

Concepts et Progr. Système (HLIN504)

Michel Meynard

UM

Univ. Montpellier

Plan

1 Introduction

- Présentation du cours
- Composantes d'un S.E.
- Programme, processus, et contexte
- Prérequis matériels
- Fonctionnement de la Pile (stack)
- Noyau et appels systèmes

Table des matières

- 1 Introduction
- 2 Représentation de l'information
- 3 Développement d'applications en C
- 4 Gestion des Entrées-Sorties
- 5 Gestion des processus
- 6 Système de Gestion des fichiers
- 7 Communications basiques entre Processus (signaux et tubes)
- 8 Synchronisation et Communication Entre Processus
- 9 Thread

Déroulement

- Cours 16.5h, TD 16.5h, TP 16,5h ;
- Contrôle : examen écrit (70/100), note CC (30/100) ;
- Prérequis : architecture, programmation C ;

Type de systèmes étudiés : monoprocesseur, multitâche.

Contenu du cours

Les points suivants seront étudiés en théorie puis en pratique en utilisant la programmation en C sous Unix.

- ➊ Besoins et rôles d'un système d'exploitation ; composantes d'un système ; noyau ; prérequis matériels ; contexte d'exécution ; pile.
- ➋ Gestion des Entrées-Sorties
- ➌ Processus : constituants, vie, ordonnancement, génération.
- ➍ Gestion des signaux
- ➎ Gestion de l'espace disque, gestion des fichiers : système de gestion des fichiers ; représentation des fichiers et répertoires ; inodes, partitions et parcours. Gestion des fichiers ouverts.
- ➏ Communications entre processus : moyens de communication simples et évolués ; synchronisation de processus et cohérence des accès.
- ➐ Gestion des threads ou fils d'exécution ou processus légers

Plan

➊ Introduction

- Présentation du cours
- Composantes d'un S.E.
- Programme, processus, et contexte
- Prérequis matériels
- Fonctionnement de la Pile (stack)
- Noyau et appels systèmes

Bibliographie

Andrew Tanenbaum, *Systèmes d'exploitation*, 2^{ème} éd., Campus Press, 2003

J.M. Rifflet, J.B. Yunès *Unix, Programmation et communication*, Dunod, 2003

Gary Nutt, *Operating systems, a modern perspective*, 3rd edition, Addison-Wesley, 2003

Samia Bouzefrane, *Les systèmes d'exploitation, Unix, Linux et Windows XP avec C et Java*, Dunod, 2003

D.P. Bovet, M. Cesati, *Le Noyau Linux*, O'Reilly, 2001

M. J. Bach, *The design of the Unix operating system*, Prentice-Hall, 1986.

Composantes d'un Système d'exploitation

Un système d'exploitation doit gérer les ressources, c'est-à-dire partager les ressources communes de la machine et les allouer *au mieux* aux processus des utilisateurs.

gestion des processus

- allocation de la ressource UC (Unité Centrale) : plusieurs processus demandeurs, mais un seul élu à la fois ;
- illusion de parallélisme : temps partagé par **quantum** de temps ;
- Assurer l'isolement et la communication entre processus ;
- éviter la **famine** : un pus attend indéfiniment ;
- **ordonnancement** (scheduling) des pus ;

Composantes d'un Système d'exploitation - suite

gestion de l'espace disque

- arborescence de répertoires et fichiers ;
- répondre aux demandes d'allocation et libération de l'espace : création, suppression, modification de fichiers et répertoires ;
- **protection** par les droits d'accès des fichiers entre utilisateurs ;

gestion de la mémoire

- répondre aux demandes d'allocation et libération d'espace ;
- protéger les accès ;
- masquer l'espace physique (*mémoire virtuelle*) ;

gestion des Entrées/Sorties E/S

- entrées/sorties généralisées : lire depuis le clavier ou depuis un fichier de la même façon ;
- efficacité pour le système : réaliser les transferts, au moment *le plus opportun* ;
- gestion **tamponnée** (buffer) des E/S ;

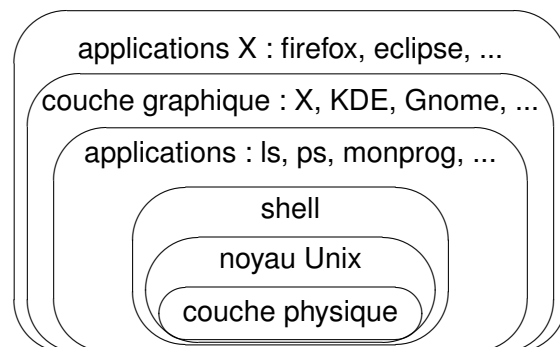
Interface de Programmation d'Applcation (API)

Le système Unix offre une API naturelle en C puisque le système est écrit en C !

Cette API se décompose en 2 niveaux :

- **appels systèmes** : fonctions de bas niveau appelant directement le noyau ; décrits dans la section 2 du manuel (`man 2 write`)
- **fonctions de la bibliothèque standard C** : opérations de plus haut niveau ; décrits dans la section 3 du manuel (`man 3 printf`)

Structure en couches d'un système



Plan

1

Introduction

- Présentation du cours
- Composantes d'un S.E.
- Programme, processus, et contexte
- Prérequis matériels
- Fonctionnement de la Pile (stack)
- Noyau et appels systèmes

Programme et processus

Un programme peut être écrit dans un langage :

- **interprété** : *script* bash ou python exécuté par l'interprète ;
- **compilé** : le *source* est compilé en *objet* puis il est *lié* avec d'autres objets et des *bibliothèques* afin de produire un *binaire* ;

Dans les deux cas, le *script* et le *binaire* doivent être *exécutables*, c'est à dire qu'ils doivent posséder le droit d'exécution `x` pour l'utilisateur.

Un processus est un programme *exécutable* en cours d'exécution. Par la suite, on s'intéressera principalement aux binaires C.

Fabrication du binaire

En deux étapes : *compilation* puis *édition de liens* :

```
gcc -c -g -ansi -Wall -std=c99 hello.c
gcc -o hello hello.o
```

Toujours en deux étapes mais avec 1 seule commande :

```
gcc -g -ansi -Wall -std=c99 -o hello hello.c
```

Options : **C**ompil seulement, **g** debug, **ansi** norme du C, **Warning** all, **std** c99 pour les commentaires à la C++ ...

Programme C : hello.c

```
#include <stdio.h> /* printf */
#include <stdlib.h> /* malloc */
#include <string.h> /* strcat */

int global=0;
char *concat(const char *s1, const char *s2){ /* concat */
    char *s=(char*)malloc(sizeof(char)*(strlen(s1)+strlen(s2)+1));
    return strcat(strcpy(s,s1),s2);
}
int nbappels(){
    static int nb=0;
    return ++nb;
}
int main(int n, char *argv[], char *env[]){
    nbappels();nbappels();nbappels();
    printf(concat("Bonjour ", "le monde !\n"));
    printf("Nb appels : %d\n",nbappels());
}
```

Déclaration et définition en C

- **déclaration** d'objet : indication du type de l'identificateur ; par exemple : `extern int i; void f();`
- **définition** d'objet : réservation d'espace mémoire ; par exemple : `int i; void f(){return;}`

Un même objet peut être déclaré plusieurs fois mais doit être **défini une seule fois**. Dans cette définition, il peut être **initialisé**.

Les fichiers **d'en-tête** (headers ou .h) ne contiennent généralement que des déclarations. Il sont de 2 sortes :

- `#include <stdio.h>` en-tête système dans `/usr/bin/include/;`
- `#include "mon.h"` en-tête personnel dans • ;

Durée de vie et visibilité

- une variable définie en dehors de toute fonction est **globale** : sa durée de vie égale celle du processus et sa visibilité est totale ;
- une variable définie en dehors de toute fonction et précédée de `static` a une durée de vie égale celle du processus et sa visibilité est locale au fichier ;
- une définition de **fonction** précédée de `static` limite sa visibilité au fichier ;
- une variable définie dans une fonction et précédée de `static` a une durée de vie égale celle du processus et sa visibilité est locale à la fonction : elle n'est initialisée qu'une seule fois !
- une variable définie dans une fonction est **locale** a une durée de vie égale celle de la fonction et sa visibilité est locale à la fonction ;

Segments de mémoire d'un processus

On distingue dans un processus quatre segments de mémoire :

- 1 **Code** contenant la suite des instructions machine (le programme binaire).
- 2 **Données statiques** contenant les données définies hors de toute fonction (dites *globales*) ou définies *static* dans une fonction.
- 3 **Pile** contenant :
 - les adresses de retour des fonctions appelantes,
 - les paramètres d'appels et le résultat,
 - les variables locales.
- 4 **Tas** contenant les objets dynamiques (*new*, *malloc*, ...)

Parfois un cinquième segment de données statiques non initialisées existe ...

Pointeurs et objets dynamiques

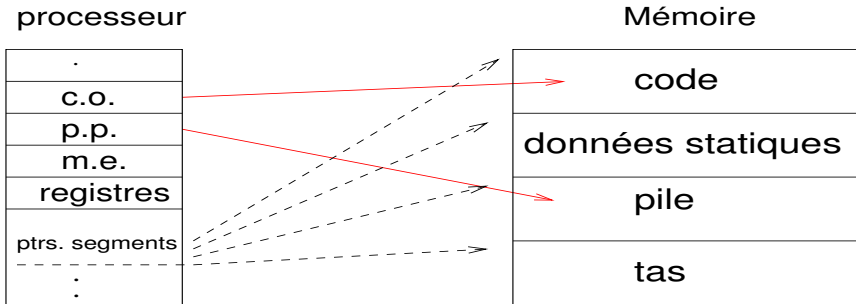
- Un objet dynamique (par opposition à `static`) peut être créé pendant l'exécution du processus grâce à la fonction `malloc(taille)` qui retourne un pointeur sur la zone du **tas** réservée.
- Ensuite, l'objet peut être modifié dans n'importe quelle fonction qui a connaissance de son adresse.
- Enfin, l'objet devra être détruit par `free(ptr)` afin de désallouer l'espace qui avait été réservé et d'éviter toute **fuite mémoire**.

Les objets dynamiques peuvent être de n'importe quel type (tableau, structure, entier, ...) et sont gérés dans le segment de **tas**.

Durée de vie des objets dans les segments

- 1 **Code** : durée de vie égale à celle du processus ; accessible en lecture seule.
- 2 **Données statiques** : durée de vie égale à celle du processus ; les données statiques sont créées dès le lancement du processus et sont détruites avant la destruction du processus lui-même ;
- 3 **Pile** : une variable locale est créée à l'exécution même de l'instruction de définition ; elle est détruite à la fin du bloc (accolade fermante dans les langages comme C, C++, java, etc.).
- 4 **Tas** : la durée de vie d'un objet dans le tas va de sa création explicite (`malloc`) jusqu'à sa destruction explicite (`free`).

Contexte d'exécution d'un processus



Le contexte d'exécution d'un processus est composé :

- le contenu des registres du processeur (compteur ordinal, pointeur de pile, mot d'état, ...);
- les segments de code, de pile et de données;
- les informations systèmes relatives au processus : utilisateur(s), groupe, identifiant de pus, identifiant du parent de pus, priorité, ...

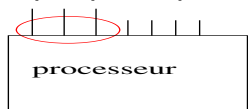
Plan

1 Introduction

- Présentation du cours
- Composantes d'un S.E.
- Programme, processus, et contexte
- **Prérequis matériels**
- Fonctionnement de la Pile (stack)
- Noyau et appels systèmes

Interruption - système mono-tâche

broches d'interruption
vers périphériques

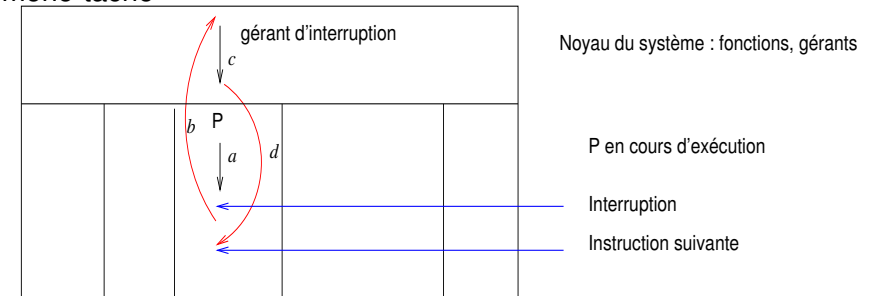


Mécanisme matériel, permettant la prise en compte d'un événement extérieur, **asynchrone**. Fonctionnement :

- 1 l'instruction en cours est terminée;
- 2 sauvegarde du contexte minimal (CO, PP, Mot d'État (PSW));
- 3 une adresse prédéterminée est forcée dans CO; à partir de là tout se passe comme tout appel de fonction;
- 4 la fonction exécutée s'appelle *gérant d'Interruption*; on dit que le gérant correspondant à l'interruption est lancé;
- 5 fin du gérant et retour à la situation avant l'interruption : restauration du contexte précédent et suite de l'exécution.

Schéma de fonctionnement d'une interruption

mono-tâche



Interruption - Utilité

Le mécanisme d'interruption est utilisé intensivement :

- par les divers périphériques, afin d'alerter le système qu'une entrée-sortie vient d'être effectuée ;
- par le système d'exploitation lui-même ; c'est la base de l'allocation de l'UC aux processus. Lors de la gestion d'une interruption, le système en *profite* pour déterminer le processus qui obtiendra l'UC. Les concepteurs du système ont écrit les gérants des interruptions dans ce but.

On en déduit que les systèmes multi-tâches vont détourner la dernière étape du déroulement de la gestion d'interruptions décrit précédemment.

Interruption - Exemple de priorités

Un exemple dans l'ordre décroissant de priorité

- 1 horloge
- 2 contrôleur disque
- 3 contrôleur de réseau
- 4 contrôleur voies asynchrones
- 5 autres contrôleurs...

Exemples d'interruptions :

- le contrôleur disque génère une interruption lorsqu'une entrée-sortie vient d'être effectuée,
- le clavier génère une interruption lorsque l'utilisateur a fini de saisir une donnée,

Interruptions - Priorités

- plusieurs niveaux de priorité des interruptions ;
- un gérant d'interruption ne peut être interrompu que par une IT de plus haut niveau ;
- une interruption de même niveau est masquée jusqu'au retour du gérant ;
- ce mécanisme doit être assuré par le matériel : par exemple, `IRET` permet de retourner d'un gérant donc de récupérer le contexte de l'interrompu !

Exceptions détectées et programmées

Une **exception** génère un comportement semblable à celui des interruptions (gérant d'exception) mais elle est **synchrone** ; 2 types

d'exception :

- Exception détectée par le processeur : division par 0, overflow, défaut de page, ... appelé aussi *déroutement*, *trappe* ;
- Exception programmée : elle permet d'implémenter un appel système grâce à une instruction privilégiée (*interruption logicielle*) ;

Horloge - Timer

- Le **temps partagé** permet aux processus de se partager l'UC. Ce mécanisme est réalisé grâce aux interruptions Horloge qui ont lieu périodiquement à chaque **quantum de temps** ;
- En quelque sorte, une minuterie qui est enclenchée au début de chaque processus et qui se déclenche toutes les $\approx 10ms$;
- Le gérant d'interruption de l'horloge va alors appeler l'**ordonnanceur** (*scheduler*) qui est chargé d'élire le prochain processus !
- Les autres gérants d'interruption (E/S disque, ...) font de même !

Mode d'exécution

2 modes d'exécution *alternatifs* d'un même processus :

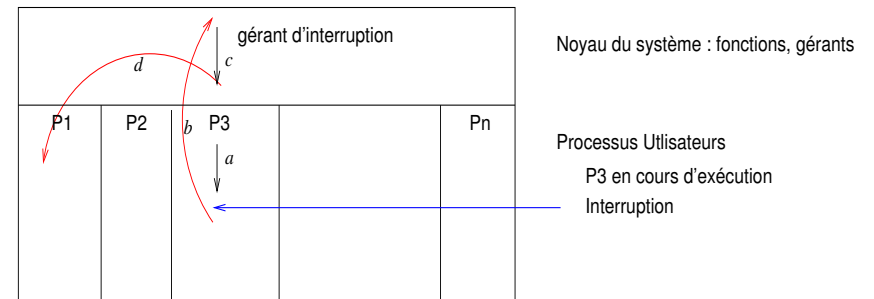
- mode utilisateur** : mode normal dans lequel les instructions machines du programme binaire sont exécutées ;
- mode noyau** : mode système ou privilégié où certaines instructions *sensibles* peuvent être exécutées ;

Exemples d'instructions privilégiées : masquer les interruptions, manipuler l'horloge ou les tables systèmes, sortir de son espace de mémoire, ...

Raisons du passage en mode noyau :

- Exception (détectée (déroutement), ou programmée (appel système)) ;
- Interruption ;

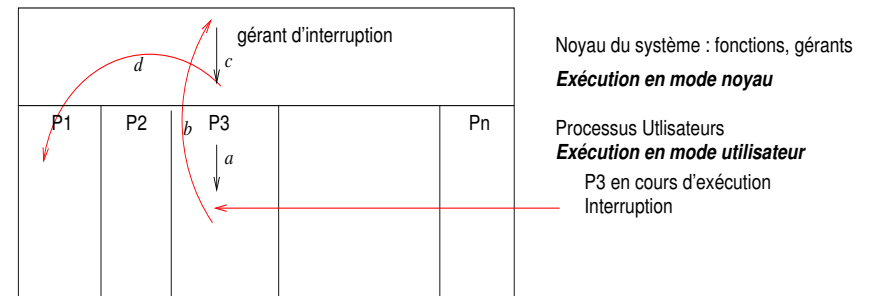
Ordonnancement des processus



Changement de mode d'exécution

Il effectue simultanément deux opérations :

- bascule un bit dans le processeur (bit du mode d'exécution) ;
- exécute le gérant d'interruption ou d'exception ;



Plan

1 Introduction

- Présentation du cours
- Composantes d'un S.E.
- Programme, processus, et contexte
- Prérequis matériels
- **Fonctionnement de la Pile (stack)**
- Noyau et appels systèmes

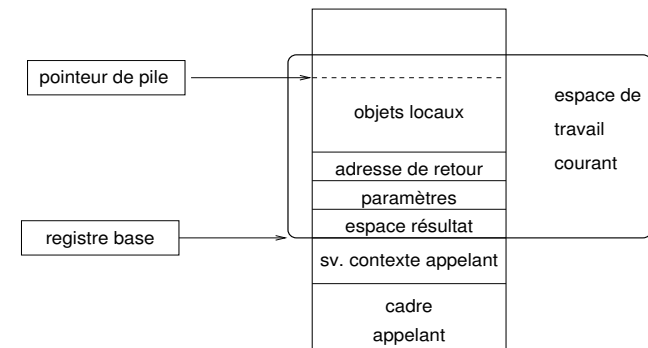
Fonctionnement de la pile

- à chaque occurrence de fonction C, correspond un **cadre** (frame) contenant :
 - résultat éventuel (void)
 - paramètres passés par l'appelant
 - adresse de retour à l'appelant empilé par CALL
 - valeur du registre BP de l'appelant
 - variables locales de cette occurrence de fonction
- ces différentes données sont accédées via le registre de base de pile BP, et le registre de sommet de pile SP
- au fur et à mesure des appels emboîtés, les cadres s'empilent
- au fur et à mesure des retours, les cadres sont dépilés

Petite histoire de la programmation

- programme : séquence d'instructions contenant des sauts conditionnels ou non (JMP, JC, ...);
- invention du registre de retour et des routines : limitation à un niveau d'appel : 1 main et des routines;
- invention de la pile et de la programmation **procédurale** (CALL et RET)!
- programmation évoluée : fonctions, passage de paramètres, variables locales, ...
- programmation par objets, exceptions (traversée de la pile) ...

Espace de travail d'une fonction ou cadre



- l'appelant empile résultat et paramètres, puis réalise le CALL
- l'appelé sauve BP, le modifie, puis installe les var locales
- pendant l'exécution, l'appelé accède aux param et var locales par adressage indexé (BP+x) ou (BP-x)
- avant de RET, l'appelé nettoie les var locales et restaure BP
- après le CALL, l'appelant dépile les paramètres et récupère le résultat

Exemple : fact.c

```
#include <stdio.h> /* printf */
#include <stdlib.h> /* atoi */
int fact(int n){ int res=1;
    if (n<=1) return res;
    else {
        res=n*fact(n-1);
        return res;
    }
}
int main(int n, char *argv[], char *env[]){
    if (n!=2){
        fprintf(stderr,"Syntaxe : %s entier !\n", argv[0]);
        return 1;
    }
    int i=atoi(argv[1]);
    printf("factorielle de %s : %d\n",argv[1],fact(i));
    return 0;
}
```

Pile - Remarques et Questions

Deux piles par processus :

- une pile utilisateur active en mode utilisateur ;
- une pile système active en mode noyau. La pile système est vide lorsqu'un processus est en mode utilisateur.

Un appel récursif se déroule comme tout appel emboîté

Cadre : fact 2

type	commentaire
int	résultat du main
int	n nombre d'arguments
char**	argv
char**	env
void*	adrs retour
void*	BP de l'appelant
int	i==2 var locale du main
int	résultat de fact(2)==2 au retour
int	n==2 paramètre de fact(2)
void*	adrs retour au main
void*	BP du main
int	res==1 (puis 2) var locale de fact(2)
int	résultat de fact(1)==1 au retour
int	n==1 paramètre de fact(1)
void*	adrs retour à fact(2)
void*	BP de fact(2)
int	res==1 var locale de fact(1)

TABLE – 3 cadres de travail : main, fact(2), fact(1)

Plan

1 Introduction

- Présentation du cours
- Composantes d'un S.E.
- Programme, processus, et contexte
- Prérequis matériels
- Fonctionnement de la Pile (stack)
- Noyau et appels systèmes

Noyau

Noyau d'un système d'exploitation : les fonctions du système d'exploitation qui **résident** en mémoire centrale dès le lancement du système jusqu'à son arrêt.

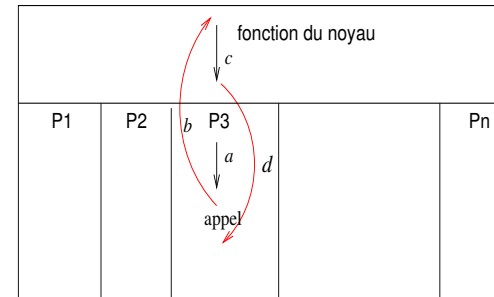
Certaines parties doivent obligatoirement rester en mémoire, sous peine de casse assurée ; d'autres peuvent temporairement être mises à l'écart, sur disque, puis ramenées en mémoire si nécessaire.

Exemples : la gestion de la mémoire centrale ne peut être délocalisée sur disque, sinon, on ne saurait ni comment lui réallouer de l'espace, ni où la réinstaller... Par contre, la gestion des files d'attente d'impression peut être absente temporairement de la mémoire.

Discussion noyau modulaire ou noyau monolithique ?

Appel système

Un *appel système* est la demande d'exécution d'une fonction du noyau faite par un processus quelconque. On parle aussi d'*appel noyau*.



Noyau du système : fonctions, gérants
Exécution en mode noyau

Processus Utilisateurs
Exécution en mode utilisateur

Appels système sous Unix

Exemples :

- générer un processus (*fork()*) ou le terminer (*exit()*) ;
- créer des répertoires, lire, écrire dans des fichiers (*mkdir()*, *read()*, *write()*),
- demander de l'espace mémoire (*brk()*, *sbrk()*) ...

Noter les parenthèses dans tous les exemples afin de ne pas confondre **commande Unix** et **fonction du noyau**.

Les commandes du système, une grande majorité des programmes, système ou utilisateurs, sont construits à partir des appels système.

Exemples : la commande *mkdir* utilise l'appel système *mkdir()*, les fonctions d'entrées-sorties connues dans les langages évolués font appel à *read()*, *write()*, les demandes d'allocation de mémoire (*new*, *malloc()*) font appel à *brk()*, *sbrk()*.

Appels système et manuel

Le manuel Unix regroupe l'ensemble des appels noyau dans le volume 2 du manuel.

Lorsqu'on les visualise, on les reconnaît au chiffre 2 inscrit entre parenthèses après le nom de l'appel, alors que les commandes habituelles se reconnaissent au chiffre 1.

Exemple : *mkdir(1)*, *touch(1)*, *mkdir(2)*, *open(2)*.

volume 3 : fonctions de bibliothèque

Plan

2 Représentation de l'information

- Généralités
- Nombres
- Caractères
- Structures de données

★

Poids fort, unités

La longueur des mots étant 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.

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)

Introduction

Les circuits mémoires et de calcul (électroniques et magnétiques) permettent de stocker des données sous forme **binaire**.

bit abréviation de binary digit, le bit constitue la plus petite unité d'information et vaut soit 0, soit 1.

mot séquence de bits numérotés de la façon suivante :

```

bn-1 bn-2 ... b2 b1 b0
1    0    ... 0  1  1

```

quartet $n = 4$, **octets** $n = 8$ (*byte*), **mots** de n bits (*word*).

Plan

2 Représentation de l'information

- Généralités
- Nombres
- Caractères
- Structures de données

Représentation des entiers positifs

Représentation en base 2 Un mot de n bits permet de représenter 2^n configurations différentes. Ces 2^n configurations sont associées aux entiers positifs compris dans l'intervalle $[0, 2^n - 1]$ de la façon suivante :

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

Exemples

00...0 représente 0
 0000 0111 représente 7 (4+2+1)
 0110 0000 représente 96 (64+32)
 1111 1110 représente 254 (128+64+32+16+8+4+2)
 0000 0001 0000 0001 représente 257 (256+1)

100...00 représente 2^{n-1}

111...11 représente $2^n - 1$

Par la suite, cette convention sera notée Représentation Binaire Non Signée (RBNS).

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	1	1	1		
1	1	1	0	0	1	0	1		0xE5
+	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)

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 !

Le complément à 2 (C2) (suite)

Avantage fondamental :

- l'addition binaire fonctionne correctement !
 $-3+3=0$: 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

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

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 !

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

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 $0,375*2=0,75*2=1,5$; $0,5*2=1,0$ soit 0,011
 $0,23*2=0,46*2=0,92*2=1,84*2=1,68*2=1,36*2=0,72*2=1,44*2=0,88$...
 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 simple précision

4 octets ordonnés : signe, exposant, mantisse.

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

Exemples 33,0 = +100001,0 = +1,000012⁵ représenté par : 0 1000 0100 0000 100... c'est-à-dire : 0x 42 04 00 00
-5,25 = -101,01 = -1,01012² 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 qcq ;
- 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

2 Représentation de l'information

- Généralités
- Nombres
- Caractères
- Structures de données

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) ;

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	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	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

Plan

2 Représentation de l'information

- Généralités
- Nombres
- Caractères
- Structures de données

Structuration

Les structures de données disponibles en MC utilisent 2 principes :

- la contiguïté (tableau, struct) ;
- le chaînage par les pointeurs (liste, arbre, graphe) ;

Les pointeurs peuvent pointer sur des données statiques, dynamiques (malloc) ou locales (pile). Attention, les Systèmes d'exploitation utilisent principalement la contiguïté (tables systèmes) pour son efficacité.

Exemples

- `int global; ... int* pglobal=&global;`
- `int* pi=(int*)malloc(sizeof(int));`
- `{int i=5; int *pi=&i;};`

Struct et Union

Syntaxiquement semblable, ces 2 outils de structuration permettent :

- soit d'agglomérer des champs de différents types ;
- soit de représenter un parmi différents types ;

Exemple

```
struct {
    char *nom;           // nom du fichier
    char type;           // soit 'f', soit 'd'
    union {
        char **liste;    // ls
        char *contenu;   // cat
    } choix ;
} x ;                   // une variable
```

Si x est un fichier (x.type=='f'), x.choix.contenu contiendra une chaîne décrivant son contenu, sinon si f est un répertoire (x.type=='d'), x.choix.liste contiendra la liste du répertoire.

Tableaux C

Tableau SD homogène permettant l'accès indexé grâce à la contiguïté des cases et à leur taille identique.

Exemples

- `char *s="toto";` chaîne de 4 caractères stockés sur 5 octets (caractère nul '\0' terminateur) ;
- `char *s="é";` chaîne de 1 caractère stocké sur 2 (ISO Latin-1) ou 3 (utf-8) octets ;
- `int tab[]={1,2,3};` tableau de 3 entiers ;
- `int *t=(int *)malloc(10*sizeof(int));` tableau dynamique de 10 entiers non initialisés ;
- `t[0]=tab[2];` pointeur \approx tableau
- Se souvenir de la taille d'un tableau (argc, argv) ou mettre une balise à la fin (env) ;

Plan

3 Développement d'applications en C

- Généralités
 - La compilation
 - L'édition de liens
 - Make, le constructeur de projet
 - Le débogueur gdb

Introduction et rappels I

- langage compilé (\neq interprété) ;
- variables et fonctions peuvent être déclarées plusieurs fois mais définies UNE SEULE FOIS ;
- l'initialisation d'une variable n'est possible qu'à sa définition ;
- les fichiers d'en-têtes ne doivent contenir que des déclarations ;
- une variable définie dans une fonction (dite **locale** à cette fonction) :
 - n'est pas initialisée par défaut !
 - est située dans le segment de pile ;
 - a une durée de vie égale au bloc dans lequel elle est définie ;
 - a une portée limitée à son bloc (visible dans son bloc uniquement) ;
- une variable **globale** (définie à l'extérieur de toute fonction) est :
 - initialisée par défaut à 0,
 - dans le segment de données statiques,

Introduction et rappels II

- a une durée de vie égale au processus,
- a une portée **globale** (visible depuis toute fonction) ;
- une variable **locale static** (`{static int i=5; ...}`) :
 - doit obligatoirement être initialisée !
 - est située dans le segment de données statiques ;
 - a une durée de vie égale au processus ;
 - a une portée limitée à son bloc (visible dans son bloc uniquement) ;
 - conserve sa dernière valeur entre deux appels de la fonction où elle est définie ;
- un **objet dynamique** n'est pas une variable : il est référencé par un pointeur initialisé lors de la création de l'objet : (`{int *p=malloc(sizeof(int)); ...}`). L'objet :
 - n'est pas initialisé (sauf si `calloc`) !
 - est située dans le segment de données dynamiques ;
 - reste accessible via un pointeur tant qu'il n'est pas désalloué (`free(p);`) ;

Déclaration, définition, initialisation

	Déclaration	Définition
variable	<code>extern int g;</code>	<code>int g;</code>
fonction	<code>char* f(int i);</code>	<code>char* f(int i){return atoi(i);}</code>

TABLE – Déclaration et définition

- L'initialisation d'une variable (`int i=5;`) est **recommandée** dans tous les cas !
- Les variables globales ne sont pas recommandées ni dans le paradigme de programmation fonctionnelle ni dans celui de la programmation par objets.

Plan

3 Développement d'applications en C

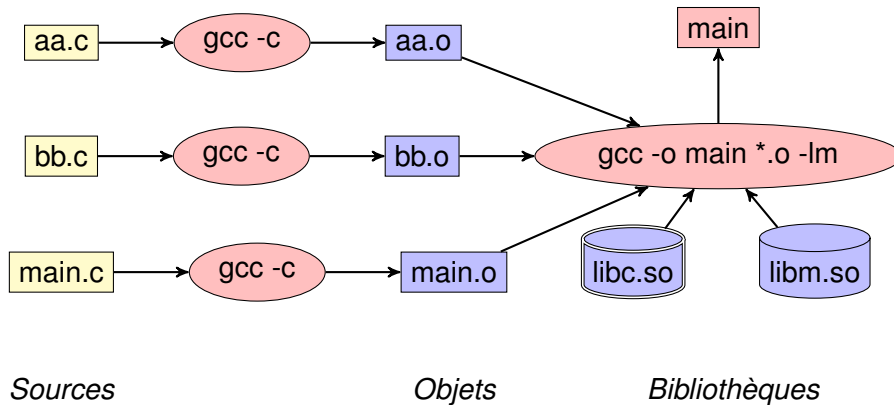
- Généralités
- La compilation
- L'édition de liens
- Make, le constructeur de projet
- Le débogueur gdb

Processus de compilation

Compilations séparées

Édition de liens

Binaire exécutable



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 !

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_;`

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'à 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 retour (1)
jmp L1 ; aller en fin commune de la proc
L2:
```

Le Langage d'assemblage III

```
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 fact)
movl %eax, -4(%ebp) ;
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
; +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

3 Développement d'applications en C

- Généralités
- La compilation
- L'édition de liens
- Make, le constructeur de projet
- Le débogueur gdb

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 (0x00007f191b872000)
/lib64/ld-linux-x86-64.so.2 (0x00007f191bc66000)
```

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 ...
```

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
...
```

Plan

3 Développement d'applications en C

- Généralités
- La compilation
- L'édition de liens
- **Make, le constructeur de projet**
- Le débogueur gdb

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 .

Introduction

La commande make présente les caractéristiques suivantes :

- Gestion de projets multi-fichiers
- Maintenance des **dépendances** inter-fichiers dans un fichier de fabrication : le **makefile**
- Reconstruction optimale du projet : les fichiers non modifiés depuis la dernière construction ne sont pas recompilés
- Couplage possible avec un gestionnaire de versions
- Possibilités de programmer la construction en Bourne shell, y compris récursivement

Vocabulaire

makefile fichier texte décrivant la façon de construire le projet et les objets dont il dépend : liste d'entrées séparées par des lignes vides ;

entrée du makefile une ligne de dépendance suivie de lignes de commandes bash, **précédées par une tabulation**, à exécuter lorsque cette dépendance n'est plus à jour :

cible : comp1 comp2 ...

<Tab> cmd1

...

<Tab> cmdn

<ligne vide>

dépendance indique les fichiers composants dont dépend le fichier cible.

cible (*target*) fichier devant être reconstruit lorsque l'un de ses composants est plus récent.

Vocabulaire - suite

- une cible est**
- ❶ un fichier que l'on veut construire : **make monprog**
 - ❷ un objet situé à gauche d'une dépendance du makefile ;
 - ❸ attention, un même fichier peut-être cible d'une entrée et composant dans une autre ;

composant fichier dont dépend la cible ; ce composant peut lui-même dépendre d'autres objets ... Dans le makefile, il est situé dans la partie droite d'une dépendance

Fichiers sources

Soit les 3 fichiers suivants :

Fichier hello2.c

```
#include <stdio.h>
int g;
hello() {
    for (g=1;g<3;g++) printf("hello\n");
}
```

Fichier princ.c

```
#include "hello2.h"
main() {hello();}
```

Fichier hello2.h

```
hello();
```

Le fichier makefile

```
monprog : hello2.o princ.o
    @echo debut edition de liens : princ et hello2
    gcc -o monprog hello2.o princ.o
    @echo fin edition de liens
```

```
princ.o : princ.c hello2.h
    @echo debut compilation princ
    gcc -c princ.c
    @echo fin compilation
```

```
hello2.o : hello2.c hello2.h
    @echo "debut compilation hello2"
    gcc -c hello2.c
    @echo fin compilation
```

Pour construire le projet : **make monprog**

Règles I

- la construction d'une cible est réursive :
 - un **make** de chacun de ses composants est lancé ...
 - puis s'il existe un composant ayant une Heure de Dernière Modification (HDM) supérieure à la sienne, on exécute séquentiellement les commandes de l'entrée ;
- construction de la cible du makefile situé dans le **répertoire courant**
- chaque règle est précédée d'un caractère de tabulation `\t` et non d'espaces multiples. Attention : couper-coller transforme les tabulations en espaces
- commentaires précédés par `#` et finis par `\n`
- une commande précédée de `@` n'est pas affichée lors de son exécution

Définition de macro

Une macro variable est toujours définie au début du makefile et est évaluée avant la vérification de la cible. Les macros permettent de faire varier les constructions (par exemple le compilateur) en ne modifiant que leurs définitions.

Exemple de définitions de macros variables

```
CC=gcc
CFLAGS= -g
CXX=g++
CXXFLAGS= -g
LEX=flex
```

Exemple d'utilisation de macros dans une entrée

```
hello2.o: hello2.c hello2.h
    $(CC) $(CFLAGS) -c hello2.c
```

Règles II

- les métacaractères `*`, `?`, `[]` sont utilisables pour les noms de fichiers dans la dépendance
- chaque ligne de commande est interprétée par un nouveau processus bash : `\t cd rep ; ls` est différent de `\t cd rep \n \t ls`
- une ligne logique peut dépasser une ligne physique : `\<Entrée>`
- si la dépendance n'a aucun composant ou si la cible n'existe pas en tant que fichier du répertoire courant, alors la construction aura toujours lieu (equiv. script bash)
- la commande **make** sans cible, construit la cible de la première entrée du makefile

macros prédéfinies

Les variables d'environnement peuvent être utilisées comme macros. En cas de conflit avec des macros du makefile, ces dernières ont priorité sauf si le make est exécuté avec l'option `-e` (comme environnement). On peut aussi définir des valeurs de macros lors de l'appel : **make monprog CC=cc**. Un certain nombre de macros par défaut existent : `CC`, `CFLAGS`, ... Certaines macros joker varient selon le contexte :

- `$(@)` nom de la cible courante (hello2.o)
- `$(*)` partie préfixe de la cible courante (hello2)
- `$(<)` nom du premier composant à l'origine de la reconstruction (hello2.c)
- `$(?)` liste de tous les composants à l'origine de la reconstruction
- `$(^)` liste de tous les composants

Règles de suffixe

Ce sont des règles **génériques** définissant la dépendance de fichiers suivant leur suffixe (extension .o ou .c ou ...).

Exemple d'entrée :

```
#regle de suffixe
.c.o:
    @echo debut compil $<
    $(CC) -c $<
    @echo fin compil de $<
```

Cette règle exprime que si un source C a été modifié depuis sa dernière compilation en fichier objet, il est nécessaire de recompiler le source. Cette règle générique ne s'applique que dans le cas où une règle explicite n'existe pas pour le fichier objet qui est la cible.

Ajouter des suffixes

Attention, les règles de suffixes ne fonctionnent que pour les suffixes connus par le make. On peut ajouter des suffixes grâce à une entrée prédéfinie :

```
.SUFFIXES:.cc .h .tex
```

On vient de rajouter les suffixes utiles pour les fichiers en-têtes, C++ et \LaTeX . Remarquons qu'un certain nombre de règles de suffixe par défaut existent.

Pour visualiser les valeurs par défaut des macros, règles de suffixes définies dans la commande make : `make -p`

Règles de suffixe monadique

Une règle de suffixe ne comportant qu'un seul suffixe indique que le second est vide. Par exemple, la règle suivante est utile pour les petits programmes C contenus dans un seul fichier :

```
#regle de suffixe
.c:
    @echo debut compilation et édition de liens de $<
    $(CC) -o $* $<
    @echo fin compilation complète de l'exécutable $*
```

Divers

- priorités des dépendances : règle explicite définie par un “:”, règles explicites définies par des “: :”, règles de suffixe du makefile, règles de suffixe par défaut
- le séparateur ‘: :’ permet d'utiliser plusieurs entrées pour une même cible
- plusieurs cibles peuvent apparaître dans une dépendance à gauche du séparateur (‘:’ ou ‘: :’)
- on peut construire des bibliothèques avec une syntaxe particulière d'entrée
- attention à la programmation en **Bourne shell** !
- on peut appeler récursivement make, puisque c'est une commande shell
- on peut inclure des fichiers dans un makefile : `include ../fic`
- regarder le manuel en ligne pour plus de précisions : `info make`

Plan

3 Développement d'applications en C

- Généralités
- La compilation
- L'édition de liens
- Make, le constructeur de projet
- Le débogueur gdb

GDB II

next exécute la prochaine instruction dans la fonction courante (pas à pas)

print *expression* affiche la valeur courante de l'expression

print *var=expression* affecte l'expression à *var*

list liste une dizaine de lignes du source courant

file *ficexec* pour déboguer un nouvel exécutable

watch *expression* pour s'arrêter lorsque l'expression change de valeur

step exécute la prochaine instruction même si on doit descendre d'un niveau dans les appels de fonctions

up permet de remonter d'un cran dans les appels imbriqués

GDB I

Pour lancer le débogueur, il faut au préalable avoir compilé le programme avec l'option `-g`. Taper ensuite la commande : **`gdb monprog`**. Lors d'une session de débogage, un grand nombre de commandes sont possibles. Celles-ci peuvent être abrégées par leurs premières lettres :

`bt` liste les appels de fonctions emboîtées (pile) *BackTrace*

`help` aide en ligne de gdb

`run` exécute le programme

`quit` pour quitter gdb

`break nomfonction` pose un point d'arrêt sur cette fonction

`info break` liste les points d'arrêt

`cont` continue après un arrêt

Un exemple d'utilisation de gdb : Mastermind I

```
$ gcc -Wall -std=c99 -g -c mm.c          # option -g
$ gcc -Wall -std=c99 -g -c main.c
$ gcc -Wall -std=c99 -g -o mm main.o mm.o
$ gdb mm                                # session db
(gdb) break main                        # point d'arrêt
Breakpoint 1 at 0x100000953: file main.c, line 8.
(gdb) list                              # listing
1 #include "mm.h"
...
7 int main(){
8     mm j=mm_creer();
9     char saisie[1024];
10    int res, CONTINUER=1;
(gdb) run                                # lancement
```

Un exemple d'utilisation de gdb : Mastermind II

```
Starting program: /Users/.../Mastermind/mm
Breakpoint 1, main () at main.c:8
8   mm j=mm_creer();
(gdb) n                                # next
10   int res, CONTINUER=1;
(gdb) p j->secret                       # print le mot secret
$1 = "8584"
...
15   res=mm_test(j,saisie);
(gdb) step                             # rentrer dans mm_test
mm_test (jeu=0x10..., essai=0x7f... "8524") at mm.c:26
26   jeu->nbessais++;
(gdb) p jeu->nbessais                   # print compteur
$2 = 0
(gdb) n
```

Un exemple d'utilisation de gdb : Mastermind III

```
27   if(strlen(essai)!=TAILLE)
(gdb) p jeu->nbessais                   # le compteur a changé
$3 = 1
(gdb) bt                                # backtrace
#0  mm_test (jeu=0x10..., essai=0x7f... "2568") at mm.c:27
#1  0x00000001000009ca in main () at main.c:15
(gdb) quit                             # fin de session
```

Un autre exemple avec un programme qui plante I

```
$ gdb segfault                          # prog générant SEGV
(gdb) run
Starting program: .../Td01/segfault

Program received signal SIGSEGV, Segmentation fault.
0x0000000100000f39 in g () at segfault.c:3
3 void g(void){printf("%c\n",s[0]);} // signal SIGSEGV
(gdb) bt                                # par qui g a été appelé
#0  0x0000000100000f39 in g () at segfault.c:3
#1  0x0000000100000f59 in f () at segfault.c:4
#2  0x0000000100000f69 in main () at segfault.c:5
(gdb) b 3                               # break à la ligne 3
Breakpoint 1 at 0x100000f36: file segfault.c, line 3.
(gdb) run                                # relancer
```

Un autre exemple avec un programme qui plante II

```
Starting program: /Users/.../segfault

Breakpoint 1, g () at segfault.c:3
3 void g(void){printf("%c\n",s[0]);} // génère SIGSEGV
(gdb) p s[0]                            # affiche le car
Cannot access memory at address 0x0 # ON A L'INFO !
(gdb) s                                 # pour voir ;
Program received signal SIGSEGV, Segmentation fault.
0x0000000100000f39 in g () at segfault.c:3
```

Plan

4 Gestion des Entrées-Sorties

- Présentation
- Types de Périphériques
- Quelques Détails
- Fichier séquentiel et flot sous Unix
- Tables internes de gestion des fichiers
- Fonctions de bib. versus appels systèmes

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.

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.

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 - suite

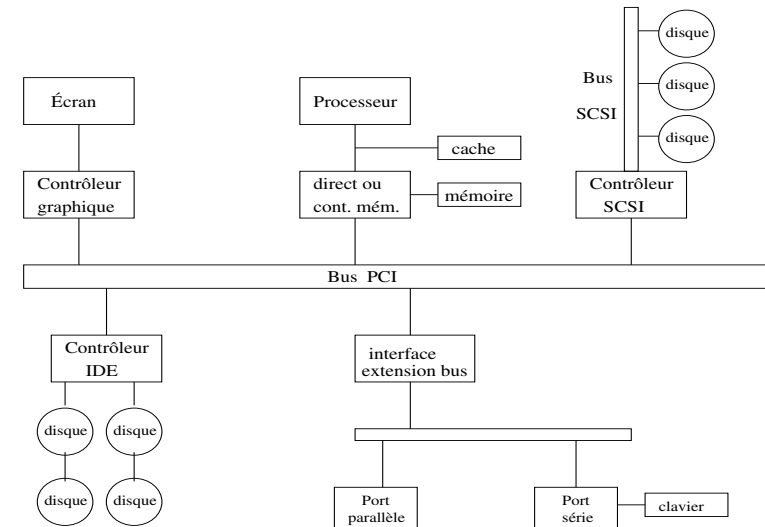
- Le gérant d'interruption réveille le processus demandeur qui exécute le pilote.
- Le pilote recopie une partie du tampon système vers l'espace utilisateur (adresse de la donnée).
- Puis, le pilote retourne un résultat à l'appel noyau permettant au processus de repasser en mode utilisateur.

Plan

4 Gestion des Entrées-Sorties

- Présentation
- Types de Périphériques
- Quelques Détails
- Fichier séquentiel et flot sous Unix
- Tables internes de gestion des fichiers
- Fonctions de bib. versus appels systèmes

Exemple - fin : Structure Classique



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.

Plan

4 Gestion des Entrées-Sorties

- Présentation
- Types de Périphériques
- Quelques Détails
- Fichier séquentiel et flot sous Unix
- Tables internes de gestion des fichiers
- Fonctions de bib. versus appels systèmes

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. . .

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.

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.

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).

Fichier séquentiel (2) et flot(3)

- Un fichier (séquentiel) est 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.

Plan

4 Gestion des Entrées-Sorties

- Présentation
- Types de Périphériques
- Quelques Détails
- Fichier séquentiel et flot sous Unix
- Tables internes de gestion des fichiers
- Fonctions de bib. versus appels systèmes

fichiers et flots ouverts

Tout processus lancé depuis un `bash` démarre avec 3 fichiers et flots 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 flot

Un flot doit être ouvert (open) sur un fichier ou un périphérique 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 flot 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 sur 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; /* tampon de lecture */
    while(1==read(fd,&c,1)){
        nb++;
    }
    close(fd);
    return nb;
}

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);
    return nb;
}
```

Accès direct

L'intérêt de l'**accès séquentiel** est son universalité : quel que soit le périphérique de mémorisation ou de communication, ce mode d'accès est possible.

Son inconvénient est sa lenteur dans la recherche d'une information particulière : il faut en moyenne lire la moitié du fichier pour trouver l'information recherchée.

Pour les fichiers sur disque, l'**accès direct** permet de rechercher un article parmi d'autres grâce à une **expression d'accès** (parfois nommée clé). Par exemple, le numéro d'étudiant (unique), ou le nom (homonymes) d'un étudiant sont des expressions d'accès.

L'appel système `lseek` permet de se déplacer dans un fichier ouvert :

```
off_t lseek(int fildes, off_t offset, int whence);
whence : SEEK_SET | SEEK_CUR | SEEK_END
```

Voir aussi `fseek()` (3)

Plan

4 Gestion des Entrées-Sorties

- Présentation
- Types de Périphériques
- Quelques Détails
- Fichier séquentiel et flot sous Unix
- Tables internes de gestion des fichiers
- Fonctions de bib. versus appels systèmes

Tableau des descripteurs de fichier

- Chaque processus possède sa propre table de descripteurs ;
- un descripteur de fichier est un petit entier positif qui indexe cette table (0,1,2,3,...) ;
- une entrée de la table est un pointeur sur *la table des fichiers ouverts* qui est une table système partagée par tous les processus ;
- chaque appel système `open()` crée une nouvelle entrée dans *la table des fichiers ouverts* puis enregistre un pointeur sur cette entrée dans la table des descripteurs du processus sur la plus petite entrée non occupée (*lowest-numbered unused*) ;
- un nouveau processus lancé depuis le bash récupère une copie de la table des descripteurs du bash ;
- après un `close()`, l'`open()` suivant récupérera le plus petit indice disponible dans la table des descripteurs qui sera souvent celui du fichier qui vient d'être fermé ;

Table des fichiers ouverts

- La table des fichiers ouverts est unique ;
- chaque entrée correspond à l'exécution d'un `open()` ;
- une entrée contient :
 - le déplacement courant dans le fichier (*offset*) qui matérialise la "tête de lecture/écriture" ;
 - le compte de descripteurs de fichiers qui pointe sur cette entrée ;
 - des pointeurs sur les tampons systèmes ;
 - différents indicateurs (*flags*) comme le mode d'ouverture ;

L'appel système `int dup(int oldfd)` permet de dupliquer un descripteur de fichier `oldfd` en un nouveau qui sera un alias. Les deux descripteurs pointent alors sur le même fichier ouvert et partagent donc le déplacement courant ;

Plan

4 Gestion des Entrées-Sorties

- Présentation
- Types de Périphériques
- Quelques Détails
- Fichier séquentiel et flot sous Unix
- Tables internes de gestion des fichiers
- Fonctions de bib. versus appels systèmes

Efficacité de la bibliothèque standard C II

- on choisit un fichier dans .git qui contient 64 Mio de données binaires
- on teste successivement la copie de ce gros fichier vers /dev/null (périphérique poubelle) en utilisant :
 - la version appel système et une taille de 1 ;
 - la version appel système et une taille de 1024 ;
 - la version fon. de bib. et une taille de 1 ;
 - la version fon. de bib. et une taille de 1024 ;
- on observe les résultats édifiants !

Efficacité de la bibliothèque standard C I

Outre sa portabilité et ses capacités plus importantes, l'utilisation des fonctions de la bibliothèque standard est plus efficace en temps ! Pour le tester :

- écriture de 2 programmes de copie de fichiers, l'un utilisant des appels systèmes `moncp3s`, l'autre des fonctions de bibliothèque `moncp3b` ;
- les 2 programmes utilisent 3 arguments `moncp src dst taille`
 - `src` désigne le fichier source de la copie ;
 - `dst` désigne la destination (copie) du source ;
 - `taille` désigne la taille des blocs qu'on lit dans le source puis qu'on écrit dans la destination ;

- on commence par rechercher un gros fichier :

```
$ find ~ -type f -size +10000k -printf "%p %s\n"
```

Efficacité de la bibliothèque standard C III

```
$ time moncp3s /auto_home/...c29568.pack /dev/null 1
real 0m53,301s
user 0m12,640s
sys 0m40,659s
$ time moncp3s /auto_home/...c29568.pack /dev/null 1024
real 0m0,070s
user 0m0,012s
sys 0m0,056s
$ time moncp3b /auto_home/...c29568.pack /dev/null 1
real 0m2,392s
user 0m2,378s
sys 0m0,004s
$ time moncp3b /auto_home/...c29568.pack /dev/null 1024
real 0m0,031s
user 0m0,008s
sys 0m0,021s
```

Efficacité de la bibliothèque standard C IV

- Dans le pire des cas (app. syst. et taille de 1) on attend 53s, tandis que dans le meilleur (fon. bib ; et taille 1024) on attend 0,031s !
- Avec une taille de 1024 octets pour les deux, la version "bibliothèque" est deux fois plus rapide que la version "système" : 0.031 vs 0.070
- En augmentant la taille, on améliore la vitesse d'exécution dans les deux cas : moins de boucle pour le temps en mode utilisateur mais le temps en mode noyau varie moins pour la version bib. au delà d'une taille raisonnable (1024).

Plan

- 5 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

schéma général système

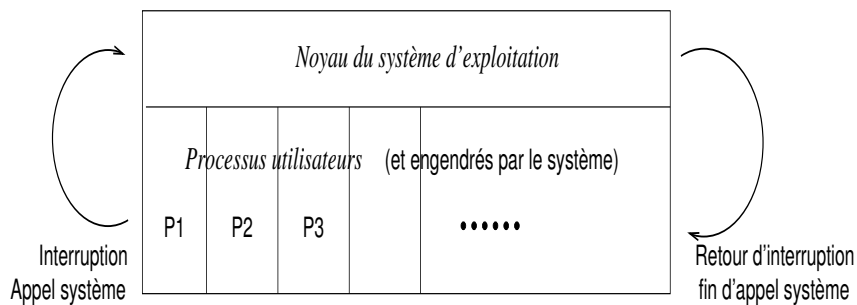


Table des processus

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.

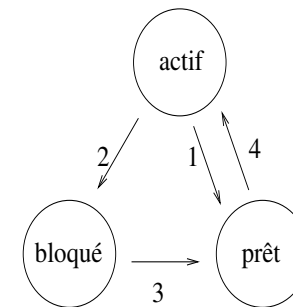
Plan

5 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 :



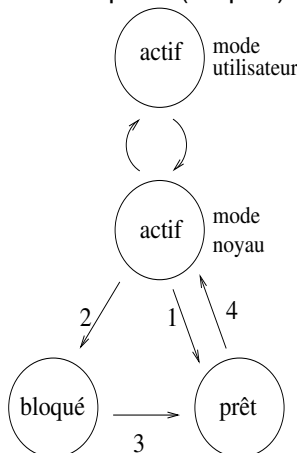
- actif** tient la ressource UC
- prêt** seule la ressource UC lui manque
- bloqué** manque au moins une autre ressource

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

5 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

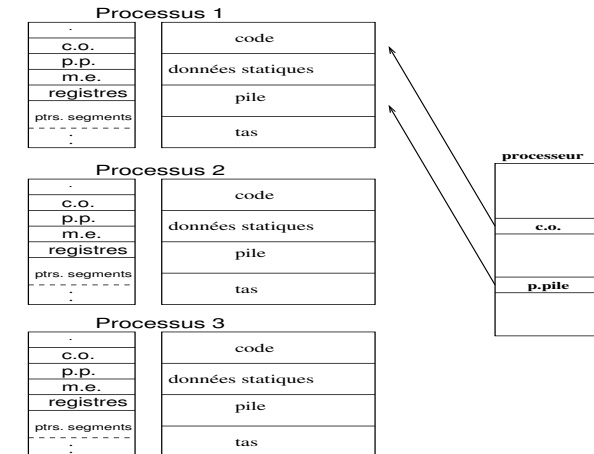
Principe du changement de contexte

- 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*).

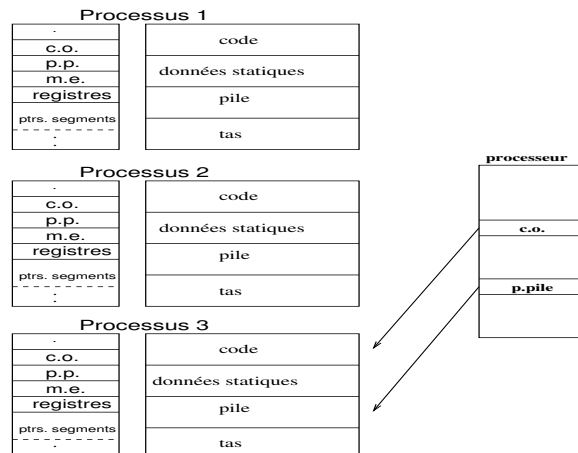
Ce changement se produit lorsque le processus actif passe à l'état bloqué ou prêt et qu'un autre processus devient actif.

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.

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.

Interrogation orale

Questions :

- ① Expliquer pourquoi la fonction noyau appelée n'est pas terminée ;
- ② 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 ?
- ③ Dans quel mode d'exécution se trouvent tous les processus en attente ?

Schéma algorithmique ordonnanceur

tant que *pas de processus élu* **faire**

```

    consulter table processus ;
    choisir 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
  
```

Plan

- 5 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 2

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

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

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 :

- 1 P1 passe en mode privilégié en faisant l'appel système *read()* ;
- 2 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, ...) ;
- 3 l'état de P1 sera passé à *bloqué* et son contexte sauvegardé ;
- 4 enfin, *read* va appeler l'ordonnanceur.

Suite Étape 2

- 6 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 ;
- 7 il y a interruption d'horloge ;
- 8 passage en mode noyau et exécution du gérant d'interruption d'horloge qui passe P2 à l'état prêt ;
- 9 sauvegarde du contexte de P2 par le gérant ;
- 10 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 reinitialisée. Suite du scénario :

- ① P3 est en cours d'exécution ; on suppose que la lecture demandée par P1 est (enfin) prête ; alors,
- ② P3 est interrompu par une interruption disque ;
- ③ 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 ;
- ④ P1 est passé à l'état *prêt* (on dit que P1 est *réveillé*) ;
- ⑤ P3 est **aussi** passé à l'état *prêt* ;
- ⑥ 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)

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 ;

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.

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.

commande ps l

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

Plan

- 5 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 ll

- 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'élu.
- TIME indique la durée passée sur le processeur en mode noyau et (+) en mode utilisateur

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

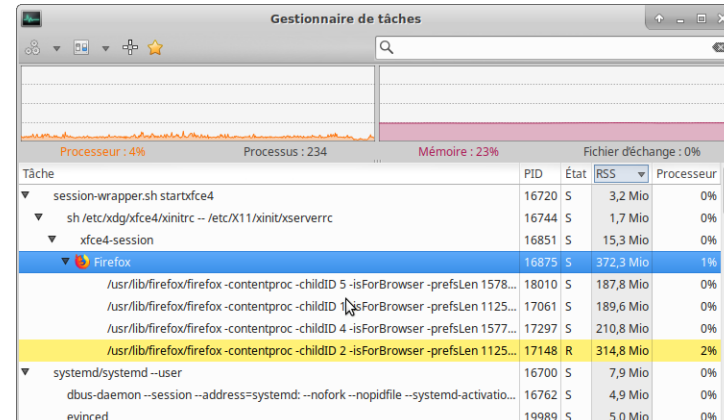
commande ps III

```
$ 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 /net/apps
mmeynard  19450  0.8  3.5 2144256 289652 ?        Sl   10:25   1:04 |  \_ ./thunderbird
mmeynard  16942  0.0  0.1 173336 14784 ?        S    09:52   0:00 \_ /usr/lib/x86_64-linux-c
mmeynard  16947  0.0  0.2 331292 18844 ?        Sl   09:52   0:00 \_ /usr/lib/x86_64-linux-c
mmeynard  16985  0.0  0.3 343600 25772 ?        Sl   09:52   0:00 \_ /usr/lib/x86_64-linux-c
mmeynard  16989  0.0  0.4 692892 34008 ?        Sl   09:52   0:08 \_ /usr/lib/x86_64-linux-c
mmeynard  17011  0.0  0.3 329712 24844 ?        Sl   09:52   0:00 \_ /usr/lib/x86_64-linux-c
mmeynard  17748  0.1  0.5 646612 41264 ?        Sl   09:53   0:10 \_ /usr/bin/xfce4-terminal
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 pus selon le pourcentage d'utilisation du processeur
- des gestionnaires de tâches existent pour chaque distribution qui permettent de visualiser les processus



Plan

- 5 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...

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 ;

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());
}
```

Exécution

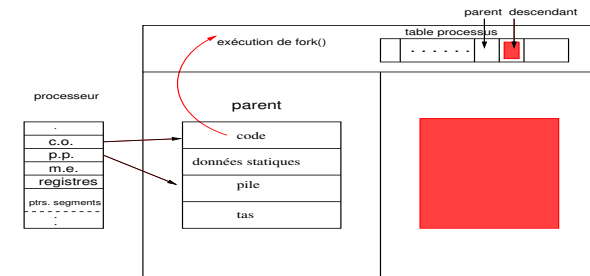
```
Exemple$ gcc -g -Wall -o forksimple forksimple.c
Exemple$ forksimple
du parent : valeur de retour de fork() : 27470
du parent : je suis 27469 de parent 77441
Qui suis-je ? : 27469
du descendant : valeur de retour de fork() : 0
du descendant : je suis 27470 de parent 1
Qui suis-je ? : 27470
```

Questions

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

Déroulement

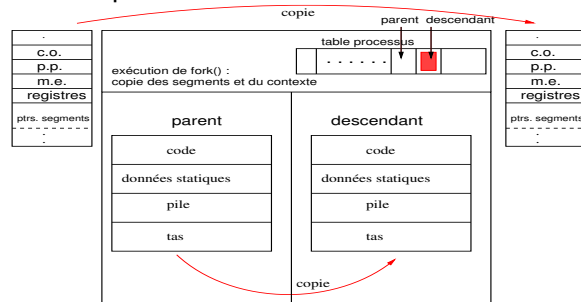
Le processus exécute ce code ; il y a appel noyau : *fork()*.



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

- 1 En reprenant l'exemple précédent, où vont avoir lieu les affichages respectifs ?
- 2 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 ?
- 3 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 ?
- 4 Donner deux exemples dans lesquels on aura un résultat négatif à *fork()*.

Plan

5 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 :

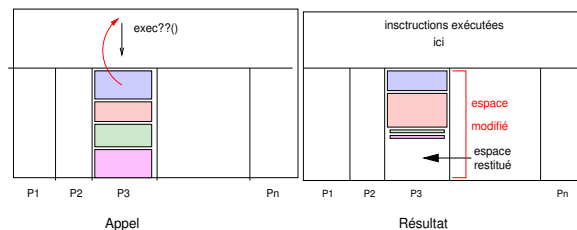
- vérifier l'existence et l'accessibilité (droits) du fichier exécutable ;
- écraser son propre segment de code par le nouvel exécutable ;
- passer éventuellement à ce code des paramètres d'exécution ;
- générer un nouveau segment de données statiques ;
- vider le segment de pile ;
- vider le tas ;
- 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

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) ?

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.

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

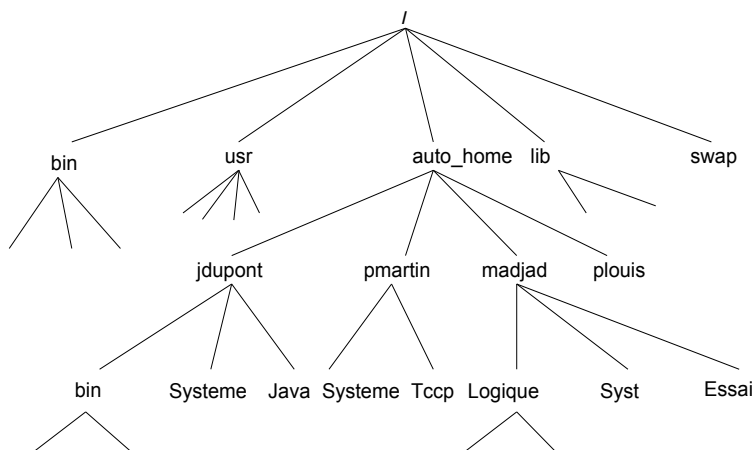
6 Système de Gestion des fichiers

- 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 :

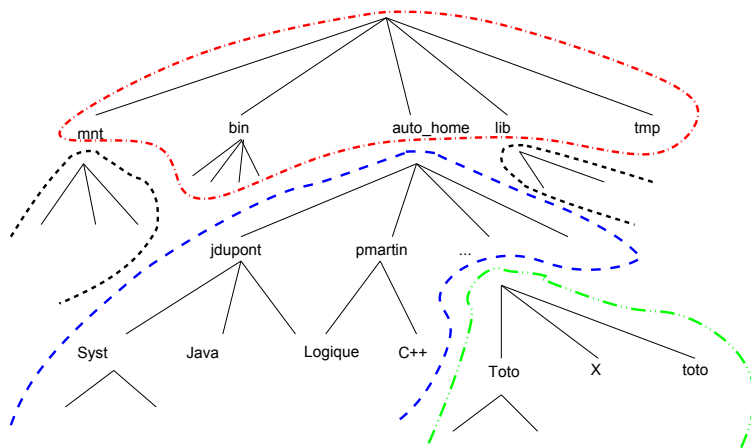
- concurrence des utilisateurs sur un même espace disque
- allocation désallocation des blocs et gestion du chaînage des blocs constituant un fichier
- minimisation de l'espace d'administration des fichiers (table des inode, table des blocs dispo., ...)
- solution générale des OS : une ou plusieurs arborescences 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

6 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%	Blocs de fichiers
données	

- 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

6 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

6 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

6 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

6 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

6 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 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 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 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 filesyst 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

- ⑥ **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 :

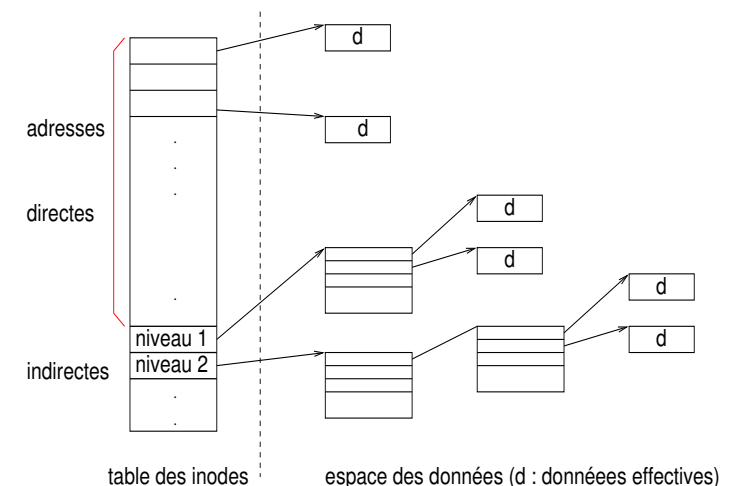
Adressage de tout fichier : un arbre dont les feuilles sont les blocs de données et les nœuds internes des blocs d'adresses.

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).

- 1 Taille maximale atteinte par les blocs directs :
 $10 \text{ blocs} \times 4Ki \text{ octets} = 40Ki \text{ octets}$
- 2 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

- 3 Taille maximale atteinte par le 1^{er} niveau :
 $1 \text{ pointeur} \times 1Ki \text{ adresses} \times 4Ki \text{ octets} = 4Ki^2 \text{ octets} = 4Mi \text{ octets}$.
- 4 Taille maximale atteinte par le 2^{ème} niveau :
 $1 \text{ ptr} \times \underbrace{1Ki \times 1Ki}_{1M \text{ adr}} \text{ adr} \times 4Ki \text{ oct} = 4Ki^3 \text{ oct} = 4Gi \text{ oct}$.
- 5 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

- 1 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 $64Tio$
- 2 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
- 3 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 :

- 4 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

6 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

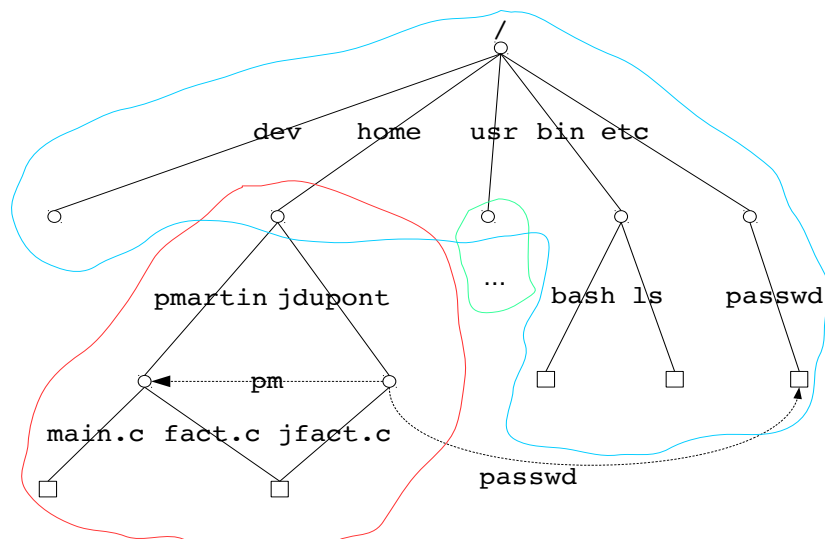
Méthode : un fichier de type nouveau (l) permettant de désigner (référencer, pointer sur) un *chemin* déjà existant dans l'arborescence

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

6 Système de Gestion des fichiers

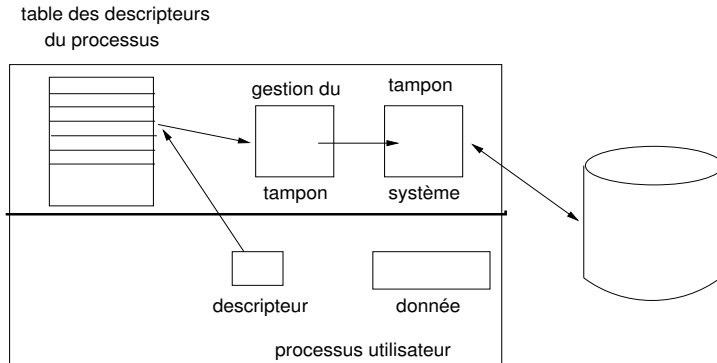
- Introduction
- Systèmes de Fichiers
- Table des inodes
- Droits Unix
- Répertoires
- Liens durs
- Appels système du SGF
- Stockage des données des Fichiers
- Lien Symbolique
- **Traitement des Fichiers Ouverts**
- Cohérence des Partitions
- Retour sur les Droits

Problèmes divers

Problèmes traités dans ce chapitre :

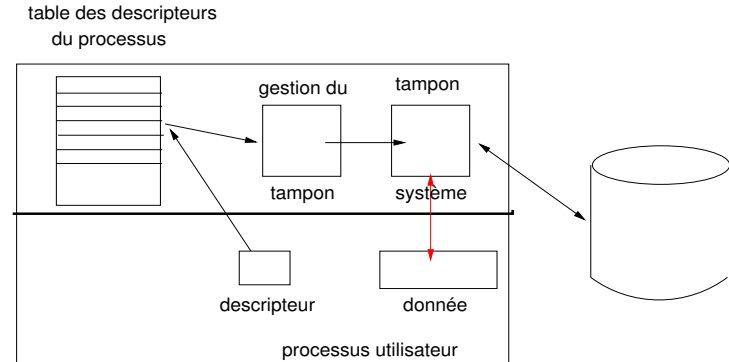
- Comment est-ce que le système gère les fichiers ouverts par les processus ? Que se passe-t-il lorsque plusieurs processus partagent le même fichier ?
- Pourquoi est-il nécessaire de vérifier la cohérence entre le contenu des répertoires et la table des inodes ? Que peut-on vérifier ? Peut-on réparer ?
- 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



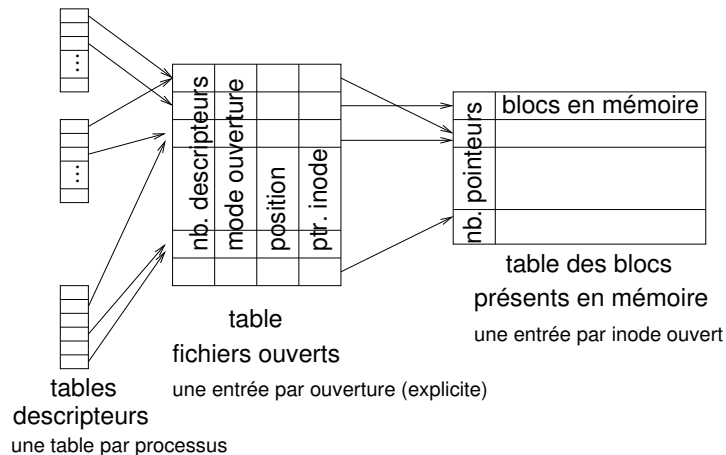
- Le descripteur est un indice vers un élément de la table des descripteurs de ce processus. Il y a une table par processus.
- Le tampon système contient au moins un bloc du fichier ouvert.
- Noter que la gestion de ce bloc dépend du type de périphérique.

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.

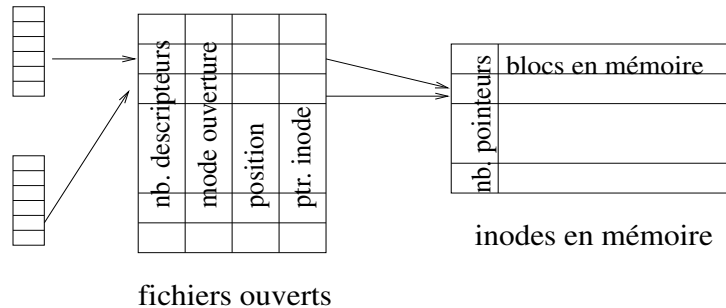
Résumé

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

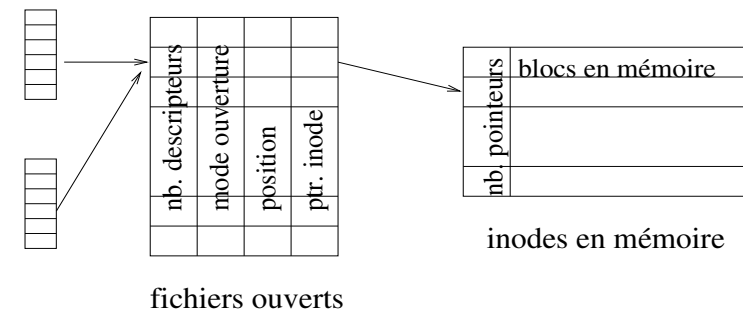
Deux processus ouvrant le même fichier, chacun avec ses propres paramètres, se représentent ainsi :



- 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 *abcdefghijkl*. 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

6 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

6 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

7 Communications basiques entre Processus (signaux et tubes)

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

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.

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.

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

7 Communications basiques entre Processus (signaux et tubes)

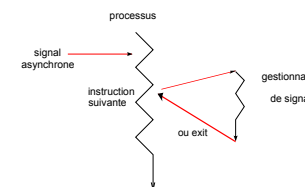
- Les Besoins
- Signaux Unix
- Tubes Simples
- Tubes Nommés

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 p. 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);

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.

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)

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.

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");
}

```

Gestion du signal SIGSEGV

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

Signal \neq 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 \Rightarrow 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 \leftrightarrow gst) reste pérenne !
- Après une trappe, le gestionnaire doit terminer le processus sinon l'instruction ayant provoqué le signal est réexécutée à l'infini !

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. `SIGUSRn` 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

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).

Plan

7 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 syst. `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 ;

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).



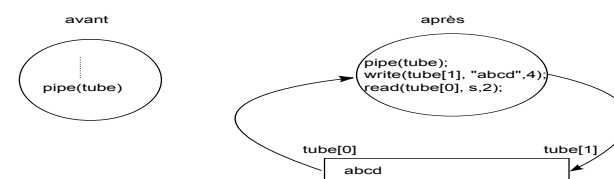
Question : par rapport aux segments mémoire des processus, où est cet espace ?

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

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()`.

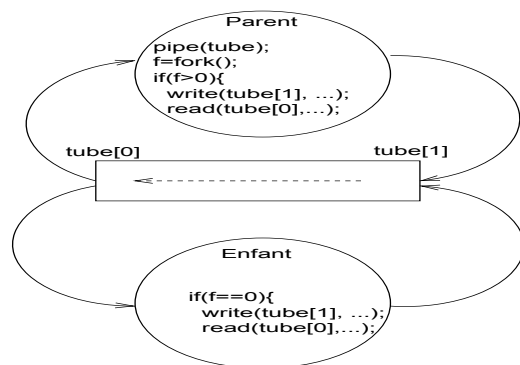


Exemple :

```
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

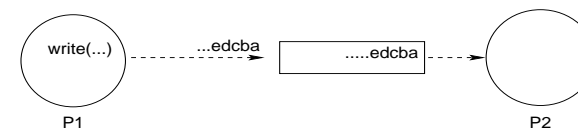
Reste à résoudre le problème du partage du tube par plus d'un processus. L'héritage des descripteurs lors d'un clonage (`fork()`) permet de répondre.



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.

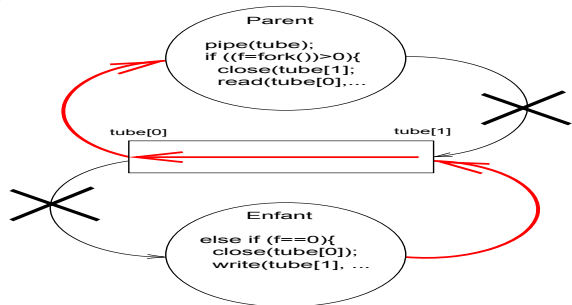
Exemple : Un processus créant un tube et tentant de lire dans ce tube est interbloqué. Dans la mesure où il existe encore un écrivain potentiel (lui-même), le noyau bloque ce processus indéfiniment sur un `read` que personne ne pourra satisfaire !

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 processus tentent de lire le tube vide : ils seront alors bloqués pour l'éternité.

Eviter l'interblocage

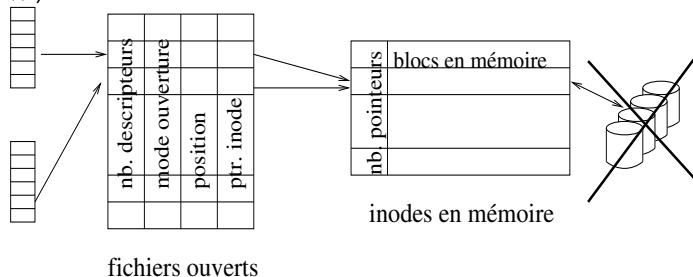
Une bonne règle à respecter concerne la **fermeture au plus tôt des descripteurs non utilisés**. Cela évitera des situations triviales d'interblocage.



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

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

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 !

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.

Plan

7 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.

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 ?

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 - 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 structure fifo en mémoire puis se bloquer ; //attente rdev

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).

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.

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 débloquent 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.

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.

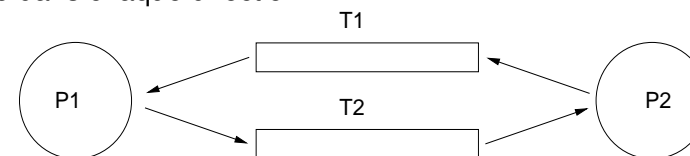
Plan

8 Synchronisation et Communication Entre Processus

- Les Besoins
 - Section Critique
 - Sémaphores
 - Verrou Fatal ou Deadlock

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.

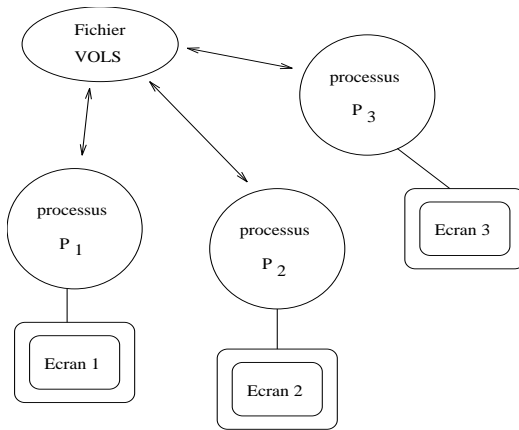
Petit exercice : Un troisième processus peut les sauver.

Objectifs du Chapitre

- Montrer que toutes les applications qui accèdent à des données, plus généralement à des ressources communes, ont besoin de réglementer ces accès ; réglementer veut dire :
 - ne pas laisser plusieurs processus accéder et modifier la même donnée *simultanément*, puisque des incohérences pourraient en résulter ;
 - assurer une certaine équité entre les accédants.
- Ces applications relèvent tant système d'exploitation que des applications des utilisateurs .
- Connaître les primitives que le système d'exploitation fournit afin de réglementer ces accès.

Exemple d'Application Utilisateurs

Réservation de places d'avion (ou train, spectacle etc.).



Variable commune accédée et modifiable par tous : *nombre de places réservées* appelée `nbPlacesRes` pour chaque vol.

Variable commune accédée en lecture seule, *nombre de places total*, appelée `nbPlacesMax` pour chaque vol.

Algorithme Classique et Problème

On suppose que trois processus `P1`, `P2`, `P3` exécutent le même code pour réserver des places. On se restreint à un vol donné dans cet exemple. Chaque processus dispose d'une variable locale `nbPlacesDem` représentant le nombre de places qu'il veut réserver suite à une demande locale. Supposons l'algorithme suivant :

```

si (nbPlacesDem ≤ nbPlacesMax - nbPlacesRes) alors
    nbPlacesRes += nbPlacesDem ;
    afficher ("réservation effectuée pour" nbPlacesDem "place(s)");
sinon
    afficher ("il reste seulement", nbPlacesMax - nbPlacesRes,
            "places");
  
```

Un Déroulement Possible

Supposons la situation globale suivante :

`nbPlacesMax` = 100 ; `nbPlacesRes` = 98 ;

`P1` veut **deux** places et `P2` en veut **une**.

`P1` est actif ; dans sa sa variable locale il a `nbPlacesDem` = 2.

`P1` effectue le test, obtient un résultat positif, puis est interrompu (horloge) de suite après le test :

```

si (nbPlacesDem ≤ nbPlacesMax - nbPlacesRes) alors
    ❌ nbPlacesRes += nbPlacesDem ;
    afficher ("réservation effectuée pour" nbPlacesDem "place(s)");
sinon
    afficher ("il reste seulement", nbPlacesMax - nbPlacesRes,
            "places");
  
```

Déroulement - Suite

On suppose que `P2` devient actif.

Dans sa variable locale il a `nbPlacesDem` = 1.

Il effectue le test, et obtient **aussi** un résultat positif.

Ainsi, `P2` effectue la modification

`nbPlacesRes` = 99, puis affichera *réservation effectuée pour 1 place(s)*.

Lorsque `P1` redeviendra actif, il effectuera la modification

`nbPlacesRes` = 101 puis affichera *réservation effectuée pour 2 places(s)*.

De façon certaine, on peut affirmer que l'algorithme ne fonctionne pas correctement,

- à cause de l'accès avec modification aux données communes,
- ou à cause de l'interruption qui s'est produite à un *mauvais moment*...

Exemples Système

Dans la gestion des processus, on sait que le nombre de processus maximal est fixé. Il faut vérifier lors de chaque création de processus qu'on ne dépasse pas ce nombre maximal.

Exemple d'algorithme avec deux variables communes accédées par le système : `nbProcEnCours` et `nbMaxProc`, autosuggestives.

Un exemple très simple d'algorithme :

```


si nbProcEnCours < nbMaxProc alors
|   nbProcEnCours++ ;
|   dernierAttrib = dernierAttrib + 1 ;
|   retourner (numProc = dernierAttrib) ;
sinon
|   retourner(erreur) ;
  
```

Déroulement Possible

Supposons que deux processus, P1 et P2, soient en cours d'exécution de `fork()`. Rappelons que les appels système sont interruptibles comme toute autre fonction (voir les quelques exceptions dans la gestion des interruptions).

Comme précédemment, il suffit d'envisager l'interruption suivante lors de l'exécution :

```

si nbProcEnCours < nbMaxProc alors
|    nbProcEnCours++ ;
|   dernierAttrib = dernierAttrib + 1 ;
|   retourner (numProc = dernierAttrib) ;
sinon
|   retourner(erreur) ;
  
```

Types de Problèmes

En général, ces problèmes se retrouvent partout où une ressource commune est accédée puis, modifiée en fonction de son contenu courant.

Dans les structures vues jusque là, voici quelques exemples :

- avec les tubes, représentant un problème classique de *producteur consommateur* : gérer le nombre d'éléments disponibles dans tubes lors des entrées-sorties. **Exercice** : pourquoi ? dans quelles circonstances exactement ?
- gestion des impressions dans un système : insertion dans des files d'attente, allocation des périphériques, ...
- allocation de la mémoire, etc, etc.

Question : Comment résoudre ce problème ?

Réponse partielle : Soit le système offre des primitives qui permettent de verrouiller temporairement une partie du code, soit il faut construire ces primitives ...

Plan

8 Synchronisation et Communication Entre Processus

- Les Besoins
- Section Critique
- Sémaphores
- Verrou Fatal ou Deadlock

Section Critique

Environnement : plusieurs processus exécutant un même code et partageant des données communes.

Une **section critique** est une partie du code qui doit être exécutée par un seul processus à la fois.

On dit qu'il y a **exclusion mutuelle** entre ces processus.

Attention : ceci ne veut pas dire que le processus en section critique est ininterrompible, mais seulement que, si un processus P1 est entré dans cette section, aucun autre ne doit pouvoir la commencer avant que P1 n'ait terminé.

Il faut un protocole d'accès (ensemble de règles) que **tous** les processus **doivent** respecter (doivent utiliser) pour exécuter cette section critique.

Tentatives de Solution

On peut vérifier que les solutions simples ne fonctionnent pas.

Exemple : une variable commune *verrou* initialisée à *faux* et utilisée ainsi dans chaque processus :

si (*non verrou*) **alors**

```

    verrou = vrai ;
    SectionCritique ;

```

verrou = faux ;

...toujours pour la même raison que les exemples déjà cités.

Voir la bibliographie pour divers développements parfois longs à ce sujet.

Section Critique - Propriétés

Remarque préalable : Si un processus décide de passer outre le protocole, alors l'ensemble de l'application *peut dysfonctionner*. Le débogage sera d'autant plus difficile que l'on ne peut pas reproduire la même situation ayant provoqué le plantage.

Une solution doit avoir les **propriétés** suivantes :

- **exclusion** : un seul processus en SC ;
- **évolution** : un processus en dehors de sa SC ne peut bloquer d'autres processus ; autrement dit, seuls les processus demandant à entrer en SC participent à la décision ;
- **attente bornée** : pas de famine. Noter qu'ainsi, la demande d'entrer en section critique ne peut être reportée indéfiniment.

supplément : pas de présomption sur le nombre de processeurs.

Besoin d'Opérations Atomiques

Si l'on dispose d'un outil permettant de façon **atomique** (d'atome au sens unité - non interruptible, pas au sens explosif) de tester et verrouiller une ressource (donnée, fichier, ...), alors on peut trouver des solutions.

En d'autres termes, on veut tester **et** modifier **simultanément** un état.

Exemple : Supposons qu'une opération *verrouFichier(nomFichier)* soit disponible et que cette opération soit **atomique**. Son fonctionnement consisterait par exemple, à réaliser l'algorithme :

- si un lien symbolique appelé *lock* sur *nomFichier* existe, alors bloquer le processus demandeur ;
- sinon, créer ce lien symbolique. Dans ce cas, tout futur demandeur sera bloqué.

Alors on pourrait résoudre le problème, sachant qu'une opération atomique *leverVerrou(nomFichier)* devrait de même permettre de supprimer le lien.

Situation Actuelle (et pour longtemps ?)

Il existe des solutions purement logicielles (voir la bibliographie) plutôt compliquées (dites aussi élégantes...).

La plupart des solutions s'appuient sur une combinaison *matériel et logiciel*.

Exemples :

- inhiber les interruptions (oui, mais) à l'intérieur d'appels système, donc en mode privilégié ;
- réquisitionner le bus des adresses ou données (cas des machines multi-processeurs).

Une des solutions les plus connues consiste à fournir des primitives gérant des *sémaphores*. Dijkstra (fin des années 1960) en est à l'origine.

Sémaphore

Les sémaphores sont des objets **communs partagés par plusieurs processus**, dotés d'opérations permettant de résoudre la synchronisation de processus.

On commence par une expression simple : Un sémaphore est une variable entière *s*, à valeurs non-négatives, à laquelle sont associées deux opérations **atomiques**, *Demander(s)* et *Libérer(s)*.

Demander (s) { si $s = 0$ alors attendre ; ; sinon $s = s - 1$; } ;	Libérer (s) { $s = s + 1$; Réveiller un processus en attente ; }
--------------------------------------------------------------------------------------	--------------------------------------------------------------------------------

Attention : implicitement, on suppose

- qu'il existe un moyen permettant de mémoriser les demandes non satisfaites ;
- que la variable *s* est initialisée avant toute demande.

Plan

8 Synchronisation et Communication Entre Processus

- Les Besoins
- Section Critique
- Sémaphores
- Verrou Fatal ou Deadlock

Autre forme

Un sémaphore est une classe :

Sémaphore S { entier val ; listeProcessus L ; }

avec deux opérations **atomiques** :

Demander (S) { $S.val = S.val - 1$; si $S.val < 0$ alors ajouter demandeur dans S.L ; le passer à l'état bloqué ; }	Libérer (S) { $S.val = S.val + 1$; si $S.val \leq 0$ alors enlever un processus de S.L ; le passer à l'état prêt ; }
--------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------

Remarque : demander $\equiv P \equiv \text{Wait} \equiv \text{Up}$
 libérer $\equiv V \equiv \text{Signal} \equiv \text{Down}$

Sémaphores - Utilisation

Dans un problème d'exclusion mutuelle, le sémaphore est une donnée commune, initialisée à 1.

Chaque processus voulant exécuter une section critique fait :

Demander(sémaphore) ;

SectionCritique ;

Libérer(sémaphore) ;

Fonctionnement :

- **Demander(sémaphore)** va être passante si le sémaphore est strictement positif et bloquante pour toute autre valeur.
- **Libérer(sémaphore)** va permettre au prochain demandeur (soit dans la file d'attente ou qui viendra plus tard) de réaliser la section critique.
- Noter que lorsque $sémaphore \leq 0$ dans la deuxième forme, sa valeur absolue représente la longueur de la file d'attente.

Exemple de Fonctionnement

On reprend l'exemple de réservation de places. Soit `verrouReserv` un sémaphore commun initialisé à 1. Chaque processus va faire :

Demander(verrouReserv) ;

si ($nbPlacesDem \leq nbPlacesMax - nbPlacesRes$) **alors**

`nbPlacesRes += nbPlacesDem ;`

 afficher ("réservation effectuée pour" `nbPlacesDem` "place(s)) ;

sinon

 afficher ("il reste seulement", $nbPlacesMax - nbPlacesRes$, "places") ;

Libérer(verrouReserv) ;

On peut reprendre maintenant l'exemple qui a mené au dysfonctionnement et montrer que cette fois-ci il n'y a plus d'incohérences.

Exemple d'Exécution - 1

On suppose que `verrouReserv` est créée et initialisée à 1 par un processus d'initialisation de l'application.

Rappel de la situation globale :

`nbPlacesMax = 100 ; nbPlacesRes = 98 ;`

P1 veut **deux** places et P2 en veut **une**.

P1 est actif ; dans sa variable locale il a `nbPlacesDem = 2`.

- P1 fait `Demander(verrouReserv)` ; alors `verrouReserv = 0` et P1 passe en section critique, effectue le test, obtient un résultat positif, puis est interrompu (horloge) de suite après le test :
- On suppose que P2 devient actif. Dans sa variable locale il a `nbPlacesDem = 1`. Il fait `Demander(verrouReserv)` ; alors `verrouReserv = -1` et P2 va dans la file d'attente de `verrouReserv`.
- Si P3 fait aussi une demande de réservation, il passera comme P2 en file d'attente et on aura `verrouReserv = -2`.

Exemple d'Exécution - 2

- Lorsque P1 sera réveillé, il finira la réservation, fera `nbPlacesRes=100` et affichera réservation effectuée pour 2 place(s).
- Tant que P1 n'a pas fait `Libérer(verrouReserv)`, aucun processus ne peut effectuer une réservation.
- P1 fera `Libérer(verrouReserv)` ; alors `verrouReserv = -1` et un processus de la file d'attente sera réveillé.
- P2 sera réveillé si la file d'attente est gérée selon le principe *premier entré premier sorti*.
- P2 affichera il reste seulement 0 places.
- P2 fera `Libérer(verrouReserv)` ; alors `verrouReserv = 0` et un processus de la file d'attente sera réveillé, en l'occurrence P3, qui donnera un résultat similaire à P2 pour la réservation et finira par remettre `verrouReserv = 1`.

Sémaphores - Question Fondamentale

Les sémaphores et les opérations associées permettent de résoudre l'**exclusion mutuelle** et aussi la **synchronisation** de processus (voir bibliographie pour la synchronisation).

Un système d'exploitation se doit d'offrir aux programmeurs de tels objets et opérations.

Question : Comment fait-il pour assurer l'atomicité des opérations **demander()** et **libérer()** ?

Réponse : par l'une des techniques signalées : masquage des interruptions, réquisition de bus, ... **mais** le système d'exploitation est auto-confiant... Noter qu'il suffit de masquer les interruptions à l'entrée de chaque opération, puis de les démasquer à la fin.

Sous **Unix**, ces primitives sont offertes par des appels système, *semget()* pour définir le(s) sémaphore(s) et *semop()* pour réaliser des opérations comme **demander()** ou **libérer()**.

Notion de Verrou Fatal

Cette notion donne lieu à des expressions très poétiques, comme *étréinte fatale* ou *embrasse mortelle*. Autant la connaître pour l'éviter. On parle de **verrou fatal** lorsqu'un processus P1 mobilise au moins une ressource R1 et attend une autre ressource R2 détenue par un processus P2 qui ne la libérera que lorsque P1 aura libéré R1.

Ceci peut être **direct**, comme dans le rendez-vous avec les tubes nommés :

- P1 tient R1 et attend R2 ;
- P2 tient R2 et attend R1.

ou **indirect** :

- P1 tient R1 et attend R2 ;
- P2 tient R2 et attend R3 ;
- P3 tient R3
- ...
- P_i tient R_i et attend R_{i+1} ;
- ...
- P_n tient R_n et attend R1 .

Plan

8 Synchronisation et Communication Entre Processus

- Les Besoins
- Section Critique
- Sémaphores
- Verrou Fatal ou Deadlock

Caractérisation

Une situation de verrou fatal existe entre processus si les conditions suivantes sont constatées simultanément :

- il y a **exclusion mutuelle** : un processus est en section critique, les autres attendant sa sortie ;
- il n'y a pas de **préemption** : les ressources ne sont pas préemptibles, autrement dit, une ressource ne peut être relâchée que volontairement par le processus qui la détient ;
- Il existe une **attente circulaire** : un ensemble de processus $\{P_0, P_1, \dots, P_i, \dots, P_n\}$ est tel que P_0 attend une ressource tenue par P_1 , ..., P_i attend une ressource tenue par P_{i+1} , ..., P_n attend une ressource tenue par P_0 .

Cette situation peut se décrire par un graphe orienté, appelé graphe d'allocation de ressources.

Résolution des Verrous Fatales

Comment peut-on traiter et résoudre le problème de ces verrous ?

- Si on veut **éviter** la situation, il faut qu'au moins une condition soit évitée. Pratiquement, ceci revient à éviter la dernière, donc à chercher un circuit dans un graphe orienté... Pas d'algorithme efficace disponible...
- Si on veut **détecter** la situation, on retombe sur le même problème. Et alors, même avec un algorithme efficace, il faudrait décider de tuer arbitrairement un processus ou préempter une ressource.
- Ne rien faire est donc une solution de repos.

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 **concurrency**
- chaque thread possède sa propre pile
- la création et le changement de contexte entre deux threads du même processus est donc très rapide

Plan

9

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

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)

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`.

Plan

9 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

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)

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 thread 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 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 !"
$
```

## 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

9

### 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

## 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 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 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<10000; i++){
 pthread_mutex_lock(&mutex);
 compteur++;
 pthread_mutex_unlock(&mutex);
 }
 return NULL;
}
```

## 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 débloquenté 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.

## 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.

## 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 débloquenté 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 débloquentés 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

## Plan

### 9 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

## 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);
 // débloque 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 débloque tous
 pthread_cond_broadcast(q->condNonVide);
 pthread_mutex_unlock(q->mutex); // fin de SC
 return q;
}
```

## 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;
}
```

## 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;
}
```

## 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
```

## 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;
...
```

## 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



### 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

## 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  $m_1$  et  $mr$
  - nb colonnes de  $m_1$ ==nb de lignes de  $m_2$
  - nb colonnes de  $m_2$  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
```

## 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
```

## 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

# Plan

## 9 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

# Perspectives et conclusion

- Gestion de la mémoire non traitée
- Autres Inter Process Communication (files de messages, ...)
- Gestion des réseaux (sockets)
- implémentation de services (Web httpd, mail, ...)

# 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