# Systèmes et Réseaux : Exercices

Gérard Berthelot Laure Petrucci

14 avril 2011

# Chapitre 1

# Système

## 1.1 Linux — Appels noyau

### Exercice 1.1.1 : Lecture et écriture dans un fichier. Table des fichiers ouverts

**Question 1 :** Écrire un programme `pg1.c` comportant une fonction `main` qui appelle une une fonction `lire()`. Celle-ci tente de lire en une seule fois 100 caractères en utilisant l'appel noyau `read()` sur l'entrée standard (c'est-à-dire le clavier). Elle affiche sur la sortie erreur standard (l'écran) le nombre de caractères effectivement lus, puis à l'aide de l'appel noyau `write` les affiche sur la sortie standard (aussi l'écran). Remarque : on peut fermer l'entrée standard par un `CTRL-D`

**Question 2 :** Récrire le même programme en utilisant les fonctions de la bibliothèque standard `fread` et `fwrite`. Quelle différence y-a-t-il avec les appels noyau ? Peux-t-on les interchangeri ?

**Question 3 :** Pourrait-on utiliser la fonction `printf(%s,tab)` pour afficher les caractères lus ?

### Exercice 1.1.2 : redirection de fichier et recouvrement de programme

**Question 1 :** Écrire un programme `pg2.c` , qui, en utilisant l'appel noyau `execlp()`, lance l'exécution de `pg1` (sans modifier ce dernier) de telle façon que `pg1` lise sur le fichier `./test.data` au lieu du clavier et écrive dans le fichier `./res.data` plutôt que sur l'écran.Indication : avant d'exécuter `pg1` rediriger l'entrée et la sortie standard sur les fichiers `./test.data` et `./res.data` préalablement ouverts en utilisant l'appel noyau `open`.

**Question 2 :** Copier la fonction `lire()` seule dans un fichier `lire.c` puis le compiler par la commande `cc -c lire.c -o lire.o`. On peut appeler la fonc-

tion `lire()` depuis un programme `pg3.c` à condition de faire au préalable une édition des liens par la commande `cc pg3.c lire.o -o pg3`. Le programme `pg3` doit appeler la fonction `lire()` de telle sorte qu'elle lise en fait sur le fichier `./test.data` et écrive dans le fichier `./res.data`. On souhaite également pouvoir à nouveau utiliser le clavier et l'écran après l'exécution de `lire()`. Indication : il ne faut pas modifier le programme de la fonction `lire()` mais rediriger l'entrée et la sortie standard après avoir sauvegardé la possibilité d'accéder au clavier et à l'écran en utilisant l'appel noyau `dup`.

### Exercice 1.1.3 : Partage de fichier (utilisations simultanées)

NOTA BENE : pour conserver un minimum de lisibilité tous les programmes utilisant l'appel système `fork()` devront être structurés comme suit ou de façon similaire.

```
int pid;
switch (pid = fork()){
case -1 : perror("echec fork");exit(1);break;
case 0 : fils(); break
default : pere();
}
```

où `pere()` et `fils()` sont des fonctions qui se terminent par `exit()`.

**Question 1 :** Écrire un programme `pg4.c`, qui commence par un appel de `fork()` pour dupliquer le processus initial (le père) de manière à obtenir un processus fils. Le père et le fils ont ensuite le même comportement : ils ouvrent le fichier `./test.data` puis y lisent 10 caractères et les affichent sur la sortie standard (c'est-à-dire l'écran) précédés sur la même ligne de leur identité ("pere" ou "fils") et de leur numéro de processus (PID).

**Question 2 :** Modifier votre programme pour que le père ouvre le fichier `./test.data` avant d'exécuter le `fork()`. Le père et le fils lisent ensuite 10 charactères chacun dans ce fichier et les affichent sur la sortie standard, précédés sur la même ligne de leur identité et de leur PID. Répéter l'exécution plusieurs fois et déterminer quels sont les caractères lus par le processus père et ceux lus par le processus fils. Comparer avec les résultats de la question précédente.

**Question 3 :** Dans le père, positionner le pointeur de lecture sur le 30ème caractère par un appel de `lseek()` avant le `fork()`. Quelle est alors la position des caractères lus par le fils ? Comparer les résultats obtenus aux questions précédentes.

### Exercice 1.1.4 : Communication entre processus par tubes et signaux

**Question 1 :** Écrire un programme qui crée 2 fils. Le processus père et les processus fils devront afficher chacun sur la même ligne leur numéro de processus, le numéro de leur père et la valeur renvoyée par `fork`. Exécutez le programme plusieurs fois. Les valeurs affichées et l'ordre d'affichage sont-ils toujours les

mêmes ? Que signifie le fait qu'un processus ait pour père le processus numéro 1 ?

**Question 2 :** Le processus père doit lire des nombres un par un au clavier et les transmettre au fur et à mesure à ses fils à l'aide de deux tubes. Les nombres impairs sont transmis au premier fils et les nombres pairs au second (on transmettra la représentation binaire de chaque entier, et non sa représentation ascii dont la longueur est variable). Chaque fils lit dans le tube correspondant les nombres et pour chacun affiche sur une seule ligne son identité son pid et la valeur lue. La valeur 0 indique la fin des entrées. Le père la transmet à ses deux fils puis attend la mort de chacun d'eux, affiche le pid du mort, s'il a été tué par un signal ou non et se termine après la mort des deux. A réception de la valeur 0 chacun des deux fils se termine après un délai de 5s. Avant de se terminer chacun des trois processus affiche sur une seule ligne son identité son pid et celui de son père.

**Question 3 :** Lorsque l'utilisateur rentre la valeur 0, le processus père ne la transmet pas a ses fils mais se contente de fermer les tubes en écriture. Chacun des fils doivent se rendre compte qu'il n'y aura plus rien à lire se et se terminer d'eux mêmes.

**Question 4 :** On va mesurer la taille maximum d'un tube. Pour ce faire lorsque l'utilisateur rentre la valeur 5, le processus père ne la transmet pas au fils 1 mais lui envoie un signal `SIGUSR1` pour le faire cesser de lire dans le tube. Ce dernier doit alors renvoyer à son pére un signal `SIGUSR2` pour accuser réception puis se mettre en pause. A réception du signal `SIGUSR2` le père doit essayer d'ajouter indéfinement un caractère `'B'` dans le tube 1 jusqu'à réception du signal `SIGPIPE`. Avant chaque tentative il doit afficher le nombre de `'B'` qu'il a réussit à écrire.

**Question 5 :** Que se passe-t-il si le fils 1 se termine au lieu de se mettre en pause ?. Indiquer quel est le signal reçu par le père.

## Exercice 1.1.5 : réalisation d'un petit shell

Écrire un programme en `C` qui propose à l'utilisateur le menu suivant jusqu'à ce que *quitter* soit choisi :

```
1/ ls -l
2/ ps
3/ traduction de la date en francais
4/ quitter
```

### Exercice 1.1.6 : Utilisation de threads

Un `thread` est une activité exécutée de façon autonome à l'intérieur de l'espace d'adresses virtuelles d'un processus. Le processus, aussi appelé thread initial, partage avec tous les threads créés en son sein, toutes les déclarations gobales, la table des fichiers ouverts et les réactions aux signaux. En revanche les dispositions de masquage des signaux ne sont pas partagées. Un thread peut être créé par appel de la fonction du noyau `pthread_create()` à laquelle il faut fournir l'adresse d'une variable qui contiendra l'identification du `thread` créé, des attributs (par défaut `NULL`), l'adresse d'une fonction par laquelle le thread commencera son exécution, et l'adresse d'un unique paramètre qui sera passé à cette fonction. Voir l'exemple ci-dessous dans lequel trois `threads` qui affichent indéfiniment, respectivement les caractères `'a'`, `'b'` et `'c'`, sont exécutés.

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

char tabc[]={'a','b','c'};
int n0=0,n1=1,n2=2;
pthread_t tabthread[3];

void *start ( void* ptn ){
   int i=*(int*)ptn;
   while(1)
   {
      write(1,&tabc[i],1);
   }
}

int main(int argc, char **argv){
    int ret;
    ret=pthread_create(&tabthread[0],NULL, start,&n0);
    ret+=pthread_create(&tabthread[1],NULL, start,&n1);
    ret+=pthread_create(&tabthread[2],NULL, start,&n2);
    if (ret <0){perror("pb creation threads");exit(1);}
    usleep(100000);;
    sleep(1);
    return 0;
}
```

**Question 1 :** Exécuter plusieurs fois de suite (au moins 3) ce programme. L'ordre d'apparition des caractères est-il prévisible ? Que se passe-t-il ?

**Question 2 :** Serait-il possible d'utiliser une seule variable `n` prenant successivement les valeurs 0,1,2 plutôt que trois variables `n0,n1,n2` ?

**Question 3 :** On souhaite maintenant que chaque `thread` affiche un seul caractère à tour de rôle, de façon à voir apparaître `abcabcabcabc...`. Pour ce faire on va utiliser une variable partagée `tour`. Lorsqu'elle vaut `i` le `thread i`

imprime son caractère puis ajoute 1 `modulo` 3. Lorsque `tour` a une valeur différente de son numéro le `thread` ne fait rien. Ecrire un nouveau programme et l'expérimenter. On constate que l'on affiche beaucoup moins de caractères dans le même temps. Pourquoi ?

**Question 4 :** Pour éviter la perte de temps due aux `attentes actives` on va suspendre in `thread` lorsque ce n'est pas son tour. Il sera ensuite réveillé le moment venu par l'appel noyau `pthread_kill()` pour envoyer un signal `SIGUSR1`. On pourrait utiliser pour ce faire l'appel noyau `pause()` et une fonction de traitement de signal mise en place par appel de `signal()` ou `sigaction()` mais il est plus simple d'utiliser les signaux `masqués`. Un signal peut être masqué pendant une partie ou la totalité de l'exécution d'un `processsus` ou d'un `thread` en utilisant l'appel noyau `sigprocmask()` (respectivement `pthread_sigmask()`). Si un signal survient pendant sa période de masquage, il n'est pas transmis au `processus` (respectivement au `thread`) mais mis en attente (`pending`) jusqu'au démasquage. Il peut toutefois être attendu et traité par l'appel noyau `sigwait()` (voir `man pthread_sigmask()` pour un exemple). Noter que le masque initial d'un `processus` ou d'un `thread` est une `copie` de celui de son créateur. Ecrire un programme dans lequel chaque `thread` affiche son caractère, puis réveille son successeur, puis se met en attente d'être réveillé à son tour.

**Question 5 :** On reprend la solution expérimentée à la question 1 mais on souhaite maintenant pouvoir arrêter les `threads` un par un à la demande, par exemple un premier `CTRL-C` arrêtera le premier `thread`, un second `CTRL-C` arrêtera le second `thread` et un troisième `CTRL-C` arrêtera le dernier `thread`. Cependant il est impossible de réaliser ceci directement car un signal, en l'occurence `SIGINT` résultat de `CTRL-C`, est envoyé de façon aléatoire soit au `processus` en cours d'exécution, soit à l'un quelconque de ses `threads`. Il faut donc masquer le signal `SIGINT` pour les trois `threads` de façon à ce que seul le `processus` le recoive. Ce dernier devra mettre en place une fonction de traitement qui, sur réception d'un `SIGINT`, arrêtera par l'appel noyau `pthread_cancel()` d'abord le premier `thread` puis ensuite le second et enfin le troisième.

## Exercice 1.1.7 : Jeux des spéculateurs

Un site dispose d'un stock de places pour les matchs de la prochaine Coupe du Monde de Football au Brésil. Il est prêt à les vendre 500 Euros pièce, mais comme ils pourront sans doute être revendus plusieurs milliers d'Euros chacun, il cherche à augmenter ses gains en les mettant en vente dans des tournois. Lors d'un tournoi deux (ou plus) spéculateurs devront se partager `N` billets en utilisant le programme client `client.c` ci-dessous. Par l'intermédiaire de son `client` un spéculateur peut voir (commande `'v'`) le nombre de places encore disponibles dans le tournoi ou en acheter une en payant au préalable la somme de 500 Euros (commande `'a'`). On estime que le paiement prend environ 5s. Celui qui achète la dernière place remporte le tournoi et recoit les places achetées pour le montant payé. Si un spéculateur tente un achat alors qu'il ny a plus de places libres, alors il ne reçoit aucune place et perd la somme payée. Si un spéculateur, par une commande `'v'` constate qu'il n'y a plus de place disponible, alors il ne reçoit que la moitiée des placées achetées (plus exactement la partie entière) pour le montant déjà payé. Si un spéculateur abandonne pendant le tournoi, il

perd la totalité de la somme payée et ne reçoit aucune des places achetées. On constate que le site vendeur conserve la moitiée des places du perdant, ou même la totalité dans le "meilleur" des cas.

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

int main(int argc, char **argv){
    int ret,reste=-1,fdr,fdw;
    int achat=0,engagement=0;
    char c;
    if (argc !=3){
        printf("usage : %s fifo_write fifo_read\n",argv[0]); exit(1);}
    fdw = open (argv[1],O_WRONLY);
    if (fdw<0){perror("pb ouverture fifo ecriture");exit(1);}
    fdr = open (argv[2],O_RDONLY);
    if (fdr<0){perror("pb ouverture fifo lecture");exit(1);}
    printf("objectif : acheter la derniere place (reste = 0)\n\
    moitiee des places si reste plus de place\n\
    engagement perdu si reste negatif ou abandon\n");
    printf("regle du jeu : 'v' pour voir le reste,\n\
   'a' pour payer puis  acheter\n");
    while (reste!=0){
        ret=read(0,&c,1);
        if(ret < 0){perror("pb lecture clavier");exit(1);}
        if (c == 'a'){
            engagement+=500;
            printf("paiement en cours, attendez!\n");
            usleep(5000000);
        }
        ret=write(fdw,&c,1);
        if(ret < 0){perror("pb ecriture fifo ");exit(1);}
        ret=read(fdr,&reste,sizeof reste);
        if(ret < 0){perror("pb lecture fifo");exit(1);}
        if ((reste < 1)&&(c=='v')){
            achat = achat/2;
            printf("fini! reste: %d achat : %d engagement: %d\n",
                    reste,achat,engagement);
            break;
        }
        if((reste <0)&&(c=='a')){
            achat=0;
            printf("you loose ! achat:%d engagement: %d\n",
                    achat,engagement);
```

```
            break;
        }
        if ((reste == 0)&&(c=='a')){
            achat++;
            printf("you win! reste: %d achat: %d engagement :%d \n",
                    reste, achat,engagement);
            break;
        }
        if ((reste>0)&&(c=='a')){
            achat++;
            printf("reste:%d achat : %d, engagement : %d\n",
                    reste,achat,engagement);
            continue;
        }
        if ((reste>0)&&(c=='v')) {
            printf("reste:  %d achat: %d engagement: %d\n",
                    reste, achat,engagement);
        }
    }
    pause();
    return 0;
}
```

Explications :
Le programme `client` se connecte au serveur par l'intermédiaire de deux tubes nommés (voir `man 7 fifo`). Un tube nommé s'utilise commne un `pipe` qu'il faut d'abord créer par l'appel noyau `mkfifo()` en lui donnant un nom comme un fichier normal. On peut ensuite ouvrir chacune des extrémités par un appel de `open()` comme pour un fichier normal. Attention cependant, une ouverture en `lecture` ne peut se terminer que si un autre `processus` ou un autre `thread` tente une ouverture en `écriture` ensuite, ou l'inverse. Donc s'ils utilisent deux fifos pour dialoguer (un pour chaque sens), des `processus` ou des `threads` doivent les ouvrir dans le même ordre mais de façon inverse. Ensuite le `client` reçoit les demandes du spéculateur, encaisse le paiement s'il y a lieu, et les transmet au serveur du site vendeur. Pour chaque demande transmise, il attend la réponse du serveur, comptabilise le nombre de places effectivement obtenues et la somme payée puis affiche ces informations et enfin termine le tournoi quand toutes les place sont achetées.

**Question 1 :** Ecrire le programme du serveur. Il devra se comporter "honnêtement" : après s'être connecté aux 4 fifos (deux par client), il devra répondre en donnant le nombre de places effectivement disponibles à l'instant sur réception de la commande `'v'`, décrémenter ce nombre et le renvoyer en cas de réception de la commande `'a'`.
Créer alors les 4 fifos dans le répertoire `/tmp` et leur donner les droits d'accès nécessaires pour que les utilisateurs de votre groupe puisse les lire ou y écrire. Lancer votre serveur, trouver 2 spéculateurs potentiels qui se logueront sur votre machine par `SSH` pour lancer leurs clients et empochez les gains (virtuels) à la fin du tournoi !

## 1.2 Gestion de disque

### Exercice 1.2.1 : Lecture d'une disquette MS-DOS

Il s'agit de lire le secteur de démarrage d'une disquette et d'afficher les informations suivantes : taille d'un secteur en octets, nombre de secteurs par cluster, taille d'un cluster en octets, nombre de clusters sur la disquette, volume de la disquette en MO, taille des FATs en nombre de secteurs, nombre de FATs, numéro du premier secteur de la zone des fichiers et répertoires.

Il faut ensuite lire le répertoire racine de la disquette et afficher les caractéristiques des fichiers qui s'y trouvent : nom, extension, attributs, taille.

Pour les fichiers *normaux* (ni répertoire, ni système, ni fichier caché), afficher au moins les 100 premiers octets de chaque fichier.

**Explications** Tant qu'il n'a pas pris connaissance par une opération `mount` du système de fichiers qui organise son contenu, le système d'exploitation voit un volume (disquette, disque amovible, partition d'un disque fixe) seulement comme un fichier régulier, c'est-à-dire un tableau de caractères, identifié par le nom du périphérique sur lequel il se trouve, ou assimilé. Le détail du découpage en secteurs, ainsi que la géométrie (nombre de cylindres, nombre de faces, nombre de secteurs par piste et taille des secteurs), est totalement masqué par le pilote (driver) du périphérique. Le fichier correspondant au lecteur de disquette est en général nommé `/dev/fd0`. Les fichiers correspondant aux partitions d'un disque fixe sont en général nommées `/dev/hda1`, `/dev/hda2`, etc pour le premier disque, `/dev/hdb1`, `/dev/hdb2`, etc pour le second disque, et ainsi de suite. Le contenu d'une disquette peut donc être obtenu par un appel système normal `read()` sur le fichier `/dev/hd0`.

On peut aussi écrire de la même façon sur une disquette ou un disque, mais au risque de détruire le système de fichiers qui s'y trouve. L'installation d'un système de fichiers sur un volume est faite par la commande `mkfs` sous unix ou `format` sous dos.

**Structure d'un système de fichiers de type FAT** Tout volume (physique ou logique) de type FAT est divisé en quatre zones :

- secteur de démarrage (boot sector) et secteurs réservés ;
- tables de localisation des fichiers (Files Allocation Tables – FATs) ;
- répertoire racine (root directory) ;
- fichiers et sous-répertoires.

Chacune de ces zones comporte un nombre entier de secteurs. Le secteur de démarrage contient des informations générales sur le volume et éventuellement un programme de démarrage qui peut se poursuivre dans les secteurs suivants.

Le répertoire racine contient les fichiers et sous-répertoires du premier niveau de l'arborescence.

**Structure du secteur de démarrage**

| déplac. | nb octets | rôle |
|---|---|---|
| 11 | 2 | taille du secteur en octets |
| 13 | 1 | taille cluster en secteurs |
| 14 | 2 | secteurs réservés dont boot sector |
| 16 | 1 | nb de FATS (redondance de sécurité) |
| 17 | 2 | nb entrées dans répertoire racine |
| 19 | 2 | nb de secteurs du volume |
| 22 | 2 | taille d'une FAT en secteurs |

**Structure des FATs** Une table de localisation des fichiers permet de savoir où sont les secteurs alloués à un fichier. Les secteurs sont alloués aux fichiers par ensembles de secteurs contigus appelés *clusters*. Le nombre de secteurs par cluster dépend de chaque volume. Ceci permet de diminuer la taille des FATs mais la place allouée et inutilisée en fin de fichier est plus importante en moyenne. Toutes les FATs sont identiques, l'une pouvant ainsi remplacer l'autre en cas de problème (perte d'un secteur, crash pendant une mise à jour, ...). À chaque entrée de la FAT (12 bits pour la FAT des disquettes) est associé le cluster de même rang dans le volume. Si une entrée correspond au dernier cluster du fichier alors elle contient `0xfff`, sinon elle contient le numéro d'entrée de la FAT à laquelle est associé le cluster suivant.

**Calcul du numéro de cluster associé à une entrée de la FAT** En fait les deux premières entrées d'une FAT ne sont associées à aucun secteur. Donc

$$numero\_cluster = numero\_entree\_de\_la\_FAT - 2$$

**Structure du répertoire racine** Le nombre d'entrées du répertoire racine se trouve dans le secteur de démarrage. Chacune comporte 32 octets occupés comme suit :

| déplac. | nb octets | rôle |
|---|---|---|
| 0 | 8 | nom du fichier |
| 8 | 3 | extension du nom du fichier |
| 11 | 1 | attributs du fichier |
| 26 | 2 | numéro de la première entrée dans la FAT |
| 28 | 4 | taille du fichier en octets |

Le premier caractère du nom contient `0x00` si l'entrée est vide et `0xe5` si le fichier a été supprimé.

**Attributs des fichiers** Ils sont codés sur 6 bits (vrai si 1 faux sinon) :

**bit 1 :** lecture seulement

**bit 2 :** fichier caché

**bit 3 :** fichier système

**bit 4 :** étiquette de volume

**bit 5 :** sous répertoire

**bit 6 :** fichier archivé

**bit 7 :** inutilisé

**bit 8 :** inutilisé

Avec cette structure les noms de fichiers étaient limités à 8 caractères dans la FAT12. Dans la FAT16, pour permettre d'avoir des noms plus longs, chaque fichier occupe 2 entrées consécutives ou même davantage. Seule la dernière de ces entrées, qui contient le début du nom, comporte les différents attributs du fichier et permet d'atteindre son contenu. La ou les autres, placées avant cette dernière contiennent le reste du nom et peuvent être considérées comme des compléments de nom. On reconnaît un complément au fait que l'entrée possède à la fois les attributs lecture seulement, fichier caché, fichier système et étiquette volume. On ne traitera pas les compléments.

**Zone des fichiers et sous-répertoires** Elle commence après le répertoire racine et est divisée en clusters, chaque cluster étant attribué entièrement à un seul fichier ou sous-répertoire.

**Calcul du numéro du premier secteur de la zone des fichiers et sous-répertoires**

$$\begin{aligned} no\_prem\_sect\_fichiers = & nb\_sect\_reserves + \\ & nb\_sect\_par\_fat * nb\_fat + \\ & nb\_entrees\_root\_dir * 32 / nb\_octets\_par\_sect \end{aligned}$$

**Calcul du numéro du secteur de début d'un cluster**

$$no\_sect = no\_prem\_sect\_fichier + no\_cluster * nb\_sect\_par\_cluster$$

**Travail à effectuer** Pour lire l'image de la disquette, compléter le programme `fatreader.c` suivant :

```
1  /* programme pour lire une disquette msdos avec utilisation
2   * d'un seul grand buffer contenant toute la disquette*/
3  #include <stdio.h>
4  #include <string.h>
5  #define DISK_SIZE 2880*512
6
7  /* info du volume */
8  int SectorSize;         /* taille du secteur en octets */
9  int ClusterSectSize;    /* taille du cluster en secteurs */
10 int ClusterByteSize;    /* taille du cluster en octets */
11 int FirstRootDirSect;   /* premier secteur du répertoire racine */
12 int NbRootEntries;      /* nombre d'entrées dans le répertoire racine */
13 int FirstDataSector;    /* premier secteur de la zone data*/
14
15 char buf[DISK_SIZE];    //pour recevoir le contenu de la disquette
16
17 void print_vol_info(const unsigned char* buf);
18 void print_dir_entry(const unsigned char* pentry);
19 unsigned char* read_data_cluster(int FirstFileCluster);
```

```
20
21  main(){
22   int i;
23
24   read(0,buf,..............);
25   print_vol_info(buf);
26   fprintf(stderr,"\nContenu du répertoire racine \n\n");
27   for (i=0 ; i<NbRootEntries ; i++)
28      print_dir_entry(buf+.......................);
29  }
30
31  /*
32   * Fonction d'affichage des caractéristiques générales
33   * d'une disquette (contenu du secteur de démarrage)
34   */
35  void print_vol_info(const unsigned char* buf) {
36   int ReservedSectors;   /* nb de secteurs réservés */
37   int FatNumber;         /* nb de FATs */
38   int SectorNumber;      /* nb de secteurs */
39   int FatSectSize;       /* taille d'une FAT en nombre de secteurs */
40   int DiskByteSize;      /* taille d'un disque en octets */
41
42   fprintf(stderr,"\ninformations sur le volume : \n\n");
43   SectorSize = *(unsigned short*) (buf+11);
44   fprintf(stderr,"taille secteur= %d octets\n",SectorSize);
45   ClusterSectSize = *(unsigned char*) (buf+13);
46   fprintf(stderr,"nb secteur par cluster= %d secteurs\n",
47       ClusterSectSize);
48   ReservedSectors = ............................;
49   fprintf(stderr,"nb secteurs réservés= %d\n",ReservedSectors);
50   FatNumber = ..................................;
51   fprintf(stderr,"nb fats= %d\n",FatNumber);
52   NbRootEntries = ..............................;
53   fprintf(stderr,"nb entrées rootdir= %d\n",NbRootEntries);
54   SectorNumber = ...............................;
55   fprintf(stderr,"nb secteurs sur disque= %d\n",SectorNumber);
56   FatSectSize = ................................;
57   fprintf(stderr,"longueur fat en secteurs= %d\n",FatSectSize);
58   ClusterByteSize = ............................;
59   fprintf(stderr,"ClusterByteSize= %d\n",ClusterByteSize);
60   DiskByteSize = ...............................;
61   fprintf(stderr,"DiskByteSize = %d, soit %3.1f MO\n",
62       DiskByteSize,(float)DiskByteSize /(1<<20));
63   FirstRootDirSect = ...........................;
64   fprintf(stderr,"FirstRootDirSect= %d\n",FirstRootDirSect);
65   FirstDataSector = ............................;
66   fprintf(stderr,"FirstDataSector= %d\n",FirstDataSector);
67  }
68
69  /*
```

```
70    * fonction d'affichage des caractéristiques d'un fichier
71    * (contenu d'une entrée du rootdir)
72    */
73   void print_dir_entry(const unsigned char* pentry) {
74   #define Name pentry       /* nom du fichier (8 premiers octets) */
75   #define EMPTY_ENTRY 0x00  /* marks file as deleted when in name[0] */
76   #define DELETED_FILE 0xe5 /* marks file as deleted when in name[0] */
77
78   #define Ext pentry+8      /* extension (3 octets suivants)*/
79   #define Attr pentry+11    /* attributs (octets suivant) */
80
81   #define ATTR_RO      1    /* read-only */
82   #define ATTR_HIDDEN  2    /* hidden */
83   #define ATTR_SYS     4    /* system */
84   #define ATTR_VOLUME  8    /* volume label */
85   #define ATTR_DIR     16   /* directory */
86   #define ATTR_ARCH    32   /* archived*/
87
88   #define COMPL_MASK (ATTR_RO | ATTR_HIDDEN | ATTR_SYS | ATTR_VOLUME)
89
90   #define ATTR_OTH (ATTR_HIDDEN | ATTR_SYS | ATTR_VOLUME | ATTR_ARCH)
91   #define ATTR_NON_NORMAL (ATTR_HIDDEN | ATTR_SYS | ATTR_VOLUME | ATTR_DIR)
92
93   #define First_cluster   *(short *)(pentry+26)
94   #define Size    *(int *)(pentry+28)     /* taille du fichier */
95
96    int i,align;
97    int normal;
98
99    /* entrée valide ? (donnée par le premier octet) */
100   if ((*Name == ........) ||             /* entrée vide */
101       (*Name == ........)) {             /* fichier supprimé */
102       fprintf(stderr,"entrée vide ou fichier détruit\n");
103       return;
104    }
105   /* complement */
106   if ((*(Attr) ............................){   //complement
107       fprintf(stderr,"complement de nom\n");
108       return;
109   }
110   /* nom du fichier  (8 premiers octets + 3 pour l'extension) */
111   /* impression du nom du fichier */
112   /* indication : les caractères affichables sont tels que
113      0x20 <= c <t 0x7F. remplacer les autres par '?' */
114
115
116
117
118
119
```

```
120    /* affichage du point */
121    /* affichage extension */
122
123
124
125
126
127
128
129
130
131
132
133
134    /* attributs (12ème octet) */
135
136
137
138
139
140
141
142    /* affichage de la taille (octets 28-29-30-31) */
143
144    /* affichage du numéro du premier cluster (octets 26-27) */
145
146    /*
147     * si ce n'est pas un fichier normal, on n'affichera pas son contenu
148     */
149    normal= (*(Attr) & ATTR_NON_NORMAL)==0 ;
150    if (normal) {
151        unsigned char* p= read_data_cluster(First_cluster);
152        fprintf(stderr,"contenu du fichier :\n");
153        for (i=0 ; i<100 && i<Size ; i++) {
154            /* affichage du contenu du fichier */
155            /* 100 octets seulement au maximum */
156
157
158        }
159        fprintf(stderr,"\n\n");
160    }
161    else fprintf(stderr,"fichier anormal\n");
162   }
163
164    /*
165     * renvoie l'adresse du premier cluster du fichier dans buf à
166     * partir de son numéro
167     */
168   unsigned char* read_data_cluster(int FirstFileCluster) {
169    return buf+.....................................;
```

```
170     }
```

# Chapitre 2

# Réseaux

## 2.1 Exercices introductifs

### Exercice 2.1.1 : Transmission par satellite

Pour transmettre des trames entre deux points A et B, on utilise un satellite situé à $36.000km$ de la terre qui renvoie immédiatement sans les stocker les trames qu'il reçoit. Les trames font $1.000bits$ et le débit des voies utilisées pour émettre les trames vers et depuis le satellite est de $50Kbits/s$.

**Question 1 :** Quel est le temps de transmission d'une trame de A vers B ?

**Question 2 :** On utilise une procédure dite *d'attente réponse* : A envoie une trame vers B et attend que B acquitte cette trame pour en envoyer une autre. La longueur d'une trame d'acquit est de $100bits$.

Calculer le taux d'utilisation de la voie, c'est-à-dire le rapport du nombre de bits de trame effectivement transmis par unité de temps au débit nominal de la voie (i.e. $50Kbits/s$).

**Question 3 :** Au vu du résultat précédent, on décide de faire de l'anticipation c'est-à-dire que A peut envoyer $k$ trames au maximum successivement, avant de recevoir l'acquittement de la première. Il y a toujours une trame d'acquit par trame émise. Calculer la valeur de $k$ que maximise le débit utile.

### Exercice 2.1.2 :

Sur une liaison hertzienne urbaine à $1.200bits/s$, on envoie des trames de $64bits$.

**Question 1 :** La fréquence d'émission est de $12trames/s$. Calculer le taux d'utilisation de la voie.

**Question 2 :** La voie étant de mauvaise qualité, le taux d'erreur par bit $p$ est compris entre $10^{-2}$ et $10^{-3}$ ($p$ représente la probabilité pour qu'un bit transmis soit faux).

1. Calculer en fonction de $p$ la probabilité pour qu'une trame soit fausse (on suppose que les erreurs altérant les bits sont indépendantes).
2. On suppose que chaque fois qu'une trame est fausse, l'émetteur détecte cette erreur et réémet la trame. Calculer le nombre moyen de transmissions en fonction de $p$.
3. Peut-on, en négligeant les temps d'attente dûs au protocole, transmettre les $12 messages/s$ dans les 2 cas $p = 10^{-2}$ et $p = 10^{-3}$ ? Conclure.

**Question 3 :** On souhaite transferer $M$ bits de l'émetteur vers le récepteur. Ils seront tranportés par des messages de $ln$ bits comportant $C$ bits de contrôle (contenant par exemple numéro de message et autres information de contrôle ...). La probabilité qu'un bit soit erroné est toujours $p$. Calculer $nbme$, le nombre moyen de bits envoyés en fonction de $M$, $ln$, $C$ et $p$.

**Question 4 :** Indiquer une méthode pour trouver la longueur optimum des messages.

## Exercice 2.1.3 : médium à diffusion : transmission hertzienne

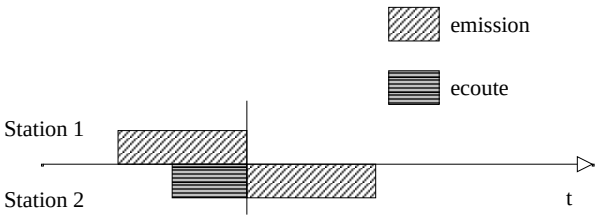
Dans ce type de communication, chaque station utilise des ondes radio (ou infra-rouge) pour échanger des trames avec les autres stations. On suppose qu'il n'y a qu'une seule plage de fréquence disponible pour les transmissions. Ce *medium* est passif et les stations ne relayent pas les trames. Lorsqu'une trame est émise par une station, elle est reçue par toutes les stations : il s'agit d'un *mécanisme de diffusion*.

**Question 1 :** quel est le problème inhérent aux médium à diffusion ?

**Question 2 :** Pendant sa propre émission une station peut-elle détecter une émission d'une autre station ? Pourquoi ?

## Exercice 2.1.4 : médium à diffusion : CSMA (Carrier Sense Multiple Access)

Lorsqu'une station souhaite émettre, elle commence par écouter le canal (*carrier sense*. Si elle détecte un signal sur la ligne, situation appelée *conflit* ou *contention* elle diffère sa transmission jusqu'à la fin de l'émission en cours (technique appelée *CSMA 1-persitant*). Ceci est illustré ci-dessous. Le débit maximum peut atteindre environ 70%.



**Question 1 :** Cette technique permet-elle de supprimer toutes les collisions ?

**Question 2 :** Que doivent faire les stations pour éviter des collisions à la fin d'émission ?

**Question 3 :** Peut-on éliminer toutes les collisions par l'écoute de la porteuse ?

**Question 4 :** Que peut-on faire pour éviter l'allongement de l'attente du médium ?

### Exercice 2.1.5 : medium à diffusion : CSMA/CA *(Collision Avoidance)*

Avec ces techniques on essaye de réserver le médium à une station.

**Question 1 :** Imaginer différentes techniques pour ce faire

### Exercice 2.1.6 : médium à diffusion : bus à diffusion(*CSMA/CD*

Les transmissions se font en bande de base (pas de porteuse), le contexte est similaire à celui de la diffusion hertzienne avec cependant une différence de taille : une station émettrice peut détecter la présence d'une autre émission pendant sa propre émission.

**Question 1 :** pourquoi

**Question 2 :** Si une station émettrice vérifie qu'il n'y a pas de collision pendant toute l'émission, peut-elle être sure qu'il n'y a pas de collision sur cette trame ?

**Question 3 :** Sachant que la norme impose $\tau < 25,6\mu s$ d'un coupleur à l'autre, répéteurs compris, calculer la longueur minimum $L_{min}$ d'une trame pour un réseau ETHERNET à $10Mb/s$ ($5km$, $200000km/s$) ? pour un réseau à $100Mb/s$, pour un réseau à $1Gb/s$ ?

**Question 4 :** Que doit faire une station émettrice si elle détecte une collision ?

## 2.2 Adressage et fragmentation dans Internet

### Exercice 2.2.1 :

**Question 1 :** quelles sont les propriétés que les adresses doivent avoir dans un réseau de communication ?

**Question 2 :** quels sont les avantages et le inconvénients d'une séparation de l'adressage en 2 parties localisatrice et identificatrice telle que celle mise en place dans l'adressage Internet ?

**Question 3 :** pourquoi l'adresse IP ne peut-elle pas être affectée à un périphérique réseau par son fabricant comme c'est le cas de l'adresse MAC pour une carte Ethernet ?

**Question 4 :** combien d'adresses IP et MAC un routeur d'un réseau possède-t-il ?

**Question 5 :** quels sont les champs de l'en-tête IP qui sont modifiés lors de la traversée d'un routeur ?

## Exercice 2.2.2 :

**Question 1 :** Donner l'adresse du réseau et le numéro de la station pour les configurations suivantes :

1. Adresse IP = 132.90.132.21 ; masque de réseau = 255.255.255.240.
2. Adresse IP = 130.97.16.132 ; masque de réseau = 255.255.255.192.
3. Adresse IP = 192.178.16.66 ; masque de réseau = 255.255.255.192.
4. Adresse IP = 128.66.12.1 ; masque de réseau = 255.255.248.0.

## Exercice 2.2.3 :

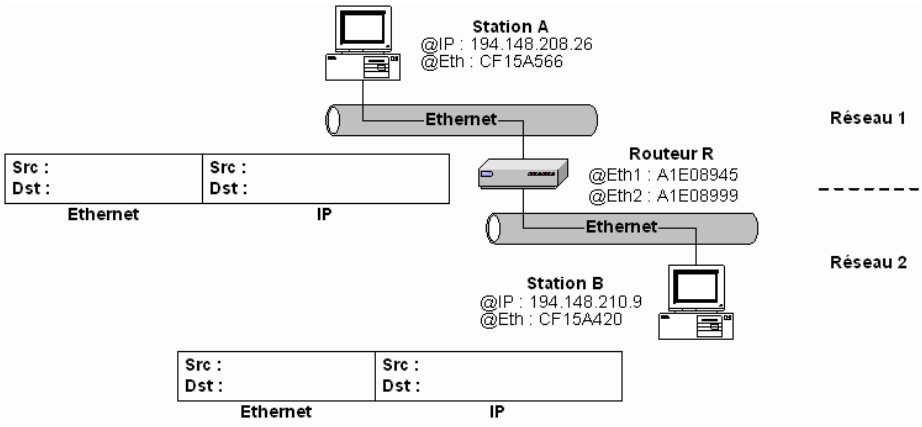
Une entreprise s'est vue attribuer l'adresse réseau 126.123.28.0.

**Question 1 :** quel est son masque de réseau pour 254 noeuds maximum ?

**Question 2 :** En pratique, un administrateur d'un réseau a besoin de créer 12 réseaux, chacun d'eux contenant 12 noeuds. Quel est le nombre de bits à prévoir pour l'identificateur de réseau ? Quel est alors le masque réseau à utiliser ?

## Exercice 2.2.4 :

On considère le réseau IP suivant composé de deux réseaux Ethernet (1 et 2) raccordés par un routeur IP.



**Question 1 :** De quelle classe sont les adresses IP ?

**Question 2 :** Chaque machine dispose d'un système de messagerie. Dessinez l'architecture de protocole complète de chacun des équipements : hôtes A et B, routeur.

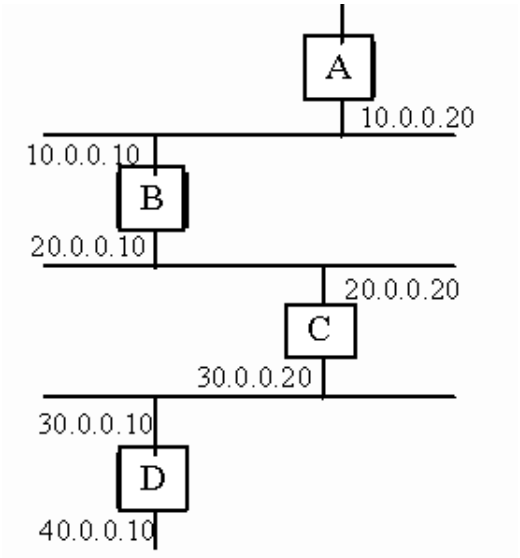
**Question 3 :** Sur le réseau 1, A envoie un mail à B. Représentez les encapsulations successives du mail transmis de A à B.

**Question 4 :** L'hôte A désire envoyer un paquet à l'hôte B, sur chaque réseau 1 et 2, ce paquet est encapsulé dans une trame Ethernet. Complétez le schéma ci-dessous pour faire figurer les adresses source et destination contenues dans la trame Ethernet et dans le paquet IP.

**Question 5 :** Donnez l'état du cache ARP du routeur.

## Exercice 2.2.5 :

**Question 1 :** on considère la figure ci-dessous. Donnez la table de routage de la passerelle D.




## Exercice 2.2.6 :

Un datagramme IP contenant 2000 octets de données est émis sur un réseau A de MTU = 4096. En passant par un routeur R1, il atteint le réseau B de MTU = 1024 octets. Il passe ensuite par un routeur R2 pour atteindre un réseau C de MTU = 512 octets. La structure de l'en-tête du datagramme dans le réseau A est présentée ci-dessous :

<table>
<tr><td>0</td><td>4</td><td>8</td><td>12</td><td>16</td><td>20</td><td>24</td><td>28</td></tr>
<tr><td colspan="2">4</td><td colspan="2">5</td><td colspan="2">0</td><td colspan="4">???</td></tr>
<tr><td colspan="4">identif = 1234</td><td>000b</td><td colspan="3">0</td></tr>
<tr><td colspan="2">9</td><td colspan="2">protocole</td><td colspan="4">total de contrôle en-tête</td></tr>
<tr><td colspan="8">adresse IP source</td></tr>
<tr><td colspan="8">adresse IP destination</td></tr>
</table>

**Question 1 :** compléter le champ en-tête ci-dessus.

**Question 2 :** indiquer la structure de l'en-tête des datagrammes dans les réseaux B et C. Le total de contrôle de l'en-tête n'est pas à calculer.

**Question 3 :** pourquoi le réassemblage de paquets IP n'est-il réalisé que par le destinataire final et pas par un noeud intermédiaire ?

**Question 4 :** que se passe-t-il si le second bit des drapeaux (Défence de Fragmenter) est positionné à 1 lors de son émission ?

**Question 5 :** si le paquet IP utilise l'option de routage par la source est fragmenté au niveau d'un routeur, le champ d'options est-il copié dans chaque fragment ou simplement dans le premier ?

## 2.3 Programmation sockets UDP/TCP

**Identification d'une machine (hôte ou host) sur internet** Une machine peut être désignée de deux façons différentes :

1. 192.70.23.150 : adresse IP notation nombres et points (ou aussi ascii)
2. 0x961746c0 : adresse IP binaire (ici hexa pour voir les octets)

La forme 2 correspond à la structure `in_addr` qui est définie ainsi dans `netinet/in.h` :

```
struct in_addr {
    unsigned long int s_addr;
}
```

On n'utilise pas directement un entier long pour pouvoir modifier ultérieurement la longueur de l'adresse (par exemple pour IPv6) sans changer les programmes. De ce fait, toute manipulation d'une adresse doit mentionner la longueur de cette adresse.

Il existe également une troisième forme : `lunix.iie.cnam.fr`, par exemple, qui est appelée *nom symbolique* (ou aussi *nom DNS*). L'utilisation de cette dernière est liée à l'existence d'une équivalence dans le fichier `/etc/hosts` ou à la disponibilité d'un service de noms (DNS, NIS, ...). Cette troisième forme peut souvent être utilisée à la place de la première, car elles sont toutes deux des chaînes de caractères (type `string`).

**Fonctions de conversion** Il est possible de passer de la forme 1 à la forme 2 et inversement à l'aide de *fonctions de conversion*.

```
int inet_aton (const char *cp, struct in_addr *inp);
```

`inet_aton()` convertit l'adresse Internet `cp` (notation nombres et points) en une donnée binaire (ordre réseau), et la place dans la structure pointée par `inp`. `inet_aton` renvoie une valeur non nulle si l'adresse est valide, et 0 sinon.

```
char *inet_ntoa (struct in_addr in);
```

`inet_ntoa()` convertit l'adresse Internet `in` (notation binaire réseau) en une chaîne de caractères dans la notation nombres et points. La chaîne est renvoyée dans un buffer alloué statiquement, qui est donc écrasé à chaque nouvel appel.

Le nom symbolique doit être traduit par le DNS sous une des formes 1 ou 2 avant de pouvoir être utilisée par les programmes. De plus, une machine (un hôte) peut être connue sous plusieurs noms symboliques différents et avoir plusieurs adresses IP différentes. C'est pourquoi toutes les informations concernant une machine sont rassemblées dans une même structure de données `hostent` qui est définie comme suit dans `netdb.h` :

```
struct hostent {
    char *h_name;                 /* Nom officiel de l'hôte */
    char **h_aliases;             /* Liste d'alias */
    int  h_addrtype;              /* Type d'adresse de l'hôte */
    int  h_length;                /* Longueur de l'adresse */
    struct in_addr **h_addr_list; /* Liste d'adresses */
}
#define h_addr h_addr_list[0]     /* pour compatibilité */
```

Les champs de la structure `hostent` sont :

`h_name` est le nom symbolique officiel de l'hôte.

`h_aliases` est une table, terminée par 0, d'autres noms symboliques de l'hôte.

`h_addrtype` est le type d'adresse (actuellement, toujours `AF_INET`).

`h_length` est la longueur, en octets, de l'adresse réseau.

`h_addr_list` est une table, terminée par 0, de pointeurs vers des adresses réseau pour l'hôte.

`h_addr` est la première adresse dans `h_addr_list` pour respecter la compatibilité ascendante.

Il est possible d'obtenir du DNS une structure `hostent` correspondant à une machine donnée, soit à partir d'un nom symbolique, soit à partir d'une adresse nombres et points, à l'aide des fonctions suivantes :

```
struct hostent *gethostbyname (const char *name);
```

`gethostbyname()` renvoie une structure de type `hostent` pour l'hôte `name`. La chaîne `name` est soit un nom symbolique d'hôte, soit une adresse IPv4 en notation pointée standard. Si `name` est une adresse, aucune recherche supplémentaire n'a lieu et `gethostbyname()` copie simplement la chaîne `name` dans le champ `h_name` et le champ équivalent `struct in_addr` dans le champ `h_addr_list[0]` de la structure `hostent` renvoyée.

```
struct hostent *gethostbyaddr (const struct in_addr * addr,
                               int len, int type);
```

`gethostbyaddr()` renvoie une structure du type `hostent` pour l'hôte dont l'adresse est `addr`. `len` est la longueur de cette adresse. Le seul type d'adresse valide est actuellement `AF_INET`.

**Ordre de rangement des octets (big endian / little endian)** L'ordre de rangement en mémoire des types mutli-octets pour les processeurs i80x86 est LSB (Least significant Byte first ou rangement en mémoire des octets de poids faible en premier ou "little endian").
Cependant l'ordre d'envoi (et donc de réception) des octets sur l'Internet est MSB (Most Significant Byte first ou rangement en mémoire des octets de poids fort en premier ou "big endian").
De ce fait, et pour simplifier l'envoi et la réception des paquets, toutes les structures de données associées à internet sont en big endian et il faut procéder à des inversions d'ordre Host-to-Network (fonction `hton()`) chaque fois qu'on remplit un entier dans une telle structure, ou Network-to-Host (`ntoh()`), chaque fois qu'on lit un entier dans une telle structure. Ces deux fonctions existent aussi pour des entiers `short`. Elles sont décrites ci-dessous.

```
unsigned short int htons (unsigned short int hostshort);
```

`htons()` convertit un entier court (short) `hostshort` depuis l'ordre des octets de l'hôte vers celui du réseau.

```
unsigned short int ntohs (unsigned short int netshort);
```

`ntohs()` convertit un entier court (short) `netshort` depuis l'ordre des octets du réseau vers celui de l'hôte.

**Mécanisme d'accès à un service transport** Le mécanisme d'accès transport est (sous unix ou windows) une entité de type `socket`. Fonctionnellement c'est l'analogue d'un tube (`pipe`) à double sens de transmission, mais distribué sur plusieurs machines et avec différents protocoles de transport (UDP, TCP, autres). Une socket est créée par un appel de la fontion `socket()` :

```
int socket(int famille, int type, int protocole);
```

`socket()` crée un point de communication, et renvoie un descripteur, c'est-à-dire un numéro dans la table des fichiers ouverts.

`famille` permet de sélectionner la famille de protocoles à employer :

- `AF_INET` : communication internet
- `AF_UNIX` : communication locale
- `AF_ISO` : communication ISO

`type` permet de sélectioner le protocole :

- `SOCK_STREAM` : mode connecté (TCP pour internet)
- `SOCK_DGRAM` : mode datagramme (UDP pour internet)

`protocole` permet de sélectionner le protocole à utiliser pour la socket, s'il n'est pas déjà déterminé par la famille et le type. Pour les protocoles de la famille internet, ce paramètre vaut IP (=0).

N.B. Après sa création, une socket **n'est associée ni à une adresse, ni à un port**. Cette association doit être faite ensuite.

Une socket peut être fermée à l'aide de la fonction `close()` (fonction standard de fermeture d'un fichier) ou `shutdown()`.

**Descripteur d'un point d'accès à un service de transport** (extrémité en terminologie unix, TSAP en terminologie OSI)

Un point d'accès à un service de transport est désigné par un couple (identification de machine, identification de port) nommé aussi extrémité. Toutefois les identifications de machine et de port sont spécifiques à chaque famille de protocoles. La structure de données générique `sockadrr` permet d'unifier les traitements dans les programmes :

```
struct sockaddr {
    unsigned short int sa_family; /* famille d'adresses réseau */
    unsigned char sa_data[14];    /* adresse proprement dite */
};
```

Cette structure générique doit être rédéfinie pour chacune des familles de protocoles. Pour la famille internet, il faut utiliser `sockaddr_in` de `netinet/in.h` :

```
struct sockaddr_in {
    unsigned short sin_family;
    in_port_t sin_port;
    struct in_addr sin_addr;
    unsigned char sin_zero[sizeof (struct sockaddr) -
                           sizeof (unsigned short int) -
                           sizeof (in_port_t) -
                           sizeof (struct in_addr)];
};
```

`sin_family` indique la famille internet, `sin_port` le numéro de port, `sin_addr` l'adresse (format binaire).

```
struct in_addr {
    in_addr_t s_addr;
};
```

`sin_zero[]` est du remplissage pour atteindre la taille générique.

**Fonctions communes UDP/TCP** Pendant son fonctionnement une socket est associée à deux extrémités : une extrémité locale et une extrémité distante (extrémité du serveur). Une socket est associée à une extrémité locale soit par une demande d'association explicite grâce à la fonction `bind()` dont un des arguments est un `sockadrr` qui doit être préalablement rempli, soit automatiquement par le système à la suite d'une interaction avec une extrémité distante (fonctions `connect()`, `recvfrom()` ou `sendto`). Lorsqu'une socket a été associée automatiquement à une extémité locale, il est possible d'obtenir la structure `sockadrr` correspondante grâce à la fonction `getsockname()`.

```
int bind(int sockfd, struct sockaddr *local_ext,
         socklen_t sockaddrlen);
```

`bind()` associe la socket `sockfd` à l'extrémité locale contenue dans `local_ext`, qui doit avoir été remplie au préalable. `sockaddrlen` indique la longueur en octets de la structure pointée par `local_ext`.

Traditionnellement cette opération est appelée *assignation d'un nom à une socket*. Le terme de nom doit être compris ici comme l'identification d'une extrémité. Dans le cas d'une socket internet c'est un triplet <famille internet, numéro de port, adresse internet>.

```
int shutdown(int s, int how);
```

`shutdown()` termine tout ou partie d'une connexion full-duplex sur la socket `s`.

```
int getsockname(int s, struct sockaddr * local_tsap,
                socklen_t * namelen )
```

La fonction `getsockname()` est utilisable pour connaîtr le numéro IP et le port lorsqu'il sont attribués par le système. Elle renseigne l'identificateur d'extrémité `sockaddr local_ext` de la socket indiquée. Le paramètre `namelen` doit être initialisé pour indiquer la taille de la zone mémoire pointée par `name`. En retour, il contiendra la taille effective (en octets) du nom renvoyé.

**fonctions orientées UDP**

```
int sendto(int s, const void *msg, size_t len, int flags,
           const struct sockaddr *to, socklen_t tolen);
```

La fonction `sendto()` permet transmettre par la socket `s` un message à l'extrémité identifiée par `to`. Il est possible aussi d'utiliser la fonction `senfmsg`.

```
int recvfrom(int s, void *buf, int len, unsigned int flags,
             struct sockaddr *from, socklen_t *fromlen);
```

La fonction `recvfrom()` est utilisée pour recevoir un message arrivant par la socket s. `from` est renseignée avec les caractérisitques de l'expéditeur. Il est possible aussi d'utiliser la fonction `recvmsg`

### Exercice 2.3.1 :

**Question 1 :** Compléter les programmes suivants pour faire communiquer un serveur et un client UDP.

```
1   /* serveuru.c  (serveur UDP) */
2
3   #include <stdio.h>
4   #include <errno.h>
5   #include <string.h>
6   #include <netinet/in.h>
7   #define BUF_LEN 256
8
9   char* id = 0;
10  short port = 0;
11  int sock = 0; /* socket de communication */
12  int nb_reponse = 0;
13
14  int main(int argc, char** argv) {
15   int ret;
16   struct sockaddr_in serveur; /* extrémité du serveur */
17
18   if (argc!=3) {
19       fprintf(stderr,"usage: %s id port\n",argv[0]);
20       exit(1);
21   }
22   id = argv[1];
23   port = atoi(argv[2]);
24   if ((sock = socket(.....................)) == -1) {
25       fprintf(stderr,"%s: socket %s\n",argv[0],strerror(errno));
26       exit(1);
27   }
28   serveur.sin_family = .......;
29   serveur.sin_port = ............;
30   serveur.sin_addr.s_addr = ..........;
31   if (bind(.............................................) < 0) {
32       fprintf(stderr,"%s: bind %s\n",
33               argv[0],strerror(errno));
34       exit(1);
35   }
36   while (1) {
37       int client_len;
38       struct sockaddr_in client; /* extrémité du client */
```

```
39        char buf_read[BUF_LEN], buf_write[BUF_LEN];
40
41        ret = recvfrom(.....................
42                       ....................................);
43        if (ret <= 0) {
44            printf("%s: recvfrom=%d: %s\n",
45                   argv[0],ret,strerror(errno));
46            continue;
47        }
48        printf("serveur %s recu le message %s de %s:%d\n",id,buf_read,
49               .............................................);
50        sprintf(buf_write,"serveur#%2s reponse%03d#",
51                id,nb_reponse++);
52        ret = sendto(.....................................
53                     ....................................);
54        if (ret <= 0) {
55            printf("%s: sendto=%d: %s\n",
56                   argv[0],ret,strerror(errno));
57            continue;
58        }
59        sleep(2);
60    }
61   return 0;
62   }
```

```
1   /* clientu.c (client UDP) */
2
3   #include <stdio.h>
4   #include <errno.h>
5   #include <netinet/in.h>
6   #include <string.h>
7   #define BUF_LEN 256
8
9   char* id = 0;
10  short sport = 0;
11  int sock = 0; /* socket de communication */
12
13  int main(int argc, char** argv) {
14   struct sockaddr_in client; /* extrémité du client */
15   struct sockaddr_in serveur; /* extrémité du serveur */
16   int client_len= sizeof client;
17   int serveur_len = sizeof serveur;
18   int nb_question = 0;
19   int ret;
20   char buf_read[BUF_LEN], buf_write[BUF_LEN];
21   if (argc != 4) {
22       fprintf(stderr,"usage: %s id host sport\n",argv[0]);
23       exit(1);
24   }
25   id = argv[1];
```

```
26    sport = atoi(argv[3]);
27    if ((sock = socket(.....................)) == -1) {
28        fprintf(stderr,"%s: socket %s\n",argv[0],strerror(errno));
29        exit(1);
30    }
31
32    getsockname(.............................);
33    serveur.sin_family = .......;
34    serveur.sin_port = ............;
35    ..................................&serveur.sin_addr);
36
37    while (nb_question < 3) {
38
39        sprintf(buf_write,"#%2s=%03d",id,nb_question++);
40        printf("client %2s: (%s,%4d) envoie a ",
41               id,.....................................);
42        printf("(%s,%4d): %s",
43               ................................................,
44               .........);
45        ret = sendto(...................................,
46                     ......................................);
47        if (ret <= 0) {
48            printf("%s: erreur dans sendto (num=%d, mess=%s)\n",
49                   argv[0],ret,strerror(errno));
50            continue;
51        }
52
53        getsockname(.............................);
54        printf("client %2s: (%s,%4d) recoit de ",
55               id,.....................................);
56        ret = recvfrom(.....................,
57                       ......................................);
58        if (ret <= 0) {
59            printf("%s: erreur dans recvfrom (num=%d, mess=%s)\n",
60                   argv[0],ret,strerror(errno));
61            continue;
62        }
63        printf("(%s,%4d) : %s\n",.........................,
64               .............................);
65    }
66    return 0;
67  }
```

### connection TCP

`int listen (int s, int nbatt);`

`listen()` informe le système du désir d'accepter des connexions entrantes et de la limite `nbatt` de la file d'entrée pour les demandes en attente. Les connexions seront ensuite acceptées avec `accept()`.

```
int accept(int sock, struct sockaddr *adresse,
           socklent_t *longueur);
```

`accept()` extrait la première connexion de la file des connexions en attente, crée une nouvelle socket avec les mêmes propriétés que `sock` et alloue un nouveau descripteur de fichier pour cette socket. L'argument `adresse` est un paramètre résultat qui est renseigné avec l'adresse de l'entité se connectant, telle qu'elle est connue par la couche de communication. Le format exact du paramètre `adresse` est fonction du domaine dans lequel la communication s'établit. Le paramètre-resultat `longueur` est renseigné avec la longueur (en octets) de l'adresse retournée. Ce paramètre doit initialement contenir la longueur du paramètre `adresse`. Si `adresse` est `NULL`, rien n'est écrit. S'il n'y a pas de connexion en attente dans la file, et si la socket est bloquante, `accept()` se met en attente d'une connexion. Si la socket est non-bloquante, et qu'aucune connexion n'est présente dans la file, `accept()` retourne une erreur. Une socket acceptée ne peut pas être utilisée pour accepter de nouvelles connexions. La socket originale `sock` reste ouverte.

```
int connect(int sockfd, struct sockaddr *serv_ext,
            socklen_t sockaddrlen);
```

La fonction `connect()` est principalement utilisée pour les processus clients orientés connexion.

Si la socket est du type `SOCK_STREAM`, cette fonction tente de se connecter à une autre socket dont l'adresse doit être indiquée par `serv_ext`. Cette autre socket doit être dans le même domaine que la socket initiale.

```
int send(int s, const void *msg, size_t len, int flags);
```

La fonction `send()` permet de transmettre les octets de `msg` à destination de l'autre extémité de la socket `s`. Il est aussi possible d'utiliser la fonction `write` si on n'utilise pas les flags.

```
int recv(int s, void *buf, int len, unsigned int flags);
```

La fonction `recv` est utilisée pour recevoir des octets depuis la socket `s`, le découpage effectif en messages est inaccesible. Il est aussi possible d'utiliser la fonction `read` si on n'utilise pas les flags.

**Question 2 :** Compléter les programmes suivants pour faire communiquer un serveur et un client TCP.

```
1  /* serveurt.c (serveur TCP) */
2
3  #include <stdio.h>
4  #include <errno.h>
5  #include <netinet/in.h>
6  #include <string.h>
7  #include <stdlib.h>
8  #define NBECHANGE 3
9  #define BUF_LEN 256
```

```
10  char* id = 0;
11  short port = 0;
12  int sock = 0; /* socket de communication */
13  int nb_reponse = 0;
14
15  int main(int argc, char** argv) {
16   struct sockaddr_in serveur; /* extrémité du serveur */
17
18   if (argc != 3) {
19       fprintf(stderr,"usage: %s id port\n",argv[0]);
20       exit(1);
21   }
22   id = argv[1];
23   port = atoi(argv[2]);
24   if ((sock = socket(.....................)) == -1) {
25       fprintf(stderr,"%s: socket %s\n",argv[0],strerror(errno));
26       exit(1);
27   }
28   serveur.sin_family = .......;
29   serveur.sin_port = ...........;
30   serveur.sin_addr.s_addr = ..........;
31   if (bind(.............................................) < 0) {
32       fprintf(stderr,"%s: bind %s\n",argv[0],strerror(errno));
33       exit(1);
34   }
35   if (listen(......) != 0) {
36       fprintf(stderr,"%s: listen %s\n",argv[0],strerror(errno));
37       exit(1);
38   }
39   while (1) {
40       struct sockaddr_in client; /* extrémité du client */
41       int client_len = sizeof(client);
42       int sock_pipe; /* socket de dialogue */
43       int ret,nb_question;
44
45       sock_pipe = accept(.................................);
46       for (nb_question = 0 ; nb_question < NBECHANGE ;
47           nb_question++) {
48           char buf_read[BUF_LEN], buf_write[BUF_LEN];
49
50           ret = read(.....................);
51           if (ret <= 0) {
52               printf("%s: read=%d: %s\n",
53                      argv[0], ret, strerror(errno));
54               break;
55           }
56           printf("serveur %s recu de (%s,%4d) : %s\n",id,
57                  ......................,
58                  ..........................);
59           sprintf(buf_write,"#%2s=%03d#",id,nb_reponse++);
```

```
60            ret = write(.....................................);
61            if (ret <= 0) {
62                printf("%s: write=%d: %s\n",
63                       argv[0], ret, strerror(errno));
64                break;
65            }
66            sleep(2);
67        }
68        close(sock_pipe);
69    }
70    return 0;
71   }
```

```
1   /* clientt.c (client TCP) */
2
3   #include <stdio.h>
4   #include <errno.h>
5   #include <netinet/in.h>
6   #include <string.h>
7
8   #define NBECHANGE 3
9   #define BUF_LEN 256
10
11  char* id = 0;
12  short sport = 0;
13  int sock = 0; /* socket de communication */
14
15  int main(int argc, char** argv) {
16   struct sockaddr_in client; /* extrémité du client */
17   struct sockaddr_in serveur; /* extrémité du serveur */
18   int nb_question = 0;
19   int ret;
20   int client_len=sizeof client;
21
22   if (argc != 4) {
23       fprintf(stderr,"usage: %s id serveur port\n",argv[0]);
24       exit(1);
25   }
26   id = argv[1];
27   sport = atoi(argv[3]);
28   if ((sock = socket(.....................)) == -1) {
29       fprintf(stderr,"%s: socket %s\n",argv[0],strerror(errno));
30       exit(1);
31   }
32   serveur.sin_family = AF_INET;
33   serveur.sin_port = htons(sport);
34   inet_aton(.....................................);
35   if (connect(.......................................... ) < 0) {
36       fprintf(stderr,"%s: connect %s\n",argv[0],strerror(errno));
37       perror("bind");
```

```
38        exit(1);
39    }
40
41    getsockname(sock,(struct sockaddr *)&client,&client_len);
42    for (nb_question = 0 ; nb_question < NBECHANGE ;
43         nb_question++) {
44        char buf_read[BUF_LEN], buf_write[BUF_LEN];
45
46        sprintf(buf_write,"#%2s=%03d",id,nb_question);
47        printf("client %2s: (%s,%4d) envoie a ",
48               id,.....................................);
49        printf(" (%s,%4d) : %s\n",.....................,
50               ............................);
51        ret = write(............................);
52        if (ret <= strlen(buf_write)) {
53            printf("%s: erreur dans write (num=%d, mess=%s)\n",
54                   argv[0],ret,strerror(errno));
55            continue;
56        }
57        printf("client %2s: (%s,%4d) recoit de ",
58               id,.....................................);
59        ret = read(..............);
60        if (ret <= 0) {
61            printf("%s:  erreur dans read (num=%d, mess=%s)\n",
62                   argv[0],ret,strerror(errno));
63            continue;
64        }
65        printf("(%s,%4d) : %s\n",inet_ntoa(serveur.sin_addr),
66               ntohs(serveur.sin_port),buf_read);
67    }
68    close(sock);
69    return 0;
70   }
```

**Question 3 :** Étudier le serveur parallèle TCP suivant.

```
/* serveurf.c  (serveur parallele TCP) */

#include <stdio.h>
#include <errno.h>
#include <netinet/in.h>
#include <string.h>
#include <stdlib.h>
#define NBECHANGE 3
#define BUF_LEN 256

char* id = 0;
short port = 0;
int sock = 0; /* socket de communication */
int nb_reponse = 0;
```

```
int main(int argc, char** argv) {
 int ret;
 struct sockaddr_in serveur; /* extrémité du serveur */

 if (argc != 3) {
     fprintf(stderr,"usage: %s id port\n",argv[0]);
     exit(1);
 }
 id = argv[1];
 port = atoi(argv[2]);
 if ((sock = socket(AF_INET,SOCK_STREAM,0)) == -1) {
     fprintf(stderr,"%s: socket %s\n",argv[0],strerror(errno));
     exit(1);
 }
 serveur.sin_family = AF_INET;
 serveur.sin_port = htons(port);
 serveur.sin_addr.s_addr = INADDR_ANY;
 if (bind(sock,(struct sockaddr *)&serveur,sizeof(serveur)) < 0) {
     fprintf(stderr,"%s: bind %s\n",argv[0],strerror(errno));
     exit(1);
 }
 if (listen(sock,5) != 0) {
     fprintf(stderr,"%s: listen %s\n",argv[0],strerror(errno));
     exit(1);
 }
 while (1) {
     struct sockaddr_in client; /* extrémité du client */
     int len=sizeof(client);
     int sock_pipe; /* socket de dialogue */
     int nb_question;

     sock_pipe = accept(sock,(struct sockaddr *)&client,&len);
     if (!fork()) {
         close(sock);
         for (nb_question = 0 ; nb_question < NBECHANGE ;
              nb_question++) {
             char buf_read[BUF_LEN], buf_write[BUF_LEN];

             ret = read(sock_pipe,buf_read,BUF_LEN);
             if (ret <= 0) {
                 printf("%s: read=%d: %s\n",
                 argv[0],ret,strerror(errno));
                 break;
             }
             printf("serveur %s recu de (%s,%4d) : %s\n",id,
                    inet_ntoa(client.sin_addr),
                    ntohs(client.sin_port),buf_read);
             sprintf(buf_write,"%s#%2s=%03d#",
                     buf_read,id,nb_reponse++);
```

```
            ret = write(sock_pipe,buf_write,strlen(buf_write)+1);
            if (ret <= 0) {
                printf("%s: write=%d: %s\n",
                       argv[0],ret,strerror(errno));
                break;
            }
            sleep(2);
        }
        close(sock_pipe);
        exit(0);
    } else nb_reponse+=NBECHANGE;
 }
 return 0;
}
```

### Exercice 2.3.2 : Piratage de liaison

**Question 1 :** Écrire des programmes serveurs et client UDP. Les faire fonctionner sur la même machine puis sur deux machines différentes.

Pendant l'exécution d'un client `A`, lancer un client `B` qui envoie ses messages sur le port du client `A` et non celui du serveur.

**Question 2 :** Que se passe-t-il ? Modifier le client pour éviter ce phénomène.

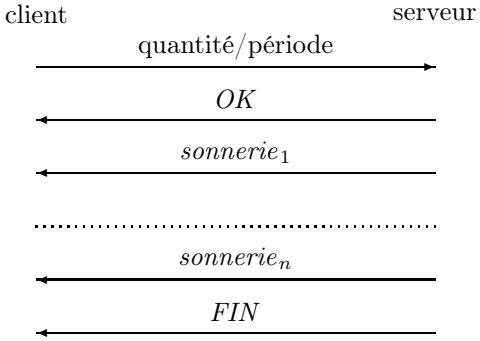
### Exercice 2.3.3 : Dialogue serveur/client UDP selon un protocole

Le client :

1. envoie un message contenant 2 entiers $(Q, P)$.
2. lit les messages et les affiche en clair (type et valeur).
3. se termine sur réception du message *FIN*.

Le serveur :

1. lit le message du client $(Q, P)$.
2. répond immédiatement par le message *OK*.
3. répète $Q$ fois :
   - attendre $P$ secondes,
   - envoyer un message de sonnerie $I$, où $I$ est le numéro de séquence.
4. envoie le message *FIN*.




**Format des PDUs**

**quantité/période** 100 caractères contenant $Q$ et $P$ en ascii.
- les caractères ne contenant pas les chiffres de $P$ et $Q$ sont des blancs.
- $P$ et $Q$ sont séparés par au moins 1 blanc.
- le dernier caractère du message est un blanc.

$sonnerie_I$ 100 caractères :
- le premier est un 'S'
- le second est un blanc
- N en ascii.
- les derniers caractères sont des blancs.

**FIN** 100 caractères :
- les 3 premiers sont 'E', 'N' et 'D'.
- les derniers caractères sont des blancs.

**OK** 100 caractères :
- les 2 premiers sont 'O' et 'K'.
- les derniers caractères sont des blancs.

Réaliser cette application en UDP. Le serveur aura 1 argument : le port. Le client aura 4 arguments : le serveur (dotted format), le port, $Q$ et $P$.

### Exercice 2.3.4 : Sockets TCP

Écrire des programmes serveurs et client TCP. Les faire fonctionner sur la même machine puis sur deux machines différentes.

### Exercice 2.3.5 : Dialogue serveur/client TCP selon un protocole

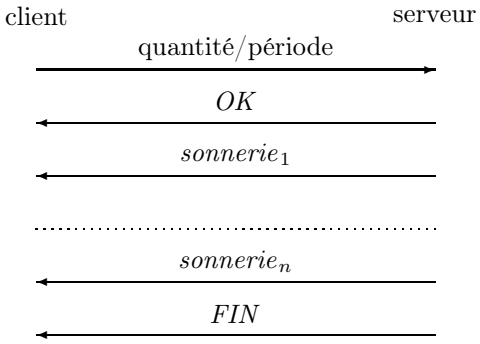
Le client :

1. envoie un message contenant 2 entiers $(Q, P)$.
2. lit les messages et les affiche en clair (type et valeur).
3. se termine sur réception du message *FIN*.

Le serveur :

1. lit le message du client $(Q, P)$.
2. répond immédiatement par le message *OK*.

3. répète $Q$ fois :
   - attendre $P$ secondes,
   - envoyer un message de sonnerie $I$, où $I$ est le numéro de séquence.
4. envoie le message *FIN*.




**Format des PDUs**

**quantité/période** 2 deux entiers sur 16 bits :
- le premier est $Q$
- le second est $P$

$sonnerie_I$ 1 octet contenant 'S' suivi d'un entier sur 16 bits donnant le $I$.

**FIN** 3 octets contenant 'E', 'N' et 'D'.

**OK** 2 octets contenant 'O' et 'K'.

**Question 1 :** Réaliser cette application en TCP. Le serveur aura 1 argument : le port. Le client aura 4 arguments : le serveur (format ascii), le port, $Q$ et $P$.

**Question 2 :** Modifier cette application pour que le serveur puisse gérer simultanément plusieurs clients.