
        etat.st_ino);
    if (S_ISREG(etat.st_mode))
        printf(" et c'est un fichier régulier\n");
}
```

Plan

- 8 Système de Gestion des fichiers
 - Introduction
 - Systèmes de Fichiers
 - Table des inodes
 - Droits Unix
 - Répertoires
 - Liens durs
 - Appels système du SGF
 - **Stockage des données des Fichiers**
 - Lien Symbolique
 - Traitement des Fichiers Ouverts
 - Cohérence des Partitions
 - Retour sur les Droits

Le Problème de la Taille

Constat : dans la table des inodes, les pointeurs sur les blocs de données sont les adresses de ces blocs. Il faut gérer avec ces pointeurs des fichiers de taille extrêmement variable (euphémisme).

Problème : Comment gérer cette taille avec un nombre fixe de pointeurs dans la table des inodes ?

Pourquoi un nombre fixe ? parce que les systèmes d'exploitation les adorent et cherchent aussi une solution efficace permettant de minimiser le nombre d'accès disque. Par exemple, le nombre de pointeurs est fixe et limité à 13 jusqu'à 16 pointeurs selon la version d'Unix. Et pourtant, il va bien falloir gérer de très gros fichiers comme des tout petits.

Blocs Directs et Indirects

Principe : la gestion de la taille se fera avec quelques adresses pointant **directement** sur les blocs de données. Ensuite, dès que les fichiers grossissent, on construit des blocs dits indirects, dont le contenu est lui-même un ensemble d'adresses, qui permettent donc d'étendre les adresses directes.

Méthode :

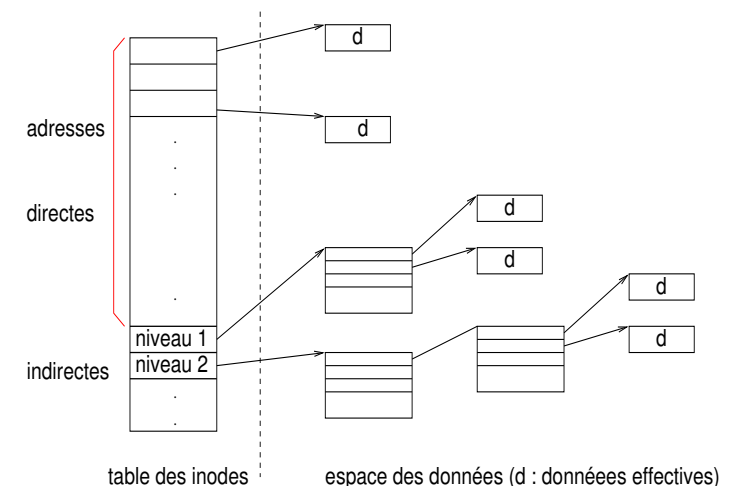
Adressage de tout fichier : un arbre dont les feuilles sont les blocs de données et les nœuds internes des blocs d'adresses.

Chaque niveau de l'arbre autre que les feuilles est une liste des adresses des enfants.

Méthode en Détails

- pointeurs directs : contiennent l'adresse de blocs de données.
- pointeurs indirects : contiennent l'adresse de blocs contenant des adresses d'autres blocs, de données ou d'adresses, selon de niveau d'indirection.
 - indirects à n niveaux : contiennent les adresses de blocs, eux-mêmes contenant des adresses de niveau $n - 1$. Une adresse de niveau 0 est une adresse de bloc de donnée.
- le premier niveau de l'arbre est dans la table des inodes ; il est de taille fixe : n_0 pointeurs directs, n_1 pointeurs de niveau 1, n_2 de niveau 2, etc.
- les divers unix : $10 \leq n_0 \leq 13$, $n_1 = 1$, $n_2 = 1$, $n_3 = 1$

Arbre d'Allocation



Exemple - Taille Maximale d'un Fichier

On prend une table d'inodes classique avec 10 pointeurs directs et un seul pointeur pour chacun des 3 niveaux suivants.

Données de base : on suppose que

- la taille des blocs de données est $4Ki$ octets,
- les adresses sont codées sur 32 bits (4 octets).

- Taille maximale atteinte par les blocs directs :
 $10 \text{ blocs} \times 4Ki \text{ octets} = 40Ki \text{ octets}$
- Pour les blocs indirects, il faut d'abord calculer le nombre d'adresse contenues dans un bloc de données, c'est-à-dire *taille bloc / taille adresse*,
ici $4Ki \text{ octets} / 4 \text{ octets} = 1Ki \text{ adresses}$.

Taille Maximale d'un Fichier - suite

- Taille maximale atteinte par le 1^{er} niveau :
 $1 \text{ pointeur} \times 1Ki \text{ adresses} \times 4Ki \text{ octets} = 4Ki^2 \text{ octets} = 4Mi \text{ octets}$.
- Taille maximale atteinte par le 2^{ème} niveau :
 $1 \text{ ptr} \times \underbrace{1Ki \times 1Ki}_{1M \text{ adr}} \text{ adr} \times 4Ki \text{ oct} = 4Ki^3 \text{ oct} = 4Gi \text{ oct}$.
- Taille maximale atteinte par le 3^{ème} niveau :
 $1 \text{ ptr} \times \underbrace{1K \times 1Ki \times 1Ki}_{1Gi \text{ adr}} \text{ adr} \times 4Ki \text{ oct} = 4Ki^4 \text{ oct} = 4Ti \text{ oct}$.

La taille maximale d'un fichier possible par l'**adressage des blocs** est de : $40Kio + 4Mio + 4Gio + 4Tio$.

On peut donc approximer cette taille à $4Tio$ ce qui représente 1/4 des adresses de blocs disponibles avec 32 bits d'adresse !

Exercices

- Si la taille des blocs de données est doublée (à $8Kio$) calculer le facteur de multiplication de la taille maximale d'un fichier (approximation) : $\times 16$ donc $64Tio$
- Avec des blocs de $4Kio$, sur quelle longueur faudrait-il coder la taille du fichier dans la table des inodes, afin de satisfaire la taille atteinte par l'adressage des blocs ? 42 bits
- On considère que les blocs non terminaux (ceux contenant des adresses de blocs) sont "volés" à l'espace des données. Combien de blocs sont ainsi subtilisés avec les blocs de $4Kio$ et un fichier de taille $4Gio$? $4 \times 2^{30} / 4 \times 2^{10} = 2^{20} \text{ adrs}$ de 4 octets, soit $2^{22} \text{ oct} = 4Mo$, soit 2^{10} blocs .

Plus difficile :

- Situation de départ : blocs de $4Kio$, taille du fichier codée sur 32 bits. On veut agrandir la capacité maximale d'un fichier et passer à $32Gio$. Que peut-on proposer ? taille sur 35 bits donc 40 bits = 5 oct

En résumé : des calculs à optimiser

Lors du **formatage du SF**, il faut prendre en compte :

- le codage de la taille du fichier dans la table des inodes
- l'espace physique réel disponible sur la partition disque
- la taille de chaque adresse de bloc
- la taille de chaque bloc
- la taille de la table des inodes (nb de fichiers)

Exemple :

- lorsque la taille du fichier dans la table des inodes est codée sur 32 bits, la taille maximale d'un fichier est de 2^{32} octets, soit $4Gio$;
- si l'espace des données de la partition est supérieur à $4Gio$, alors $4Gio$ reste la taille maximale réelle ;
- dans ce cas, avec les blocs de $4Kio$ de l'exemple, le 3^{ème} niveau d'indirection est non utilisé.

Plan

8 Système de Gestion des fichiers

- Introduction
- Systèmes de Fichiers
- Table des inodes
- Droits Unix
- Répertoires
- Liens durs
- Appels système du SGF
- Stockage des données des Fichiers
- **Lien Symbolique**
- Traitement des Fichiers Ouverts
- Cohérence des Partitions
- Retour sur les Droits

Lien Symbolique ou raccourci

Objectif : dépasser les limites des liens durs :

- limitation à une même partition (SF)
- impossibilité de pointer sur un répertoire
- basé sur numéro de inode et pas sur un chemin symbolique

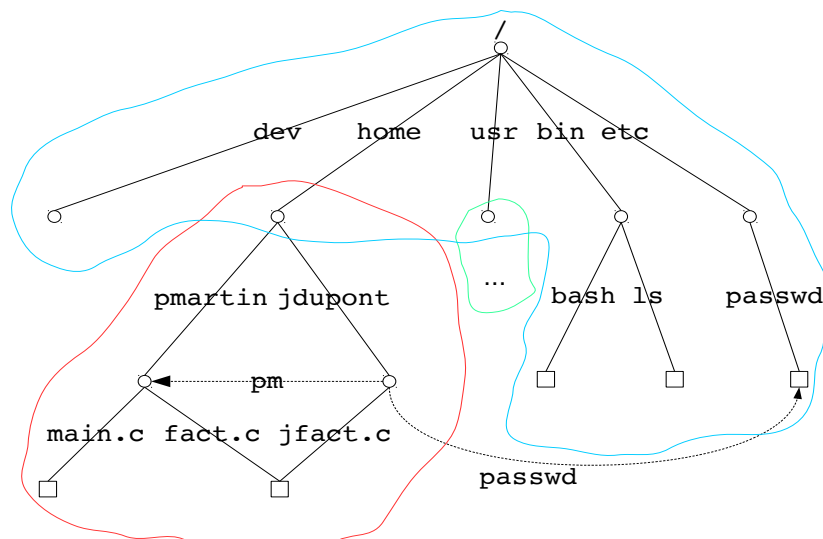
Méthode : un fichier de type nouveau (l) permettant de désigner (référer, pointer sur) un *chemin* déjà existant dans l'arborescence

Exemples :

- `ln -s /etc/passwd ./passwd`
- `ln -s /home/pmartin/ pm`
- `ln -s /home/pmartin/main.c ./monmain.c`

Point de vue du système : toute action demandée sur le raccourci agira sur le fichier lié, sauf quelques exceptions (lstat)

Exemples de Lien Symbolique



Inode lien symbolique

Table des inodes

| num. | type | droits | liens | prop. | taille | dates | pointeurs |
|------|------|-----------|-------|-------|--------|-------|-----------|
| 3232 | l | rwxr-xr-x | | | 11 | | 99999 |

Attention :

- noter le type `l` : ni fichier régulier (`-`), ni répertoire (`d`), mais lien symbolique (`l`)
- noter la taille `11` : exactement celle de la chaîne de caractères `/etc/passwd` (contenu dans le bloc `99999`)

Caractéristiques des Liens Symboliques

Les liens symboliques peuvent se faire sur des répertoires, dans une même partition ou dans des partitions différentes.

Problèmes

- un lien symbolique vers un répertoire ascendant crée un circuit qu'il ne faut pas prendre dans un parcours récursif (`find`) : `ln -s .. c`
- circuit : `touch a; ln -s a b; rm a; ln -s b a; cd a` provoque une erreur après un certain nombre de résolution de raccourcis !
- lien mort : un raccourci peut référencer un chemin qui n'existe plus ; une vérification d'existence est parfois réalisée à la création (linux)

Commandes et fonctions concernant les Liens Symboliques

- Commande : `ln -s [ancien] [raccourci]`
- Appel système : `symlink()`.
- **lien dur** commande : `ln [ancien] [nouveau]`, appel système : `link()`
- `stat("raccourci", ...)` accèdera au fichier référencé tandis que `lstat` accèdera au raccourci
- `open("raccourci", ...)` ouvrira le fichier référencé donc `cat raccourci` ou `emacs raccourci` aussi
- `rm raccourci` supprimera le raccourci,
- `cd raccourci` concatènera le mot `"/raccourci"` au répertoire courant, ce qui peut être bizarre : `ln -s .. c; cd c; pwd; cd .. a` comme effet de revenir dans le répertoire de départ !
- les droits et le propriétaire sont ceux relatifs au raccourci (`chmod`, `chown`, `chgrp`)

Plan

8 Système de Gestion des fichiers

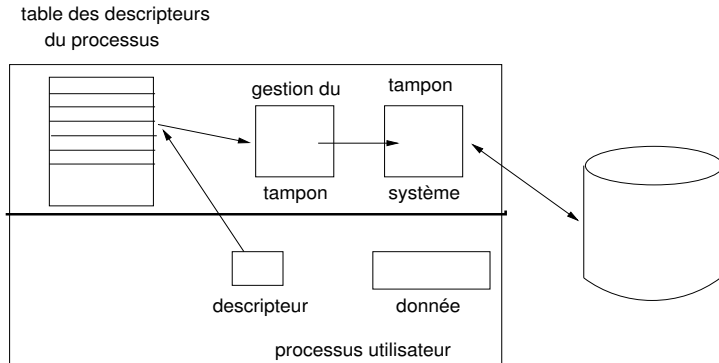
- Introduction
- Systèmes de Fichiers
- Table des inodes
- Droits Unix
- Répertoires
- Liens durs
- Appels système du SGF
- Stockage des données des Fichiers
- Lien Symbolique
- **Traitement des Fichiers Ouverts**
- Cohérence des Partitions
- Retour sur les Droits

Problèmes divers

Problèmes traités dans ce chapitre :

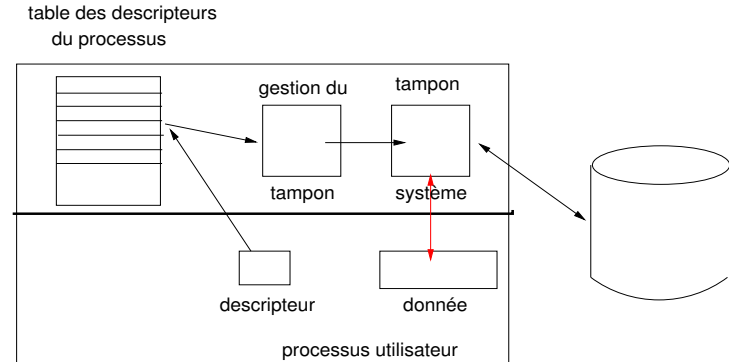
- Comment est-ce que le système gère les fichiers ouverts par les processus ? Que se passe-t-il lorsque plusieurs processus partagent le même fichier ?
- Pourquoi est-il nécessaire de vérifier la cohérence entre le contenu des répertoires et la table des inodes ? Que peut-on vérifier ? Peut-on réparer ?
- Comment est réalisée l'association des systèmes de fichiers simple en un système de fichiers global ?
- Des questions sont restées sans réponse sur les droits des fichiers et répertoires.

Ouverture d'un Fichier



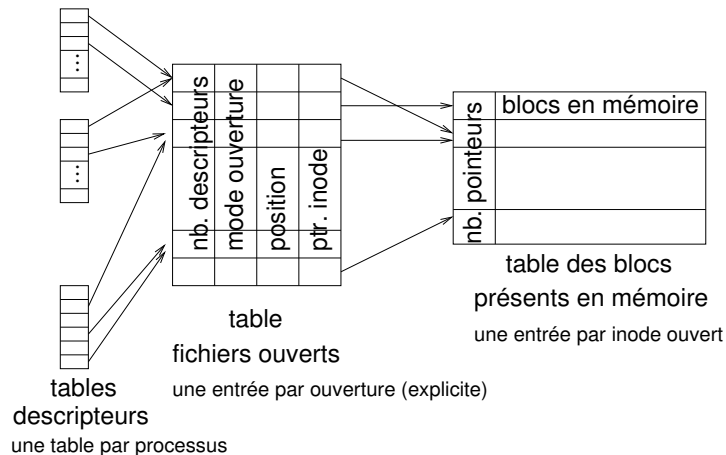
- Le descripteur est un indice vers un élément de la table des descripteurs de ce processus. Il y a une table par processus.
- Le tampon système contient au moins un bloc du fichier ouvert.
- Noter que la gestion de ce bloc dépend du type de périphérique.

Accès à une donnée



- La demande de l'utilisateur provoque un transfert tampon système \Leftrightarrow espace utilisateur. Elle ne provoque pas toujours un transfert disque \Leftrightarrow tampon système.

Table des Fichiers Ouverts du Système



Le système gère toutes les demandes d'ouverture de fichiers par la *table des fichiers ouverts du système*.

Questions Soulevées

Plusieurs questions se posent à la vue de cette table des fichiers ouverts et les tables associées :

- Quel rapport entre la table des blocs présents en mémoire (i.e. des inodes ouverts) et le tampon système vu précédemment ?
- À quelle situation correspond le fait d'avoir des descripteurs de processus différents pointant sur la même entrée dans cette table ?
- Dans quelles conditions peut-on avoir pour un même processus deux descripteurs pointant vers la même entrée de cette table ?
- À quelle situation correspond le fait d'avoir plusieurs pointeurs de la table des fichiers ouverts vers la même entrée dans la table des inodes ?

Table des Blocs Présents en Mémoire

La *table des blocs présents en mémoire* (appelée aussi -malheureusement¹ - *table des inodes en mémoire*) représente l'ensemble des tampons du système pour l'ensemble des fichiers ouverts par tous les processus.

On constate que pour des raisons d'efficacité, le système va ramener en mémoire centrale **quelques** blocs de chaque fichier ouvert, en fonction de **prévisions** qu'il peut faire, de sorte à gagner du temps sur les entrées-sorties.

Il y a donc un **asynchronisme** entre les demandes des utilisateurs et leurs réalisation réelle : les données à écrire seront conservées en mémoire, dans la tables des inodes, jusqu'au moment jugé opportun par le système pour les transporter sur le périphérique ; des prévisions sur les lectures permettront de les devancer.

1. ce n'est pas une copie en mémoire de la tables des inodes !

Ouvertures Multiples de Fichiers

Lorsque deux processus ouvrent le **même fichier** (ils demandent explicitement un *open()*) ils auront chacun son propre pointeur sur la table des fichiers ouverts.

Ceci est vrai que cette demande d'ouverture soit en lecture, écriture ou les deux. Seuls comptent leurs droits de réaliser ces opérations.

Dans ce cas, chaque processus aura dans sa table des descripteurs un pointeur vers une entrée différente dans la table des fichiers ouverts. Ces deux éléments de la table des fichiers ouverts pointeront vers la même entrée de la table des inodes en mémoire.

Noter l'expression *une entrée par ouverture explicite* dans le schéma.

Contenus des Tables

nb. descripteurs nombre d'éléments pointant sur la même entrée de la table des fichiers ouverts (\neq nombre de processus)

mode d'ouverture classique : lecture, écriture, lecture et écriture

position position courante atteinte (voir *lseek()*) ; par exemple, pour un fichier ouvert en lecture, après lecture de 3 caractères, la position courante est 4. Noter que c'est la comparaison entre cette position et la taille du fichier qui permet de savoir si la fin de fichier est atteinte.

ptr. inode un pointeur sur la table des inodes en mémoire.

nb. pointeurs nombre d'entrées de la tables des fichiers pointant sur le même élément ; permet de savoir si on peut fermer effectivement un fichier, c'est-à-dire vider sur le périphérique s'il y a eu écriture et libérer l'espace.

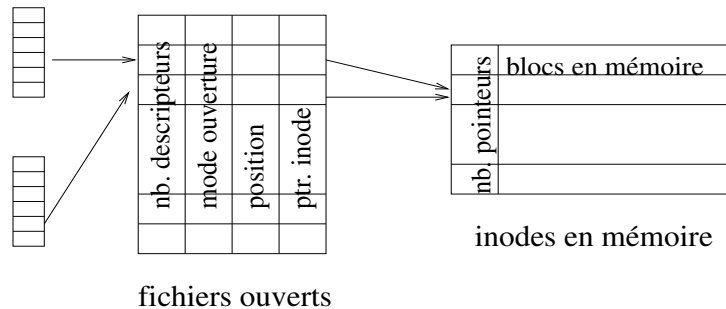
Résumé

Si un processus a fait *open()* il a forcément obtenu une nouvelle entrée dans la table des fichiers ouverts, pointant soit vers un nouvel élément dans la table des inodes en mémoire, soit vers un élément déjà existant dans cette dernière.

On peut partager une même entrée dans la table des fichiers ouverts soit par héritage entre processus, soit en demandant un nouveau descripteur sur un fichier précédemment ouvert dans le même processus.

Ouvertures Multiples de Fichiers - Exemple

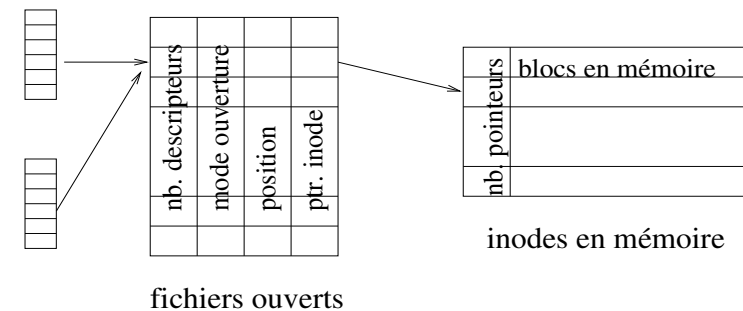
Deux processus ouvrant le même fichier, chacun avec ses propres paramètres, se représentent ainsi :



- Noter que les ouvertures peuvent être conflictuelles : les deux processus en écriture sur la même donnée du fichier, l'un en lecture et l'autre en écriture sur la même donnée, ...

Accès Multiples à des Fichiers - Exemple

Un processus hérite de l'ouverture faite par un autre :



- Noter que les opérations peuvent être conflictuelles, avec les mêmes remarques que précédemment.
- **Attention** : la même entrée dans la tables des fichiers ouverts est accessible aux deux processus.

Remarques, Questions

- Un même processus peut obtenir plusieurs descripteurs sur une même entrée de la table des fichiers ouverts, **pas** en faisant deux ouvertures, mais en utilisant *dup()*, *dup2()*
- Que se passe-t-il lorsqu'un processus ouvre le même fichier en faisant deux demandes *open()* ?
- Dans le cas d'ouverture multiple en écriture, par deux processus, quelle sera la version enregistrée sur disque ?
- Un fichier contient la chaîne de caractères *abcdefghij*. Il est ouvert par un processus *P1* qui crée un clone *P2*. *P1* veut lire 4 caractères puis 2 autres dans une lecture suivante. *P2* veut lire 1 caractère puis 2. Qu'obtiennent-ils comme résultats des lectures ?

Plan

8 Système de Gestion des fichiers

- Introduction
- Systèmes de Fichiers
- Table des inodes
- Droits Unix
- Répertoires
- Liens durs
- Appels système du SGF
- Stockage des données des Fichiers
- Lien Symbolique
- Traitement des Fichiers Ouverts
- Cohérence des Partitions
- Retour sur les Droits

Besoin de Contrôler la Cohérence

Un dysfonctionnement grave peut avoir lieu si le système de gestion des fichiers est corrompu. Exemples de défauts :

- un bloc se trouve à la fois libre (dans la liste des blocs libres) et occupé, (pointé par la table des inodes),
- la taille des fichiers n'est pas cohérente avec le nombre de blocs occupés,
- le nombre de liens durs dans la table des inodes n'est pas cohérent avec le nombre de pointeurs,
- deux fichiers occupent le même bloc de données. . .

Certains défauts sont faciles à rectifier. D'autres sont difficiles ou impossibles à corriger. Enfin, certaines corrections faciles peuvent entraîner la perte d'un ou plusieurs fichiers.

Exercices

- 1 Proposer une correction lorsque le nombre de liens pour un inode i dans la table des inodes est supérieur (resp. inférieur) au nombre f de fichiers trouvés référençant i .
- 2 Montrer que la situation où deux fichiers occupent un même bloc de données est forcément incohérente ; Étudier les solutions possibles et proposer la correction qui vous semble la mieux adaptée.
- 3 Donner un exemple où la correction que vous venez de suggérer à la question 2 est mauvaise ou inadaptée.

Une Perte de Temps à s'Offrir

Constat : Certaines vérifications ne peuvent se faire qu'en parcourant toute l'arborescence. Donc c'est forcément long. Et **indispensable** à faire.

Quand ? Le moins souvent *possible*... À chaque démarrage du système ? Lorsque le système a été arrêté improprement ? Plus le volume des disques augmente, plus le volume des partitions grandit, moins on en a envie. Il n'y a pas une bonne solution.

Comment ? Il faut balayer toutes les partitions (tous les sous-arbres) au moins une fois. Donc

- il faut avoir des algorithmes efficaces, qui évitent le plus possible de refaire des parcours,
- il faut faire des exécutions parallèles permettant de vérifier plusieurs partitions à la fois.

Attention : Un parcours séquentiel de la table des inodes est utile, mais ce type d'accès n'est pas offert en tant qu'appel système.

Plan

8 Système de Gestion des fichiers

- Introduction
- Systèmes de Fichiers
- Table des inodes
- Droits Unix
- Répertoires
- Liens durs
- Appels système du SGF
- Stockage des données des Fichiers
- Lien Symbolique
- Traitement des Fichiers Ouverts
- Cohérence des Partitions
- Retour sur les Droits

Droits supplémentaires

Question : Peut-on améliorer les droits de base, soit pour interdire l'effacement de fichiers non propriétaires, soit pour accéder à des fichiers ou répertoires dans des conditions restreintes.

Idées générales : Ajouter quelques informations et associer des droits différents aux processus selon que l'on regarde les droits d'accès aux fichiers ou l'aspect propriété du processus.

| 4 bits | 1 bit | 1 bit | 1 bit | 9 bits |
|---------|---------|---------|---------------|-------------------|
| type f. | set_uid | set_gid | <i>sticky</i> | droits classiques |

Noter que pour le type on connaît maintenant plus de deux types...

Plan

9 Communications basiques entre Processus (signaux et tubes)

Développement

L'élément noté *sticky* sert à empêcher la destruction de fichiers dont on n'est pas propriétaire. Voir typiquement */tmp*. Ce droit est représenté par la lettre *t* et on le positionne par `chmod +t [nomRépertoire]` :

```
drwxrwxrwt 9 root root 8192 ..... /tmp
```

Hors cadre de ce cours

Le bit `set_uid` permet de prendre temporairement l'identité du propriétaire du fichier exécutable. Temporairement se réduit uniquement à la durée d'exécution.

`set_gid` agit de façon identique, mais sur le groupe du propriétaire de l'exécutable.

Plan

9 Communications basiques entre Processus (signaux et tubes)

- Les Besoins
 - Signaux Unix
 - Tubes Simples
 - Tubes Nommés

Communication entre processus

Jusque là, chaque processus vivait de façon **isolée**, disposant seul de son espace mémoire. Un besoin de communication avec les autres processus devient pressant.

La communication entre processus a plusieurs objectifs :

- avertir un processus lorsque des événements particuliers surviennent ;
- **synchronisation** des processus pour éviter qu'ils accèdent concurremment à des ressources **critiques** (imprimante, fichier, ...);
- échange ou partage de données communes afin de réaliser des calculs parallèles.

Ces échanges et partages sont utiles tant pour les processus systèmes (démons) qu'utilisateurs.

Les Solutions Unix

signaux Unix, sorte d'interruption logicielle ;

tubes FIFO synchronisée permettant la gestion sans perte ni duplication de caractères ; 2 possibilité : tube simple ou nommé ;

IPC Inter Processes Communication est un acronyme qui désigne des mécanismes sophistiqués, non étudiés dans ce cours, tels que :

sémaphore le plus célèbre des mécanismes de synchronisation ;

files de messages permettant aux pds d'une même machine de s'échanger des messages ;

mémoire partagée permettant à des processus d'accéder à un segment commun de mémoire ;

processus légers ou *thread* ou fils d'exécution qui partagent des segments mémoire d'un même processus (lourd) ;

Exemples de synchronisation et de communication

- Système de réservation ou d'allocation de places (spectacle, transport, salle, ...);
- Spooler d'imprimante ;
- processus parent prenant connaissance de la fin d'un enfant (wait) ; synchronisation **simple** !
- communication **complexe** qui calcule le nombre de sources C dans l'arborescence issue du répertoire parent :

```
ls -R . | grep "\.c$" | wc -w
```

 - lancement de 3 processus (ls, grep, wc) ;
 - création de 2 tubes leur permettant de relier leur sortie standard à l'entrée standard du suivant ;

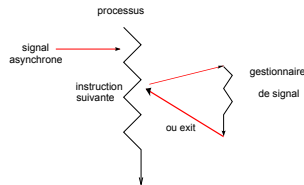
Plan

9 Communications basiques entre Processus (signaux et tubes)

- Les Besoins
- Signaux Unix
- Tubes Simples
- Tubes Nommés

Signaux : définition et objectif

- sorte d'*interruption* logicielle ayant un comportement **asynchrone** : le processus ne sait ni s'il recevra ni quand il recevra un signal ;



- moyen pour informer un processus d'un événement urgent ;
- tous les signaux sont répertoriés dans une liste complète et exhaustive du système.

Exemple de synchronisation Parent - Enfant

Principe (sous Unix) : Tout processus qui se termine annonce à son parent sa fin par un signal `SIGCHLD`.

Le parent peut :

- prendre connaissance de cette fin avec les primitives `wait(status)` ou `waitpid()` et analyser des informations relatives à cette fin (s'est-il terminé normalement ? ...);
- ignorer toute fin de descendant.

L'enfant terminé est dans un état `zombi` (appelé aussi `defunct` dans certains systèmes) à partir de sa terminaison jusqu'à la prise en considération par le parent.

Noter que l'enfant ne sait pas que son parent ne fera pas de `wait()`, c'est pourquoi il reste dans un état `zombi` tant que le parent n'est pas terminé.

Signaux : exemples

- Une erreur de *segmentation* provoque l'expédition par le noyau de `SIGSEGV` au processus fautif (actif) ;
- `Ctrl C` (contrôle c) génère l'expédition de `SIGINT` dont le traitement par défaut est l'arrêt du processus ;
- `kill -9 32600` expédie le signal numéro 9=`SIGKILL` au processus numéro 32600 qui se terminera.

Il est **préférable** de désigner les signaux par leurs noms plutôt que par leurs numéros (**portabilité entre différents Unix**).

Exemple : `kill -SIGHUP 32600` est préférable à `kill -1 32600`, car portable.

Signaux : états et gestion

- un signal est **pendant** lorsqu'il a été enregistré dans la table des signaux pendants du processus mais qu'il n'a pas été pris en compte ;
- un signal est **délivré** au processus visé à la fin de son exécution en mode noyau juste avant de repasser en mode utilisateur ;
- une fonction *gestionnaire* (handler) est alors exécutée avant de repasser en mode utilisateur si le gestionnaire n'a pas terminé le pus (exit) ;
- Aucune information n'est associée à un signal : ce n'est qu'un bit positionné dans la table (perte possible de signal).

| Num | pendant ? | masqué ? | gestionnaire |
|--------|-----------|----------|---------------|
| 1 | 0 1 | 0 1 | void (*)(int) |
| 2 | 0 1 | 0 1 | void (*)(int) |
| ... | ... | ... | ... |
| NSIG-1 | 0 1 | 0 1 | void (*)(int) |

Gestion d'un signal

Le gestionnaire peut au choix :

- réaliser l'action prévue par défaut (souvent terminaison) `SIG_DFL`,
- l'ignorer (pas tous les signaux) `SIG_IGN`,
- gérer le signal (pas tous les signaux) avec une fonction ad hoc.

Particularités :

- Un processus peut recevoir plusieurs signaux du même type, mais le système ne mémorise qu'un seul signal par type de signal (un bit/type).
- Comme pour les interruptions (matérielles), l'arrivée d'un signal pendant le traitement d'un signal peut poser problème. Un mécanisme complexe de masquage permet de ne pas prendre en compte certains signaux pendant le traitement d'un signal.

Emission d'un signal

• Explicite

Seuls les processus issus du même propriétaire peuvent échanger des signaux (mis à part root).

appel système : `int kill(pid_t pid, int sig);`

commande : `kill [-s signal | -p] [-a] pid ...`

• Implicite

division par 0 36/0.0 engendre `SIGFPE` : "Floating Point Exception";

violation mémoire *segmentation fault* `SIGSEGV`

tube sans lecteur reçu lors d'un `write()` dans un tube par un écrivain sans lecteur : `SIGPIPE`!

mort d'un fils envoyé par un fils à son père lors de sa terminaison
`exit : SIGCHLD`

Gestion POSIX d'un signal

Pour Programmer la gestion d'un signal il faut :

- 1 écrire le gestionnaire en C (i.e. la fonction *handler* qui est de type :
`void gst(int signal);`);
- 2 créer une struct `sigaction` et la remplir en indiquant le gestionnaire ;
- 3 associer l'identité du signal et la structure créée précédemment grâce à la fonction `sigaction()`. Attention à l'homonymie !

Struct sigaction et fonction sigaction()

```
struct sigaction {
    void      (* sa_handler) (int); // le gestionnaire
    sigset_t   sa_mask;             // signaux à bloquer
                                     // pendant l'exécution du gst
    int        sa_flags;             // diverses options
                                     // dont SA_RESTART
    ...}

```

`sa_handler` ci-dessus est un *pointeur sur fonction*. Noter qu'en C, le *nom* d'une fonction est un pointeur sur son code.

La fonction sigaction()

```
int sigaction(int signum,          numéro du signal
               const struct sigaction *act, adrs de la struct
               struct sigaction *oldact); anc. pour réinstaller

```

Exemple : Gestion du signal SIGSEGV (*seg. fault*)

```
#include<stdio.h>
#include<stdlib.h>
#include<signal.h>
#include<errno.h>
void gst(int s){
    printf("gestion du signal %d ! Erreur num %d\n",s,errno);
    perror("Message d'erreur ");
    exit(1);
}
int main(int argc, char *argv[]){
    struct sigaction action;
    action.sa_handler = gst;
    sigaction(SIGSEGV, &action, NULL); /* association */
    int i; i=*(int *)NULL; /* émission du signal SIGSEGV */
    perror("Bizarre, d'habitude on n'en revient pas !\n");
}
```

Quelques Signaux

| Signal | Valeur | Action | Signal | Valeur | Action |
|---------|--------|--------|---------|----------|--------|
| SIGHUP | 1 | T | SIGPIPE | 13 | T |
| SIGINT | 2 | T | SIGALRM | 14 | T |
| SIGQUIT | 3 | T | SIGTERM | 15 | T |
| SIGILL | 4 | T | SIGUSR1 | 30,10,16 | T |
| SIGFPE | 8 | M | SIGUSR2 | 31,12,17 | T |
| SIGKILL | 9 | TEF | SIGCHLD | 20,17,18 | I |
| SIGSEGV | 11 | M | SIGCONT | 19,18,25 | |
| | | | SIGSTOP | 17,19,23 | DEF |

Action désigne l'action par défaut. SIGUSR_n sont des signaux sans signification particulière, laissés à la disposition de l'utilisateur.

T : terminer processus D : interrompre processus
 I : ignorer signal E : ne peut être géré
 M : image mémoire F : ne peut être ignoré

Gestion du signal SIGSEGV

```
Exemple$ segfault
gestion du signal 11 ! Erreur num 0
Message d'erreur : Undefined error: 0
Exemple$
```

Signal ≠ Erreur

Une erreur (`errno>0`) est détectée de manière synchrone à la suite d'un appel système retournant -1 ou NULL ;
 Une exception (ou trappe) provoque un signal suite à une instruction : SIGSEGV, SIGFPE, ...
 3.0/0.0 donne l'infini et pas SIGFPE ! La division entière par 0 provoque SIGFPE mais pas à l'initialisation !

Remarques

- Un seul bit par signal est utilisé dans la table du processus concerné ⇒ des signaux peuvent être perdus dans le cas d'expéditions en rafale.
- Pour ignorer un signal le gestionnaire s'appelle SIG_IGN : dans l'exemple, `action.sa_handler=SIG_IGN`.
- De même, SIG_DFL désigne le gestionnaire par défaut.
- L'association (signal ↔ gst) reste pérenne !
- Après une trappe, le gestionnaire doit terminer le processus sinon l'instruction ayant provoqué le signal est réexécutée à l'infini !

Questions et Réponses

- Faut-il relancer une entrée-sortie interrompue par un signal ?

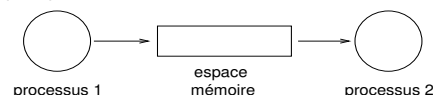
Certains systèmes le font, d'autres (dont linux) non. Dans ce dernier cas, il faut relancer l'entrée-sortie, en attribuant à `sa_flags` (voir la structure `sigaction`) la valeur `SA_RESTART`; dans l'exemple, `action.sa_flags=SA_RESTART`.

- Quand est-ce que le système consulte et avertit les processus de l'arrivée de signaux ?

Lors du passage du mode noyau au mode utilisateur (\neq temps réel).

Définition de Tube

Un *tube* est une file d'attente (fifo) **synchronisée** d'octets située dans une zone de mémoire accessible par plusieurs processus. Un processus (ou plusieurs) peu(ven)t y ajouter des octets (écrire) et/ou retirer des octets (lire).



Question : par rapport aux segments mémoire des processus, où est cet espace ?

Réponse : dans l'espace mémoire du système d'exploitation.

Plan

9 Communications basiques entre Processus (signaux et tubes)

- Les Besoins
- Signaux Unix
- Tubes Simples
- Tubes Nommés

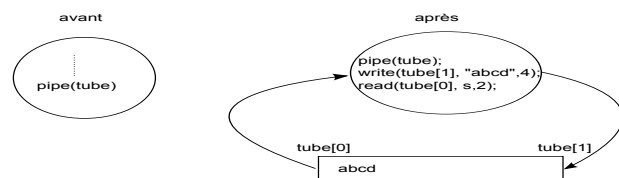
Caractéristiques des tubes simples

- la création d'un tube correspond à l'allocation d'un i-node ayant un compteur de liens à 0 ;
- il existe tant que le nombre d'ouvertures sur cet i-node n'est pas nul ;
- l'appel `sys. pipe(int tube[2])` crée 2 entrées dans la table des fichiers ouverts, l'une en lecture, l'autre en écriture qui pointent sur cet i-node ;
- la table des i-node des tubes correspond au système de fichier des tubes qui n'est pas associé à une partition disque ;

Tubes Simples

Un *tube simple* est une zone de donnée en mémoire créée par un processus :

- Un appel système spécifique de création : `pipe(int t[2]);`
- Résultat : deux descripteurs, permettant des accès par `read()` et `write()` puis la fermeture `close()`.

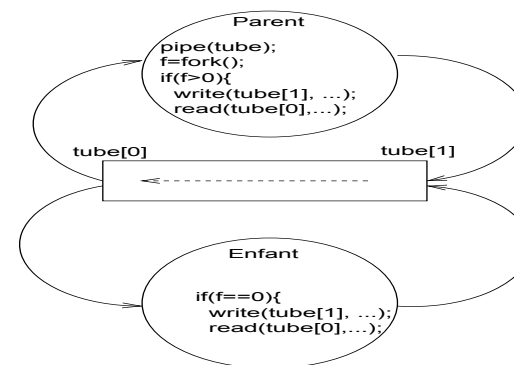


Exemple :

```
int tube[2]; //définition
int r=pipe(tube); //création
int n=write(tube[1], s, taille) ; // écriture
int k=read(tube[0], ch ,size) ; // lecture
```

Modèle Lecteur-Écrivain et Tubes

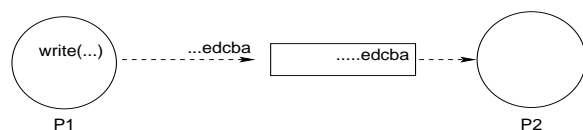
Reste à résoudre le problème du partage du tube par plus d'un processus. L'héritage des descripteurs lors d'un clonage (`fork()`) permet de répondre.



On dispose ainsi d'un espace mémoire dans lequel parent et enfant peuvent lire et écrire.

Tubes - Caractéristiques

- flot de caractères avec une gestion *premier entré, premier sorti* synchronisée.



- Tout élément lu est retiré du tube : c'est une file d'attente ... ;
- c'est un flot séquentiel qui ne permet pas l'accès direct (random access) \Rightarrow pas de déplacement avant ou arrière (pas de `lseek()`) ;
- un tube simple est limité à une hiérarchie de processus issue du processus créateur (donc appartenant à un seul utilisateur) ;
- la taille du tube est bornée ;
- les entrées-sorties sont *atomiques*². Approximation : toute opération commencée est seule à modifier le tube ; elle est entièrement terminée avant le passage à la suivante.

Tubes - Synchronisation

Le système prend en charge le fonctionnement suivant :

- Lecteur bloqué sur une demande de lecture lorsque le tube est vide.
- Écrivain bloqué sur une demande d'écriture lorsque le tube est plein.
- Lecteur averti s'il veut lire alors que le tube est vide et qu'il n'y a plus d'écrivains : `read()` retourne 0 (zéro) et c'est le seul cas où zéro est retourné. S'il y a moins de caractères que demandé, alors lecture partielle.
- Écrivain averti quand il veut écrire et qu'il n'y a plus de lecteurs, quel que soit l'état du tube : ce n'est *pas* le résultat de `write()` ! mais la réception du signal SIGPIPE qui matérialise l'avertissement.
- Une exclusion mutuelle pour l'accès au tube est assurée (toute entrée-sortie est terminée avant de réaliser une autre) : pas de perte ni de duplication.

Tubes - Interblocage

Un interblocage (deadlock) est une situation où un ou plusieurs processus se retrouvent mutuellement bloqués en attente d'une ressource qu'ils ne pourront jamais obtenir. La seule solution pour résoudre ce problème est la suppression d'un des processus participant.

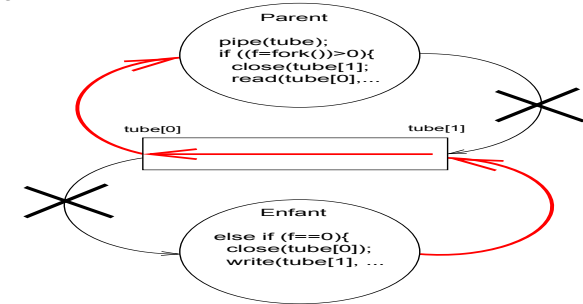
Exemple : Un processus créant un tube et tentant de lire dans ce tube est interbloqué. Dans la mesure où il existe encore un écrivain potentiel (lui-même), le noyau bloque ce processus indéfiniment sur un read que personne ne pourra satisfaire !

Pour éviter cet exemple simple, il convient de réfléchir au protocole de communication utilisant le tube.

Exemple : Dans le modèle précédent de lecteurs-écrivains où un parent et son fils lisent et écrivent dans le même tube, il peut également survenir un interblocage si les deux pus tentent de lire le tube vide : ils seront alors bloqués pour l'éternité.

Eviter l'interblocage

Une bonne règle à respecter concerne la **fermeture au plus tôt des descripteurs non utilisés**. Cela évitera des situations triviales d'interblocage.



On peut aussi faire des entrées-sorties non bloquantes (`pipe2(tube, O_NONBLOCK)`), mais il faudra alors prendre en charge la synchronisation.

Interprète de Langage de Commande et Tubes

Schéma algorithmique de bash face à : `ls | grep "\.c$"`

```
se cloner en clône1;
créer dans clône1 un tube tube ;
clôner clône1 en clône2 ;
si (processus clône1) alors
    dup2(tube[0],0) ; fermer tube[1] ;
    se recouvrir par grep ... ;
```

```
si (processus clône2) alors
    dup2(tube[1],1) ; fermer tube[0] ;
    se recouvrir par ls ;
```

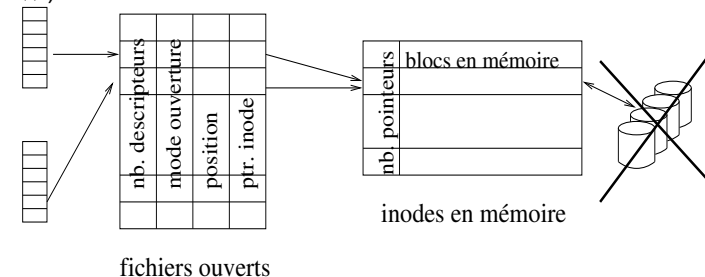
```
si (dernier caractère est &) alors reprendre lecture clavier;
;
```

```
sinon attendre la fin de clône1 ;;
```

`dup2(oldfd,newfd)` permet d'obtenir un second descripteur (`newfd`) référençant un fichier déjà ouvert (`oldfd`). Si besoin est, `newfd` aura été préalablement fermé !

Gestion des Tubes par le Système

Les tubes sont gérés comme un fichier, dans la table des fichiers ouverts du système. Mais contrairement aux fichiers, aucun transfert (disque ↔ mémoire centrale) n'est effectué et aucun déplacement (`lseek()`) n'est autorisé. Pour mémoire :



fichiers ouverts

Limites des Tubes Simples

Un rappel des limites des tubes simples vues précédemment :

- héritage obligatoire de descripteurs si on veut partager le tube entre plusieurs processus ;
- les processus appartiennent au même utilisateur ;
- le partage entre plus de deux processus peut devenir délicat, sauf si on ne cherche pas à savoir qui parmi les écrivains a écrit ou qui parmi les lecteurs a lu.

Les deux premières limites peuvent être dépassées avec les tubes nommés décrits dans le paragraphe suivant.

La dernière est inhérente au fonctionnement des tubes en général.

Présentation, Caractéristiques

Les tubes nommés offrent les possibilités générales de communication décrites pour les tubes, en étendant l'accès à des processus appartenant à des utilisateurs différents. Caractéristiques :

- Une structure localisée en mémoire centrale, toujours dans l'espace du système, identifiée de façon à permettre l'accès à des processus issus de propriétaires différents.
- Possibilité de Rendez-Vous entre processus.
- Des droits d'accès permettent d'autoriser ou interdire l'accès en fonction des propriétaires des processus.

Elles entraînent plusieurs questions :

- Qui (quel processus) va créer la structure ? Comment ?
- Comment identifier la structure ?
- Comment savoir qu'un processus est présent au Rendez-Vous ?

Plan

9 Communications basiques entre Processus (signaux et tubes)

- Les Besoins
- Signaux Unix
- Tubes Simples
- Tubes Nommés

Identification

Besoin : donner un identifiant à la structure pour que tout processus puisse demander l'accès.

Deux appels système vont être utilisés :

- 1 `mkfifo()` qui permet de réserver un nom, **sans** créer la structure en mémoire,
- 2 `open()` tout simplement, qui va créer la structure si elle n'existe pas et demander l'accès.

Principes :

- `mkfifo()` crée un fichier de type `p`, donc un fichier spécial, qui sert **uniquement** à mémoriser un nom, un propriétaire et des droits d'accès.
- une demande `open()` d'un fichier de type `p` provoque la **création** de la structure en mémoire si elle n'existe pas, réalise le **rendez-vous** (détails plus loin) et permet de faire des entrées-sorties conformément aux droits du fichier spécial.

Exemple

```
int r=mkfifo("monfifo",
S_IRWXU|S_IRGRP|S_IWGRP|I_IROTH) ;
```

va créer un **inode**, de type spécial `p`, avec les droits interprétés comme dans toute ouverture de fichier (ici, en octal, 764) et une entrée dans le répertoire de création.

Dans la table des inodes on aura :

| num. | type | droits | proprio. | ... | pointeurs |
|-------|------|------------------|----------|-----|-----------|
| 48761 | p | comme tout inode | | | vide |

Dans le répertoire on aura :

| num | nom |
|-------|---------|
| 48761 | monfifo |

Exemple - fin

Le lecteur effectuera :

```
int n=read(in,s,TAIILE);
```

et toutes les lectures se feront dans le tube, avec un blocage si **aucun** caractère n'est présent ; la lecture sera satisfaite (retour positif) dès que le tube ne sera pas vide \Rightarrow le résultat de `read()` contiendra la longueur réellement lue.

De même, si un autre processus écrivain fait

```
int out = open("monfifo", O_WRONLY) ;
```

alors ses écritures

```
int n=write(out,t,K);
```

retourneront le nombre de car. effectivement écrits.

Exemple - suite

Si un processus lecteur fait

```
int in=open("monfifo",O_RDONLY);
```

le système vérifie qu'il a les droits correspondants sur le fichier spécial, puis :

- si il existe des écrivains bloqués sur ce tube ;
- alors tous les débloquent ; // c'est le rendez-vous
- sinon créer la stucture fifo en mémoire puis se bloquer ; //attente rdv

La structure fifo sera gérée en mémoire comme un tube simple.

Toutes les entrées sur `in` se feront dans cette structure fifo.

Rendez-Vous

Question : Comment est réalisé le rendez-vous (qui attend qui) ?

le Rendez-Vous est mis en œuvre à l'ouverture, et garantit l'existence d'au moins un lecteur et d'au moins un écrivain.

`open()` est bloquant : lors d'une demande d'ouverture en lecture (resp. écriture), le système vérifie qu'il n'y ait pas au moins un écrivain (resp. lecteur). Sinon, le processus demandeur est endormi. Si oui, c'est que des écrivains attendent ; ils ont fait une demande `open()` en écriture et ont été bloqués ; le système les réveille.

Attention : Un interblocage est possible suite à des défauts de programmation (voir ci-après).

Fermeture

Question : Que se passe-t-il à la fermeture et éventuellement à la réouverture ?

Le fonctionnement est identique à celui des tubes simples concernant la synchronisation et l'avertissement des processus.

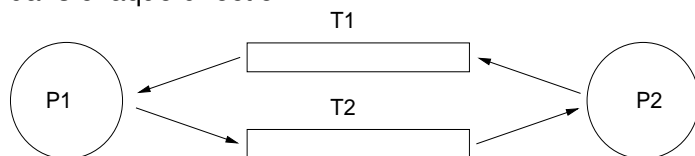
Donc lecteurs et écrivains seront avertis de l'absence d'acolytes (seulement lorsque le tube aura été vidé pour les lecteurs).

Lorsque tous les processus ont fermé leurs descripteurs, le fichier spécial n'est **pas détruit**.

Lorsqu'un processus a été débloquenté à la demande d'ouverture, il n'est plus possible de se remettre en attente. En d'autres termes, si un lecteur (resp. écrivain) reste seul, il ne sera pas endormi en attendant un autre processus, sauf si étant seul, il ferme le tube et le réouvre.

Interblocage - un Exemple

Deux processus veulent échanger des données en utilisant un tube nommé dans chaque direction.



La situation suivante :

| P1 | P2 |
|-----------------------------------|-----------------------------------|
| <code>ouvrir(T1, lecture)</code> | <code>ouvrir(T2, lecture)</code> |
| <code>ouvrir(T2, écriture)</code> | <code>ouvrir(T1, écriture)</code> |

provoque un interblocage, chaque processus attendant un déblocage.

Petit exercice : Un troisième processus peut les sauver.

Droits

Question : Quelles vérifications sont faites sur les droits d'accès ?

Comme pour les fichiers en fonction du propriétaire du processus demandeur. Il n'est pas nécessaire d'être propriétaire du fichier spécial pour créer la structure en mémoire (penser au rendez-vous toujours)

Noter que le fichier spécial reste **vide** tout au long de l'utilisation !

Seule la structure en mémoire se remplit et se vide. Le contenu d'un tube n'est donc **pas** conservé si tous les lecteurs sont partis en laissant des données orphelines dans le tube.

La destruction du fichier spécial se fait comme tout fichier (`rm` ou `unlink()`) et obéit aux mêmes vérifications de droits.

Une commande `mkfifo` existe aussi.

Plan

10 Thread

Plan

10 Thread

- Introduction
 - Les threads utilisateurs POSIX : pthread
 - Gestion de la concurrence entre pthreads
 - Un exemple : file d'attente partagée
 - Exemple 2 : paralléliser un produit de matrice
 - Conclusion sur les threads

thread noyau et thread utilisateur I

- les threads du noyau sont des entités du système d'exploitation (natives) et sont gérées dans l'espace système ; leur manipulation est réalisée à travers des appels systèmes (`man 2`, `clone()`)
- les threads utilisateurs (comme la bibliothèque pthread) sont gérés dans une bibliothèque (`man 3`) généralement portable sur plusieurs SE
- Afin d'assurer la portabilité des programmes et d'utiliser des concepts de plus haut niveau , il est préférable d'utiliser des threads utilisateurs (pthread)
- l'implémentation de la bibliothèque dans un S.E. peut reposer sur l'utilisation de threads noyaux ou pas (machine virtuelle)

Introduction

Un thread (ou fil d'exécution ou processus léger ou activité) est similaire à un processus par les aspects suivants :

- tous deux exécutent une séquence d'instructions en langage machine ;
- les exécutions semblent se dérouler en parallèle (temps partagé, ordonnancement) ;

Les différences essentielles :

- un thread s'exécute toujours dans un processus dont il dépend : la terminaison du processus entraîne la terminaison de ses threads
- les threads d'un même processus partagent :
 - leur segment de code, de données statique et dynamique (tas)
 - leur table des descripteurs de fichiers ouverts
 - il est donc indispensable de gérer leur **concurrence**
- chaque thread possède sa propre pile
- la création et le changement de contexte entre deux threads du même processus est donc très rapide

thread noyau et thread utilisateur II

Création d'un thread noyau sous Linux

```
int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg, ... );
```

crée un nouveau thread noyau exécutant la fonction `fn` grâce à la pile `child_stack`.

thread noyau et thread utilisateur III

Implantation des threads utilisateur dans un SE

- *Many to one* : tous les threads utilisateur sont implantés dans un unique thread noyau : l'ordonnanceur de thread est dans la bibliothèque (green thread Java)
- *One to one* : chaque thread utilisateur est associé à un unique thread noyau. Approche la plus simple qui est adoptée par **la bibliothèque pthread sous Linux**
- *Many to many* : différents threads utilisateur sont associés à un nombre inférieur ou égal de threads noyau (pthread sous Solaris)

Caractéristiques

- partage des segments de code, statiques et dynamiques
- chaque pthread à son propre **segment de pile** et son propre masque de signaux
- partage de l'unique table des descripteurs de fichiers
- partage des tampons utilisateurs (printf, fputs, ...)
- le pthread (main) existe initialement
- chaque pthread possède un identifiant de thread (tid) différent
- chaque pthread d'une même famille possède le même pid
- tout signal synchrone (SIGSEGV, SIGFPE) est délivré au thread fautif
- tout signal asynchrone (kill()) est délivré à l'un des pthread
- sous Linux, un fork() ne duplique que le pthread l'exécutant (différents Unix)

Plan

10 Thread

- Introduction
- **Les threads utilisateurs POSIX : pthread**
- Gestion de la concurrence entre pthreads
- Un exemple : file d'attente partagée
- Exemple 2 : paralléliser un produit de matrice
- Conclusion sur les threads

Création et terminaison d'un pthread I

L'API POSIX `pthread` définit un grand nombre de fonctions C de nom `pthread_xyz()` permettant de manipuler les processus légers. Cette API est présente sur tous les systèmes Unix mais aussi MacOS X, Windows ...

pthread_create

```
int pthread_create (pthread_t *p_tid, // thread id
                  pthread_attr_t attr, // attributs
                  void *(*fonction) (void *arg), // LA fonction
                  void *arg // les arguments passés à la fonction
                );
```

- retourne 0 si OK, sinon code d'erreur
- l'id du nouveau thread est placé à l'adresse `p_tid`;

Création et terminaison d'un pthread II

- `attr` attribut (joignable ou détaché) ..., utiliser `pthread_attr_default` ou `NULL` ;
- fonction correspond à la fonction exécutée après la création : point d'entrée (`main`). Un retour de cette fonction correspondra à la terminaison de ce thread ;
- `arg` est transmis à la fonction au lancement de l'activité ;
- au lancement d'un `pus`, un thread est créé qui exécute le `main` ; à sa terminaison, tous les threads et le processus terminent ;
- `exit()` termine tous les threads et le processus ;

Création et terminaison d'un pthread III

pthread_exit

```
int pthread_exit (void *retval);
```

- `retval` valeur retour de la thread
- le thread termine

pthread_cancel

```
int pthread_cancel(pthread_t tid);
```

- tente de résilier `tid` (non bloquant)
- le thread doit atteindre un point de résiliation pour être terminé
- retourne 0 ou code d'erreur

Terminaison propre de thread

pthread_join

```
int pthread_join(pthread_t tid, void **retval);
```

- équivalent du `wait` des processus, par défaut un thread est joignable
- l'appelant attend la fin de `tid`, récupère son résultat et libère les ressources
- on ne peut attendre un thread détaché

thread détaché

Un thread peut être détaché :

- à sa création (attributs)
 - pendant son exécution
- ```
int pthread_detach (pthread_t *tid);
```

## Terminaison

Un thread termine dans l'une des conditions suivantes :

- il appelle `pthread_exit()` en spécifiant une valeur d'exit accessible par n'importe quel autre thread du même processus appelant `join`
- il retourne de sa fonction en spécifiant une valeur qui pourra être récupérée par un `join`
- il est supprimé par `pthread_cancel()` par l'un de ses pairs ; il peut être joint ensuite et sa valeur sera `PTHREAD_CANCELED` ; on peut désactiver/activer le fait qu'un thread soit destructible
- un des threads du processus appelle `exit()` ou le `main` retourne ; cela termine tous les threads ;

## Un exemple simple I

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void * travail(void* arg){
 printf("%u: travail(%s)\n",pthread_self(),(char*) arg);
 return (void *)"fini !";
}
int main(int argc, char * argv[]){
 pthread_t tid; /* id du thread */
 if (0 != pthread_create (&tid, NULL, travail, "Bonjour !")){
 // (id, attributs, fonction à exécuter, arg de la fon)
 printf("création du thread impossible !\n");
 exit(1);
 }
 printf("%u: thread %u créé et lancé !\n",pthread_self(),tid);
 void* res;
 pthread_join(tid, &res); // wait du thread et récup retour
```

## Plan

### 10 Thread

- Introduction
- Les threads utilisateurs POSIX : pthread
- **Gestion de la concurrence entre pthreads**
- Un exemple : file d'attente partagée
- Exemple 2 : paralléliser un produit de matrice
- Conclusion sur les threads

## Un exemple simple II

```
printf("%u: thread %u terminé en renvoyant \"%s\"\n",
 pthread_self(), tid, (char *)res);
return 0;
}
```

### Compilation et édition de liens

```
gcc -g -Wall -std=c99 -c thrEx1.c
gcc -g -Wall -std=c99 -o thrEx1 thrEx1.o -lpthread
```

### Trace de l'exécution

```
$./thrEx1
2474460096: thread 149549056 créé et lancé !
149549056: travail(Bonjour !)
2474460096: thread 149549056 terminé en renvoyant "fini !"
$
```

## La concurrence sauvage et ses effets I

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

int compteur=0; // var statique

void * travail(void* arg){
 for(int i=0; i<10000; i++)
 compteur++;
 return NULL;
}

int main(int argc, char * argv[]){
 pthread_t tid1,tid2; /* ids des threads */
 if (0 != pthread_create (&tid1, NULL, travail, NULL) ||
 0 != pthread_create (&tid2, NULL, travail, NULL)){
 printf("création du thread impossible !\n");
 exit(1);
 }
 travail(NULL); // appel par thread main
 void* res;
 pthread_join (tid1, &res);
```



## La concurrence sauvage et ses effets II

```
pthread_join (tid2, &res);
printf("Valeur finale du compteur incrémenté 30000 fois : %d \n",
compteur);
return 0;
}
```

### Trace de l'exécution

```
Exemple$ thrEx2
Valeur finale du compteur incrémenté 30000 fois : 22007
Exemple$ thrEx2
Valeur finale du compteur incrémenté 30000 fois : 23076
Exemple$ thrEx2
Valeur finale du compteur incrémenté 30000 fois : 22577
```

**La non-atomicité de l'opérateur d'incrémentement produit des incohérences !**

## Section critique avec les mutex II

### Modification de l'exemple précédent

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
// mutex statique initialisé

void * travail(void* arg){
 for(int i=0; i<100000; i++){
 pthread_mutex_lock(&mutex);
 compteur++;
 pthread_mutex_unlock(&mutex);
 }
 return NULL;
}
```

## Section critique avec les mutex I

- Un mutex est un **sémaphore** à deux états soit libre, soit verrouillé
- il peut être initialisé statiquement ou grâce à une fonction

### Verrouillage

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

trylock est non bloquant contrairement à lock

### Déverrouillage

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

le sémaphore est à nouveau libre

## Section critique avec les mutex III

### Trace de l'exécution

```
Exemple$ thrEx3
Valeur finale du compteur incrémenté 30000 fois : 30000
Exemple$ thrEx3
Valeur finale du compteur incrémenté 30000 fois : 30000
Exemple$ thrEx3
Valeur finale du compteur incrémenté 30000 fois : 30000
```

### Sémaphores généraux non nommés

- initialisation  
int sem\_init(sem\_t \*sem, int pshared, int value);
- Puis-je() bloquant ou non bloquant :  
int sem\_wait(sem\_t \*sem); int sem\_trywait(...);
- Vas-y() : int sem\_post(sem\_t \*sem);

## Les conditions (Moniteurs de HOARE) I

L'exclusion mutuelle n'est pas le seul modèle de concurrence !

On souhaite conditionner l'exécution d'une section de code à un état atteint par une variable d'un type quelconque, par exemple :

- une file d'attente n'est pas pleine
- une variable est strictement positive

Une variable de type `pthread_cond_t` sera utilisé conjointement à un mutex afin qu'un thread puisse être bloqué sur cette condition grâce à `pthread_cond_wait()`. Ce thread sera débloqué lorsqu'un autre thread signalera le changement d'état grâce à `pthread_cond_signal()`. Remarquons que le `wait` relâche le mutex préalablement verrouillé.

## Les conditions (Moniteurs de HOARE) II

Schéma du thread attendant la réalisation de la condition

```
int x,y; // la condition est x>y
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
...
pthread_mutex_lock(&mut); // Section Critique
while (x <= y) { // si condition fausse
 pthread_cond_wait(&cond, &mut);
 // se bloquer sur cond en débloquent mutex
}
/* agir sur x et y */
pthread_mutex_unlock(&mut); // fin Section Critique
```

## Les conditions (Moniteurs de HOARE) III

Schéma du thread réalisant la condition

```
...
pthread_mutex_lock(&mut); // Section Critique
x=y+1; // la condition est x>y
pthread_mutex_unlock(&mut); // fin Section Critique
pthread_cond_signal(&cond); // débloquent un "waiter"
```

## API des conditions I

### wait

```
int pthread_cond_wait(pthread_cond_t *cond,
 pthread_mutex_t *mutex);
```

déverrouille atomiquement le mutex et bloque le thread sur la condition `cond`. Lorsque `cond` sera signalée, un thread sera débloqué et redemandera le verrouillage du mutex

### signal

```
int pthread_cond_signal(pthread_cond_t *cond);
```

relance l'un des threads attendant la variable condition `cond`. S'il n'existe aucun thread attendant, rien ne se produit et rien n'est mémorisé.

## API des conditions II

### diffusion de signaux

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

relance tous les threads attendant sur la variable condition cond. Rien ne se passe s'il n'y a aucun thread attendant sur cond.

### initialisation d'une cond

```
int pthread_cond_init(pthread_cond_t *cond,
 pthread_condattr_t *cond_attr);
ou
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

La seconde forme n'est possible que pour une initialisation statique. Dans la première forme `cond_attr` est NULL la plupart du temps.

## Lien entre mutex et condition I

- L'appel `pthread_cond_wait(&cond, &mutex)` provoque le blocage du thread appelant sur cond **après** avoir déverrouillé (unlock) le mutex
- Ainsi plusieurs *waiters* peuvent être bloqués sur la condition
- Lorsqu'un signal est appelé par un autre thread, un seul waiter aléatoire est débloqué de la cond puis **tente un lock du mutex associé** qu'il avait perdu avant de retourner à son code "normal"
- Lorsqu'un `broadcast` est appelé par un autre thread, tous les waiters sont débloqués de la cond et tentent un lock du mutex associé : un seul l'obtiendra, les autres resteront bloqués sur le mutex (mais plus sur la cond)
- l'appel à `signal` ou `broadcast` peuvent (doivent) être effectués en dehors de toute section critique

## API des conditions III

### Destuction d'une cond

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

détruit une variable condition en libérant les ressources qu'elle possède. Aucun thread ne doit attendre sur la condition.

## Plan

### 10 Thread

- Introduction
- Les threads utilisateurs POSIX : pthread
- Gestion de la concurrence entre pthreads
- Un exemple : file d'attente partagée
- Exemple 2 : paralléliser un produit de matrice
- Conclusion sur les threads

## Un exemple : file d'attente partagée I

On souhaite programmer une file d'attente partagée (type queue) d'objets génériques (`void *`) où des thread puissent ajouter un objet à la fin et en retirer un en tête

- une queue est implantée dans un tableau initialisé avec une taille maximale à sa création
- elle possède un indice de tête et une taille courante toutes deux initialisées à 0
- au fur et à mesure des ajouts et retraits la queue va se déplacer dans le tableau considéré comme "circulaire" (la case suivant celle d'indice `tailleMax-1` est la case d'indice 0)
- chaque queue possède un `mutex` et deux conditions `condNonPlein`, `condNonVide` afin de bloquer les ajouts sur queue pleine et les retraits sur queue vide

## Un exemple : file d'attente partagée II

### Le type queue

```
typedef struct{
 uint32_t tailleMax; // taille du tableau qui stocke
 uint32_t tete; // indice de tête
 uint32_t taille; // taille de la file d'attente
 void **tab; // tableau contenant la queue
 pthread_mutex_t *mutex;
 pthread_cond_t *condNonPlein;
 pthread_cond_t *condNonVide;
} queue;
```

## Un exemple : file d'attente partagée III

### Constructeur

```
queue* queueCreer(uint32_t tailleMax){
 queue *q=malloc(sizeof(queue));
 q->tailleMax=tailleMax;
 q->tete=0;
 q->taille=0;
 q->tab=(void**) malloc(tailleMax*sizeof(void*));
 q->mutex=malloc(sizeof(pthread_mutex_t));
 pthread_mutex_init(q->mutex, NULL);
 q->condNonPlein=malloc(sizeof(pthread_cond_t));
 pthread_cond_init(q->condNonPlein, NULL);
 q->condNonVide=malloc(sizeof(pthread_cond_t));
 pthread_cond_init(q->condNonVide, NULL);
 return q;
}
```

## Ajouter un objet à la fin de la queue

```
queue* queueAjouter(queue *q, void *o){
 pthread_mutex_lock(q->mutex); // Section Critique
 while(q->taille==q->tailleMax){ // queue pleine !
 pthread_cond_wait(q->condNonPlein, q->mutex);
 // débloquent le mutex et attend un signal
 } // la queue n'est pas pleine : on peut ajouter
 uint32_t pos=(q->tete+q->taille)%q->tailleMax;
 q->tab[pos]=o; // ajout
 q->taille++;
 if(q->taille==1) // plus vide : je débloquent tous
 pthread_cond_broadcast(q->condNonVide);
 pthread_mutex_unlock(q->mutex); // fin de SC
 return q;
}
```

## Retirer un objet en tête

```
void * queueRetirer(queue *q){
 pthread_mutex_lock(q->mutex); // Section Critique
 while(q->taille==0){ // queue vide !
 pthread_cond_wait(q->condNonVide, q->mutex);
 // débloque le mutex et attend un signal
 } // la queue n'est pas vide : on peut retirer
 void *o=q->tab[q->tete];
 q->tab[q->tete]=NULL;
 q->tete=(q->tete+1)%q->tailleMax;
 q->taille--;
 if(q->taille==q->tailleMax-1) // plus plein
 pthread_cond_broadcast(q->condNonPlein);
 pthread_mutex_unlock(q->mutex); // fin de SC
 return o;
}
```

## Thread Producteur, thread consommateur II

```
/** Consommateur retirant n entiers de la queue */
void * cons(void* a){
 for(int i=0; i<((argument*)a)->n; i++){
 int *pi=(int *)queueRetirer(((argument*)a)->q);
 printf("%d retiré; ",*pi);
 free(pi);
 }
 return NULL;
}
```

## Thread Producteur, thread consommateur I

```
typedef struct{queue *q; int n;} argument;

/** Producteur ajoutant n entiers dans la queue */
void * prod(void* a){
 for(int i=0; i<((argument*)a)->n; i++){
 int j=rand()%1000; // entier aléatoire [0,1000[
 int *pj=malloc(sizeof(int));
 *pj=j;
 queueAjouter(((argument*)a)->q, (void *)pj);
 printf("%d ajouté; ",*pj);
 }
 return NULL;
}
```

## Le main I

```
queue *q; // une seule queue

int main(int argc, char * argv[]){
 srand(time(NULL)); // initialisation de rand
 q=queueCreer(atoi(argv[1])); // à faire varier ..
 argument ac;ac.q=q;ac.n=100; // nb retraits
 pthread_t tidc,tidp; /* id des threads */
 for(int i=0;i<3;i++){ // 3 conso
 if (0 != pthread_create(&tidc, NULL, cons, &ac)){
 printf("création du thread impossible !\n");
 exit(1);
 }
 } // 3 conso *100
```

## Le main II

```

argument ap; ap.q=q; ap.n=75; // nb retraits
for(int i=0; i<4; i++){ // 4 prod
 if (0 != pthread_create(&tidp, NULL, prod, &ap)){
 printf("création du thread impossible !\n");
 exit(1);
 }
} // 4 prod * 75
void* res;
pthread_join (tidc, &res); // on attend un cons
printf("fin du main \n");
return 0;
}

```

## Plan

### 10 Thread

- Introduction
- Les threads utilisateurs POSIX : pthread
- Gestion de la concurrence entre pthreads
- Un exemple : file d'attente partagée
- **Exemple 2 : paralléliser un produit de matrice**
- Conclusion sur les threads

## Quelques exécutions

```

Exemple$ thrProdCons 1
995 ajouté; 995 retiré; 881 ajouté; 881 retiré; 48 ajouté; 48 retiré;
809 ajouté; 809 retiré; 701 ajouté; 701 retiré; 707 ajouté; 707 retiré;
306 ajouté; 306 retiré; 768 ajouté; 768 retiré; 138 ajouté; 138 retiré;
...
Exemple$ thrProdCons 2
5 ajouté; 501 ajouté; 5 retiré; 501 retiré; 349 ajouté; 797 ajouté;
349 retiré; 357 ajouté; 797 retiré; 357 retiré; 515 ajouté; 580 ajouté;
515 retiré; 324 ajouté; 580 retiré; 97 ajouté; 324 retiré; 342 ajouté;
...
Exemple$ thrProdCons 20
985 ajouté; 715 ajouté; 987 ajouté; 509 ajouté; 985 retiré; 715 retiré;
987 retiré; 32 ajouté; 538 ajouté; 211 ajouté; 779 ajouté; 509 retiré;
32 retiré; 538 retiré; 852 ajouté; 845 ajouté; 592 ajouté; 770 ajouté;
211 retiré; 779 retiré; 852 retiré; 180 ajouté; 387 ajouté; 491 ajouté;
...

```

## paralléliser un produit de matrice

- le calcul d'un produit de 2 matrices carrées ( $m_1$ ,  $m_2$ ) de  $n$  lignes par  $n$  colonnes a une complexité en  $O(n^3)$
- $(n \text{ multiplications} + n-1 \text{ additions}) * n^2$  cases
- l'accès aux cases des deux matrices données est en lecture
- l'accès aux cases de la matrice résultat ( $mr$ ) est en écriture mais sans concurrence car on écrit une seule fois dans chaque case
- on peut donc décomposer le calcul en le confiant à plusieurs threads :
  - 1 thread par case calculée de  $mr$ , soit  $n^2$  threads. Une étude avec  $n=1000$  produisant donc  $10^6$  threads n'est pas concluante. Le calcul est trop court donc le coût de gestion des threads est prohibitif. De plus, selon les machines, il est impossible de créer autant de thread.
  - 1 thread par colonne calculée de  $mr$ , soit  $n$  threads. Une étude avec  $n=1000$  produisant donc 1000 threads est concluante et détaillée par la suite.

## 1 thread par colonne calculée

```
mat* matProdThread(mat *m1, mat* m2){
 if(m1->nbc == m2->nbl){ // multiplication possible
 mat* mr=matCreer(m1->nbl,m2->nbc);
 pthread_t tabth[m2->nbc]; /* id des threads pour les join */
 for(int k=0;k<m2->nbc;k++){ // pour chaque colonne de mr et de m2
 arg* a=malloc(sizeof(arg));
 a->m1=m1; a->m2=m2; a->mr=mr; a->k=k;
 if (0 != pthread_create (&tabth[k], NULL, matProd1, a)){
 perror("Création du thread impossible !");
 exit(1);
 }
 }
 void* res;
 for(int k=0;k<m2->nbc;k++){
 pthread_join(tabth[k], &res); // attendre tous les thread
 }
 return mr;
 } else {
 return NULL;
 }
}
```

## La fonction de thread

```
void * matProd1(void* argr){ // calcul d'une colonne k de mr
 arg *a=(arg*)argr;
 double cumul=0.0;
 for(int i=0;i<a->m1->nbl;i++){ //pour ligne de m1 et de mr
 cumul=0.0;
 for(int j=0;j<a->m1->nbc;j++){ //pour col de m1 et ligne de m2
 cumul+=a->m1->tab[i][j]*a->m2->tab[j][a->k];
 }
 matSet(a->mr,i,a->k,cumul);
 }
 free(a);
 return NULL;
}
```

## Tests de performance I

### Programme prodmat2.c

- 1 seul programme calculant séquentiellement si 3 arguments, parallèlement si 4 arguments
- Les 3 premiers argument :
  - nb lignes de m1 et mr
  - nb colonnes de m1==nb de lignes de m2
  - nb colonnes de m2 et mr
- les matrices sont créées dans le tas et remplies aléatoirement
- des tests de consistance ont été effectués
- deux machines cibles : Mac OSX (2 cœurs), Linux (2x6 cœurs)

## Tests de performance II

### Mac OSX, 2.6 GHz, 2 cores

```
$ time prodmat2 1000 1000 1000 > /dev/null
```

```
real 0m19.345s
user 0m18.959s
sys 0m0.118s
```

```
$ time prodmat2 1000 1000 1000 th > /dev/null
```

```
real 0m11.918s
user 0m40.337s
sys 0m0.185s
```

## Tests de performance III

### Linux, 2.3 GHz, 2 CPU x 6 cores

```
$ time prodmat2 1000 1000 1000 > /dev/null

real 0m10,607s
user 0m10,517s
sys 0m0,040s

$ time prodmat2 1000 1000 1000 th > /dev/null

real 0m3,511s
user 0m11,411s
sys 0m0,103s

$ cat /proc/cpuinfo
... info sur le CPU
```

## Performances II

### Résultats

- un rapport de 3 à 4 confirme l'utilité de paralléliser les calculs complexes
- les machines utilisées sont plus ou moins chargées durant les tests (Linux multi-utilisateurs)
- la répartition des cœurs aux threads est réalisée par l'ordonnanceur ...
- la commande `htop` de Linux visualise la charge des processeurs et cœurs

## Performances I

### Linux en augmentant le temps de calcul

```
$ time prodmat2 10000 1000 1000 > /dev/null

real 1m40,112s
user 1m39,463s
sys 0m0,280s

$ time prodmat2 10000 1000 1000 th > /dev/null

real 0m28,598s
user 1m51,098s
sys 0m0,322s
```

## Plan

### 10 Thread

- Introduction
- Les threads utilisateurs POSIX : pthread
- Gestion de la concurrence entre pthreads
- Un exemple : file d'attente partagée
- Exemple 2 : paralléliser un produit de matrice
- Conclusion sur les threads



## Conclusion

- détachement : lors de la terminaison d'un thread détaché, ses ressources sont désallouées aussitôt, il ne peut être "joint"
- un code est dit "thread safe" lorsqu'il peut être exécuté par plusieurs threads concurrents sans mener à une incohérence
- une "race condition" ou situation de concurrence peut survenir dès que plusieurs processus (légers ou lourds) tentent d'accéder à une ressource partagée et qu'au moins l'un d'entre eux tente de modifier son état. On résout cette situation grâce aux outils de synchronisation (sémaphore, moniteurs, ...)
- la réentrance est la propriété pour une fonction d'être utilisable simultanément par plusieurs tâches. La réentrance permet d'éviter la duplication en mémoire vive de cette fonction. `strtok` est un exemple de fonction C non réentrante

## Conclusion I

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

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

## Plan

### 11 Conclusion

## Conclusion II

- d'oser utiliser des systèmes d'exploitation dont le code source est connu !