SR01 2003 - Cours Unix 1 - Interface programmes/système

1 SR01 2003 - Cours Unix 1 - Interface programmes/système

1.1 Buts des UVs systèmes

Les UVs "systèmes" introduisent des **concepts** :
- process, multitâche, programmation d'activités simultanées,
- mémoire virtuelle, accès partagé,
- pilotes d'entrées-sorties, interruptions,
- programmation par évènements,
- bibliothèque, interface de programmation,
- passage d'arguments, de pointeurs,

Les UVs "systèmes" introduisent des **savoirs** :
- architecture de l'ordinateur,
- fonctionnement de l'ordinateur,
- fonctionnement des éléments principaux, UC, mémoire, E/S
- interconnexion de systèmes,

Les UVs "systèmes" introduisent des **savoirs-faire** :
- programmation système,
- utilisation des pointeurs, gestion de mémoire,
- utilisation d'ordinateurs en réseau,
- utilisation des interfaces de commande (shell,graphique),

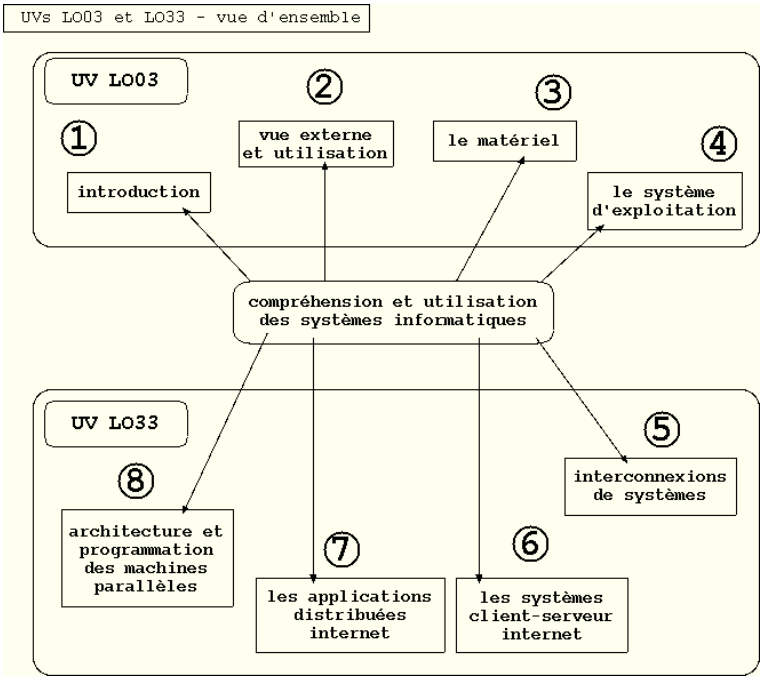


FIG. 1 – UVs LO03 et LO33 - vue d'ensemble

Certains concepts sont seulement exposés en cours et ne font pas l'objet d'une application en TD (impossible en seulement 34h de TD). Certains savoirs-faire ne sont pas

détaillés en cours : essentiellement liés à une pratique, ils ne sont présentés qu'en TD. Mais c'est **l'ensemble** qui constitue l'UV. La réunion de tout ce qu'on voit en cours **et** de tout ce qu'on voit en TD.

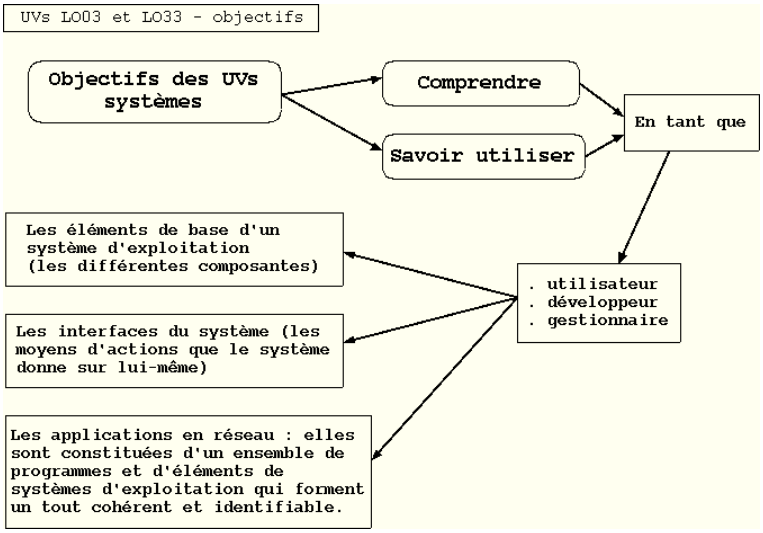


FIG. 2 – UVs LO03 et LO33 - objectifs

1.2 Vue extérieure du système

Un ordinateur est une machine de type automate très complexe, qui obeit à une **suite d'instructions**. Mais ces instructions élémentaires sont complexes à utiliser, nécessitent une bonne compréhension du fonctionnement interne de la machine et ne font chacune que très peu de "travail". Il faut donc en écrire beaucoup.

Pour simplifier et accélérer le processus d'utilisation d'un ordinateur on ajoute à celui-ci un ensemble de programmes pré-écrits. On utilise alors la machine en donnant des instructions à ces programmes et non pas directement au matériel. Cet ensemble de programmes porte le nom de "système d'exploitation", car il permet d'exploiter plus facilement le matériel sous jacent.

D'une certaine façon, le système d'exploitation **s'interpose** entre l'utilisateur et le matériel. Savoir utiliser un ordinateur se réduira dans 99% des cas, à savoir actionner les fonctions principales du système d'exploitation. Toutefois, de même que l'utilisation d'une automobile peut avoir des degrés de compétences :

- savoir conduire en conditions normales,
- savoir conduire sur la neige,
- savoir changer une ampoule de phare,
- savoir changer une roue,
- savoir changer une bougie,
- savoir diagnostiquer un refus de démarrage,
- etc ...,

l'utilisation d'un ensemble ordinateur+système d'exploitation va présenter des degrés de compétence :

- savoir lancer le traitement de texte, saisir et imprimer,
- savoir sauvegarder tout un répertoire,

– savoir diagnostiquer un refus d'imprimer,
– savoir écrire un programme simple,
– savoir écrire un programme faisant des appels complexes au système,
– savoir écrire un programme utilisant l'interface graphique,
– savoir écrire un programme communiquant avec d'autres programmes,
– savoir écrire un programme communiquant avec d'autres machines,
– etc ...,

Nous allons débuter par une présentation des fonctions principales et l'apprentissage de leur utilisation. Puis, pour chacune d'elles, on ouvrira la boîte pour expliquer son fonctionnement interne.

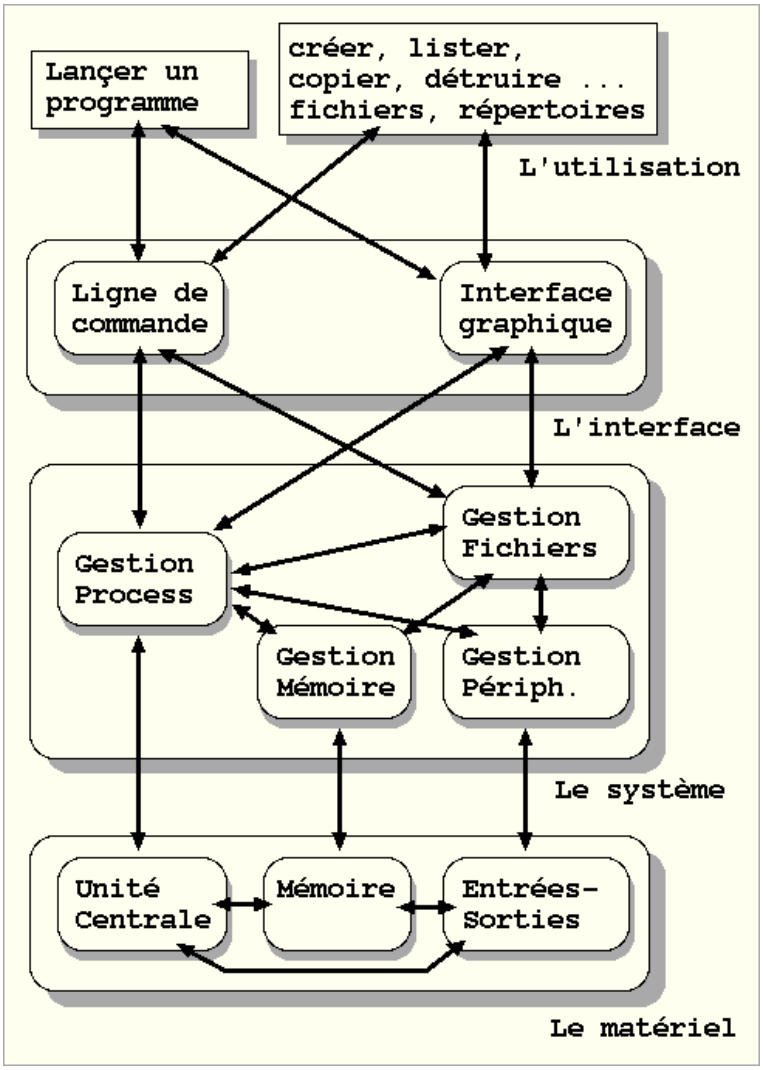


FIG. 3 – Éléments de base d'un système et d'un ordinateur

Les trois éléments de base du matériel sont l'UC, la mémoire et les E/S (figure 3 page 3). À chacun d'eux correspond une fonction principale du système : gestion des process, de la mémoire et des périphériques d'E/S. En raison de son importance particulière (stockage permanent des données et parfois du système lui-même), le système de gestion de l'espace des disques magnétiques est traité comme un sous-système indépendant. Il est appéle système de gestion de fichiers.

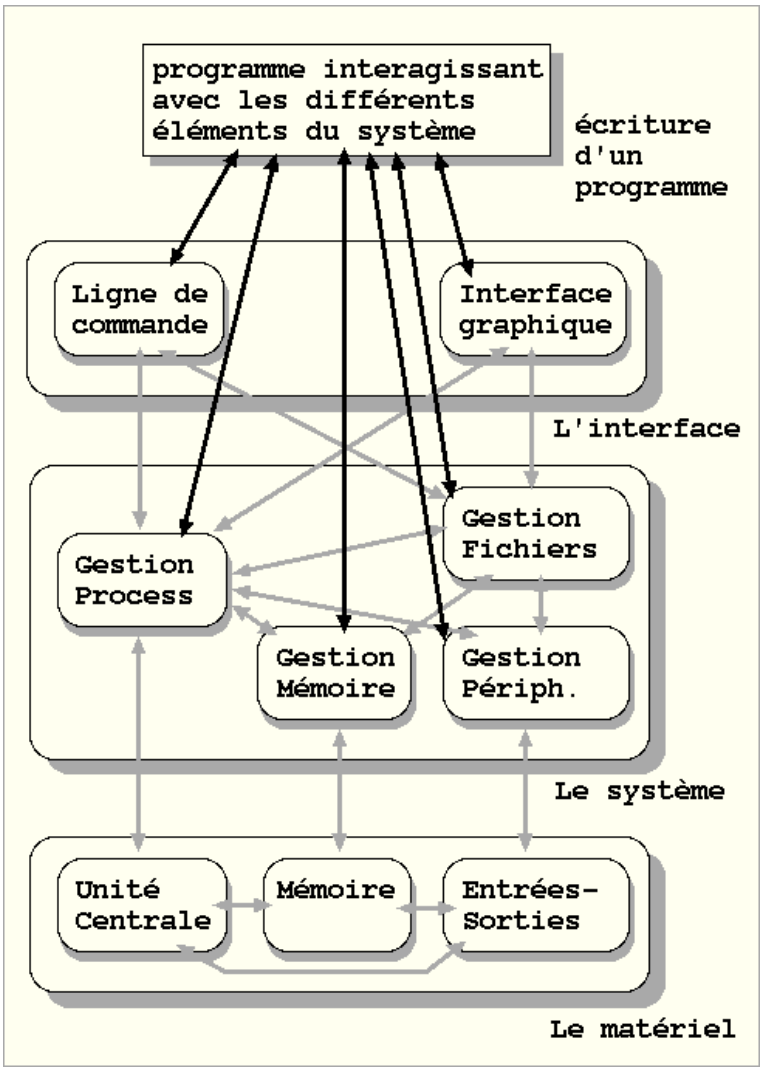


FIG. 4 – Un programme peut interagir avec les différents sous-ensembles

"Au-dessus", on va trouver les interfaces du système avec l'utilisateur. Il y en a de deux sortes : l'interface de type "ligne de commande" et l'interface de type "graphique".

Il y aura donc pour commençer un apprentissage minimum des deux types d'interfaces permettant d'agir sur le système.

Mais cette utilisation n'est pas suffisante pour un informaticien : on attend de lui, en plus, qu'il sache **écrire des programmes** qui vont s'ajouterà la collection des programmes fournis avec le système d'exploitation. Ces programmes écrits par l'utilisateur vont pouvoir s'interfacer avec le système à plusieurs niveaux :

- avec la ligne de commande (le shell),
- avec l'interface graphique,
- directement avec le système :
  - gestion de process (création, coopération, destruction,...),
  - gestion de mémoire (allocation, partage, ...),
  - gestion d'entrées-sorties,
  - gestion de fichiers.
- avec d'autres programmes, par l'intermédiaire d'appels au système.

1.3 Fonctionnement élémentaire de l'UC (Unité Centrale)

L'Unité Centrale (UC, en anglais "CPU" (Central Processing Unit)) ou processeur est un automate au fonctionnement répétitif. Elle va toujours, par construction exécuter l'instruction dont l'adresse en mémoire est contenue dans un registre particulier.

```
ia->  instruc_i
      instruc_i+1
      instruc_i+2
```

La prochaine instruction à exécuter est désignée par un registre spécial de l'UC appelé "ia" (instruction address) ou "pc" (program counter) selon les fabriquants

! un registre est un mot mémoire particulier construit à l'intérieur de l'UC, de telle sorte qu'il est accessible en un seul cycle d'horloge.

1.4 Notion d'appel système

Un appel système est une interruption synchrone : il provoque l'arrêt de l'exécution "normale" du programme en cours pour aller exécuter une fonction du système, avant de revenir au programme en cours. C'est équivalent à un appel de fonction, sauf que pendant l'exécution de cette fonction, il y a basculement de l'UC du mode "normal" au mode "privilégié".

Lorsqu'un programme a besoin d'accéder (lire, modifier) une donnée ou un élément géré par le système, il ne peut pas le faire directement :

le système se protège et protège les objets qu'il gère en utilisant une particularité du matériel (présente dans tous les ordinateurs) : l'accès soit en lecture, soit en écriture soit les deux à certaines zones de la mémoire peut être interdit en mode d'exéction "normal" (user mode).

Cet accès n'est possible que lorsque l'UC fonctionne en mode privilégié (ou système) (system mode ou kernel mode).

Donc :

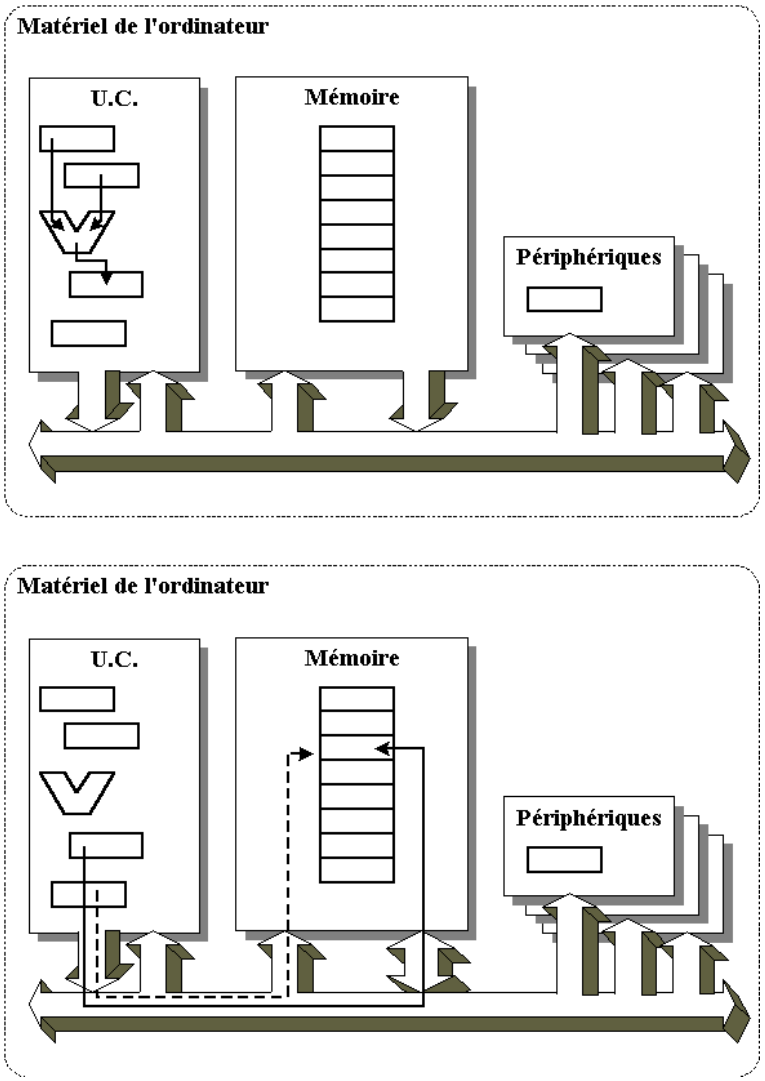


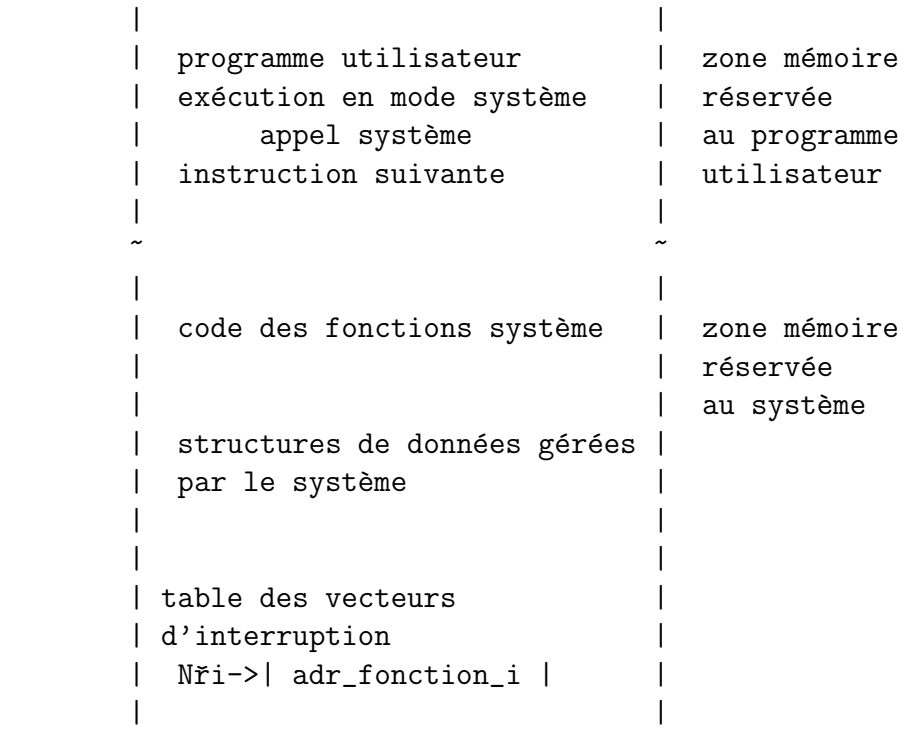
FIG. 5 – L'UC va chercher des données en mémoire et les manipule dans des registres

1. les instructions d'un programme utilisateur s'exécutent en mode non privilégié.
2. les instructions d'une fonction système s'exécutent en mode système

Mais : les programmes utilisateurs sont "lançés" par le système, il faut donc pouvoir passer du mode système au mode utilisateur,

Et : les programmes utilisateurs ont parfois besoin d'un accès à une donnée système. Cet accès devra être fait par une fonction du système s'éxécutant en mode système. Donc il faut passer du mode utilisateur au mode système.

Ainsi :



Il faut donc un mécanisme repide et efficace pour passer du mode utilisateur au mode système et retour

Pour cela tous les CPUs possèdent une paire d'instructions assembleur spécialisée :

```
system_call # appel au système
rti # return from interrupt
```

L'instruction system_call (qui peut avoir des noms différents selon les CPUs : SVC, INT, ...) va provoquer une séquence d'évènements qui va faire passer en mode système et exécuter la fonction système demandée en mode système, avant de revenir au programme appelant en mode utilisateur :

system_call va avoir au moins un argument : le numéro de l'appel système

Tous les appels disponibles dans le système sont rassemblés dans une table une entrée dans la table contient l'adresse de la fonction système le programme appelant exécute par exemple :

```
system_call i
```

pour demander l'exécution de l'appel système numéro i

L'intérêt de cette table intermédiaire, est qu'elle rend possible la modification du système sans avoir a reconstruire le programme utilisateur ainsi un programme utilisateur contruit sur la version "n" du système fonctionnera encore sur la version "n-1"

Séquence d'évènements :

1. exécution de l'appel system_call i
2. le matériel sauve les registres "pc" et "ps", soit dans des registres spéciaux dédiés à cet usage, soit le plus souvent sur une pile spéciale, dite "kernel stack"
3. le matériel charge une nouvelle valeur dans le registre physique "ps" ceci fait basculer le processeur en mode système
4. le matériel écrit dans le registre "pc" le contenu du champ adresse de l'entrée numéro i de la table des "vecteurs d'interruption" *** comme le registre "pc" contient, PAR DÉFINITION, l'adresse de la prochaine instruction à exécuter, ceci provoque un branchement sur la fonction système demandée

   Rem. : en (2) (3) et (4) on a bien dit "le matériel..." toute exécution d'une instruction system_call provoque automatiquement cette séquence pré-cablée dans le matériel PAR CONSTRUCTION. Jusqu'ici tout a été automatique et fait par le matériel
5. on exécute les instructions de la fonction système
6. à la fin la fonction système se termine par une instruction "rti" à la place du "return" standard Cette instruction va restaurer les registres "pc" et "ps" d'origine par construction l'instruction "rti" utilise le contenu de la pile système pour restaurer l'état de l'appelant, contrairement au "ret" qui utilise la pile "user" de chaque programme
7. le process qui a demandé l'appel système continue sur l'instruction suivant cet appel

1.5 Exemple : coût d'un appel système

```
/* mesure_apsys.c  */
/* mesurer la durée d'un appel système */
/* extrait de "Operating Systems, a design oriented approach"
        Charles Crowley, IRWIN, 1997 */

/*#define SANSAPSYS */
#ifdef SANSAPSYS
int getpid() { return 11;}
#endif

int main( int argc, char *argv[] ) {
    int i,a, boucle=atoi(argv[1]);
    for (i=0; i<boucle; i++) a = getpid();
}

-------------------------------------------------

$ gcc -o mesure_apsys mesure_apsys.c
$ time ./mesure_apsys 10000000
```

```
real    0m3.047s
user    0m1.600s
sys     0m1.440s
---> 1.440s / 10000000 = 0.1440 10-6 = 0.1440 microsec

$ gcc -o mesure_apsys -DSANSAPSYS mesure_apsys.c
$ time ./mesure_apsys 10000000

real    0m0.118s
user    0m0.110s
sys     0m0.000s

-------------------------------------------------

$ gcc -o mesure_apsys -DSANSAPSYS mesure_apsys.c
$ time ./mesure_apsys 40000000

real    0m0.480s
user    0m0.470s
sys     0m0.000s

$ gcc -o mesure_apsys mesure_apsys.c
$ time ./mesure_apsys 40000000

real    0m12.482s
user    0m7.610s
sys     0m4.680s
-------------------------------------------------
```

1.6 Notion de process

```
programme    objet statique
             existe dans l'espace (disque ou mémoire)
             consiste en une suite d'instructions

process      objet dynamique
             existe dans le temps
             consiste en une suite d'exécutions d'instructions

Par exemple, le programme :
        main() {
          int i,r;
          for( i=0; i<100; i++) r = r * i;
        }
est constitué de quelques instructions,
```

quand on demande son exécution, le système va créer un process qui va exécuter 100 instructions de contrôle de boucle et 100 instructions de multiplication puis ce process se terminera et sera détruit

si on demande une nouvelle exécution du MÊME programme, un NOUVEAU process sera créé

```
ps

ps -aef

ps -aux
```

1.7 Interface programme / système

(figure 6 page 10)

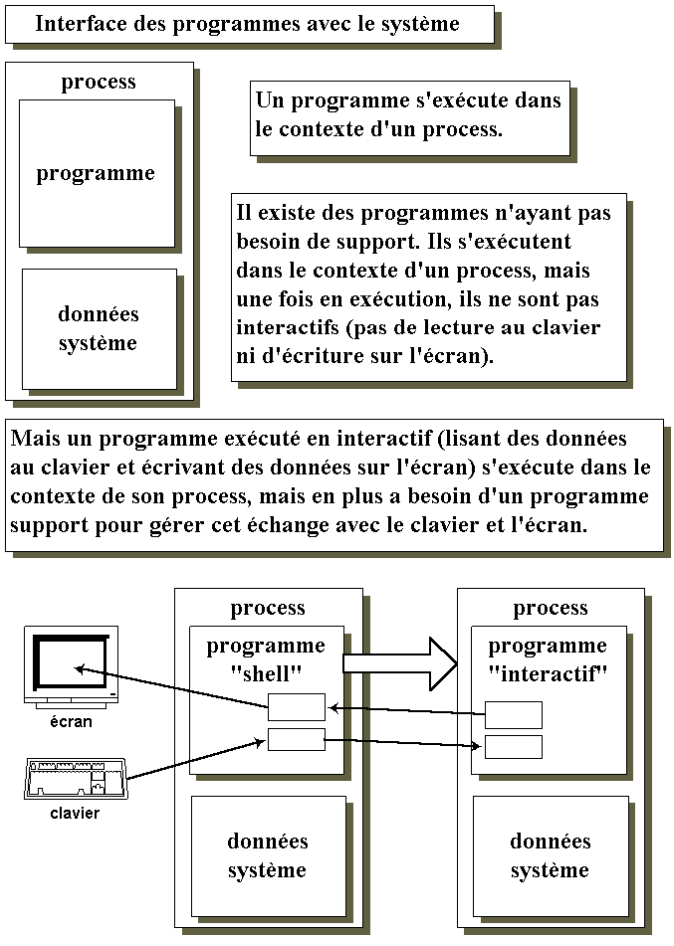


FIG. 6 – Interface des programmes avec le système

(figure 7 page 11)

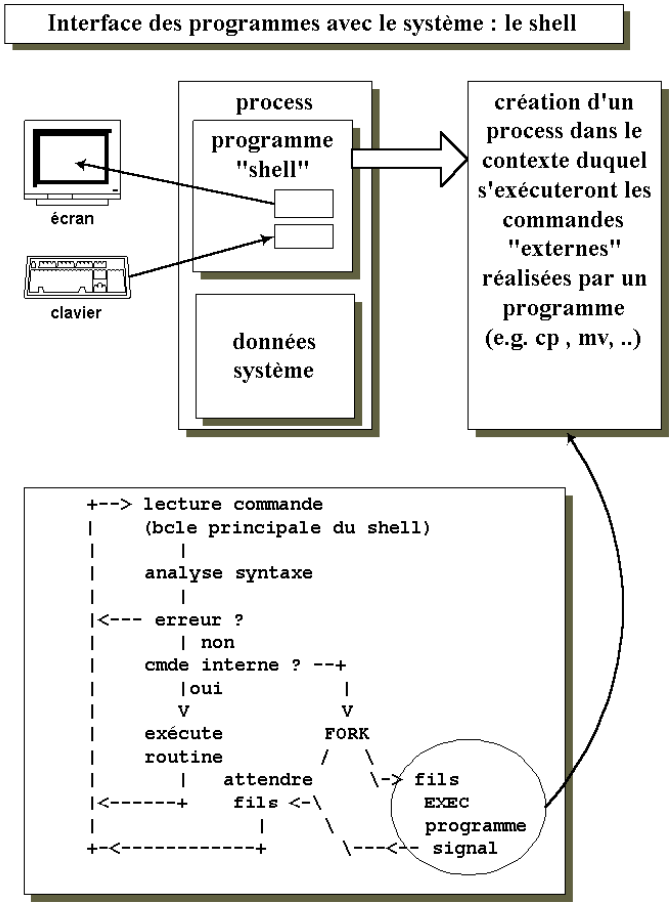


FIG. 7 – Interface des programmes avec le système : le shell

1.8 Interface des programmes avec le système : fork() unix

(figure 8 page 12)

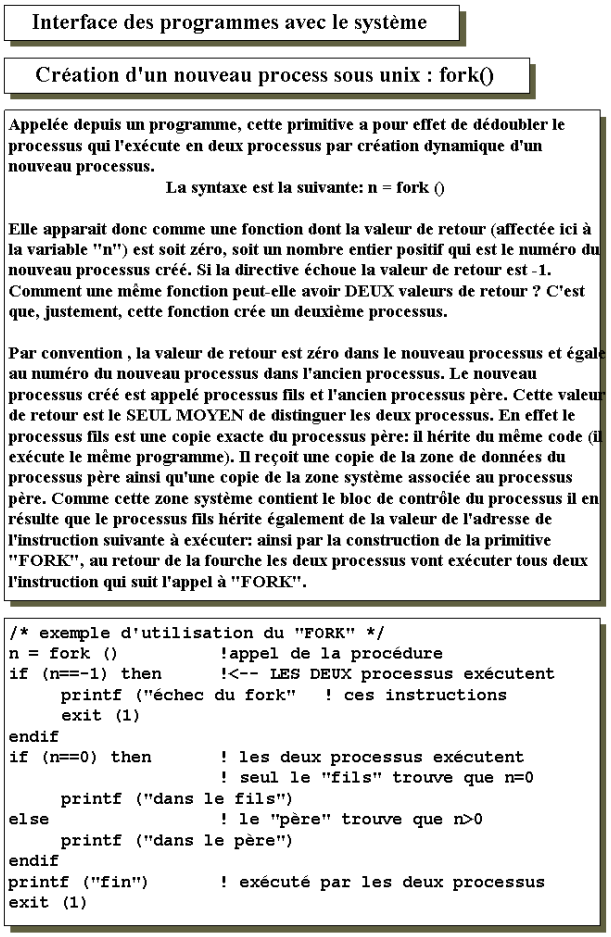


FIG. 8 – Interface avec le système ; création de process unix : fork()

(figure 9 page 13)

1.9 Code d'un shell simple

```
/* shell0.c */
/* montrer fonctionnement d'un shell */
/* l' Michel.Vayssade@utc.fr oct 2003 */

int ARGLNG=20;
int ARGNUM=5;

int intf_pid(); /* prototype */
int intf_fin(); /* prototype */

#define NCMDES 2
char *intcmdes[NCMDES]={"pid","fin"};
int (*intpters[NCMDES])() ={intf_pid,intf_fin} ;
```

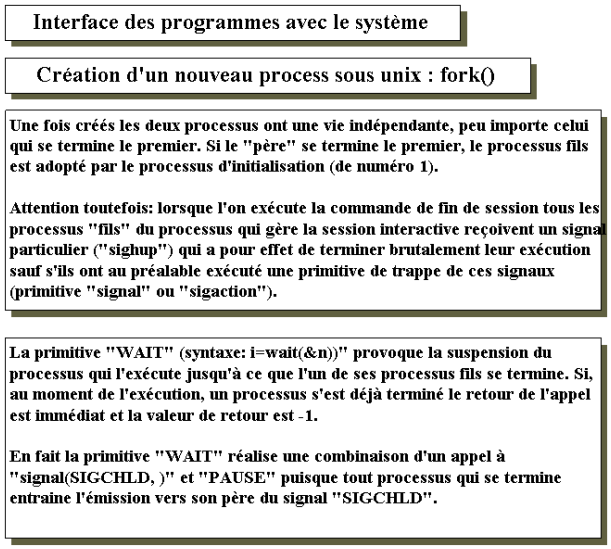


FIG. 9 – Interface avec le système : fork()

```
char buf[80];

int main( int argc, char *argv[] ) {
    int lnlue, n;
    while (1) {

        printf("mon_shell> ");
        lnlue=scanf("%s",buf);
        printf("lu=%d %s\n",lnlue,buf);

        n = strcmp(buf,intcmdes[0]);
        printf("n=%d\n",n);

        if ( n==0 ) {
          printf("cmde pid\n");
          n = intpters[0]();
          printf("le pid=%d\n",n);
        }

        n = strcmp(buf,intcmdes[1]);
        if ( n==0 ) {
          printf("cmde fin\n");
          n = intpters[1]();
          printf("jamais ici n=%d\n",n);
        }

    }/*end while(1) */
```

```
}/*end main */


int intf_pid() {
        return getpid();
}
int intf_fin() {
        _exit(1);
}


$ ./shell0
mon_shell> pid
lu=1 pid
n=0
cmde pid
le pid=4335
mon_shell> aaa
lu=1 aaa
n=-1
mon_shell> fin
lu=1 fin
n=-1
cmde fin
```

```
/* l' Michel.Vayssade@utc.fr oct 2003 */
/* gcc -o shell1 shell1.c */

#include <sys/wait.h>
#include <stdio.h> /* pour define stdout */

#define ARGLNG 20
#define ARGNUM 5
char arg[ARGNUM][ARGLNG];
char cmd[ARGLNG];

int intf_pid(); /* prototype */
int intf_fin(); /* prototype */

#define NCMDES 2
char *intcmdes[NCMDES]={"pid","fin"};
int (*intpters[NCMDES])() ={intf_pid,intf_fin} ;

char buf[80];

int main( int argc, char *argv[] ) {
    int lnlue, n,m, status;
    int tour=0;
```

```
    while (tour<=5) {
        tour++;
        fprintf(stdout,"mon_shell pid %d> ",getpid());
        fflush(stdout);
        lnlue=read(0,buf,80);
        printf("lu=%d %s\n",lnlue,buf);
        buf[lnlue-1]=0;

        n = strcmp(buf,intcmdes[0]);
        printf("n=%d\n",n);
        if ( n==0 ) {
          printf("cmde pid\n");
          n = intpters[0]();
          printf("le pid=%d\n",n);
          continue;
        }
        n = strcmp(buf,intcmdes[1]);
        if ( n==0 ) {
          printf("cmde fin\n");
          n = intpters[1]();
          printf("jamais ici n=%d\n",n);
          continue;
        }

        /* ici si pas une cmde "interne" */
        printf("cmde externe\n");

        n=fork();/*  */
        if ( n==-1 ) {printf("échec du fork\n"); continue;}
        if ( n==0 ) {
          printf("dans fils buf=%s pid=%d\n",buf,getpid());
          n= system(buf);
          printf("status of system(cmde)=%d\n",WIFEXITED(n));
          _exit(3);
        }else{
          m=waitpid(n,&status,WUNTRACED);
          if (m!=n) printf("** m=%d",m);
          if(WIFEXITED(status)) {
            printf("Code retour fils: %d\n",WEXITSTATUS(status));
          }
        }

    }/*end while(1) */

}/*end main */
```

```
int intf_pid() {
        return getpid();
}
int intf_fin() {
        _exit(1);
}


$ ./shell1
mon_shell pid 4341> pid
lu=4 pid

n=0
cmde pid
le pid=4341
mon_shell pid 4341> ls *.c
lu=7 ls *.c

n=-1
cmde externe
dans fils buf=ls *.c pid=4342
mesure_apsys.c  shell00.c  shell0.c  shell1-tests.c  tok.c
redir.c         shell01.c  shell1.c  shelline.c
status of system(cmde)=1
Code retour fils: 3
mon_shell pid 4341> aaa
lu=4 aaa
.c
n=-1
cmde externe
dans fils buf=aaa pid=4344
sh: aaa: command not found
status of system(cmde)=1
Code retour fils: 3
mon_shell pid 4341> fin
lu=4 fin
.c
n=-1
cmde fin
```

SR01 2003 - Cours Unix 2 - Les signaux sous unix

2 SR01 2003 - Cours Unix 2 - Les signaux sous unix

2.1 Interface des programmes avec le système par l'intermédiaire du shell

Interface des programmes avec le système sous unix

Dans le cours précédent on a montré qu'un process est créé par le shell pour servir de "support" (de "contenant") à l'exécution d'un programme (ğ 1.7).

On a aussi expliqué le mécanisme de création de process, qui utilise l'appel système fork() (ğ 1.8 et suiv.).

Remarquons que ce mécanisme unix (création d'un nouveau process pour tout programme lancé depuis un shell) n'est pas le seul qui ait été utilisé dans les systèmes d'exploitation : (figure 10 page 17)

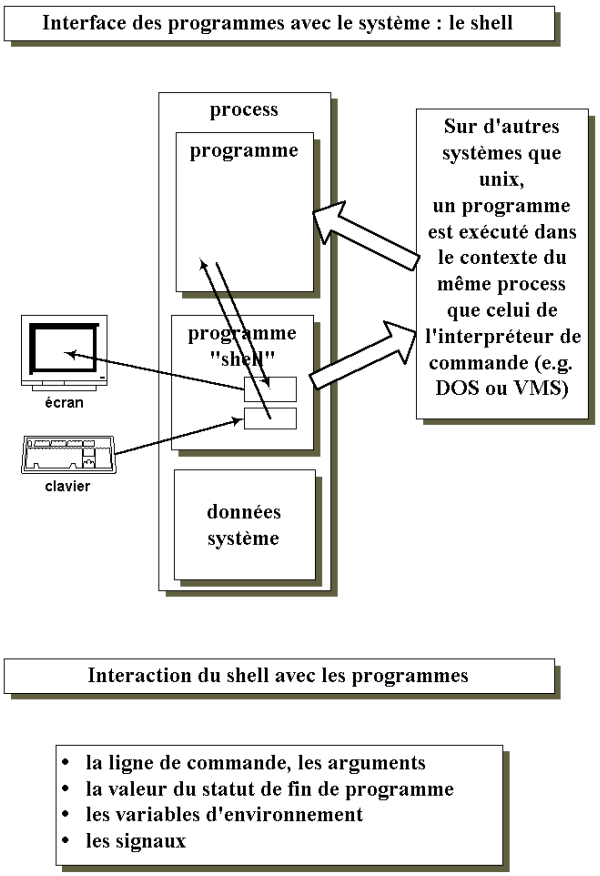


FIG. 10 – Certains systèmes ne crèent pas de nouveau process pour exécuter un programme.

Unix a généralisé le mécanisme de création de process par fork() en le complémentant avec l'ensemble des directives execxx() qui permettent, après avoir dupliqué un process, de remplacer, dans le process fils, le programme en cours d'exécution par un autre programme. C'est ainsi que le système unix crèe des hiérarchies de process. (figure 11 page 18)

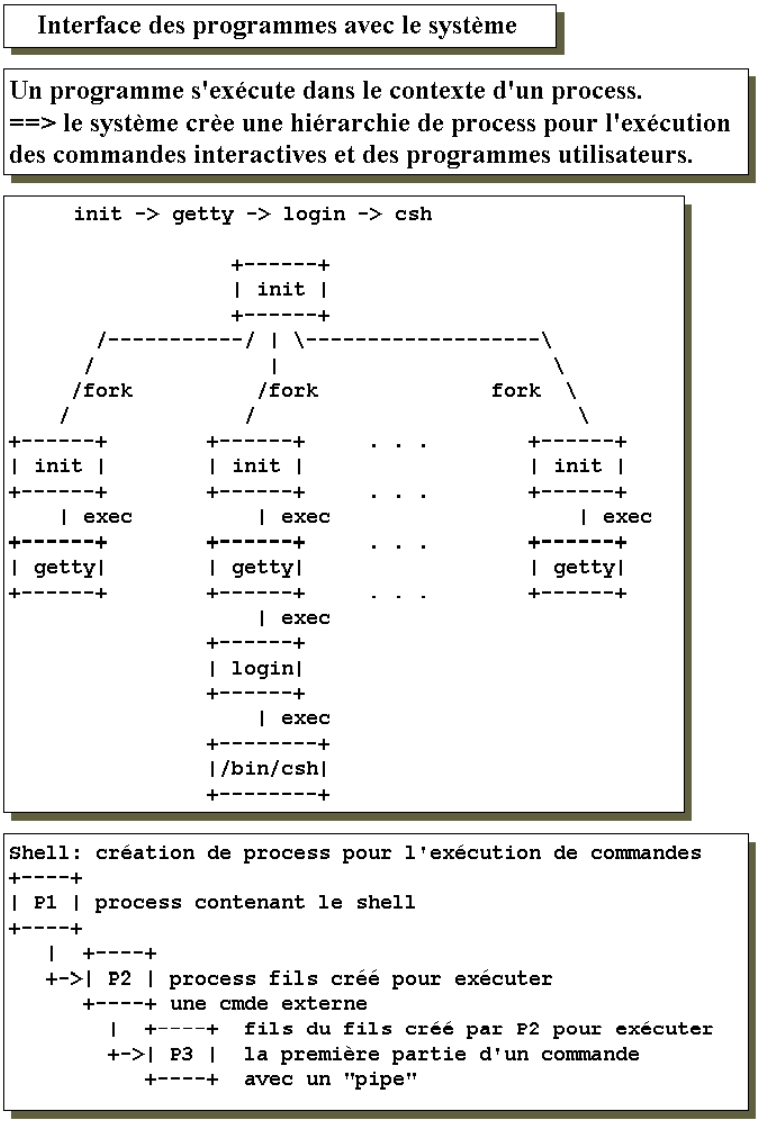


FIG. 11 – Hiérarchie de process créée par un système unix.

Interface des programmes avec le système sous unix

Le shell utilise plusieurs mécanismes du système pour échanger avec les process créés des informations ou des signaux de contrôle. (figure 12 page 19 et suivantes)

Exemple de passage d'information de "statut" de fin entre programme et shell. helstatus.txt

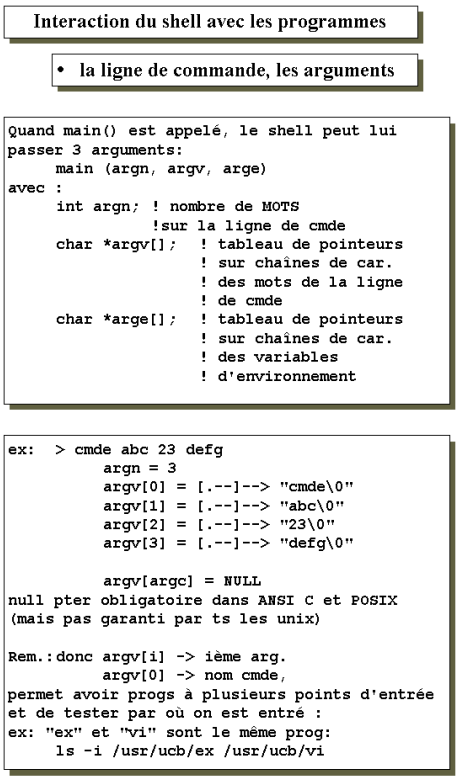


FIG. 12 – Ligne de commande et arguments avec un shell unix.

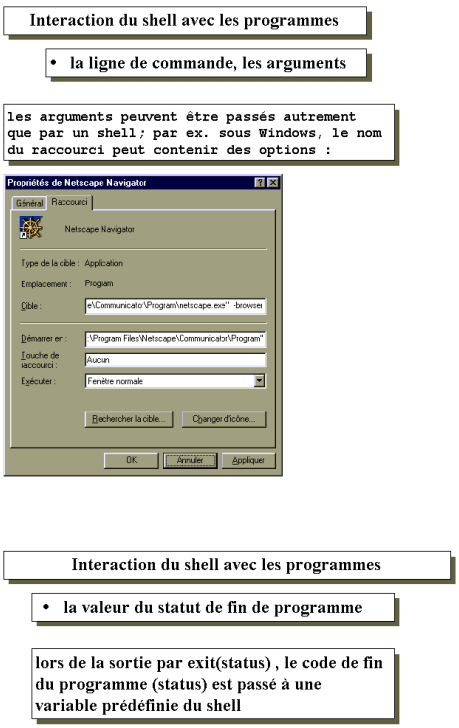


FIG. 13 – Ligne de commande et arguments sous windows.

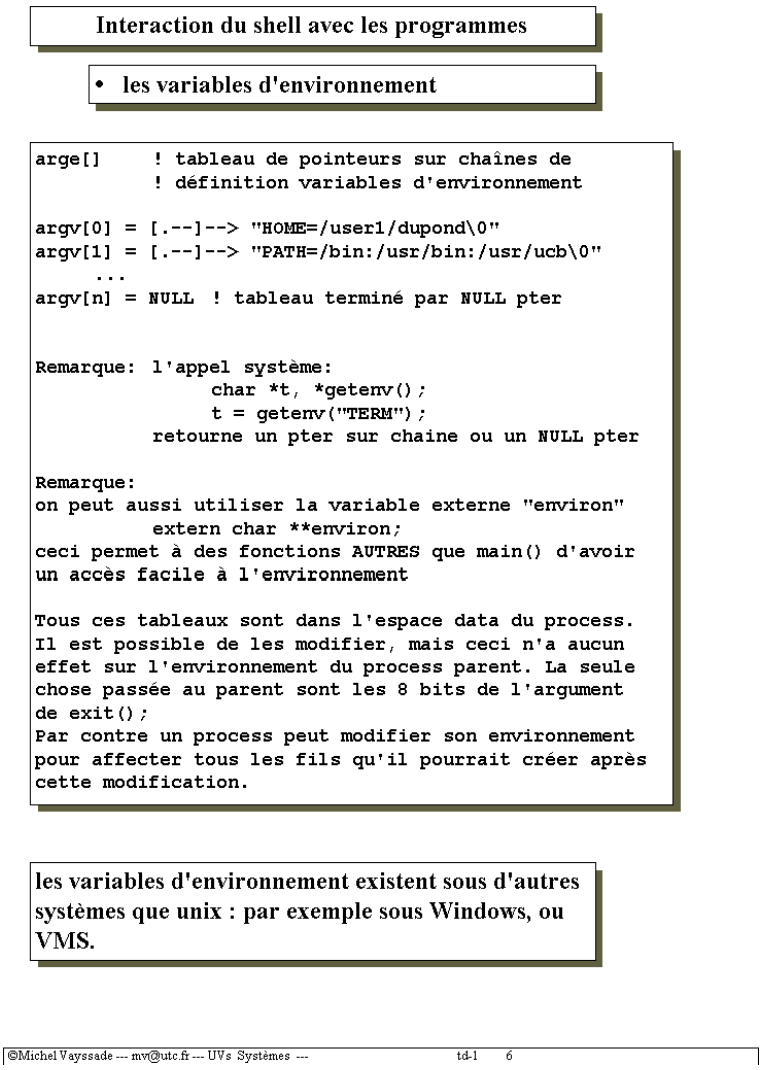


FIG. 14 – Variables d'environnement.

```
> more hello.c

#include <stdio.h>
    main ()
    {
     printf ("Hello !\n") ;
    }
-----------------------------
> more hello0.c

#include <stdio.h>
    main ()
    {
     printf ("Hello !\n") ;
     exit(0);
    }
-----------------------------
> more hel

#!/bin/csh
#               hel

set status=2
echo "status=" $status

hello
echo "status=" $status
-----------------------------
> more hell

#!/bin/csh
#               hell
set status=2
echo "status=" $status

hello

if  ( $status == 0 )  then
        echo "hello OK"
else
        echo "hello NON"
endif
---------------------------------------

659 lo03 sunserv:~/mv/espace-virt> hel
status= 2
Hello !
```

```
status= 1

660 lo03 sunserv:~/mv/espace-virt> hel0
status= 2
Hello !
status= 0

661 lo03 sunserv:~/mv/espace-virt> hell
status= 2
Hello !
hello NON

662 lo03 sunserv:~/mv/espace-virt> hell0
status= 2
Hello !
hello OK
----------------------------------------
```

Un autre mécanisme permet l'échange d'informations de type "évènement" : l'envoi et la réception de "signaux". (figure 15 page 22)

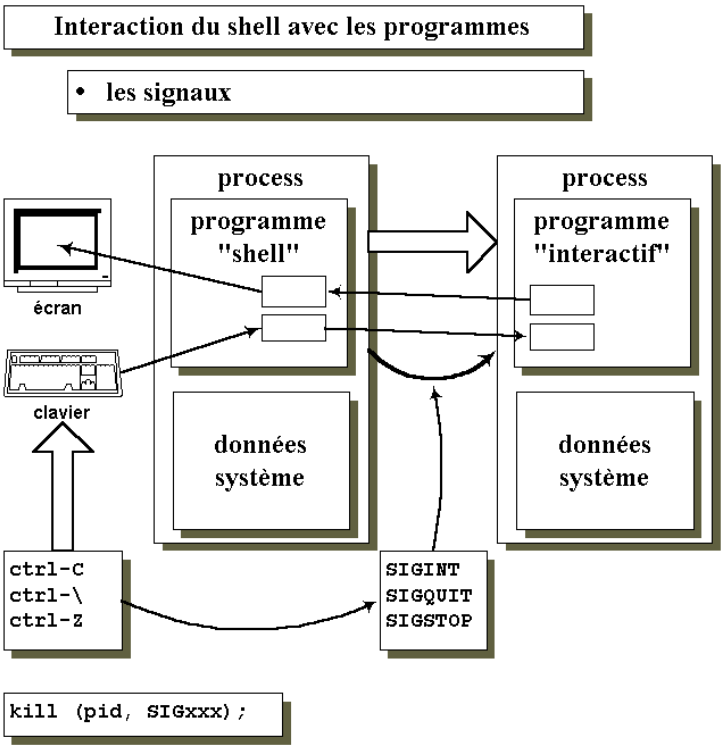


FIG. 15 – Envoi de signaux par le shell sous unix.

Ce sujet sera développé dans les paragraphes suivants.

Interface des programmes avec le système sous unix

Avant de détailler le fonctionnement des signaux on va montrer un autre exemple de création de process et les problèmes que pose le partage mal géré de la ”fenêtre console” entre le shell, un process parent et un process fils. (figure 16 page 23 et suivantes)

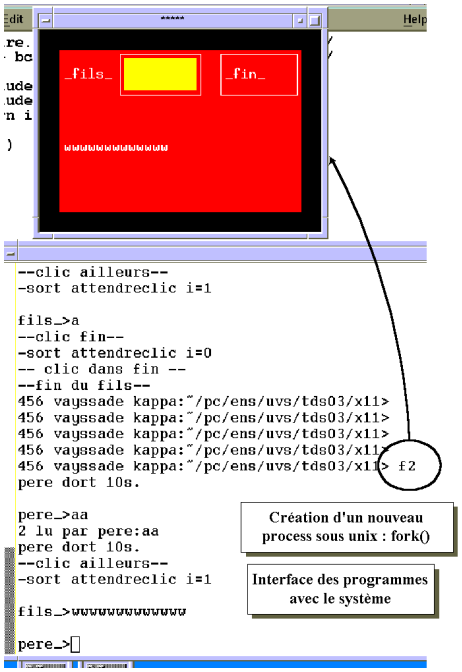


FIG. 16 – Le shell lance un programme qui crèe un fils.

L’exemple des figures 16 et suivantes : le père et fils lisent tous les deux sur la console ; Le fils gère une fenêtre graphique.

```
/* f2.c exemple fork+X11 */
/* appelle routines definies dans fx.c
 * > gcc -c fx.c
 * > gcc -o f2 f2.c fx.o -lX11
 * > gcc -o f2 f2.c fx.o -L/usr/X11R6/lib -lX11  (linux)
 */
/* main fork un fils qui : initialise rectangle rouge,
 * attends dans attendreclic();
 * Lors d’un clic dans bouton en couleur, "toggle" couleurs.
 * Sort si clic dans bouton fin.
 */

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

int pidfils;
#define lng 40
char buf[lng];
```

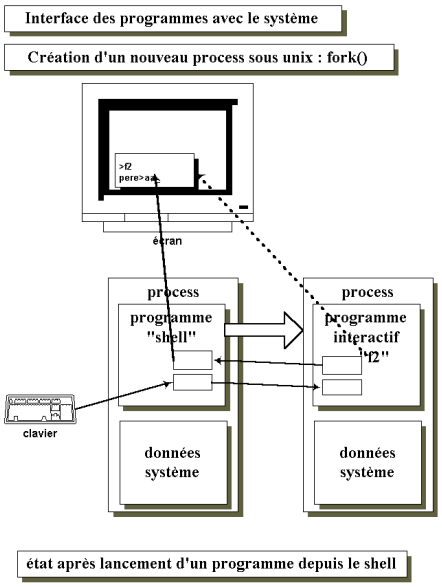


FIG. 17 – Interactions shell<–>programme avant création du fils.

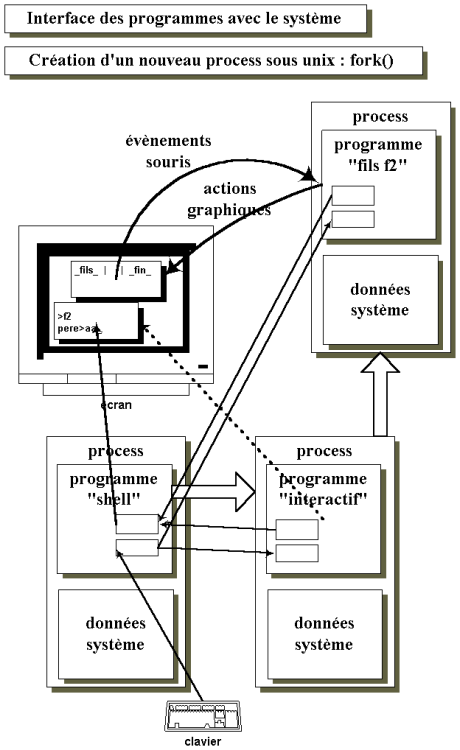


FIG. 18 – Interactions shell<–>programme<–>fils.

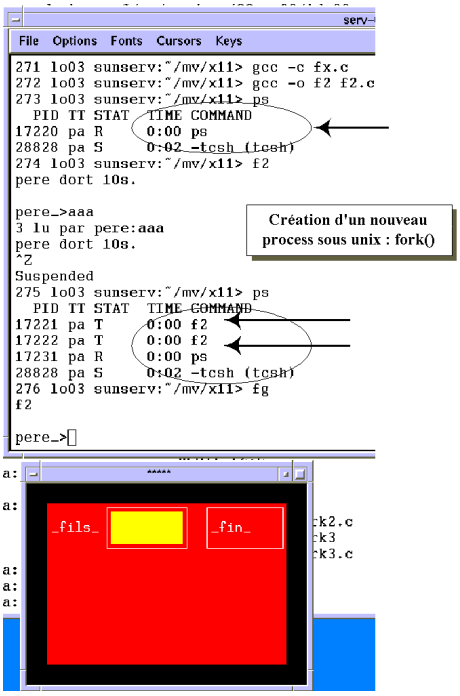


FIG. 19 – Les process support du shell, du programme et du fils.

```
int liretty (char *prompt, char *buffer)
{ int i;
printf("\n%s",prompt);
i = scanf ("%s",buffer);
return strlen(buffer);
}

main ()
{  int i,nlu;
pidfils = fork();
if (pidfils==-1) {perror("echec fork"); exit(0);}

if (pidfils==0) {  /* fils */

initrec(); /* creer rectangle rouge */
do {
i = attendreclic ();
printf("-sort attendreclic i=%d\n",i);
if (i == 0) fprintf (stderr, "-- clic dans fin --\n");
if (i == 1) {
nlu = liretty("fils_>",buf);
ecritrec (buf,nlu);
}
} while (i==1);

        printf("--fin du fils--\n");
```

```
detruitrec(); /* detruire la fenetre rectangle */
exit(EXIT_SUCCESS);
}
else  { /* pere */

do {
printf("pere dort 10s.\n");
         sleep (10);
nlu = liretty("pere_>",buf);
printf("%d lu par pere:%s\n",nlu,buf);
} while (buf[0]!='f');

        printf("--fin du pere--attendre fin du fils--\n");
wait();
exit(EXIT_SUCCESS);
}
}
```

SR01 2003 - Cours Unix 2 - Les signaux sous unix

2.2 Notion de signal unix

Envoi de signaux par le shell à un process fils

Le shell transforme la lecture de certains caractères spéciaux en signaux envoyés aux process fils.

Un signal peut interrompre un autre signal.

Dans l'exemple suivant, on capture les signaux SIGINT et SIGQUIT correspondant aux caractères ctrl-C et ctrl-\.

```
/* signal_10.c */
/* Montrer fonctionnement directive signal() traditionnelle */
/* UTC UV SR01 - (c) Michel.Vayssade@utc.fr oct 2003 */
/* gcc -o signal_10 signal_10.c */

#include <signal.h>
#include <stdio.h>

/* Declaration de variables globales de comptage */
int cp1=0, cp2=0;

/* fonctions et procedures */

/* fonction executée quand le process fils a capté un signal SIGINT */
void captSIGINT(int sig){
```

```
signal(SIGINT,captSIGINT);
cp1++;
printf("Le process a capté un Sigint.\n");
printf("3 secondes de pause pendant lesquelles "\
"on peut générer une autre interruption.\n");
sleep(5);
/* la pause de 3 secondes va nous permettre de provoquer une
autre interruption pendant le traitement de la premiere.*/
printf("fin des 3 secondes de pause. Sortie de captSIGINT().\n");
/* permet de savoir quand le traitement de l'interruption
   va s'achever. */
}

/* ^\ provoque une sortie après la 3ème capture.*/
void captSIGQUIT(int sig){
signal(SIGQUIT,captSIGQUIT);
cp2++;
printf("SIGQUIT capte par le process.\n");
if (cp2>=3)
{ printf("Le process a capté son troisieme et dernier SIGQUIT. Exit.\n");
  exit(10);
}
}


/* Programme principal */
main() {
  int n;
  signal(SIGINT,captSIGINT);
  signal(SIGQUIT,captSIGQUIT);

  printf("On est dans le process de pid: %d\n",getpid());
  while(1) {
    printf("a\n"); /* Boucle infinie */
    sleep(1);
  }
}
```

---

signal_10.txt : exécution sur Linux

```
$ gcc -o signal_10 signal_10.c
$ ./signal_10
On est dans le process de pid: 2693
a
a
Le process a capté un Sigint.
```

```
3 secondes de pause pendant lesquelles on peut générer une autre interruption.
SIGQUIT capte par le process.
fin des 3 secondes de pause. Sortie de captSIGINT().
a
a
a
Le process a capté un Sigint.
3 secondes de pause pendant lesquelles on peut générer une autre interruption.
SIGQUIT capte par le process.
fin des 3 secondes de pause. Sortie de captSIGINT().
a
a
Le process a capté un Sigint.
3 secondes de pause pendant lesquelles on peut générer une autre interruption.
SIGQUIT capte par le process.
Le process a capté son troisieme et dernier SIGQUIT. Exit.
```

2.3 Les signaux sous unix

Pour des raisons historiques, il y a quatre environnements de signaux qui cohabitent sous unix

- System V non-fiable, "traditionnel"
- BSD 4.x
- System V fiable ("reliable") (SVR3)
- POSIX

Environnement de signaux System V non-fiable ("traditionnel")

Il y a trois limitations :

- le traitement récursif est toujours permis
- les handlers sont remis à SIG_DFL avant l'appel
- les appels systèmes sont interrompus par la délivrance d'un signal

C'est la combinaison de ces trois comportements qui rend les signaux System V traditionnels non-fiables : comme les signaux peuvent être déclenchés récursivement **et** que la routine de traitement **doit** ré-enclencher l'appel "manuellement", il y a une "fenêtre" pendant laquelle le comportement par défaut peut être déclenché. Le plus souvent le comportement par défaut d'un signal "capté" est de l'ignorer. Ceci entraine la perte du signal. Dans certains cas l'action par défaut peut même être de terminer le process !

Les appels systèmes peuvent être interrompus par la délivrance d'un signal. Pourtant très peu d'applications testent le cas d'erreur EINTR et redémarrent l'appel correspondant.

Environnement de signaux System V non-fiable ("traditionnel")

L'interface traditionnelle est la directive **signal()** :

int signal (int signal_number , void * sighandler)

```
$ man signal
NAME
    signal - ANSI C signal handling
SYNOPSIS
    #include <signal.h>
    void ( *signal(int signum,
           void (*sighandler)(int)))(int);
DESCRIPTION
The signal()  system call installs  a new signal
handler  for the signal with number signum.  The
signal handler is  set to  sighandler  which may
be a user specified function,  or either SIG_IGN
or SIG_DFL.
```

Environnement de signaux BSD 4.x ("traditionnel")

Pour corriger les défauts du System V traditionnel, les concepteurs de BSD ont modifié le comportement de la directive signal().

- Les signaux sont bloqués pour la durée d'exécution du handler (les appels récursifs d'un même signal ne sont pas possibles) ;
- Un **masque de signaux** permet de bloquer la plupart des signaux peendant les opérations critiques ;
- La plupart des appels systèmes sont redémarrés après reception d'un signal.

Environnement de signaux BSD 4.x ("traditionnel")

L'interface BSD inclue la directive signal() avec le comportement modifié PLUS les fonctions **sigvec()**, **sigstack()**, **siginterrupt()**, **sigsetmask()**, **sigblock()**, **sigpause()**.

- **int signal (int signal_number , void * sighandler)**
- **int sigvec (int sig_num, struct sigvec *new_sigvec, struct sigvec *old_sigvec) ;**
  sigvec() change ou demande l'état du handler du signal "sig_num".
- **int siginterrupt (int signal_number , int interrupt) ;**
  Modifie la propriété SV_INTERRUPT du handler de signal_number. Si interrupt=0 les appels systèmes sont redémarrés, sinon ils renvoient EINTR.

Environnement de signaux System V fiable

À partir de la version SVR3, une nouvelle interface est introduite dans System V. Malheureusement, elle différente (et moins souple) que celle de BSD.

int sigset (int signal_number , int (* sighandler))

Cette interface fournit un environnement similaire à celui de BSD, à l'exception du fait qu'un appel à sigset() remets à zéro le masque de signaux, rendant impossible la construction d'une section critique dans laquelle un handler de signal peut être changé.

De plus, cette interface ne permet pas de faire de façon atomique le dépbloquage et l'attente sur plus d'un signal à la fois.

Environnement de signaux POSIX

Pour être sûr que personne ne puisse écrire facilement du code portable, le comité POSIX à ajouté un autre environnement de signaux !

Toutefois cet environnement est largement dérivé de celui de BSD et permet d'émuler tous les autres, si nécesaire.

Malheureusement le comité POSIX n'a pas standardisé toutes les options, ce qui fait que des implémentations différentes peuvent avoir des comportements différents dans certains cas.

Environnement de signaux POSIX

Exemples de signaux :
"extérieurs" au process :
- intr : cntrl-C = SIGINT
- quit : cntrl-D = SIGQUIT
- susp : cntrl-Z = SIGSTOP

"intérieurs" au process :

– violation accès mémoire = SIGSEGV

La fonction principale de l'interface est sigaction() :

```
int sigaction (int signal_number ,
               struct sigaction *new_handler,
               struct sigaction *old_handler); )
```

Les autres fonctions de l'interface POSIX sont : sigprocmask(), sigemptyset(), sigfillset(), sigaddset(), sigdelset(), sigismember(), sigpending() ;

Environnement de signaux POSIX

La structure sigaction décrit l'environnement du signal :

```
struct sigaction {
   void (*sa_handler)(int); /* fonction à exécuter */
   sigset_t sa_mask; /* nouveau masque de signaux */
   int sa_flags;     /* options */
  };
```

Elle s'utilise ainsi :

```
struct sigaction nact, oact; /* déclaration */
nact.sa_handler = captint ; /* fonction handler */
sigemptyset(&nact.sa_mask);
nact.sa_flags = 0;
i = sigaction (SIGINT,&nact,&oact);
```

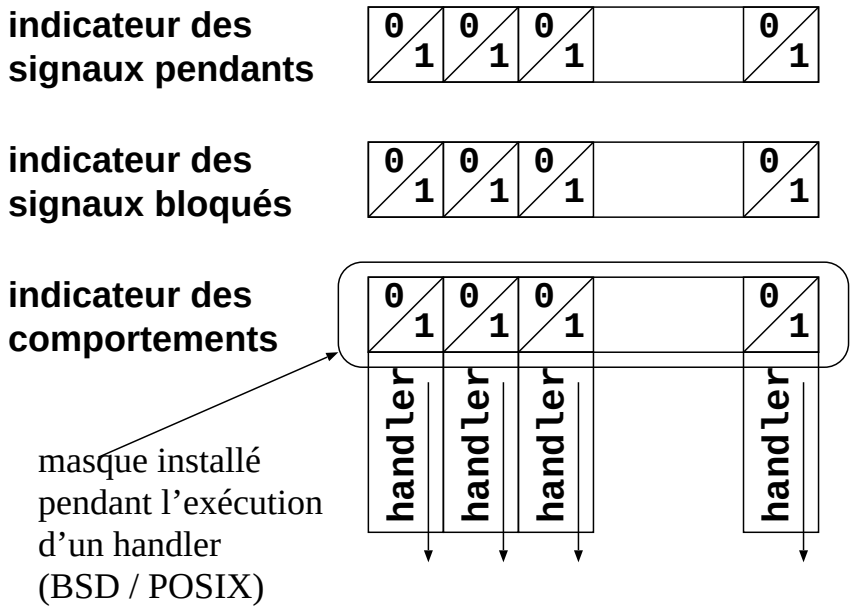
Environnement de signaux POSIX

Trouver le liste des signaux : **man 7 signal** ou **kill -l**

```
The signal()  system call installs  a new signal
SIGNAL(7)   Linux Programmer's Manual  SIGNAL(7)
NAME
   signal - list of available signals
DESCRIPTION
   Linux supports the signals listed below.
   Several signal numbers are architecture
   dependent.  First the signals described
   in POSIX.1.

   Signal Value  Action Comment
   ------------------------------
   SIGHUP    1    A    Hangup detected ..
                 or death of process
   SIGINT    2    A    Interrupt ...
   SIGQUIT   3    C    Quit from keyboard
```

Environnement de signaux POSIX

La structure des blocs de contrôle des signaux est la suivante :



Environnement de signaux POSIX

Fonctionnement des signaux :
- un signal **pendant** ("pending") a été envoyé au process mais pas encore pris en compte ;
- le signal bascule **un bit de 0 à 1** : si un autre signal arrive avant la prise en compte, il est perdu ;
- toutefois les version BSD et POSIX peuvent **bloquer** (ou masquer) un ensemble de signaux ; elles peuvent aussi indiquer un champ d'un masque installé automatiquement pendant l'exécution du handler.
- un signal peut être **capté** en donnant l'adresse d'une fonction de traitement exécutée à la délivrance du signal ;

Environnement de signaux POSIX

Délivrance des signaux :
- le moment où le signal est délivré n'est pas contrôlable par le process ;
- un signal est **délivré** ("delivered") à un processus (le bit est mis à 0) quand le processus passe de l'état "actif noyau" à l'état "actif utilisateur" ;
  Ceci peut se produire :
  - au retour d'une interruption
  - au retour d'un appel système
  - lorsque le process est élu par l'ordonnanceur

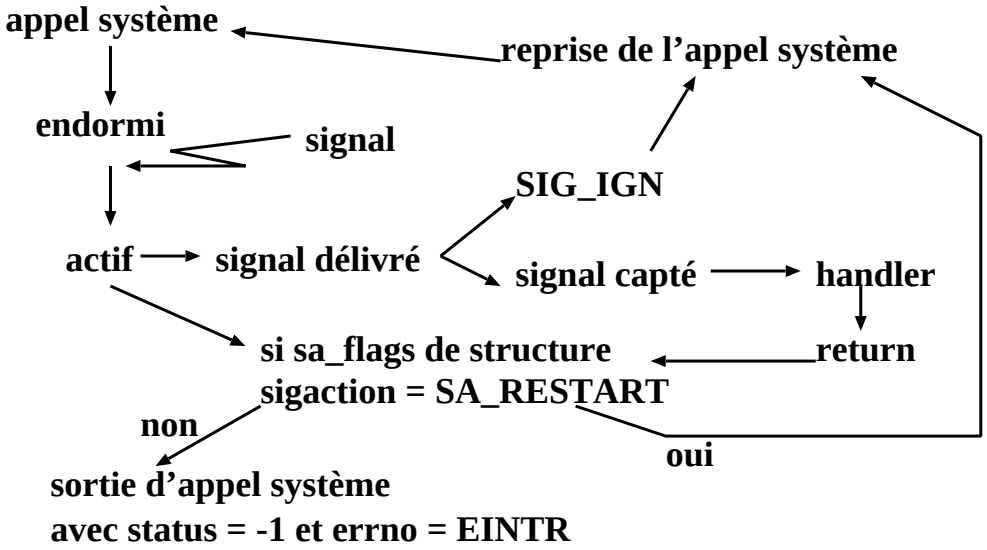
Environnement de signaux POSIX

Délivrance des signaux :
- quand un signal est envoyé à un processus endormi, celui-ci passe de l'état endormi à l'état "prêt", puis quand il est élu, le signal est délivré.

– ==> si le système est chargé, le process n'est pas élu tout de suite, et le délai entre l'arrivée du signal et sa prise en compte n'est pas contrôlable. C'est (entre autres) cela qui fait qu'un système unix "standard" n'est **pas temps réel**.

Environnement de signaux POSIX

Délivrance des signaux et appels systèmes



Environnement de signaux POSIX

Cas particuliers :

– **SIGHUP** à la fin d'un process shell, ce signal est envoyé à tous les process de la session, **sauf si** ils ont été lancés par :
  – nohup COMMAND [ARG]...
  – COMMAND [ARG]... &

  SIGSEGV envoyé si violation accès mémoire (peut être capté).
–
– **SIGALARM** permet la mise en place de "timeouts" : directive alarm() ou mieux setitimer()
– **SIGCHLD** envoyé par un fils qui se termine à son process parent. La capture de SIGCHLD se fait par les directives wait() ou waitpid().
  Un process serveur qui réponds aux requêtes par création de process doit faire attention de capter ces signaux pour éviter une prolifération de process "zombie".

Exemple de signaux : appel signal()

```
#include <signal.h>
#include <stdio.h>
int captint(), captquit();

main()
{       signal (SIGINT,captint);
        signal (SIGQUIT,captquit);
        pause();
```

```
        printf("--fin--\n");
}
captint()
{       printf("--dans captint ^C--\n");
        sleep(4);
        printf("--fin captint--\n");
}
captquit()
{       printf("--dans captquit ^\--\n");
        printf("--fin captquit--\n");
}
```

Exemple de signaux : appel signal() - exécution

```
224 gamma:~/> cc -o sig sig.c
225 gamma:~/> ./sig
^C--dans captint ^C--
--fin captint--
--fin--

227 gamma:~/> ./sig
^Z
Suspended
228 gamma:~/> fg
./sig
^C--dans captint ^C--
^\--dans captquit ^--
--fin captquit--
--fin captint--
--fin--
229 gamma:~/>
```

Les fonctions sigsetjmp() et siglongjmp() de POSIX

```
   #include <setjmp.h>
   int setjmp(jmp_buf env);
   int sigsetjmp(sigjmp_buf env, int savesigs);

DESCRIPTION
   setjmp()  and longjmp() are useful for dealing with errors
   and interrupts encountered in a low-level subroutine of  a
   program.   setjmp() saves the stack context/environment in
   env for later use by longjmp().  The stack context will be
   invalidated if the function which called setjmp() returns.

   sigsetjmp()  is  similar  to  setjmp().   If  savesigs  is
   nonzero,  the  set  of blocked signals is saved in env and
   will be restored if a siglongjmp() is later performed with
   this env.
```

sigsetjmp() et siglongjmp() rendent les mêmes services que setjmp() et longjmp(), mais en plus sauvegardent et restorent les masques de signaux.

setjmp() et longjmp() : exemple

```
#include <stdio.h>
#include <setjmp.h>
int main() {
   jmp_buf env;
   int n, val=5;

if((n=setjmp(env))==0)
/* setjmp retourne 0 quand elle ne revient pas d'un longjmp */
{   printf("Premier passage. setjmp retourne %d\n", n);
}
else
/* setjmp retourne "val" quand elle revient d'un longjmp appelé
   avec "val" en deuxieme argument */
{   printf("Deuxieme passage. setjump retourne %d\n", n);
    exit(0); /* sortir après le deuxième passage dans setjmp */
}
printf("Apres le setjmp, avant le longjmp\n");
longjmp(env, val);
printf("Apres le longjmp: normalement jamais atteint.\n");
}
```

2.4 Exemples de signaux unix. Mise en évidence des fonctionnements différents sous BSD, System V et POSIX.

signal interrompant un appel système

Dans l'exemple ci-dessous, le signal va interrompre l'exécution de l'appel système pause(). Celui-ci ressort avec le status EINTR.

signal_11.c montre aussi que par défaut sous Linux les handlers de capture déclarés sont réactivés automatiquement, mais que l'on peut donner au programme un comportement "System V" en ajoutant :
`#define _XOPEN_SOURCE  (résultat dans signal_11.txt)`

```
/* signal_11.c */
/* Montrer fonctionnement directive signal() traditionnelle */
/* UTC UV SR01 - (c) Michel.Vayssade@utc.fr oct 2003 */
/* gcc -o signal_11 signal_11.c */
/* montrer EINTR, mise à SIG_DFL */

#define _XOPEN_SOURCE /* donner à signal() comportement System V */
      /* !! à mettre AVANT include <signal.h> !! */
#include <signal.h>
#include <stdio.h>
#include <errno.h>

/* Declaration de variables globales de comptage */
int cp1=0, cp2=0;

/* fonction executee quand process fils capte un signal SIGINT */
void captSIGINT(int sig){
/* signal(SIGINT,captSIGINT); */
/* en Sys V : si pas réarmé, ne marche qu'une fois */
cp1++;
printf("Le process a capté un Sigint.\n");
printf("3 secondes de pause pendant lesquelles "\
"on peut générer une autre interruption.\n");
sleep(5);
/* la pause va nous permettre de provoquer une autre
   interruption pendant le traitement de la premiere.*/
printf("fin des 3 secondes de pause. Sortie de captSIGINT().\n");
/* permet de savoir quand le traitement  de l interruption
   va s'achever. */
```

```
}

/* ^\ provoque une sortie après la 3ème capture.*/
void captSIGQUIT(int sig){
/* signal(SIGQUIT,captSIGQUIT); */
/* en Sys V, si pas réarmé, ne marche qu'une fois */
cp2++;
printf("SIGQUIT capte par le process.\n");
if (cp2>=3)
{ printf("Le process a capté son troisieme et dernier SIGQUIT. Exit.\n");
  exit(10);
}
}

/* Programme principal */
main() {
  int n;
  signal(SIGINT,captSIGINT);
  signal(SIGQUIT,captSIGQUIT);

  printf("On est dans le process de pid: %d\n",getpid());
  while(1) {
    printf("a\n"); /* Boucle infinie */
    sleep(1);
    printf ("captsig errno=%d (EINTR=%d)\n",errno,EINTR);
  }
}
```

---

signal_11.txt : exécution sur Linux

```
$ gcc -o signal_11 signal_11.c

$ ./signal_11
On est dans le process de pid: 2733
a
captsig errno=0 (EINTR=4)
a
Le process a capté un Sigint.
3 secondes de pause pendant lesquelles on peut générer une autre interruption.
fin des 3 secondes de pause. Sortie de captSIGINT().
captsig errno=4 (EINTR=4)
a
captsig errno=4 (EINTR=4)
a
captsig errno=4 (EINTR=4)
a
Le process a capté un Sigint.
```

```
3 secondes de pause pendant lesquelles on peut générer une autre interruption.
SIGQUIT capte par le process.
fin des 3 secondes de pause. Sortie de captSIGINT().
captsig errno=4 (EINTR=4)
a
captsig errno=4 (EINTR=4)
a
captsig errno=4 (EINTR=4)
a
captsig errno=4 (EINTR=4)
a
SIGQUIT capte par le process.
captsig errno=4 (EINTR=4)
a
SIGQUIT capte par le process.
Le process a capté son troisieme et dernier SIGQUIT. Exit.

**** le signal est quand même capté plusieurs fois !?

--> man signal  sous Linux donne l'explication :

       The  original  Unix  signal()  would  reset the handler to
       SIG_DFL, and System V (and the Linux kernel  and  libc4,5)
       does  the same.  On the other hand, BSD does not reset the
       handler, but blocks new  instances  of  this  signal  from
       occurring  during  a  call  of  the  handler.   The glibc2
       library follows the BSD behaviour.


#define _XOPEN_SOURCE  /* donner à signal() le comportement System V */
/* !! à mettre AVANT include <signal.h> !! */
]$ gcc -o signal_11 signal_11.c

$ ./signal_11
On est dans le process de pid: 2832
a
captsig errno=0 (EINTR=4)
a
Le process a capté un Sigint.
3 secondes de pause pendant lesquelles on peut générer une autre interruption.

$ ./signal_11
On est dans le process de pid: 2833
a
captsig errno=0 (EINTR=4)
a
captsig errno=0 (EINTR=4)
```

```
a
captsig errno=0 (EINTR=4)
a
SIGQUIT capte par le process.
captsig errno=4 (EINTR=4)
a
captsig errno=4 (EINTR=4)
a
Quit
```

---

non fiabilité de la directive signal() System V

L'exemple suivant montre la non fiabilité de la directive signal() traditionnelle de System V : il y a risque de perte de signaux, et donc dans certains cas, de blocage infini du programme et, dans d'autres cas, d'arrêt intempestif.

```
/* signal_11b.c */
/* Montrer fonctionnement directive signal() traditionnelle */
/* Montrer qu'il rend les signaux non fiables */
/* UTC UV SR01 - (c) Michel.Vayssade@utc.fr oct 2003 */
/* gcc -o signal_11b signal_11b.c */

/*
C'est la combinaison de ces trois comportements qui rend les signaux
System V traditionnels non-fiables : comme les signaux peuvent être
déclenchés récursivement ET que la routine de traitement DOIT
ré-enclencher l'appel "manuellement", il y a une "fenêtre" pendant
laquelle le comportement par défaut peut être déclenché.
*/

#define _XOPEN_SOURCE /* donner à signal() comportement System V */
      /* !! à mettre AVANT include <signal.h> !! */
#include <signal.h>
#include <stdio.h>
#include <errno.h>

/* Declaration de variables globales de comptage */
int cp1=0, cp2=0;

/* fonction executee quand process fils capte un signal SIGINT */
void captSIGINT(int sig){
cp1++;
printf("Le process a capté un Sigint.\n");
printf("1 seconde de pause pendant laquelle "\
"on peut générer une autre interruption.\n"\
"avant d'avoir pu ré-armer la capture\n");
```

```
sleep(1);
signal(SIGINT,captSIGINT); /* si pas réarmé, marche qu'une fois */
printf("3 secondes de pause pendant lesquelles "\
"on peut générer une autre interruption.\n"\
"mais cette fois la capture est ré-armée\n");
sleep(3);
/* la pause va nous permettre de provoquer une autre
interruption pendant le traitement de la premiere.*/
printf("fin des 3 secondes de pause. Sortie de captSIGINT().\n");
}

/* ^\ provoque une sortie après la 3ème capture.*/
void captSIGQUIT(int sig){
/* signal(SIGQUIT,captSIGQUIT); si pas réarmé, marche qu'une fois */
cp2++;
printf("SIGQUIT capte par le process.\n");
if (cp2>=3)
{ printf("Le process a capté son troisieme et dernier SIGQUIT. Exit.\n");
  exit(10);
}
}


/* Programme principal */
main() {
  int n;
  signal(SIGINT,captSIGINT);
  signal(SIGQUIT,captSIGQUIT);

  printf("On est dans le process de pid: %d\n",getpid());
  while(1) {
    printf("a\n"); /* Boucle infinie */
    sleep(1);
    printf ("captsig errno=%d (EINTR=%d)\n",errno,EINTR);
  }
}
```

---

signal_11b.txt

fonctionnement directive signal() traditionnelle
fonctionnement rendu non fiable par la conjonction
du rétablissement du comportement par défaut lors du
traitement d'un signal ET de la possibilité pour un
signal d'interrompre son propre handler

```
$ gcc -o signal_11b signal_11b.c
$ ./signal_11b
On est dans le process de pid: 4674
a
captsig errno=0 (EINTR=4)
a
captsig errno=0 (EINTR=4)
a
captsig errno=0 (EINTR=4)
a
Le process a capté un Sigint.
1 seconde de pause pendant laquelle on peut générer une autre interruption.
avant d'avoir pu ré-armer la capture
3 secondes de pause pendant lesquelles on peut générer une autre interruption.
mais cette fois la capture est ré-armée
Le process a capté un Sigint.
1 seconde de pause pendant laquelle on peut générer une autre interruption.
avant d'avoir pu ré-armer la capture
3 secondes de pause pendant lesquelles on peut générer une autre interruption.
mais cette fois la capture est ré-armée
fin des 3 secondes de pause. Sortie de captSIGINT().
fin des 3 secondes de pause. Sortie de captSIGINT().
captsig errno=4 (EINTR=4)
a
Le process a capté un Sigint.
1 seconde de pause pendant laquelle on peut générer une autre interruption.
avant d'avoir pu ré-armer la capture

****** ici le deuxième signal arrivant avant le réarmement
****** provoque l'arrêt du programme, car le comportement
****** par défaut a été rétabli par le traitement du 1er signal
$
```

```
/*--------------------Test des sigaux----------------------------*/
/* int100.c : fils envoi 100 sigint au père */

#include <stdio.h>
#include <signal.h>
#include <errno.h>

int recu=0;

void captint(int sig)
{ recu++;
  printf ("capte signal %d recu=%d\n",sig,recu);
 }
```

```
main()
{ int i=0,n=0;

signal(SIGINT,captint);
printf("capter ctrl-C\n");
i = fork();
if (i==-1) { printf("echec fork\n"); exit(0);}
if (i==0) { /* fils */
while (n<100)
{ printf("boucle envoi SIGINT n=%d\n",n);
kill (getppid(), SIGINT);
n++;
if (n%10==0) { /* wait tous les 10 */
printf("---n=%d\n",n); usleep(1);
}
}
}
else { /* pere */
waitfils:
n = wait();
if (n==-1 && errno==EINTR) goto waitfils;
exit(0);

}/* fin if else */
}/* fin main */

/* 100 000 usec = 0.1s tous reçus */
/* 0 => reçu 1 */
/* 1 000 usec = 0.001s => reçu tous */
/* 1 usec => reçu tous */
/* if (n%10==0) { printf("---n=%d\n",n); usleep(1) => reçu 55 */
```

liste des signaux

La liste des signaux : sous Linux man 7 signal

```
SIGNAL(7)           Linux Programmer's Manual           SIGNAL(7)

NAME
       signal - list of available signals

DESCRIPTION
       Linux  supports  the  signals listed below. Several signal
       numbers are  architecture  dependent.  First  the  signals
       described in POSIX.1.

       Signal     Value     Action   Comment
```

```
      ------------------------------------------------------------------
      SIGHUP        1        A      Hangup detected on controlling terminal
                                    or death of controlling process
      SIGINT        2        A      Interrupt from keyboard
      SIGQUIT       3        C      Quit from keyboard
   ......................
```

fonctionnement asynchrone du signal

L'exemple suivant montre le fonctionnement asynchrone des signaux :

```
/* signal_12.c */
/* Montrer fonctionnement asynchrone des signaux */
/* UTC UV SR01 - (c) Michel.Vayssade@utc.fr oct 2003 */
/* gcc -o signal_12 millidelay.o signal_12.c */

#include <signal.h>
#include <stdio.h>
#include <errno.h>

/* Declaration de variables globales de comptage */
int cp1=0, cp2=0, ni=0;

/* fonction executee quand process fils capte un signal SIGINT */
void captSIGINT(int sig){
/* signal(SIGINT,captSIGINT);  */
        /* en Sys V : si pas réarmé, ne marche qu'une fois */
cp1++;
printf("dans captSIGINT ni=%d\n",ni);
}

/* ^\ provoque une sortie après la 3ème capture.*/
void captSIGQUIT(int sig){
/* signal(SIGQUIT,captSIGQUIT);  */
cp2++;
printf("SIGQUIT capte par le process.\n");
if (cp2>=3)
{ printf("Le process a capté son troisieme et dernier SIGQUIT. Exit.\n");
  exit(10); }
}

/* Programme principal */
main() {
  int n;
  signal(SIGINT,captSIGINT);
  signal(SIGQUIT,captSIGQUIT);
```

```
  printf("On est dans le process de pid: %d\n",getpid());
  while(1) {
    ni = 1;
    printf("a1\n"); /* Boucle infinie */
    millidelay(100); /* 100 millisec = 0.1 sec */
    ni++;
    printf("a1\n"); /* Boucle infinie */
    millidelay(100); /* 100 millisec = 0.1 sec */
    ni++;
    printf("a1\n"); /* Boucle infinie */
    millidelay(100); /* 100 millisec = 0.1 sec */
    ni++;
    printf("a1\n"); /* Boucle infinie */
    millidelay(100); /* 100 millisec = 0.1 sec */
    ni++;
    printf("a1\n"); /* Boucle infinie */
    millidelay(100); /* 100 millisec = 0.1 sec */
  }
}
```

```
$ ./signal_12
On est dans le process de pid: 2927
a1
a1
dans captSIGINT ni=1
a1
a1
dans captSIGINT ni=3
a1
a1
a1
dans captSIGINT ni=1
a1
a1
a1
dans captSIGINT ni=4
a1
a1
a1
dans captSIGINT ni=2
a1
a1
a1
dans captSIGINT ni=5
a1
```

SR01 2003 - Cours Unix 2 - Les signaux sous unix

2.5 Exemples de gestion de signaux par sigaction() et exemples setjmp()

exemple avec appel système sigaction()

```
struct sigaction nact,oact;  /* new act , old act */
nact.sa_handler = handler;
sigemptyset(&nact.sa_mask);
nact.sa_flags = 0;
i = sigaction (sig,&nact,&oact);
--> signact.c + sigact.c

nact.sa_mask = ensemble des signaux bloqués pendant exécution du handler
le signal déclencheur est automatiquement bloqué, sauf si on demande le
contraire par SA_NODEFER dans nact.sa_flags --> sigact-sysV.c
```

exemple avec appel système sigaction() –> signact.c

```
/*--------------------Test des sigaux----------------------------*/
/* signact.c : utilise sigact() qui appelle sigaction()
 * > gcc -c sigact.c
 * > gcc -o signact sigact.o signact.c
 */
/* (c) Michel.Vayssade@utc.fr oct 2003 */

#include <signal.h>  /* définition de SIGINT, SIGQUIT */

int tab[20], max, itab;

void captint(int sig)
{ printf ("captint capte signal %d \n",sig);
  sleep(6);
  printf ("fin captint\n");
 }
void captquit(int sig)
{ printf ("captquit capte signal %d \n",sig);
  sleep(4);
  printf ("fin captquit\n");
 }

main() {
```

```
    int i=0;
    sigact(SIGINT,captint);
    sigact(SIGQUIT,captquit);
    i=0;
    printf("capter ctrl-C et ctrl-\\\n");
    while (i<4) {
         printf("boucle i=0,3 i=%d\n",i);
         sleep(2);
         i++;
    }
    exit(0);
}
```

exemple avec appel système sigaction() –> sigact.c

```
/* sigact.c : simule signal() de BSD avec sigaction()
 * > gcc -c sigact.c
 * > gcc -o signact sigact.o signact.c
 * un signal x n'interrompt pas le handler d'un x précédent
 */

#include <errno.h>
#include <signal.h>

sigact(int sig, void *handler)
{  int i;
   struct sigaction nact,oact;  /* new act , old act */

   nact.sa_handler = handler;
   sigemptyset(&nact.sa_mask);
   nact.sa_flags = 0;
   i = sigaction (sig,&nact,&oact);
   if (i==-1) perror ("sigact: echec sigaction\n");
   if (i!=0) perror ("sigact: should not happen\n");
   return;
}
```

exemple avec l'appel système setjmp() de saut non local

```
/* setjmp.c */
/* Montrer fonctionnement setjmp/longjmp */
/* UTC UV SR01 - (c) Michel.Vayssade@utc.fr oct 2003 */
/* gcc -o setjmp setjmp.c */

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

```
int main()
{
jmp_buf env;
int n;

if((n=setjmp(env))==0)
/* setjmp retourne 0 quand elle ne revient pas d'un longjmp */
{
        printf("Premier passage. setjmp retourne %d\n", n);
}
else
/* setjmp retourne 1 quand elle revient d'un longjmp appelé
   avec 0 en deuxieme argument */
{

        printf("Deuxieme passage. setjump retourne %d\n", n);
        exit(0); /* sortie apres deuxieme passage dans setjmp */
}

printf("Apres le setjmp, avant le longjmp\n");

longjmp(env, 2); /* essayer avec 0,1,2 en 2ème argument */
/* setjmp renvoie 1 quand le deuxieme argument passe a longjmp */
/* est zero , sinon setjmp renvoie la valeur de ce 2ème arg */

printf("Apres le longjmp: normalement jamais atteint.\n");
}
```

---

setjmp() et les signaux

```
ex-setjmp.c  la capture de ^C ne fonctionne qu'une fois,
après cela, le signal reste bloqué car longjmp()
ne rétablit pas le masque de signaux avant de
sortir du handler
ex-sigsetjmp.c la capture de ^C est rétablie après l'appel
siglongjmp()
```

```
/*---------------- Test sigaux et setjmp -------------------*/
/* ex-setjmp.c  montre interaction à problème setjmp<-->signal
 * > gcc -c sigact.c
 * > gcc -o ex-setjmp sigact.o ex-setjmp.c
 */
/* (c) Michel.Vayssade@utc.fr oct 2003 */
```

```
#include <signal.h>  /* définition de SIGINT, SIGQUIT */
#include <setjmp.h>

int tab[20], max, itab, ibcle=0;
jmp_buf env;

void captint(int sig)
{ printf ("captint capte signal %d ibcle=%d\n",sig,ibcle);
  sleep(3);
  printf ("fin captint\n");
  longjmp(env, ibcle);
}
void captquit(int sig)
{ printf ("captquit capte signal %d \n",sig);
  sleep(4);
  printf ("fin captquit\n");
 }

main() {
    int i=0, k=0, n;
    sigact(SIGINT,captint);
    sigact(SIGQUIT,captquit);

    i=0;
    printf("capter ctrl-C et ctrl-\\\n");
    while (ibcle<10) {
if (k==0) {
     k = 1;
   if((n=setjmp(env))==0) {
   /* setjmp retourne 0 quand ne revient pas d'un longjmp */
       printf("Premier passage. setjmp retourne %d\n", n);
   } else {
   /* setjmp retourne 1 quand elle revient d'un longjmp */
   /* appele avec 0 en deuxieme argument */
       printf("Deuxieme passage. setjump retourne %d\n", n);
   }
}
         printf("boucle i=0,3 i=%d\n",ibcle);
         sleep(1);
         ibcle++;
    }
    exit(0);
}
```

```
/*---------------- Test sigaux et sigsetjmp -------------------*/
```

```
/* ex-sigsetjmp.c  montre interaction correcte sigsetjmp<-->signal
 * > gcc -c sigact.c
 * > gcc -o ex-sigsetjmp sigact.o ex-sigsetjmp.c
 */
/* (c) Michel.Vayssade@utc.fr oct 2003 */

#include <signal.h>  /* définition de SIGINT, SIGQUIT */
#include <setjmp.h>

int tab[20], max, itab, ibcle=0;
jmp_buf env;

void captint(int sig)
{ printf ("captint capte signal %d ibcle=%d\n",sig,ibcle);
  sleep(3);
  printf ("fin captint\n");
  siglongjmp(env, ibcle);
}
void captquit(int sig)
{ printf ("captquit capte signal %d \n",sig);
  sleep(4);
  printf ("fin captquit\n");
 }

main() {
    int i=0, k=0, n;
    sigact(SIGINT,captint);
    sigact(SIGQUIT,captquit);

    i=0;
    printf("capter ctrl-C et ctrl-\\\n");
    while (ibcle<10) {
if (k==0) {
     k = 1;
   if((n=sigsetjmp(env,1))==0) {
   /* setjmp retourne 0 quand ne revient pas d'un longjmp */
       printf("Premier passage. setjmp retourne %d\n", n);
   } else {
   /* setjmp retourne 1 quand elle revient d'un longjmp */
   /* appele avec 0 en deuxieme argument */
       printf("Deuxieme passage. setjump retourne %d\n", n);
   }
}
         printf("boucle i=0,3 i=%d\n",ibcle);
         sleep(1);
         ibcle++;
    }
```

```
    exit(0);
}
```

exemple avec programme graphique

le process père capte signal et le renvoie à son fils qui le capte lui aussi

```
/* sig2.c = f2.c exemple fork+X11 + gestion signaux */
/* appelle routines definies dans fx.c
 * UTC UV SR01 - (c) Michel.Vayssade@utc.fr oct 2003
 * > gcc -c fx.c
 * > gcc -o sig2 sig2.c fx.o -lX11    (solaris)
 * > gcc -o sig2 sig2.c fx.o -L/usr/X11R6/lib -lX11  (linux)
 */
/* main fork un fils qui : initialise rectangle rouge,
 * attends dans attendreclic();
 * Lors d'un clic dans bouton en couleur, "toggle" couleurs.
 * Sort si clic dans bouton fin.
 */
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <signal.h>

int nbre_int = 0, max_int=3, pidfils;

#define lng 40
char buf[lng];

int liretty (char *prompt, char *buffer)
{ int i;
printf("\n%s",prompt);
i = scanf ("%s",buffer);
return strlen(buffer);
}

void captintfils()
{       printf("--dans captint ^C du fils--\n");
        rectvert(2);    /* rectangle en vert pendant 2 secondes */

        nbre_int++;
        printf("--captint fils nbre_int=%d--\n",nbre_int);
        if (nbre_int>=max_int) {
        printf("--fin du fils--\n");
                detruitrec();   /* detruire la fenetre rectangle */
                exit(0);
        }
```

```
        printf("--fin captint fils--\n");
}

void captint()
{       printf("--dans captint ^C du pere--\n");
/*        kill (pidfils,SIGINT); */
        nbre_int++;
        printf("--captint pere nbre_int=%d--\n",nbre_int);
        if (nbre_int>=max_int) {
        printf("--fin du pere--\n");
                wait();
                exit(0);
        }
        printf("--fin captint pere--\n");
}

main ()
{  int i,nlu;
   struct sigaction nact,oact;  /* new act , old act */
   struct sigaction qact;

pidfils = fork();
if (pidfils==-1) {perror("echec fork"); exit(0);}

if (pidfils==0) {  /* fils */

nact.sa_handler = captintfils; /* establish SIGINT handler (cntrl-C) */
sigemptyset(&nact.sa_mask); nact.sa_flags = 0;
i = sigaction (SIGINT,&nact,&oact);
if (i==-1) perror ("echec sigaction SIGINT\n");
if (i!=0) perror ("should not happen\n");

initrec(); /* creer rectangle rouge */
do {
i = attendreclic ();
printf("-sort attendreclic i=%d\n",i);
if (i == 0) fprintf (stderr, "-- clic dans fin --\n");
if (i == 1) {
nlu = liretty("fils_>",buf);
ecritrec (buf,nlu);
}
} while (i==1);

        printf("--fin du fils--\n");
detruitrec(); /* detruire la fenetre rectangle */
exit(EXIT_SUCCESS);
} /* fin fils */
```

```
else  { /* pere */

nact.sa_handler = captint;  /* establish SIGINT handler (cntrl-C) */
sigemptyset(&nact.sa_mask); nact.sa_flags = 0;
i = sigaction (SIGINT,&nact,&oact);
if (i==-1) perror ("echec sigaction SIGINT\n");
if (i!=0) perror ("should not happen\n");

do {
printf("pere dort 10s.\n");
         sleep (10);
nlu = liretty("pere_>",buf);
printf("%d lu par pere:%s\n",nlu,buf);
} while (buf[0]!='f');

        printf("--fin du pere--attendre fin du fils--\n");
wait();
exit(EXIT_SUCCESS);
} /* fin pere */
}
```

SR01 2003 - Cours Unix 3 - Les fichiers sous unix

3 SR01 2003 - Cours Unix 3 - Les fichiers sous unix

3.1 Le système de fichiers unix
Organisation des fichiers : arborescence, répertoire

Organisation des fichiers : arborescence, répertoire

arborescence, sommet = /, noeuds = répertoires, feuilles = fichiers

répertoire : représenté par un fichier qui contient une liste d'entrées chaque entrée représente un fichier (un sous-répertoire ou un fichier ordinaire)

chaque répertoire contient 2 entrées spéciales : "." et ".." représente le catalogue de login de l'utilisateur

nom absolu : /home/sr01/sr01001/td1/test.c nom relatif : ./test.c ../td1/td2.c

Cas particulier u*x : fichiers cachés : commençant par un "." option -a pour les voir : ls -a

Utilisation du "/"

```
! Attention le / a DEUX fonctions : c'est à la fois un séparateur de
répertoire (ex: dupond/bench/fichier) ET le nom du répertoire racine

   Quand il apparait en PREMIER, il indique une destination ABSOLUE.
   Ex:     cd /usr          | crée             /
           mkdir /util      |          +---------------+
                                     usr             util
   Alors que:
           cd /usr          | crée           /
           mkdir util       |              usr
                                           util
```

Les caractères de groupement (wildcard)

```
   ** Pas de règle absolue. Le résultat de l'usage de wildcard DÉPENDS
    de la commande.
           ls *    -> ts les fichiers dans la dir courrante PLUS
                      ts ceux 1 niveau en-dessous
           wc *    -> ne traite QUE les fichiers de la dir cour.
```

```
          ls .    -> donne ts les fichiers pointées par . (donc ts
                  ceux de la dir courrante)
          wc .    traite le fichier . lui-même

  u*x     ls *.*  -> ts les fichiers dont le nom contient 1 "."
                  (ex: le fichier "truc" n'est pas donné)

  ls, *, et fichiers cachés:  ! -a et * ne font pas bon ménage !
          > ls -a
          .joverc  myjoverc
          > ls -a *jov*           #! ne montre que +
          myjoverc                        <--------+
          > ls -a .*jov*
          .joverc
          > ls -a .*jov* *jov*    #! pour voir les 2
          .joverc  myjoverc
          > ls .*jov* *jov*      #! idem ! -a et * pas bon ménage !
          .joverc  myjoverc

  ls -a | grep "^\."      lister les seuls fichiers commencant par "."
  ls -ad .*               idem ("d" empêche d'explorer les dirs)
```

---

ensemble des répertoires connus

```
$ ls -F /
bin/   etc/    initrd/      misc/  root/  usr/
boot/  lib/    mnt/         opt/   sbin/  var/
dev/   home/   lost+found/  proc/  tmp/

$ ls -F /usr
bin/   games/               kerberos/  local/       share/  X11R6/
dict/  i386-glibc21-linux/  lib/       lost+found/  src/
etc/   include/             libexec/   sbin/        tmp@
```

```
    /bin    executables system les + utilisés
    /dev    fichiers spéciaux d'accès aux prériphériques
    /etc    fichiers divers (groups, passwd, rc.local,...) admin système
    /lib    bibliothèques partageables (.so)
    /tmp    fichiers temporaires
    /usr/bin     exécutables moins fréqu. utilisés
    /usr/lib     bibliothèques modules objets (.a)
    /usr/share/     logiciels, applications
    /usr/share/man/ man1 ... manuels en ligne
    /usr/local      installations logiciels optionnels
    /var/spool      fichiers mis en queue
```

les fichiers de login

```
avec csh/tcsh
             /etc/csh.cshrc  /etc/csh.login
     puis    ~/.cshrc        si .tcshrc existe pas
             ~/.login        si login

avec bash
             /etc/bashrc     /etc/profile
     puis
             .bashrc         .bash_profile

     définissent :
             variables d'environnement       printenv
             variables internes du shell     set
             alias                           alias

     répertoires de manuel:
             /usr/man/man1   commandes shell
             /usr/man/man2   appels système
             /usr/man/man3   bibliothèques de subroutines
             /usr/man/man4   description i/o dev. drivers
             /usr/man/man5   formats et include files
             /usr/man/man6   jeux
             /usr/man/man7   fichiers spéciaux
             /usr/man/man8   procédures system
             /usr/man/manl   fichiers locaux ajoutés

     Options utiles du man
     ---------------------
     > man -f cmde   -> description brève (synonyme: whatis)
     > man -k chaine -> synonyme: apropos
                     donne ttes les cmdes dans le man desquelles
                     on trouve "chaine" (grep chaine /man*/*)
     > man 2 time    -> force la recherche en section 2
                     affiche la fonction time; sinon affiche la
                     commande time en section 1

     pour trouver si une fonction de même nom qu'une commande existe:
     $ apropos time | grep ^time
```

3.2 Types de fichiers dans un système de fichiers unix

```
répertoire ("directory")

fichier normal ("ordinary file" ou "regular file")

lien symbolique ("symbolic link")
        pointeur sur un autre fichier
        créé par ln -s nom_du_fichier nom_du_lien
       ln [OPTION]... TARGET [LINK_NAME]

fichier spécial ("special files")
        associé à un pilote de périphérique
        quand on agit sur ce "fichier", on agit en fait sur le
périphérique
        (par exemple /dev/ttys0 accède au port série Nř0)

        $ tty   montre le tty du shell courant
        $ stty  montre options actives du terminal courant


tube nommé ("named pipe")
        canal de communication pouvant être utilisé pour échanger des
        données entre deux processus de la même machine

socket
        canal de communication pouvant être utilisé pour échanger des
        données entre deux processus de la même machine OU situés sur
        des machines différentes reliées en réseau par TCP/IP
```

3.3 Commandes et appels systèmes relatifs à la gestion des répertoires et des fichiers

Commandes et appels systèmes relatifs à la gestion des répertoires et des fichiers

```
        mkdir rmdir dircolors
        ln -s target link_name
        cp  cp -p  cp -r  mv
```

```
install - copy files and set attributes
rm  shred
df  du  sync

opendir, readdir, closedir, rewinddir, seekdir

tree - list contents of directories in a tree-like format.
tree -d
ftw - file tree walk
find  grep

dans "ls" standard pas distinction entre les ss-dirs et fichiers
dir *.dir       -> ls -F        ajoute un / derrière chq dir
                ou ls -l | grep "^d"

l'utilitaire "file" de u*x permet de connaître le type d'un
fichier:        % file prog     -> executable file

> df -i ! donne infos sur i-nodes
```

Lister les fichiers dans un répertoire

```
        > ls -lg                contenu du répert. courrant

        > ls -lg
-rw-r--r--  1 vayssade users        9071 Nov 29 11:09 myjoverc
doooggggwww  ^   owner  group      taille date dernière   nom
prot.(rwx)  |                      octets modification
            +--> nombre de liens sur ce fichier
                 minimum 1 : vers la dir qui le contient
                 une dir n+2: n = nbre de dirs qu'elle contient
1er car.                (faire ls -aF dir et compter les /)
type:   -       fichier
        d       directory
        s       socket
        b       block-type special file
        c       charac-type special file
        |       lien symbolique (ln -s ..)
```

Lister les fichiers dans un arbre de répertoires

```
        > ls -R         à partir du répert. par défaut
                        liste TOUT l'arbre

        > ls *          montre LA dir. courrant + 1 niveau
        > ls */*        montre LA dir. courrant + 2 niveaux

Autres options de ls
--------------------
ls -sF  ---> s donne taille en Ko de chq fichier
        ---> F fait suivre chaque dir d'un "\"

Une arborescence peut également être visualisée de manière semi-
graphique par l'utilitaire "tree"
```

Le "wildcarding" dans les noms de fichiers

```
Attention:
- chaque commande U*X a sa propre façon d'interpréter les
caractères de groupement (*, ?).
- de plus cette interprétation peut varier selon les options
d'une même commande.
- le caractère "." n'a pas de signification particulière, c'est
un caractère comme les autres.

        *       signifie n'importe quelle chaine
        ?                n'importe quel caractère
        [c_deb-c_fin]   signifie n'importe quel caractère
                        contenu entre c_deb et c_fin

        * peut etre utilisé dans les noms de catalogues.

    Exemples :
    ----------
ls -l *.f
ls -l a?e.f
ls -l toto.?
ls -l /usr/*/*.f
```

Copie d'arborescence

```
> cp -p -r ./ssdir /usr2/truc/ssdir
        p préserve les dates et protections
        r récursivement le sous-arbre sous ssdir
        /ssdir est créée dans /usr2/truc et le sous arbre copié
$ backup [user1...] [user2.user1...]
> cp -r /user1 /user2
                AVANT                    APRÈS
        /user1          /user2                  /user2
          |               |                       |
---+----+----+---     --+----+---       --+----+-------+------
  tmp  bin  srce       truc trav         truc trav    user1
                                                       |
                                                   ---+----+----+---
                                                     tmp  bin  srce

arbres /user1 et /user2 demeurent intactes, /user1 est dupliqué
sous /user2/user1
```

Exemples d'utilisation de find

```
# find / -name 'Xt*' -print

cherche tous les fichiers dont le nom matche Xt* dans
toute l'arborescence sous / et imprime leur nom complet
avec le chemin (1 ligne par occurence)

# find /usr \( -name a.out -o -name '*.o' \) -exec ls -l {} \;

attention au blanc entre {} et \;
\( et \) encadrent x -o y avec x et y conditions et -o = OU
et !x = non x
-exec indique que la commande qui suit (ici ls -l) est a faire
sur chaque fichier qui satisfait les conditions ({} représente
le paramètre de la commande remplaçé par le fichier trouvé).

# find /dir/ssdir -type d -print        ! les dirs du ss-arbre

# find /dir/ssdir -type f -print        ! les fichiers standards

# find /user2/truc -name "*exe*" -print ! donne ts fic.avec *exe*

# find /usr1 -mtime +30 -print  ! donne nom de tous les fichiers
qui n'ont pas été modifiés pendant les 30 derniers jours
(mtime: last modified, +30 il y plus de 30 jours)
```

```
# find /usr1/dupond -name '*.f' -atime -30 -print
donne nom des fichiers de arborescence de dupond qui se terminent
par .f et qui ONT ÉTÉ accédés dans les 30 derniers jours
(atime: last accessed, -30 dans les derniers 30 jours)

# find /user1/dupond -name '*.f' -o (-mtime 3 -atime 6) -exec rm {} \;
cherche les fichiers finissant par .f, modifiés dans les 3 derniers
jours OU accédés dans les 6 derniers, fait rm de chq fichier qui
répond au critère
```

Chercher une chaine dans des fichiers

```
u*x     > grep 'chaine de caractères' fichier
grep:   case sensitive par défaut

> grep -i 'chaine' fichier      ! case insensitive
 (avec alias grep 'grep -i' peut devenir le défaut)

> grep -n chaine file   ! affiche en plus le numéro de ligne
23:il y a "chaine" dans cette ligne

> grep -c chaine file   ! affiche le compte de lignes
4                       ! 4 lignes contiennent chaine
> grep -v chaine file   ! affiche ttes les lignes SAUF celles
                        ! qui contiennent chaine
> grep chaine$ file     ! lignes se terminant par chaine
> grep "^chaine" file   ! lignes commençant par chaine
```

3.4 Liens sur les fichiers

```
Une partie du path name ou chemin d'accès peut être remplacée par
une sorte de nom logique appelé sous U*X lien symbolique.

Exemple: on a défini le lien symbolique "user" équivalent à
         /usr/users
cd /usr/users           peut alors s'écrire     cd /user
mais pwd                donne toujours          /usr/users
```

En fait sous unix on peut créer plusieurs liens vers un même fichier. Le fichier est alors accessible par plusieurs noms différents. Le système maintient un compteur du nombre de liens vers chaque fichier. Quand le dernier lien est détruit, alors le fichier est effectivement détruit. Le compteur du nombre de liens est le deuxième champ affiché par la commande "ls -l" :

```
[vayssade@virgo poly_c_es]$ echo $PS1
[\u@\h \W]\$
[vayssade@virgo poly_c_es]$ PS1="\\$ "
$ cat <<fin > aa
> abc def
> ghi jkl
> fin
$ cat aa
abc def
ghi jkl
$ ls -l aa
-rw-r--r--    1 vayssade users          16 Nov  7 10:40 aa
$ ln aa bb
$ ls -l aa bb
-rw-r--r--    2 vayssade users          16 Nov  7 10:40 aa
-rw-r--r--    2 vayssade users          16 Nov  7 10:40 bb
$ rm aa
$ ls -l bb
-rw-r--r--    1 vayssade users          16 Nov  7 10:40 bb
$ cat bb
abc def
ghi jkl
$ rm bb
$
```

```
Restrictions:
il est impossible de créer des liens :
 - sur des répertoires (arbre->graphe->cycles)
 - entre fichiers sur des systèmes de fichiers différents
   (partitions différentes)

Restrictions levées pour les liens symboliques.
Le noyau ne "suit pas ces liens" lors d'opérations récursives sur les
arborescences.
```

3.5 Les attributs des fichiers

Lister les attributs

```
Sur Linux (module stat installé) :
$ stat fopwrcl.c
  File: "fopwrcl.c"
```

```
  Size: 855        Blocks: 8         Regular File
Access: (0644/-rw-r--r--)   Uid: ( 100/vayssade)  Gid: ( 100/  users)
Device: 8          Inode: 432437     Links: 1
Access: Tue Nov  4 15:52:38 2003
Modify: Fri Oct 31 15:40:53 2003
Change: Mon Nov  3 08:08:54 2003

Sur Solaris pas de commande toute faite.
man stat
NAME
     stat, lstat, fstat - get file status

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

     int stat(const char *path, struct stat *buf);


/* mystat.c
 * gcc -o mystat mystat.c
 */

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

main(int argc, char **argv)
{   char *nom;
    int i;
    struct stat buf;
    char *buft;

    if(argc != 2) {
        printf("Usage: %s <file>\n",argv[0]);
        exit(0);
    }
    nom = argv[1];
    printf("status of %s :\n",nom);

    i= stat(nom, &buf);
    printf("mode=%o ino=%d  dev=%d devspe=%d nlink=%d\n",
     buf.st_mode, buf.st_ino, buf.st_dev, buf.st_rdev, buf.st_nlink);
    printf("uid=%d gid=%d size=%d blocks=%d blksize=%d \n",
     buf.st_uid, buf.st_gid, buf.st_size, buf.st_blocks, buf.st_blksize);
    printf("atime=%d mtime=%d ctime=%d \n",
     buf.st_atime, buf.st_mtime, buf.st_ctime);

    buft=(char *)ctime(&buf.st_atime);
```

```
    printf("atime=%s",buft );
    printf("mtime=%s", ctime(&buf.st_mtime));
    printf("ctime=%s", ctime(&buf.st_ctime));
    if (S_ISDIR(buf.st_mode)) printf("%s is DIR\n",nom);
    if (S_ISREG(buf.st_mode)) printf("%s is REG\n",nom);
}
```

```
$ gcc -o mystat mystat.cmystat mystat.c
$ ./mystat mystat.c
status of mystat.c :
mode=100644 ino=432448  dev=8 devspe=0 nlink=0
uid=100 gid=100 size=986 blocks=8 blksize=4096
atime=1068206754 mtime=1068206752 ctime=1068206752
atime=Fri Nov  7 13:05:54 2003
mtime=Fri Nov  7 13:05:52 2003
ctime=Fri Nov  7 13:05:52 2003
mystat.c is REG
$ ./mystat .
status of . :
mode=40755 ino=432434  dev=8 devspe=0 nlink=0
uid=100 gid=100 size=4096 blocks=8 blksize=4096
atime=1068199177 mtime=1068206754 ctime=1068206754
atime=Fri Nov  7 10:59:37 2003
mtime=Fri Nov  7 13:05:54 2003
ctime=Fri Nov  7 13:05:54 2003
. is DIR
$
```

Les dates associées à un fichier

```
Dates
    Access: Tue Nov  4 15:52:38 2003
    Modify: Fri Oct 31 15:40:53 2003
    Change: Mon Nov  3 08:08:54 2003

modifiables par utime(2)
```

Propriétaire, groupe et protections. Les protections associées aux fichiers

---

```
Les fichiers ont trois niveaux de protection:
user ( OWNER ), group ( GROUP ) et other ( WORLD ).
Visualisation par ls -l
-rwx--x--- 1 dupond 23 23-oct-90 11:35 nom_fichier
TOOOGGGWWW
T = type (=d pour un répertoire, =- pour fichier normal)
```

```
OOO = protections du Owner, GGG du Groupe, WWW du World.
Le niveau système n'apparait pas car il a en permanence tous les
droits .

La modification de protection d'un fichier se fait à l'aide
de la commande:
        chmod   masque_protection nom_fichier
où masque_protection est sous la forme ci-dessous :

Exemple syntaxe :
        chmod u+x file
        chmod u+rw,g-rwx,o-rwx file  ou chmod 600 file

La mise d'un fichier à une protection standard est :

% chmod u=rwx,g=x,o= nom_fichier
--> affecte ce masque de priorité au fichier sans tenir compte
    des antécédents:
-rwx--x--- 1 dupond 14 23-oct-90 11:35 nom_fichier

% chmod g+r,o+x nom_fichier   --> ajoute à group le mode read et
                                  aux autres le mode éxécute:
-rwxr-x--x 1 dupond 14 23-oct-90 11:35 nom_fichier
```

Les bits de protections et permissions

```
      {[who]signe[permission],}
     /      |               \
u user      + ajouter             r read
g group     - oter                w write
o other     = assignation         x execute
a all         absolue (remplace   s "set"
              l'ancien masque)    t save text

"set" = le proprietaire du fichier peut changer le
        owner_id ou group_id des fichiers


         ||  owner  |  group  | others
+-++-++-+||+-++-++-+|+-++-++-+|+-++-++-+
|4||2||1||||4||2||1|||4||2||1|||4||2||1|
+-++-++-+||+-++-++-+|+-++-++-+|+-++-++-+
 su sg s || r  w  x | r  w  x | r  w  x
         ||         |         |
\---v---/
```

```
affect only executable files (programs) and,
on some systems, directories:

su = Set-UID bit :      permet à un exécutable de modifier
                        l'UID effectif du process; permet
                        de donnerà un user la possibilité
                        de faire des actions réservées à root
sg = Set-GID bit :      permet à un exécutable de modifier
                        le GID effectif du process; permet
                        d'accéder à des fichiers en fonction
                        de la protection "groupe"
s = "sticky bit"        save the program's text image on the
                        swap device so it will load more quickly
                        when run
For directories on some systems, prevent users from removing or
renaming a file in a directory unless they own the file or the
directory; this is called the "restriction deletion flag"
for the directory.
```

Les bits 's' et 't'

```
Lorsque ces bits sont positionnés, ils sont visibles dans "ls -l" par
une modification de la valeur du bit "x" correspondant qui prends
alors les valeurs suivantes :
    's'   If the setuid or setgid bit and the corresponding executable
          bit are both set.

    'S'   If the setuid or setgid bit is set but the corresponding
          executable bit is not set.

    't'   If the sticky bit and the other-executable bit are both set.

    'T'   If the sticky bit is set but the other-executable bit is not
          set.

    'x'   If the executable bit is set and none of the above apply.

    '-'   Otherwise.
```

Le groupe d'un fichier

Le groupe donné au fichier à la création est le groupe effectif du process créateur, SAUF SI le répertoire d'accueil possède le bit Set-GID positionné ; alors, le fichier reçoit le même groupe que celui du répertoire (par exemple public_html).

```
Changer le groupe d'un fichier
------------------------------
Un utilisateur peut appartenir à plusieurs groupes; la liste en
est donnée par la commande:   > groups
Un fichier appartient à UN groupe.
La cmde > chgrp nouveau-grpe fichier  change le groupe de fichier

Elle doit être faite par le owner qui doit être membre du nouveau
groupe qu'il donne au fichier.
Ex: > groups ; ls -lg file1 ; chgrp meca file1 ; ls -lg file1
    staf meca gestion   ! output cmde groups
    -rw-r----- 1 dupond gestion  55 Feb 10 10:30 file1  ! ls -lg
    -rw-r----- 1 dupond meca     55 Feb 10 10:30 file1  ! ls -lg
```

Commandes et appels systèmes relatifs aux protections

```
Commandes et appels systèmes relatifs aux protections

                commande        appel système
        chmod   man chmod       man 2 chmod
        chown   man chown       man 2 chown
        chgrp   man chgrp       man 2 chown <--*** pas une erreur
        umask   man umask       man 2 umask

        access  man access      man 2 access

        touch   man touch       -
        stat    man stat        man 2 stat      man 2 utime
        readlink  man readlink  man 2 readlink  lire lien symbolique
```

3.6 Primitives d'E/S (API) : flux de données et descripteurs de fichiers

---

flux de données et descripteurs de fichiers

```
2 ensembles de primitives:
        - interface des flux (fopen,fread,....)
          fonctions de bibliothèque,
          manipulent des objets de type "FILE"
```

```
        - primitives du noyau (open, read,....)
          appels système, travaillent sur des descripteurs

flux = stream

flux de données et descripteurs de fichiers sont des
notions complémentaires mais distinctes

descripteur = val de type int
        c'est un index dans une table du process, table que seul
        le noyau peut modifier (à la demande du process par
        l'intermédiaire d'appels systèmes tels que open, close,
        dup, ...
        le noyau donne le descripteur au process après avoir associé
        ce descripteur à un objet tel que fichier régulier, répertoire,
        pipe, socket, fichier de périphérique (/dev/xxx), ...

flux de données : objet de type opaque représenté par une structure
        les champs de cette structure n'étant pas accessibles au process
        le process manipule des variables de type FILE *

en interne,un flux est associé par la noyau à un descripteur de fichiers

le flux possède en plus du descripteur des champs donnant son état et
une zone de mémoire temporaire
```

---

flux de données

```
les flux définissent une interface au-dessus de l'interface directe
du noyau :

        programme applicatif
                |
                V
        interface des flux : fopen, fclose, fread, fwrite
        (implémentée dans la glibc)
                |
                V
        interface du noyau : open, close, read, write, fcntl
```

ceci explique que la libc du C Ansi ne connaisse pas les descripteurs qui sont spécifiques du noyau unix

pour écrire un programme portable il est donc préférable d'utiliser l'interface des flux

les flux proposent des fonctions pratiques absentes de l'interface de base du noyau, par exemple fgets() ou les très utiles fflush() et fsync().

MAIS : certaines fonctions spéciales de fcntl() ne sont pas accessibles à travers les flux (par ex. lecture non bloquante, verrous)

Il est possible d'obtenir le descripteur associé à un flux avec fileno() et d'utiliser directement sur ce descripteur des opérations de l'interface du noyau.

Il est aussi possible de demander la construction d'un flux "autour" d'un descripteur obtenu précédemment avec fdopen().

3.7 L'interface des flux

L'interface des flux

```
Ouverture normale d'un flux :
       #include <stdio.h>
       FILE *fopen(const char *path, const char *mode);
       FILE *fdopen(int fildes, const char *mode);
man fopen pour les différents modes

** attention, il n'existe pas de fonction système permettant de
retrouver le nom (pathname) d'un objet FILE ou d'un descripteur.

Fermeture d'un flux
       #include <stdio.h>
       int fclose(FILE *stream);

Fermeture de tous les flux
       fcloseall()
à utiliser par exemple dans un handler de signal avant abort() ou bien
dans un handler atexit().

Exemple : fopwrcl.c  fopwrcl.in  fopwrcl.out
```

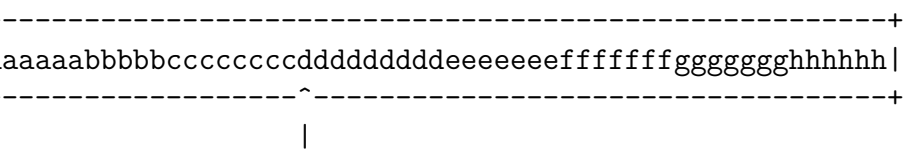
L'interface des flux

```
Fonctions de lecture et écriture dans un flux

lecture et écriture formattées
        fprintf(), vfprintf(), fscanf(), vfscanf()
lecture et écriture de caractères
        fputc(), fputs(), fgetc(), fgets(), fungetc()
lecture et écriture de données binaires
        fwrite(), fread()
```

Positionnement dans un flux

Positionnement dans un flux

Le noyau gère, pour chaque descripteur de fichier ouvert, un index indiquant ou se fera dans le fichier la prochaine opération d'E/S. Cette position est modifiée après chaque lecture ou écriture.



Accès ou modification de la position :

```
       #include <stdio.h>

       int fseek(FILE *stream, long offset, int whence);
       long ftell(FILE *stream);
       void rewind(FILE *stream);
ou
       int fgetpos(FILE *stream, fpos_t *pos);
       int fsetpos(FILE *stream, fpos_t *pos);
ou
       #include <stdio.h>
       int fseeko(FILE *stream, off_t offset, int whence);
       off_t ftello(FILE *stream);
```

fseeko permet (avec #define FILE_OFFSET_BITS 64) de manipuler des fichiers dont la taille dépasse 2 Go, ce qui n'est pas possible avec fseek() puisqu'elle stocke la longueur sur un long de 32 bits.

Ces fonctions ne fonctionnent pas sur les flux liés à des sources de données intrinsèquement séquentielles : pipes, socket et fichiers spéciaux d'accès aux périphériques (ex : /dev/ttys0).

Test des cas d'erreur sur les flux

Test des cas d'erreur sur les flux

```
       #include <stdio.h>

       void clearerr(FILE *stream);
       int feof(FILE *stream);
       int ferror(FILE *stream);
       int fileno(FILE *stream);

       The function ferror tests  the  error  indicator  for  the
```

```
       stream  pointed  to by stream, returning non-zero if it is
       set.
```

Buffers et mises à jour

Buffers et mises à jour

De même qu'avec les flux il y a deux niveaux d'interface, il y a deux niveaux de buffers : un au niveau du flux associé au descripteur et un au niveau de l'interface du noyau pour limiter les accès disque.

Le premier niveau (flux) se vide par fflush(). Cet appel garanti la cohérence des données écrites sur le flux si plusieurs programmes le partagent. Nota : fflush() sur un flux ouvert en lecture seule n'a pas de signification et doit être évité.

Le deuxième niveau (cache disque) est géré par le noyau. Pour assurer que les données sont écrites sur disque, le noyau possède l'appel système sync().

Toutefois, certains contrôleurs de disque possèdent des mémoires caches (pouvant contenir des dizaines voire des centaines de mégaoctets) et ne garantissent pas nécessairement une écriture immédiate. En cas d'arrêt d'urgence (coupure d'alimentation) des données peuvent être perdues. Ce point doir être vérifié en consultant la documentation du matériel dans le cas ou on implante un système critique.

```
L'appel sync() concerne tout le système. On peut vouloir assurer de
l'écriture sur disque des données d'UN flux de deux façons :
        write() après un fcntl(fd,O_SYNC)
        ffsync(fd) (déclaré dans <unistd.h>)
```

3.8 Les appels systèmes de base

Les appels systèmes de base : Descripteurs de fichiers

```
Un descripteur est un entier compris entre 0 et OPEN_MAX.

993 sr01 sunserv:~> grep OPEN_MAX /usr/include/limits.h
#define OPEN_MAX     64    /* max # of files a process can have open */

Descripteurs réservés :
        0       stdin
        1       stdout
        2       stderr
        Il existe des constantes symboliques :
995 sr01 sunserv:~> grep _FILENO /usr/include/*.h
/usr/include/unistd.h:#define   STDIN_FILENO    0
/usr/include/unistd.h:#define   STDOUT_FILENO   1
/usr/include/unistd.h:#define   STDERR_FILENO   2
```

Le modèle de base du système d'E/S est : une séquence d'octets accédés soit aléatoirement soit séquentiellement. Il n'y a pas de méthode d'accès ou de blocs de contrôle. Dans la plupart des cas un canal d'E/S est vu seulement comme un "courant d'octets" (stream of bytes).

Un process u*x utilise un DESCRIPTEUR pour référencer un courant d'octets. Un descripteur est un petit entier non signé. Un "read" ou un "write" peuvent être faits sur un "descripteur" pour transférer des données. Un "close" est utilisé pour désalouer un descripteur.

```
Un descripteur peut représenter trois types d'objets:
     1- un fichier ("file"), tableau linéaire d'octets
        peut être un fichier ordinaire, un répertoire ou un
        périphérique
        .    un fichier existe tant que tous les noms qui le
             désignent n'ont pas été détruits (rm) et qu'un
             process possède un descripteur vers lui.
        .    les périphériques d'E/S sont accédés comme des
             fichiers.
        .    on obtient un descripteur sur un fichier par "open"
     2- un tube ("pipe"), utilisé seulement comme un I/O stream
        et est unidirectionel.
        .    il n'a pas de nom, donc on ne pas faire de "open"
        .    il est créé par l'appel système "pipe" qui retourne
             DEUX descripteurs: l'un accepte des octets en entrée
             et les envoie sur l'autre de façon fiable, dans l'ordre
             et sans duplication.
        .    seuls des process ayant un ancêtre commun peuvent
             communiquer par un pipe créé par cet ancêtre.
     3- un "socket" est un objet utilisé dans la communication
        inter-process
        .    il est créé par l'appel "socket" qui renvoie un
             descripteur
        .    il existe tant qu'un process détient un descripteur
             qui le référence
        .    il y a plusieurs types de sockets qui supportent des
             sémantiques différentes (fiabilité, préserver ordre,
             préserver frontières de messages)
```

Avant BSD 4.2 les pipes étaient implémentés par le file system (un pipe = un fichier), à partir de BSD 4.2 les pipes sont implémentés par des sockets.

Le noyau maintient une table des descripteurs pour chaque process. Le "descripteur" (fd) fournit en réponse à un "fd = open" est un index dans cette table.

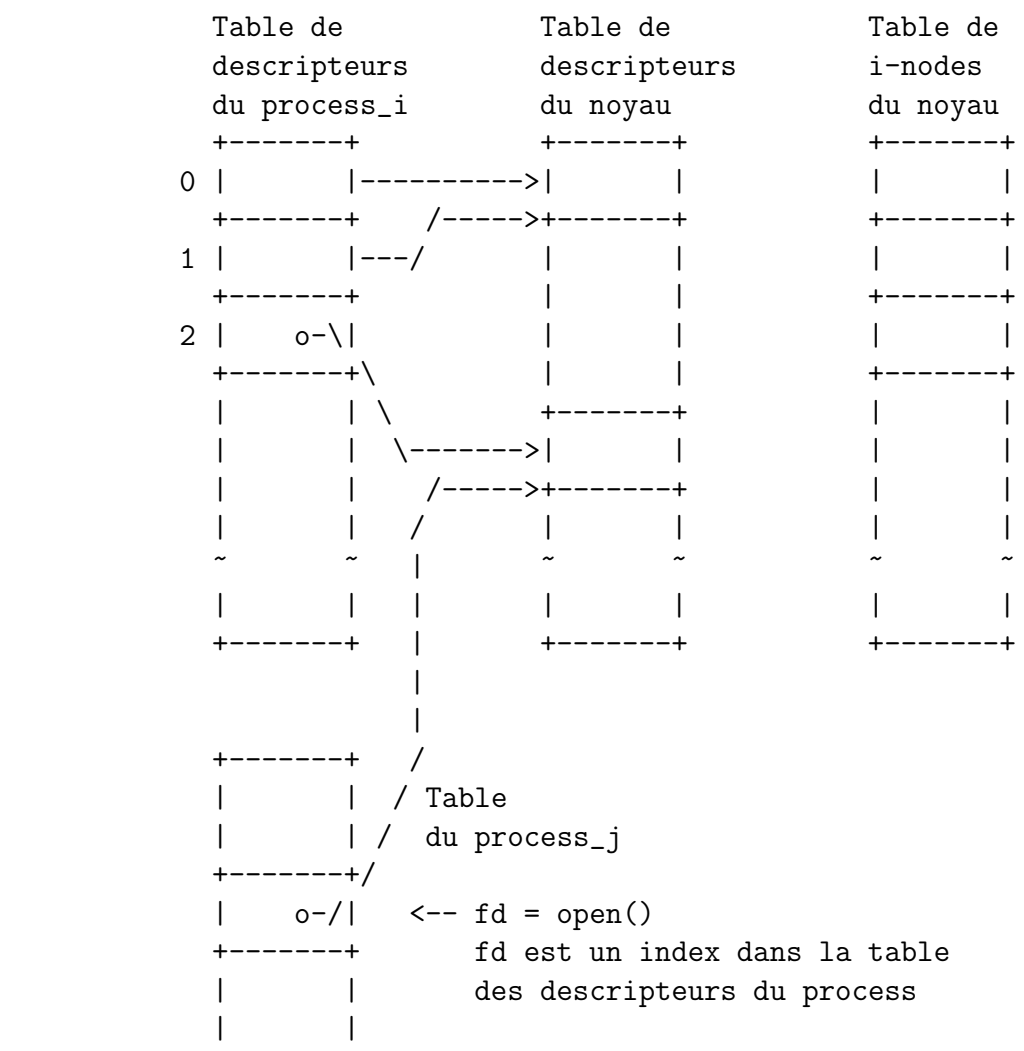
Cette table est héritée par le fils lors d'un "fork", ainsi le parent et le fils partagent les accès aux mêmes fichiers. Un descripteur peut être passé à un autre process à travers un socket, dans un message (mais n'a de signification que sur la même machine).

Chaque entrée dans la table des descripteurs pointe sur une structure qui contient un décalage (byte offset) en octets depuis le début du fichier. Les opérations de lecture / écriture sont faites à partir de ce décalage. Pour les objest qui permettent l'accès aléatoire

(les fichiers), le décalage peut être modifié par "lseek".

Quand un process termine, le système récupère tous les descripteurs. Si un descripteur était la dernière référence, le gestionnaire de l'objet est notifié afin de faire le ménage (détruire un fichier ou désallouer un socket).

```
          +-----+
        0 |     | stdin         Les 3 premiers descripteurs sont
        1 |     | stdout        préassignés par défaut au tty et
        2 |     | stderr        hérités à travers fork et exec
          |     |
```




```
Fonctions disponibles sur les descripteurs :
        open(), close()         man open
        read(), write()         man 2 read
        readv(), writev()       man readv  scatter/gather
        lseek()                 man lseek
        dup(), dup2()           man dup
        fcntl()                 man fcntl
        truncate()              man truncate
        mknod , mknod()         man mknod , man 2 mknod
```

```
****    Never use gets()
       Never use gets().  Because it is impossible to tell  with-
       out knowing the data in advance how many characters gets()
       will read, and because gets() will continue to store char-
       acters past the end of the buffer, it is extremely danger-
       ous to use.  It has been used to break computer  security.
       Use fgets() instead.

NAME
       open, creat - open and possibly create a file or device

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

       int open(const char *pathname, int flags);
       int open(const char *pathname, int flags, mode_t mode);
       int creat(const char *pathname, mode_t mode);

open(), pipe(), socket() créent un nouveau descripteur qui est tjrs
le premier numéro non utiilisé de la table des descripteurs.
```

Redirection

Redirection : close (stdx) puis open(file)

Redirection sur pipe : créer pipe, close (std), dup

Attention ! dup() prends le premier libre : si on veut stdout et que stdin était aussi fermé c'est lui qui sera assigné ; pour éviter cela, on peut utiliser dup2() qui désigne explicitement le descripteur visé : s'il n'est pas libre, dup2() en fait d'abord un close avant de le réassigner.

Périphériques (devices)

Les périph. matériels ont des noms de fichiers (dans /dev) et peuvent être accédés par les mêmes appels systèmes que ceux des fichiers. Ces fichiers sont des "fichiers spéciaux" (special file) distingués comme tels par le noyau.

Ils sont créés par l'appel "mknod()". L'appel "ioctl()" permet de manipuler les paramètres spécifiques d'un périphérique.

Sockets

Un socket est un point d'arrivée d'un canal de communication, référencé par un descripteur (comme un fichier ou un pipe). Deux process peuvent chacun créer un socket, et ensuite connecter les deux points d'arrivée pour produire un canal "flot d'octets" fiable.

Une fois connectés les descripteurs d'un socket peuvent être lus ou écrits avec les appels standards read write.

Cette transparence permet de faire une redirection sur un process situé sur une autre machine.

```
pipe # socket:        pipe besoin d'un parent commun
       socket peut lier deux process quelconques même sur deux
       machines différentes

Pour des connexions de type "flot d'octets": read, write
Pour des messages formattés:  send, sendto, sendmsg
                              recv, recvfrom, recvmsg
sendmsg et recvmsg étant les plus généraux.
```

Translating inode numbers to filenames

```
Function: errcode_t ext2fs_get_pathname
         (ext2_filsys fs, ext2_ino_t dir, ext2_ino_t ino, char **name)

ret = ext2fs_open(argv[1], 0, 0, 0, unix_io_manager, &fs);
ext2fs_get_pathname(fs, dir, 0, &path);

The EXT2FS Library   Version 1.27  March 2002
The EXT2FS library is designed to allow user-level programs to
manipulate an ext2 filesystem.

--> getname.c
```

```
/* getname.c
 *
 * compile: gcc -o getname getname.c -lext2fs
 */

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <ext2fs/ext2fs.h>

int main(int argc, char **argv)
{
    errcode_t ret;
/*    ext2_ino_t ino, dir; */
    int ino, dir;
```

```
ext2_filsys fs;
    char *path;


    if(argc != 4) {
        printf(
"Usage: %s <fs e.g. /home > <inode dir> <inode file>\n",
        argv[0]);
        exit(0);
    }
    dir = atoi(argv[2]);
    ino = atoi(argv[3]);

    ret = ext2fs_open(argv[1], 0, 0, 0, unix_io_manager, &fs);
    if (ret) {
        com_err(argv[1], ret, "while opening filesystem");
        exit(1);
    }
    ret = ext2fs_get_pathname(fs, dir, ino, &path);
    if (ret) {
        com_err(argv[1], ret, "while get pathname");
        exit(1);
    }
    printf("path=%s\n",path);
    free(path);
    return 0;
}
```

Exploration des répertoires

Exploration des répertoires

--> lisrep.c lisrep.txt

```
/* lisrep.c
 * gcc -o lisrep lisrep.c
 */
/* UTC - UV SR01 - (c) Michel.Vayssade@utc.fr oct 2003 */

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

main() {
    DIR *ladir;
    struct dirent *unfic; /* une entrée de la DIR */
```

```
    ladir = opendir("."); /* ouvrir current dir */
    if (ladir==NULL) { perror("opendir . failed"); exit(1);}

    unfic = readdir(ladir); /* initialiser la lecture de ladir */
    while (unfic != NULL) { /* pour chaque entrée de ladir */
     printf("%s \t ino=%d\n", unfic->d_name, unfic->d_ino);
unfic = readdir(ladir); /* passer à l'entrée suivante */
    }
    closedir (ladir);
    exit(0);
}
```

```
1060 sr01 sunserv:~/mv> gcc -o lisrep lisrep.c
1061 sr01 sunserv:~/mv> ./lisrep
.        ino=271460
..       ino=217911
inx.c    ino=271461
ptr.c    ino=271462
inx.o    ino=271463
ptr      ino=271464
mystat.c         ino=271479
mystat   ino=271480
lisrep.c         ino=271481
lisrep   ino=271478
1062 sr01 sunserv:~/mv>
1062 sr01 sunserv:~/mv>
```

Scatter/Gather I/O : Entrées/Sorties dispersées/rassemblées

À partir de 4.2 BSD. les appels "readv" et "writev" permettent de faire en une seule opérations d'E/S des lectures ou écritures dans plusieurs buffers dispersés.

Ceci évite la copie des buffers dans un buffer unique contigu.

```
  readv()
     The readv() function is equivalent to read(), but places the
     input  data into the iovcnt buffers specified by the members
     of the iov array: iov0, iov1, ..., iov[iovcnt-1]. The iovcnt
     argument  is  valid if greater than 0 and less than or equal
     to IOV_MAX.

     The iovec structure contains the following members:
        caddr_t   iov_base;
        int       iov_len;

     Each iovec entry specifies the base address and length of an
     area  in  memory  where  data should be placed.  The readv()
```

```
function always fills an area completely  before  proceeding
to the next.
```

3.9 Structure interne du système de fichiers dans le noyau

Chaque disque contient une ou plusieurs partitions. Une partition contient un et un seul système de fichiers (file system).

Sur les unix récents, on peut avoir des partitions logiques s'étendant sur plusieurs partitions physiques appartenant éventuellement à des disques différents .

Un système de fichiers contient :

- des fichiers de données,
- des répertoires (directories) qui contiennent des pointeurs vers les fichiers de données et/ou d'autres répertoires.

L'ensemble des répertoires forme un arbre.

– Les répertoires –

Les répertoires sont alloués en unités appelées "chunk" (morceau) dont la taille est choisie pour pouvoir être écrits en une seule opération sur le disque, pour garantir l'atomicité des modifications dans les répertoires. Les morceaux sont découpés en "entrées" de longueur variable afin de permettre des noms de fichier de longueur arbitraire (maxi 255 car.). Une entrée n'est jamais à cheval sur deux morceaux.

```
Une entrée de dir. contient les champs suivants:
               |       |
      +--<---  |-- n   | taille de cette entrée
      |        |  lng  | longueur du nom du fichier dans cette entree
      |        | i-pt  | pointeur sur le i-node du fichier
      |        |abcdefg| nom du fichier, terminé par un zéro,
      V        |hi\o...| remplit jusqu'à une frontière de 4 octets
      |        |///////|
      |        |///////| espace inutilisé dans cette entrée car le nom
      +-----> |       | actuel est petit que le précédent
```

Quand une entrée est libérée l'espace qu'elle occupait est fusionné à celui de l'entrée qui la précède dans le même morceau. Si la première entrée d'un morceau est libérée (i.e. on fait rm du fichier référencé dans cette entrée) le pter sur le i-node est mis à zéro pour indiquer que l'entrée est libre.

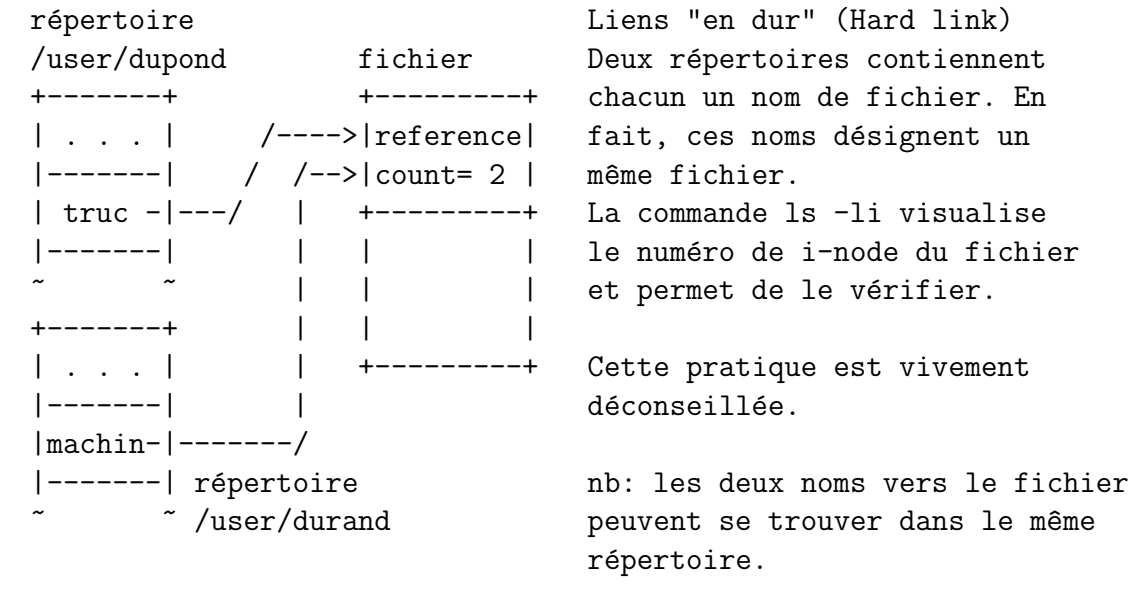
BSD 4.2 a introduit une interface d'accès aux fichiers répertoires : opendir() readdir() rewindir() closedir() seekdir() telldir()

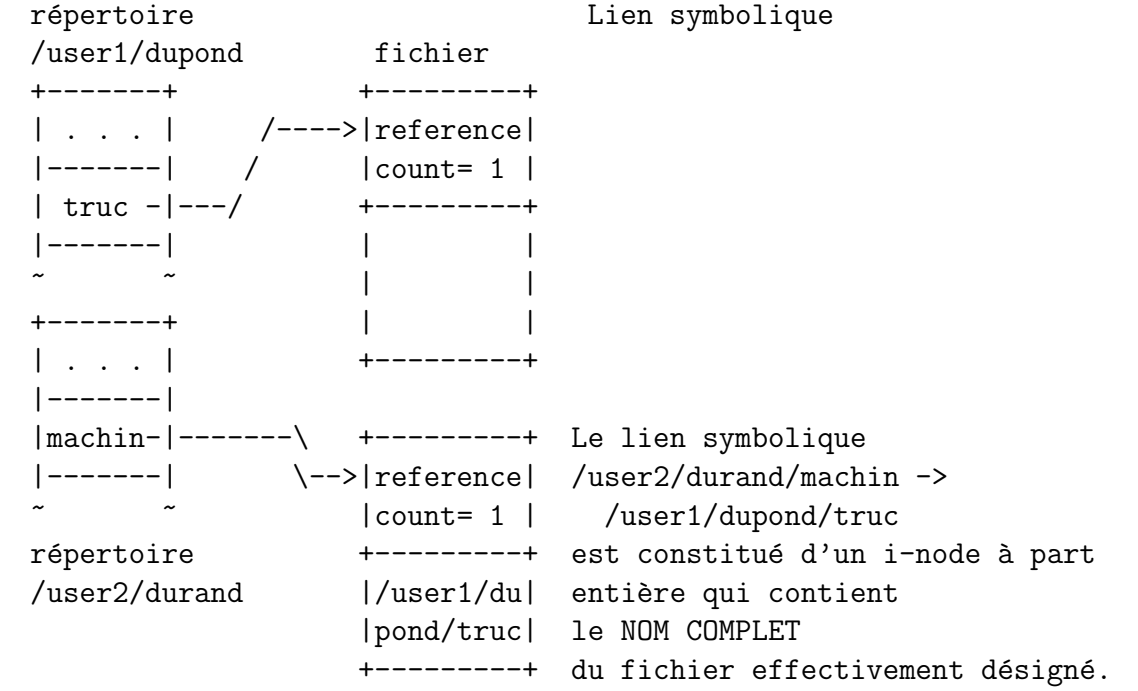
– Liens –

Chaque fichier possède un unique i-node qui contient les informations permettant d'y accéder. Mais un fichier peut avoir plusieurs entrées dans un ou plusieurs répertoires.

Les i-nodes ne résident pas dans les répertoires mais existent indépendamment des répertoires et sont référençés par des LIENS. Un i-node contient un compteur de référence des liens qui "pointent" sur lui. Quand tous les liens vers un i-node sont enlevés, le i-node est détruit.

```
hard link:    2 entrées de répertoire ont un pointeur vers le même
              i-node, le ref. count de cet i-node est 2
soft link:    un lien symbolique est implanté comme un fichier qui
              contient un chemin (path name)
```





Le lien symbolique peut ainsi désigner un fichier situé dans un autre système de fichiers ; mais le fichier "pointé" (ici dupond/truc) "ne sait pas" qu'un lien pointe sur lui. Si le fichier "truc" est détruit par dupond, le lien pointera sur du "vide". Si on essaie d'y accéder, on a un message "no such file".

Quand open() est appelée sur un lien symbolique il renvoie un descripteur sur le fichier pointé par le lien et non pas sur le lien lui-même. Il en est de même de la plupart des appels qui prennent un nom de fichier en argument : ils "suivent" le lien.

Une variante de stat() appelée lstat() permet d'obtenir le statut du lien lui-même. Un lien symbolique peut se référer à un répertoire ou un fichier résidant dans un autre "file system" alors qu'un lien en dur doit rester dans le même système de fichiers.

Comme une série de liens peut provoquer des boucles, le noyau vérifie qu'un nombre maximum (8) de liens ne soit par dépassé lors de la traduction d'un nom complet (path-name). Si oui, il renvoie l'erreur EMLINK.

```
Rem:  pwd                          pwd
      /usr/truc                    /usr/truc
      ls -l src                    ls -l src
      /usr/truc/src --> /usr/src   /usr/truc/src
      cd src                       cd src
      pwd                          pwd
      /usr/src                     /usr/truc/src
      cd ..                        cd ..
      pwd                          pwd
      /usr                         /usr/truc
```

Le noyau ne garde pas trace de la façon dont est parvenu à /usr/src

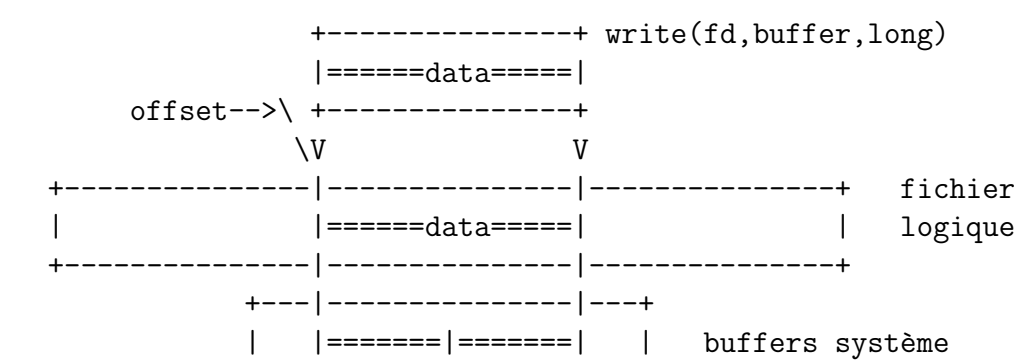
---

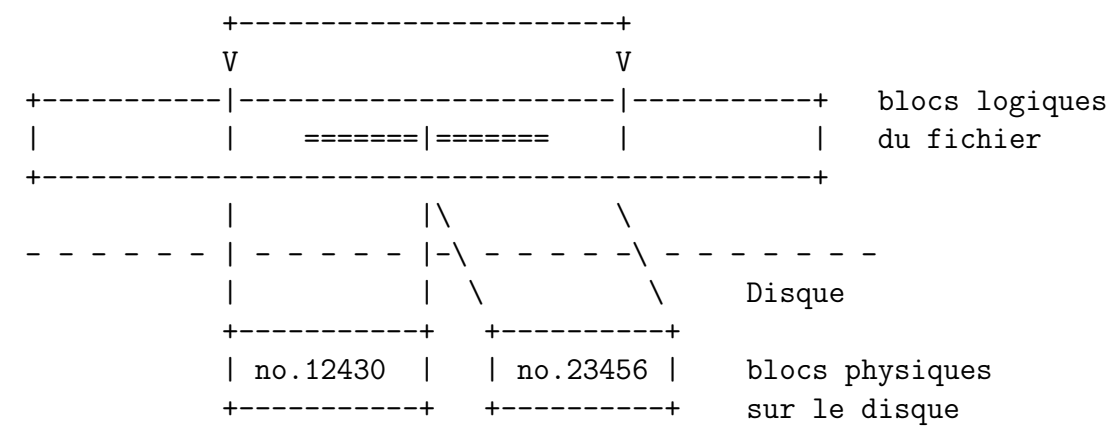
– E/S par blocs sur les disques –

L'utilisateur voit une suite d'octets, le disque lit et écrit seulement des blocs.
Pour modifier UN octet sur disque, il faut :

1. lire le bloc disque dans un buffer du cache,
2. copier l'octet modifié dans le cache,
3. écrire le bloc sur disque.

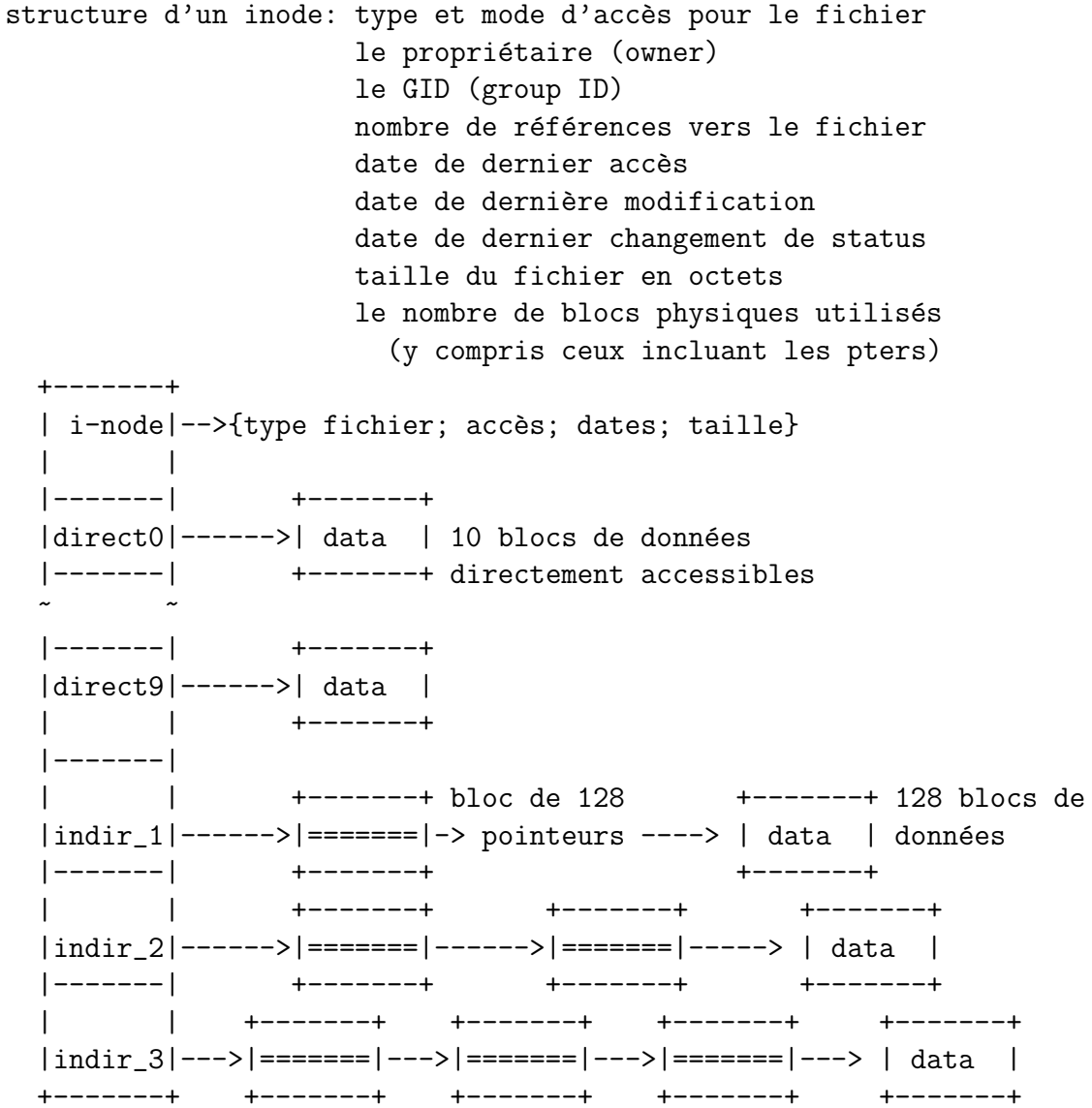
```
Relation offset -> bloc logique -> bloc virtuel
```





– recherche et allocation des blocs sur disque –

Structure d'un inode : blocs directs et indirects



– les buffers système : le cache

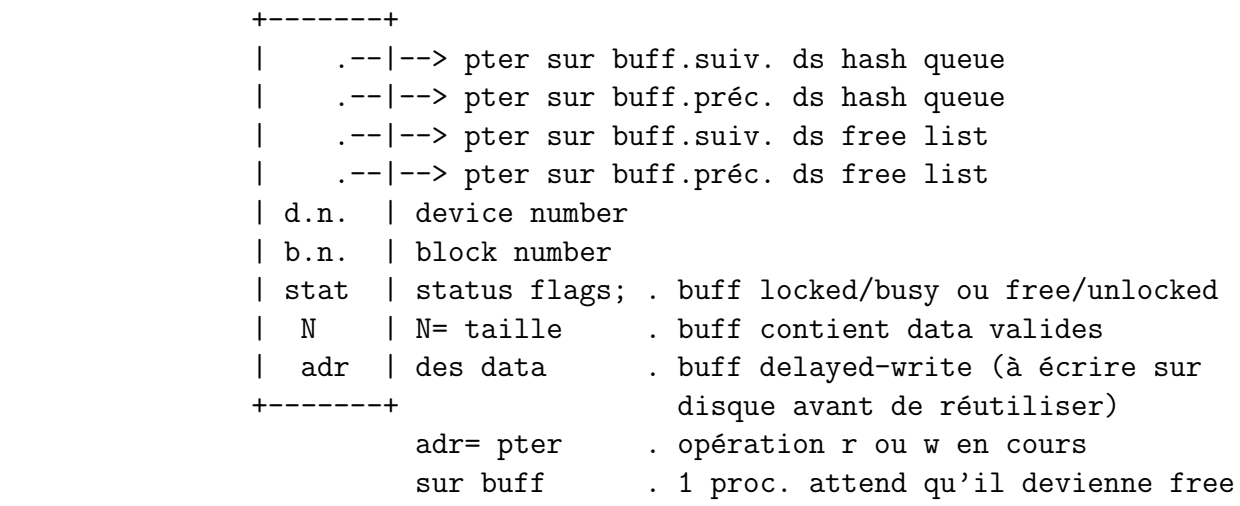
La sémantique du système de fichiers d'u*x implique l'exécution de beaucoup d'E/S disque. Si toutes devaient réellement se faire, l'UC passerait beaucoup de temps à en attendre la fin.

Le pool de buffers a deux rôles :
– gérer la mémoire qui sert d'intermédiaire entre le disque et les process,
– agir comme un cache pour les opérations d'E/S sur disque.

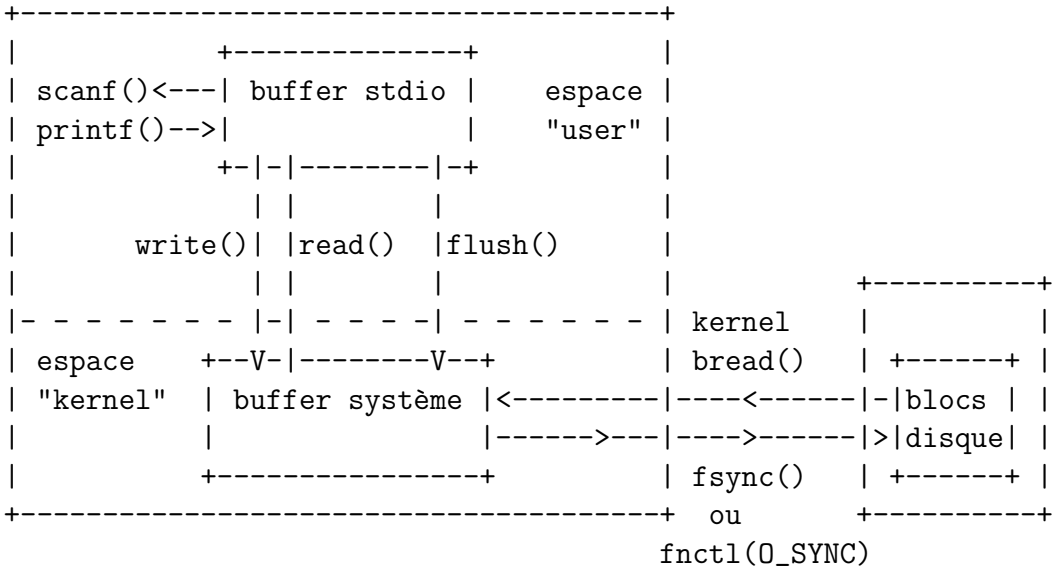
Sur un syst, u*x typique, environ 85% des transferts effectifs entre disque et mémoire entrainés par des requêtes d'E/S sont évitées par suite de la présence des blocs demandés dans le cache.

Chaque buffer = header+contenu

le header contient des informations de gestion (queues chainées) et une description du contenu : device. numéro de bloc, taille, status (modifié ou non, utilisé ou libre).



Un appel système vers un fichier: passage par le cache



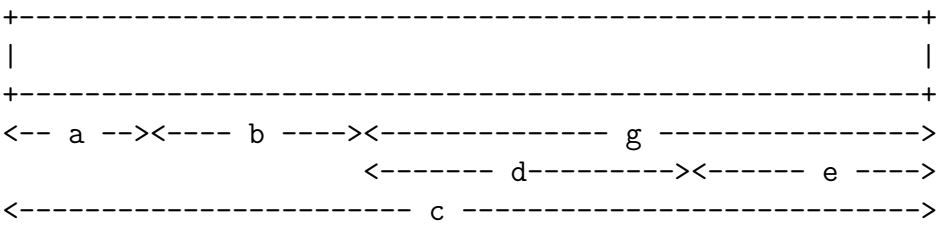
Header et data buff dans zones séparées, car buffer = taille multiple d'une page et utilisation des capacités du matériel pour gérer les pages de mémoire virtuelle (protections, écritures).

Un buffer est tjrs au moins aussi long que les data qu'il contient (ie pas de fragments). Les requêtes du file system varient de 512 octets à 8192. À chaque buffer est alloué 8192 octets de mémoire virtuelle contigue, mais n'est initialisé qu'avec 2048 octets de mémoire physique. La mémoire physique est ainsi utilisée au mieux tout en conservant une "vue" contigue de chaque buffer, bien que physiquement, celui-ci puisse être éclaté en plusieurs morceaux, ceci en profitant des capacités de mapping du matériel.

La présence du cache (qui peut utiliser de 10 à 20% de la mémoire totale de la machine selon le réglage fait par l'administrateur du système), impose l'usage des directives sync(), fsync(), ... quand on veut s'assurer de l'écriture effective de données sur le disque.

3.10 Structure interne du système de fichiers "ext2" sur les disques

Les disques : partitions et montage



Un disque physique est découpé en partitions. Une partition est un disque logique qui va contenir un et un seul système de fichiers.

Notion de formattage

```
/dev/hda -> {hda1 (hda2) hda3 hda4}
résultat d'une opération de formatage

FDISK(8)            Linux Programmer's Manual            FDISK(8)
NAME    fdisk - Partition table manipulator for Linux

[root@virgo /root]# fdisk /dev/hda

The number of cylinders for this disk is set to 1662.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
```

```
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help):

Command (m for help): m
Command action
   a   toggle a bootable flag
   b   edit bsd disklabel
   c   toggle the dos compatibility flag
   d   delete a partition
   l   list known partition types
   m   print this menu
   n   add a new partition
   o   create a new empty DOS partition table
   p   print the partition table
   q   quit without saving changes
   s   create a new empty Sun disklabel
   t   change a partition's system id
   u   change display/entry units
   v   verify the partition table
   w   write table to disk and exit
   x   extra functionality (experts only)

Command (m for help): p

Disk /dev/hda: 255 heads, 63 sectors, 1662 cylinders
Units = cylinders of 16065 * 512 bytes

   Device Boot    Start       End    Blocks   Id  System
/dev/hda1             1        64    514048+  83  Linux
/dev/hda2            65       162    787185   82  Linux swap
/dev/hda3           163       927   6144862+  83  Linux
/dev/hda4           928      1662   5903887+  83  Linux
```

Notion de "montage" de système de fichiers

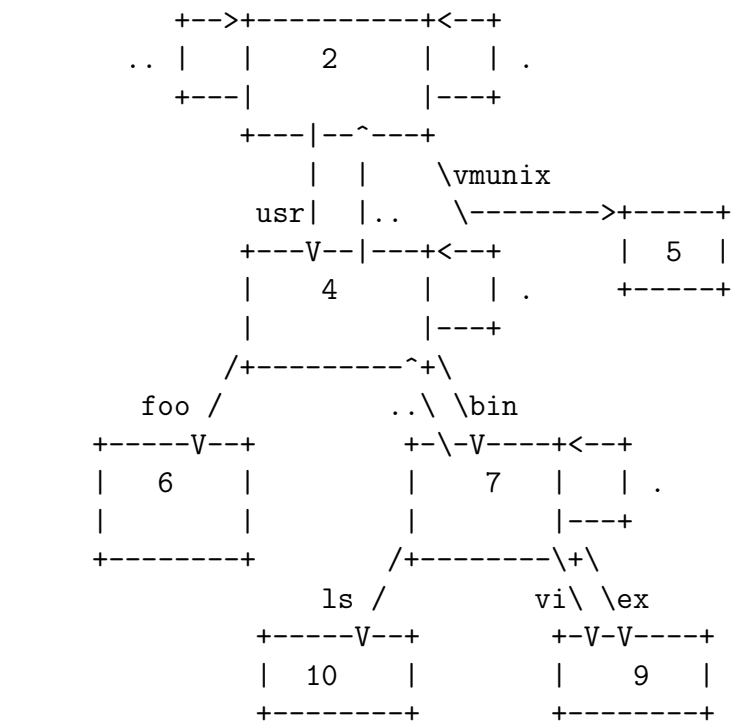
```
Liste des partitions "montées" :

$ df
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/hda1               497829     78259    393868  17% /
/dev/hda4              5811084   4338504   1177388  79% /home
/dev/sda1              4182468   1380412   2589600  35% /u1
```

```
/dev/hda3              6048352   1792036   3949076  32% /usr
kappa:/user7/users/vayssade
                      35304336  16826253   3776856  82% /kappa
simnav4.utc:/home/vayssade
                       5036316   4122336    658148  87% /simnav4

$ df -m
Filesystem           1M-blocks      Used Available Use% Mounted on
/dev/hda1                  486        77       384  17% /
/dev/hda4                 5675      4237      1149  79% /home
/dev/sda1                 4084      1349      2528  35% /u1
/dev/hda3                 5907      1751      3856  32% /usr
kappa:/user7/users/vayssade
                         34477     16433      3688  82% /kappa
simnav4.utc:/home/vayssade
                          4918      4026       642  87% /simnav4

$ ls -F /
bin/   dev/  home/   lib/         misc/  opt/   root/  tmp/  usr/
boot/  etc/  kappa/  lost+found/  mnt/   proc/  sbin/  var/

$ ls -F /usr
bin/   etc/    include/   lib/      lost+found/  sbin/   tmp@
dict/  games/  java/      libexec/  man/         share/  X11R6/
doc/   html/   kerberos/  local/    pvm3/        src/

$ stat /
  File: "/"
  Size: 1024       Blocks: 2          Directory
Access: (0755/drwxr-xr-x)     Uid: (  0/  root)  Gid: (  0/  root)
Device: 301        Inode: 2           Links: 22

$ stat /usr
  File: "/usr"
  Size: 4096       Blocks: 8          Directory
Access: (0755/drwxr-xr-x)     Uid: (  0/  root)  Gid: (  0/  root)
Device: 303        Inode: 2           Links: 21

$ stat /home
  File: "/home"
  Size: 4096       Blocks: 8          Directory
Access: (0755/drwxr-xr-x)     Uid: (  0/  root)  Gid: (  0/  root)
Device: 304        Inode: 2           Links: 13

$ cd /
$ /simnav4/pc/uvs/sr01-2003/poly_c_es/lisrep
.        ino=2
```

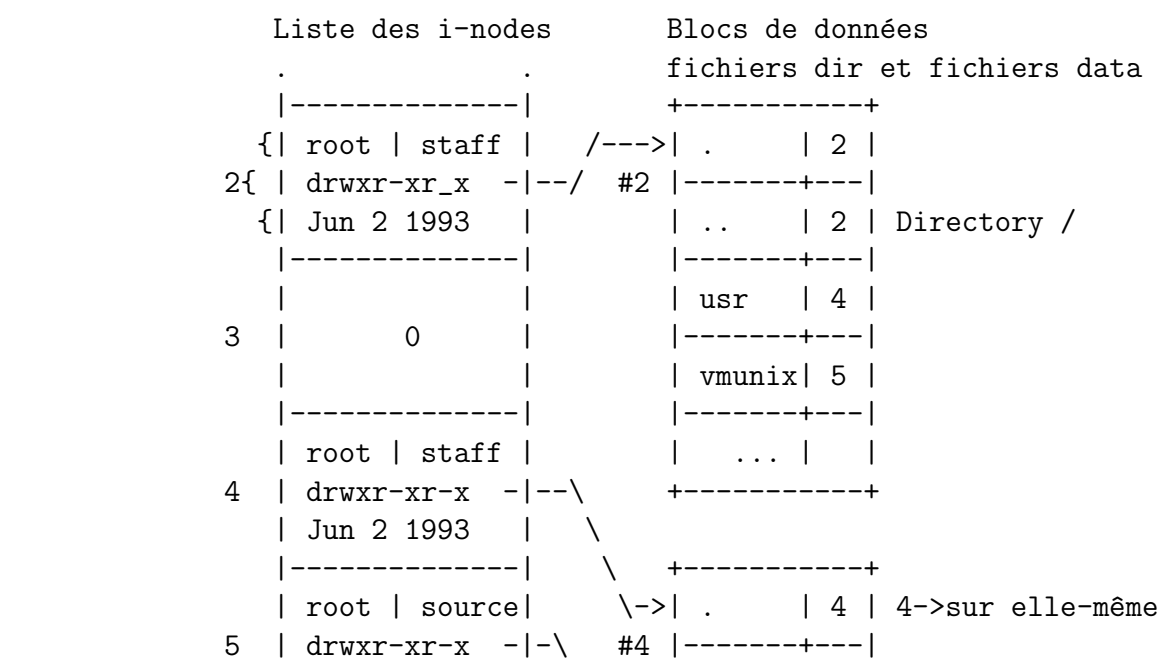
```
..        ino=2
lost+found        ino=11
home      ino=4081
u1        ino=8161
usr       ino=12241
....

(Device: 304  Inode: 2)---> /home (ino=4081  Device: 301)
(Device: 303  Inode: 2)---> /usr  (ino=12241 Device: 301)


Partitions montées au démarrage du système :

$ more /etc/fstab
LABEL=/           /               ext2    defaults        1 1
/dev/hda4         /home           ext2    defaults        1 2
/dev/fd0          /mnt/floppy     auto    noauto,owner    0 0
/dev/sda1         /u1             ext2    defaults        1 2
LABEL=/usr        /usr            ext2    defaults        1 2
none              /proc           proc    defaults        0 0
none              /dev/pts        devpts  gid=5,mode=620  0 0
/dev/hda2         swap            swap    defaults        0 0


D'autres partitions peuvent être montées à tout moment :

Exemple : montage d'un disque USB :
# mkdir /flash
# mount -t vfat /dev/sda1  /flash

Exemple : montage d'une partition NFS :
# mkdir /virgo
# mount -t nfs virgo:/home/vayssade /virgo

et démontées par : umount /nom_du_point_de_montage
```

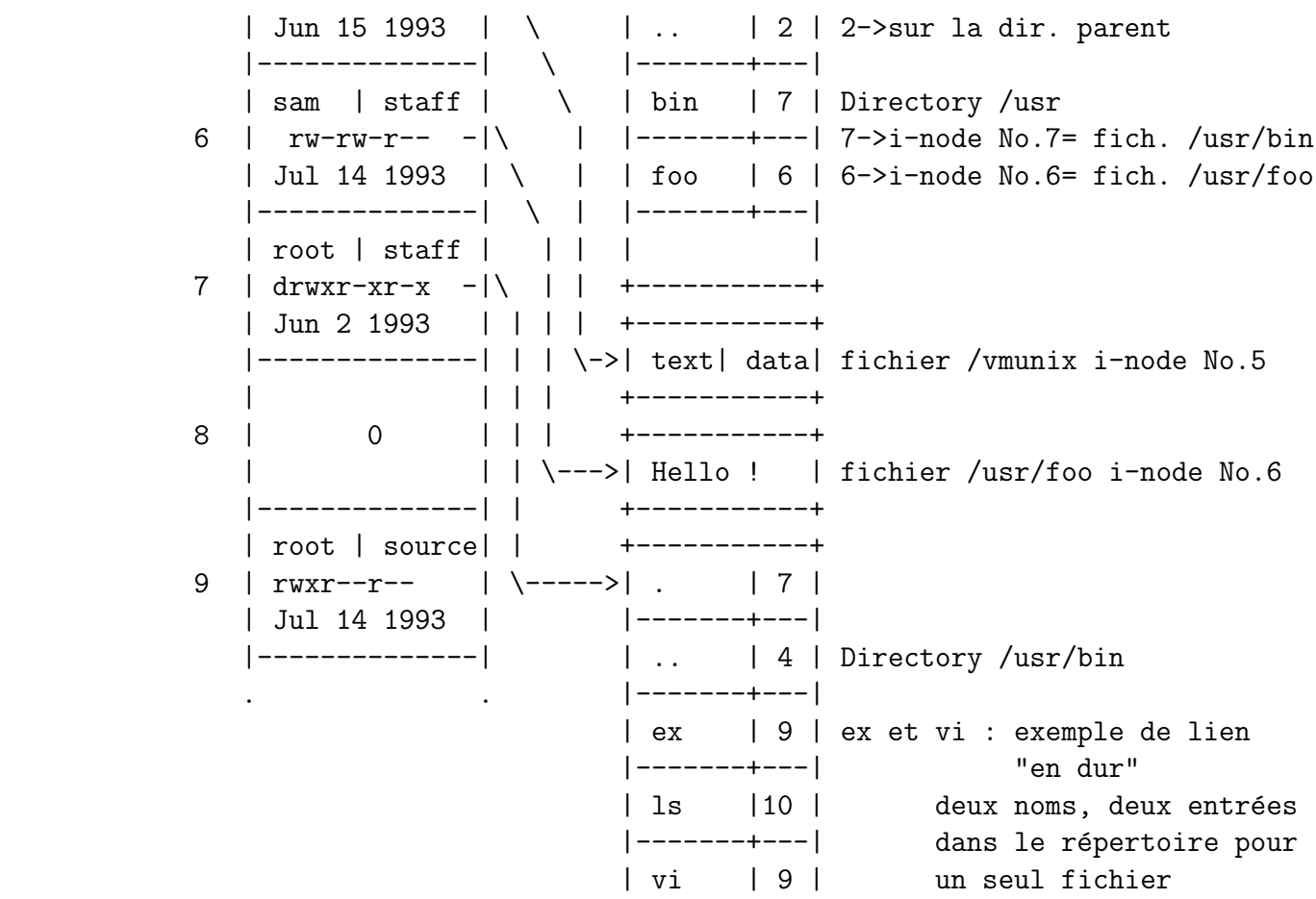
Structure logique : inode, blocs d'un fichier, répertoires

un exemple de petit file system: diagramme des fichiers
figure ci-dessous :



un exemple de petit file system: contenu des dirs et des i-nodes





Dans cette petite arborecence, l'exécution de /usr/bin/vi va se traduire par la série de tâches suivantes:

- trouver le inode de / (numéro 2 par convention)
- lire les blocs de /
- trouver dedans l'entrée usr, l'inode de usr est 4
- lire ce inode
- lire les blocs de usr
- trouver l'entrée bin, son inode est 7
- lire le inode 7
- lire les blocs de bin
- chercher l'entrée vi, son inode est 9
- lire le inode 9
- lire les blocs contenant l'exécutable vi

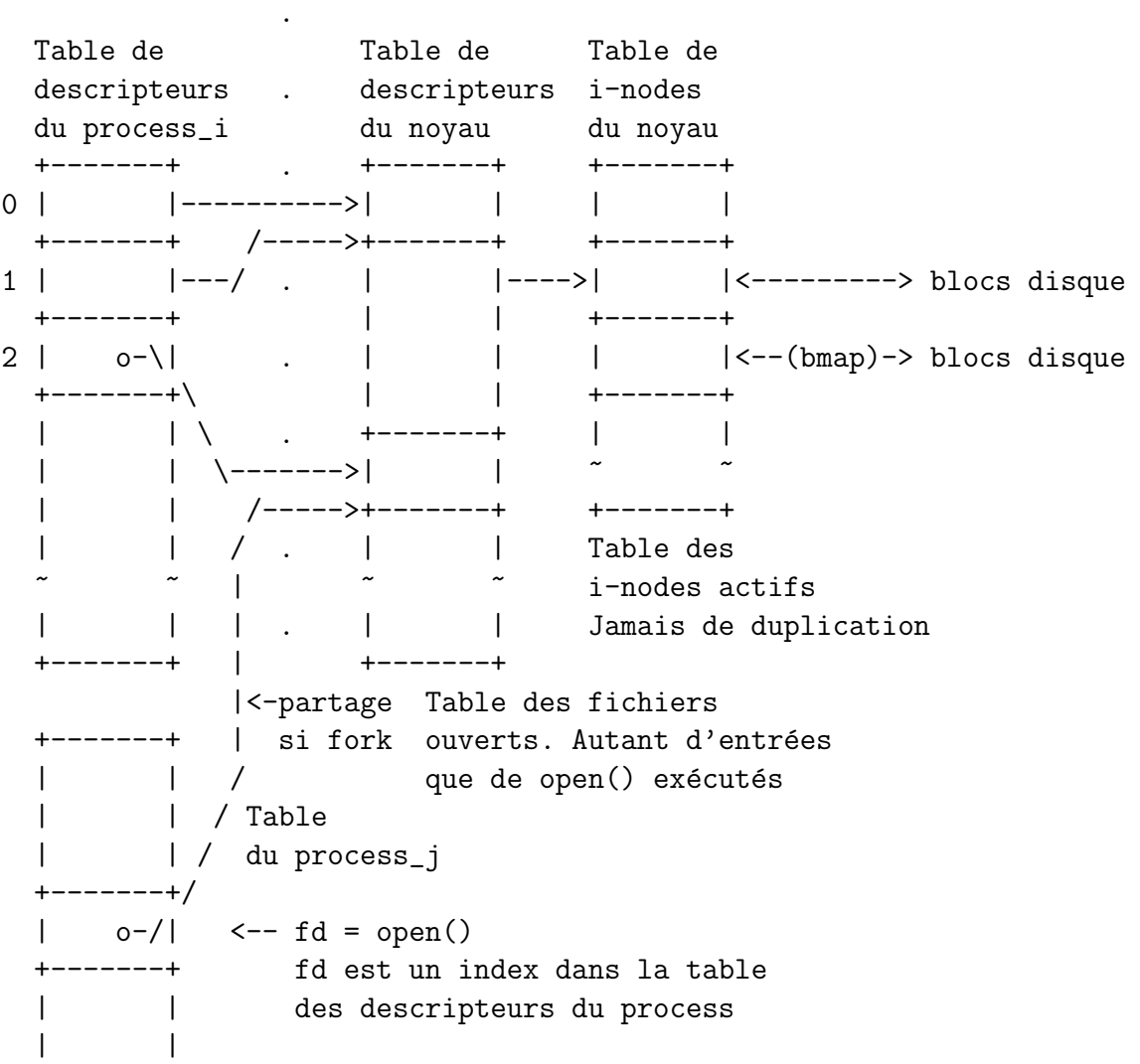
Tout ceci représente une énorme quantité d'opérations. On ne peut parvenir à une performance raisonable que si on réduit par des mécanismes de caches les accès effectifs au disque. Il y a trois structures maintenues par le noyau qui font office de caches pour accélérer les accés sur les disques:

- la table des inodes recemment accédés

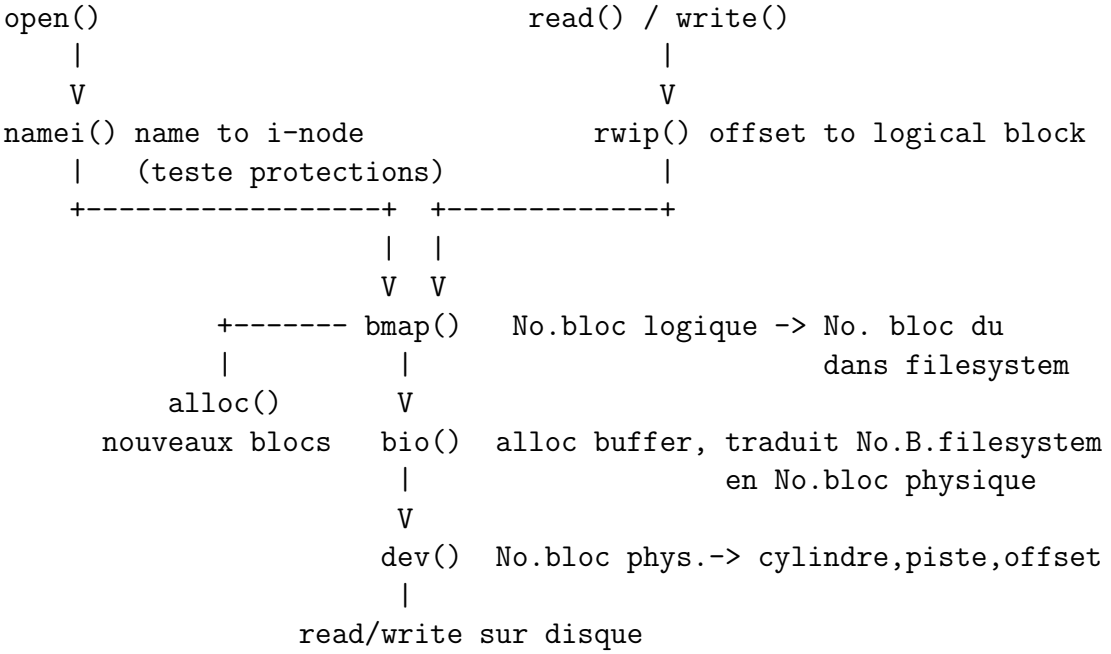
- la table des blocs recemment accédés (le buffer cache dont
  on a déja parlé)
- la table des traduction de noms en inode

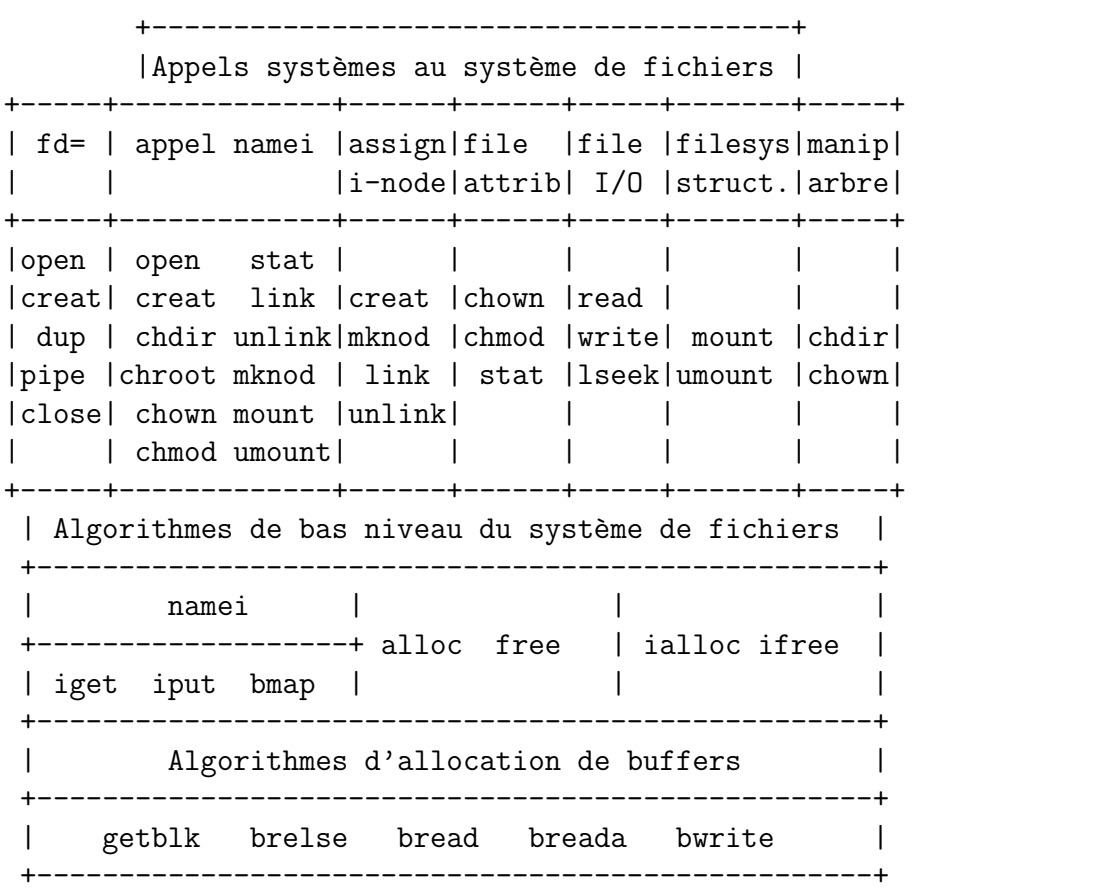
Ces trois caches permettent d'obtenir 85% des informations nécessaires sans avoir besoin de lecture sur disque effective.

Les tables d'accès: descriptor, open file, inode table



Les procédures internes du noyau pour l'accès au système de fichier





Une fois le inode en mémoire on accède aux blocs de données:

open() appelle namei(): convertit path-name en No. de i-node
crèe une entrée dans la file table (kernel)

```
        crèe une entrée dans la descriptor table (process)
read()  } préparent l'e/s entre espace user et kernel
write() } (utilisation u-area), puis appellent rwip()
rwip()  convertit offset ds fichier en No. de bloc logique
bmap()  utilise les pointeurs trouvés dans le inode pour
        convertir les numéros de blocs logiques à l'intérieur
        du fichier en numéros de blocs virtuels sur la
        partition contenant le file system, puis appelle bio()
bio()   convertit bloc virtuel en bloc physique sur disque
        appelle dev(): bloc phys.-> (cylindre, piste, secteur)
```

Structure physique : secteurs, groupes de blocs, super-bloc

```
disque = ensemble de partitions

partitions = suite d'ensembles de blocs

+---------+----------+----------+     ----------+----------+
| secteur | ensemble | ensemble | ...  ensemble | ensemble |
| de boot | de blocs | de blocs | ...  de blocs | de blocs |
+---------+----------+----------+     ----------+----------+

ensemble de blocs =

+----------+--------------+--------+----------+-----------+----------+
| copie du | descripteurs | bitmap | bitmap   | table des | blocs de |
| superbloc| du file syst | blocs  | i-noeuds | i-noeuds  | données  |
+----------+--------------+--------+----------+-----------+----------+
```

La table complète des i-noeuds de la partition est constituée de la réunion des tables des i-noeuds de chaque ensemble de blocs.

Chaque ensemble de blocs contient une copie du superbloc et des descripteurs. Ces 2 éléments cruciaux sont donc en plusieurs exemplaires dans la partition.

La partie "descripteurs" de chaque ensemble de blocs contient une copie (pour la redondance) de TOUS les descripteurs (un descripteur pour chaque ensemble de blocs).

Un descripteur d'un ensemble de blocs contient les adresses des blocs de l'ensemble contenant les bitmap et la table des i-noeuds.

Un ensemble de blocs contient la partie de la table des i-noeuds qui concerne des fichiers dont les blocs de données sont eux-mêmes dans cet ensemble de blocs. Ils sont ainsi "proches" physiquement, diminuant les mouvements des têtes de lecture.

Un numéro de "device" repère de façon unique une partition. Un numéro de "i-node" repère de façon unique un fichier sur une partition.

Les fichiers sont EN PLUS, repérés par leurs noms stokés dans l'arborescence des répertoires. Cette arborescence est une structure indépendante de celle des descripteurs, ensembles de blocs et i-noeuds.

Lors d'un accès à un fichier, le système commence par chercher la correspondance nom_de_fichier<–>i-noeud, puis tous les accès aux données sont fait par l'intermédiaire du i-noeud.

– super-bloc –

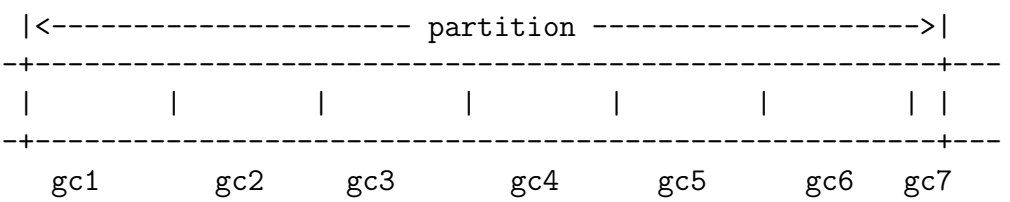
```
Un système de fichiers est décrit par son "super-block" qui contient
des paramètres tels que:
   [] nombre de blocs de données dans le file system
   [] le nombre maximum de fichiers
   [] un pointeur sur la liste des blocs libres (free list)
      ou une bitmap des blocs libres à partir de BSD 4.3
   [] la taille d'un bloc logique (choisie lors de la création
      du file system)
```

Le super-bloc est localisé au début du file-system, avec un certain nombre de copies réparties sur la partition par sécurité.

Depuis BSD 4.3, un bloc disque logique est au moins de 4096 octets afin de pouvoir faire des fichiers de 2**32 octets avec seulement deux niveaux d'indirection.

– structure du système de fichiers : détail d'une partition –

Dans un système u*x traditionnel, la partition était découpée en deux parties : une partie pour les i-nodes et une partie pour les blocs de data des fichiers.



À partir de BSD 4.3, la partition est divisée en "groupes de cylindres" (cylinder groups) comportant plusieurs cylindres consécutifs sur le disque. Chaque groupe contient une copie du super-bloc, une zone pour des i-nodes, une carte de bits (bit-map) indiquant les blocs libres de la zone des blocs de données. Pour chaque groupe de cylindres un nombre FIXE (statique) de inodes est créé lors de la création du file-system. La choix par défaut est de créer un inode pour chaque paquet de 2048 octets dans le groupe de cylindres (avec un maxi de 2048 inodes) avec la supposition que ceci est plus qu'il ne sera jamais nécessaire.

Tous les groupes de cylindres à l'intérieur d'une partition ont la même taille, sauf éventuellement le dernier qui peut être plus petit. Si la place restant pour ce dernier groupe est trop petite cette place est inutilisée. Si nf est un numéro de fragment dans la partition, et nfg le nombre de fragments par groupe, le numéro du groupe qui contient le fragment "nf" est calculé par ng = nf / nfg.

Cette organisation a pour but de répartir les inodes sur tout le disque plutôt que de les grouper tous au début du disque. De plus les routines d'allocation essaient d'allouer les blocs d'un fichier près du inode du fichier afin de diminuer les déplacements des têtes de lecture.

Les copies du super-bloc dans chaque groupe de cylindres ne sont pas toujours situées au début du groupe sinon elles se trouveraient toutes sur le même plateau. Les copies sont décalées à chaque fois, le décalage est d'un peu plus d'une piste par rapport à celui du groupe précédant. La copie suivante est donc un plateau "plus bas" et plus au centre que la précédente.

```
   |<---------------- groupe de cylindres --------------->|
---+-------------------------------------------------------+---
   |  data  | a.SB | cg B |  inode table  |   data         |
---+-------------------------------------------------------+---
data : 16 K octets mini pour décaler les copies de super-blocs
a.SB : alternate super-block (8K: copie du SB de la partition)
cg.B : cylinder group block (1 bloc)
a.SB , cg.B et inide-table sont toujours adjacents

Cas spécial: le cg 0
   |<---------------- groupe de cylindres 0 ------------>|
---+-------------------------------------------------------+---
   |boot| SB| a.SB | cg B |  inode table  |summ.area| data |
---+-------------------------------------------------------+---
les 16 K du début contiennent:
      8K pour le boot bloc (utilisé seulement si ce cg 0 est aussi
         le bloc disque physique 0)
      8K pour le super-bloc du file system de cette partition
le début des data contient:
      une zone résumé d'informations sur chaque grpe de cyl de la
         partition: suite de structures (une par gc):
                     . nbre de répertoires dans le gc
                     . nbre de blocs libres
                     . nbre de inodes inutilisés
                     . nbre de fragments libres


   |<---------------------- super-block ------------------>|
---+-------------------------------------------------------+---
   | données globales  |  tables d'arrangement  |table des |
   | volatiles ou non  |  rotationel            |partitions|
---+-------------------------------------------------------+---
   <---- struct fs ---->                       <-struct pt->

. détails des données globales du SB: #include <sys/fs.h>
. tables d'arrangement rotationel: valeurs précalculées dépendant de
  la géométrie du disque et destinée au calcul du placement optimal
```

```
  des blocs.
. table des partitions: décrit ttes les partitions du disque

   |<------------- bloc du groupe de cylindre ------------>|
---+-------------------------------------------------------+---
   | données du groupe  |  bit map des blocs |pad to|//////|
   | de cylindres       |  libres            |a frag|//////|
---+-------------------------------------------------------+---
   <---- struct cg ---->
   <----------------------- fs_cgsize -------------->
```

La table de bits reflète l'allocation au niveau du fragment (un bit= 1 fragment). Pour trouver un bloc libre on cherche nf fragments libres débutant sur une frontière de bloc. Cette table cartographie la totalité du groupe de cylindre. Les structures de données pré-allouées ont leurs fragments marqués par le programme mkfs de création d'un système de fichiers.

– rotation et positionnement des blocs –

```
Le système de gestion de fichiers essaie d'allouer les nouveaux blocs
d'un fichier dans le même cylindre (pour éviter les mouvements des
têtes) et à des numéros de blocs décalés de façon optimale compte tenu
de la vitesse de rotation du disque.
Ainsi, pour une lecture de 2 blocs:
  - lecture du premier bloc,
        . temps de traitement de l'e/s        | pendant ce temps le
        . traitement de l'interruption        | disque continue sa
        . préparation de la lecture suivante  | rotation
  - lecture du deuxième bloc qui "arrive" sous la tête de lecture au
    bon moment car le nombre de blocs qui le sépare du précédent a été
    calculé pour cela.
```

complément : lecture article

complément : lecture article
http://e2fsprogs.sourceforge.net/ext2intro.html

Design and Implementation of the Second Extended Filesystem

Rémy Card, Laboratoire MASI--Institut Blaise Pascal, and
Theodore Ts'o, Massachussets Institute of Technology, and
Stephen Tweedie, University of Edinburgh

Accessible sur sunserv: http://www4.utc.fr/~sr01/ext2intro.html

4 SR01 2003 - Cours Unix 4 - Introduction aux interfaces graphiques

4.1 Présentation des interfaces graphiques par une suite de programmes simples

À l'aide d'une succession de programmes graphiques auxquels on va ajouter petit à petit de nouvelles fonctions, on va montrer comment sont gérées les interfaces graphiques.

Les programmes de cette série ont été gardés les plus simples possibles (environ 100 lignes de C chacun) afin qu'ils puissent être rapidement commentés et compris.

Ensuite, dans la section suivante, toujours selon la même idée, on construira un tout petit "toolkit" graphique qui permettra de comprendre comment sont construits les toolkits couramment utilisés.

Remarque : sur les dessins ou copies d'écran ci-après, les repères encadrés ("1A", "1B", etc ...) correspondent aux repères de mêmes noms présents dans les listings des programmes sources. Ils permettent d'associer des lignes sources à un effet visuel.

4.2 Premier programme : minix1.c

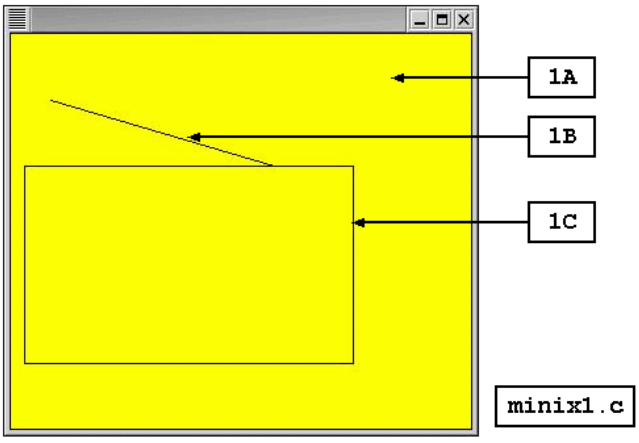


FIG. 20 – L'exécution de minix1.c

minix1.c

minix1.c (suite)

```
 1  /* minix1.c     */
 2  /* un premier exemple de programme graphique utilisant la xlib
 3   * UTC UV SR01 - (c) Michel.Vayssade@utc.fr oct 2003
 4   * solaris : gcc -o minix1 minix1.c -lX11
 5   * linux :  gcc -o minix1 minix1.c -L/usr/X11R6/lib -lX11
 6   */
 7
 8  #include <stdio.h>      /* pour printf() */
 9  #include <signal.h>     /* pour signal() */
10  #include <unistd.h>     /* pour pause() */
11  #include <stdlib.h>
12  #include <X11/Xlib.h>
13
14  /*-- structures de données pour l'utilisation de X11 --*/
15          Display *xdisplay;
16          Window xwindow;
17          int xscreen;
18          unsigned long xforeground, xbackground;
19          XEvent xevent;
20          GC xgc;
21          Colormap cmap;
22          XColor color, colorrgb;
23
24  /* --------------------- main --------------- */
25  void captsig() {        printf("captsig\n");
26                          signal (SIGINT, captsig);
27  }
28
29  main ()
30  {    int x1,y1,x2,y2, x,y,larg,hter, width=350,height=300;
31       char *col;
32       signal (SIGINT, captsig); /* préparer usage de pause() */
33
34       xdisplay = XOpenDisplay (""); /* connect to the X server */
35       if (xdisplay == NULL) {
36               fprintf (stderr, "cannot connect to server\n");
37               exit (EXIT_FAILURE); }
38
39       xscreen= DefaultScreen(xdisplay); /* get default screen */
40
41       /* get black and white representation on current screen */
42       xbackground = BlackPixel (xdisplay, xscreen);
43       xforeground = WhitePixel (xdisplay, xscreen);
44
45       /* Create window at (ix,iy), width 0, height,
46          border width 2, in default root  */
47       xwindow = XCreateSimpleWindow (xdisplay,
48               DefaultRootWindow(xdisplay),10,10, width,height,2,
```

minix1.c (suite)

```
49              xforeground, xbackground);
50      if(xwindow==0) { fprintf (stderr, "cannot open window\n");
51                       exit (EXIT_FAILURE); }
52
53      /* ask for exposure event */
54      XSelectInput(xdisplay, xwindow,
55        ButtonPressMask | ExposureMask | KeyPress | KeyRelease);
56
57      /* pop this window up on the screen */
58      XMapRaised (xdisplay, xwindow);
59
60      /* wait for the window showing up before continuing */
61      XNextEvent (xdisplay, &xevent);
62
63      sleep(2);
64      XMoveWindow (xdisplay,xwindow,50,100); /* déplace fen */
65
66      /* set GC of main rectangle and get color map */
67      xgc= XCreateGC (xdisplay, xwindow, 0, 0);
68      if (DisplayPlanes (xdisplay, xscreen)!= 1) {
69          cmap = DefaultColormap (xdisplay, xscreen); }
70
71  col="yellow";
72  if (XAllocNamedColor(xdisplay, cmap, col, &color, &colorrgb))
73      XSetForeground (xdisplay, xgc, color.pixel);
74  x=0; y=0; larg=width; hter=height;                          1A
75  XFillRectangle (xdisplay, xwindow, xgc, x, y, larg, hter);
76
77  col="black";
78  if (XAllocNamedColor(xdisplay, cmap, col, &color, &colorrgb))
79      XSetForeground (xdisplay, xgc, color.pixel);
80  x1=30; y1=50; x2=200; y2=100;                               1B
81  XDrawLine (xdisplay, xwindow, xgc, x1, y1, x2, y2);
82  XFlush (xdisplay); /* flush X request queue to server */
83
84          pause();            /* attendre ctrl-c */
85          printf("après pause 1\n");
86
87  x=10; y=100; larg=250; hter=150;                            1C
88  XDrawRectangle (xdisplay, xwindow, xgc, x, y, larg, hter);
89
90  XFlush (xdisplay); /* flush X request queue to server */
91
92          pause();            /* attendre ctrl-c */
93          printf("après pause 2\n");
94
95          XDestroyWindow(xdisplay, xwindow);
```

minix1.c (suite)

```
96          XCloseDisplay (xdisplay);
97          exit(EXIT_SUCCESS);
98  }
```

4.3 Programme : minix2.c

Le programme minix2.c ajoute à minix1 l'écriture de chaines de catactères et il encapsule dans des fonctions plus simples à utiliser les appels à la bibliothèque Xlib.

Les fonctions encapsulées sont regroupées dans un fichier mx2.c afin de clarifier le code du programme principal.

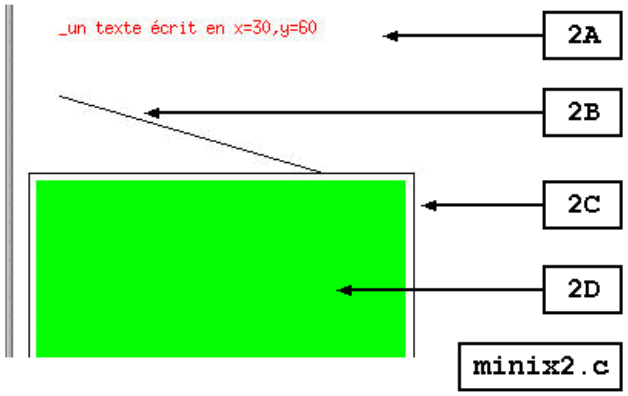


FIG. 21 – L'exécution de minix2.c

minix2.c

```
 1  /* minix2.c     */
 2  /* un 2ème exemple de programme graphique utilisant la xlib
 3   * UTC UV SR01 - (c) Michel.Vayssade@utc.fr oct 2003
 4   *          gcc -c mx2.c
 5   * solaris : gcc -o minix2 minix2.c mx2.o -lX11
 6   * linux :  gcc -o minix2 minix2.c mx2.o -L/usr/X11R6/lib -lX11
 7   */
 8
 9  #include <stdio.h>      /* pour printf() */
10  #include <signal.h>     /* pour signal() */
11  #include <unistd.h>     /* pour pause() */
12  #include <stdlib.h>
13  #include <X11/Xlib.h>
14
15  #define red     0
```

minix2.c (suite)

```
16 #define green   1
17 #define blue    2
18 #define yellow  3
19 #define white   4
20 #define black   5
21
22 /* ---------------------- main ----------------- */
23 void captsig() {
24         printf("captsig\n");
25         signal (SIGINT, captsig);
26 }
27
28 main ()
29 {   int lar=250, htr=200, lr1,ht1,i;
30     lr1 = lar-10; ht1 = htr-10;
31     signal (SIGINT, captsig); /* */
32
33     initfx(450,400); /* init fen.X, lar 450, hter 400 */
34     fillrec(0,0,450,400,white); /* fen. en blanc */
35
36     flushfx();       /* vider le buffer X vers l'écran */
37     i=pause();       /* attendre ctrl-c */
38     printf("après pause 1\n");
39
40     drawstr(30,60,"_un texte écrit en x=30,y=60",28,red);   2A
41     drawvec(30,100,200,150,black); /* vect x1,y1-x2,y2 */   2B
42
43     flushfx();       /* vider le buffer X vers l'écran */
44     pause();         /* attendre ctrl-c */
45     printf("après pause 2\n");
46
47     drawrec(10,150,lar,htr,black);/* rect x1,y1,lar,hter */ 2C
48
49     flushfx();       /* vider le buffer X vers l'écran */
50     pause();         /* attendre ctrl-c */
51     printf("après pause 3\n");
52
53     fillrec(15,155,lr1,ht1,green); /* rectangle en vert */  2D
54
55     flushfx();       /* vider le buffer X vers l'écran */
56     pause();         /* attendre ctrl-c */
57     printf("après pause 4\n");
58
59     detruitfx();         /* detruire la fenetre X */
60     exit(EXIT_SUCCESS);
61 }
```

mx2.c

```
 1  /* mx2.c        */
 2  /* encapsulation fonctions graphiques de base de la xlib
 3   * UTC UV SR01 - (c) Michel.Vayssade@utc.fr oct 2003
 4   * gcc -c mx2.c
 5   */
 6
 7  #include <stdio.h>      /* pour printf() */
 8  #include <signal.h>     /* pour signal() */
 9  #include <unistd.h>     /* pour pause() */
10  #include <stdlib.h>
11  #include <X11/Xlib.h>
12
13  #define red     0
14  #define green   1
15  #define blue    2
16  #define yellow  3
17  #define white   4
18  #define black   5
19  char *couleur[9] = {    "red","green","blue",
20                          "yellow","white","black",
21                          "purple","cyan","gold"  };
22
23  /*-- structures de données pour l'utilisation de X11 --*/
24          Display *xdisplay;
25          Window xwindow;
26          int xscreen;
27          unsigned long xforeground, xbackground;
28          XEvent xevent;
29          GC xgc;
30          Colormap cmap;
31          XColor color, colorrgb;
32
33  /* ------------- fonctions de dessin Xlib ----------- */
34
35  drawstr (int x, int y, char *str, int lng, int col)
36  {
37  /* Draw string at (x,y) in col */
38  if (XAllocNamedColor (xdisplay, cmap, couleur[col],
39           &color, &colorrgb))
40      XSetForeground (xdisplay, xgc, color.pixel);
41  XDrawString (xdisplay, xwindow, xgc, x, y, str, lng);
42  }
43
44  drawrec (int x, int y, int larg, int hter, int col)
45  {
46  /* Draw rect at (x,y) in col */
47  if (XAllocNamedColor (xdisplay, cmap, couleur[col],
48           &color, &colorrgb))
```

mx2.c (suite)

```
 49      XSetForeground (xdisplay, xgc, color.pixel);
 50  XDrawRectangle (xdisplay, xwindow, xgc, x, y, larg, hter);
 51  }
 52
 53  drawvec (int x1, int y1, int x2, int y2, int col)
 54  {
 55  /* Draw vector in col */
 56  if (XAllocNamedColor (xdisplay, cmap, couleur[col],
 57            &color, &colorrgb))
 58      XSetForeground (xdisplay, xgc, color.pixel);
 59  XDrawLine (xdisplay, xwindow, xgc, x1, y1, x2, y2);
 60  }
 61
 62  fillrec (int x, int y, int larg, int hter, int col)
 63  {
 64  /* Fill rectangle at (x,y), width , height, */
 65  if (XAllocNamedColor (xdisplay, cmap, couleur[col],
 66            &color, &colorrgb))
 67     XSetForeground (xdisplay, xgc, color.pixel);
 68  XFillRectangle (xdisplay, xwindow, xgc, x, y, larg, hter);
 69  }
 70
 71  flushfx ()
 72  {    XFlush (xdisplay); /* flush X request queue to server */
 73  }
 74
 75  detruitfx()
 76  {
 77     XDestroyWindow(xdisplay, xwindow);
 78     XCloseDisplay (xdisplay);
 79  }
 80
 81  /* ------------- initialisation de la Xlib ---------- */
 82
 83  initfx(int width, int height)
 84  {
 85     xdisplay = XOpenDisplay (""); /* connect to X server */
 86     if (xdisplay == NULL) {
 87           fprintf (stderr, "cannot connect to server\n");
 88           exit (EXIT_FAILURE); }
 89
 90     xscreen=DefaultScreen(xdisplay);/* get default screen */
 91
 92     /* get black,white representation on current screen */
 93     xbackground = BlackPixel (xdisplay, xscreen);
 94     xforeground = WhitePixel (xdisplay, xscreen);
 95
 96     /* Create window at (ix,iy), width 450, height 250,
```

mx2.c (suite)

```
 97        border width 2, in default root  */
 98     xwindow = XCreateSimpleWindow (xdisplay,
 99           DefaultRootWindow(xdisplay), 10,10,width,height,2,
100           xforeground, xbackground);
101     if (xwindow == 0) {
102             fprintf (stderr, "cannot open window\n");
103             exit (EXIT_FAILURE); }
104
105     /* ask for exposure event */
106     XSelectInput(xdisplay, xwindow,
107     ButtonPressMask | ExposureMask | KeyPress | KeyRelease);
108
109     /* pop this window up on the screen */
110     XMapRaised (xdisplay, xwindow);
111
112     /* wait for the window showing up before continuing */
113     XNextEvent (xdisplay, &xevent);
114
115     /* set graphics context of main rectangle */
116     xgc= XCreateGC (xdisplay, xwindow, 0, 0);
117     if (DisplayPlanes (xdisplay, xscreen)!= 1) {
118         cmap = DefaultColormap (xdisplay, xscreen);
119     }
120     return 0;
121 }
122 /* ----------------- fin de mx2.c --------------- */
123
```

4.4 Programme : minix3.c

Le programme minix3.c ajoute à minix2 la gestion évènements ButtonPress, Expose, ... ainsi que la gestion des évènements KeyPress, KeyRelease (dans la suite on utilisera plus ces évènements KeyPress, KeyRelease).

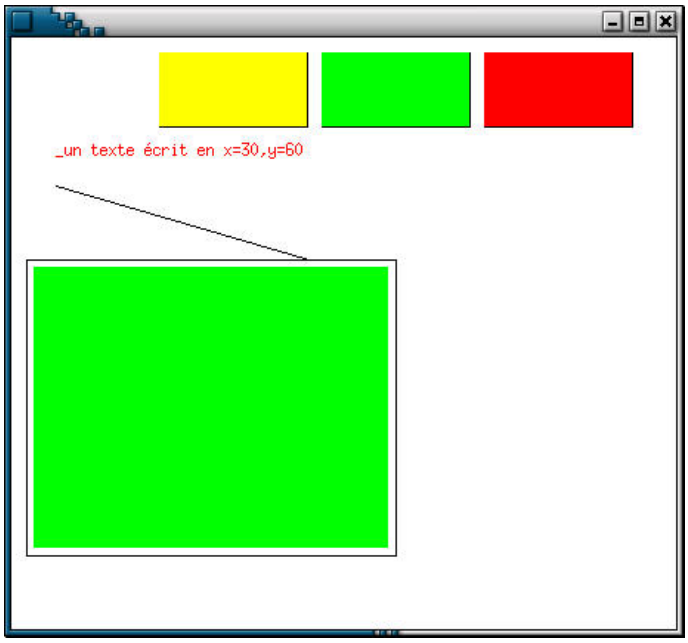


FIG. 22 – L'exécution de minix3.c

minix3.c

```
 1  /* minix3.c     */
 2  /* un 3ème exemple de programme graphique utilisant la xlib
 3   * démonstration de la gestion des évènements d'entrée
 4   * UTC UV SR01 - (c) Michel.Vayssade@utc.fr oct 2003
 5   *          gcc -c mx2.c
 6   * solaris : gcc -o minix3 minix3.c mx2.o -lX11
 7   * linux :
 8   gcc -o minix3 minix3.c mx2.o -L/usr/X11R6/lib -lX11
 9   */
10
11  #include <stdio.h>      /* pour printf() */
12  #include <signal.h>     /* pour signal() */
13  #include <unistd.h>     /* pour pause() */
14  #include <stdlib.h>
15  #include <X11/Xlib.h>
16  #include <X11/Xutil.h>  /* : #define IsFunctionKey(keysym) */
17  #include <X11/keysym.h> /* idem (include keysymdef.h) */
18
19  #define red     0
20  #define green   1
21  #define blue    2
22  #define yellow  3
23  #define white   4
24  #define black   5
25
26  extern Display *xdisplay;       /* définis dans mx2.c */
```

minix3.c (suite)

```
27  extern XEvent xevent;
28
29  /* -------------------- main ---------------- */
30  void captsig() {
31          printf("captsig\n");
32          signal (SIGINT, captsig);
33  }
34  void redraw() {
35          printf("redraw pas encore implémentée\n");
36  }
37
38  int traiter_clic(int x, int y) {
39          printf("traiter_clic : clic en x,y=%d %d\n",x,y);
40          if (x>320 && x<420 && y>10 && y<60) return -1;
41          return 0;
42  }
43
44
45  main ()
46  {   int lar=250, htr=200, lr1,ht1,i;
47
48      /* data pour gestion évènements clavier KeyPress */
49      KeySym ksym, ksym1;
50      char *ksymname;
51      char buf[10];
52      int lbuf=10;
53      XComposeStatus istat;
54
55      lr1 = lar-10; ht1 = htr-10;
56      signal (SIGINT, captsig); /* */
57
58      initfx(450,400); /* init fen X, larg 450, hter 400 */
59      fillrec(0,0,450,400,white); /* colorier fen. en blanc */
60      drawstr(30,60,"_un texte écrit en x=30,y=60",28,red);
61      drawvec(30,100,200,150,black); /* vect x1,y1-x2,y2 */
62      fillrec(15,155,lr1,ht1,green); /* rect en vert */
63
64      drawrec(100,10,100,50,black);
65      drawrec(210,10,100,50,black);
66      drawrec(320,10,100,50,black);
67      fillrec(100,10,100,50,yellow);
68      fillrec(210,10,100,50,green);
69      fillrec(320,10,100,50,red);
70      flushfx();          /* vider le buffer X vers l'écran */
71      printf("pause 1, attendre ctrl-c\n");
72      pause();            /* attendre ctrl-c */
73      printf("après pause 1\n");
74
```

minix3.c (suite)

```
 75  /* --------------- gestion évènements ----------- */
 76  /* l'appel XNextEvent est synchrone : on reste dans XNextEvent
 77     jusqu'à ce qu'un évènement se produise.
 78  */
 79
 80      while (1) {
 81      XNextEvent (xdisplay, &xevent);
 82        switch (xevent.type) {
 83           case Expose :
 84                redraw ();
 85                break;
 86           case KeyPress :
 87                printf("- press   -");
 88                goto key;
 89           case KeyRelease :
 90                printf("- release -");
 91      key :
 92           ksym = XLookupKeysym (&xevent.xkey,0);
 93           ksymname = XKeysymToString (ksym);
 94           XLookupString (&xevent.xkey,buf,lbuf,&ksym1,&istat);
 95
 96           printf("-keycode =%d -keysym = %d keysymname = %s\
 97   -keysym1 = %d char=%s-\n",
 98           xevent.xkey.keycode, ksym, ksymname, ksym1, buf);
 99
100           if (IsFunctionKey(ksym))
101             printf("-key %d is a function key-\n",
102                    xevent.xkey.keycode);
103
104                  break;
105           case ButtonPress :
106                i= 0;
107                i= traiter_clic (xevent.xkey.x, xevent.xkey.y);
108                break;
109        } /* fin switch */
110        if (i==-1) break;
111      } /* fin while(1) */
112
113      printf("sortir après pause 2\n");
114      flushfx();           /* vider le buffer X vers l'écran */
115      pause();             /* attendre ctrl-c */
116      printf("après pause 2\n");
117
118      detruitfx();         /* detruire la fenetre X */
119      exit(EXIT_SUCCESS);
120  }
```

```
- press   --keycode =24 -keysym = 97 keysymname = a -keysym1 = 97 char=a--
```

```
- release --keycode =24 -keysym = 97 keysymname = a -keysym1 = 97 char=a--
- press   --keycode =50 -keysym = 65505 keysymname = Shift_L -keysym1 = 65505 c
- press   --keycode =45 -keysym = 107 keysymname = k -keysym1 = 75 char=K--
- release --keycode =45 -keysym = 107 keysymname = k -keysym1 = 75 char=K--
- release --keycode =50 -keysym = 65505 keysymname = Shift_L -keysym1 = 65505 c
- press   --keycode =37 -keysym = 65507 keysymname = Control_L -keysym1 = 65507
- press   --keycode =56 -keysym = 98 keysymname = b -keysym1 = 98 char=--
- release --keycode =56 -keysym = 98 keysymname = b -keysym1 = 98 char=--
- release --keycode =37 -keysym = 65507 keysymname = Control_L -keysym1 = 65507
- press   --keycode =113 -keysym = 65406 keysymname = Mode_switch -keysym1 = 65
- press   --keycode =17 -keysym = 95 keysymname = underscore -keysym1 = 92 char
- release --keycode =17 -keysym = 95 keysymname = underscore -keysym1 = 92 char
- release --keycode =113 -keysym = 65406 keysymname = Mode_switch -keysym1 = 65
```

4.5 Programme : minix4.c

Le programme minix4.c ajoute à minix3 la gestion évènements MotionNotify. Ces évènements sont envoyés à l'application chaque fois que l'on bouge la souris.

Le vecteur est redessiné à chaque évènement MotionNotify, ce qui explique le faisceau de traits partant du point de clic initial.

Le fichier mx2.c est complété et devient mix.c qui nous servira de "bibliothèque de base" dans la suite de l'exposé.

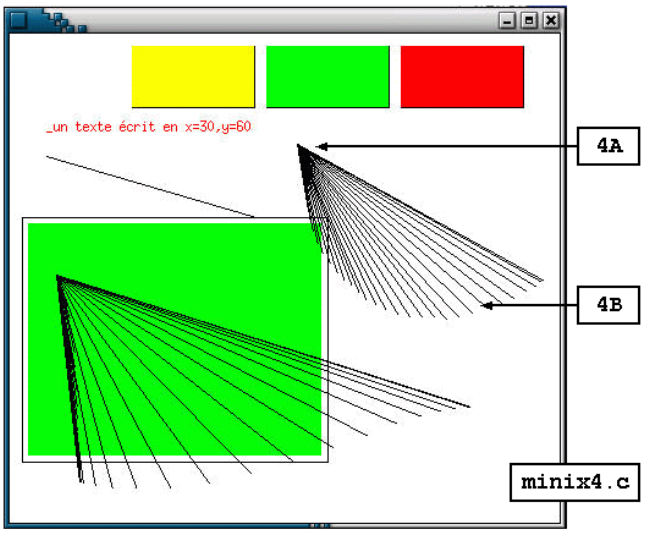


FIG. 23 – L'exécution de minix4.c

minix4.c

```
1  /* minix4.c     */
2  /* un 4ème exemple de programme graphique utilisant la xlib
```

minix4.c (suite)

```
 3   * démonstration de la gestion des évènements MotionNotify
 4   * UTC UV SR01 - (c) Michel.Vayssade@utc.fr oct 2003
 5   *          gcc -c mix.c
 6   * solaris : gcc -o minix4 minix4.c mix.o -lX11
 7   * linux :  gcc -o minix4 minix4.c mix.o -L/usr/X11R6/lib -lX11
 8   */
 9
10  #include <stdio.h>      /* pour printf() */
11  #include <signal.h>     /* pour signal() */
12  #include <unistd.h>     /* pour pause() */
13  #include <stdlib.h>
14  #include <X11/Xlib.h>
15
16  #define red     0
17  #define green   1
18  #define blue    2
19  #define yellow  3
20  #define white   4
21  #define black   5
22
23  extern Display *xdisplay;  /* définis dans mix.c */
24  extern XEvent xevent;
25
26  /* --------------------- main ---------------- */
27  void captsig() {
28          printf("captsig\n");
29          signal (SIGINT, captsig);
30  }
31  void redraw() {
32          printf("redraw pas encore implémentée\n");
33  }
34
35  int traiter_clic(int x, int y) {
36          printf("traiter_clic : clic en x,y=%d %d\n",x,y);
37          if (x>320 && x<420 && y>10 && y<60) return -1;
38          return 0;
39  }
40
41
42  main ()
43  {   int lar=250, htr=200, lr1,ht1,i;
44      int x,y, x0,y0;
45
46      lr1 = lar-10; ht1 = htr-10;
47      signal (SIGINT, captsig); /* */
48
49      initfx(450,400);  /* init fen X, larg 450, hter 400 */
50      fillrec(0,0,450,400,white); /* colorier fen. en blanc */
```

minix4.c (suite)

```
51      drawstr(30,60,"_un texte écrit en x=30,y=60",28,red);
52      drawvec(30,100,200,150,black); /* vect x1,y1-x2,y2 */
53      fillrec(15,155,lr1,ht1,green); /* rectangle en vert */
54
55      drawrec(100,10,100,50,black);
56      drawrec(210,10,100,50,black);
57      drawrec(320,10,100,50,black);
58      fillrec(100,10,100,50,yellow);
59      fillrec(210,10,100,50,green);
60      fillrec(320,10,100,50,red);
61      flushfx();       /* vider le buffer X vers l'écran */
62      printf("pause 1, attendre ctrl-c\n");
63      pause();             /* attendre ctrl-c */
64      printf("après pause 1\n");
65
66  /* -------------- gestion évènements ----------- */
67
68      while (1) {
69      XNextEvent (xdisplay, &xevent);
70        switch (xevent.type) {
71          case Expose :
72                  redraw ();
73                  break;
74          case MotionNotify :
75                  x = xevent.xbutton.x;                          4B
76                  y = xevent.xbutton.y;
77                  drawvec(x0,y0,x,y,black);
78                  break;
79
80          case ButtonPress :
81                  x0 = xevent.xbutton.x;                         4A
82                  y0 = xevent.xbutton.y;
83                  i= 0;
84                  i= traiter_clic(xevent.xkey.x, xevent.xkey.y);
85                  break;
86        } /* fin switch */
87        if (i==-1) break;
88      } /* fin while(1) */
89
90      printf("sortir après pause 2\n");
91      flushfx();        /* vider le buffer X vers l'écran */
92      pause();          /* attendre ctrl-c */
93      printf("après pause 2\n");
94      detruitfx();         /* detruire la fenetre X */
95      exit(EXIT_SUCCESS);
96  }
```

mix.c

```
  1  /* mix.c        */
  2  /* encapsulation fonctions graphiques de base de la xlib
  3   * = mx2.c + qques fonctions (arc) et évènements  */
 71
 72  drawarc (int x, int y, int larg, int hter,
 73          int ang1, int ang2, int col)
 74  {
 75  /* Draw arc at (x,y) in white */
 76  if (XAllocNamedColor (xdisplay, cmap, couleur[col],
 77          &color, &colorrgb))
 78      XSetForeground (xdisplay, xgc, color.pixel);
 79  XDrawArc (xdisplay,xwindow,xgc, x,y,larg,hter,ang1,ang2);
 80  }
 81
 82  fillarc (int x, int y, int larg, int hter, int col)
 83  { /* Fill arc at (x,y), width , height, with col */
 84  if (XAllocNamedColor (xdisplay, cmap, couleur[col],
 85          &color, &colorrgb))
 86     XSetForeground (xdisplay, xgc, color.pixel);
 87  XFillArc (xdisplay,xwindow,xgc, x,y,larg,hter, 0,360*64);
 88  }
 89
102  initfx(int width, int height)
103  {
123
124      /* ask for exposure event */
125      XSelectInput(xdisplay, xwindow,
126      ButtonPressMask | ExposureMask | ButtonMotionMask | \
127      ButtonReleaseMask );
128
143
144  /* ----------------- fin de mix.c -------------- */
145
```

4.6 Programme : minix5.c

Le programme minix5.c ajoute dans traitement de l'évènement MotionNotify, l'effacement de l'ancienne position du vecteur.

Mais la méthode utilisée est trop simpliste : elle détruit les objets graphiques déja présents dans la fenêtre.

De plus, minx5 ne gère toujours pas le redessin au cas où la fenêtre est recouverte puis découverte.

minix5.c

```
 1  /* minix5.c     */
44      int x,y, x0,y0, x1,y1;
74
75            case MotionNotify :
76                  x = xevent.xbutton.x;
77                  y = xevent.xbutton.y;
78                  drawvec(x0,y0,x1,y1,white); /* efface ancien */
79                  drawvec(x0,y0,x,y,black);   /* trace nouveau */
80                  x1=x; y1=y; /* sav new end point */
81                  break;
82
```

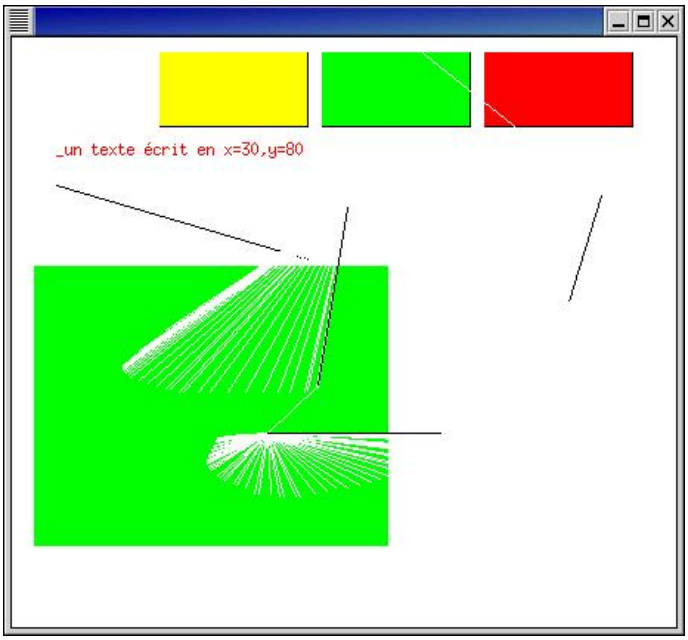


FIG. 24 – L'exécution de minix5.c

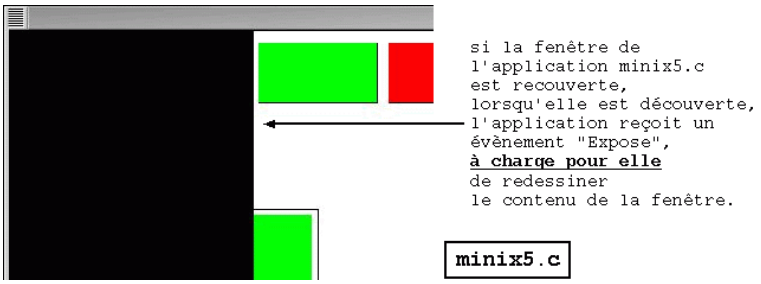


FIG. 25 – minix5.c ne gère pas la mise à jour

4.7 Programme : minix6.c

Le programme minix6.c ajoute le traitement des objets graphiques à retracer en cas de réception de l'évènement Expose.

Mais la méthode utilisée est aussi trop simpliste : si des objets sont abimés par un vecteur que l'on fait bouger, ils ne sont redessinés correctement que lors d'un rafraichissement (pour le voir, provoquer l'envoi d'un Expose vers l'application).

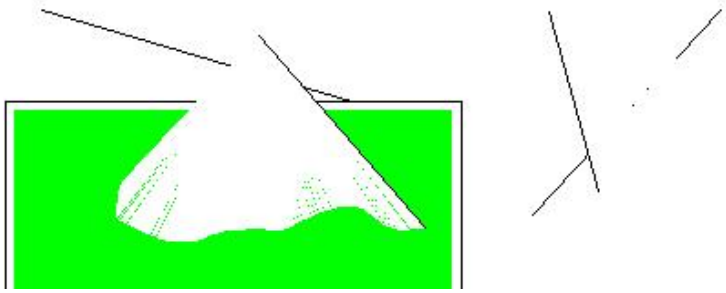


FIG. 26 – L'exécution de minix6.c (1/2)

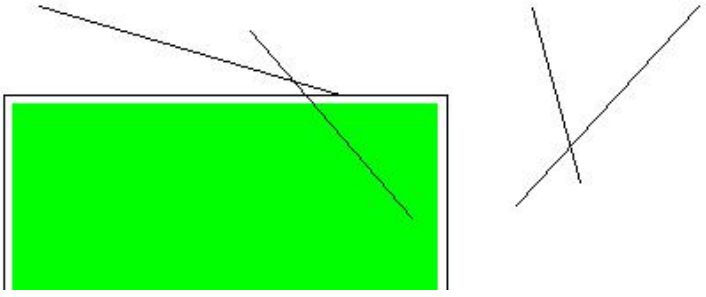


FIG. 27 – L'exécution de minix6.c (2/2)

minix6.c

```
1  /* minix6.c     */
```

minix6.c (suite)

```
 2  /* un 6ème exemple de programme graphique utilisant la xlib
 3   * complément à la gestion des évènements MotionNotify
 4   * gérer les objets à retracer en cas d'évènmt expose */
26
27  /*-- structure de données pour stocker lignes dessinées --*/
28  #define N_LIGNES (20)
29  typedef struct
30  { int x1, y1, x2, y2; } Typeligne;
31  Typeligne Lignes[N_LIGNES];
32  int Nblignes = 0;
33  int index = 0;
34
35  /* -------------------- main ---------------- */
36  void captsig() {
37          printf("captsig\n");
38          signal (SIGINT, captsig);
39  }
40
41  void drawini() {
42     int lar=250, htr=200, lr1,ht1;
43     lr1 = lar-10; ht1 = htr-10;
44     fillrec(0,0,450,400,white); /* colorier fen. en blanc */
45     drawstr(30,80,"_un texte écrit en x=30,y=80",28,red);
46     drawvec(30,100,200,150,black); /* vect x1,y1-x2,y2 */
47     drawrec(10,150,lar,htr,black); /* rect x1,y1-lar,hter*/
48     fillrec(15,155,lr1,ht1,green); /* rectangle en vert */
49
50     drawrec(100,10,100,50,black);
51     drawrec(210,10,100,50,black);
52     drawrec(320,10,100,50,black);
53     fillrec(100,10,100,50,yellow);
54     fillrec(210,10,100,50,green);
55     fillrec(320,10,100,50,red);
56  }
57
58  void redraw() { int i;
59       drawini();
60       for (i=0; i<index; i++) {
61          drawvec (Lignes[i].x1, Lignes[i].y1,
62                   Lignes[i].x2, Lignes[i].y2, black ); }
63  }
64
65  int traiter_clic(int x, int y) {
66          printf("traiter_clic : clic en x,y=%d %d\n",x,y);
67          if (x>320 && x<420 && y>10 && y<60) return -1;
68          return 0;
69  }
70
```

minix6.c (suite)

```
 71
 72  main ()
 73  {   int x,y, x0,y0, x1,y1, i;
 74      signal (SIGINT, captsig); /* */
 75
 76      initfx(450,400);  /* init fen X, larg 450, hter 400 */
 77      drawini();        /* dessin initial */
 78      flushfx();        /* vider buffer X vers l'écran */
 79
 80      printf("pause 1, attendre ctrl-c\n");
 81      pause();             /* attendre ctrl-c */
 82      printf("après pause 1\n");
 83
 84  /* --------------- gestion évènements ------------ */
 85
 86      while (1) {
 87      XNextEvent (xdisplay, &xevent);
 88        switch (xevent.type) {
 89          case Expose :
 90                 redraw ();
 91                 break;
 92
 93          case MotionNotify :
 94                 x = xevent.xbutton.x;
 95                 y = xevent.xbutton.y;
 96                 drawvec(x0,y0,x1,y1,white); /* efface ancien */
 97                 drawvec(x0,y0,x,y,black);   /* trace nouveau */
 98                 x1=x; y1=y; /* sav new end point */
 99                 break;
100
101          case ButtonPress :
102                 x0 = xevent.xbutton.x;
103                 y0 = xevent.xbutton.y;
104                 x1=x0; y1=y0;
105                 i= 0;
106                 i= traiter_clic (xevent.xkey.x, xevent.xkey.y);
107                 break;
108
109          case ButtonRelease :
110                 Lignes[index].x1 = x0;
111                 Lignes[index].y1 = y0;
112                 Lignes[index].x2 = x1;
113                 Lignes[index].y2 = y1;
114                 index++;
115                 printf("index=%d\n",index);
116                 break;
117
118       } /* fin switch */
```

minix6.c (suite)

```
119         if (i==-1) break;
120       } /* fin while(1) */
121
```

4.8 Programme : minix7.c

Le programme minix7.c supprime l'effet d'effacement des objets graphiques en appelant la fonction redraw() à chaque évènement "motion".

Mais la méthode utilisée est encore une fois trop simpliste : le redessin complet à chaque évènement "motion" (qui sont très fréquents) provoque un effet désagréable de clignotement ("flicker").

minix7.c

```
  1  /* minix7.c     */
  2  /* un 7ème exemple de programme graphique utilisant la xlib
  3   * complément à la gestion des évènements MotionNotify
  4   * gérer les objets à retracer en cas d'évènmt expose
  5   * mise à jour de la fenêtre à chq évènmt "Motion"    */
 93
 94           case MotionNotify :
 95                 x = xevent.xbutton.x;
 96                 y = xevent.xbutton.y;
 97                 /* drawvec(x0,y0,x1,y1,white); efface ancien */
 98                 redraw();
 99                 drawvec(x0,y0,x,y,black);   /* trace nouveau */
100                 x1=x; y1=y; /* sav new end point */
101                 break;
102
```

4.9 Programme : tstpix.c

Pour supprimer l'effet de clignotement ("flicker") du programme minix7, il va falloir recourir à une technique complètement différente : celle des "pixmap" et du travail en double buffer.

Avant d'en voir l'application on va montrer sur un programme test l'utilisation des pixmaps.

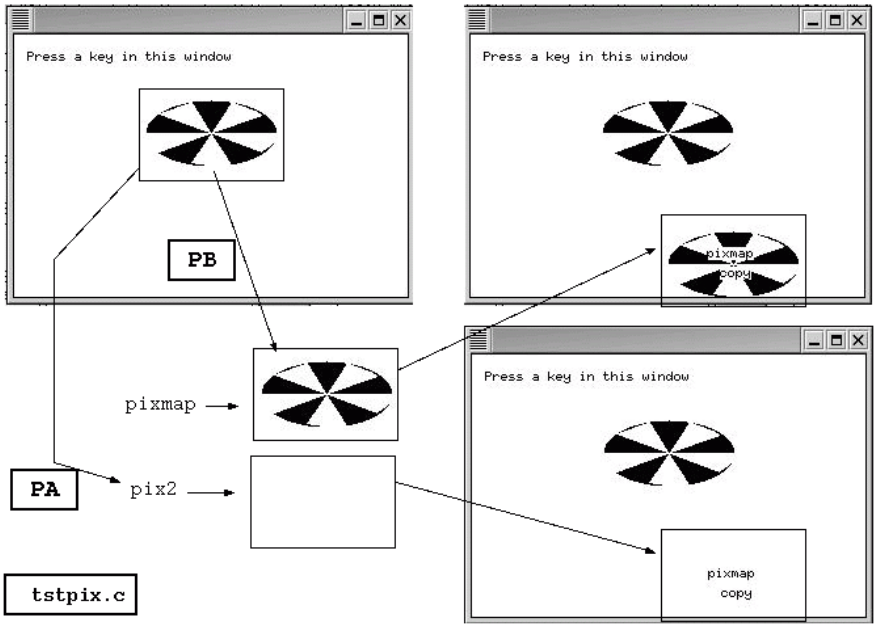


FIG. 28 – Fonctionnement de tstpix.c

tstpix.c

```
 1  /* tstpix.c     */
 2  /* un exemple de programme graphique utilisant les pixmaps  */
 3  /* UTC UV SR01 - (c) Michel.Vayssade@utc.fr oct 2003
 4   * solaris : gcc -o tstpix tstpix.c -lX11
 5   * linux :  gcc -o tstpix tstpix.c -L/usr/X11R6/lib -lX11
 6   */
 7  #include <X11/cursorfont.h>
 8  #include <stdio.h>
 9  #include <X11/Xlib.h>
10  #include <X11/keysym.h>
11        Display *display;
12        Window  win1;
13        XSetWindowAttributes attributes;
```

tstpix.c (suite)

```
14        XFontStruct *fontinfo;
15        GC gr_context1;
16        XArc arcs[10];
17        Pixmap pixmap, pix2;
18        Visual *visual;
19        int screen, depth, i, togle=1, firstexp=1;
20        KeySym key;
21
22  main ()
23  {     XGCValues gr_values;
24        XEvent event;
25
26        setbuf (stdout, NULL); /* supprimer bufferisation */
27        setbuf (stderr, NULL); /* pour affichage immédiat */
28        display = XOpenDisplay(NULL);
29        screen = DefaultScreen(display);
30        visual = DefaultVisual(display,screen);
31        depth  = DefaultDepth(display,screen);
32        attributes.background_pixel = XWhitePixel(display,screen);
33        attributes.border_pixel     = XBlackPixel(display,screen);
34        attributes.override_redirect= 0;
35        for(i=0;i<10;i++){  /* préparer le dessin des arcs */
36             arcs[i].x = 100;arcs[i].y = 50;
37             arcs[i].width = 100;arcs[i].height = 50;  }
38        for(i=0;i<5;i++){
39             arcs[i].angle1 = 72*64*i;
40             arcs[i].angle2 = 35*64;  }
41        for(i=5;i<10;i++){
42             arcs[i].angle1 = 72*64*i + 36*64;
43             arcs[i].angle2 = 35*64;  }
44
45        win1= XCreateWindow(display, XRootWindow(display,screen),
46              200,200, 300,200,5, depth, InputOutput, visual,
47              CWBackPixel | CWBorderPixel | CWOverrideRedirect,
48              &attributes);
49
50        XSelectInput(display,win1,
51           ExposureMask | ButtonPressMask | KeyPressMask);
52
53        pixmap = XCreatePixmap(display,win1,200,100,depth);
54        pix2 =   XCreatePixmap(display,win1,200,100,depth);
55
56        fontinfo = XLoadQueryFont(display,"6x10");
57        gr_values.font =   fontinfo->fid;
58        gr_values.function =   GXcopy;
59        gr_values.plane_mask = AllPlanes;
60        gr_values.foreground = BlackPixel(display,screen);
61        gr_values.background = WhitePixel(display,screen);
```

tstpix.c (suite)

```
 62       gr_context1=XCreateGC(display,win1,
 63               GCFont | GCFunction | GCPlaneMask | \
 64               GCForeground | GCBackground, &gr_values);
 65       XDefineCursor(display,win1,XCreateFontCursor(display,XC_heart));
 66       XMapWindow(display,win1);
 67
 68  /* XCopyArea(display, src, dest, gc,
 69                  src_x, src_y, width, height, dest_x, dest_y); */
 70  do{
 71    XNextEvent(display,&event);
 72    if (event.type == Expose){
 73      if (firstexp==1) {
 74        printf("firstexp : sleep 5 sec. \n"); sleep(5);
 75        /* copier la zone blanche dans pix2 */                          PA
 76        XCopyArea(display,win1,pix2,gr_context1,50,25,200,100,0,0);
 77        firstexp=0;    }
 78
 79        draw_ellipse(); /* dessiner les arcs dans gr_context1 */
 80        /* et copier la zone avec dessin dans pixmap */                 PB
 81        XCopyArea(display,win1,pixmap,gr_context1,50,25,200,100,0,0);
 82
 83        XSetFunction(display,gr_context1,GXinvert);
 84        XDrawImageString(display,pixmap,gr_context1,80,45,"pixmap",6);
 85        XDrawImageString(display,pixmap,gr_context1,90,60,"copy",4);
 86        XDrawImageString(display,pix2  ,gr_context1,80,45,"pixmap",6);
 87        XDrawImageString(display,pix2  ,gr_context1,90,60,"copy",4);
 88        XSetFunction(display,gr_context1,GXcopy);
 89        XDrawString(display,win1,gr_context1,10,20,
 90            "Press a key in this window",26);
 91    }
 92
 93    if (event.type == KeyPress){
 94       key = XLookupKeysym(&event.xkey, 0);
 95       if(togle==1) { togle=2;
 96          XCopyArea(display,pixmap,win1,gr_context1,
 97                  0,0,200,100,100,125);
 98       } else { togle=1;
 99          XCopyArea(display,pix2,win1,gr_context1,
100                  0,0,200,100,100,125);
101       }
102    }
103  }while  (key != XK_q);
104
105       XCopyArea(display,pixmap,win1,gr_context1,
106                  0,0,200,100,100,125);
107       XDrawString(display,win1,gr_context1,10,32,
108                   "Now press a key to exit",23);
109       XFlush(display);
```

tstpix.c (suite)

```
110
111       do{ XNextEvent(display,&event);
112       }while  (event.type!=KeyPress);
113       printf("closing display\n");
114       XCloseDisplay(display);
115  }
116
117  draw_ellipse() /* dessiner les arcs dans gr_context1 */
118  {
119       XSetArcMode(display,gr_context1,ArcPieSlice);
120       XFillArcs(display,win1,gr_context1,arcs,5);
121       XSetArcMode(display,gr_context1,ArcChord);
122       XFillArcs(display,win1,gr_context1,arcs+5,5);
123  }
```

4.10 Programme : minixdb.c

En utilisant les pixmaps, minixdb va permettre de supprimer l'effet de clignotement ("flicker") du programme minix7.

minixdb.c

```
  1  /* minixdb.c    */
  2  /* l'exemple minix6.c modifié pour fonctionner en
  3   * "double buffer" en utilisant les pixmaps */
 22
 23  extern Display *xdisplay;       /* définis dans mix.c */
 24  extern XEvent xevent;
 25  extern int xscreen;
 26  extern Window xwindow;  /* XCopyArea en a besoin */
 27  extern GC xgc;          /* idem */
 28
 36
 37  /*-- structure de données pour gérer les pixmaps --*/
 38        XSetWindowAttributes attributes;
 39        XFontStruct *fontinfo;
 40        Pixmap pix2; /* pixmap pour sauver état fenêtre */
 41        int depth;
 42        XGCValues gr_values;
 43        XGCValues gcval;
 44
 79  main ()
 82
 83      initfx(450,400);   /* init fen X, larg 450, hter 400 */
```

minixdb.c (suite)

```
 84
 85      /* init pixmap sauvegarde état fenêtre, et prépare GC */
 86      depth = DefaultDepth(xdisplay,xscreen);
 87      pix2 =  XCreatePixmap(xdisplay,xwindow, 450,400, depth);
 88          XGetGCValues (xdisplay, xgc, -1, &gcval);
 89       fontinfo = XLoadQueryFont(xdisplay,"6x10");
 90       gcval.font =       fontinfo->fid;
 91       gcval.function =   GXcopy;
 92       gcval.plane_mask = AllPlanes;
 93       gcval.foreground = BlackPixel(xdisplay,xscreen);
 94       gcval.background = WhitePixel(xdisplay,xscreen);
 95          XChangeGC (xdisplay, xgc, GCFunction, &gcval);
 96
 97      drawini();           /* dessin initial */
 98
 99     /* sauver aspect fenêtre;
100        le premier expose a été attendu dans initfx() */
101     XCopyArea(xdisplay,xwindow,pix2, xgc, 0,0, 450,400,0,0);
102     /* faire le flush APRÈS le XCopyArea sinon on copie
103        une zone noire si la fenêtre est cachée */
104
105      flushfx();  /* vider le buffer X vers l'écran */
106
112
113  while (1) {
114  XNextEvent (xdisplay, &xevent);
115    switch (xevent.type) {
116      case Expose :
117          redraw ();
118          break;
119
120      case MotionNotify :
121         x = xevent.xbutton.x;
122         y = xevent.xbutton.y;
123         printf("motion from %d %d to %d %d\n",x0,y0,x1,y1);
124         /* drawvec(x0,y0,x1,y1,white); /* efface ancien */
125         /* remettre état fenêtre SANS la nouvelle ligne, PUIS,
126          * tracer la nouvelle position de cette nvelle ligne */
127         XCopyArea(xdisplay,pix2,xwindow,xgc, 0,0, 450,400,0,0);
128         drawvec(x0,y0,x,y,black);   /* trace nouveau */
129         x1=x; y1=y; /* sav new end point */
130         break;
131
132      case ButtonPress :
133         x0 = xevent.xbutton.x;
134         y0 = xevent.xbutton.y;
135         x1=x0; y1=y0;
136         i= 0;
```

minixdb.c (suite)

```
137         i= traiter_clic (xevent.xkey.x, xevent.xkey.y);
138         break;
139
140      case ButtonRelease :
141         /* sauver état fenêtre incluant la nouvelle ligne */
142         XCopyArea(xdisplay,xwindow,pix2,xgc,0,0,450,400,0,0);
143         Lignes[index].x1 = x0;
144         Lignes[index].y1 = y0;
145         Lignes[index].x2 = x1;
146         Lignes[index].y2 = y1;
147         index++;
148         printf("index=%d\n",index);
149         break;
150
151    } /* fin switch */
152    if (i==-1) break;
153  } /* fin while(1) */
154
```

5 SR01 2003 - Cours Unix 5 - Notion de toolkit pour interfaces graphiques

5.1 Introduction : Objectif des toolkits

L'objectif des toolkits graphique est en général de simplifier la réalisation de programmes mettant en oeuvre une interface graphique.

Le concept clé des toolkits graphiques est la notion de widget. Le mot widget est une contraction de l'expression "window object".

Un widget est un objet graphique auquel le toolkit associe des actions lorsque l'utilisateur agit (avec la souris et/ou avec le clavier) sur le widget.

Par exemple :

- placer le curseur de la souris sur le widget et appuyer sur le bouton de gauche => évènement 1 du widget (par exemple cliquer sur le widget bouton "File" provoque l'apparition d'un menu)
- le toolkit transforme les évènements de base que l'on a déja vu (par ex. ButtonPress) en évènements "de plus haut niveau"
- ainsi le toolkit va transformet un clic qui se produit dans la surface du widget directement en un appel de la fonction prévue par ce widget pour le cas du clic
- selon la complexité du widget, celui-ci peut réagir à plusieurs sortes d'évènements en appelant chaque fois une fonction différente
- par exemple on peut vouloir considérer comme appelant des actions différentes les évènements :
  - ButtonPress
  - ButtonRelease
  - SHIFT + Clic
  - double clic
  - Control + clic

  ici SHIFT ou Control indiquent que la touche correspondante du clavier est enfoncée en même temps que se produit l'action de la souris

Une interface graphique sera donc constituée d'un assemblage de widgets, inclus les uns dans les autres (un bouton dans un menu dans une fenêtre ...) ou juxtaposés sur l'écran.

Le programme associé sera constitué d'une collection de fonctions, chacune "liée" à une des actions de l'un des widgets.

5.2 Organisation des programmes en trois couches

Au fur et à mesure de la création et de l'utilisation de systèmes graphiques divers, on s'est rendu compte qu'il était utile et plus simple de séparer les programmes en trois sous-ensembles organisés en "couches" logicielles.

Ces trois couches sont les suivantes :

- Le programme lui-même contenant le code réalisant les fonctions demandées
- Le toolkit graphique contenant :
  - des fonctions d'initialisation
  - des fonctions de gestion de widgets
  - des fonctions de gestion des évènements

  le toolkit propose des fonctions permettant au programme utilisateur de "s'abonner" à certains évènements en se faisant rappeler par le toolkit dans des fonctions de rappel ("callback") fournies par le programme et appelées par le toolkit lorsque l'évènement associé se produit
- La bibliothèque graphique de base contenant des fonctions qui encapsulent les appels aux fonctions de pilotage du matériel graphique de la machine. Ces fonctions graphiques étant très souvent complexes à utiliser, la bibliothèque a pour but d'en simplifier l'usage.

On retrouve ces trois couches dans plusieurs systèmes graphiques : X11 + Motif, GTK + GDK, Windows, Mac OS, ...

Certains systèmes graphiques cachent complètement la couche la plus basse en ne la laissant pas accessible aux programmes utilisateurs, par exemple le toolkit Tk (TCL/Tk, Python/Tk ou Perl/Tk).

5.3 Présentation d'un mini toolkit : wdg1.c + mtk.c + mix.c

Les trois programmes (wdg1.c + mtk.c + mix.c) reproduisent en miniature la structure, le fonctionnement et les fonctions essentielles d'un toolkit graphique (figure 29 page 123 et figure 30 page 123) :

1. initialisation de(s) fenêtre(s) de l'application,
2. enregistrement des fonctions de rappel,
3. attente des évènements,
4. le travail de l'application est fait dans les fonctions de rappel.

5.4 Enregistrement d'une adresse de fonction et appel de cette fonction

Les programmes ptf.c et ptft.c ci-dessous démontrent en langage C la façon de faire l'enregistrement d'une adresse de fonction dans un pointeur de fonction, puis la façon d'utiliser ce pointeur pour faire un appel à cette fonction.

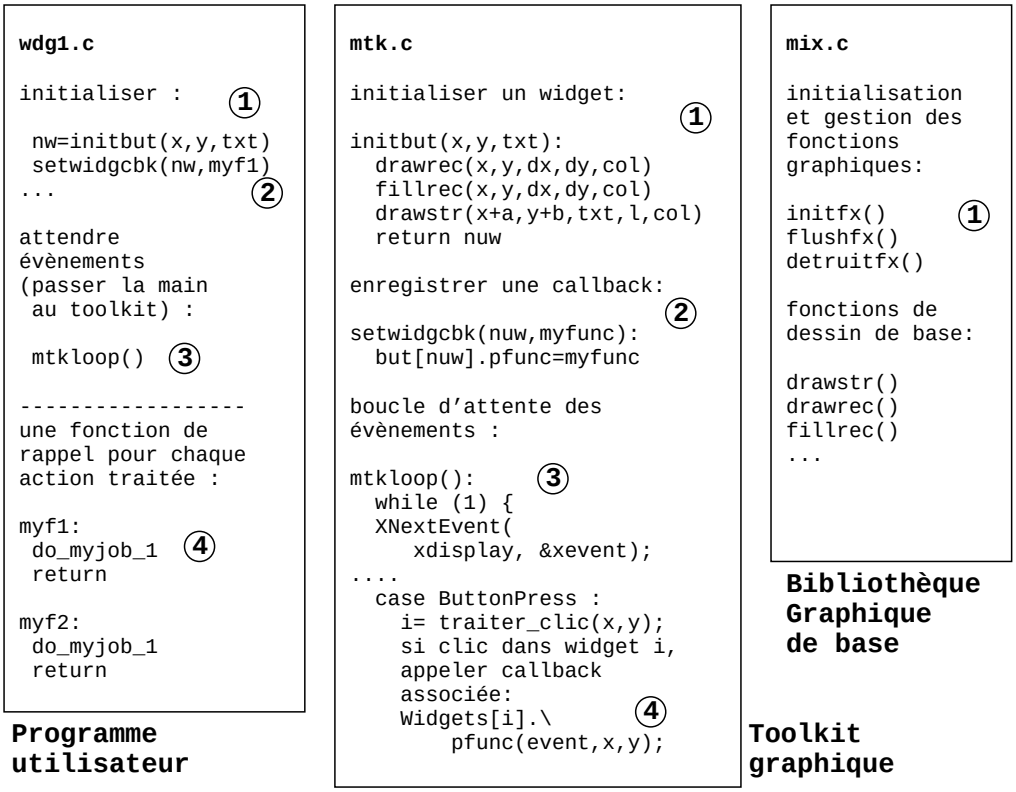


FIG. 29 – Structure d'une application graphique utilisant un toolkit

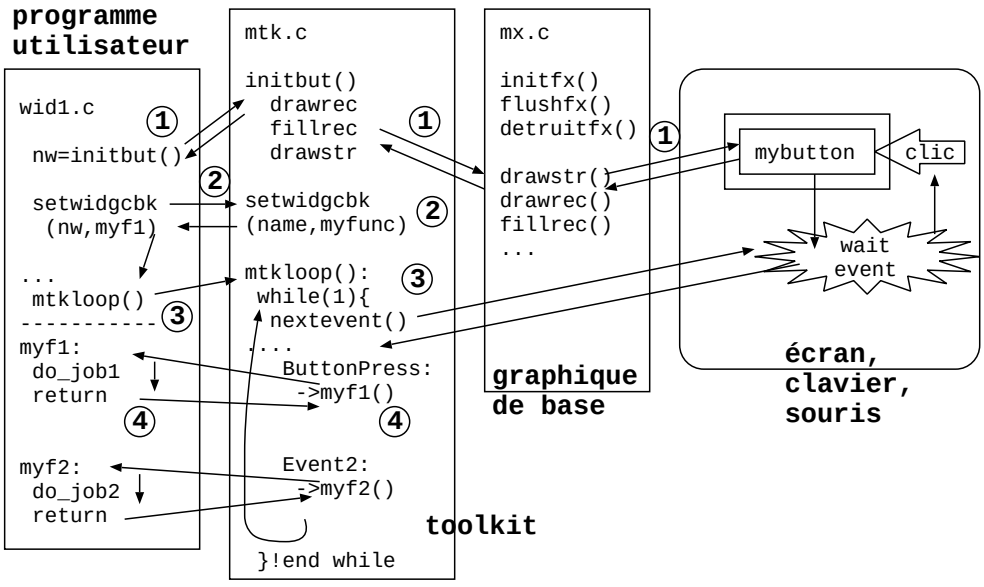


FIG. 30 – Flux de contrôle dans une application graphique utilisant un toolkit

```
/* ptf.c - exemple de pointeur de fonction
 * compil : gcc -o ptf ptf.c
*/
#include <stdio.h>/* pour printf() */

int mafonction(int a, int b)
{ return a+b;
}

main ()
{ int (*ptfonc)();
int a=4, b=5, r=0;

ptfonc = &mafonction;
printf("a=%d b=%d r=%d\n",a,b,r);
r = ptfonc(a,b);
printf("r=%d\n",r);
}
```

```
/* ptft.c - exemple de tableau de pointeurs de fonction
 * compil : gcc -o ptft ptft.c
*/
#include <stdio.h>/* pour printf() */

int fonc1(int a, int b)
{ return a+b;
}
int fonc2(int a, int b)
{ return a*b;
}

main ()
{ int (*ptftab[2])();
int a=4, b=5, r=0, i;

ptftab[0] = &fonc1;
ptftab[1] = &fonc2;
printf("a=%d b=%d r=%d\n",a,b,r);
for (i=0 ; i<2; i++) {
r = ptftab[i](a,b);
printf("r=%d\n",r);
}
}
```

5.5 Le mini toolkit expliqué : wdg1.c + mtk.c + mix.c

Maintenant on dispose de tous les éléments pour comprendre le fonctionnement du mini toolkit.

wdg1.c

```
 1  /* wdg1.c  exemple : utilise mini-toolkit mtk.c + mix.c  */
 2  /* UTC UV SR01 - (c) Michel.Vayssade@utc.fr oct 2003
 3   * compil :     gcc -c mix.c
 4   *              gcc -c mtk.c
 5   * gcc -o wdg1 wdg1.c mtk.o mix.o -L/usr/X11R6/lib -lX11
 6   */
 7
 8  #include <stdio.h>      /* pour printf() */
 9  #include <signal.h>     /* pour signal() */
10  #include <unistd.h>     /* pour pause() */
11  #include <stdlib.h>     /* pour EXIT_FAILURE EXIT_SUCCESS */
12  #include <X11/Xlib.h>   /* pour Expose ButtonPress ... */
13
14  #define red     0
15  #define green   1
16  #define blue    2
17  #define yellow  3
18  #define white   4
19  #define black   5
20
21  extern int larb,htrb;   /* définis dans mtk.c */
22
23  /*-- structure de données pour stocker lignes dessinées --*/
24  #define N_LIGNES (20)
25  int Nblignes = 0, iic=0;
26  typedef struct  /* x,y pts de départ et d'arrivée */
27  { int x1, y1, x2, y2; } Typeligne;
28  Typeligne Lignes[N_LIGNES];
29
30  /* -------------- fonctions callback ----------- */        4
31
32  void fb1(int ev, int x, int y) {  /* callback for button 1 */
33          printf("but1 clicked\n");
34          fillrec(15,155,larb,htrb,yellow);
35  }
36  void fb2(int ev, int x, int y) {  /* callback for button 2 */
37          printf("but2 clicked\n");
38          fillrec(15,155,larb,htrb,green);
39  }
40  void fb3(int ev, int x, int y) {  /* callback for button 3 */
41          printf("but3 clicked - sortir\n");
42          fillrec(15,155,larb,htrb,red); flushfx ();
```

wdg1.c (suite)

```
43          sleep(2); detruitfx();
44          exit(EXIT_SUCCESS);
45  }
46  void cvbk(int ev, int x, int y) {  /* callback for canvas */
47      printf("canvas callback ev=%d x=%d y=%d\n", ev,x,y);
48      if (ev==ButtonPress) { /* only on clic */
49          if (Nblignes>=N_LIGNES) Nblignes=0;
50          if (iic==0) {   Lignes[Nblignes].x1 = x;
51                          Lignes[Nblignes].y1 = y; iic=1;
52          } else {        Lignes[Nblignes].x2 = x;
53                          Lignes[Nblignes].y2 = y; iic=0;
54          drawvec(Lignes[Nblignes].x1,Lignes[Nblignes].y1,
55                  x,y,black);
56          Nblignes++;
57          }
58      }
59  }
60
61  /* ------- fonctions redessin après event "Expose" ------ */
62
63  void redraw() { int i;
64          drawini();
65          for (i=0; i<Nblignes; i++)
66             drawvec (    Lignes[i].x1, Lignes[i].y1,
67                          Lignes[i].x2, Lignes[i].y2, black );
68  }
69
70  /* --------------- fonctions initialisation ----------- */
71
72  int drawini() {
73       fillrec(0,0,450,400,white);   /* fen. en blanc */
74       drawstr(30,80,"_wid1.c (écrit en x=30,y=60)",28,red);
75       drawvec(30,100,200,150,black); /* vect x1,y1-x2,y2 */
76  }
77
78  void iniwid() {
79       int b1,b2,b3,cv;
80
81       b1 = initbut (100,10,"aa 11"); /* x,y,texte */          1
82       b2 = initbut (210,10,"bb 22");
83       b3 = initbut (320,10,"cc 33");
84       cv = initcanv (10,150,250,200); /* x,y,lar,hter */
85
86       setwidgcbk(b1,&fb1);                                    2
87       setwidgcbk(b2,&fb2);
88       setwidgcbk(b3,&fb3);
89       setwidgcbk(cv,&cvbk);
90  }
```

wdg1.c (suite)

```
 91  /* ---------------------- main ----------------- */
 92  main ()
 93  {
 94      int event,x,y,key , x0,y0,x1,y1,i;
 95      char c;
 96      /* signal (SIGINT, captsig); */
 97
 98      initfx(450,400);/* init fenêtre X */                    1
 99      drawini();      /* dessin initial */
100      iniwid();       /* init widgets (buttons) */            2
101      flushfx();      /* vider buffer X vers l'écran */
102
103      printf("attendre actions souris\n");
104      mtkloop ();     /* go in mtk loop, never get back */    3
105
106      exit(EXIT_SUCCESS);
107  }
```

mtk.c

```
  1  /* mtk.c  : mini toolkit : widget bouton avec callbacks */
  2  /* UTC UV SR01 - (c) Michel.Vayssade@utc.fr oct 2003
  3   * creation de widgets "boutons" + gestion clics
  4   * compil :     gcc -c mix.c
  5   *              gcc -c mtk.c
  6   * gcc -o wdg1 wdg1.c mtk.o mix.o -L/usr/X11R6/lib -lX11
  7   */
  8
  9  #include <stdio.h>      /* pour printf() */
 10  #include <stdlib.h>     /* pour EXIT_FAILURE EXIT_SUCCESS */
 11  #include <X11/Xlib.h>   /* pour Expose ButtonPress ... */
 12
 13  #define red     0
 14  #define green   1
 15  #define blue    2
 16  #define yellow  3
 17  #define white   4
 18  #define black   5
 19
 20  extern Display *xdisplay;  /* définis dans mix.c */
 21  extern XEvent xevent;
 22
 23  /*-- structure de données pour widgets --*/
 24  #define N_WIDGET (20)
 25  int Nbwid = 0;
 26  typedef struct
```

mtk.c (suite)

```
27  { int typ, x, y, l, h;
28    int (*pfunc)(); char *name; } Typewidget;
29  Typewidget Widgets[N_WIDGET];
30
31  int typb = 1;   /* widget de type button */
32  int larb = 90;  /* lar et htr des boutons */
33  int htrb = 50;
34  int typc = 2;   /* widget de type canvas */
35
36  /* ------------- fonctions du mini-toolkit ---------- */
37
38  int initbut (int x, int y, char *txt) {                    1
39      int n;
40      drawrec(x,y,larb,htrb,black); /* rect x,y,larg,hter */
41      drawstr(x+5,y+10,txt,strlen(txt),black);
42      printf("init button x,y=%d %d Nbwid=%d\n",x,y,Nbwid);
43      n= initwid(typb,x,y,larb,htrb,txt);
44      return n;
45  }
46
47  int initcanv (int x, int y, int larg, int hter) {          1
48      int n;
49      drawrec(x,y,larg,hter,black); /* rect x,y,larg,hter */
50      printf("init canv at x,y=%d %d Nbwid=%d\n",x,y,Nbwid);
51      n= initwid(typc,x,y,larg,hter," ");
52      return n;
53  }
54
55  int initwid(int type, int x, int y,
56                  int larg, int hter, char *txt) {
57          int n;
58          Widgets[Nbwid].typ = type;                         1
59          Widgets[Nbwid].x = x;
60          Widgets[Nbwid].y = y;
61          Widgets[Nbwid].l = larg;
62          Widgets[Nbwid].h = hter;
63          Widgets[Nbwid].name = txt;
64          n=Nbwid; Nbwid++;
65          return n;
66  }
67
68  void setwidgcbk (int numwid, int (*pfunc)() ) {            2
69          Widgets[numwid].pfunc = pfunc;
70  }
71
72  int traiter_clic(int x, int y) {
73          int i;
```

mtk.c (suite)

```
 74          printf("traiter_clic : clic en x,y=%d %d\n",x,y);
 75          /* clic dans quel widget? */
 76          for (i=0; i<Nbwid; i++) {
 77          if ( x>Widgets[i].x && y>Widgets[i].y &&
 78               x<(Widgets[i].x+Widgets[i].l) &&
 79               y<(Widgets[i].y+Widgets[i].h) ) return i;
 80          }
 81          return -1;
 82  }
 83
 84  void redrawid() {
 85     /* redessiner widgets - appelée si event "Expose" */
 86     int i;
 87     for (i=0; i<Nbwid; i++) {
 88       drawrec(Widgets[i].x,Widgets[i].y,
 89               Widgets[i].l,Widgets[i].h,black);
 90       drawstr(Widgets[i].x+5,Widgets[i].y+10,
 91               Widgets[i].name,strlen(Widgets[i].name),black);
 92     }
 93  }
 94
 95  /* --------------- gestion évènements ------------ */
 96
 97  mtkloop ()                      /* mtk main event loop */
 98  /* called ONCE by the app.  call callbacks on events.
 99   * never return; applic. end handled in a callback; */
100  /* liste des events dans /usr/include/X11/X.h  */
101  {
102      int event,x,y,key , x0,y0,x1,y1,i;
103      char c;
104
105      while (1) {
106      XNextEvent (xdisplay, &xevent);                    3
107        x = xevent.xbutton.x;
108        y = xevent.xbutton.y;
109        event = xevent.type;
110        switch (xevent.type) {
111           case Expose :
112             printf("expose redraw\n");
113             redraw(); /* the app MUST provide redraw func */
114             redrawid(); /* redraw registered widgets */
115             break;
116           case MotionNotify :
117             i = traiter_clic (x, y);
118             if (i>=0) {
119               if (Widgets[i].typ == typc )
120                    Widgets[i].pfunc(event,x,y);          4
121             }
```

mtk.c (suite)

```
122             break;
123           case ButtonPress :
124             i = traiter_clic (x, y);
125             if (i>=0) {
126                  printf("clic dans widget %d\n",i);
127                  Widgets[i].pfunc(event,x,y);            4
128             } else {
129                  printf("clic outside widgets\n");
130             }
131             break;
132           case ButtonRelease :
133             i = traiter_clic (x, y);
134             if (i>=0) {
135                  printf("clic dans widget %d\n",i);
136                  Widgets[i].pfunc(event,x,y);            4
137             } else {
138                  printf("clic outside widgets\n");
139             }
140             break;
141           default :
142                   printf("get event %d\n",event);
143         } /* fin switch */
144       } /* fin while(1) */
145       printf("should never go here\n");
146   } /* fin mtkloop */
147
```

5.6 Toolkit et composition d'évènements

Comment un toolkit peut composer des évènements élémentaires en évènements plus complexes auxquels il associe une action : exemple "évènement SHIFT+Press bouton 1" ou "évènement Control+Press bouton 2".

Le programme tstevent.c ci-dessous utilise la variable bshift pour conserver l'état d'enfoncement d'une touche "SHIFT". Il récupère aussi le numéro du bouton de la souris utilisé pour provoquer un évènement ButtonPress. À partir de là, il peut facilement faire un test du genre "si SHIFT enfoncé et ButtonPress avec bouton 2", alors...

tstevent.c

```
 1  /* tstevent.c   */
 2  /* un programme graphique utilisant la xlib
 3   * pour tester évènements X11 divers   */
14  #include <X11/Xlib.h>
```

tstevent.c (suite)

```
 15  #include <X11/keysym.h> /* pour évènmt keypress/keyrelease */
 16  /* dans mix.c il faut :
 17      XSelectInput(xdisplay, xwindow,
 18      ButtonPressMask | ExposureMask | ButtonMotionMask | \
 19      ButtonReleaseMask | KeyPress | KeyRelease);
 20  */
 75
 76  main ()
 77  {   int x,y, x0,y0, x1,y1, i;
 78      int bshift=0, bcntrl=0, nbut;
 79      /* data pour gestion évènements clavier KeyPress */
 80      KeySym ksym, ksym1;
 81      char *ksymname;
 89
 90      while (1) {
 91      XNextEvent (xdisplay, &xevent);
 92        switch (xevent.type) {
 93          case Expose :
 94            redraw ();
 95            break;
 96
 97          case MotionNotify :
 98            x = xevent.xbutton.x;
 99            y = xevent.xbutton.y;
100            drawvec(x0,y0,x1,y1,white); /* efface ancien */
101            drawvec(x0,y0,x,y,black);   /* trace nouveau */
102            x1=x; y1=y; /* sav new end point */
103            break;
104
105          case KeyPress :
106            i = xevent.xkey.keycode;
107            ksym = XLookupKeysym (&xevent.xkey,0);
108            ksymname = XKeysymToString (ksym);
109            if (ksym==XK_Shift_L) {
110              printf("Shift_L press\n"); bshift=1;
111            }
112            printf("-keycode =%d -keysym = %d keysymname = %s\n",\
113                       i, ksym, ksymname);
114            break;
115
116          case KeyRelease :
117            i = xevent.xkey.keycode;
118            ksym = XLookupKeysym (&xevent.xkey,0);
119            ksymname = XKeysymToString (ksym);
120            if (ksym==XK_Shift_L) {
121              printf("Shift_L release\n"); bshift=0;
122            }
123            break;
```

tstevent.c (suite)

```
124
125          case ButtonPress :
126            x0 = xevent.xbutton.x;
127            y0 = xevent.xbutton.y;
128            nbut = xevent.xbutton.button;
129            printf("press bouton nř %d bshift=%d\n",nbut,bshift);
130            x1=x0; y1=y0;
131            i= 0;
132            i= traiter_clic (xevent.xkey.x, xevent.xkey.y);
133            break;
134
135          case ButtonRelease :
136            Lignes[index].x1 = x0;
137            Lignes[index].y1 = y0;
138            Lignes[index].x2 = x1;
139            Lignes[index].y2 = y1;
140            index++;
141            printf("index=%d\n",index);
142            break;
143
144        } /* fin switch */
145        if (i==-1) break;
146      } /* fin while(1) */
147
```

5.7 Discussion sur la modification des arguments de fonctions dans une bibliothèque

Dans le troisième exercice du TD sur les interfaces graphiques, on demande d'introduire le traitement d'évènements composites tels que "ButtonPress" alors que "Shift" est enfonçé, ou ButtonPress plus l'information sur le numéro du bouton de souris utilisé.

Or la version initiale de mtk.c ne prévoyait comme retour d'information sur les évènements qu'une seule variable de type "int" contenant le numéro d'évènement (au sens X11).

Or, nous voudrions **à la fois** :

- ajouter la possibilité de renvoyer une information plus complète,
- garder la **compatibilité** de la bibliothèque avec les anciens programmes qui l'utilisent déja.

Nota : il va de soi, qui si on admet de changer à la fois la bibliothèque et tous les programmes qui l'utilisent, il n'y a pas de difficulté pour ajouter des arguments ou renvoyer des arguments plus complexes (tableaux ou structures).

Cette discussion saisi l'occasion d'un besoin de modifier l'interface initialement utilisée entre le mini toolkit mtk.c et les programmes utilisateurs, pour discuter des avantages

et inconvénients, surtout en termes de "compatibilité ascendante" de différentes méthodes d'évolution d'une bibliothèque.

Il faut donc ajouter des informations dans les paramètres renvoyés par le toolkit au programme utilisateur.

Pour cela on peut :

1. Ajouter un ou des paramètres aux fonctions de rappel ; par exemple un paramètre contenant le nř de bouton lors des évènements "ButtonPress" ou "ButtonRelease" et un paramètre contenant le nom du modificateur ("null", "shift", "control", ...).
2. Garder le même nombre de paramètre, mais compliquer la structure du paramètre concerné (ici le premier paramètre "ev").

La première méthode, bien que simple à concevoir, à un énorme inconvénient : **tous** les programmes déja écrits qui utilisent l'ancienne version de mtk.c deviennent **incompatibles** avec la nouvelle version. Non seulement il faut les recompiler, mais **pire que tout**, il faut les **modifier**, *même s'ils n'utilisent pas les nouvelles fonctionnalités* introduites. Pour le responsable du développement et de la maintenance d'une bibliothèque, cette méthode doit absolument être évitée.

La deuxième méthode est déclinable en plusieurs versions.

1. Ajouter des valeurs nouvelles aux valeurs de retour pouvant être contenues dans le paramètre. Par exemple ici, le paramètre "ev" peut avoir une valeur égale à un numéro d'évènement X11. Ces numéros sont listés dans le fichier /usr/include/X11/X.h et sont compris entre 2 et 35. Il suffit de décider que le pseudo évènement ButtonPress-B2 est égal à 36, que Shift-ButtonPress est égal à 37, etc ...

   Parfait ?

   Hélas non ! Si on fait ainsi, les anciens programmes fonctionneront correctement avec la nouvelle bibliothèque, **mais** avec un petit effet de bord : avec l'ancienne version un clic bouton-souris-1 et un clic bouton-souris-2 avaient le même effet ; avec la nouvelle version le clic bouton-souris-2 n'aura plus d'effet car la nouvelle valeur de "ev" fera échouer le test "if (ev==ButtonPress)".
2. Transformer le premier argument en une structure ou un tableau : on continue, comme avant, à mettre dans le premier champ de la structure ou du tableau le numéro d'évènement X11, et on ajoute dans les mots mémoire suivants les informations supplémentaires désirées :

   ```
   struct ev {
      int ev.event;  /* nř d'évènement X11 */
      int ev.button; /* nř du bouton souris utilisé */
      int ev.modif;  /* modificateur (shift, control, ...) */
   }
   ```

   ceci semble possible (et facile) car nous sommes dans le cas très particulier des fonctions de rappel. C'est la bibliothèque qui appelle la fonction de l'utilisateur. Elle pourrait appeler cette fonction avec un argument qui soit parfois une valeur et parfois une adresse, **à condition** que la fonction dispose d'un moyen pour décider si l'argument effectif lors de l'appel est une valeur simple ou une adresse (sinon le plantage est assuré).

   C'est cette solution qui sera montrée un peu plus loin. Toutefois cette méthode viole un certain nombre de principes de "bonne programmation"

Dans le cas où on souhaite modifier les arguments d'une fonction de la bibliothèque **appelée** par les programmes, transformer un argument scalaire, même déja passé par adresse en un tableau ou une structure n'est pas viable car si la fonction modifie ce tableau, dans les programmes anciens qui ne passent qu'un scalaire, la fonction appelée va écraser des mots mémoire "quelque part".

Dans ce cas, on ne peut s'en sortir que de deux façons :

1. Avoir prévu à l'avance une possibilité d'extension compatible, par exemple en prévoyant, en plus du paramètre "adresse", un paramètre donnant la **longueur** du premier paramètre, afin que le toolkit évite d'écrire dans une zone mémoire non allouée, et ne remplisse que ce qui a été demandé. Ainsi les anciens programmes fonctionnent sans être modifiés et les nouveaux peuvent demander plus d'informations.
2. Créer une nouvelle fonction, avec une nouvelle liste d'arguments, en plus de la fonction existante, en conservant celle-ci. Les anciens programmes fonctionnent sans être modifiés en appelant l'ancienne fonction, les nouveaux programmes utilisent la nouvelle fonction pour bénéficier des nouvelles possibilités. Cette méthode à l'inconvénient de multiplier les fonctions qui font "presque" la même chose, mais comme le montre l'examen des appels systèmes dans un système d'exploitation qui est utilisé et évolue sur le très long terme, elle est fréquemment choisie car la plus simple et la plus sure.

Nous allons maintenant présenter la méthode évoquée plus haut qui renvoie vers la fonction de rappel tantôt un argument scalaire, tantôt un argument structure. Rappelons d'abord que ce genre de programmation doit être évité chaque fois que s'est possible. Il ne sera utilisé qu'en dernière extrémité. Mais il a l'intérêt de montrer un exemple de manipulation des types en langage C.

On va jouer sur le fait que le langage C ne vérifie pas les types des arguments effectifs passés aux fonctions : on renvoie comme argument "ev", soit un int contenant effectivement le numéro de l'évènement (ancienne version), soit un **pointeur** sur une structure telle que définie ci-dessus. Cette structure est instanciée de façon statique dans le toolkit. Dans la fonction callback, on joue sur le fait qu'une adresse ne peut jamais avoir une valeur aussi faible que celle des int représentant les évènements pour décider si la fonction est appelée avec un int en argument ou bien avec un pointeur sur une structure. Ensuite pour chaque cas on fait un transtypage (cast) de l'argument vers le type adéquat. Bien sûr, si les int passés en arguments scalaire pouvaient prendre n'importe quelle valeur, cette méthode ne serait pas utilisable.

wdg3.c

```
29
30  typedef struct {
31          int ev,b,m;} eventstruct;
32
35
36  /* ----------------- fonctions callback ------------- */
37
38  void fb1(void *ev, int x, int y) {  /* callback for button 1 */
```

wdg3.c (suite)

```
39      /* void accept any type of call */
40      eventstruct *p;
41      int e;
42      e = (int)ev; /* cast ev to int */
43      printf("but1 clicked e=%d\n",e);
44      if (e<=LASTEvent) { /* simple, "old" callback method */
45        fillrec(15,155,larb,htrb,yellow);
46        redrawcerc(white); xcerc= xcerc - 10; redrawcerc(blue);
47      } else { /* test new method : ev is a pointer */
48        p = (eventstruct *)ev; /* force cast to pointer */
49        printf("composed event = %d %d %d\n",p->ev,p->b,p->m);
50      }
51  }
52  void fb2(int ev, int x, int y) {  /* callback for button 2 */
53        printf("but2 clicked\n");
54        fillrec(15,155,larb,htrb,green);
55        redrawcerc(white); xcerc= xcerc + 10; redrawcerc(blue);
56     if (ev==ButtonPress|ev==ButtonRelease) { /* only on clic B1 */
57        printf("but2 event e=%d\n",ev);
58     } else {
59        printf("but2 unknown event e=%d\n",ev);
60     }
61  }
```

mtk2.c

```
34
35  typedef struct {
36          int ev,b,m;} eventstruct;
37  static eventstruct evstr;
38
124           case ButtonPress :
125              i = traiter_clic (x, y);
126              nbut = xevent.xbutton.button;
127              if (i>=0) {
128                  printf("clic dans widget %d nbut=%d\n",i,nbut);
129                  if (nbut==1) {
130                  Widgets[i].pfunc(event,x,y);}
131                  else {
132                  evstr.ev = event;
133                  evstr.b = nbut;
134                  evstr.m = 12345;
135                  Widgets[i].pfunc(&evstr,x,y);
136                  }
137              } else {
138                  printf("clic outside widgets\n");
```

mtk2.c (suite)

```
139             }
140             break;
```

5.8

SR01 2003 - Cours Unix 6 - Fonctions de base d'un système d'exploitation

6 SR01 2003 - Cours Unix 6 - Fonctions de base d'un système d'exploitation

6.1 Introduction

Cet exposé montre les fonctions élémentaires prises en compte par un système d'exploitation pour permettre de rendre utilisable cette machine très particulière et très complexe que constitue un ordinateur.

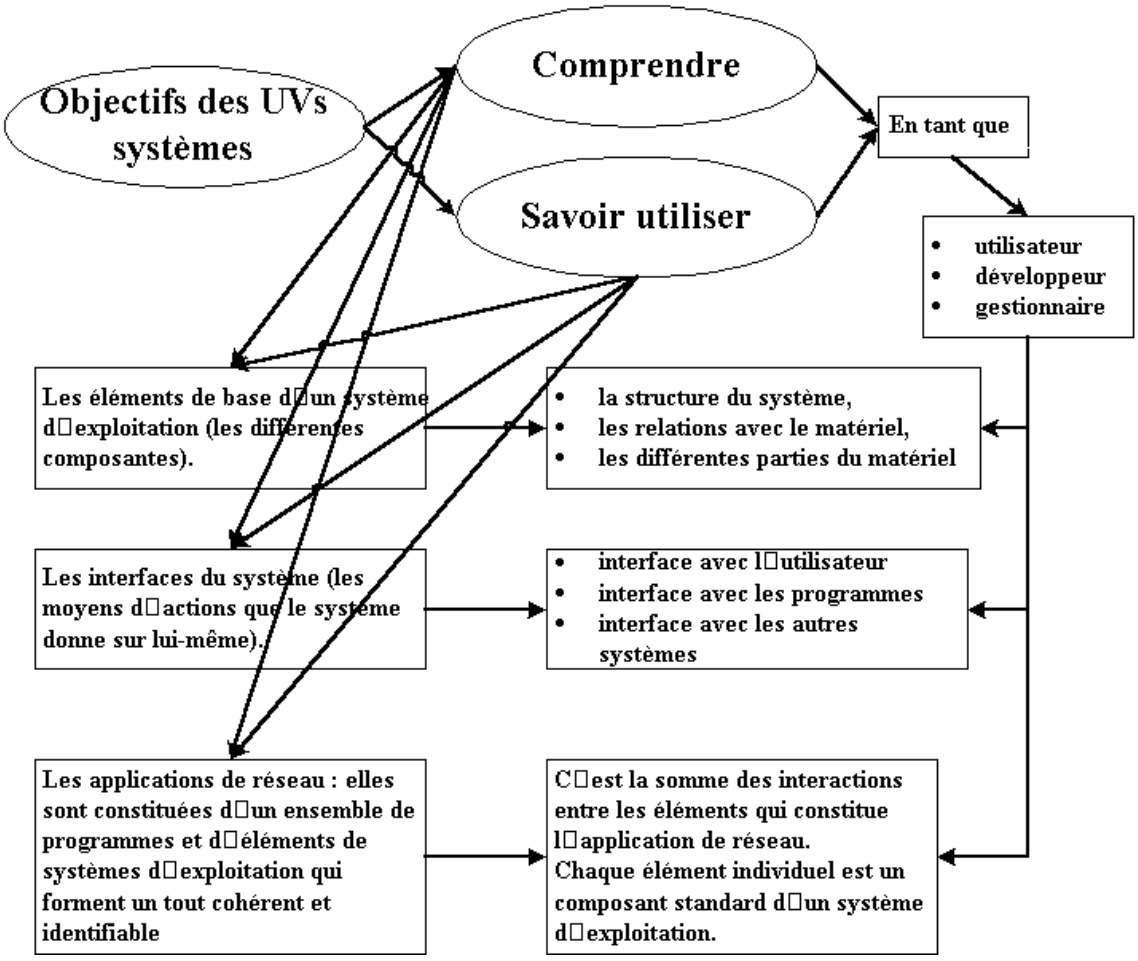


FIG. 31 – Objectifs des UVs systèmes

Savoirs faire :

- se connecter, utiliser l□interface graphique
- utiliser des logiciels :
  - bureautique : traitements de textes, tableurs, dessin, image, ...
  - scientifiques : CAO, Calcul, Maths, ...
  - réseau :
    - connexion à distance / navigation internet
    - échanges de données (ftp, mail, news, ...)
    - connexion en réseau postes clients sur serveur
- développer des programmes simples :
  - éditeur de textes / compilation / édition de liens
  - débogueur / profileur / archiveur de bibliothèques
- développer des programmes systèmes
- développer des applications de réseau
  - web, cgi, javascript, java, Corba, ...
- modifier des éléments du système
-

FIG. 32 – Savoir-faire visés par les UVs systèmes

Qu'est-ce qu'un système d'exploitation ?

Besoin

Le double rôle

Complexité => structure

FIG. 33 – Qu'est-ce qu'un système d'exploitation ?

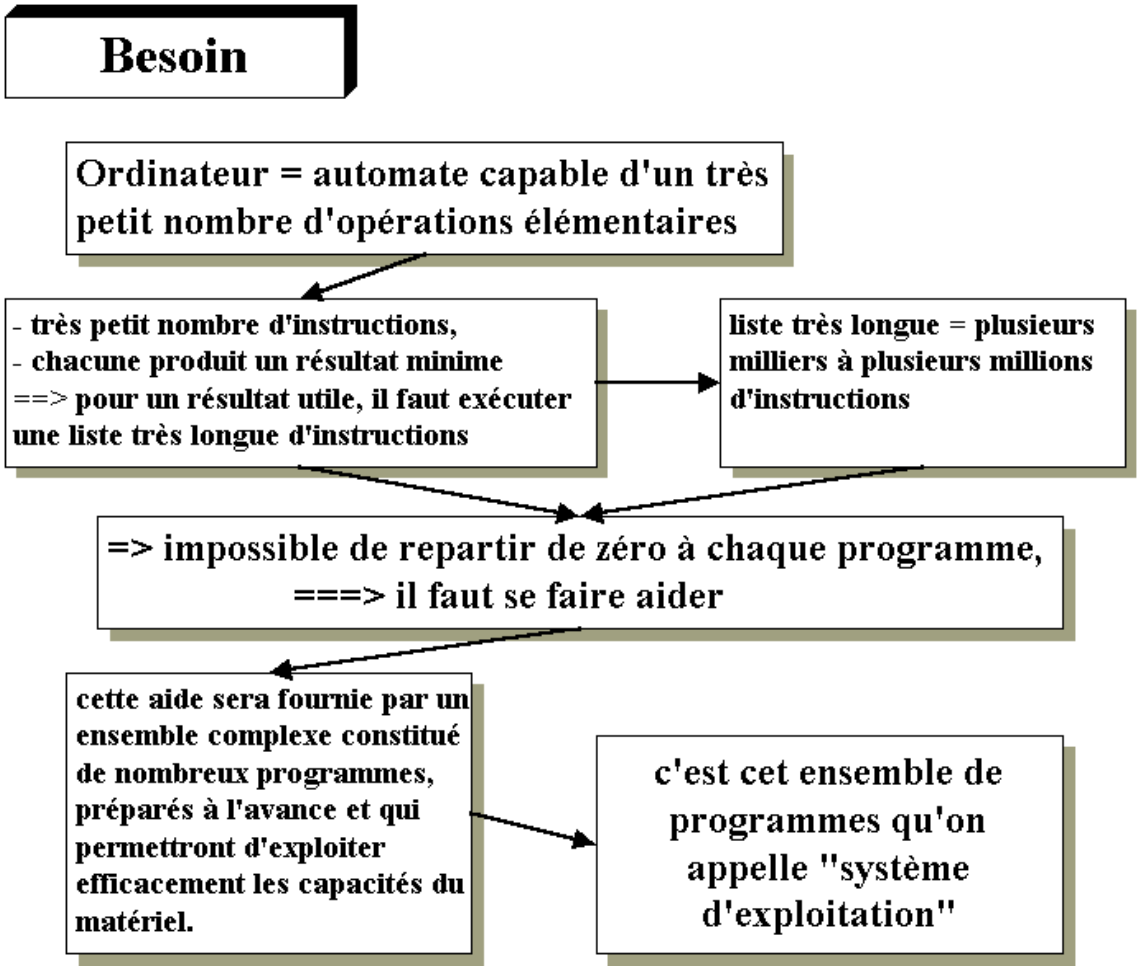


FIG. 34 – Le besoin d'un système d'exploitation

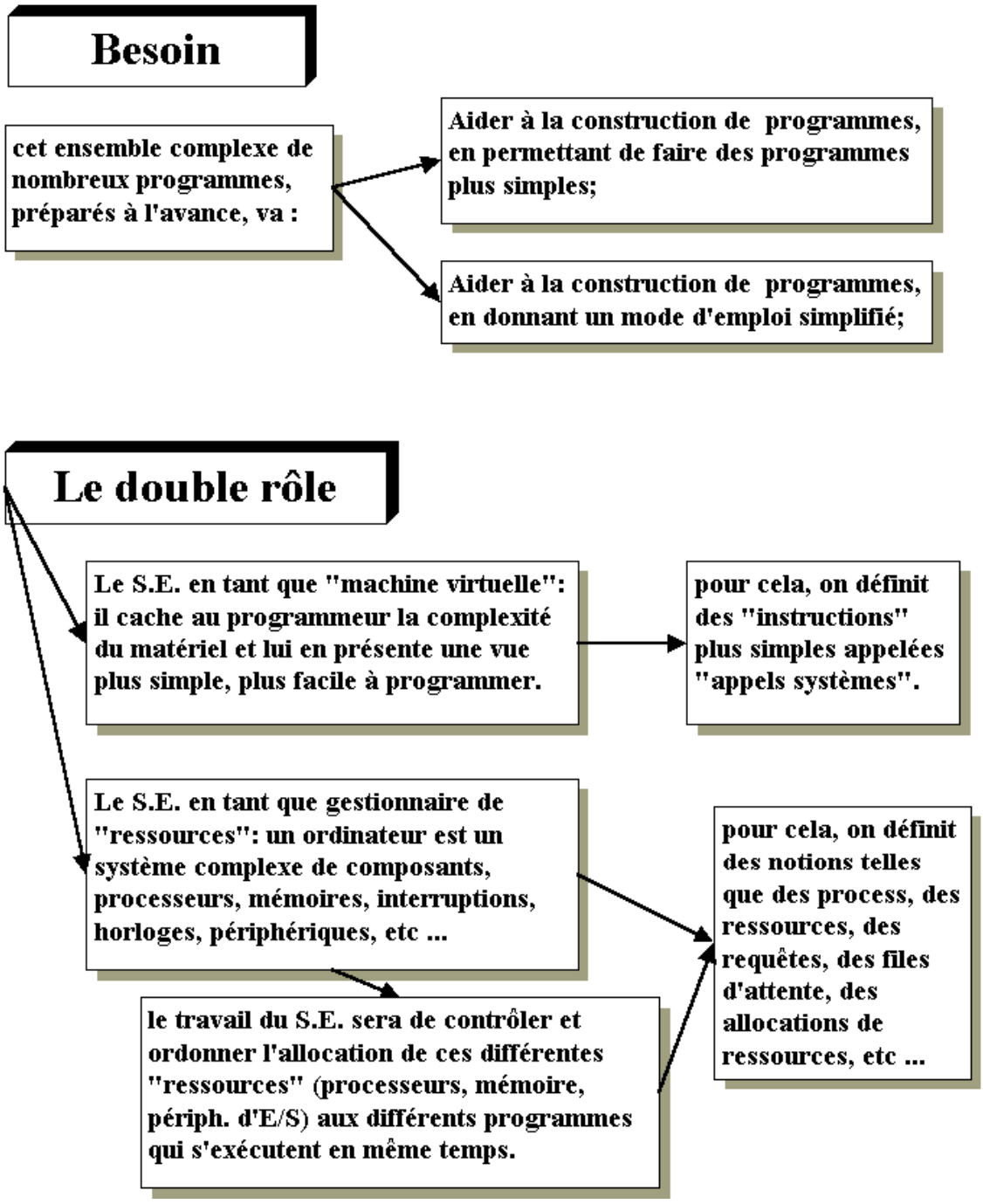


FIG. 35 – Le besoin et le double rôle d'un système d'exploitation

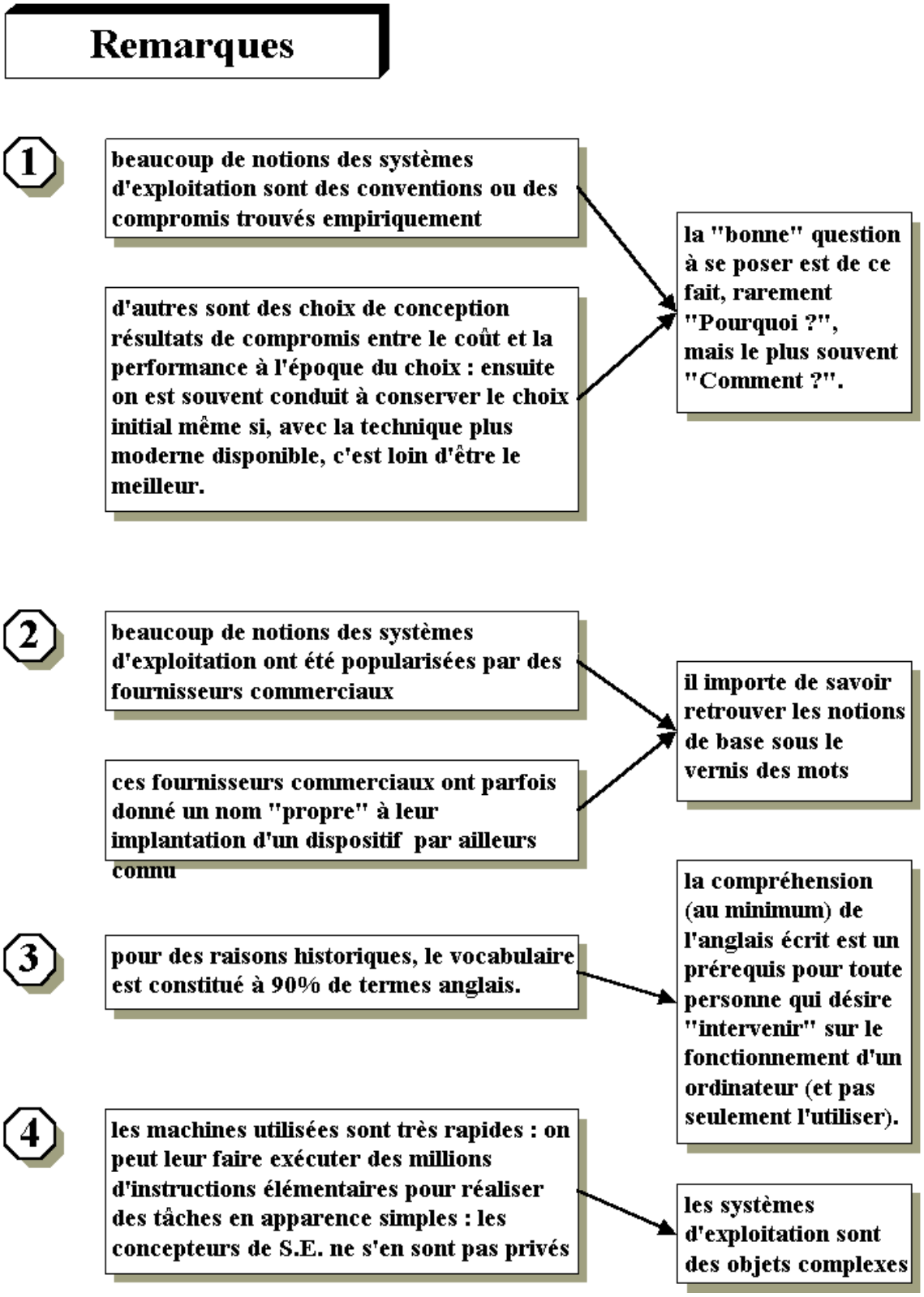


FIG. 36 – Remarque sur la compréhension des systèmes d'exploitation

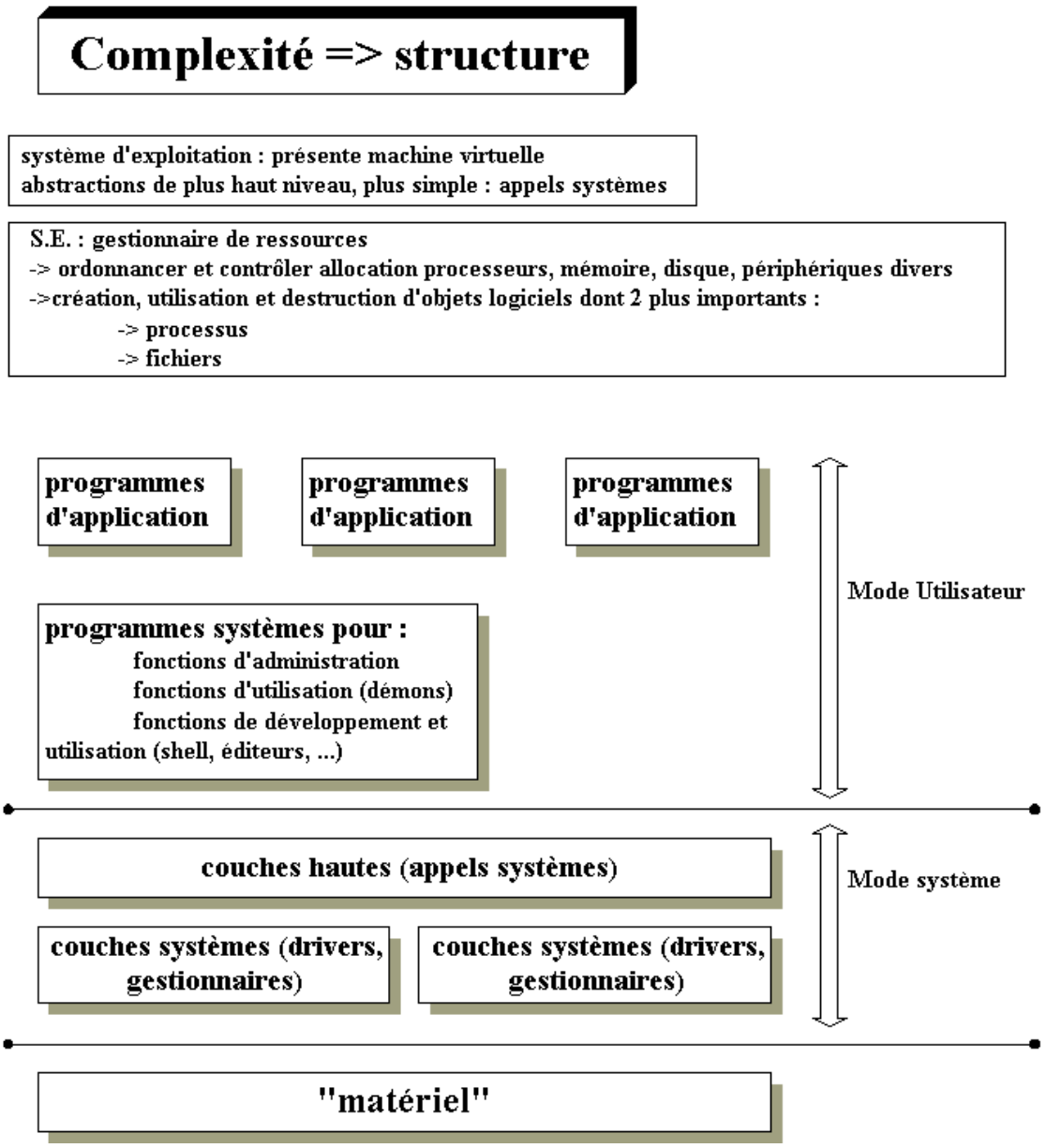


FIG. 37 – Complexité et structure dans les systèmes d'exploitation

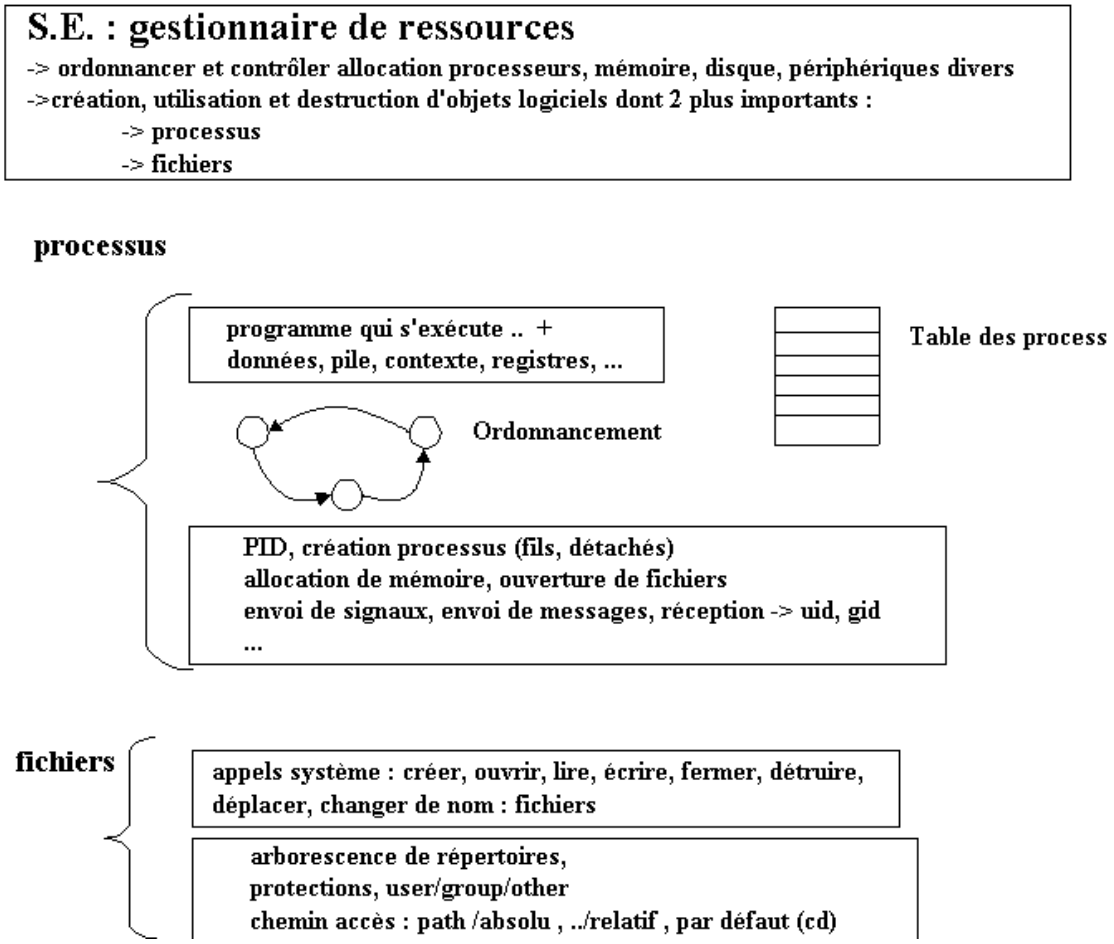


FIG. 38 – Le système d'exploitation, gestionnaire de ressources

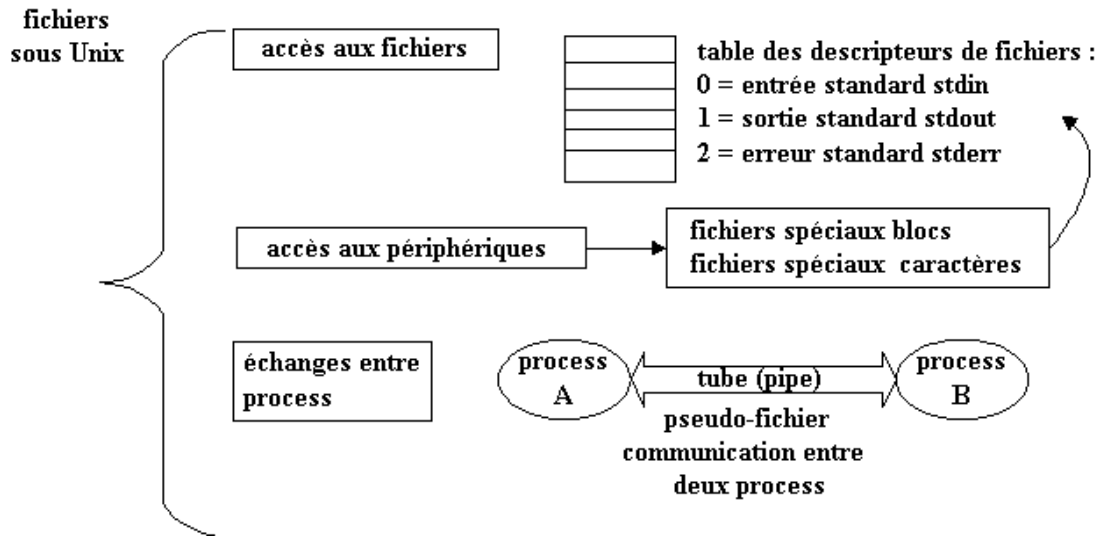


FIG. 39 – Les fichiers, concept de simplification sous unix

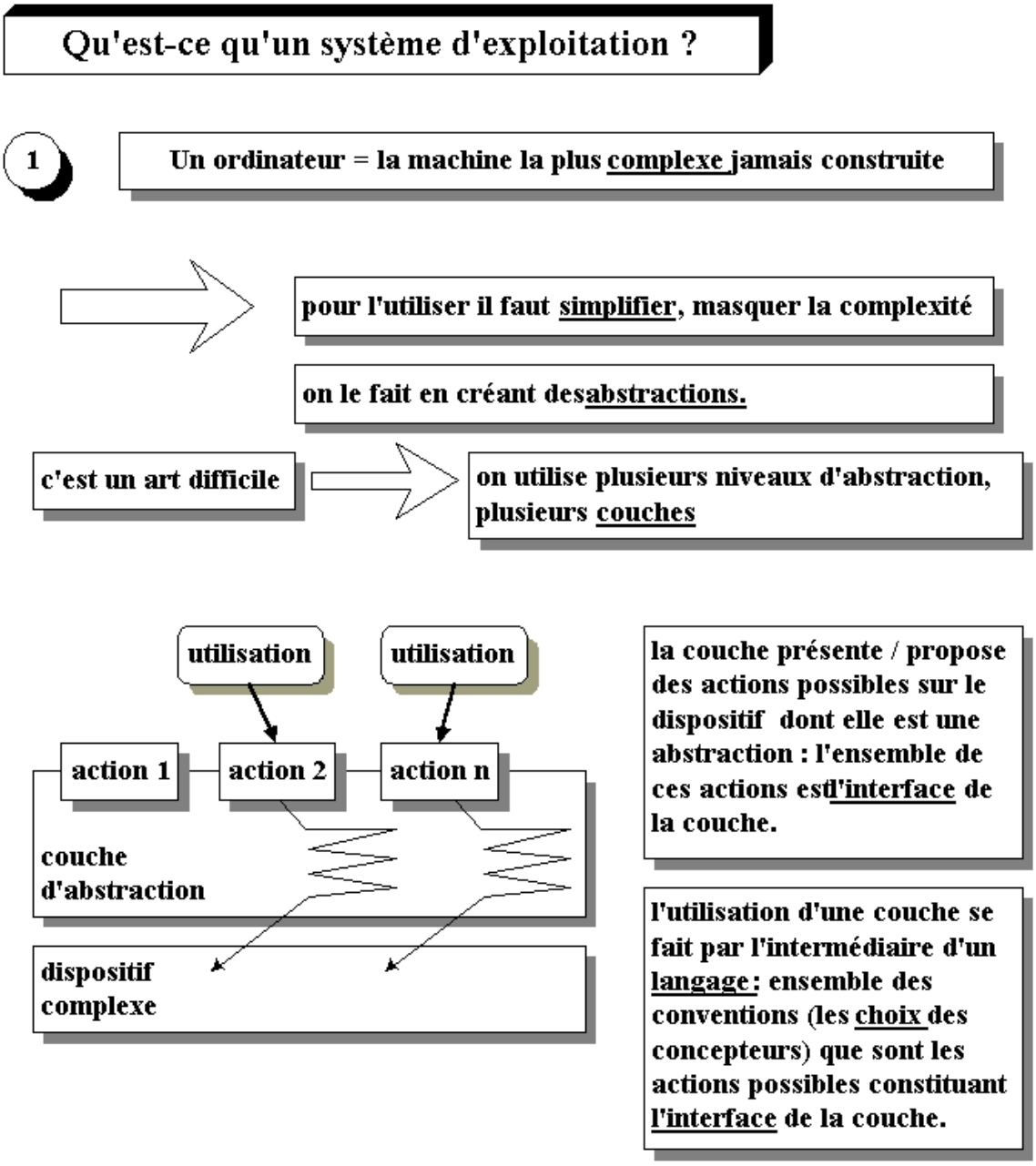


FIG. 40 – Qu'est-ce qu'un système d'exploitation ? (1/7)

Qu'est-ce qu'un système d'exploitation ?

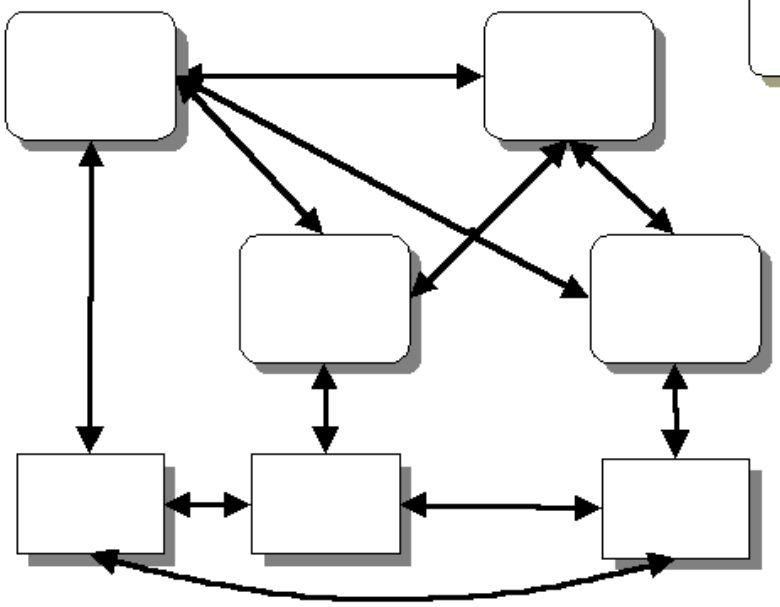
2

Un ordinateur, cette machine "la plus complexe", est composée de plusieurs grands sous-ensembles.

Chaque sous-ensemble est géré par une couche logicielle qui en propose une abstraction.

Mais, chaque sous-ensemble a besoin des autres dans son fonctionnement. Pour faire appel aux autres sous-ensembles, il utilise leur couche d'abstraction.

Il y a donc, entre sous-ensembles, des interactions nombreuses et entrecroisées.



Un système d'exploitation n'est pas un programme linéaire qui commence "au début" et se poursuit jusqu'à "la fin".
C'est un ensemble de programmes ayant chacun un rôle et interagissant les uns sur les autres par échange continuel de messages, d'appels, de signaux, auxquels ils sont conçus pour réagir.

FIG. 41 – Qu'est-ce qu'un système d'exploitation ? (2/7)

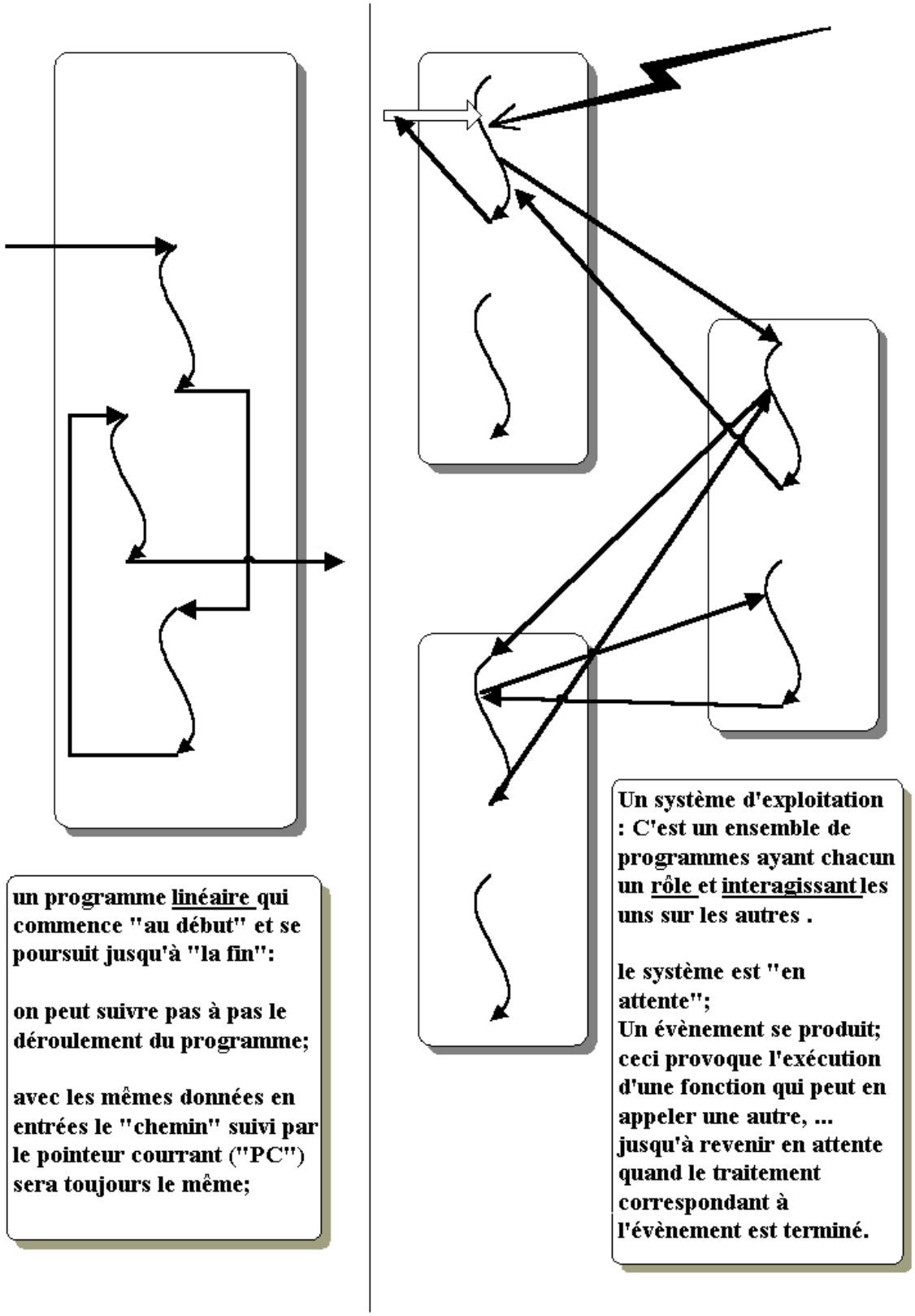


FIG. 42 – Qu'est-ce qu'un système d'exploitation ? (3/7)

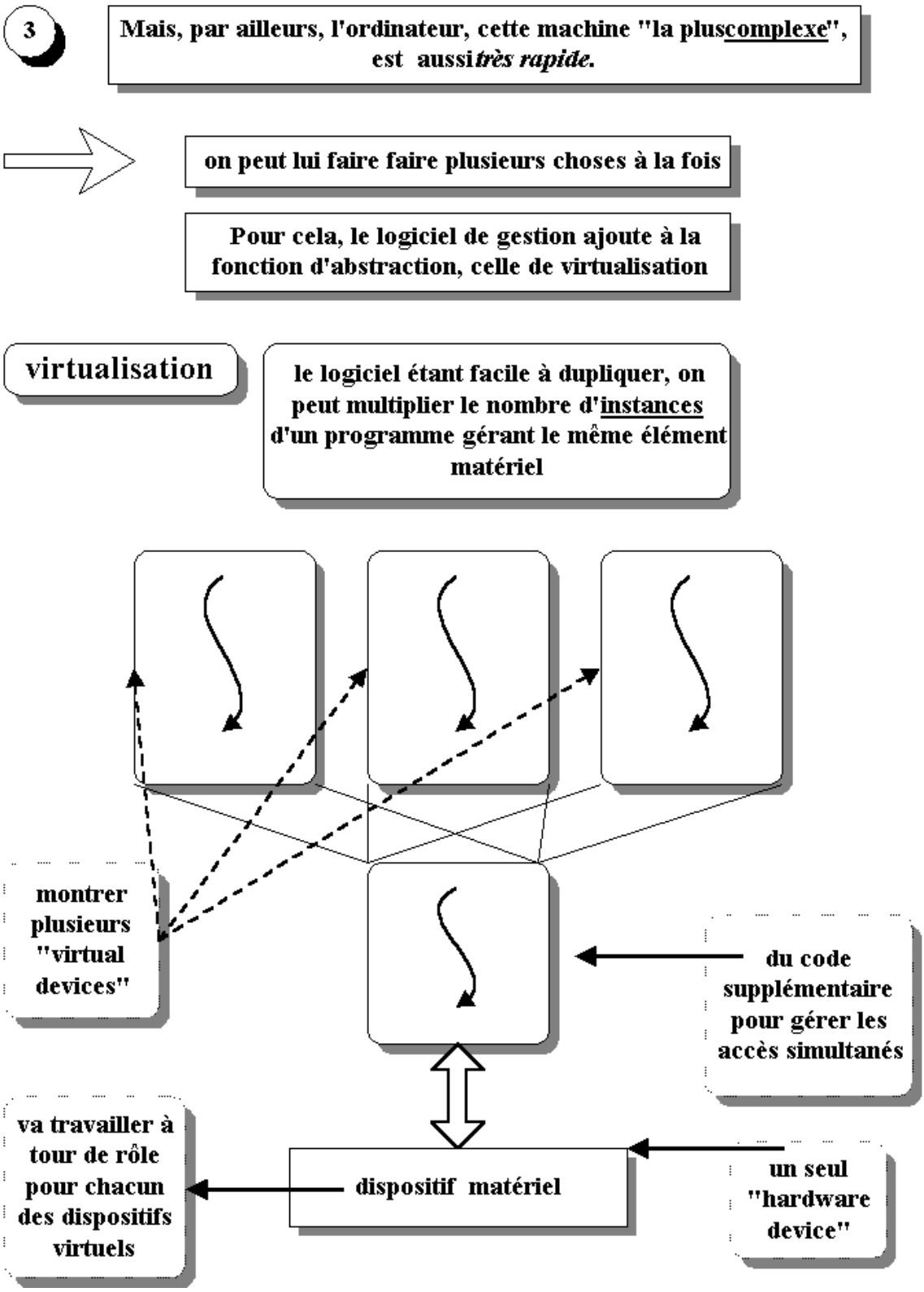


FIG. 43 – Qu'est-ce qu'un système d'exploitation ? (4/7)

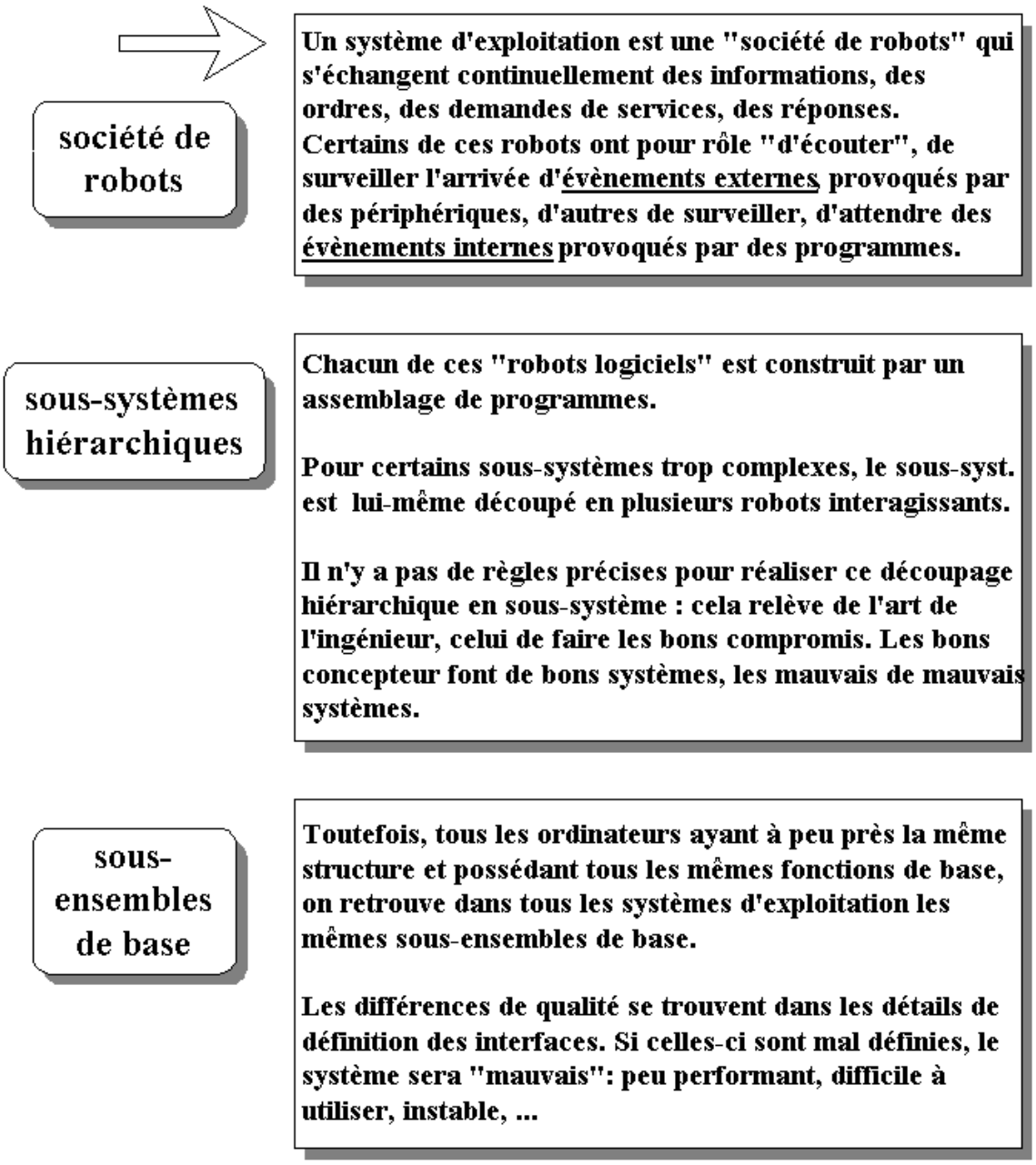
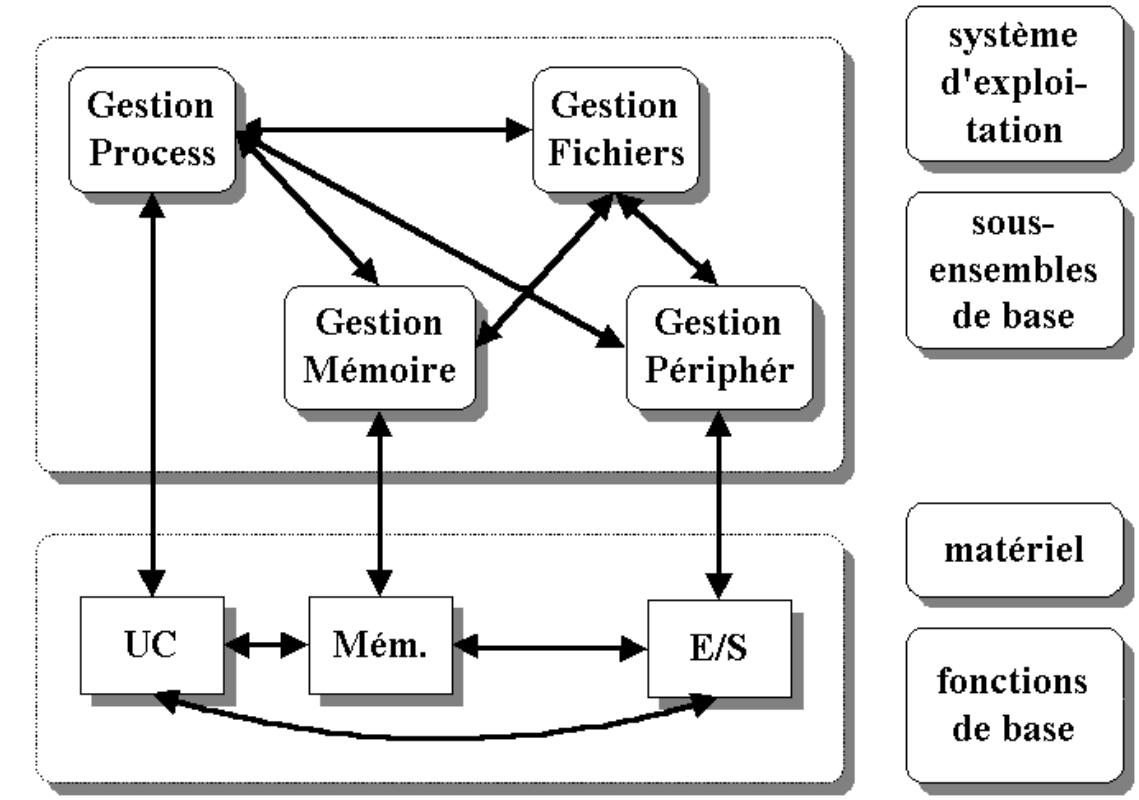


FIG. 44 – Qu'est-ce qu'un système d'exploitation ? (5/7)



Les 3 éléments de base d'un ordinateur sont l'unité centrale, la mémoire et les périphériques d'entrées-sorties.

Les 3 éléments de base d'un système d'exploitation sont la gestion des process, la gestion de la mémoire et la gestion des périphériques d'entrées-sorties.

En raison de son importance dans le fonctionnement du système (le système lui-même y est stocké) la gestion logique des disques sous forme d'un système de fichiers est traité comme un sous-système indépendant par presque tous les systèmes d'exploitation.

FIG. 45 – Qu'est-ce qu'un système d'exploitation ? (6/7)

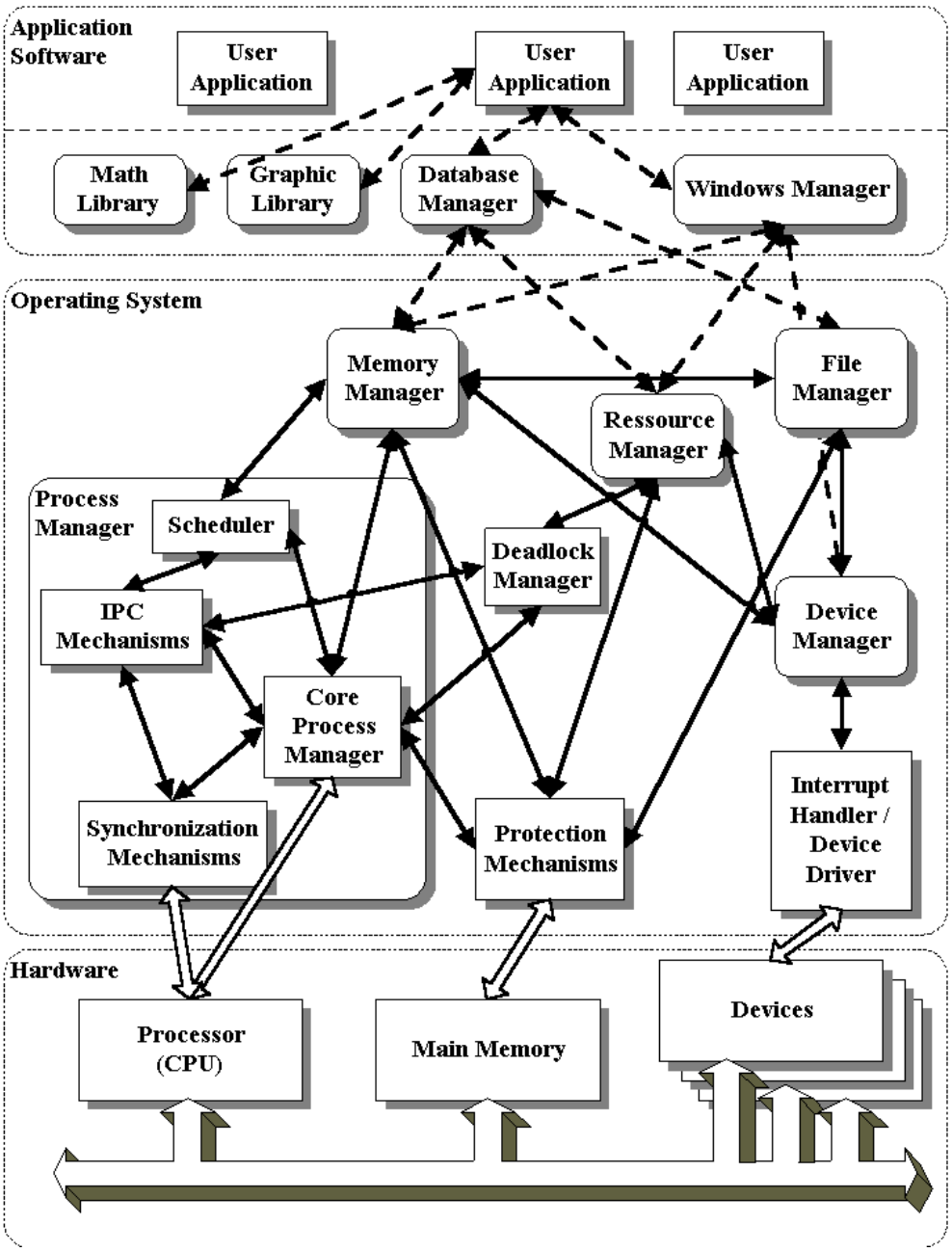


FIG. 46 – Qu'est-ce qu'un système d'exploitation ? (7/7)

SR01 2003 - Cours Unix 7 - Vue externe d'un système d'exploitation

7 SR01 2003 - Cours Unix 7 - Vue externe d'un système d'exploitation (unix)

7.1 Introduction

Les fonctions de base de l'utilisation d'unix concernent les actions élémentaires que tout utilisateur doit connaitre.

- connexion, login, arbre des process,
- jobs avant plan, arrière plan,
- netscape, site de l'UV,
- créer répertoire, nedit, xterm,
- commandes unix de base ;

7.2 Connexion au système

Connexion : Lors de la connexion sur un système unix, le système crée un process pour valider la connexion (login, password). Si cette connexion est valide, ce process crée un sous-process qui va exécuter le shell par défaut de l'utilisateur. Avant de donner la main à l'utilisateur, le shell va exécuter successivement plusieurs fichiers de commande. Les fichiers exécutés dépendent du shell qui est utilisé. Pour le shell **tcsh**, ce sont :

- /etc/csh.login (exécuté pour les "login-shell" seulement)
- /etc/csh.cshrc (exécuté par tous les shells)
- ~/.login (exécuté pour les "login-shell" seulement)
- ~/.cshrc (exécuté par tous les shells)
- **attention :**pour un shell **tcsh** ~/.tcshrc est cherché d'abord, et seulement s'il n'existe pas, le tcsh exécute ~/.cshrc

Pour les shells de type **bash** ce sont /etc/bashrc, /etc/profile, ~/.profile, ~/.bashrc.

Dans ces fichiers de commande, on va placer des définitions d'alias de commandes, de variables du shell et de variables d'environnement. Ainsi ces définitions seront valides pour tous les programmes exécutés par le même utilisateur.

Le listing ci-dessous montre les process créés par la connexion de l'utilisateur "michel" sur la machine. On y voit le process de login, puis le process bash qui est son fils, et le process créé par l'exécution dans ce shell de la commande "more".

```
[dupond@host work]$ ps -aux | grep login
root       992  0.0  1.3  2288 1248 tty1   S 09:54  0:00 login -- dupond
root      1357  0.0  1.2  2284 1212 pts/4  S 10:08  0:00 login -- michel
dupond    1920  0.0  0.6  1596  604 pts/2  S 19:09  0:00 grep login
[dupond@host work]$ ps -lu michel
```

```
  F S   UID   PID  PPID  C PRI  NI ADDR  SZ WCHAN  TTY        TIME CMD
100 S   100  1358  1357  0  69   0  -   582 wait4  pts/4  00:00:00 bash
000 S   100  1916  1358  0  69   0  -   387 read_c pts/4  00:00:00 more
```

Notion de compte : le login (ou username) est créé par l'administrateur du système. Il lui associe un **UID** (User ID) unique, ainsi qu'un **GID**. Ces deux nombres serviront à définir les protections des fichiers de l'utilisateur.

Mot de passe : l'administrateur du système affecte à chaque utilisateur un **mot de passe** ("password"). L'utilisateur peut normalement changer ce mot de passe. Un "login/password" est en général soumis à une **charte d'utilisation** qui définit le droit d'usage, les restrictions éventuelles et les précautions à prendre. Enfreindre ces règles revient à se mettre en faute vis-à-vis du règlement intérieur. Dans une entreprise, ceci peut être considéré comme un motif "légitime et sérieux" de licenciement pour faute.

Déconnexion : il faudra prendre soin de fermer toutes les fenêtres ouvertes lors d'une session interactive, faute de quoi, quelqu'un d'autre aurait accès à votre compte, et **vous** en seriez responsable, par négligence.

7.3 Avant plan et arrière plan ("background and forground")

Notion d'arbre de process : les process créés à partir d'un même process, ainsi que leurs fils et les fils de leurs fils, etc... constituent un arbre de process. On peut reconstituer cet arbre avec les informations "PID" et "PPID" données par la commande "**ps**". Si un process disparait (se termine), ses process fils "détachés" continuent leur vie dans le système, et sont pour cela **adoptés** par le process **init** de PID=1, père de tous les process (le process init est créé lors du lancement ("bootstrap") du système).

Les process détachés sont créés à la ligne de commande par la notion de jobs d'arrière-plan ("background").

Par exemple : la sortie ci-dessous montre quatre process "xterm" créés par un process qui s'est terminé (le process de lancement de l'interface graphique). À chacun de ces "xterm" est associé un process fils qui exécute le shell par défaut de l'utilisateur (user d'UID 500), ici "bash".

```
[dupond@host work]$ ps -lu dupond | grep "xterm\|bash"
100 S   500   999   991  0  69   0 -   597 wait4  tty1   00:00:00 bash
000 S   500  1091     1  0  69   0 -  1367 do_sel ?  00:00:00 xterm
000 S   500  1093     1  0  70   0 -  1367 do_sel ?  00:00:00 xterm
000 S   500  1095     1  0  69   0 -  1367 do_sel ?  00:00:00 xterm
000 S   500  1097     1  0  69   0 -  1367 do_sel ?  00:00:00 xterm
000 S   500  1103  1091  0  69   0 -   616 wait4  pts/2  00:00:00 bash
000 S   500  1104  1093  0  72   0 -   619 wait4  pts/0  00:00:00 bash
000 S   500  1106  1095  0  69   0 -   618 wait4  pts/3  00:00:00 bash
000 S   500  1107  1097  0  69   0 -   616 read_c pts/1  00:00:00 bash
[dupond@host work]$ ps -l
  F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY      TIME   CMD
000 S   500  1104  1093  0  69   0 -   619 wait4  pts/0  00:00:00 bash
000 S   500  1200  1104  0  69   0 -  1461 do_sel pts/0  00:00:11 nedit
000 R   500  1522  1104  0  75   0 -   749 -  pts/0  00:00:00 ps
```

Les trois dernières lignes ci-dessus, montrent l'arbre du process "1104". Il contient deux process fils : "1200" résultat de la commande "nedit &" et "1522" résultat de la commande "ps -l". Si on termine le process "1104", par la commande "exit", le process "1200" continuera (l'éditeur n'est pas détruit (!)) et sera adopté par "init", comme on peut le voir ci-dessous.

```
*** AVANT terminaison de "1200"
[dupond@host work]$ ps -lu dupond | grep nedit
000 S   500  1200  1104  0  69   0 -  1461 do_sel pts/0  00:00:11 nedit

*** ici terminaison de "1200"
[dupond@host work]$ ps -lu dupond | grep nedit
000 S   500  1200     1  0  69   0 -  1461 do_sel ?      00:00:11 nedit
```

Lancement des jobs en arrière-plan et rappel en avant-plan :

```
> prog1 &  # en background
[1]  1820       <-- donne le numéro du job lancé en arrière
                    plan, ainsi que le PID du processus
                    correspondant ( 1820 ) : on dit
                    que ce processus tourne en "background"

> prog2
ctrl-Z
[2]+  Stopped <-- prog2 reçoit le signal "Stop" (envoyé par ctrl-Z)
    le process exécutant prog2 passe dans l'état Stopped
    et il est ajouté à la liste des jobs
> jobs
[1]   Running    prog1 &
[2]+  Stopped  prog2
> bg   (ou bg %2) <-- passer prog2 en arrière plan
    prog2 passe de Stopped à Running
> fg %n <-- rappeler le job %n en avant plan
```

7.4 Fonctions de base du shell

Apprentissage de l'utilisation de base du shell unix "tcsh "

- **édition de la ligne de commande :** flèches du clavier, [ctrl-D] (choix possibles de fichiers ou commandes), [TAB] compléter le nom du fichier si non ambigu, [ctrl-E], [ctrl-A], etc ...
-

Variables :

- **alias**
- **variables du shell** référencées par $nomvar
- **variables d'environnement** idem, accessibles depuis un programme par l'appel système **getenv**

- **voir liste des variables :** echo $[ctrl-D] en tcsh (en bash faire echo $[TAB][TAB]
-
- variables d'environnement sont héritées par les process fils
- voir les variables d'environnement : **printenv**
- variables d'environnement importantes : **DISPLAY** et **PATH**

variables d'environnement (bash linux) :

```
[dupond@host work]$ echo $[tab][tab]
$_              $HOME       $MAIL        $RANDOM
$BASH           $HOSTNAME     $MAILCHECK       $SECONDS
$BASH_ENV       $HOSTTYPE     $OLDPWD        $SESSION_MANAGER
$BASH_VERSINFO  $i       $OPTERR        $SHELL
$BASH_VERSION   $IFS       $OPTIND        $SHELLOPTS
$COLORS         $INPUTRC      $OSTYPE        $SHLVL
$COLUMNS        $KDEDIR       $PATH        $sourced
$DIRSTACK       $LANG       $PIPESTATUS      $SSH_ASKPASS
$DISPLAY        $langfile     $PPID        $SUPPORTED
$EUID           $LESSOPEN     $PROMPT_COMMAND  $TERM
$GROUPS         $LINENO       $PS1         $UID
$HISTCMD        $LINES        $PS2         $USER
$HISTFILE       $LOGNAME      $PS4         $WINDOWID
$HISTFILESIZE   $LS_COLORS    $PWD         $XMODIFIERS
$HISTSIZE       $MACHTYPE     $QTDIR
```

alias (bash linux) :

```
[dupond@host work]$ echo $[return]
alias acro='/usr/local/Acrobat4/bin/acroread'
alias l.='ls -d .[a-zA-Z]* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
alias which='alias | /usr/bin/which --tty-only \
--read-alias --show-dot --show-tilde'

[dupond@host work]$  which ls
alias ls='ls --color=tty'
        /bin/ls
```

7.5 Le site de l'UV

Site web de l'UV Toutes les UVs de l'UTC ont un site web dédié à l'enseignement. Celui de SR01 est **http ://www4.utc.fr/~sr01**.

Sur le site de l'UV on trouvera :

- des informations sur le déroulement de l'UV (par exemple date des examens)
- le programme (sujets) des TDs
- des documents (manuels)

Les inscrits à l'UV sont **tenus** de consulter le site régulièrement.

Lancement de Netscape : "netscape" est un programme disponible sur le serveur unix **sunserv**. On va le lançer par la commande **"> netscape &"**, après avoir positionné correctement la variable d'environnement **DISPLAY**.

DISPLAY : si vous êtes connectés depuis le terminal X "termx005.utc" (le nom du terminal est écrit sur une étiquette collée sur le boîtier), positionner DISPLAY par :

- pour un shell csh ou tcsh : **setenv DISPLAY termx005.utc :0**
- pour un shell bash : **export DISPLAY=termx005.utc :0**

Ceci permet aux fenêtres créées par netscape de s'afficher sur l'écran du terminal X, plutôt que sur l'écran du serveur.

Comment cela fonctionne-t-il ? Nous avons là un exemple de fonctionnement du type "client-serveur".

Netscape sur un terminal X, depuis un serveur (figure 47)

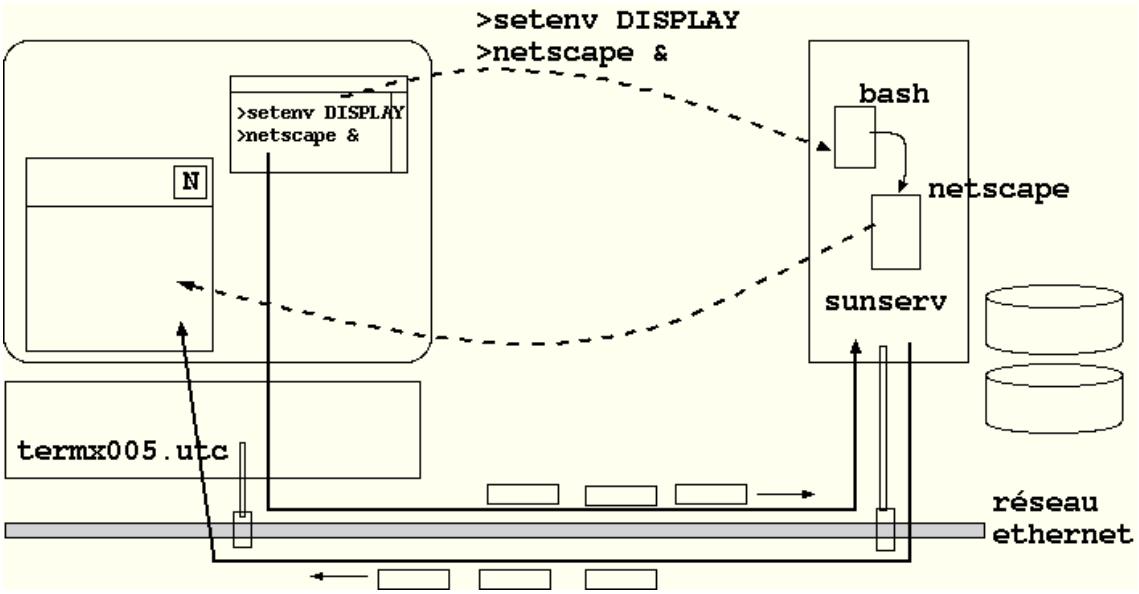


FIG. 47 –

7.6 Gestion des répertoires

Créer un répertoire .

commandes de manipulation des répertoires :

- répertoire par défaut (~) ("home directory") **cd**
- **cd rep**
- **cd ./rep/ssrep**
- **cd ../../rep**
- **mkdir rep**
- **mkdir rep/ssrep**
- **rmdir rep**
- **rm -R ./rep** ***** danger ! ! !**
-
- df

– du -kcs *
–

Lançer nedit et des xterm :
– **setenv DISPLAY termx005.utc :0**
– **nedit test.c &**
– **xterm &**
–

Création de répertoire sur le serveur (figure 48)

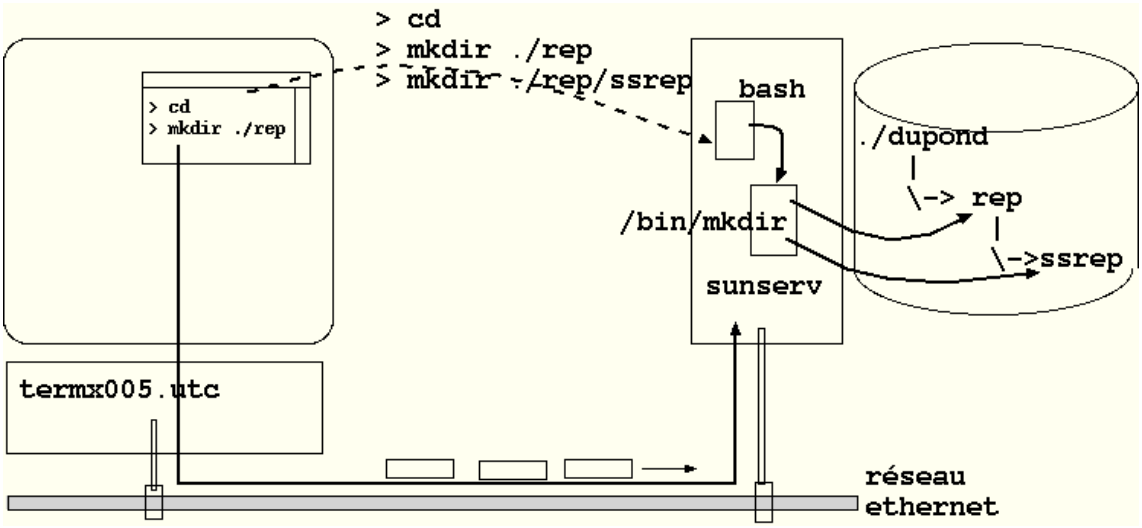


FIG. 48 –

Lancement d'un nedit et d'un xterm (figure 49)

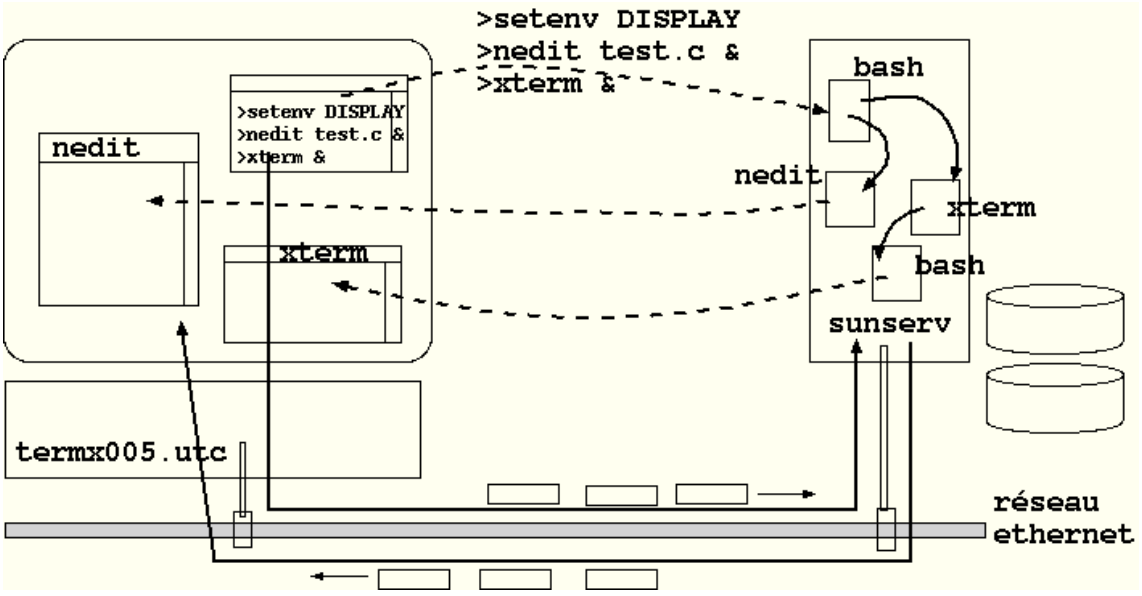


FIG. 49 –

7.7 Exploration du compte et des process de l'utilisateur

Identité :
- **whoami**
- **who am i**
- **who -m**
- **id**
-
- **tty**
- **stty**
- **stty -a**

Process de l'utilisateur :
- **ps**
- **ps -u**
- **ps e**
- **ps -j**
- **jobs**
-
- détruire un process : **kill PID** ou **kill %n**

```
[dupond@host work]$ ps -u
USER       PID %CPU %MEM   VSZ  RSS TTY  STAT START  TIME COMMAND
dupond  1106  0.0  1.5  2480 1428 pts/3  S    11:12  0:01 bash
dupond  7675  0.5  3.7  5800 3560 pts/3  S    19:16  0:02 nedit a.xml
dupond  7692  0.0  0.7  2588  748 pts/3  R    19:24  0:00 ps -u
```

7.8 Exploration du système

Identité :
- **uname -a**
- **host**

Process dans le système :
- **ps -aux**
- **ps -aef**
- **w**
- **who**

Statistiques :
- **top**
- **mpstat**
- **iostat**
- **vmstat**
- **netstat** : Active Internet connections
- **netstat -i** : interfaces
- **netstat -r** : routes
- **netstat -s** : statistiques
- **ifconfig (sous root)**

7.9 Poste de travail connecté à un serveur

Connexion telnet depuis un poste sur un serveur (figure 50)

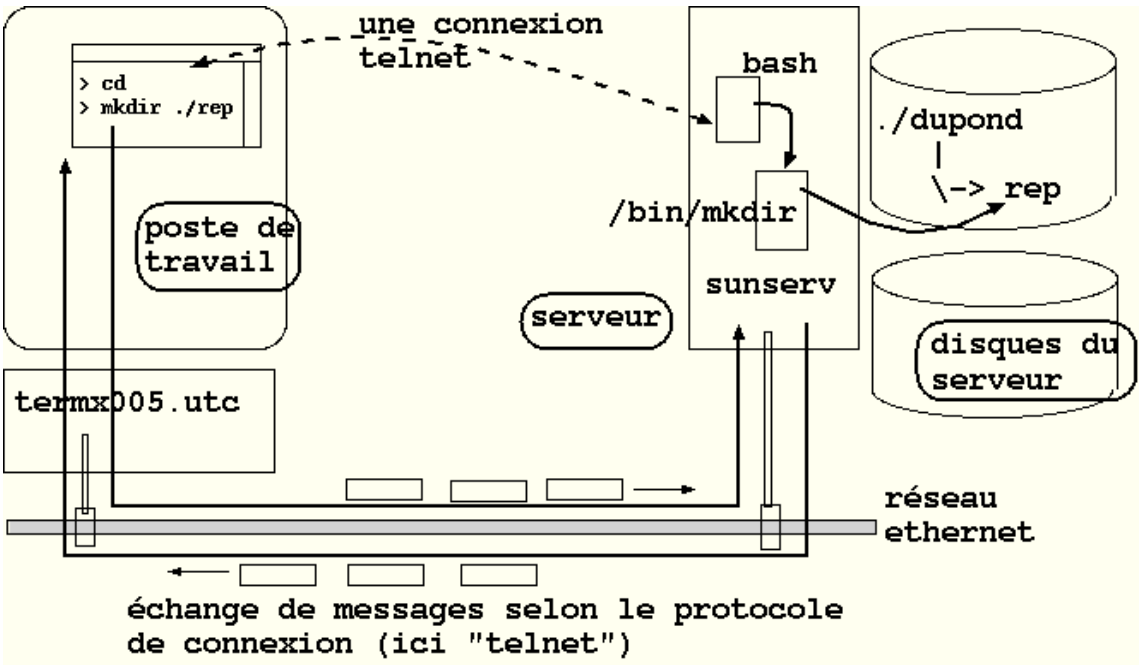


FIG. 50 –

Plusieurs connexions simultanées entre un poste et un serveur (figure 51 page 159)

Connexions multiples entre postes et serveurs (figure 52 page 159)

7.10 Gestion de fichiers et répertoires

Création
- **cp**
- **cat > file**
- **touch**

Destruction
- **rm**
- **mv**

Liste
- **ls , ls -l , ls -lgd , ls -rtl**
- **du**
- **df**
- **quota**

Protection
- **chown**
- **chgrp**

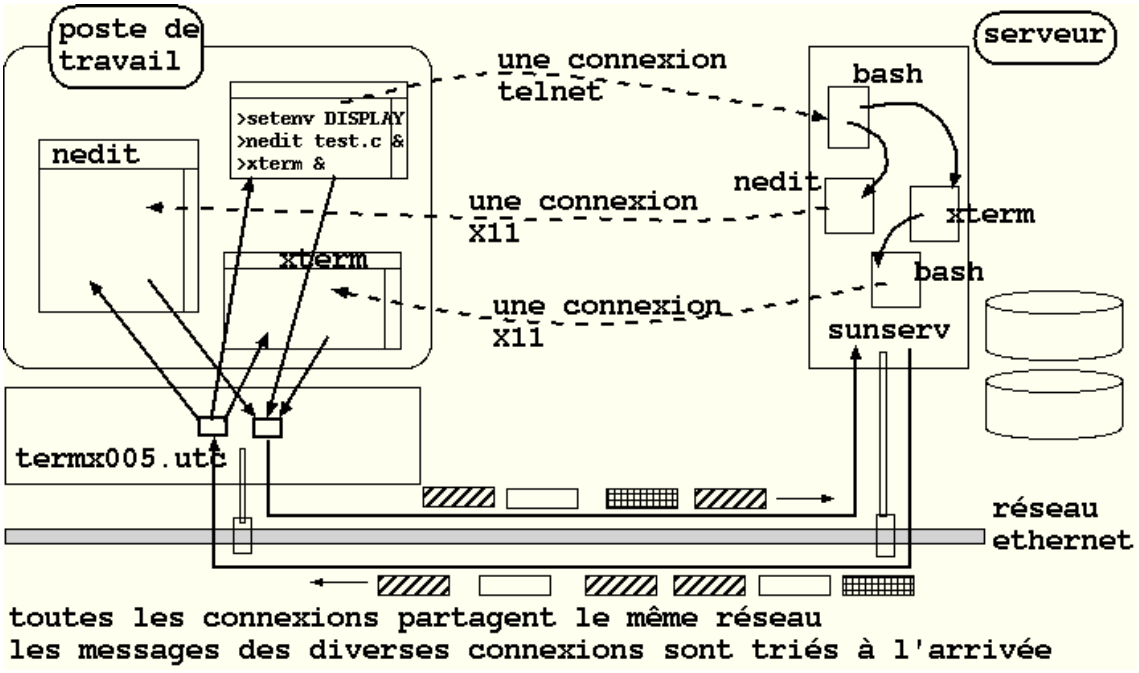

poste de travail
serveur
une connexion telnet
bash
>setenv DISPLAY
>nedit test.c &
>xterm &
nedit
nedit
une connexion X11
xterm
xterm
bash
une connexion X11
sunserv
termx005.utc
réseau ethernet
toutes les connexions partagent le même réseau
les messages des diverses connexions sont triés à l'arrivée

FIG. 51 –

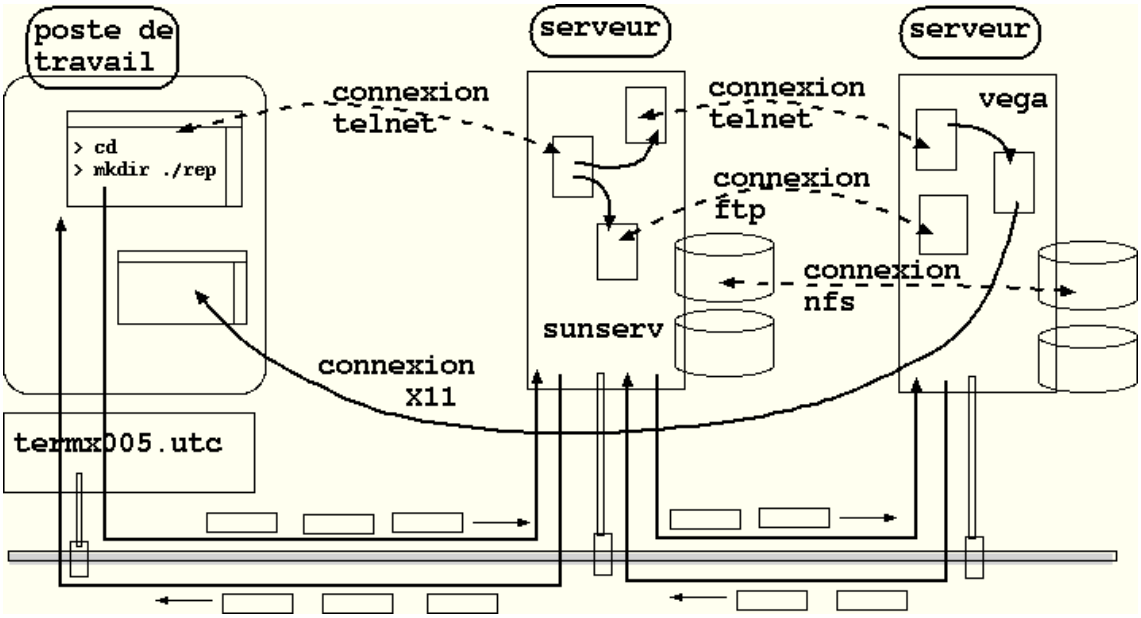

poste de travail
serveur
serveur
connexion telnet
connexion telnet
vega
> cd
> mkdir ./rep
connexion ftp
connexion nfs
sunserv
connexion X11
termx005.utc

FIG. 52 –

– **chmod**
Sauvegarde, restauration
– **zip** , **unzip**
– **compress** , **uncompress**
– **zcat**
– **gtar cf file ./rep** , **gtar xf file**
Recheche
– **find**
– find . \(-name '*.c' \) -print -exec grep -n "==" {} \ ;
– **grep**
– **locate**

7.11 Types de postes de travail

Différence entre "PC" et "Terminal X"
Aspect matériel
– **Commun :** UC, mémoire, carte réseau, carte graphique.
– **PC seul :** disque local (peut démarrer ("boot") seul.
– **TermX :** pas de disque : ne peut démarrer seul, a besoin d'un serveur réseau.
Aspect logiciel
– **Commun :**écran graphique, clavier, souris, gestionnaire de fenêtres ("window manager") qui gère les fenêtres et les évènements clavier et souris.
– **PC :** système exploitation local
– **TermX :** pas de système local. Seulement gestion graphique et fenêtres.
Exploitation
– **PC :** système exploitation local (sur disque dur local le plus souvent, mais pas obligatoirement), s'exécute sur l'UC locale et dans la mémoire locale ; les fichiers stockés soit localement, soit sur un **serveur de fichiers** (NFS en unix, SMB en windows).
– **TermX :** aucun stockage local. Un programme de démarrage stocké en mémoire morte (PROM) charge par le réseau, depuis un serveur le logiciel qui fait fonctionner le terminal : un "serveur X11" (gestion de l'affichage graphique au niveau des pixels) et un gestionnaire de fenêtres. Selon la configuration le gestionnaire de fenêtres peut être exécuté par un serveur unix (cas des logins en "XDM").
Exécution des applications
– **PC :** exécution locale dans la mémoire du PC et par l'UC du PC.
– **TermX :** exécution distante ("remote") dans la mémoire du serveur et par l'UC du serveur.
Fichiers utilisateurs
– **PC :** au choix, en local ou sur un disque du serveur, partagé et monté à distance ("remote mount") par l'utilisation de protocoles de partage de fichiers (NFS en unix, SMB en windows).
– **TermX :** sur le serveur.
Contrôle d'accès
– **PC :** local ou par le serveur (choix par l'administrateur du système).
– **TermX :** celui du serveur sur lequel on se connecte.

– **TermX :** depuis un même terminal X, on peut se connecter simultanément sur plusieurs serveurs, ceci avec des identifications différentes (par ex. jdupond/passxx sur vega et lo03199/passyy sur sunserv.

En cas de plantage

– **PC :** risque de détérioration de fichiers ou du système de fichiers sur le disque local, et eventuellement de non redémarrage.
– **TermX :** redémarre toujours sauf panne matérielle ou arrêt du serveur.

7.12 Éviter les confusions

Différences entre notions similaires

PC et Terminal X

– disque local / prog. téléchargé

Terminal X et Terminal

– graphique (X11) / alphanumérique

Terminal et Telnet

– terminal physique / logiciel de connexion

Telnet et Ethernet

– logiciel de connexion / matériel de mise en réseau

Ethernet et TCP/IP

– matériel de mise en réseau / protocoles et logiciels

TCP/IP et accès réseau

– protocoles et logiciels TCP/IP + matériel réseau => donnent accès réseau

Accès réseau et accès Internet

– accès réseau interne n'implique pas accès Internet

Accès Internet et web

– le web n'est que l'une des applications Internet

web et ftp

– web = une application Internet d'accès à des serveurs de pages d'information
– ftp = une application Internet de transfert de fichiers

ftp et adresse e-mail

– ftp : transfert de fichiers entre systèmes de fichiers / e-mail : échange de messages géré par les serveurs mail et les MTA (Mail Transfert Agent)

adresse e-mail et vega

– l'adresse e-mail jean.dupond@etu.utc.fr est gérée par la machine vega.utc.fr

vega et compte user/passwd

– vega gère des comptes user/passwd depuis lesquels on peut exécuter un MTA (netscape ou pine)

compte user/passwd et directory

– vega gère des comptes user/passwd, chacun d'eux possédant un répertoire (directory) dans lequel sont stockés les fichiers de l'utilisateur du compte.

directory et partage de fichiers

– un répertoire (directory) peut être partagé entre la machine serveur (celle sur lequel est connecté le disque qui contient ce répertoire et une ou plusieurs autres machines.

partage de fichiers et fichiers

– un partage de fichiers consiste à mettre en place les autorisations et logiciels capables de donner accès à un fichier depuis une autre machine. Une fois mis en place

sur un répertoire, tous les fichiers du répertoire sont accessibles. Le partage n'est pas lui-même un fichier.

fichier et programme

– un fichier contient des données (textes ou binaires), un programme est un texte particulier, contenu dans un fichier : fichier = conteneur, programme = contenu.

programme et process

– un programme est un texte particulier, contenu dans un fichier. Lorsqu'il est exécuté, il est amené en mémoire et rangé dans une structure particulière, créée dynamiquement, appelée process. Programme = texte statique, process = zone mémoire dont le contenu évolue pendant l'exécution du process.

process et fenêtre

– un process = des structures en mémoire. Fenêtre = objet graphique. Un process peut demander au système la création d'une fenêtre. Des appels systèmes permettent au programme de modifier le contenu de la fenêtre.

fenêtre et PC ou fenêtre et TermX

– Fenêtre = objet graphique, associé à un process et géré (emplacement, recouvrement, évènements) par le gestionnaire de fenêtre qui lui-même s'exécute sur le PC ou sur le TermX ou sur le serveur.

7.13 Insertion d'un système d'exploitation dans un réseau

Aujourd'hui, dans pratiquement tous les cas, y compris pour les systèmes personnels, un ordinateur et son système d'exploitation sont insérés dans un réseau de machines et utilisent des fonctions diverses pour les échanges entre machines : connexion à distance (telnet, NIS), partage de disques ou de sous-ensembles de disques (NFS, SMB), partage d'écrans (X11, VNC), échange de fichiers (FTP), requêtes et réponses web (http), connexion à distance entre programmes (sockets), appel de procédures à distance (RPC, XMLRPC), etc ...

On va détailler un peu quelques unes de ces connexions.

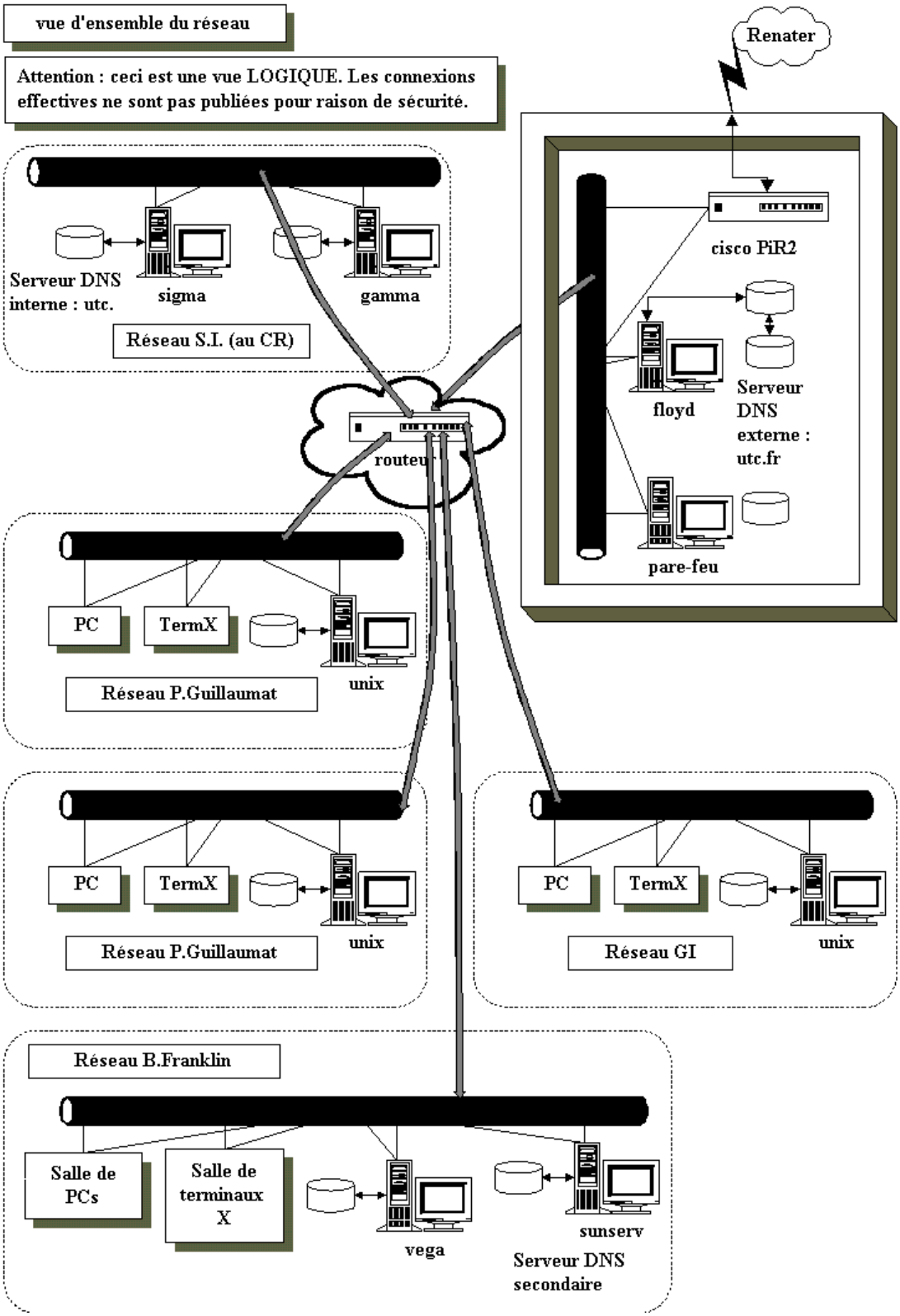


FIG. 53 – Insertion d'un système dans un réseau

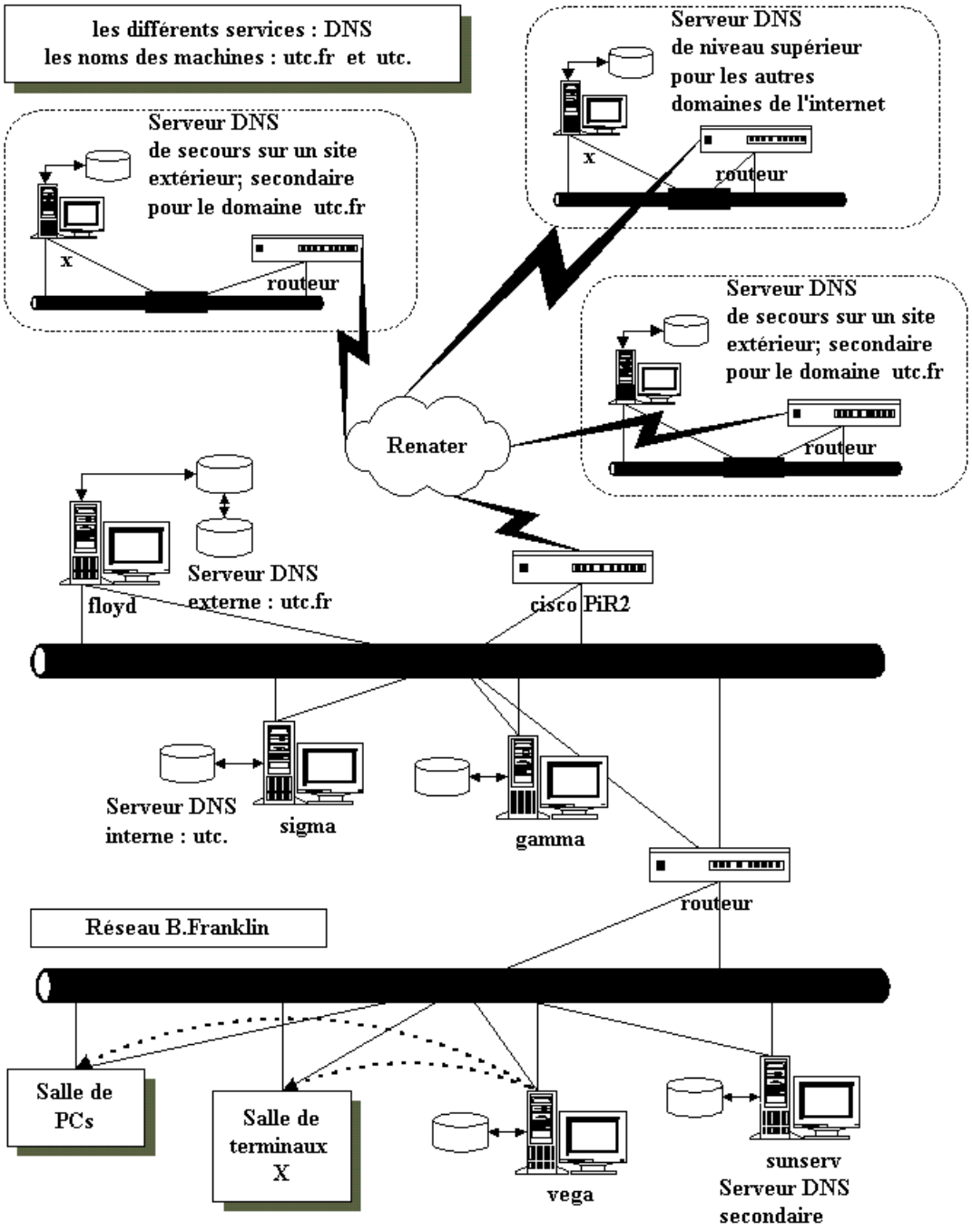


FIG. 54 – Insertion d'un système dans un réseau : DNS

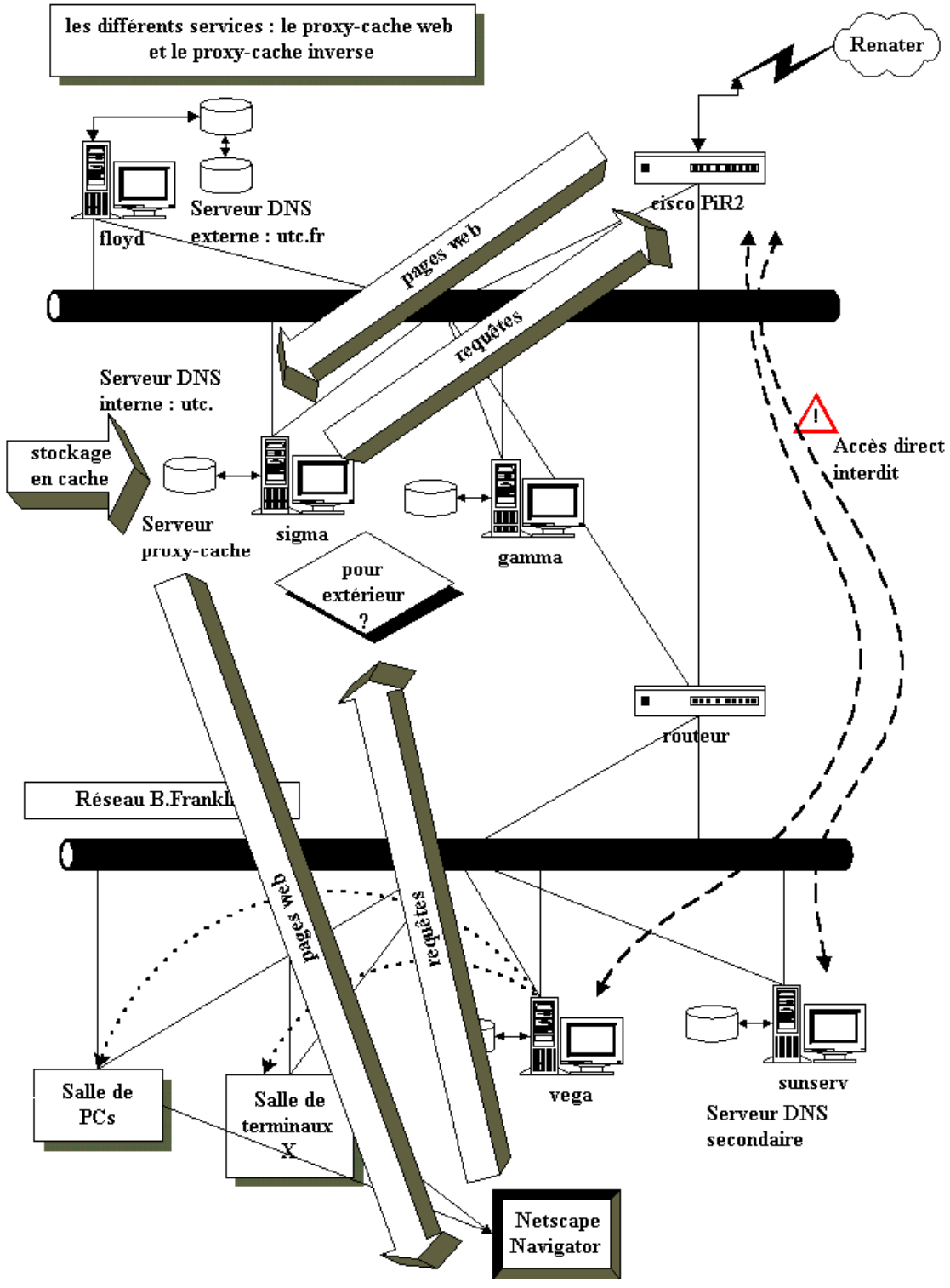


FIG. 55 – Insertion d'un système dans un réseau : poxy

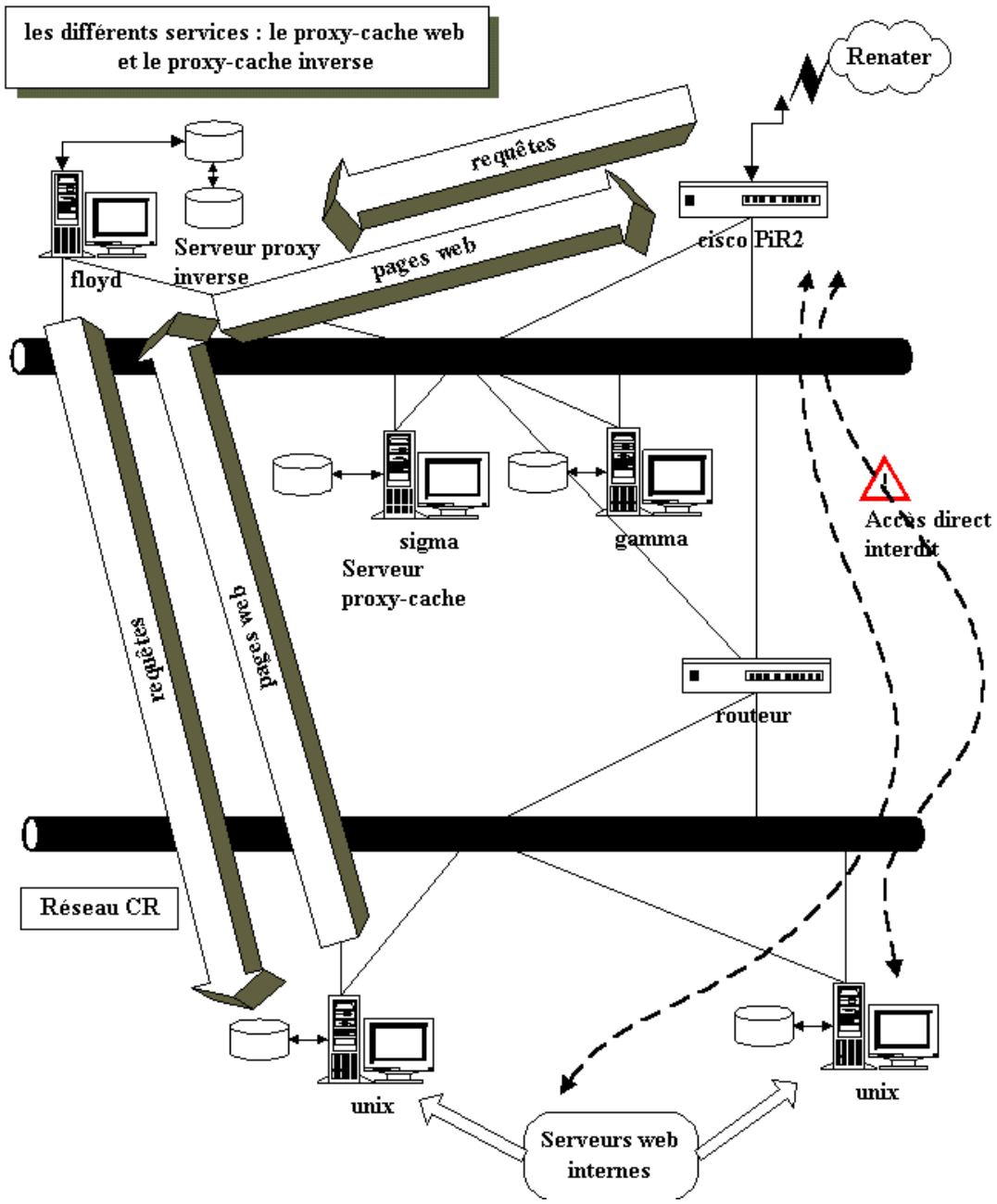


FIG. 56 – Insertion d'un système dans un réseau : proxy inverse

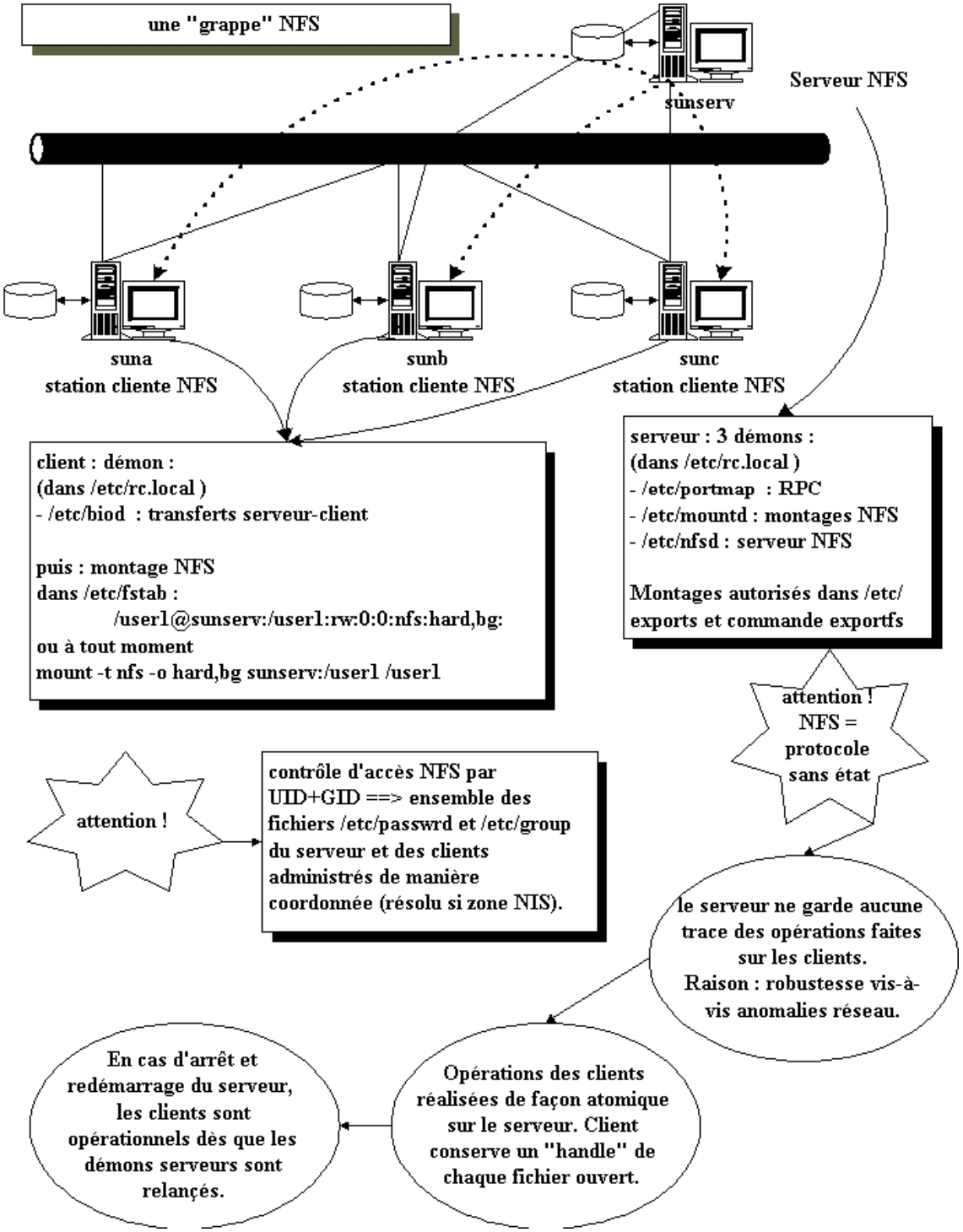


FIG. 57 – Insertion d'un système dans un réseau : NFS

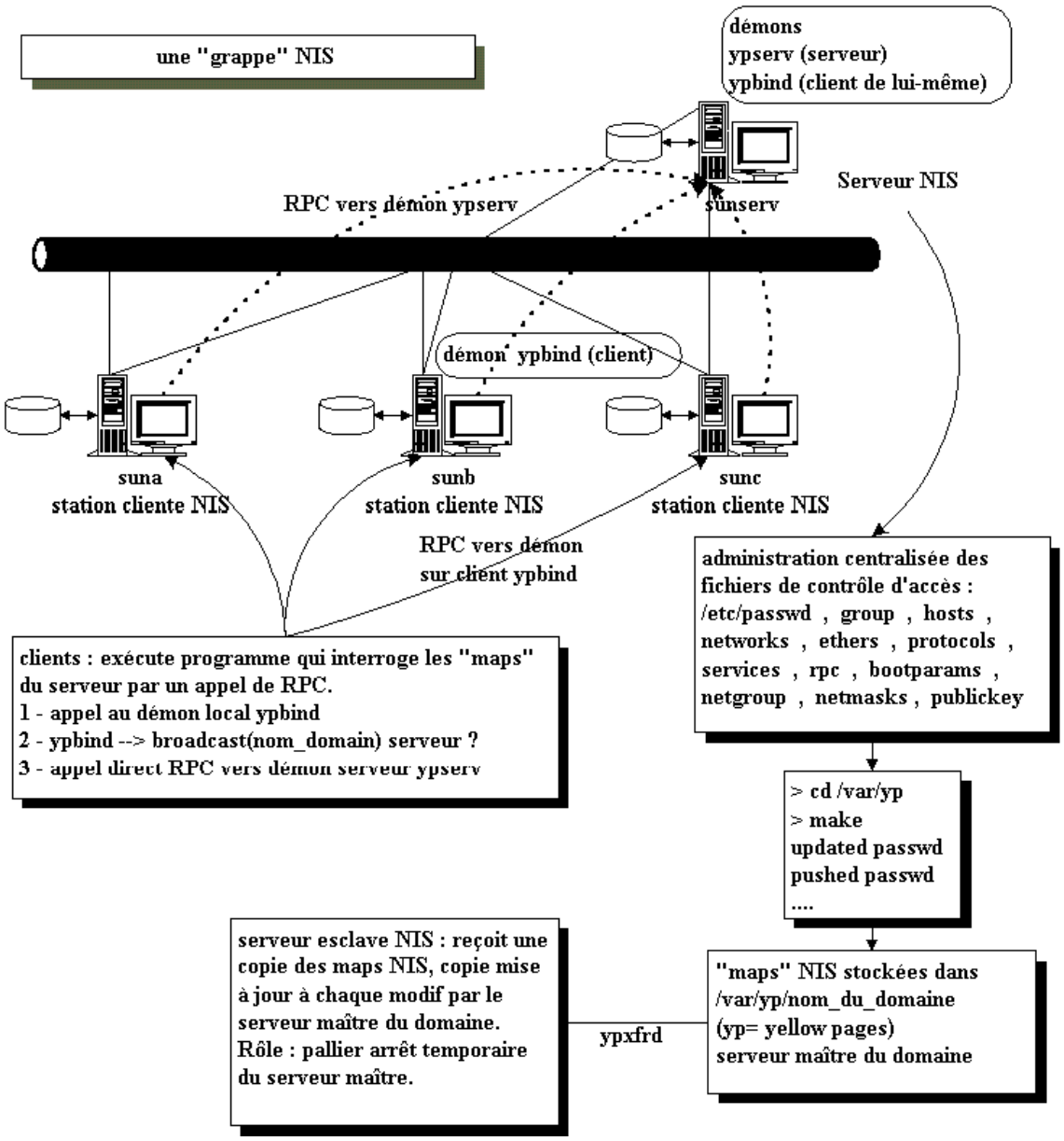


FIG. 58 – Insertion d'un système dans un réseau : NIS

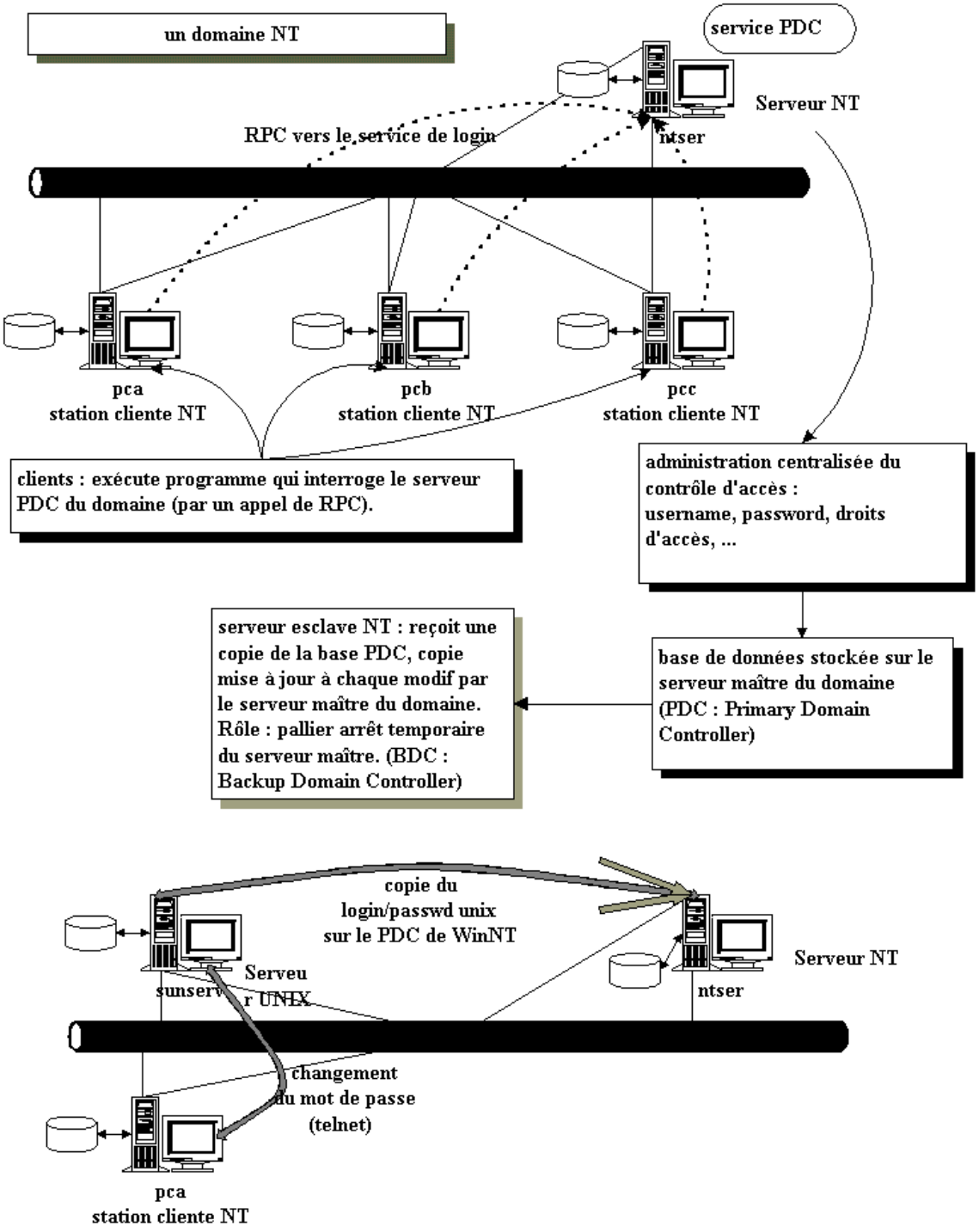


FIG. 59 – Insertion d'un système dans un réseau : PDC

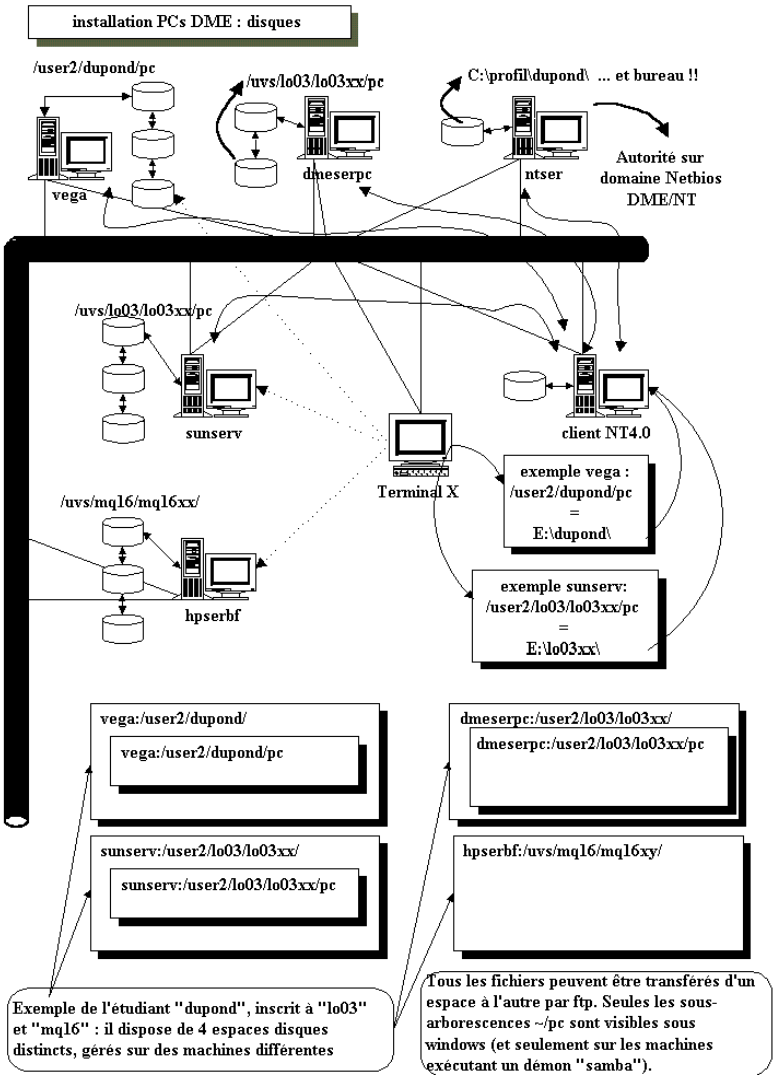


FIG. 60 – Insertion d'un système dans un réseau : SMB

SR01 2003 - Cours Unix 8 - Écriture d'un programme et compilation

8 SR01 2003 - Cours Unix 8 - Écriture d'un programme et compilation

8.1 Introduction

Cet exposé montre le passage du texte d'un programme dans un langage de programmation, à des instructions exécutables par un ordinateur.

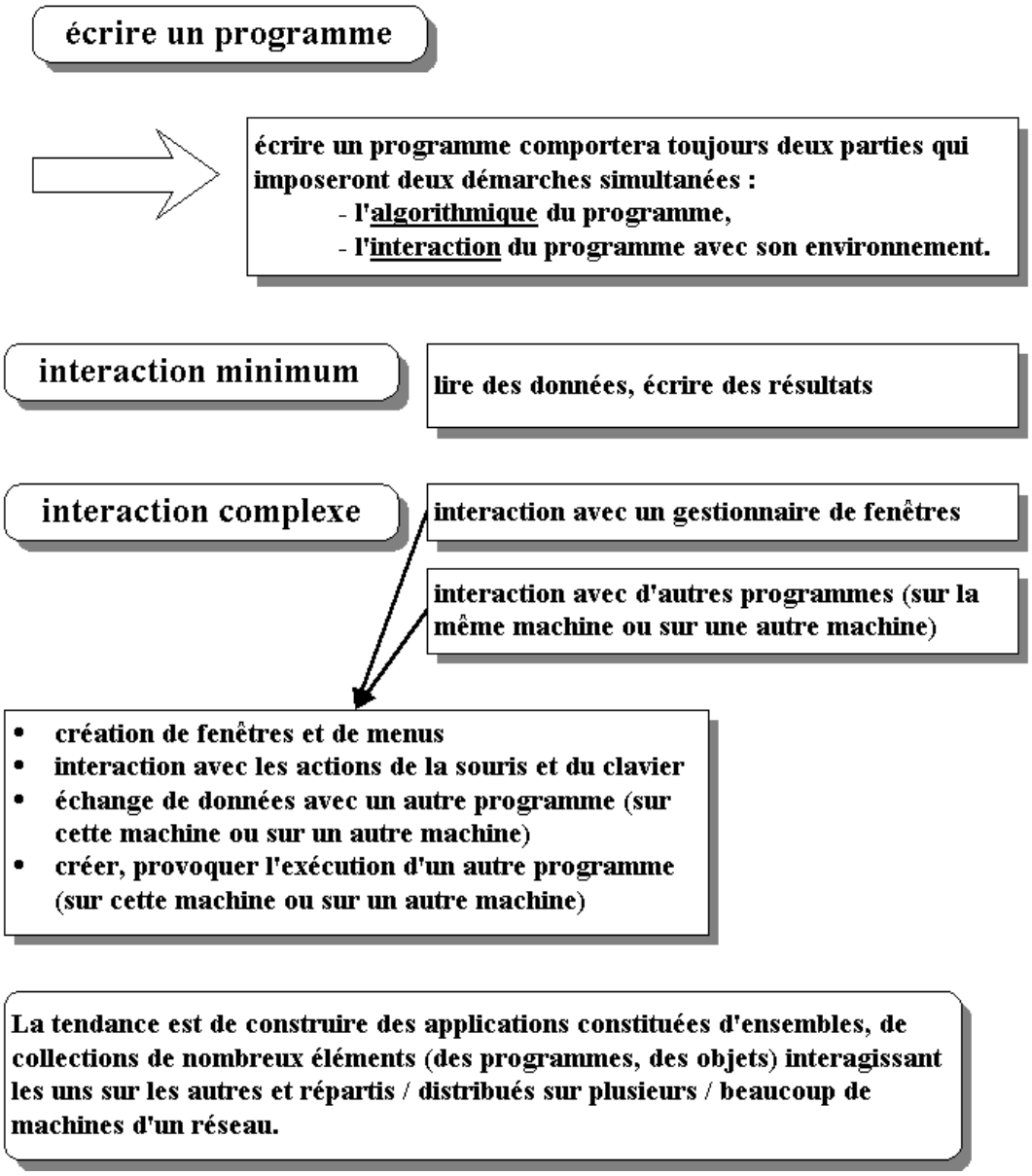
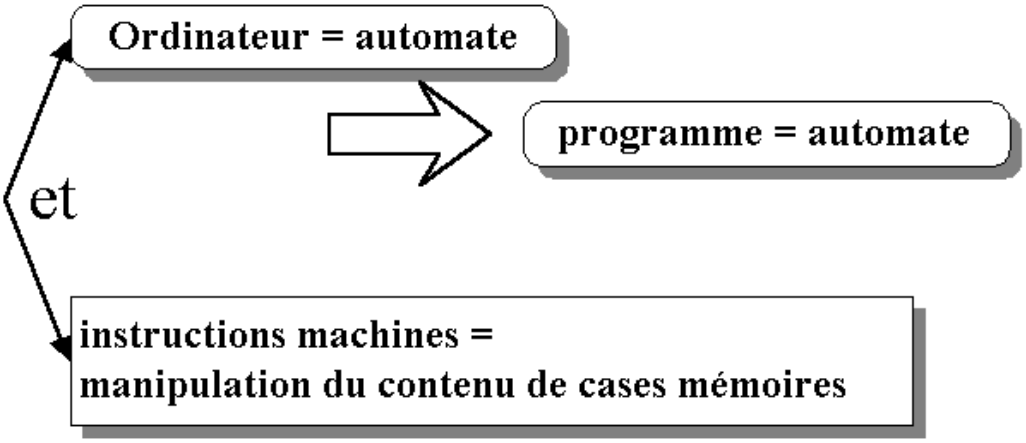


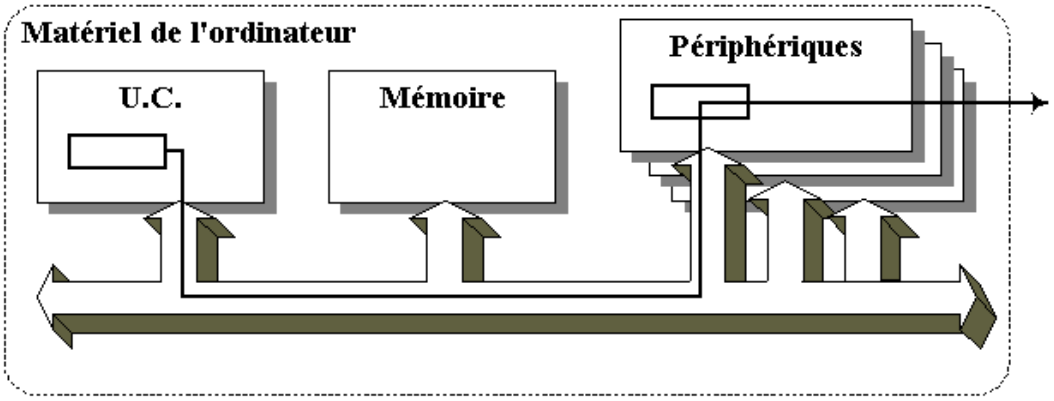
FIG. 61 – Écrire un programme (1/5)

FIG. 62 – Écrire un programme (2/5)



... alors comment la machine fait-elle pour communiquer ??
(avec d'autres sur un réseau ou avec l'utilisateur)

Certaines de ces cases mémoire sont en réalité des "portes" de commande des dispositifs matériels d'entrées-sorties.



MAIS :

Il faut des ***milliers*** de ces instructions élémentaires pour faire des actions telles que :

- écrire un fichier sur disque,
- afficher une fenêtre sur un écran,
- ouvrir un menu, suite à un "clic" de souris,
- lire des caractères tapés sur le clavier,
- ...

FIG. 63 – Écrire un programme (3/5)

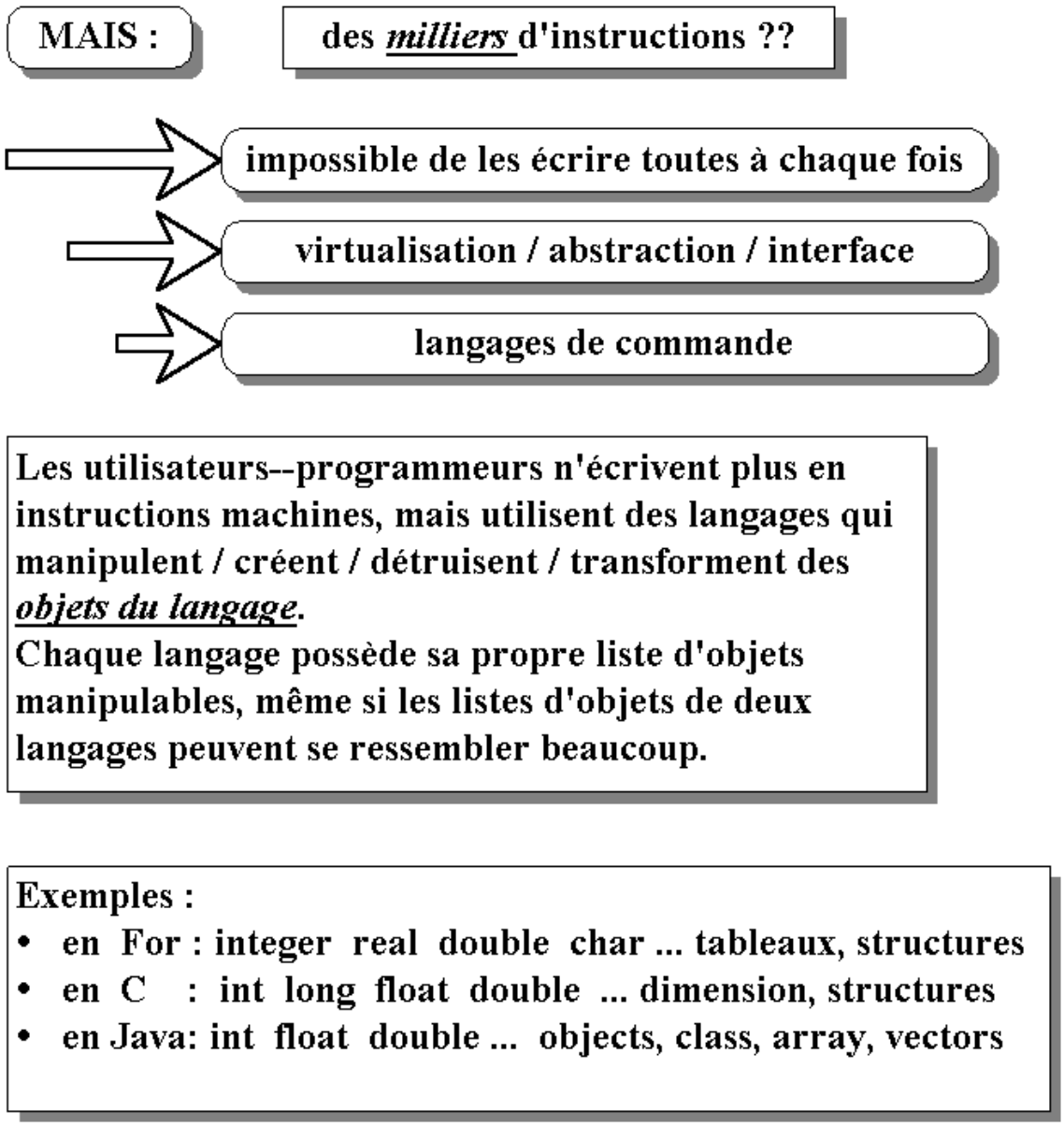


FIG. 64 – Écrire un programme (4/5)

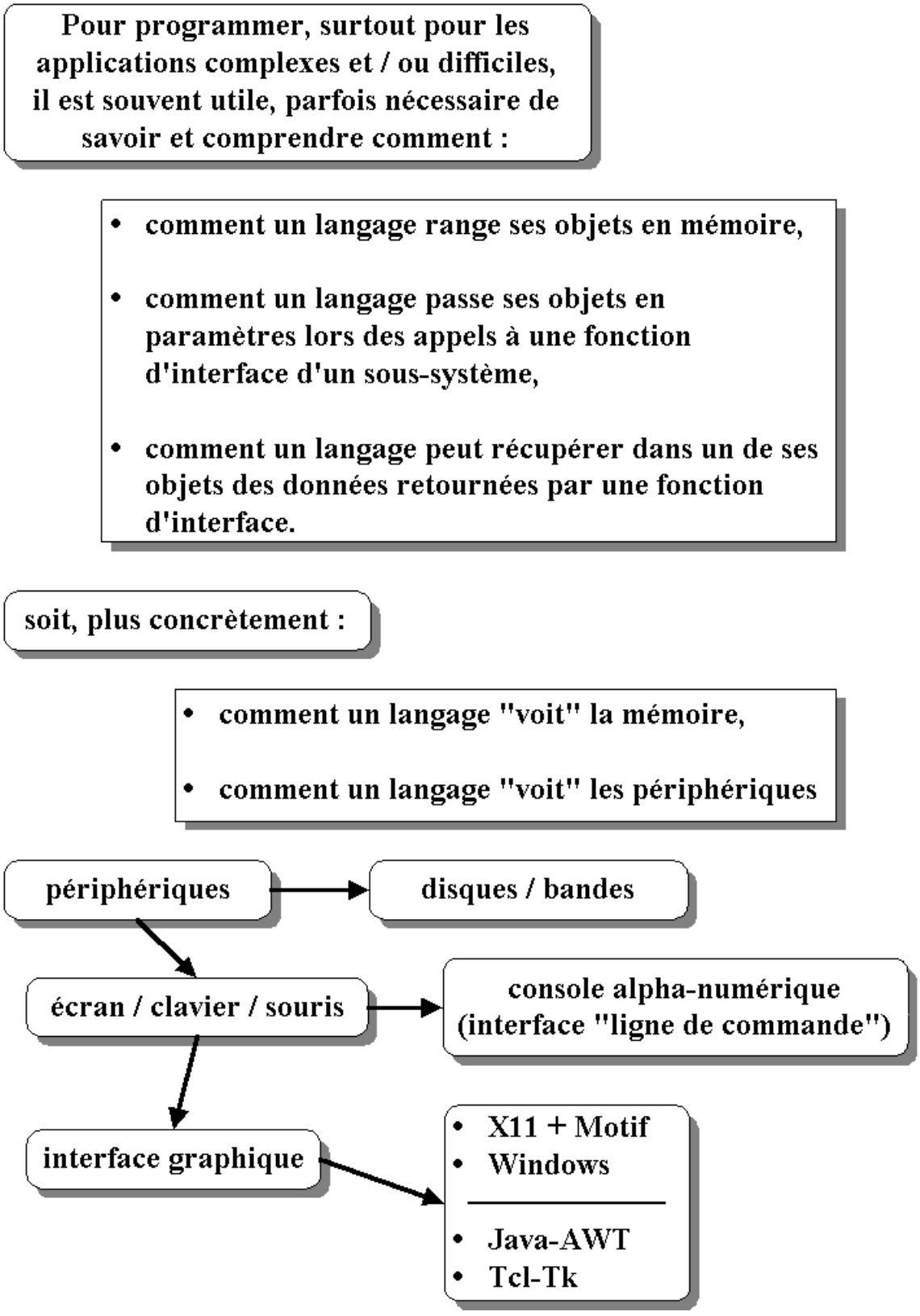


FIG. 65 – Écrire un programme (5/5)

8.2 Langages interprétés et langages compilés

Il y a deux grandes classes de langages qui permettent de transformer un texte de programme écrit dans le langage en une suite d'instructions machine exécutables sur un ordinateur donné : les langages interprétés et les langages compilés.

Langages interprétés

Le texte source du programme est lu par un **interpéteur** (qui est lui même un programme). L'interpréteur examine chaque instruction et vérifie sa conformité à la syntaxe du langage. Puis il utilise une sorte de table interne qui va lui donner, pour chaque instruction du langage, la **traduction** de cette instruction en une suite d'opérations qui seront réalisées par l'interpréteur lui-même.

Exemple simple

Le programme présente une instruction `a = b + c ;`. L'interpéteur analyse cette instruction comme "b et c sont les arguments d'une opération d'addition dont le résultat doit être rangé dans a". L'interpéteur possède une fonction du genre `re= somme(a1,a2) ;`.

Il va faire quelque chose ressemblant à la séquence suivante :

```
a1 = b;  # ranger le contenu de b dans la variable interne a1
a2 = c;  # idem pour c et a2
i = index("somme")  # trouver l'index i de la fonction somme
re = ptftab[i](a1,a2);  # appel à une fonction interne dont
# l'adresse est rangée dans un tableau de pointeurs de fonctions
a = re;  # ranger le résultat dans la variable destination
```

Exemples de langages interprétés : perl, python, php, javascript, lisp,...

Langages compilés

Le texte source du programme est lu par un **compilateur**. Celui-ci examine chaque instruction et vérifie sa conformité à la syntaxe du langage. Puis il utilise une sorte de table interne qui va lui donner, pour chaque instruction du langage, la **traduction** de cette instruction en une suite **d'instructions machines élémentaires**.

Toutes ces instructions machines (assembleur) sont assemblées pour former un module exécutable, selon un processus de **compilation et d'édition de liens ("link")** qui va être décrit ci-après.

Exemples de langages compilés : Pascal, C, Fortran, C++, Ada, Java, ...

Remarque : certains langages interprétés peuvent aussi posséder une option de compilation : perl, python, lisp,...

Remarque : le cas de java est un peu à part ; en effet, il est compilé, mais le résultat de la compilation (bytecode) est lui-même interprété.

L'avantage du mode interprété est évident : l'exécution est immédiate s'il n'y a pas d'erreurs de syntaxe, mais elle implique du travail supplémentaire par rapport à un programme compilé exécutant directement du code machine. Donc, l'avantage du mode interprété est la vitesse d'exécution (en général de 2 à 4 fois plus vite).

8.3 Compilation : passage au langage machine (Fig. 66 à 89)

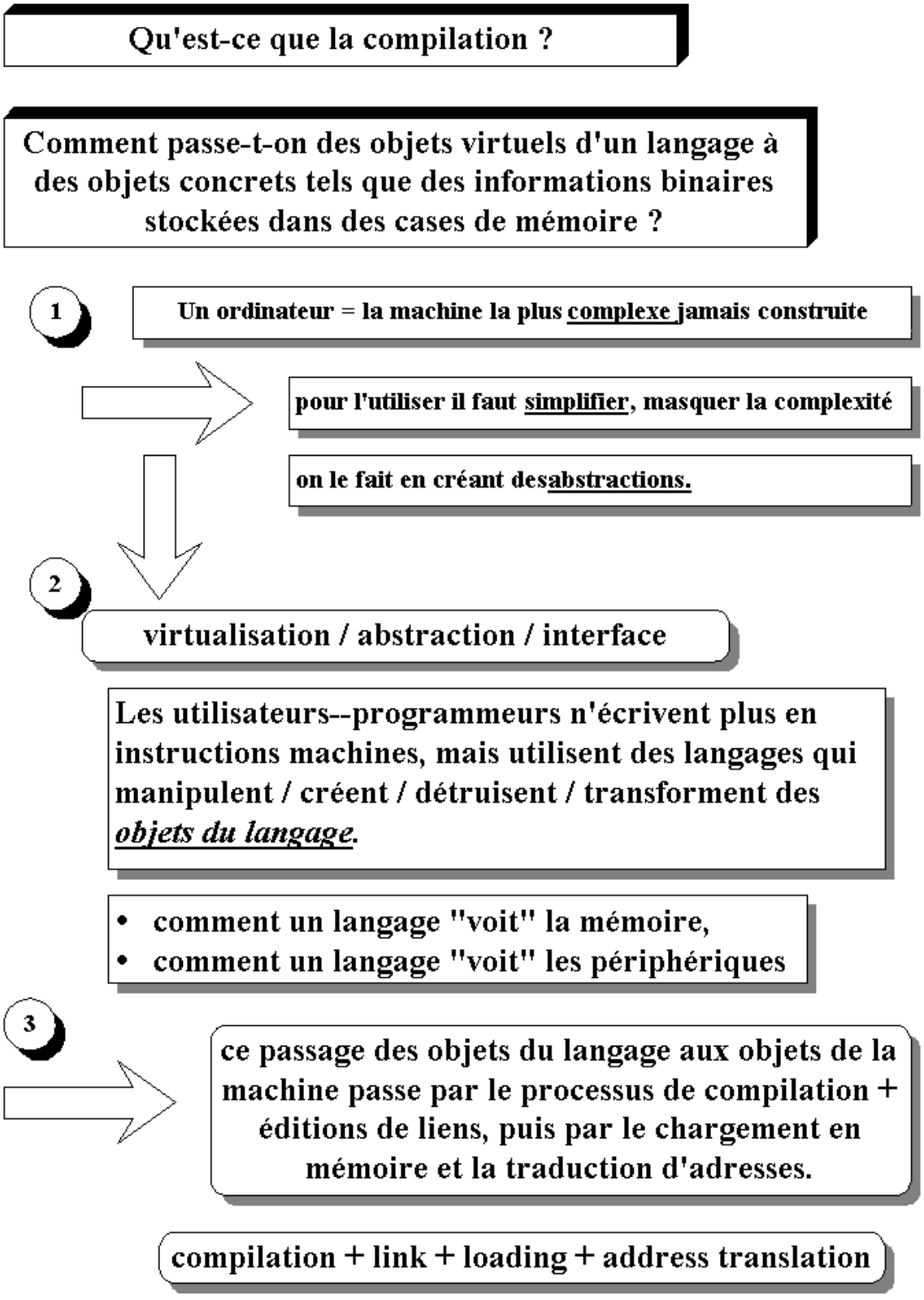
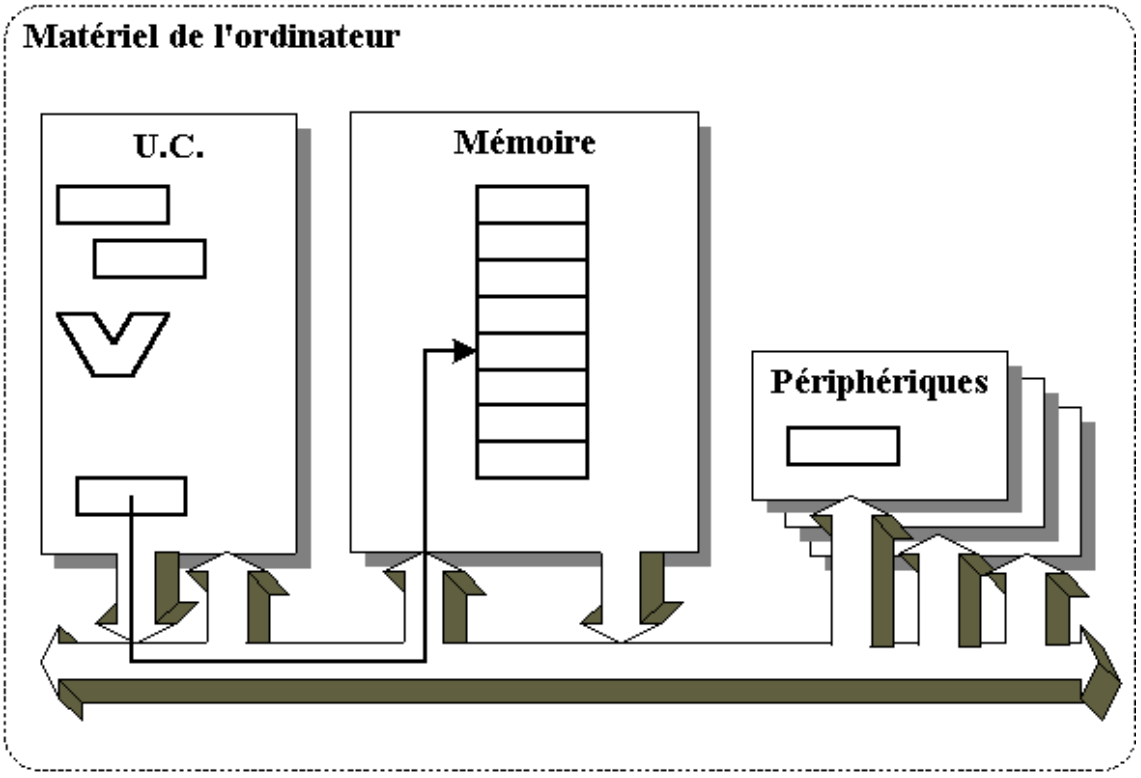


FIG. 66 – Compilation (1/24)

Ordinateur = automate

instructions machines =
manipulation du contenu de cases mémoires



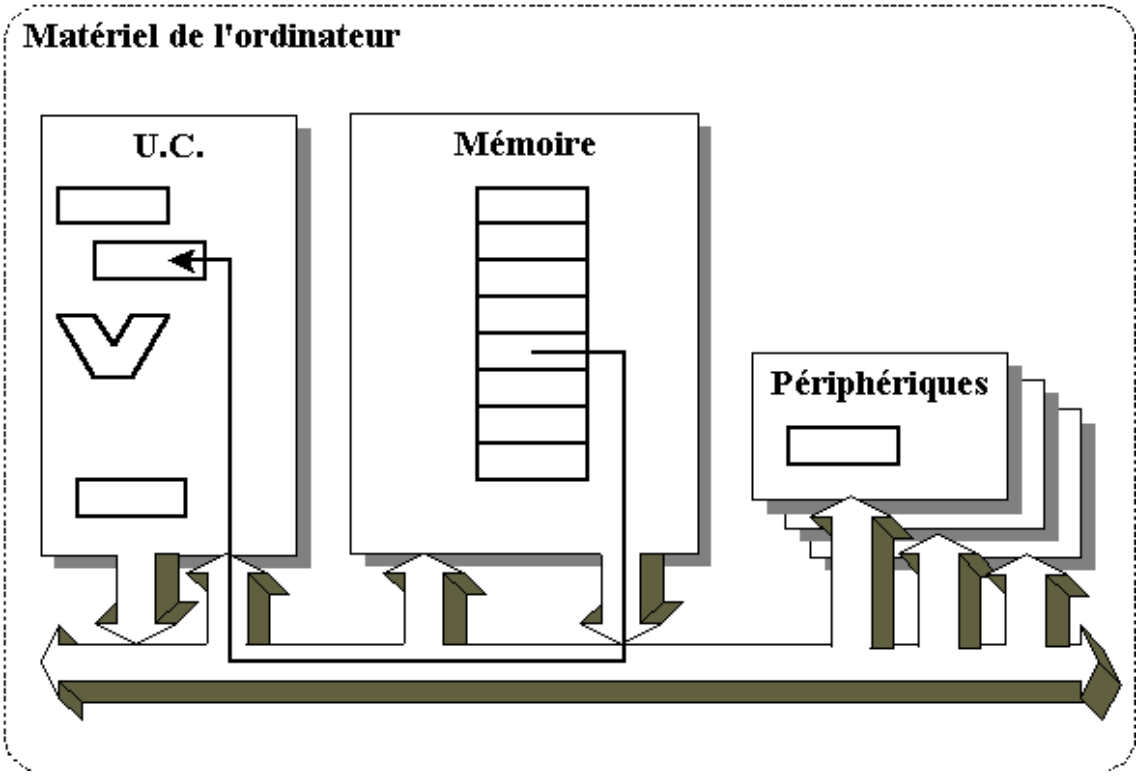


FIG. 67 – Compilation (2/24)

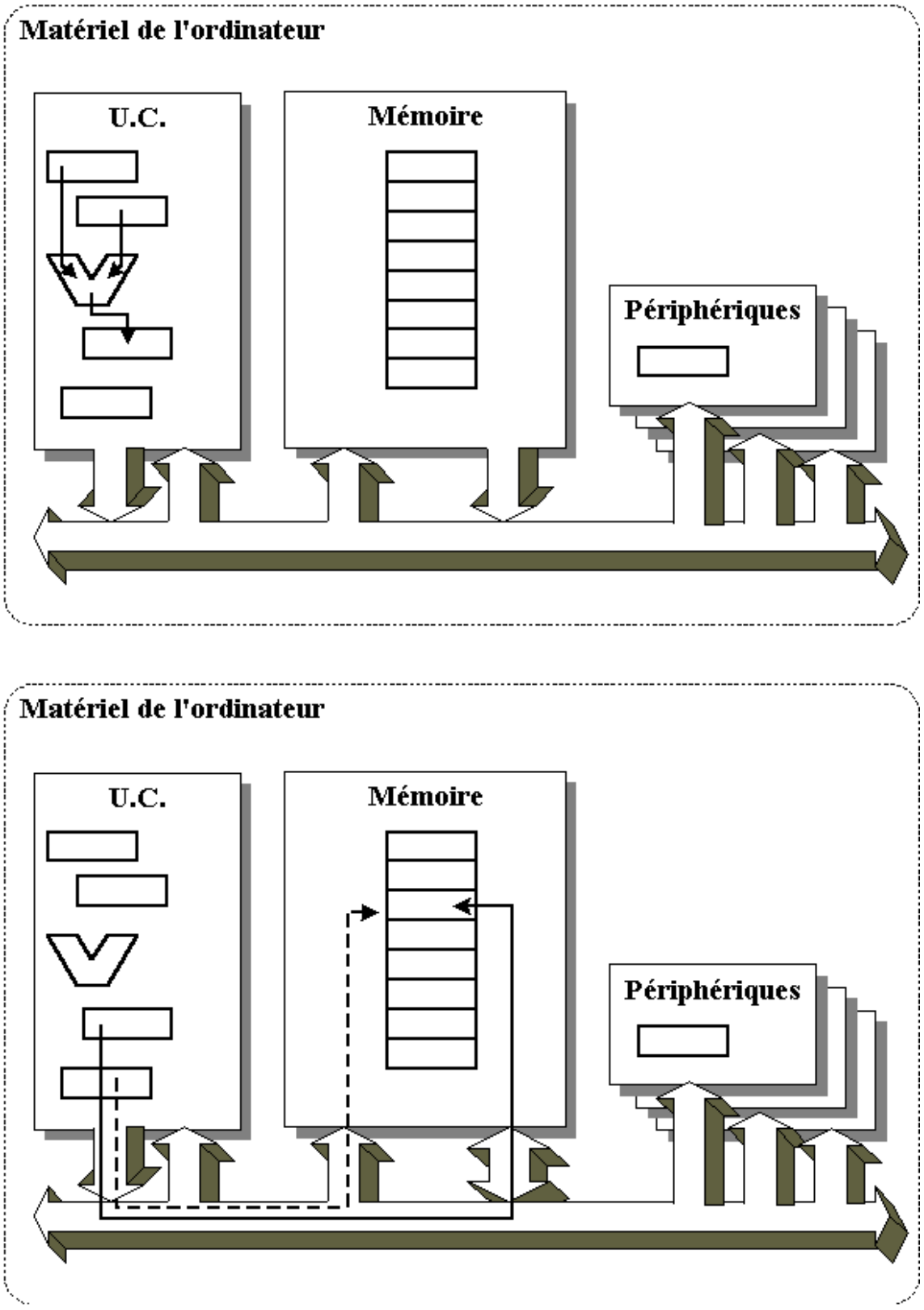


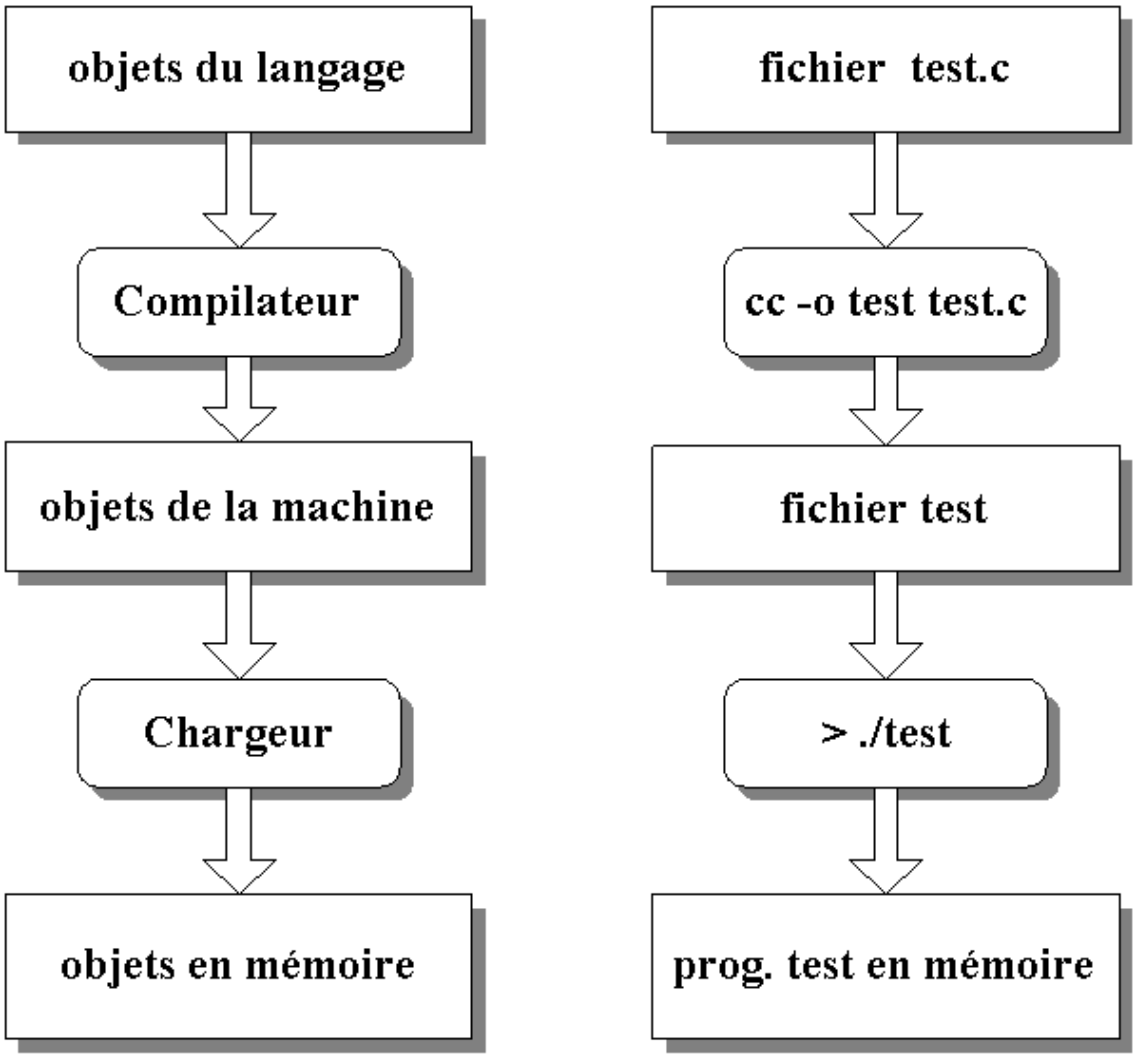
FIG. 68 – Compilation (3/24)

Compilation = passage
des objets du langage --> aux objets de la machine

les objets de la machine = contenu de cases mémoires
= valeurs binaires

- **ces valeurs binaires n'ont pas de signification particulière**
- **c'est l'interprétation que l'on en fait qui leur confère une signification**
- **ces valeurs binaires sont donc des représentations d'objets du monde réel**
- **la correspondance entre les nombres binaires et les objets réels est un choix arbitraire des concepteurs de la machine et des langages avec lesquels ont la programme**
- **pour programmer il faut donc :**
    - **comprendre cette représentation**
    - **comprendre comment le langage effectue la correspondance**
    - **comprendre comment la machine manipule les représentations**
    - **comment on peut, par des expressions du langage, provoquer les traitements qui vont réaliser les fonctions de l'application que l'on est en train d'écrire.**

FIG. 69 – Compilation (4/24)




```
/* prog. test */
#include <stdio.h>
int A = 12;    /* déclarations    */
int B = 23;    /* initialisations */
int C;         /* de variables    */
/* code exécutable */
main()
{    C = A + B;
     printf ("C= %d\n",C);
}
```

FIG. 70 – Compilation (5/24)

```
/* prog. test */
#include <stdio.h>
int A = 12;    /* déclarations    */
int B = 23;    /* initialisations */
int C;         /* de variables    */
/* code exécutable */
main()
{    C = A + B;
     printf ("C= %d\n",C);
}
```

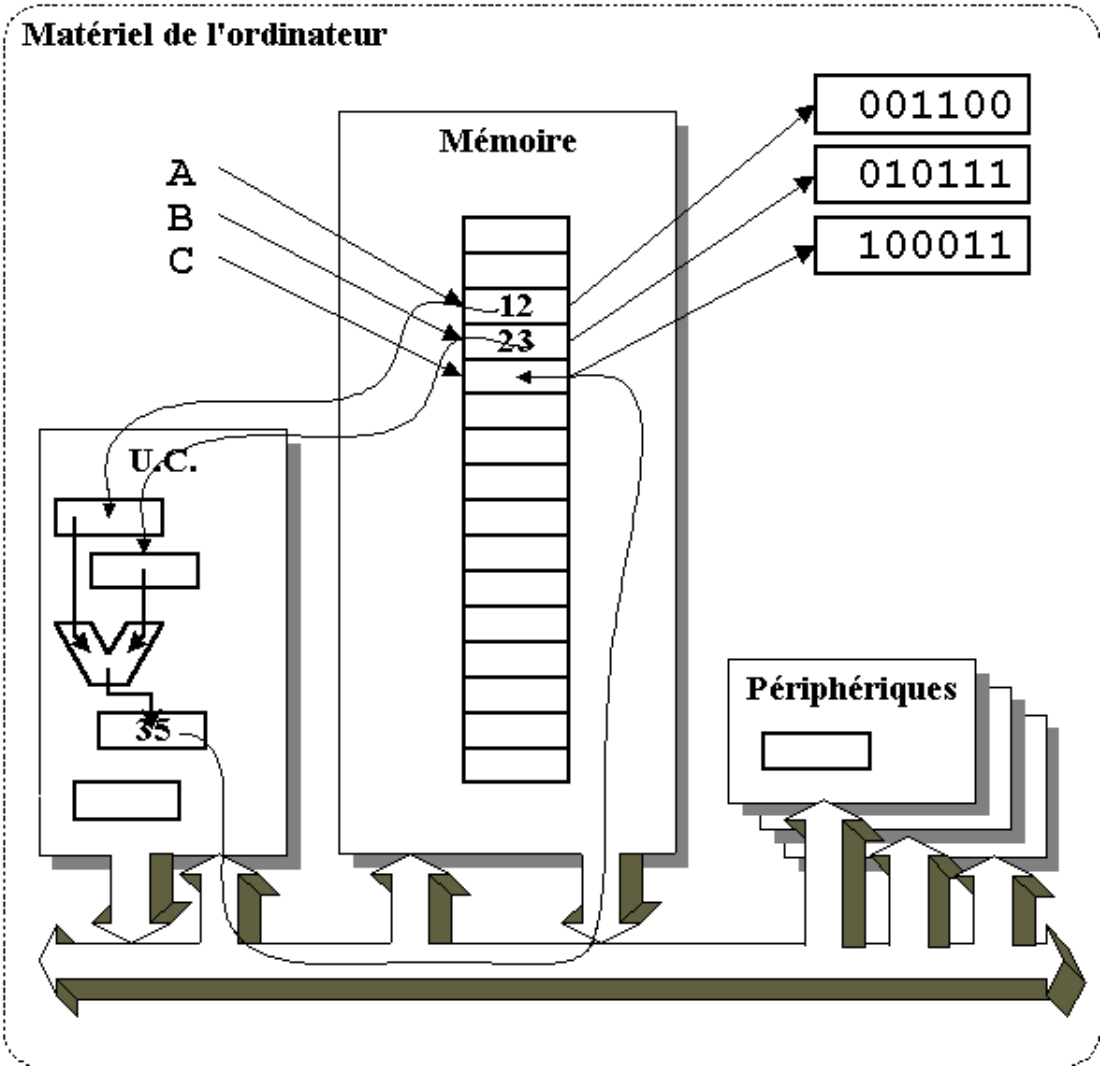
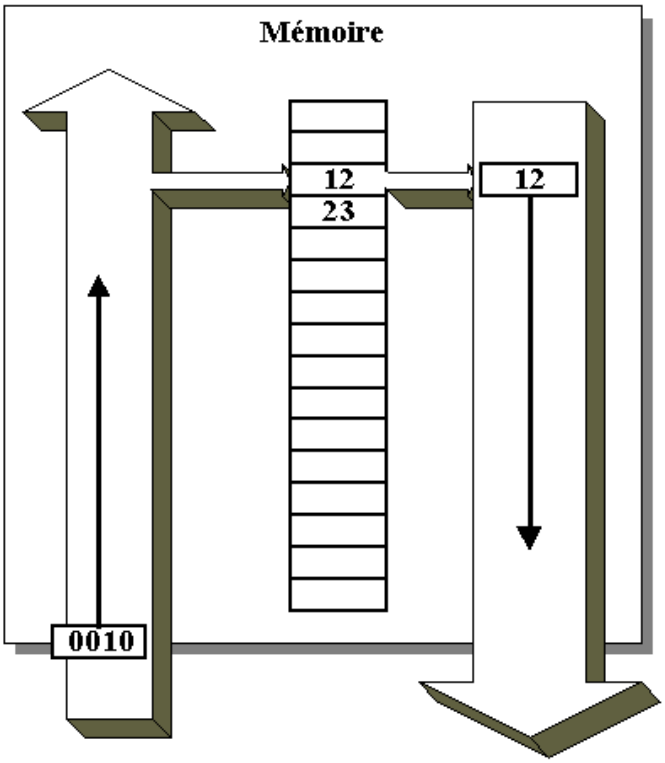


FIG. 71 – Compilation (6/24)

Adresses et Contenus

instructions machines =
manipulation du contenu de cases mémoires

Comment la machine désigne-t-elle / retrouve-t-elle une case mémoire particulière ?



Toutes les cases de mémoire sont numérotées (de 0 à 2^N-1).

Ce numéro s'appelle leur *adresse*.

Il est utilisé par le matériel pour désigner la case mémoire dans laquelle on veut lire ou écrire.

Une adresse mémoire est pour la machine une valeur binaire comme une autre. Elle peut donc la manipuler comme elle manipule les contenus.

La machine peut donc manipuler deux sortes d'informations : des adresses et des contenus.

FIG. 72 – Compilation (7/24)

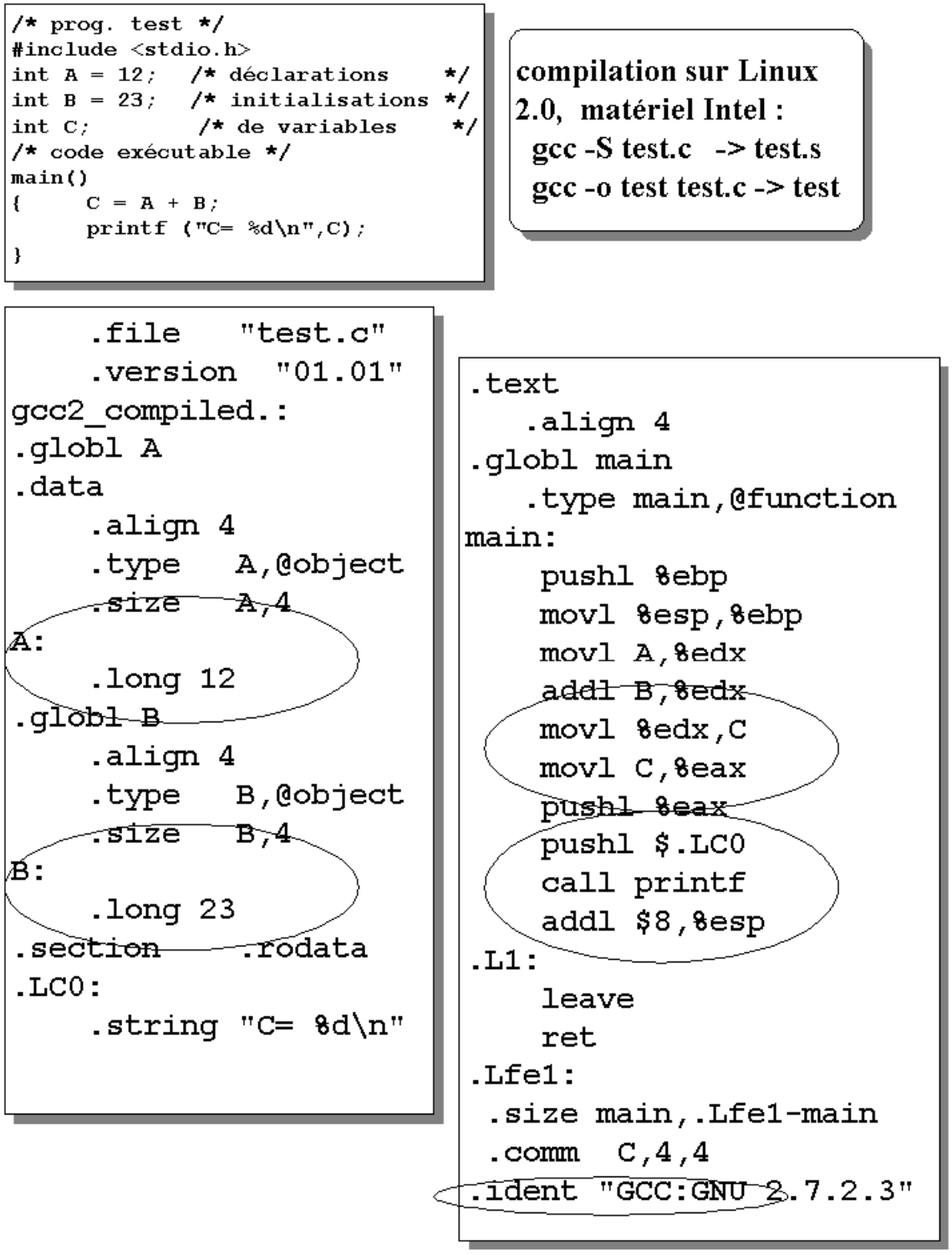


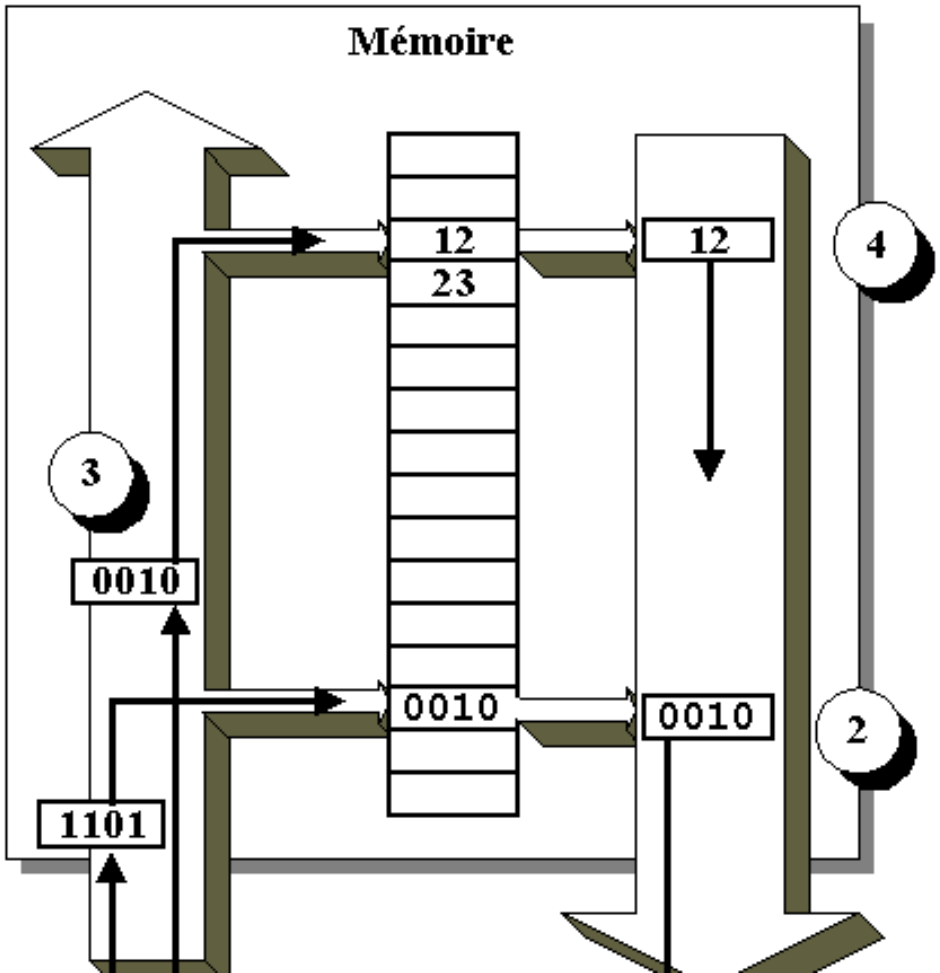
FIG. 73 – Compilation (8/24)

Adresses et Contenus

La machine peut donc manipuler deux sortes d'informations : des adresses et des contenus.

Comme adresses et contenus sont tous deux des nombres binaires, la machine peut utiliser un contenu pour en faire une adresse et aussi utiliser des adresses dans des opérations pour produire de nouvelles adresses et les stocker comme des contenus.

Un contenu utilisé comme une adresse s'appelle un pointeur (ou une référence).



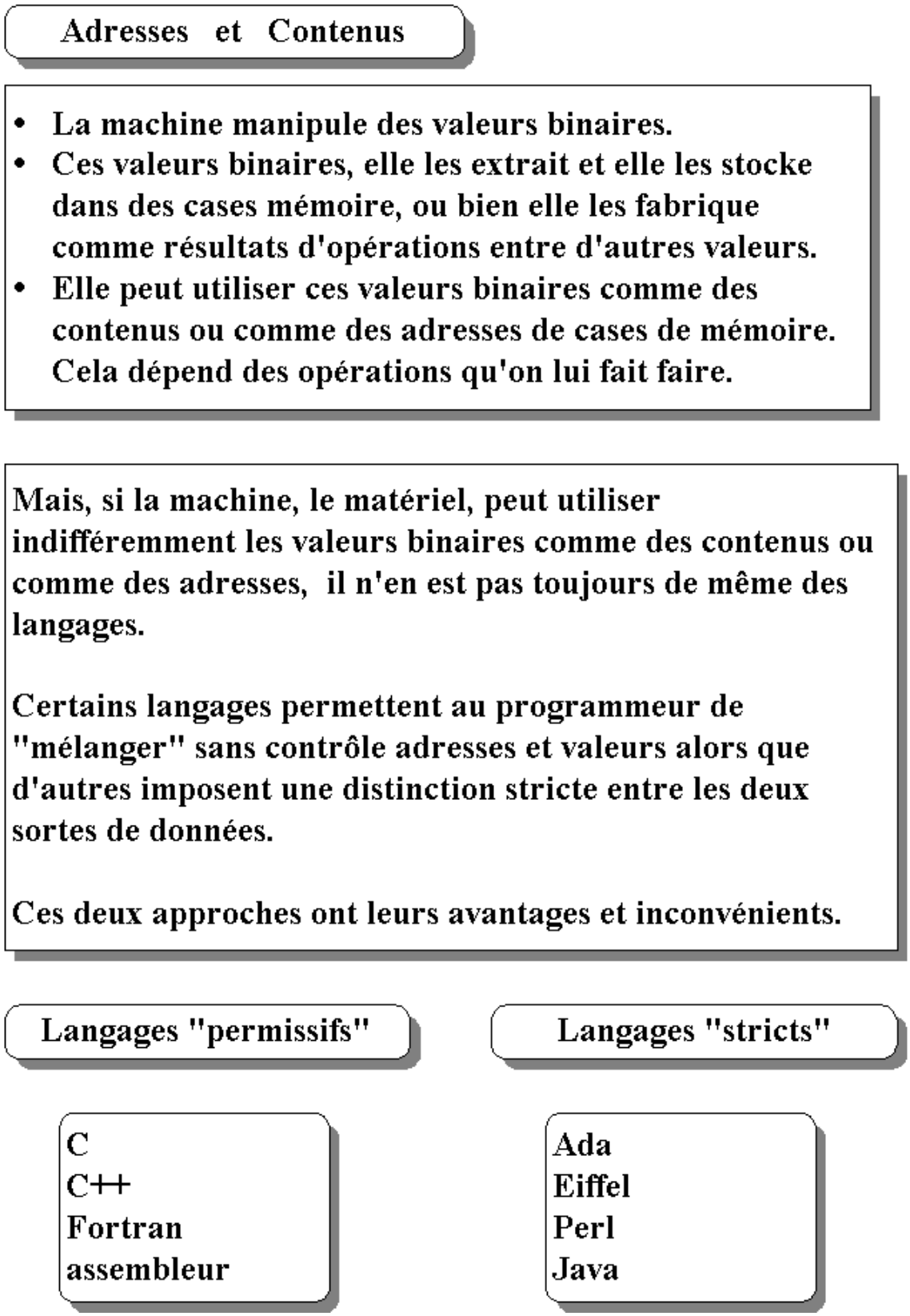
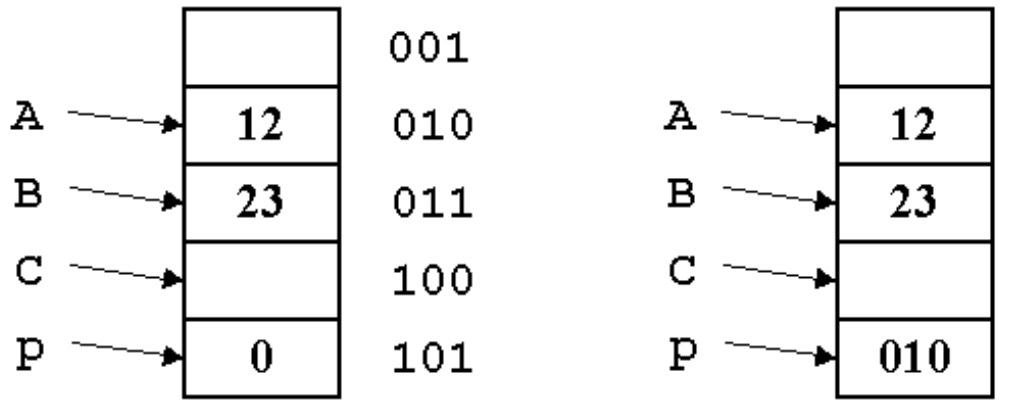


FIG. 75 – Compilation (10/24)

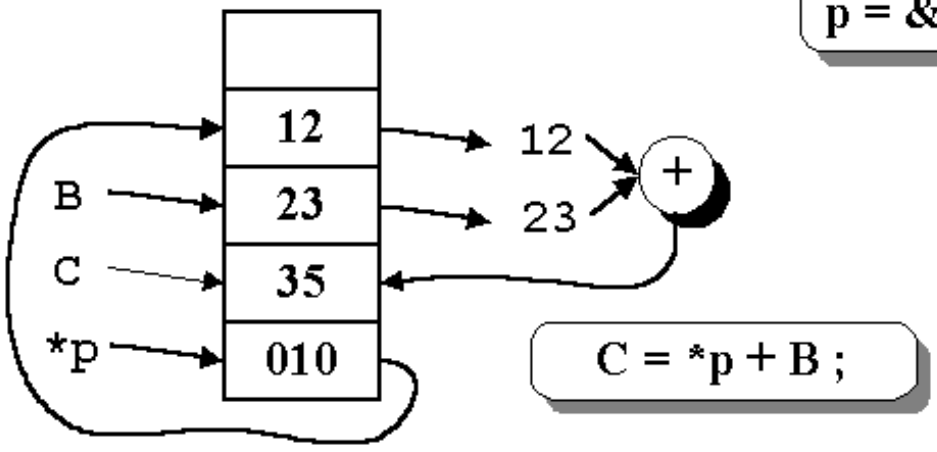
Les pointeurs en C

```
/* prog. ptr */
#include <stdio.h>
int A = 12;
int B = 23;
int C;
int *p;     /* un pointeur */
/* code exécutable */
main()
{    p = &A;
     C = *p + B;  /* C = A + B */
     printf ("C= %d\n",C);
}
```

Un pointeur contient l'adresse d'une variable



p = &A ;



C = *p + B ;

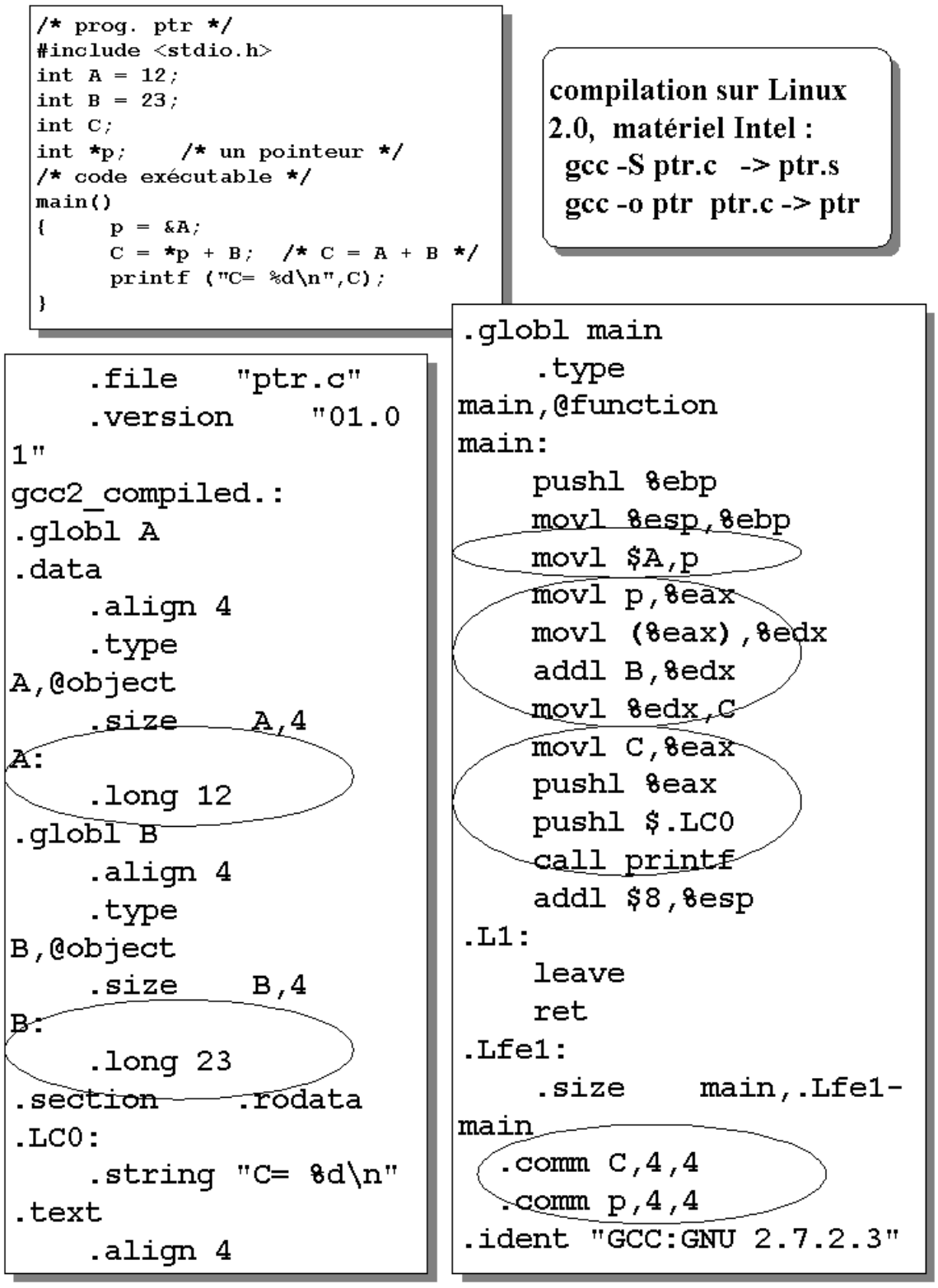


Fig. 77 – Compilation (12/24)

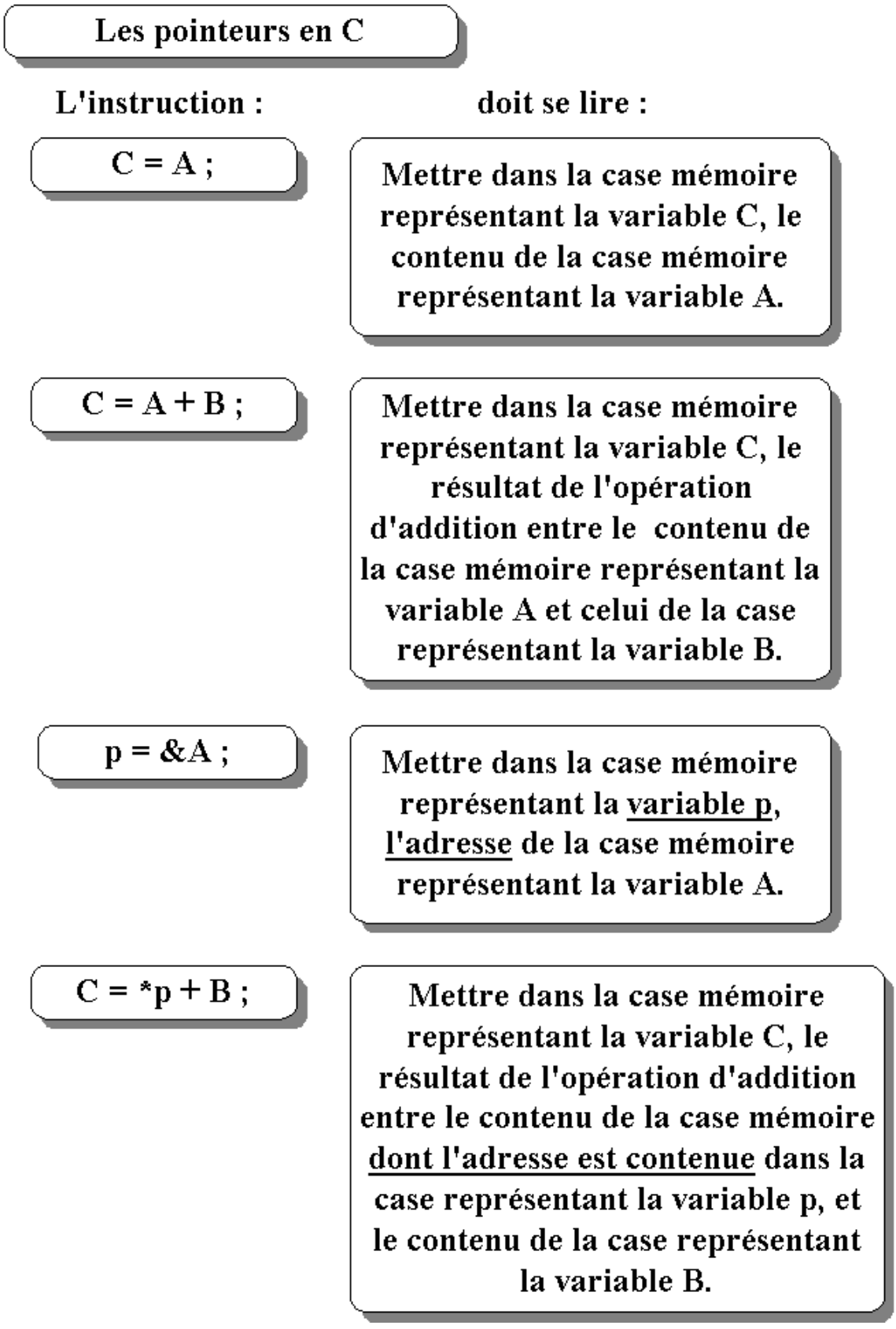
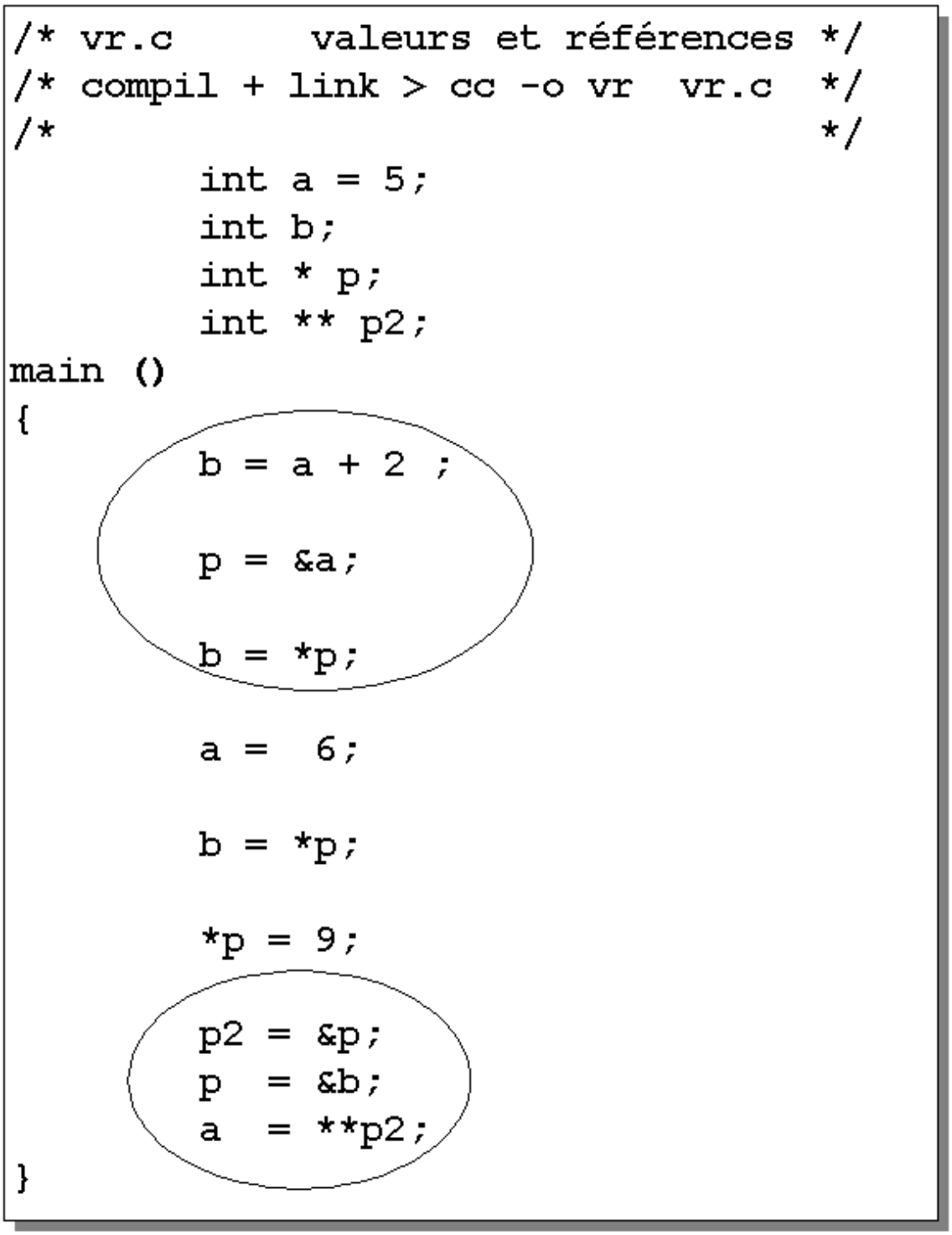


FIG. 78 – Compilation (13/24)



```
/* vr.c      valeurs et références */
/* compil + link > cc -o vr  vr.c  */
/*                                 */
        int a = 5;
        int b;
        int * p;
        int ** p2;
main ()
{
        b = a + 2 ;

        p = &a;

        b = *p;

        a =  6;

        b = *p;

        *p = 9;

        p2 = &p;
        p  = &b;
        a  = **p2;
}
```

FIG. 79 – Compilation (14/24)

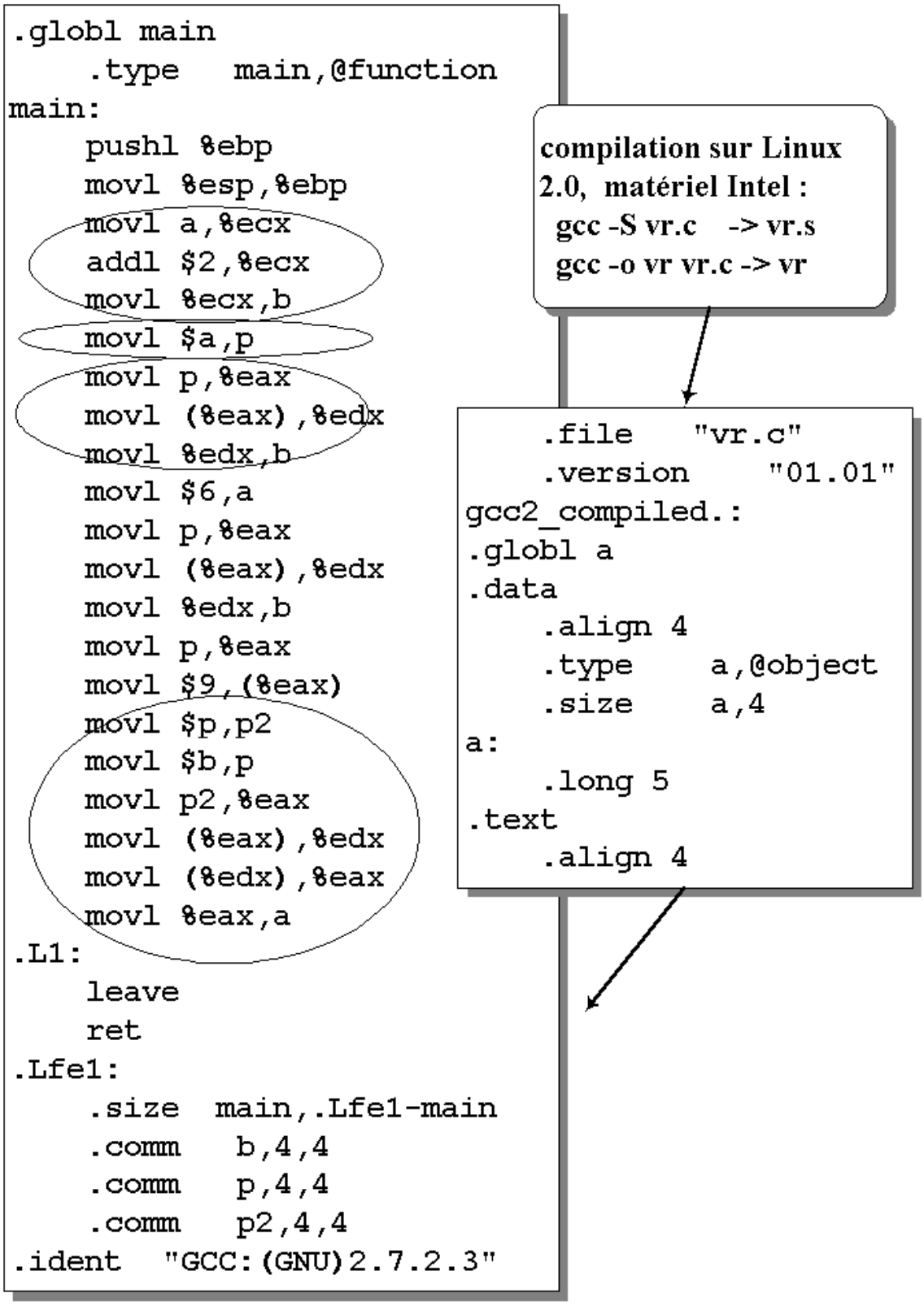


FIG. 80 – Compilation (15/24)

```
/* chaine.c -- chaines de caractères */
/*   et réservation de place mémoire */
#include <stdio.h>
#include <string.h>
extern int errno;

main()
{       int n,i;
        char *ch1="Bonjour la compagnie!";
        char *ch2;
        n = sizeof (ch1);
        i = sizeof (ch2);
        printf("n= %d, i= %d\n",n,i);
        strcpy (ch2,ch1); /* ch2<-ch1 */
        printf("ch2=%s\n",ch2);
}
```

```
938 vayssade kappa> cc -o chaine chaine.c
939 vayssade kappa> chaine
n= 8, i= 8
Segmentation fault (core dumped)
```

FIG. 81 – Compilation (16/24)

```
/* chain2.c -- chaines de caractères */
/*   et réservation de place mémoire */
#include <stdio.h>
#include <string.h>
extern int errno;

main()
{       int n,i;
        char *ch1="Bonjour la compagnie!";
        char *ch2;
        n = sizeof (ch1);
        i = sizeof (ch2);
        printf("n= %d, i= %d\n",n,i);
        ch2 = (char *) malloc (strlen(ch1));
        if (ch2!=NULL) strcpy (ch2,ch1);
        printf("ch2=%s\n",ch2);
}
```

```
943 vayssade kappa> cc -o chain2 chain2.c
944 vayssade kappa> chain2
n= 8, i= 8
ch2=Bonjour la compagnie!
```

FIG. 82 – Compilation (17/24)

```
/* tab1.c -- tableaux et pointeurs */
#include <stdio.h>
main()
{   int n,i;
    int tab[10];
    int chf[10];
    n = sizeof (tab); i = sizeof (chf);
    printf("n= %d, i= %d\n",n,i);
    for(i=0;i<=10;i++) chf[i]=i;

    printf("tab[0]=%d,tab[1]=%d,\
        tab[9]=%d,tab[10]=%d\n",\
        tab[0],tab[1],tab[9],tab[10]);
    printf("chf[0]=%d,chf[1]=%d,\
        chf[9]=%d,chf[10]=%d\n",\
        chf[0],chf[1],chf[9],chf[10]);

        for (i=1;i<=10;i++) tab[i] = chf[i];

    printf("tab[0]=%d,tab[1]=%d,\
        tab[9]=%d,tab[10]=%d\n",\
        tab[0],tab[1],tab[9],tab[10]);
    printf("chf[0]=%d,chf[1]=%d,\
        chf[9]=%d,chf[10]=%d\n",\
        chf[0],chf[1],chf[9],chf[10]);
}
```

```
952 vayssade kappa> cc -o tab1 tab1.c
953 vayssade kappa> tab1
n= 40, i= 44
tab[0]=-1,tab[1]=536876188,tab[9]=-1073729152,tab[10]=1023
chf[0]=0,chf[1]=1,chf[9]=9,chf[10]=10
tab[0]=-1,tab[1]=1,tab[9]=9,tab[10]=10
chf[0]=0,chf[1]=1,chf[9]=9,chf[10]=10
954 vayssade kappa>
```

FIG. 83 – Compilation (18/24)

```
/* tab2.c -- tableaux et pointeurs */
#include <stdio.h>
main()
{   int n,i;
    int taa[10];
    int tab[10];
    n = sizeof (taa); i = sizeof (tab);
    printf("n= %d, i= %d\n",n,i);
    for(i=0;i<=10;i++) tab[i]=i;

    printf("taa[0]=%d,taa[1]=%d,\
        taa[9]=%d,taa[10]=%d\n",\
        taa[0],taa[1],taa[9],taa[10]);
    printf("tab[0]=%d,tab[1]=%d,\
        tab[9]=%d,tab[10]=%d\n",\
        tab[0],tab[1],tab[9],tab[10]);

        for (i=1;i<=10;i++) taa[i] = tab[i];

    printf("taa[0]=%d,taa[1]=%d,\
        taa[9]=%d,taa[10]=%d\n",\
        taa[0],taa[1],taa[9],taa[10]);
    printf("tab[0]=%d,tab[1]=%d,\
        tab[9]=%d,tab[10]=%d\n",\
        tab[0],tab[1],tab[9],tab[10]);
}
```

FIG. 84 – Compilation (19/24)

```
959 vayssade kappa> cc -o tab2 tab2.c
960 vayssade kappa> tab2
n= 40, i= 40
taa[0]=10,taa[1]=-1,taa[9]=1023,taa[10]=536875212
tab[0]=0,tab[1]=1,tab[9]=9,tab[10]=10
taa[0]=10,taa[1]=1,taa[9]=9,taa[10]=10
tab[0]=0,tab[1]=1,tab[9]=9,tab[10]=10
Segmentation fault (core dumped)
961 vayssade kappa>

              for (i=0;i<10;i++) taa[i] = tab[i];

967 vayssade kappa> cc -o tab2 tab2.c
968 vayssade kappa> tab2
n= 40, i= 40
taa[0]=10,taa[1]=-1,taa[9]=1023,taa[10]=536875212
tab[0]=0,tab[1]=1,tab[9]=9,tab[10]=10
taa[0]=0,taa[1]=1,taa[9]=9,taa[10]=536875212
tab[0]=0,tab[1]=1,tab[9]=9,tab[10]=10
969 vayssade kappa>  ls -l core

-rw-------   1 vayssade si         376832 Feb 16 23:20 core

970 vayssade kappa> rm core
```

```
tab1 et tab2 sont identiques à ceci près que les noms des
tableaux "tab" et "chf" deviennent "taa" et "tab".

ce changement de nom provoque un changement de l'ordre
d'affectation en mémoire.

ce changement d'affectation fait apparaitre l'erreur de
programmation présente dans tab1 mais qui restait masquée,
et qui cette fois provoque un plantage du programme.

dans la deuxième exécution de tab2 on a corrigé l'erreur
dans la boucle for. Le programme ne plante plus et taa[0] a
désormais la bonne valeur.
```

FIG. 85 – Compilation (20/24)

```
/* tableaux et pointeurs */
```

```
float *ptab;     /* un pointeur */
                 /* initialisé à 0 */

float tab[20];   /* un tableau */
        /* "tab" est une constante */
tab[0] = 23; tab[1] = 45;

ptab = tab;      /* (1) */

ptab = &tab[0]; /* (2) identique à (1) */

ptab = &tab[10]; /* (3) */

tab = &tab[10]; /* (4) interdit */
```

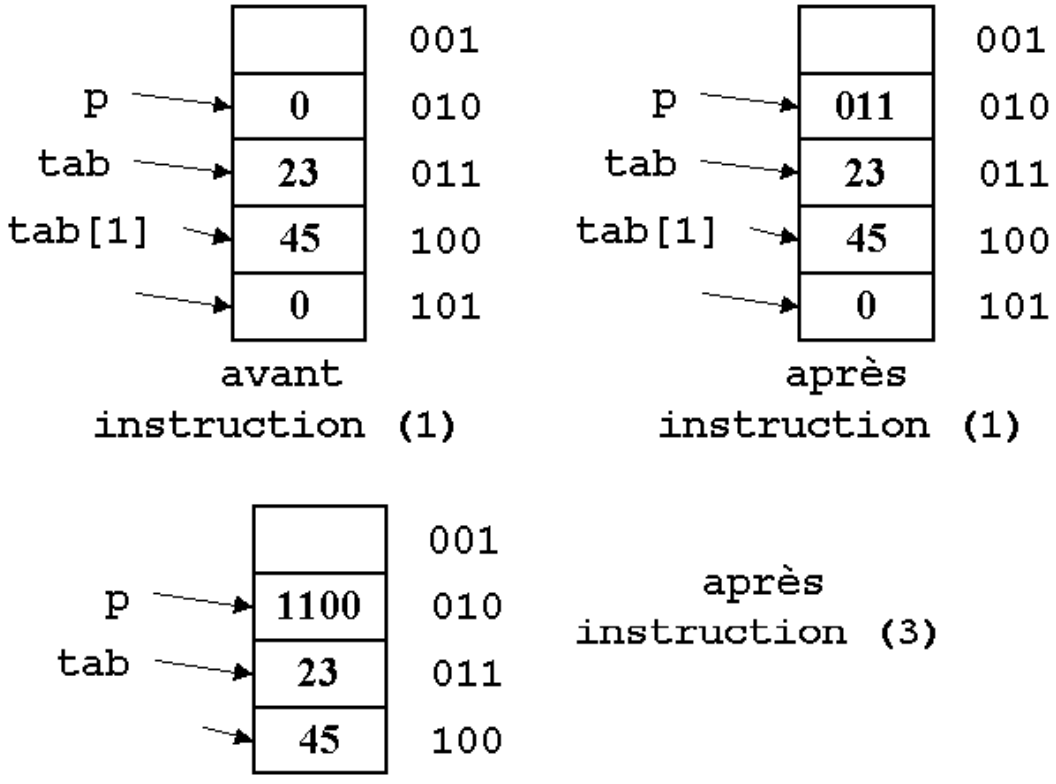


FIG. 86 – Compilation (21/24)

```
/* tableaux et pointeurs */
```

Un pointeur est une variable à part entière :
on peut :

- **l'affecter : pter =**
- **la comparer : if (pter == ...)**
- **l'incrémenter : pter = pter++**
- **etc ...**

Le nom "tab" est une adresse constante, fixée à la compilation et non modifiable par programme.

Toutes les opérations sur les tableaux sont compilées en opérations de l'arithmétique des pointeurs.

l'arithmétique des pointeurs tiens compte de la taille des objets pointés :

```
float *pf ; /* pointeur sur un float */
```

si sizeof (float) est égal à 4 (machine 32 bits) alors pf++ incrémente pf de 4

Plus généralement :

a[i] est compilé comme *(a+i)
soit a + i * sizeof (a[0])

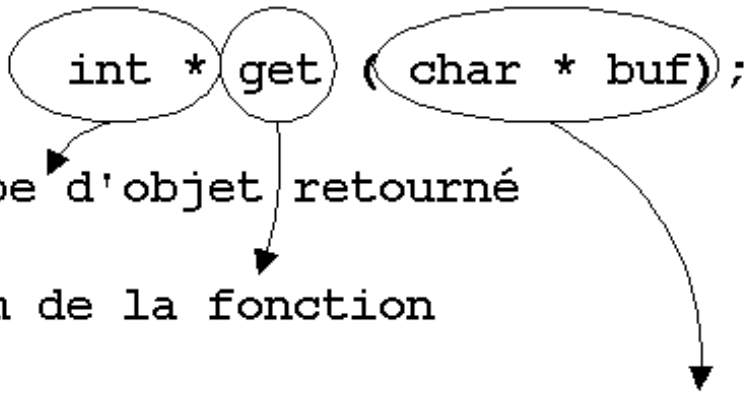
et &a[i] est identique à a+i (au sens des pters).

FIG. 87 – Compilation (22/24)

```
/* tableaux et pointeurs */
```

Passer un tableau en argument d'un appel de fonction est équivalent à passer un pointeur.

Soit la fonction dont le prototype est le suivant :




```
int * get ( char * buf);
/* get retourne une chaine dans
 * "buf" et la longueur de celle-ci
 * comme valeur de fonction
 */
char buffer[80];
int *n;
char *  ptr , p1 , p2;

n = get (buffer); /* oui */

p1 = buffer;
n = get(p1);     /* oui */

n = get(ptr);    /* NON ! */

p2 = (char *)malloc(80);
if (p2!=NULL) n=get(p2); /* oui */
```

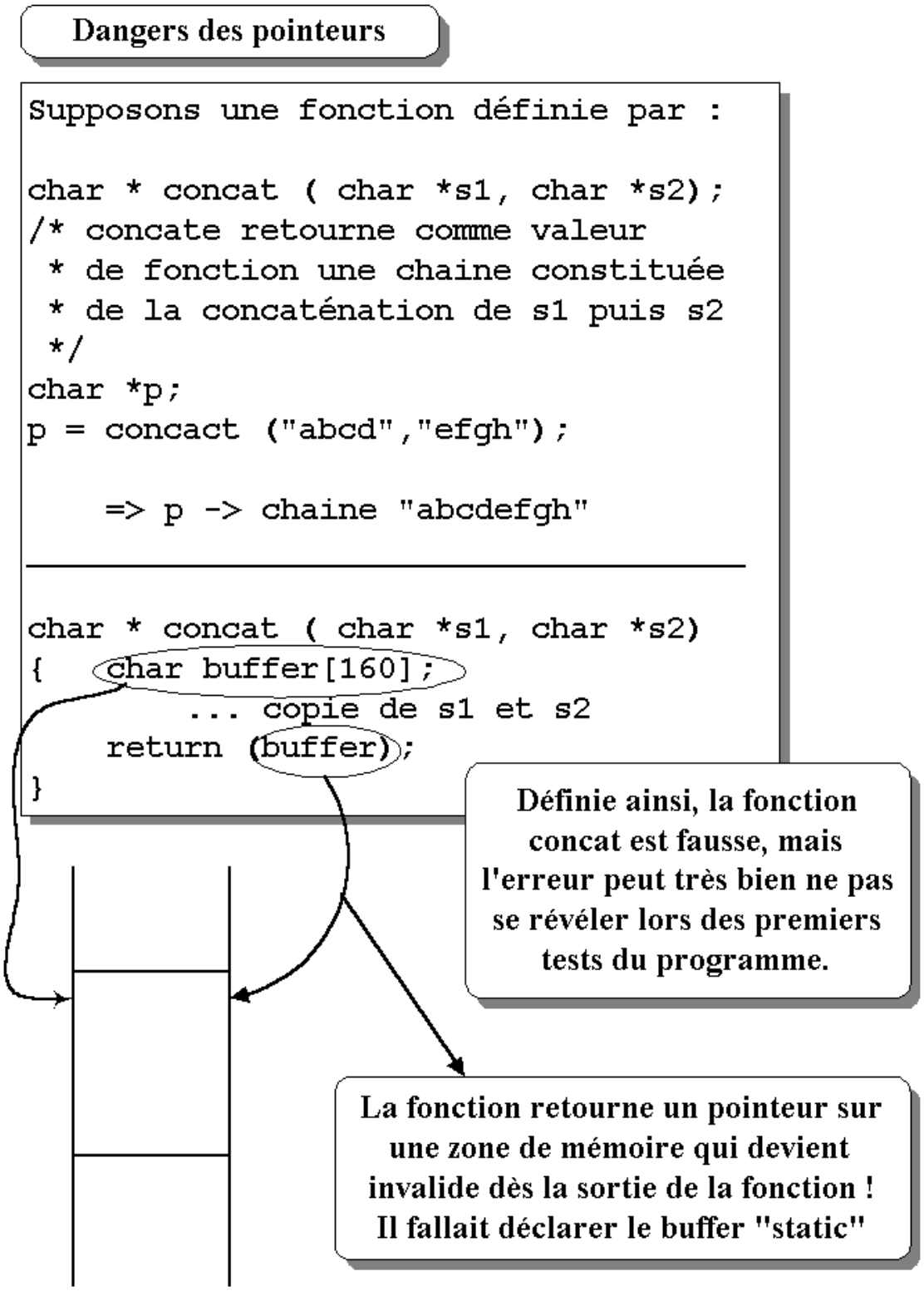


FIG. 89 – Compilation (24/24)

8.4 Construction d'un programme exécutable (Fig. 90 à 101)

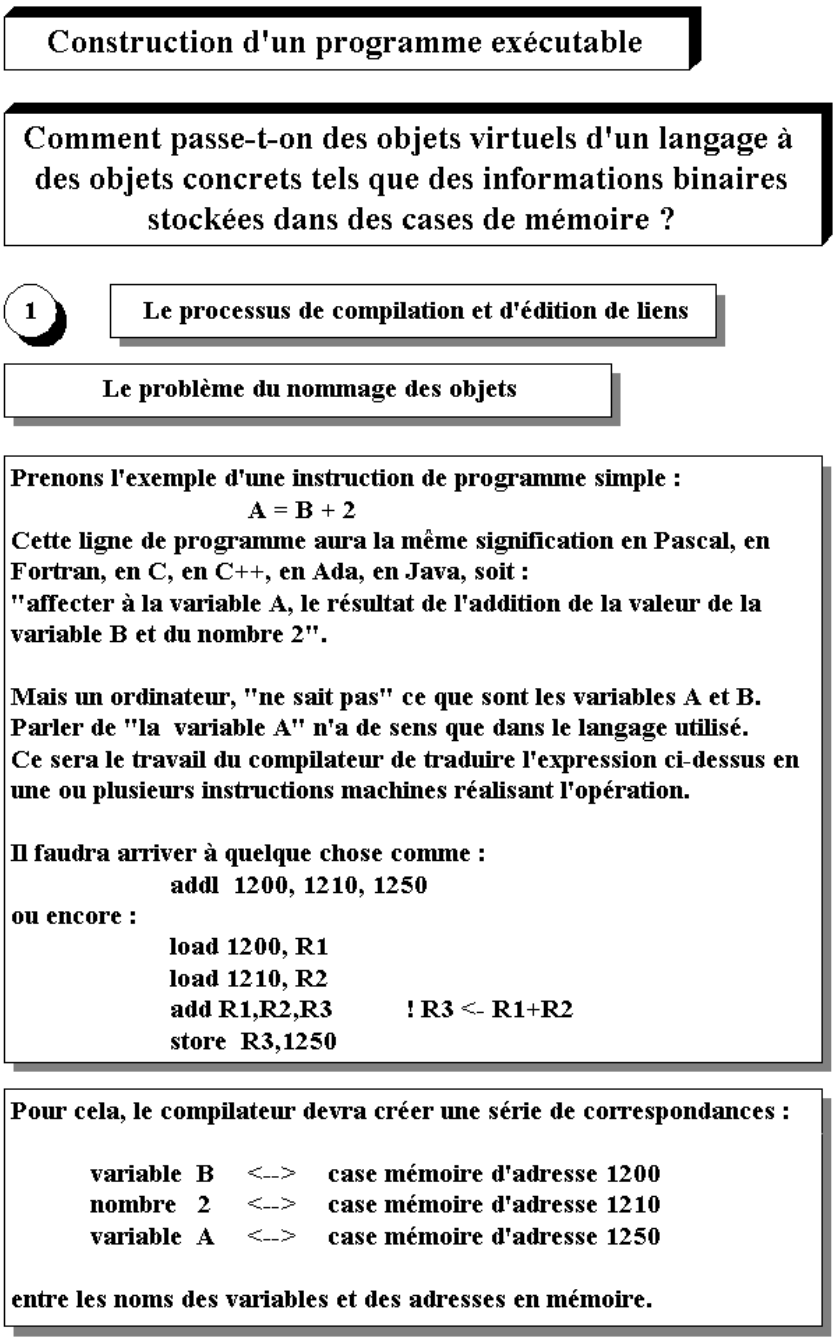


FIG. 90 – Construction d'un exécutable(1/24)

Remarque : la plupart des langages permettent d'utiliser plusieurs fois le même nom de variable, pourvu que ce soit dans des contextes différents : des blocs différents, des fonctions différentes, etc ...
Dans ce cas, le compilateur commence par donner à chaque variable un nom unique en préfixant son nom par celui de la fonction dans laquelle elle est déclarée ou en utilisant un identifiant interne unique.
Ainsi la variable "i" de la fonction "somme" et la variable "i" de la fonction "moyenne" ne sont pas confondues et le compilateur leur fait correspondre des cases mémoire différentes.

compilation = traduction + affectation d'adresses

Le compilateur va traduire les instructions du langage (C, C++, Java) en une suite d'instructions assembleur.
À chaque variable rencontrée dans le texte du programme à compiler (le "source"), il va faire correspondre une réservation de la place mémoire nécessaire et construire une table (table des symboles) permettant de faire les liens :

<nom de variable> <--> <adresse en mémoire>

Ces tables seront utilisées par l'éditeur de liens pour construire le module exécutable, et éventuellement par le débugger pour permettre une utilisation des noms symboliques lors d'une session de mise au point interactive.

édition de liens ("link") = recoller les morceaux

Tous les langages modernes (C, C++, Java,...) sont modulaires.
C'est-à-dire qu'il peuvent être découpés en modules sources indépendants, chaque module source étant compilé séparément.
Pour chaque source compilé, le compilateur va créer un texte en assembleur accompagné de sa table des symboles (noms <-> adresses).

Ce sera le travail d'un autre composant du système, l'éditeur de liens, de réunir les différents modules composant un programme et de produire un seul module exécutable.

FIG. 91 – Construction d'un exécutable(2/24)

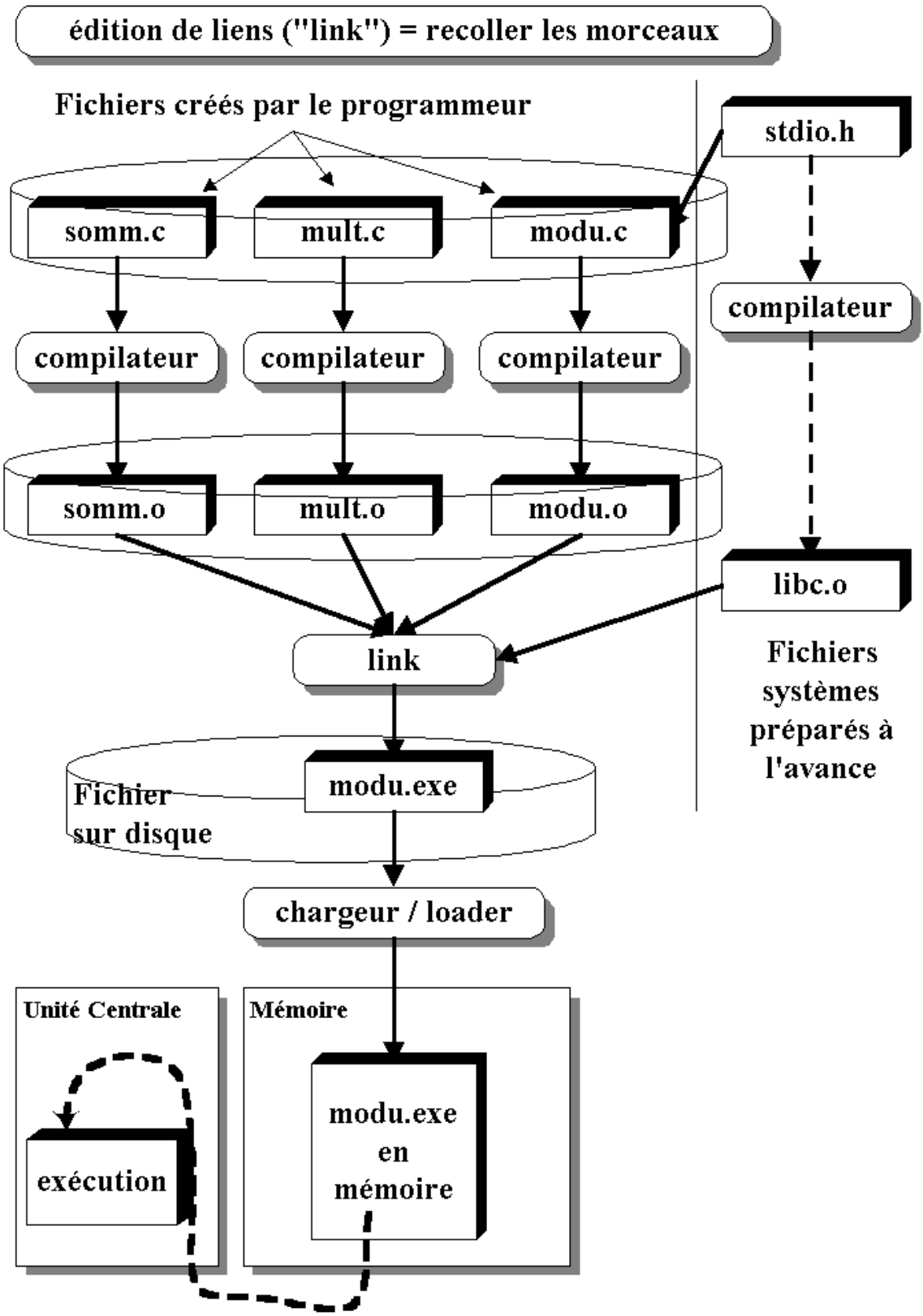


FIG. 92 – Construction d'un exécutable(3/24)

```
/* prog. modu -- fonction main */
#include <stdio.h>

main()
{    /* variables locales à main */
     int A = 3;
     int B = 5;
     int C,D;

     somm (A,B,&C);
     D = mult (A,B);
     printf ("C= %d, D=%d\n",C,D);
}
```

```
/* prog. modu -- fonction somm */

void somm (int i, int j, int *k)
{    /* variables locales à somm */
     int A , B , C ;
     A = i ;
     B = j ;
     C = A + B ;
     *k = C ;
}
```

```
/* prog. modu -- fonction mult */

int mult (int i, int j)
{    /* variables locales à mult */
     int A , B , C ;
     A = i ;
     B = j ;
     C = A * B ;
```

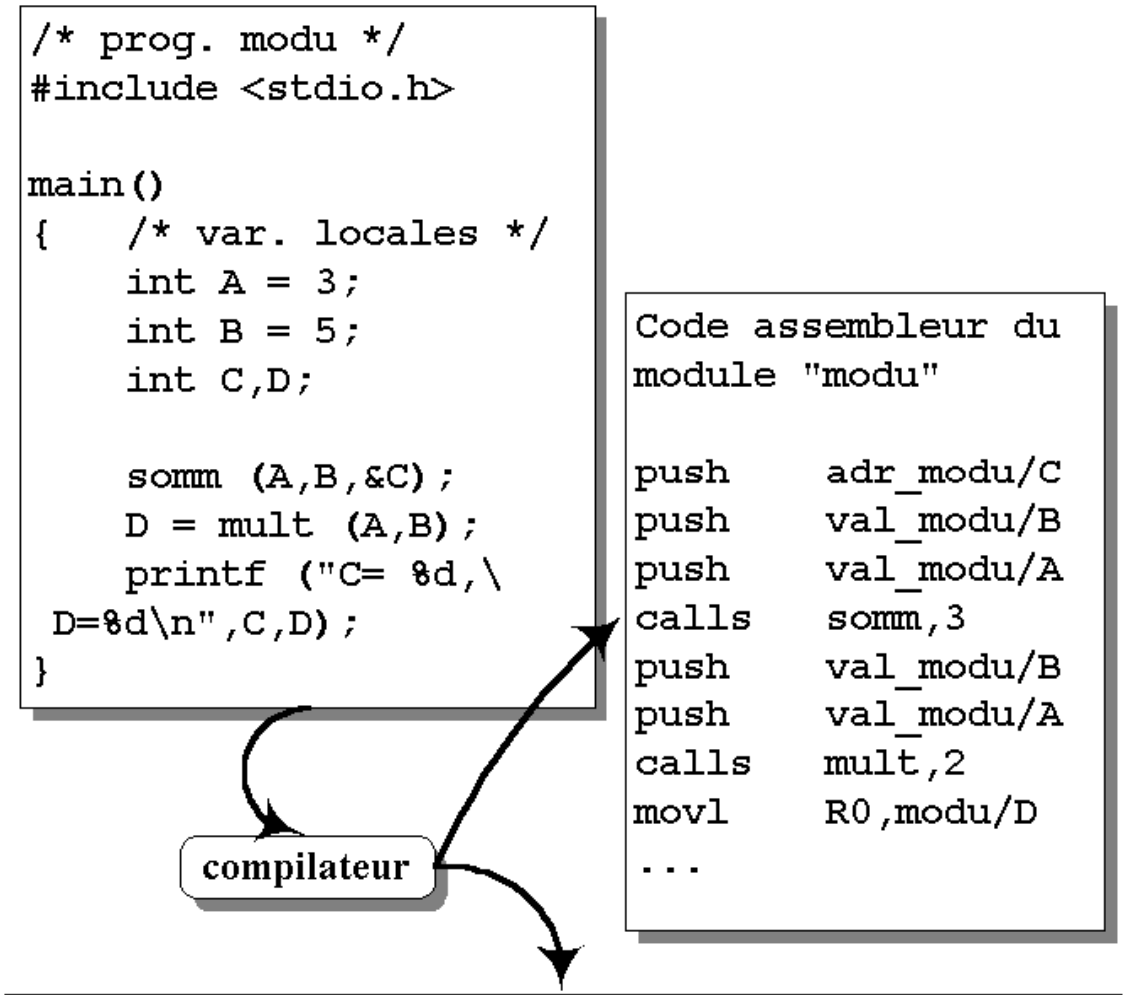


FIG. 94 – Construction d'un exécutable(5/24)

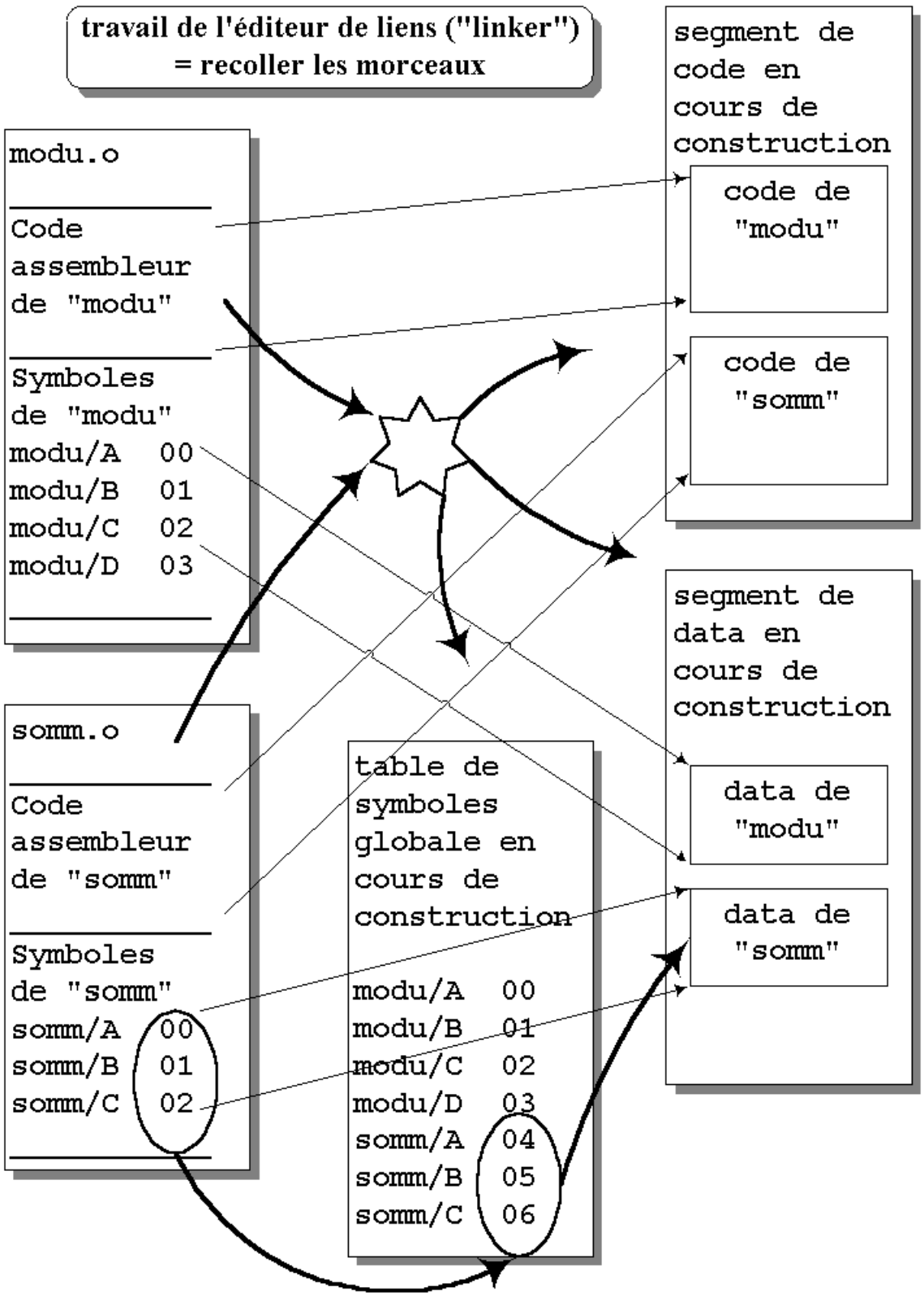


FIG. 95 – Construction d'un exécutable(6/24)

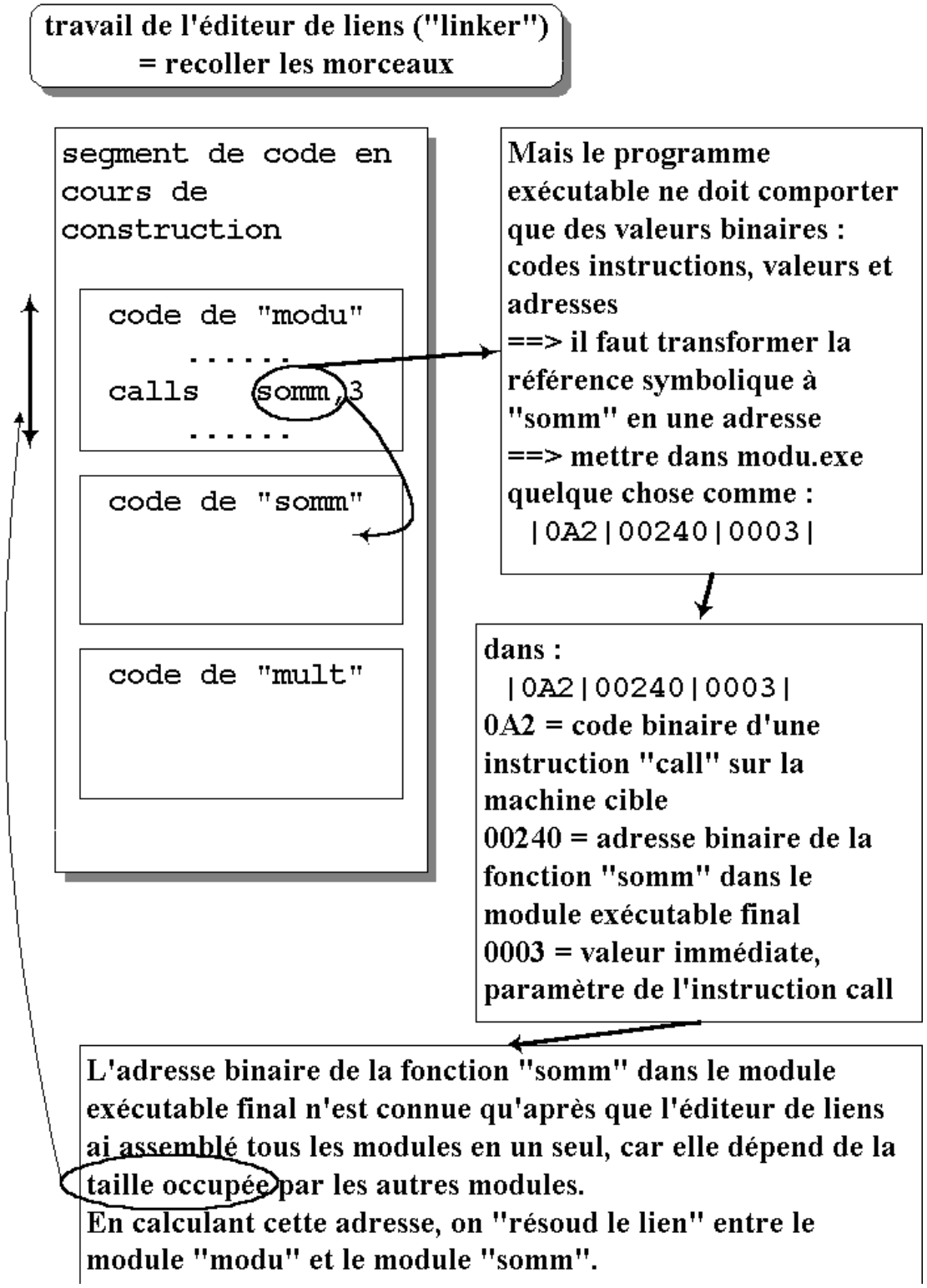


FIG. 96 – Construction d'un exécutable(7/24)

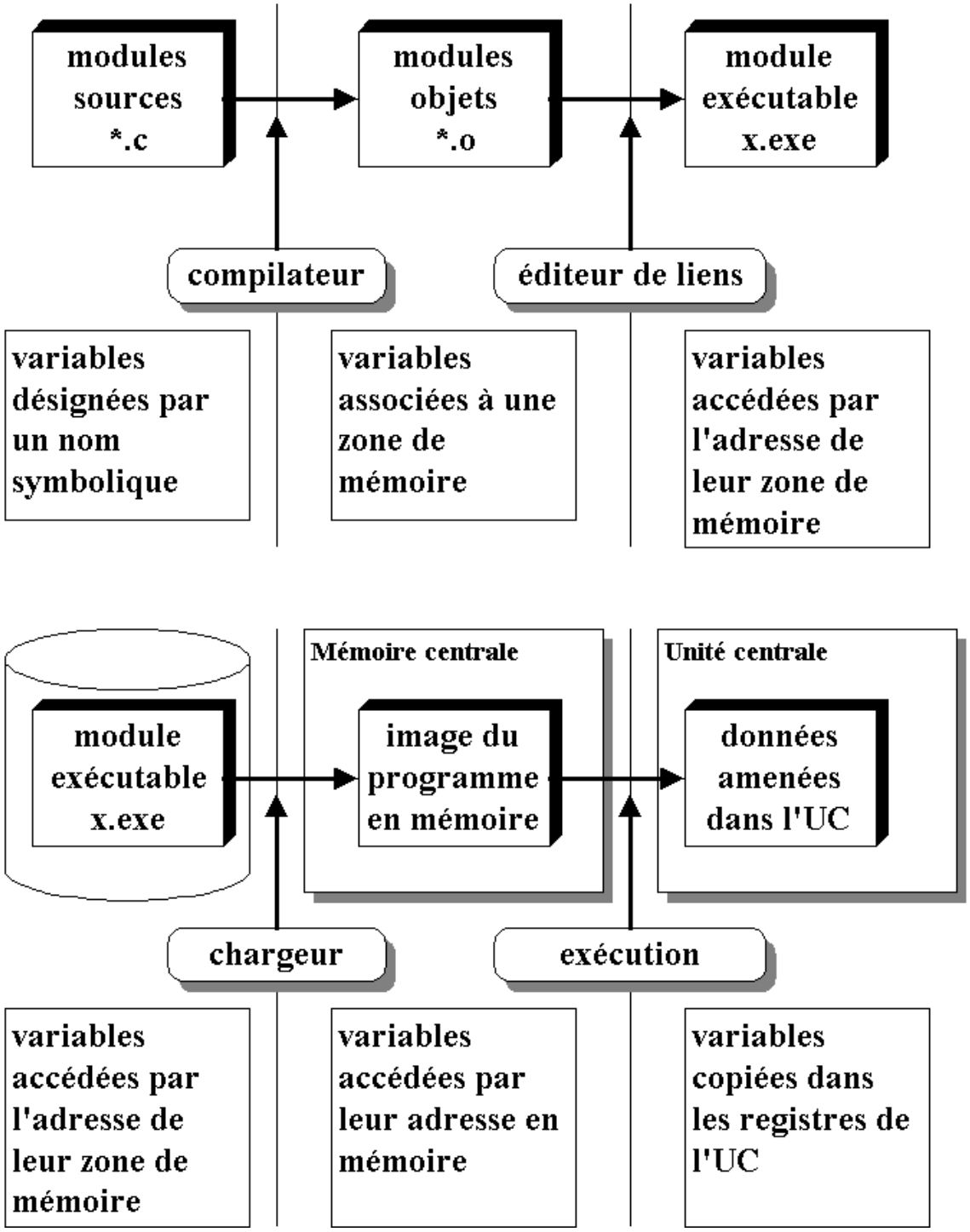


FIG. 97 – Construction d'un exécutable(8/24)

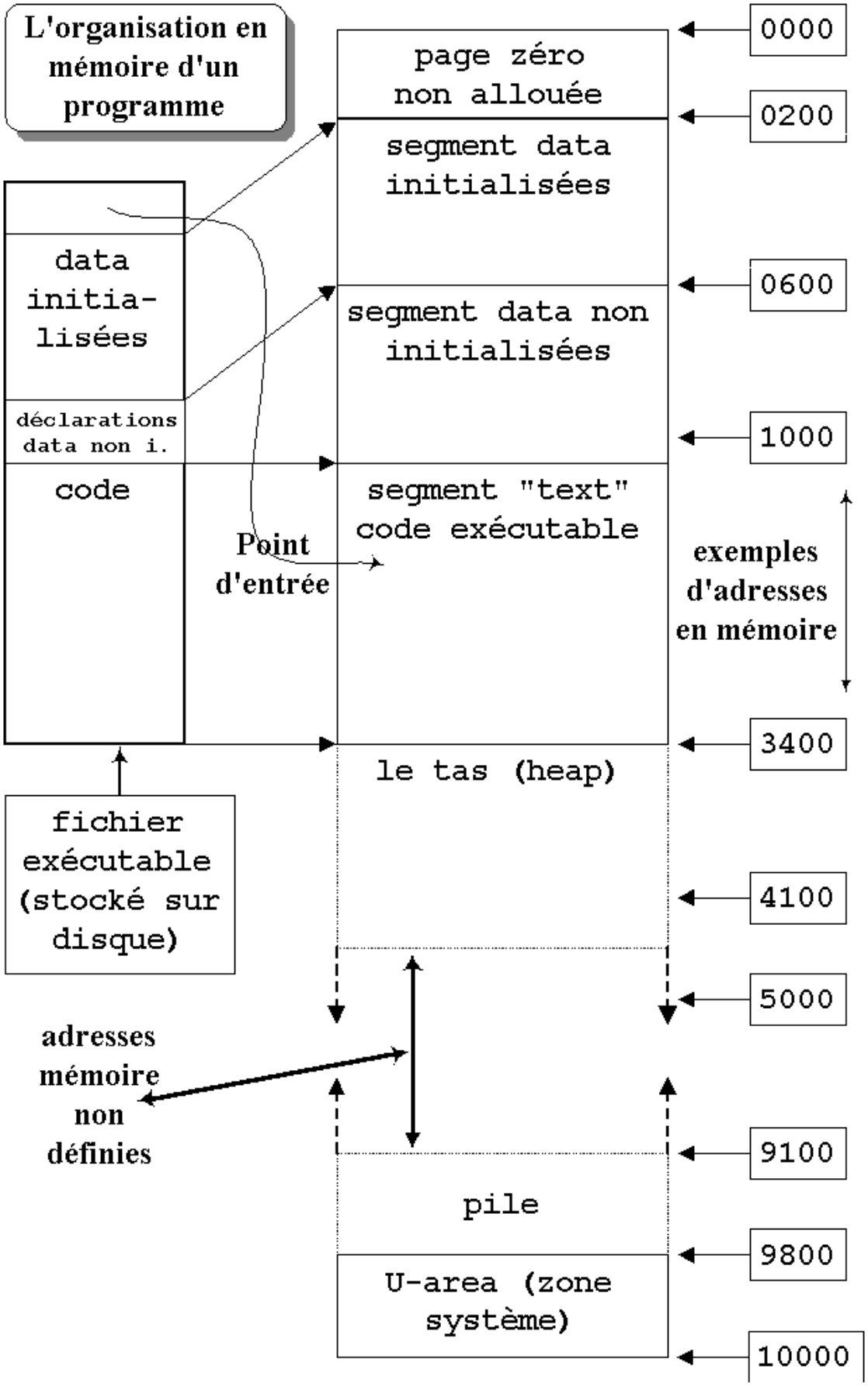


FIG. 98 – Construction d'un exécutable(9/24)

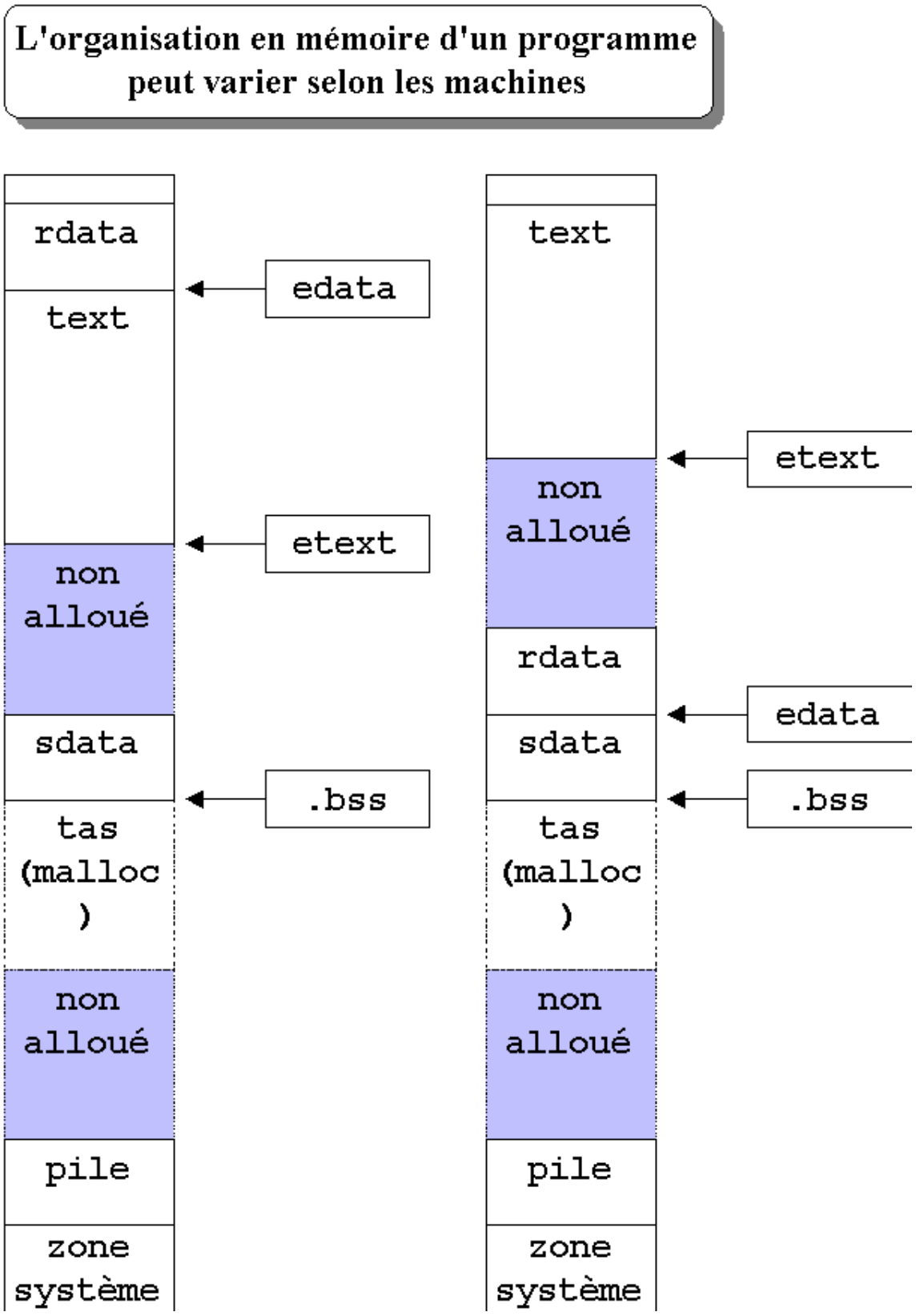


FIG. 99 – Construction d'un exécutable(10/24)

```
/* ev1.c -- esapce virtuel d'un programme */

#include <stdio.h>
extern int errno;
extern int etext, edata, end;

main()
{ int n,fd,i;
  double *x, x1;
  printf("\n etext= %d %x",&etext,&etext);
  printf("\n edata= %d %x",&edata,&edata);
  printf("\n end  = %d %x",&end,&end);
  n = &edata - &etext;
  printf("\n edata-etext= size init data=\
         %d %x",n,n);
  n = &end - &edata;
  printf("\n end-edata= size uninit data=\
         %d %x",n,n);
  i = sbrk(4);
  printf("\n sbrk limite= %d %x", i, i);
  x =  malloc ( 1000000 * sizeof(x1) );
  printf("\n apres malloc end  = \
        %d %x",&end,&end);
  n = &end - &edata;
  printf("\n end-edata= size uninit data= \
        %d %x",n,n);
  i = sbrk(4);
  printf("\n sbrk limite= %d %x", i, i);
  printf("\n");
}
```

FIG. 100 – Construction d'un exécutable(11/24)

```
904 vayssade kappa> ev1
 etext= 536876544 20001600
 edata= 1073742480 40000290
 end  = 1073742512 400002b0
 edata-etext= size init data= 134216484 7fffb24
 end-edata= size uninit data= 8 8
 sbrk limite= 1073799856 4000e2b0
 apres malloc end  = 1073742512 400002b0
 end-edata= size uninit data= 8 8
 sbrk limite= 1081803448 407b02b8
905 vayssade kappa>
905 vayssade kappa> size ev1
   text    data    bss    dec   hex filename
   4424     656     32   5112  13f8 ev1

910 vayssade kappa> size -Ax ev1
section      size           addr
.dynamic    0x1a0   0x1200004b0
.liblist     0x20   0x120000650
.rel.dyn     0x20   0x120000670
.dynstr     0x210   0x120000690
.dynsym     0x510   0x1200008a0
.hash       0x1e0   0x120000db0
.text       0x530   0x120001050
.init        0x40   0x120001580
.fini        0x40   0x1200015c0
.got        0x100   0x140000190
.data       0x110   0x140000000
.xdata       0x70   0x140000110
.pdata       0x48   0x120001000
.rconst      0x70   0x120000f90
.sdata       0x10   0x140000180
.sbss        0x20   0x140000290
.comment    0x240            0x0
Total      0x1638
```

FIG. 101 – Construction d'un exécutable(12/24)

8.5 Bibliothèques statiques (.a) et dynamiques (.so)

```
/* prog. modu -- fonction main */
#include <stdio.h>

main()
{ /* variables locales à main */
int A = 3;
int B = 5;
int C,D;

somm (A,B,&C);
D = mult (A,B);
printf ("C= %d, D=%d\n",C,D);
}
```

```
/* prog. modu -- fonction somm */

void somm (int i, int j, int *k)
{ /* variables locales à somm */
int A , B , C ;
A = i ;
B = j ;
C = A + B ;
*k = C ;
}
```

```
/* prog. modu -- fonction mult */

int mult (int i, int j)
{ /* variables locales à mult */
int A , B , C ;
A = i ;
B = j ;
C = A * B ;
return C ;
}
```

```
Compilation statique "standard"
===============================
$ gcc -o modu modu.c somm.c mult.c
$ ll
total 28
-rwxr-xr-x  1 vayssade users   14055 Dec  2 13:13 modu
-rwxr-----  1 vayssade users  210 Dec  2 13:13 modu.c
-rwxr-----  1 vayssade users  166 Dec  2 13:13 mult.c
-rwxr-----  1 vayssade users  173 Dec  2 13:13 somm.c
```

```
$ ./modu
C= 8, D=15
-----------------------------------------------------------------

Compilation et mise en bibliothèque objet (.a)
==============================================

$ gcc -c mult.c
$ gcc -c somm.c
$ ll
total 24
-rwxr-----    1 vayssade users         210 Dec  2 13:13 modu.c
-rwxr-----    1 vayssade users         166 Dec  2 13:13 mult.c
-rw-r--r--    1 vayssade users         756 Dec  2 13:16 mult.o
-rwxr-----    1 vayssade users         173 Dec  2 13:13 somm.c
-rw-r--r--    1 vayssade users         760 Dec  2 13:17 somm.o

! créer la nouvelle archive modu.a et y mettre mult.o
$ ar -cr modu.a mult.o

! ajouter (ou remplacer) somm.o dans l'archive modu.a
$ ar -r modu.a somm.o

! lister le contenu de l'archive
$ ar -t modu.a
mult.o
somm.o

$ ar -tv modu.a
rw-r--r-- 100/100    756 Dec  2 13:16 2003 mult.o
rw-r--r-- 100/100    760 Dec  2 13:17 2003 somm.o

Utilisation de l'archive :

$ gcc -o modu modu.c modu.a
$ ll
-rwxr-xr-x  1 vayssade users  14055 Dec  2 13:26 modu
-rw-r--r--  1 vayssade users   1726 Dec  2 13:24 modu.a
-rwxr-----  1 vayssade users    210 Dec  2 13:13 modu.c
-rwxr-----  1 vayssade users    166 Dec  2 13:13 mult.c
-rwxr-----  1 vayssade users    173 Dec  2 13:13 somm.c

$ nm modu.a ! liste des symboles définis dans les objets

mult.o:
00000000 t gcc2_compiled.
00000000 T mult

somm.o:
```

```
00000000 t gcc2_compiled.
00000000 T somm

-----------------------------------------------------------------

Compilation et mise en bibliothèque dynamique (.so)
===================================================

$ gcc -fPIC -c mult.c !!! PIC obligatoire
$ gcc -fPIC -c somm.c  !!! Position Independant Code
$ ld -shared -o modu.so somm.o mult.o
$ ll modu.so
-rwxr-xr-x    1 vayssade users        2171 Dec  2 14:20 modu.so

$ gcc -o modu modu.c -L. -lmodu
/usr/bin/ld: cannot find -lmodu
collect2: ld returned 1 exit status
$
$ mv modu.so libmodu.so !!! -lmodu  recherche libmodu.so
$ gcc -o modu modu.c -L. -lmodu
$ ll modu
-rwxr-xr-x    1 vayssade users    14037 Dec  2 14:23 modu
$ ./modu
C= 8, D=15
$ ldd modu
        libmodu.so => ./libmodu.so (0x40018000)
        libc.so.6 => /lib/i686/libc.so.6 (0x40029000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)

$ nm --dynamic libmodu.so
000012f8 A _DYNAMIC
000012ec A _GLOBAL_OFFSET_TABLE_
00001328 A __bss_start
00001328 A _edata
00001328 A _end
000002c8 T mult
000002a0 T somm

-----------------------------------------------------------------

Passage par un .a
=================

-rw-r--r--  1 vayssade users     1726 Dec  2 14:40 libmodu.a

$ ld -shared -o modu.so --whole-archive -L. -lmodu
$ ll modu.so
-rwxr-xr-x  1 vayssade users     2171 Dec  2 15:05 modu.so
```

```
This is normally used to turn an archive file into a  shared
library, forcing every object to be included in the resulting
shared library.

$ mv modu.a libmodu.a
$ mv modu.so libmodu.so
$ ll libmodu.*
-rw-r--r--  1 vayssade users  1726 Dec  2 14:40 libmodu.a
-rwxr-xr-x  1 vayssade users     2171 Dec  2 15:05 libmodu.so
$ gcc -o modu modu.c -L. -lmodu
$ ll modu
-rwxr-xr-x    1 vayssade users       14037 Dec  2 15:07 modu
$ ldd modu
        libmodu.so => ./libmodu.so (0x40018000)
        libc.so.6 => /lib/i686/libc.so.6 (0x40029000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
-----------------------------------------------------------------

LD_LIBRARY_PATH

$ echo $LD_LIBRARY_PATH

$ gcc -o modu modu.c -L. -lmodu
$ ldd modu
        libmodu.so => not found <---!!!***
        libc.so.6 => /lib/i686/libc.so.6 (0x40027000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$ export LD_LIBRARY_PATH=.
$ ldd modu
        libmodu.so => ./libmodu.so (0x40018000)
        libc.so.6 => /lib/i686/libc.so.6 (0x40029000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
-----------------------------------------------------------------
```