
*Programmation
système et réseau sous
Windows et Unix*

*Patrick Ducrot
dp@ensicaen.fr*

Plan du document

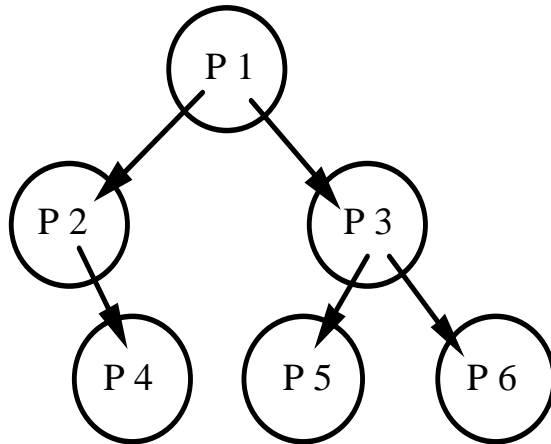
<i>Description des processus</i>	<i>3</i>
<i>Gestion des processus et communication par tube sous Unix</i>	<i>9</i>
<i>Gestion de processus et communication par tube sous Windows</i>	<i>27</i>
<i>Synchronisation de processus</i>	<i>43</i>
<i>Les sémaphores sous Unix</i>	<i>46</i>
<i>Les sémaphores sous Windows</i>	<i>50</i>
<i>Les threads</i>	<i>55</i>
<i>Les threads sous Unix</i>	<i>60</i>
<i>Les threads sous Windows</i>	<i>63</i>
<i>Synchronisation de threads</i>	<i>67</i>
<i>Synchronisation de threads sous Unix</i>	<i>70</i>
<i>Synchronisation de threads sous Windows</i>	<i>79</i>
<i>Programmation réseau</i>	<i>89</i>
<i>Remote Procedure Call</i>	<i>129</i>
<i>La bibliothèque SDL</i>	<i>157</i>
<i>Programmation réseau IPv6</i>	<i>161</i>

Description des processus

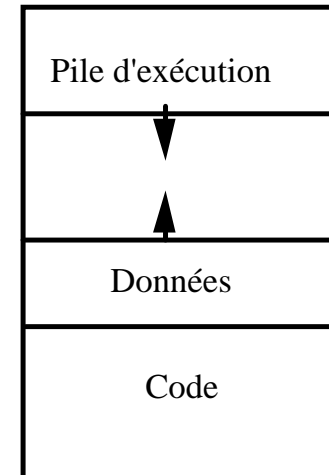
Définition d'un processus

- Un processus correspond à l'exécution d'une tâche.
- Aspect dynamique d'un programme.
- Un système d'exploitation doit exécuter « simultanément » plusieurs processus.
- Chaque processus est composé en mémoire:
 - Un espace pour le code
 - Un espace pour les données
 - Un espace pour la pile d'exécution

Définition des processus



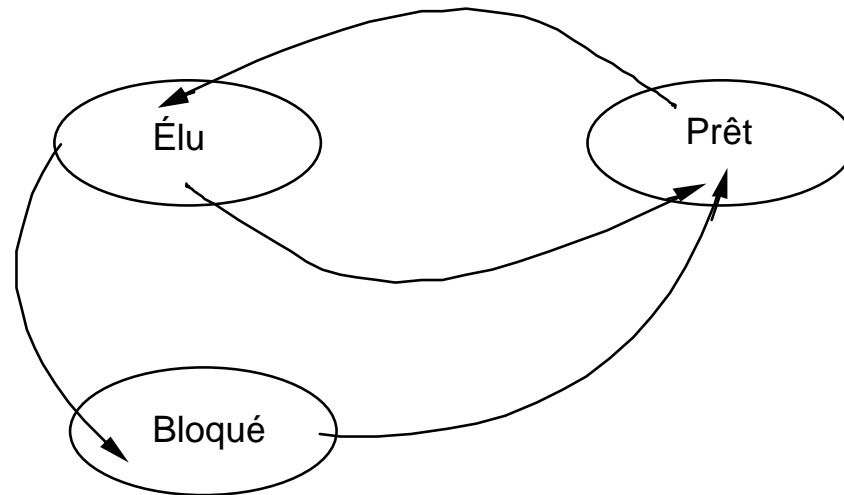
Hiérarchie des processus



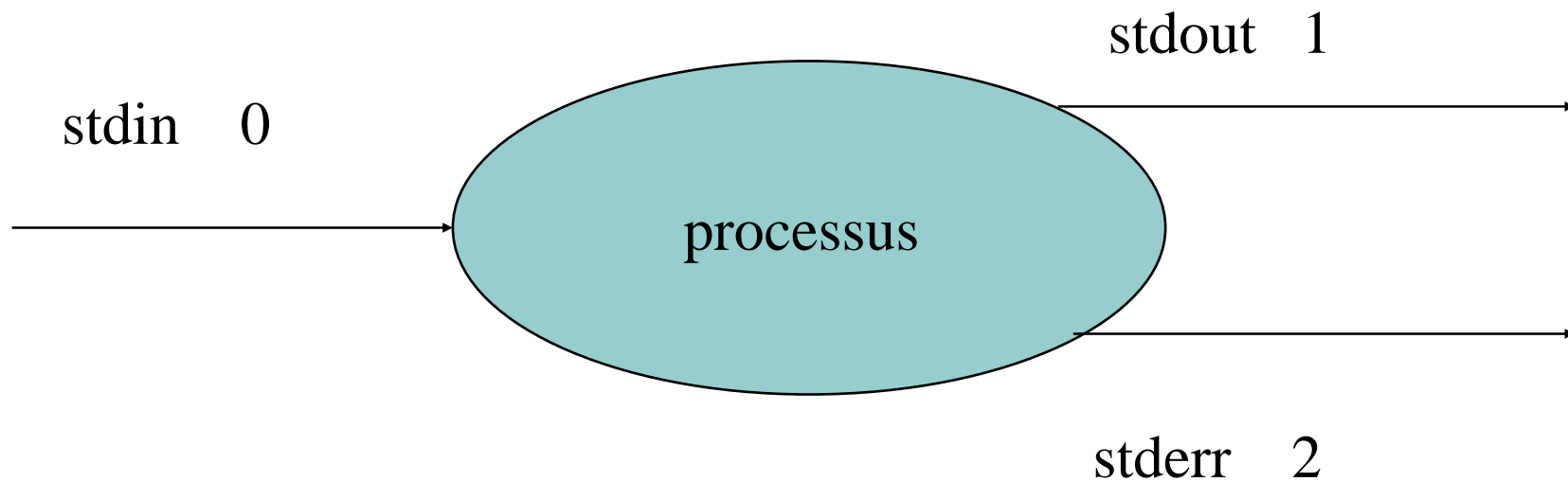
Structure d'un processus

État d'un processus

- Le passage d'un processus à un autre est assuré par un ordonnanceur.

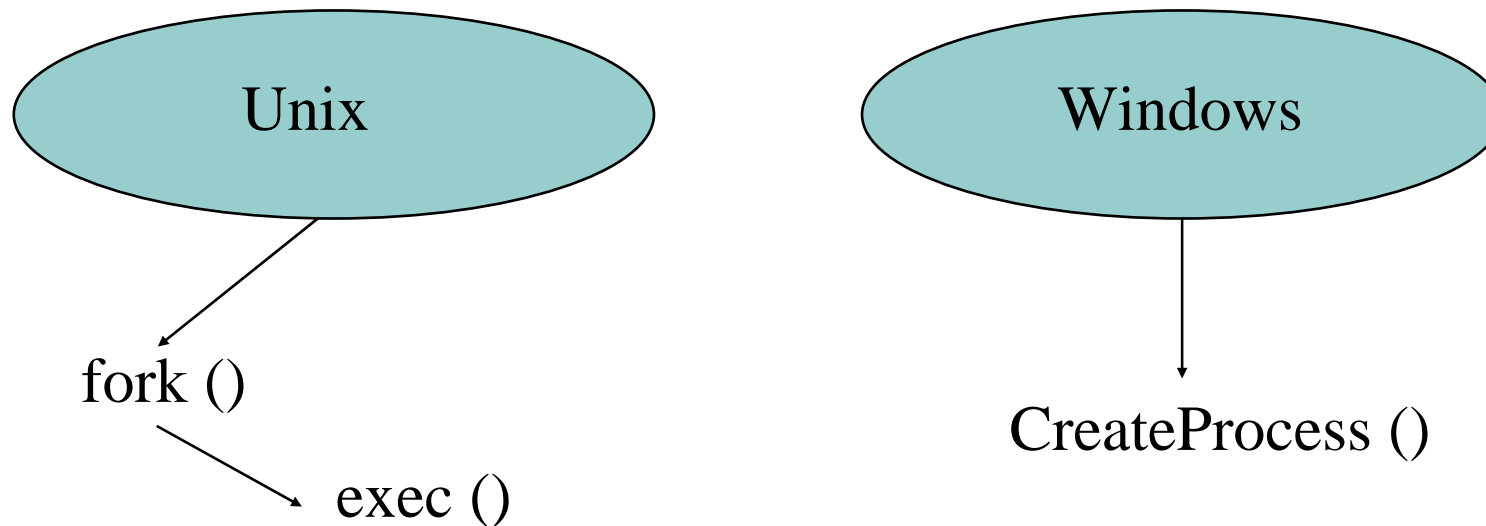


Descripteurs d'Entrée/Sortie



Création de processus

- Fonctionnement très différent entre Windows et Unix:



Gestion de processus sous UNIX

Création d'un processus sous Unix

- Sous Unix, un processus peut se dupliquer:

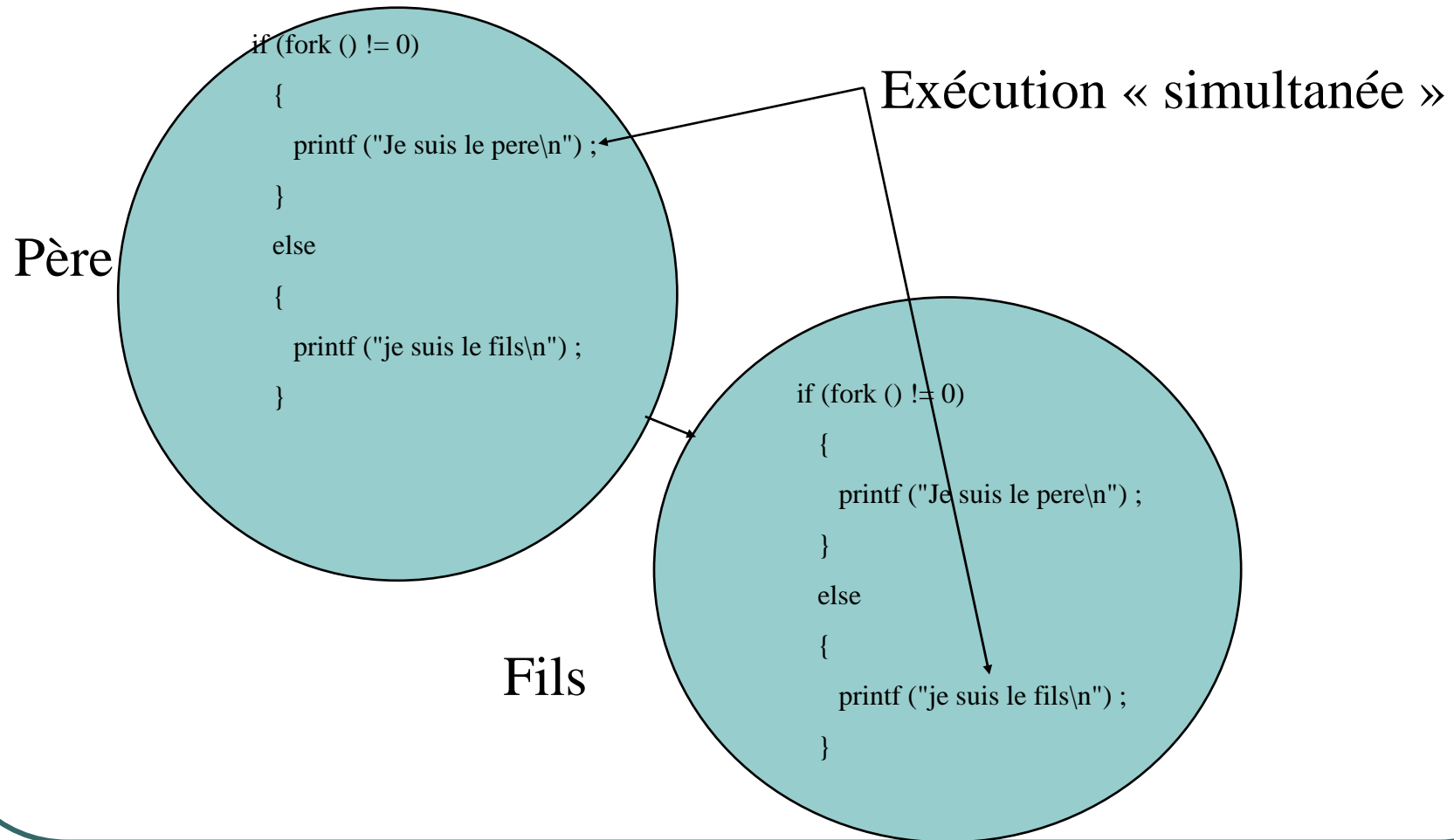
pid_t fork(void);

- La fonction renvoie:
 - -1 si erreur
 - 0 pour le processus fils
 - >0 pour le processus père (Process IDentification)

Duplication de processus UNIX

```
void main ()
{
  if (fork () != 0)
  {
    printf ("Je suis le pere\n");
  }
  else
  {
    printf ("je suis le fils\n");
  }
}
```

État après le fork ()



Invocation d'un programme existant

- L'invocation d'un programme existant ne peut se faire que par le recouvrement du code du processus appelant grâce à une des fonctions de la famille *exec*.
- D'où l'intérêt voire la nécessité de dupliquer préalablement le processus.

La famille de fonctions exec

*int execl (char *path, char *arg0, ..., char *argn, NULL);*

*int execv (char *path, char *argv []);*

*int execl (char *path, char *arg0, ..., char *argn, NULL, char *envp []);*

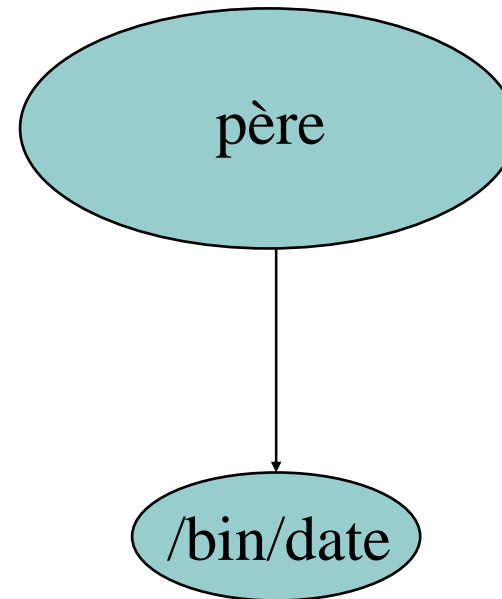
*int execve (char *path, char *argv [], char *envp []);*

*int execlp (char *file, char *arg0, ..., char *argn, NULL);*

*int execvp (char *file, char *argv []);*

Exemple exec

```
#include <stdio.h>
void main ()
{
  if ( fork () != 0)
  {
    printf ("je suis le pere, je continue ici\n") ;
  }
  else
  {
    printf ("fils: je me transforme\n") ;
    execl ("/bin/date","date",NULL) ;
  }
}
```



Fin d'un processus

- Un processus s'arrête lorsque:
 - Il n'a plus d'instruction à exécuter
 - Il exécute un appel à la fonction
void exit (int status) ;
- Lorsqu'un processus fils est terminé, celui-ci sera retiré de la table des processus si:
 - Le père acquitte la fin du processus fils par un appel à la fonction: *int wait(int *code_de_sortie) ;*
 - Le père ignore la fin des processus fils par un appel à la fonction:
signal (SIGCHLD,SIG_IGN) ;

Gestion de signaux

- Un signal permet de dérouter l'exécution d'un processus.
- A réception d'un signal un processus peut:
 - se terminer (comportement par défaut)
 - l'ignorer
 - exécuter une fonction particulière
- Un signal peut être envoyé par:
 - une commande shell:
kill -num_signal PID
 - par programmation
int kill (int pid, int signal) ;

Quelques signaux Unix

Nom du signal	No signal	Commentaires
SIGUP	1	signal émis lors d'une déconnexion
SIGINT	2	^C
SIGQUIT	3	^\
SIGKILL	9	signal d'interruption radicale
SIGCLD	20	signal émis par un fils qui se termine à son père

Détournement d'un signal

- Un processus peut détourner les signaux reçus et modifier son comportement par l'appel de la fonction :

```
void (*signal(int signal, void (*fonction)(int)))
```

le deuxième argument pouvant prendre les valeurs:

Nom	Action
SIG_IGN	le processus ignorera l'interruption correspondante
SIG_DFL	le processus rétablira son comportement par défaut lors de l'arrivée de l'interruption (la terminaison)
<code>void fonction(int n)</code>	Le processus exécutera <code>fonction</code> , définie par l'utilisateur, à l'arrivée de l'interruption <code>n</code> . Il reprendra ensuite au point où il a été interrompu

Exemple détournement signal

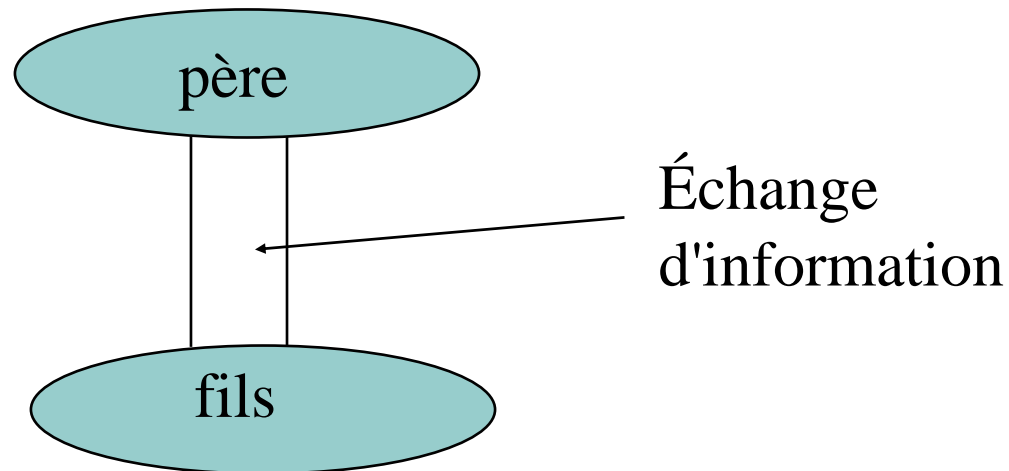
```
#include <stdio.h>
#include <sys/signal.h>

void detourne (int num_signal)
{
    printf ("signal num %d\n",num_signal) ;
}

void main ()
{
    signal (SIGINT,detourne) ;
    while (1) sleep (10) ;
}
```

Communication par tube anonyme

- 2 processus disposant d'un lien de parenté peuvent échanger de l'information par l'intermédiaire d'un tube anonyme.



Les tubes anonymes

- Le support de communication est le système de fichier
- La communication est en mode caractère.
- Création d'un tube anonyme:

```
int p [2] ;
```

```
pipe (p) ;
```

p [0] descripteur en lecture

p[1] descripteur en écriture



Les tubes anonymes

- Lecture/Ecriture dans un tube anonyme :

*int read (int desc, char *buf, int nb)*

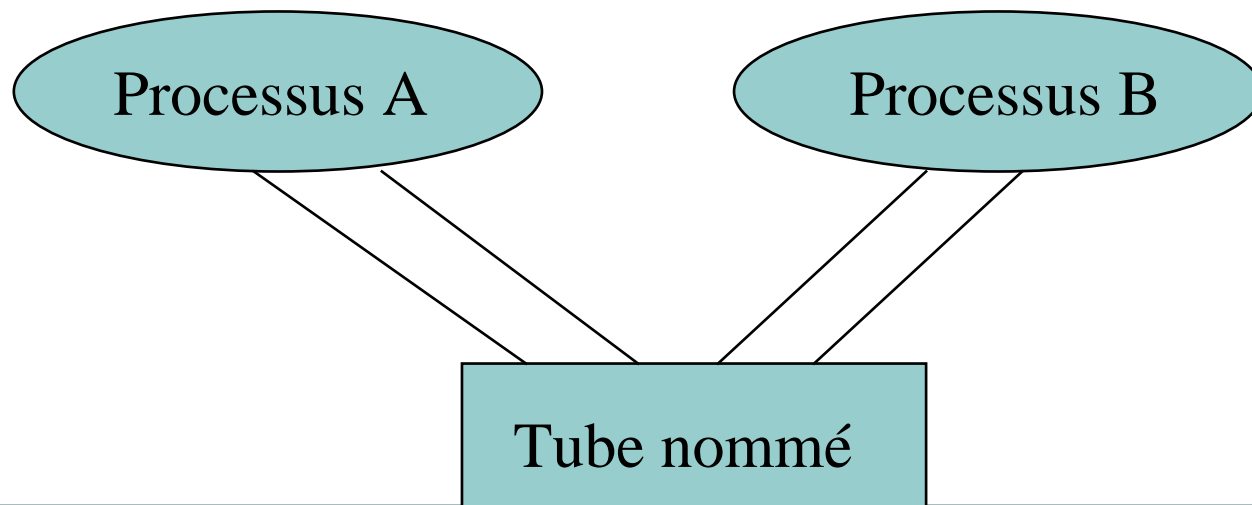
*int write (int desc, char *buf, int nb)*

- Fermeture d'un descripteur

void close (int desc)

Les tubes nommés

- Les tubes nommés portent un nom ;)
- Ils permettent à deux processus sans lien de parenté de s'échanger de l'information.



Tubes nommés

- Création d'un tube nommé avec une commande shell

mkfifo tube

- Ouverture/Lecture/Ecriture/Fermeture

*int open (char *file,int mode,int droits)*

*int read (int desc,char *buf,int nb)*

*int write (int desc,char *buf,int nb)*

void close (desc)

Duplication de descripteurs

- Un descripteur de fichier peut être dupliqué vers une entrée disponible
- Duplication d'un descripteur sur le plus petit descripteur disponible

int dup (int desc)

- Duplication d'un descripteur vers un descripteur spécifié (avec éventuellement fermeture de celui-ci)

int dup2 (int desc1, int desc2)

Gestion de processus sous Windows

Création de processus sous Windows

- **Lorsqu'un processus sous Windows est créé, son premier thread (thread primaire) est créé automatiquement par le système.**
- **Le thread primaire peut créer d'autres threads qui peuvent à leur tour en créer d'autres.**
- **Un thread quelconque peut créer un processus par un appel à la fonction *CreateProcess ()***
- **Rôle de *CreateProcess ()* :**
 - **Crée un espace d'adressage virtuel pour ce processus**
 - **Charge le processus dans cet espace**
 - **Définit le thread primaire pour ce processus**
 - **Lance l'exécution du thread primaire**
- ***CreateProcess* renvoie TRUE si la création est un succès**

La fonction `CreateProcess ()`

```
BOOL CreateProcess (  
    LPCTSTR lpApplicationName,  
    LPCTSTR lpCommandLine,  
    LPSECURITY_ATTRIBUTES lpProcessAttributes,  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    BOOL bInheritHandles,  
    DWORD dwCreationFlags,  
    LPVOID lpEnvironment,  
    LPTSTR lpCurrentDirectory,  
    LPSTARTUPINFO lpStartupInfo,  
    LPPROCESS_INFORMATION lpProcessInformation  
);
```

Paramètres de CreateProcess ()

1/5

- **lpApplicationName** : identifie le nom du fichier exécutable qui peut se trouver :
 - Dans le répertoire courant, dans le répertoire SYSTEM et dans les répertoires spécifiés dans la commande PATH
 - On peut passer NULL : dans ce cas, on considère que le nom du fichier exécutable est le 1ier élément (jusqu'au premier espace) du paramètre **lpCommandLine**
- **lpCommandLine** : spécifie les arguments de la ligne de commande à passer au processus incluant le nom de l'exécutable si lpApplicationName vaut NULL.

Paramètres de CreateProcess ()

2/5

- **lpProcessAttributes** et **lpThreadAttributes** : : pointent sur une structure SECURITY_ATTRIBUTES qui indique qui a le droit de se partager l'objet et si d'autres processus ont le droit de le modifier; On peut passer NULL, dans ce cas, le processus et le thread reçoivent le descripteur par défaut et personne ne pourra en hériter
- **bInheritHandles** : indique si l'identificateur de l'objet processus/thread est héritant. En général un objet possède des Id's différents lorsqu'il est utilisé dans des processus différents. Toutefois, Windows permet aux Id's d'être hérités

Paramètres de CreateProcess ()

3/5

- **dwCreationFlags** : identifie des indicateurs (séparés par des OU logiques) qui affectent la création
Exemples : `CREATE_SUSPENDED`
`CREATE_NEW_CONSOLE`
...
- **lpEnvironment** : pointe sur un bloc de mémoire contenant les chaînes d'environnement utilisées par le processus. Le plus souvent on passe NULL, ce qui signifie que le processus enfant emploie les mêmes chaînes d'environnement que son père

Paramètres de `CreateProcess ()`

4/5

- **`lpCurrentDirectory`** : permet au processus appelant `CreateProcess` d'indiquer à Windows le nom du répertoire courant pour le nouveau processus. Si on passe `NULL`, le répertoire courant est le même que celui de l'application l'ayant créée
- **`lpStartupInfo`** : pointe sur une structure `STARTUPINFO` qui contient des membres que l'on peut affecter, soit pour des applications console ou pour des applications graphiques ou les deux, pour le lancement

Paramètres de CreateProcess ()

5/5

- **lpProcessInformation** : pointe sur une structure PROCESS_INFORMATION que CreateProcess remplira avant de retourner ; la structure a l'aspect suivant :

```
typedef struct _PROCESS_INFORMATION  
{  
    HANDLE hProcess;  
    HANDLE hThread;  
    DWORD dwProcessId;  
    DWORD dwThreadId;  
} PROCESS_INFORMATION;
```

Exemple de CreateProcess

```
#include <windows.h>
void main ()
{
    STARTUPINFO si ;
    PROCESS_INFORMATION pi ;

    GetStartupInfo (&si) ; // Initialisation de la STARTUPINFO
    CreateProcess (NULL,
                  "notepad",
                  NULL,NULL,
                  FALSE,
                  0,NULL,NULL,
                  &si,&pi) ;
}
```

La structure STARTUPINFO

```
typedef struct STARTUPINFO
{
    DWORD cb; // taille de la structure
    LPTSTR lpReserved; // doit être laissé à NULL
    LPTSTR lpDesktop; // nom de l'objet bureau
    LPSTR lpTitle; // légende pour le titre de la fenêtre (mode console)
    DWORD dwX; // coin supérieur gauche de la fenêtre
    DWORD dwY;
    DWORD dwXSize; // largeur nouvelle fenêtre
    DWORD dwYSize; // hauteur nouvelle fenêtre
    DWORD dwXCountChars; // largeur nouvelle fenêtre console
    DWORD dwYCountChars; // hauteur nouvelle fenêtre console
    DWORD dwFillAttribute; // couleur texte et fond pour la console
    DWORD dwFlags; // activation des champs de la structure
    WORD wShowWindow; // valeur de iCmdShow
    WORD cbReserved2; // zéro
    LPBYTE lpReserved2; // NULL
    HANDLE hStdInput; // handle d'entrée standard
    HANDLE hStdOutput; // handle de sortie standard
    HANDLE hStdError; // handle d'erreur standard
} STARTUPINFO, *LPSTARTUPINFO;
```

Création/Fermeture d'un tube anonyme sous Windows

```
BOOL CreatePipe ( PHANDLE hRead,           // Handle de lecture  
                  PHANDLE hWrite,         // Handle d'écriture  
                  LPSECURITY_ATTRIBUTES lpPipeAttributes, // Privilège d'accès  
                  DWORD nSize) // taille du tube, valeur par défaut si 0
```

```
typedef struct _SECURITY_ATTRIBUTES  
{  
    DWORD nLength;           // Doit contenir la taille de la structure  
    LPVOID lpSecurityDescriptor; // Paramètre de sécurité  
    BOOL bInheritHandle;     // TRUE si le pipe doit être transmis par héritage  
} SECURITY_ATTRIBUTES;
```

```
BOOL CloseHandle( HANDLE hObject );           // Handle à fermer
```

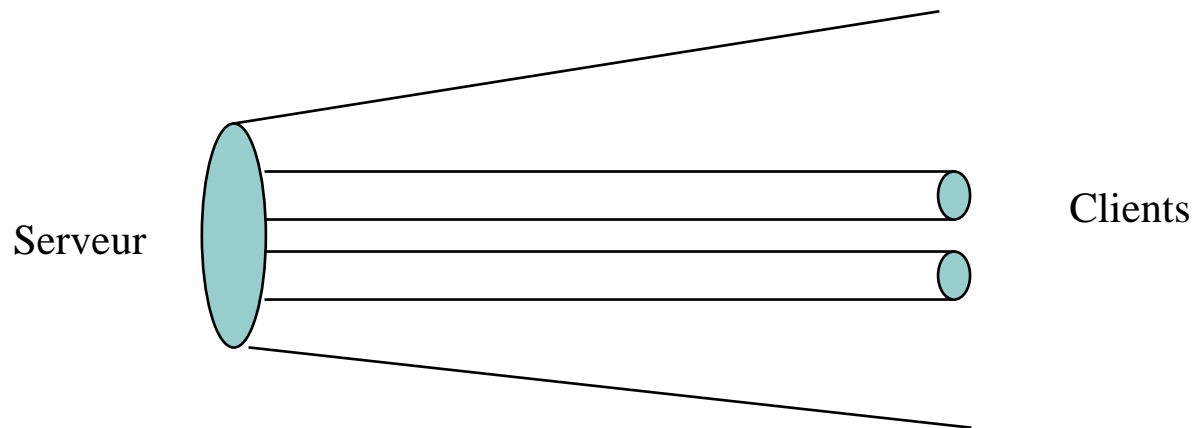
Utilisation d'un tube anonyme sous Windows

```
BOOL ReadFile (  
    HANDLE hFile,                // handle de lecture  
    LPVOID lpBuffer,            // adresse de réception des données  
    DWORD nNumberOfBytesToRead, // Nombre d'octets réservés  
    LPDWORD lpNumberOfBytesRead, // adresse de réception du nombre d'octets lus  
    LPOVERLAPPED lpOverlapped // lecture asynchrone  
);
```

```
BOOL WriteFile (  
    HANDLE hFile,                // handle d'écriture  
    LPVOID lpBuffer,            // Adresse des données à écrire  
    DWORD nNumberOfBytesToWrite, // nombre d'octets à écrire  
    LPDWORD lpNumberOfBytesWritten, // adresse de réception du nombre d'octets à écrire  
    LPOVERLAPPED lpOverlapped // Ecriture asynchrone  
);
```

Tubes nommés sous Windows

- Concept très évolué sous Windows:
 - Un serveur (créateur du tube nommé) peut avoir plusieurs clients (instance)
 - Les clients et le serveur peuvent être sur des machines distinctes (moyennant les problèmes de droits d'accès).



Création d'un tube nommé sous Windows

```
HANDLE CreateNamedPipe(  
    LPCTSTR lpName,           // Nom du tube: \\.\pipe\nom  
    DWORD dwOpenMode,       // Mode d'ouverture du fichier  
    DWORD dwPipeMode,       // Mode d'ouverture spécifique au tube  
    DWORD nMaxInstances,     // Nombre maximal d'instances  
    DWORD nOutBufferSize,   // Taille du buffer de sortie  
    DWORD nInBufferSize,    // Taille du buffer d'entrée  
    DWORD nDefaultTimeOut, // Timeout si un client appelle  
    WaitNamedPipe  
    LPSECURITY_ATTRIBUTES lpSecurityAttributes // Privilège d'accès  
);
```

Exemple tube nommé sous Windows

```
tube = CreateNamedPipe ("\\\\.\\pipe\\ServeurDessin",
    PIPE_ACCESS_DUPLEX | FILE_FLAG_WRITE_THROUGH,
    PIPE_TYPE_MESSAGE | PIPE_WAIT,
    1,0,0,0,NULL) ;

if (tube == INVALID_HANDLE_VALUE)
{
    LPVOID lpMsgBuf;FormatMessage(
    FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,  NULL,
    GetLastError(),
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
    (LPTSTR) &lpMsgBuf,  0,  NULL );// Display the string.
    ::MessageBox( NULL, (char *)lpMsgBuf, "GetLastError", MB_OK|MB_ICONINFORMATION );
    LocalFree( lpMsgBuf );
    MessageBox ("Le tube de communication n'a pas pu être créé","Erreur",MB_ICONERROR |
        MB_OK) ;
    PostQuitMessage (0) ;
}
```

Connexion à un tube nommé sous Windows

```
HANDLE CreateFile (  
    LPCTSTR lpFileName,           // Nom du tube  
    DWORD dwDesiredAccess,       // Mode d'accès (lecture/écriture)  
                                     // GENERIC_READ si le tube a été créé avec PIPE_ACCESS_OUTBOUND  
                                     // GENERIC_WRITE si le tube a été créé avec PIPE_ACCESS_INBOUND  
                                     // GENERIC_WRITE | GENERIC_READ si duplex  
    DWORD dwShareMode,          // 0 pour interdire le partage du tube avec d'autres processus  
    LPSECURITY_ATTRIBUTES lpSecurityAttributes, // Privilège d'accès  
    DWORD dwCreationDistribution, // OPEN_EXISTING puisqu'on ouvre un tube existant  
    DWORD dwFlagsAndAttributes, // FILE_FLAG_WRITE_THROUGH pour une écriture synchrone  
    HANDLE hTemplateFile         // Pas de signification pour les tubes  
    );
```

Synchronisation de processus

Synchronisation de processus

- Plusieurs processus peuvent entrer en conflit pour l'accès à une ressource:
 - Accès en écriture à un fichier texte.
 - Accès à de la mémoire partagée.
 - ...
- On peut souhaiter limiter le nombre d'accès simultanés à une ressource:
 - Accès en lecture à un fichier texte.
 - ...

Définition d'un sémaphore

- Les sémaphores est un concept introduit par Dijkstra.
- Un sémaphore est une valeur entière, initialisée avec le nombre d'accès simultanés autorisés, associée à 2 fonctions atomiques (P (sem) et V (sem)) pour acquérir et rendre un accès et une file d'attente.
- Acquérir un accès revient à décrémenter la valeur du sémaphore; si la valeur devient <0 , le processus est placé en file d'attente.
- Rendre un accès revient à incrémenter la valeur du sémaphore et réveiller un processus si la valeur est ≤ 0 .

Les Sémaphores sous Unix

Les sémaphores sous Unix

- Unix gère les sémaphores à la norme "Posix" (proche de la définition des sémaphores de Dijkstra) qui seront vus au chapitre des threads et des sémaphores "Unix" qui sont des tableaux de sémaphores.
- Les types et prototypes nécessaires pour manipuler les sémaphores "Unix" sont:

```
#include <sys/ipc.h>  
#include <sys/sem.h>
```

Manipulations de sémaphores sous Unix (1/2)

- Création:

```
int semget(key_t clé, int NombreSéma, int flag);
```

Exemple:

```
cle = ftok("nom_fichier_existant", 'A');
```

```
idSema = semget(cle, 5, IPC_CREAT | 0600); // Création d'un tableau de 5 sémaphores  
// avec des droits d'accès exclusifs.
```

- Manipulation (initialisation, consultation, destruction):

```
int semctl(int idSema, int NombreSéma, int commande, ushort semarray);
```

Exemple:

```
semctl (semid,0,SETVAL,1) ; // initialisation à 1 du sémaphore numéro 0
```

Manipulations de sémaphores sous Unix (2/2)

- **Acquisition/Libération:**

```
int semop(int idSema, struct sembuf *sops, size_t NbOpérat) ;
```

Exemple:

```
struct sembuf op ;  
op.sem_num = 0 ; // Indice du sémaphore au sein du tableau de sémaphore  
op.sem_op = 1 ; // 1 pour acquisition, -1 pour libération  
op.sem_flg = 0 ; // 0 pour acquisition bloquante  
  
semop (semid,&op,1) ; // Acquisition, si la valeur est < 0, la fonction sera bloquante
```

Les sémaphores sous Windows

Création sémaphore Windows

- **Création d'un sémaphore:**

```
HANDLE CreateSemaphore (  
LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,  
LONG lInitialCount,  
LONG lMaximumCount,  
LPCTSTR lpName );
```

- **Ouverture d'un sémaphore existant:**

```
HANDLE OpenSemaphore  
( DWORD dwDesiredAccess, // SEMAPHORE_ALL_ACCESS  
// SEMAPHORE_MODIFY_STATE  
BOOL bInheritHandle,  
LPCTSTR lpName );
```

Utilisation sémaphore Windows

- Demande d'un accès:

```
DWORD WaitForSingleObject( HANDLE hHandle, // Handle du sémaphore  
                           DWORD dwMilliseconds // time-out en millisecondes );
```

- Restitution d'accès:

```
BOOL ReleaseSemaphore ( HANDLE hSemaphore,           // handle du sémaphore  
                       LONG lReleaseCount,          // Valeur à ajouter au sémaphore  
                       LPLONG lpPreviousCount // Récupération de la valeur  
                                                         // précédente  
                       );
```

Exemple (1/2)

```
#include <windows.h>
#include <stdio.h>
void main ()
{
    HANDLE sem ;
    LONG previous ;
    int i ;

    sem= CreateSemaphore (NULL,1,1,"semaphore") ;
    if (sem == NULL) { exit (1) ; }
    for (i = 0 ; i != 10 ; i++)
    {
        WaitForSingleObject (sem,INFINITE) ;
        printf ("A: j'ai pris\n") ;
        Sleep (5000) ;
        printf ("A: je libere\n") ;
        ReleaseSemaphore (sem,1,&previous) ;
    }
}
```

Exemple (2/2)

```
#include <windows.h>
#include <stdio.h>
void main ()
{
    HANDLE sem ;
    LONG previous ;
    int i ;

    sem = OpenSemaphore (SEMAPHORE_ALL_ACCESS,FALSE,"semaphore") ;
    if (sem == NULL) { exit (1) ; }
    for (i = 0 ; i != 10 ; i++)
    {
        WaitForSingleObject (sem,INFINITE) ;
        printf ("B: j'ai pris\n") ;
        Sleep (3000) ;
        printf ("B: je libere\n") ;
        ReleaseSemaphore (sem,1,&previous) ;
    }
}
```

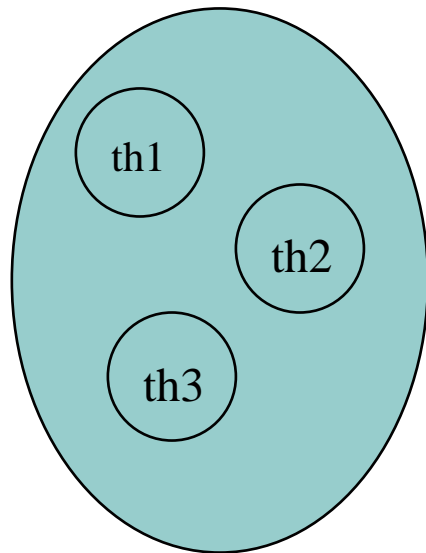
Les threads

Définition d'un Thread

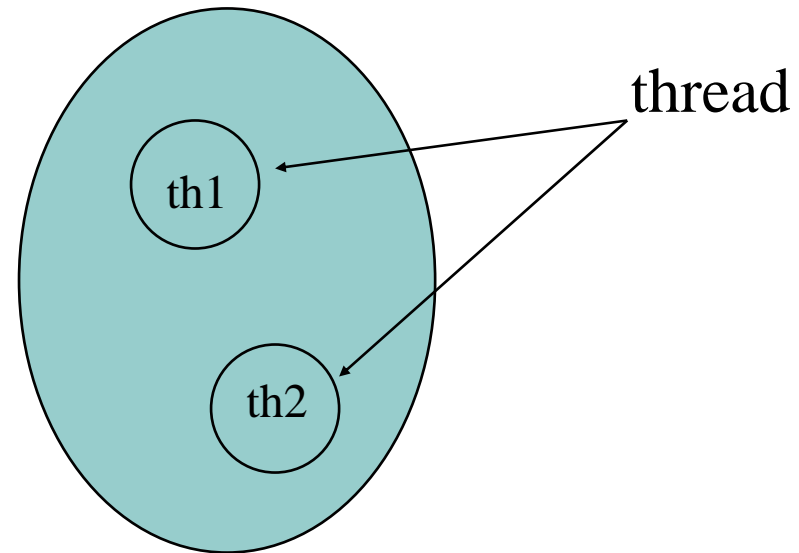
- Un thread est une unité d'exécution au sein d'un processus.
- Un processus peut être composé de plusieurs threads.
- Des quantums de temps seront alloués à chacun des threads, en fonction de leur priorité.
- Tous les threads d'un même processus partagent le même espace mémoire.
- L'échange d'informations entre deux threads est plus rapide qu'entre deux processus mais peut engendrer des conflits d'accès aux ressources.

Définition d'un thread

Processus 1



Processus 2



Similitudes threads Unix/Windows

- Chaque thread dispose d'un point d'entrée (fonction).
- La fonction thread dispose d'un et d'un seul argument totalement à la charge du programmeur.
- La fonction thread retourne une valeur.
- La fonction thread doit utiliser les paramètres et variables locaux aussi souvent que possible pour éviter les conflits avec la zone globale.
- La mort du thread principal entraîne la mort de tous les threads.

Création de threads

Unix

`pthread_create ()`

`pthread_exit ()`

`pthread_join ()`

Windows

`CreateThread ()`

`ExitThread ()`

`WaitForSingleObject ()`

Threads Unix

Unix: création d'un thread

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

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

Exemple Thread Unix

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

char message[] = "Bonjour" ;

void *thread_function (void *arg)
{
    printf ("thread démarre avec %s\n", (char *) arg) ;
    sleep (3) ;
    strcpy (message,"Bye") ;
    pthread_exit ("au revoir") ;
}

int main ()
{
    pthread_t a_thread ;
    void *thread_result ;
    if ( pthread_create (&a_thread,NULL,thread_function,
                        (void *) message) < 0)
    {
        perror ("erreur creation du thread") ;
        exit (EXIT_FAILURE) ;
    }
    printf ("On attend la fin du thread\n") ;
    if ( pthread_join (a_thread,&thread_result) < 0)
    {
        perror ("erreur join") ;
        exit (EXIT_FAILURE) ;
    }
    printf ("Fin du join, le thread a retourné %s\n", (char *)
thread_result) ;
    printf ("message contient maintenant : %s\n",message) ;
    exit (EXIT_SUCCESS) ;
}
```

Threads Windows

Création thread Windows

HANDLE CreateThread

```
(  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    DWORD dwStackSize,  
    LPTHREAD_START_ROUTINE lpStartAddress,  
    LPVOID lpParameter,  
    DWORD dwCreationFlags,  
    LPDWORD lpThreadId  
);
```

Exemple thread Windows (1/2)

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

char message [ ] = "Bonjour" ;

DWORD WINAPI thread_function (PVOID arg)
{
    printf ("thread démarre avec %s\n", (char *) arg) ;
    Sleep (3000) ;
    strcpy (message,"Bye") ;
    return 100 ;
}
```

Exemple thread Windows (2/2)

```
void main ()
{
    HANDLE a_thread ;
    DWORD a_threadId ;
    DWORD thread_result ;
    if ( (a_thread = CreateThread (NULL,0,thread_function,(PVOID)message,0,&a_threadId) ) == NULL)
    {
        perror ("erreur creation du thread") ;
        exit (EXIT_FAILURE) ;
    }
    printf ("On attend la fin du thread\n") ;
    if ( WaitForSingleObject (a_thread,INFINITE) != WAIT_OBJECT_0)
    {
        perror ("erreur join") ;
        exit (EXIT_FAILURE) ;
    }
    GetExitCodeThread (a_thread,&thread_result) ;
    printf ("Fin du join, le thread a retourne %d\n",thread_result) ;
    printf ("message contient maintenant : %s\n",message) ;
    exit (EXIT_SUCCESS) ;
}
```

Synchronisation des threads

Synchronisation threads

- Tous les threads s'exécutent dans le même espace mémoire.
- Deux threads peuvent vouloir accéder à la même zone mémoire "en même temps".
- Sans précaution, le résultat sera imprévisible.
- L'accès à des zones mémoires partagées doit donc être réglementé.

Un outil de synchronisation: le mutex

- Un mutex est un sémaphore binaire pouvant prendre un état verrouillé ou déverrouillé.
- Un mutex ne peut être partagé que par des threads d'un même processus.
- Un mutex ne peut être verrouillé que par un seul thread à la fois.
- Un thread qui tente de verrouiller un mutex déjà verrouillé est suspendu jusqu'à ce que le mutex soit déverrouillé.

Synchronisation Threads Unix

Exemple programme erroné

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

int compteur = 0 ;

void *function (void
*arg)
{
    int i,a ;
    for (i = 0 ; i < 10 ; i++)
    {
        a = compteur ; a = a
+ 1 ;
        sleep (1) ;
        compteur = a ;
    }
}
```

```
int main ()
{
    int res1,res2 ;
    pthread_t a_thread1,a_thread2 ;
    void *thread_result1,*thread_result2 ;

    res1 = pthread_create (&a_thread1,NULL,function,NULL) ;
    res2 = pthread_create (&a_thread2,NULL,function,NULL) ;

    printf ("On attend la fin des threads\n") ;
    pthread_join (a_thread1,&thread_result1) ;
    pthread_join (a_thread2,&thread_result2) ;
    printf ("compteur = %d\n",compteur) ;
}
```

Synchronisation des threads Unix

- La synchronisation des threads peut se faire avec:
 - un sémaphore « Unix » (vu avec les processus).
 - Un sémaphore « Posix ».
 - un mutex.

Les sémaphores "Posix"

- Unix gère les sémaphores à la norme "Posix" (proche de la définition des sémaphores de Dijkstra). Sous Linux, ils ne sont utilisables que pour les threads (2ème argument de `sem_init` doit être 0).
- Les types et prototypes nécessaires pour manipuler les sémaphores "Posix" sont dans le fichier d'entête "*semaphore.h*".
- Un sémaphore est une variable de type *sem_t*.

Manipulation d'un sémaphore Posix

- Initialisation:

*int sem_init(sem_t *sem, int pshared, unsigned int value);*

- Acquisition d'un accès:

- bloquant: *int sem_wait(sem_t * sem);*

- non bloquant: *int sem_trywait(sem_t * sem);*

- Libération d'un accès:

*int sem_post(sem_t * sem);*

- Consultation de la valeur d'un sémaphore:

*int sem_getvalue(sem_t * sem, int * sval);*

- Destruction d'un sémaphore:

*int sem_destroy(sem_t * sem);*

Exemple sémaphore Posix

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
int compteur = 0 ;
sem_t sem ;
void *function (void *arg)
{
    int i,a ;
    for (i = 0 ; i < 10 ; i++)
    {
        sem_wait (&sem) ;
        a = compteur ;
        a = a + 1 ;
        sleep (1) ;
        compteur = a ;
        sem_post (&sem) ;
    }
}
```

```
int main ()
{
    int res1,res2 ;
    pthread_t a_thread1,a_thread2 ;
    void *thread_result1,*thread_result2 ;

    sem_init (&sem,0,1) ;

    res1 = pthread_create (&a_thread1,NULL,function,NULL) ;
    res2 = pthread_create (&a_thread2,NULL,function,NULL) ;

    printf ("On attend la fin des threads\n") ;
    pthread_join (a_thread1,&thread_result1) ;
    pthread_join (a_thread2,&thread_result2) ;
    sem_destroy (&sem) ;
    printf ("compteur = %d\n",compteur) ;
}
```

Mutex sous Unix

- Un mutex est une variable de type *pthread_mutex_t*
- Elle peut être initialisée:
 - par la constante *PTHREAD_MUTEX_INITIALIZER*
 - Par un appel à la fonction:

```
int pthread_mutex_init ( pthread_mutex_t *mutex,  
                        const pthread_mutexattr_t *mutexattr ) ;
```

Utilisation mutex Unix

- Verrouillage d'un mutex

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

- Déverrouillage d'un mutex

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

Exemple mutex Unix

```
pthread_mutex_t monMutex = PTHREAD_MUTEX_INITIALIZER ;
```

```
void *function (void *arg)
{
    int i,a ;
    for (i = 0 ; i < 10 ; i++)
    {
        pthread_mutex_lock (&monMutex) ;
        a = compteur ;
        a = a + 1 ;
        sleep (1) ;
        compteur = a ;
        pthread_mutex_unlock (&monMutex) ;
    }
}
```

Synchronisation

Threads Windows

Synchronisation des threads Windows

- La synchronisation des threads peut se faire avec:
 - un sémaphore (déjà vu avec les processus).
 - un mutex.
 - une section critique.
 - un verrou sur les variables.

Mutex Windows

- Création d'un mutex:

```
HANDLE CreateMutex (  
    LPSECURITY_ATTRIBUTES lpMutexAttributes,  
    BOOL bInitialOwner, // Si TRUE, le mutex est créé et pris  
    LPCTSTR lpName // Nom du mutex  
);
```

- Ouverture d'un mutex existant:

```
HANDLE OpenMutex (  
    DWORD dwDesiredAccess, // MUTEX_ALL_ACCESS ou SYNCHRONIZE  
    BOOL bInheritHandle, // Flag d'héritage  
    LPCTSTR lpName // Nom du mutex  
);
```

Mutex Windows

- Acquisition d'un mutex:

```
DWORD WaitForSingleObject (  
HANDLE hHandle,  
DWORD dwMilliseconds  
);
```

- Libération d'un mutex:

```
BOOL ReleaseMutex (HANDLE hMutex);
```

Exemple Mutex 1/2

```
#include <stdio.h>
#include <windows.h>
int compteur = 0 ;
DWORD WINAPI thread (LPVOID arg)
{
    int i,a ;
    for (i = 0 ; i != 10 ; i++)
    {
        WaitForSingleObject ( (HANDLE) arg,INFINITE) ;
        a= compteur ;
        Sleep (1000) ;
        a = a + 1 ; compteur = a ;
        ReleaseMutex ( (HANDLE) arg) ;
    }
    return 0 ;
}
```

Exemple mutex 2/2

```
void main ()
{
    DWORD ThreadId ;
    HANDLE ta,tb ;
    HANDLE mutex ;

    mutex = CreateMutex (NULL,FALSE,"MonMutex") ;

    ta = CreateThread (NULL,0,thread,mutex,0,&ThreadId) ;
    tb = CreateThread (NULL,0,thread,mutex,0,&ThreadId) ;

    WaitForSingleObject (ta,INFINITE) ;
    WaitForSingleObject (tb,INFINITE) ;

    printf ("compteur = %d\n",compteur) ;
}
```

Section critique

- `VOID InitializeCriticalSection (`
 `LPCRITICAL_SECTION lpCriticalSection // Adresse de la section critique`
 `);`
- `VOID EnterCriticalSection (`
 `LPCRITICAL_SECTION lpCriticalSection // Adresse de la section critique`
 `);`
- `VOID LeaveCriticalSection (`
 `LPCRITICAL_SECTION lpCriticalSection // Adresse de la section critique`
 `);`
- `VOID DeleteCriticalSection (`
 `LPCRITICAL_SECTION lpCriticalSection // Adresse de la section critique`
 `);`
- `BOOL TryEnterCriticalSection (`
 `LPCRITICAL_SECTION lpCriticalSection // Adresse de la section critique`
 `);`

Exemple de section critique

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

CRITICAL_SECTION verrou ;

DWORD WINAPI function_thread (LPVOID param)
{
    int i ;
    char *name = (char *) param ;
    EnterCriticalSection (&verrou) ;
    for (i = 0 ; i != 1000 ; i++)
    {
        printf ("%s: %d\n",name,i) ;
    }
    LeaveCriticalSection (&verrou) ;
    return 0 ;
}
```

Exemple de section critique (2/2)

```
void main (int argc, char **argv [])
{
    HANDLE ta, tb ;
    DWORD ThreadId ;
    InitializeCriticalSection (&verrou) ;

    ta = CreateThread (NULL, 0, function_thread, "Thread A", 0, &ThreadId) ;
    tb = CreateThread (NULL, 0, function_thread, "Thread B", 0, &ThreadId) ;

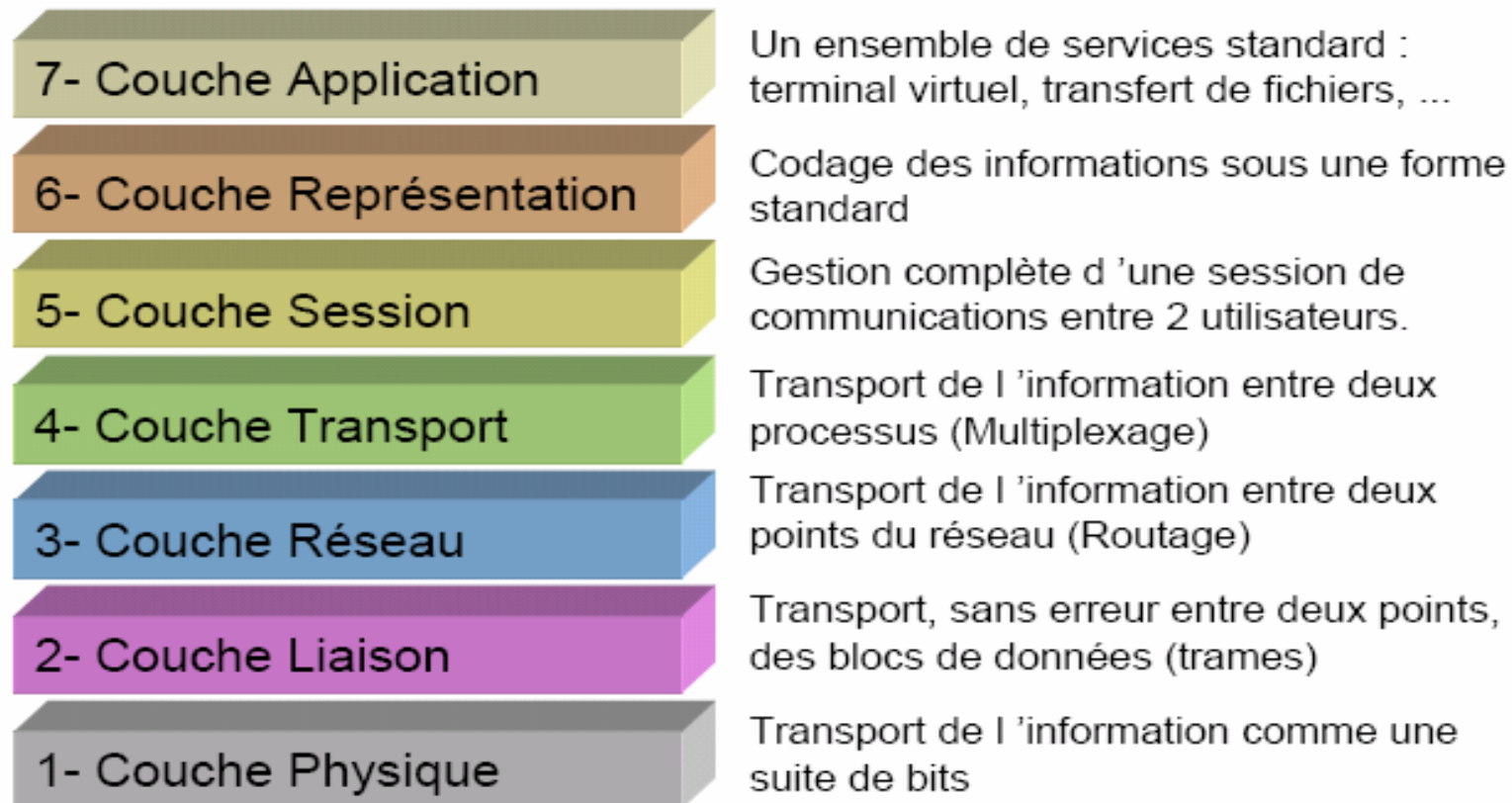
    Sleep (10000) ;
}
```

Les fonctions "Interlocked"

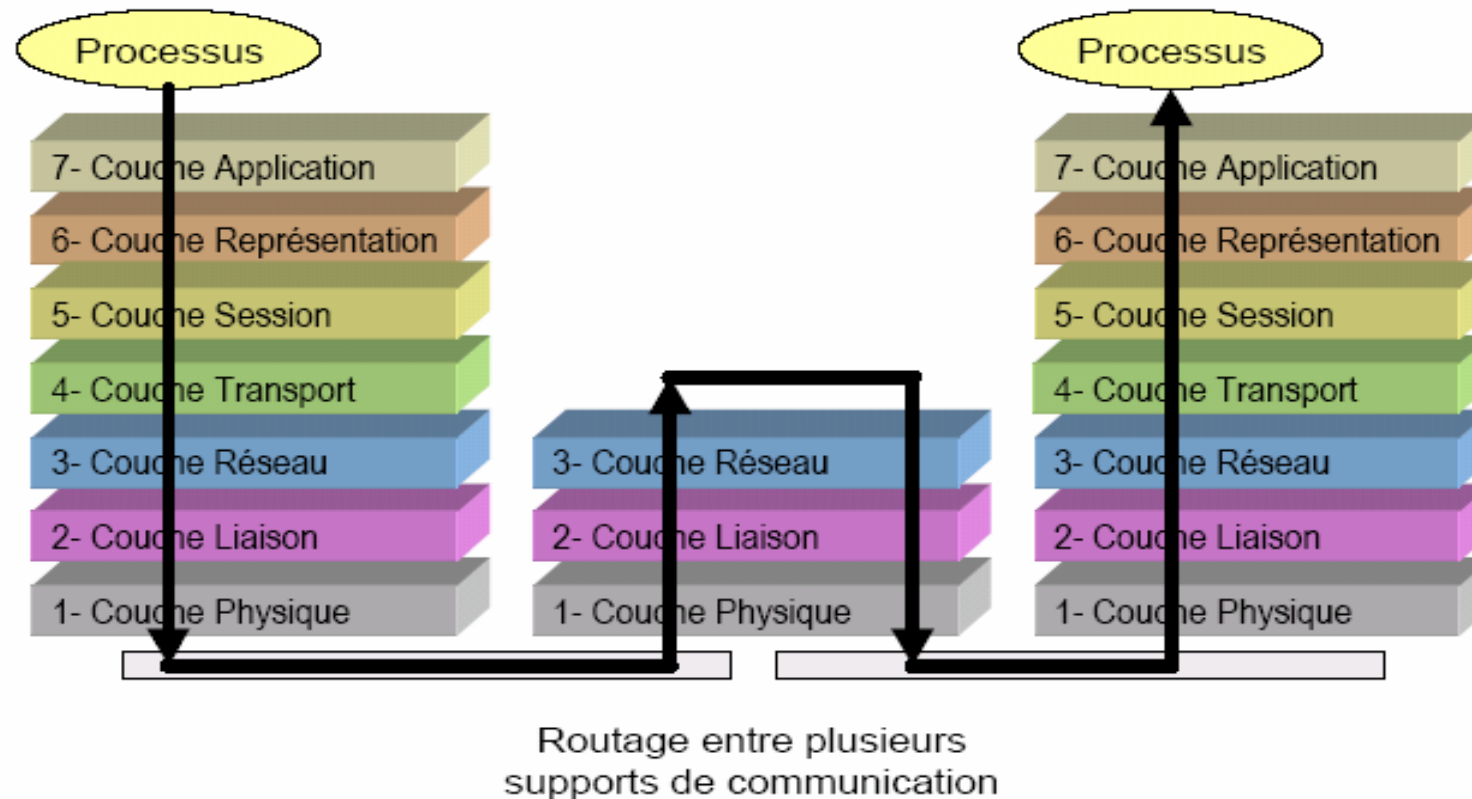
- **LONG InterlockedIncrement(LPLONG *lpAddend*);** // Pointeur sur la variable à incrémenter
- **LONG InterlockedDecrement(LPLONG *lpAddend*);** // Pointeur sur la variable à décrémenter
- **PVOID InterlockedCompareExchange(**
 PVOID **Destination*, // pointer to the destination pointer
 PVOID *Exchange*, // the exchange value
 PVOID *Comperand* // the value to compare
);
- **LONG InterlockedExchange(LPLONG *Target*, // pointer to the 32-bit value to exchange**
 LONG *Value* // new value for the LONG value pointed to by *Target*
);
- **LONG InterlockedExchangeAdd (**
 PLONG *Addend*, // pointer to the addend
 LONG *Increment* // increment value
);

Programmation réseau

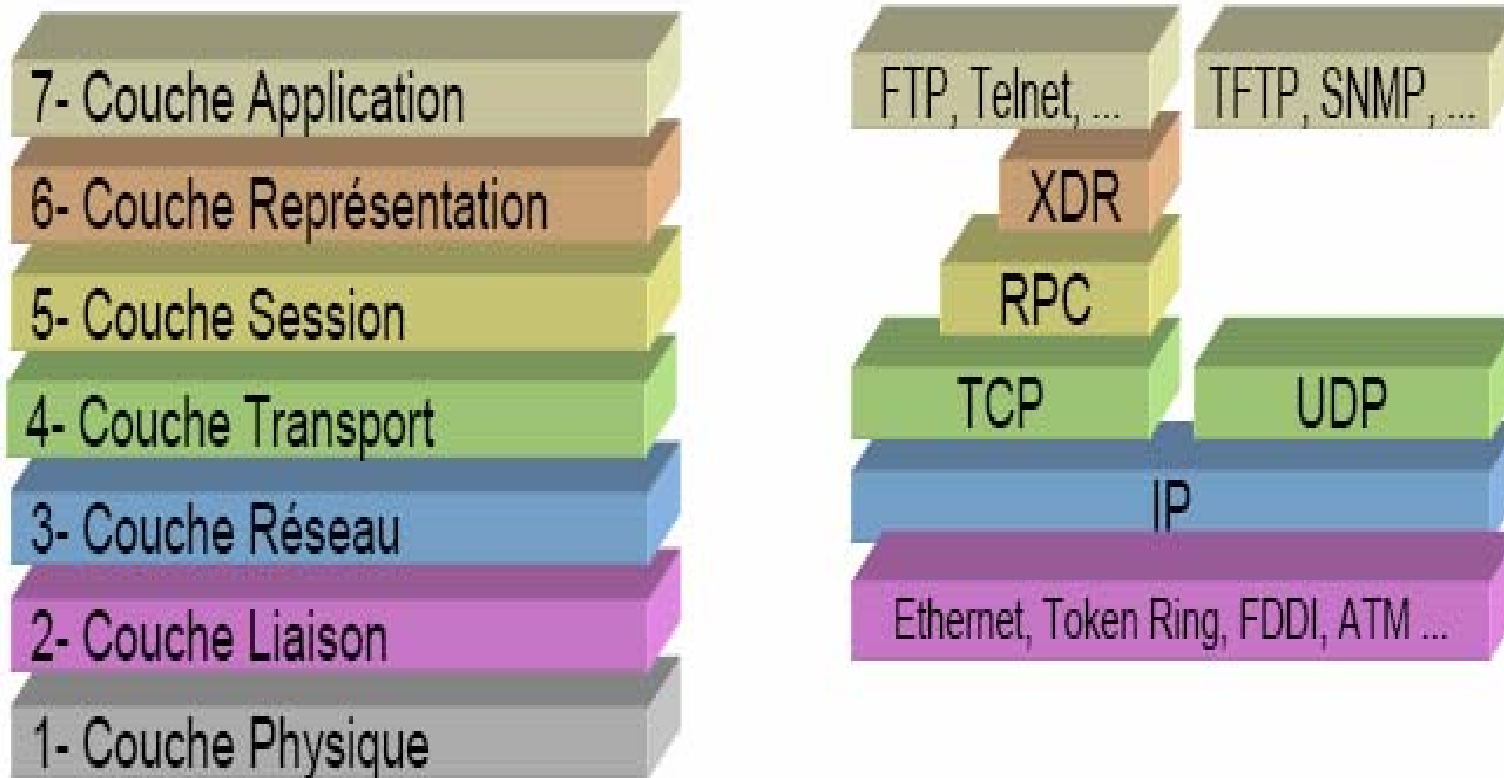
Modèle OSI



Communication dans le modèle OSI



L'implémentation IP

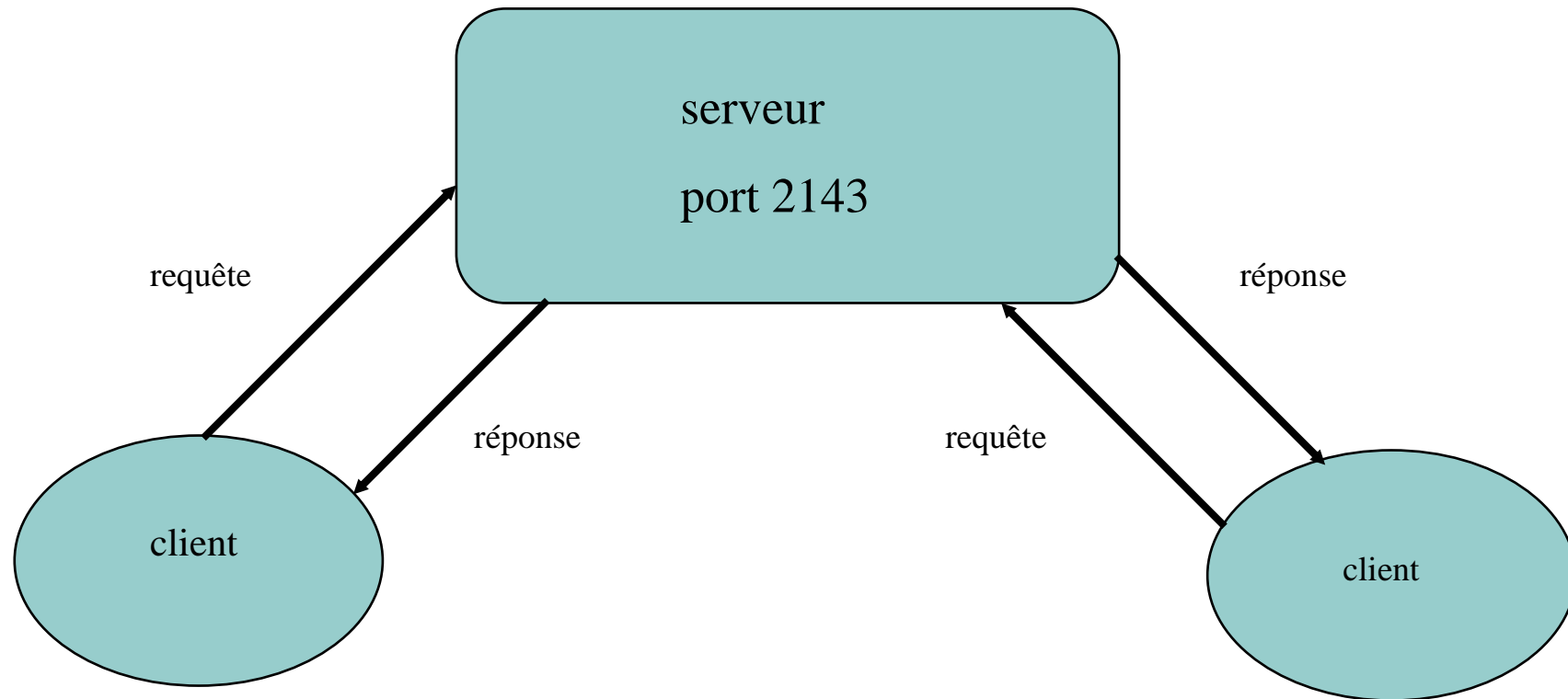


,7

Architecture client/serveur

- On construit, au-dessus des protocoles de transmission, des applications reposant sur un ou plusieurs serveurs et des clients.
- Un serveur propose un ou plusieurs services (par exemple l'heure, le transfert de fichiers, etc.).
- Un client utilise les services proposés.
- Un client émet une requête dans un protocole donné (TCP/IP, UDP/IP) en précisant l'adresse de la machine et le service souhaité: le port de communication.
- Le serveur est à l'écoute de ce port, détecte la requête et fournit la réponse à l'adresse et sur le port du client.

Architecture client/serveur



Allocation des ports

- Un port est un entier sur 16 bits.
- Les ports < 1024 sont privilégiés.
- Certains ports sont alloués à des services reconnus; une liste peut être trouvée dans:
 - */etc/services* sous Unix
 - *C:\WINNT\system32\drivers\etc\services* sous Windows XP

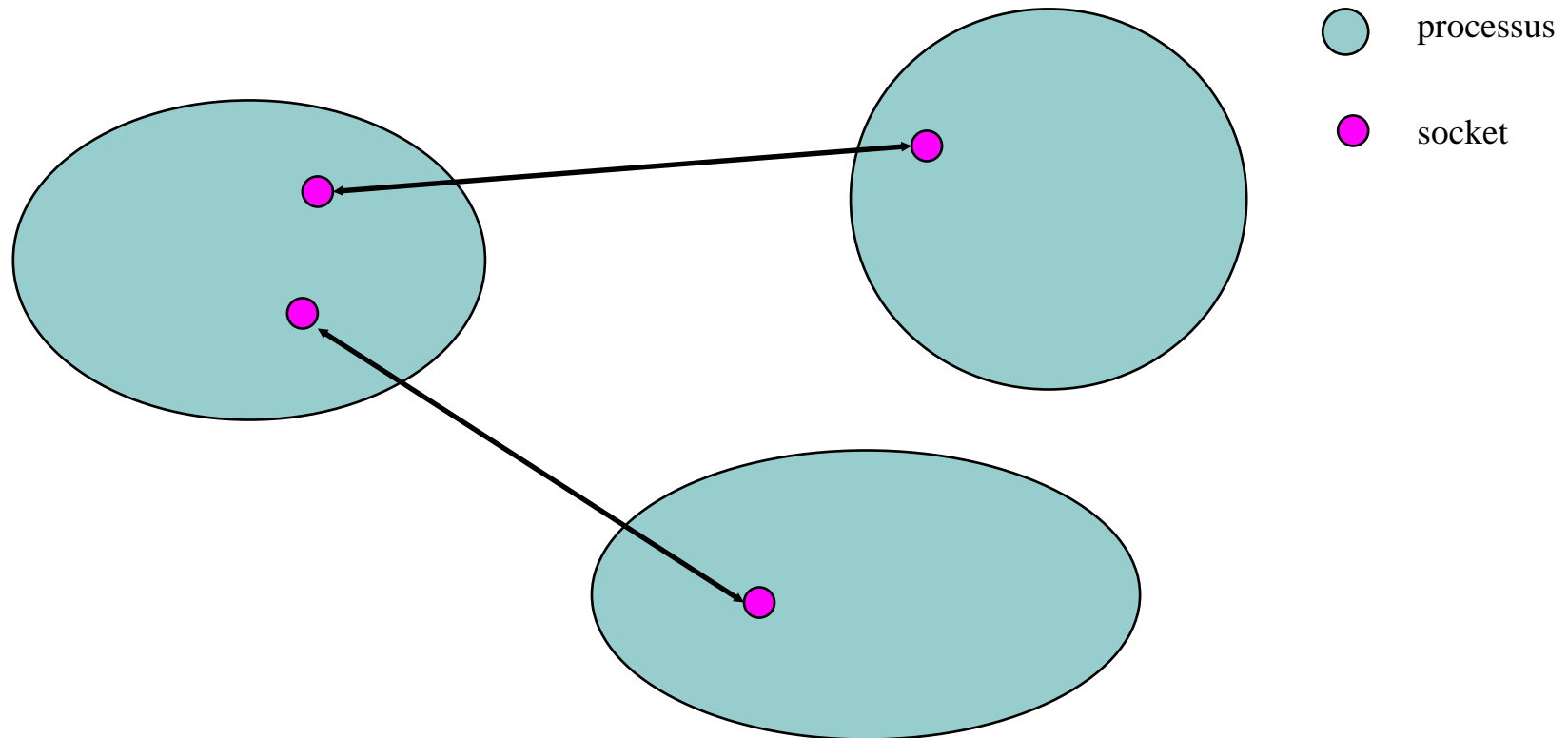
Extrait du fichier "services"

```
echo                7/tcp
echo                7/udp
discard            9/tcp      sink null
discard            9/udp      sink null
systat             11/tcp     users      #Utilisateurs actifs
systat             11/tcp     users      #Utilisateurs actifs
daytime            13/tcp
daytime            13/udp
chargen            19/tcp     ttytst source #Générateur de caractères
chargen            19/udp     ttytst source #Générateur de caractères
ftp-data           20/tcp
ftp                21/tcp     #FTP contrôle
ssh                22/tcp     #SSH Remote Login
telnet             23/tcp
smtp               25/tcp     mail       #Format SMTP
```

Les sockets

- Interface de programmation proposée par l'Unix de Berkeley et implémenté par WinSock de Trumpet pour Windows 3.x
- Une socket peut être vue comme un point de communication entre processus.
- Une socket peut être vue comme un triplé (protocole, adresse, port).
- Les sockets permettent des échanges bidirectionnels entre processus.
- Une socket est considérée comme un descripteur de fichiers par le système d'exploitation.

Processus/Socket



Les fonctions de programmation réseau

- Fonctions sensiblement identiques sous Unix et Windows.
- Sous Windows, l'utilisation d'une socket doit être précédée par un appel à la fonction:

*int WSAStartup (WORD wVersionRequested,
LPWSADATA lpWSAData);*

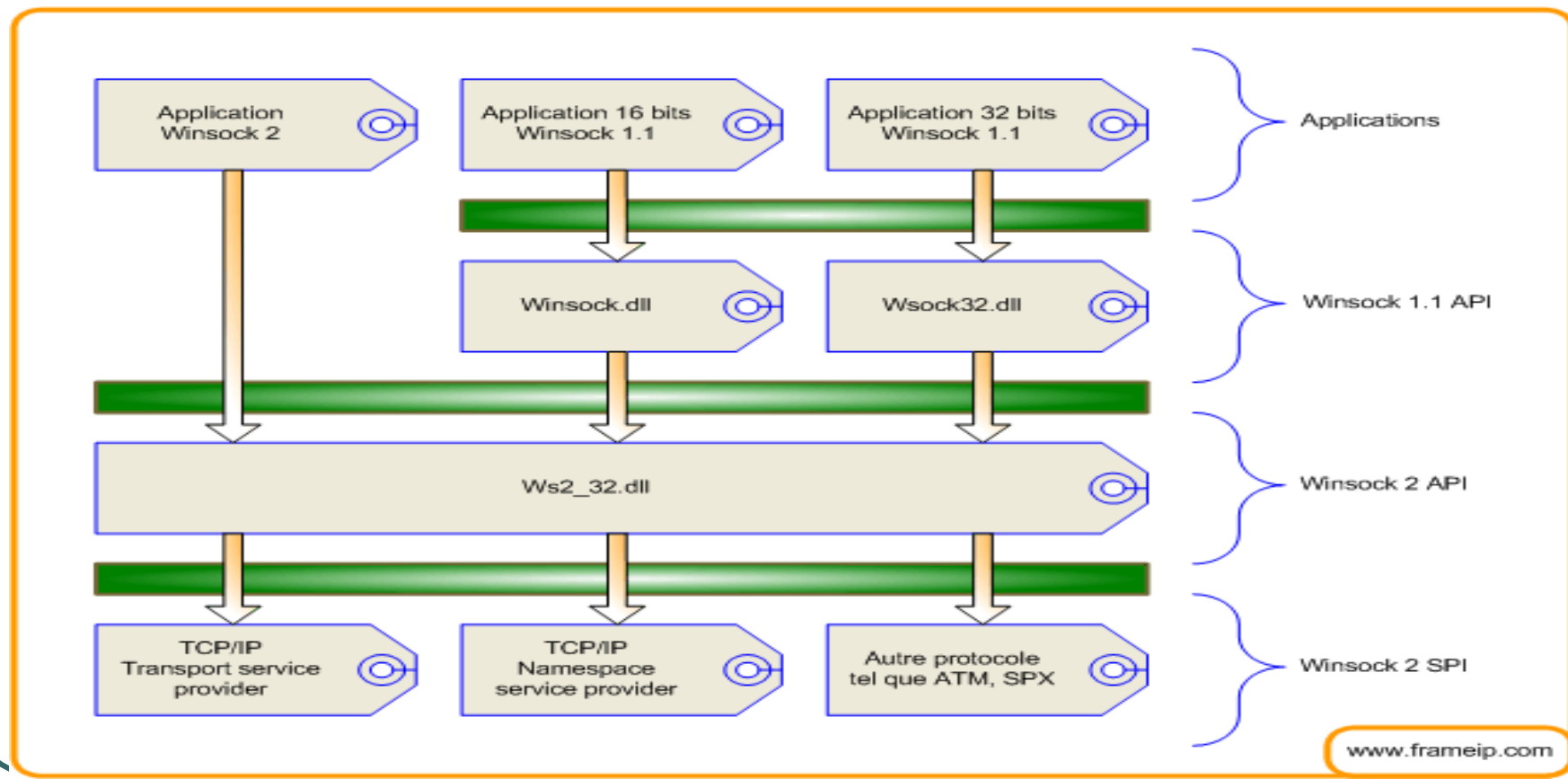
Cela permet d'indiquer quelle version des sockets Windows (winsock) est nécessaire.

Exemple d'appel de WSASStartup

```
WORD wVersionRequested;
WSADATA wsaData;

int err;
wVersionRequested = MAKEWORD( 2, 2 );
err = WSASStartup( wVersionRequested, &wsaData );
if ( err != 0 ) {
    /* Probleme avec la DLL winsock */
    return;
    }
// Utilisation des sockets
// ...
WSACleanup () ; // libération des zones mémoires allouées
```

La compatibilité winsock



Création d'une socket

- **Unix:**

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

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

```
int socket(int domain, int type, int protocol);
```

Retourne -1 en cas d'échec

- **Windows:**

```
#include <windows.h>
```

```
SOCKET socket ( int af, int type, int protocol );
```

Retourne INVALID_SOCKET en cas d'échec

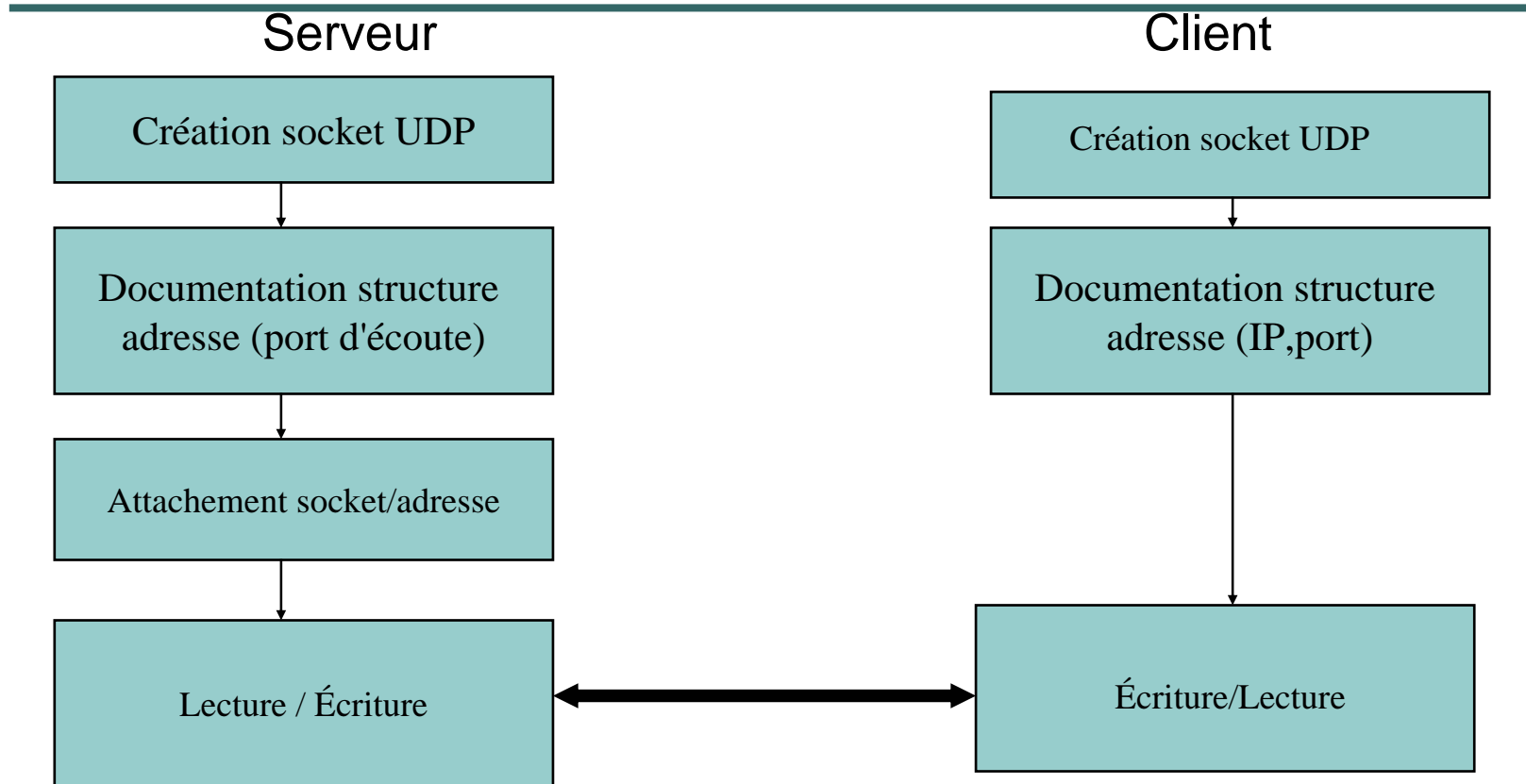
Arguments de la fonction socket

domain	PF_UNIX protocole local PF_INET protocole Internet PF_INET6 protocole Internet IPv6 ...
type	SOCK_STREAM Garantie l'intégrité de la transmission. SOCK_DGRAM Transmission sans connexion, sans garantie de datagrammes de longueur fixe. SOCK_RAW Transmissions internes aux système (nécessite un accès privilégié). ...
protocol	IPPROTO_TCP, IPPROTO_UDP, IPPROTO_RAW En général, il y a un protocole par type et il n'est pas utile de le spécifier. S'il y en a plusieurs, on peut l'indiquer.

Exemples de création de socket

- Socket internet sans connexion (UDP/IP)
socket (PF_INET, SOCK_DGRAM, IPPROTO_UDP)
- Socket internet avec connexion (TCP/IP)
socket (PF_INET, SOCK_STREAM, IPPROTO_TCP)
- Socket de bas niveau:
socket(PF_INET, SOCK_RAW, IPPROTO_RAW)

Communication en mode Internet non connecté



Structure d'adresse

- On manipule des structures d'adresses dont le type générique est: *struct sockaddr*
- Toutes les fonctions manipulant des adresses sont prototypées avec ce type.
- En fonction du domaine, on utilisera des structures d'adresses particulières:
- Dans le domaine Unix (PF_UNIX), on manipule des adresses de type *struct sockaddr_un*
- Dans le domaine Internet (PF_INET), on manipule des adresses de type *struct sockaddr_in*

La structure *struct sockaddr_in*

```
#include <netinet/in.h>
struct sockaddr_in
{
    short sin_family;           /* AF_INET */
    unsigned short sin_port;    /* Port d'écoute ou de connexion */
    struct in_addr sin_addr;    /* INADDR_ANY ou IP de connexion */
    char sin_zero[8];          /* Pas utilisé */
};

struct in_addr { u_long s_addr; };
```

Remplissage du champ *sin_port*

- Attention à la représentation des entiers: utiliser la fonction *htons ()*

Exemple:

```
struct sockaddr_in addr ;  
addr.sin_port = htons (2000) ;
```

- Les fonctions de conversion
unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);

Remplissage du champ *sin_addr.addr*

- Méthode 1

```
addr.sin_addr.addr = inet_addr ("193.49.200.147") ;
```

- Méthode 2

```
struct hostent {  
    char *h_name; /* Nom officiel de la machine */  
    char **h_aliases; /* liste d'alias */  
    int h_addrtype; /* type d'adresse */  
    int h_length; /* longueur de l'adresse */  
    char **h_addr_list; /* liste d'adresses */  
}
```

```
struct hostent *host ;  
host = gethostbyname ("www.ensicaen.fr") ;  
memcpy ( &addr.sin_addr.s_addr,host->h_addr_list [0],host->length) ;
```

Attachement socket/adresse

- Unix :

*int bind(int sockfd, struct sockaddr *my_addr, int addrlen);*

Retourne -1 en cas d'erreur, 0 sinon

- Windows:

int bind (SOCKET s, const struct sockaddr FAR name, int namelen);*

Retourne SOCKET_ERROR en cas d'erreur, 0 sinon

Lecture d'une socket en mode non connecté

- Unix :

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

Retourne: Nombre d'octets lus ou -1 sinon

- Windows:

```
int recvfrom ( SOCKET s, char FAR* buf, int len, int flags,  
              struct sockaddr FAR* from, int FAR* fromlen )
```

Retourne: Nombre d'octets lus ou 0 si la socket a été fermée ou SOCKET_ERROR

Écriture sur une socket en mode non connecté

- Unix :

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

Retourne le nombre de caractères écrits, -1 sinon

- Windows:

```
int sendto ( SOCKET s, const char FAR * buf, int len, int flags,  
const struct sockaddr FAR * to, int tolen );
```

Retourne le nombre de caractères écrits, SOCKET_ERROR sinon

Fermeture d'une socket

- Unix :

int close(int fd);

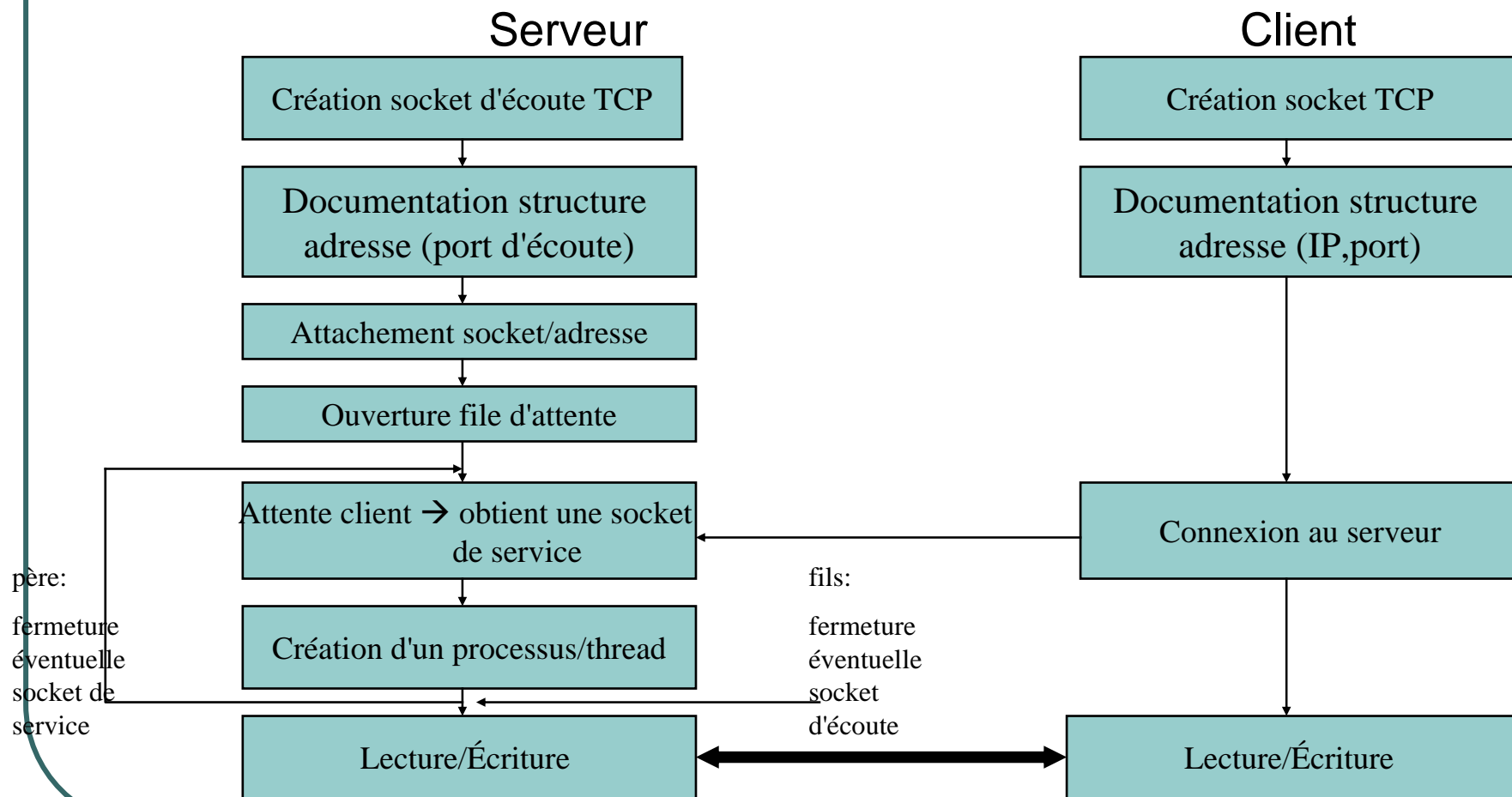
Retourne 0 ou -1 en cas d'échec

- Windows:

int closesocket (SOCKET s);

Retourne 0 ou SOCKET_ERROR en cas d'échec

Communication en mode Internet connecté



Ouverture de la file d'attente en mode connecté

- Unix :

int listen(int s, int backlog);

Retourne 0 ou -1 en cas d'erreur

- Windows:

int listen (SOCKET s, int backlog);

Retourne 0 ou SOCKET_ERROR en cas d'erreur

Attente d'une connexion cliente en mode connecté

- Unix :

*int accept(int s, struct sockaddr *addr, int *addrlen);*

Retourne une socket de service ou -1 en cas d'erreur

- Windows:

SOCKET accept (SOCKET s, struct sockaddr FAR addr, int FAR* addrlen);*

Retourne une socket de service ou INVALID_SOCKET en cas d'échec

Connexion à un serveur en mode connecté

- Unix:

```
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

Retourne 0 ou -1 en cas d'erreur

- Windows:

```
int connect ( SOCKET s, const struct sockaddr FAR* name, int namelen );
```

Retourne 0 ou SOCKET_ERROR en cas d'erreur

Lecture d'une socket en mode connecté

- Unix:

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

Retourne le nombre d'octets lus ou -1 en cas d'erreur

- Windows:

int recv (SOCKET s, char FAR buf, int len, int flags);*

Retourne le nombre d'octets lus ou SOCKET_ERROR en cas d'erreur

Écriture sur une socket en mode connecté

- Unix:

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

Retourne le nombre d'octets écrits ou -1 en cas d'erreur

- Windows:

```
int send ( SOCKET s, const char FAR * buf, int len, int flags );
```

Retourne le nombre d'octets écrits ou SOCKET_ERROR en cas d'erreur

Fermeture d'une socket en mode connecté

- La fonction *close ()* (Unix) ou *closesocket ()* (Windows) ferme la socket.
- Toute écoute de cette socket par un hôte distant (*recv ()*) retournera 0.
- Tout envoi de données sur cette socket (*send()*) retournera -1.
- En mode connecté, il est conseillé d'appeler préalablement la méthode *shutdown ()*:
 - Unix:
int shutdown(int s, int how);
 - Windows:
int shutdown (SOCKET s, int how);

le paramètre *how* peut contenir:

- 0 (SD_RECEIVE) pour interdire la lecture sur cette socket.
- 1 (SD_SEND) pour interdire l'envoi de données sur cette socket.
- 2 (SD_BOTH) pour interdire les deux.

Manipulations simultanées de plusieurs sockets

- Unix/Windows:

```
int select ( int n, fd_set *readfds, fd_set *writefds,  
            fd_set *exceptfds, struct timeval *timeout);
```

```
FD_CLR (int fd, fd_set *set);
```

```
FD_ISSET (int fd, fd_set *set);
```

```
FD_SET (int fd, fd_set *set);
```

```
FD_ZERO (fd_set *set);
```

- 1er argument de *select ()*: majorant des descripteurs sous Unix, ignoré sous Windows
- Valeur de retour de *select ()*:
 - . Nombre de sockets restant dans les ensembles
 - . 0 si la sortie est due au time out
 - . -1 (ou SOCKET_ERROR) en cas d'erreur

Options des sockets

- Unix:

```
int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);  
int setsockopt(int s, int level, int optname, const void*optval, socklen_t optlen);
```

Retourne 0 ou -1 en cas d'erreur

- Windows:

```
int getsockopt ( SOCKET s, int level, int optname, char FAR* optval, int FAR* optlen );  
int setsockopt ( SOCKET s, int level, int optname, const char FAR * optval, int optlen );
```

Retourne 0 ou SOCKET_ERROR en cas d'erreur

Paramètres des fonctions de gestion des options

- *s:*
Socket concernée par l'option
- *level:*
Niveau auquel s'applique l'option: SOL_SOCKET, IPPROTO_TCP, ...
- *optname:*
Valeur de l'option en fonction du niveau choisi.
- *optval:*
Adresse d'une zone mémoire contenant la valeur de l'option.
- *optlen:*
Taille de la zone mémoire contenant la valeur de l'option.

Exemple 1: émission d'un datagramme UDP en broadcast

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
typedef int SOCKET ;
#define SOCKET_ERROR -1
int main ()
{
    SOCKET s ;
    struct sockaddr_in addr,addr_exp ;
    int lg = sizeof (addr) ;
    int value = 1 ;      /* Valeur de l'option broadcast */
    char buffer [1024] ; /* Buffer de réception du datagramme */
    s = socket (PF_INET,SOCK_DGRAM,IPPROTO_UDP) ;
    addr.sin_family = AF_INET ; addr.sin_port = htons (2000) ;
    addr.sin_addr.s_addr = inet_addr ("192.168.16.255") ;
    setsockopt (s,SOL_SOCKET,SO_BROADCAST,(char *)&value,sizeof (value)) ;
    strcpy (buffer,"test broadcast") ;
    sendto (s,buffer,strlen (buffer),0,&addr,lg) ;
}
```

Exemple 2: envoi de paquets en multicast

- **Serveur:**

```
struct ip_mreq mreq;  
mreq.imr_multiaddr.s_addr=mcastAddr.s_addr;  
mreq.imr_interface.s_addr=htonl(INADDR_ANY);  
setsockopt(s,IPPROTO_IP,IP_ADD_MEMBERSHIP, (void *) &mreq, sizeof(mreq));
```

- **Client:**

```
unsigned char ttl = 1;  
setsockopt(sd,IPPROTO_IP,IP_MULTICAST_TTL, &ttl,sizeof(ttl))
```

Exemple 3: Réutilisation forcée d'un port TCP

```
int sockdes ;  
int on = 1 ;  
sockdes=socket(AF_INET,SOCK_STREAM,0);  
setsockopt(sockdes,SOL_SOCKET,SO_REUSEADDR,(char *)&on,sizeof(on)) ;
```

Quelques fonctions utilitaires

- Conversion d'une chaîne de caractère contenant une adresse IP en valeur 32 bits:

*unsigned int inet_addr (char *str) ;*

- Conversion d'une adresse IP 32 bits en chaîne de caractères:

*char *inet_ntoa (struct in_addr ip) ;*

- Obtention des informations sur l'association locale d'une socket:

*int getsockname(int s,struct sockaddr * name ,socklen_t * namelen)*

- Obtention du nom de la machine locale dans *name* contenant au plus *length* caractères:

*int gethostname (char *name,int length) ;*

Quelques fonctions utilitaires

- Lecture non bloquante d'une socket:

```
#include <fcntl.h>
```

```
int optSock
```

```
fcntl(sd, F_GETFL, &optSock); /* Lit options sur la socket */
```

```
optSock |= O_NDELAY;          /* Définit options */
```

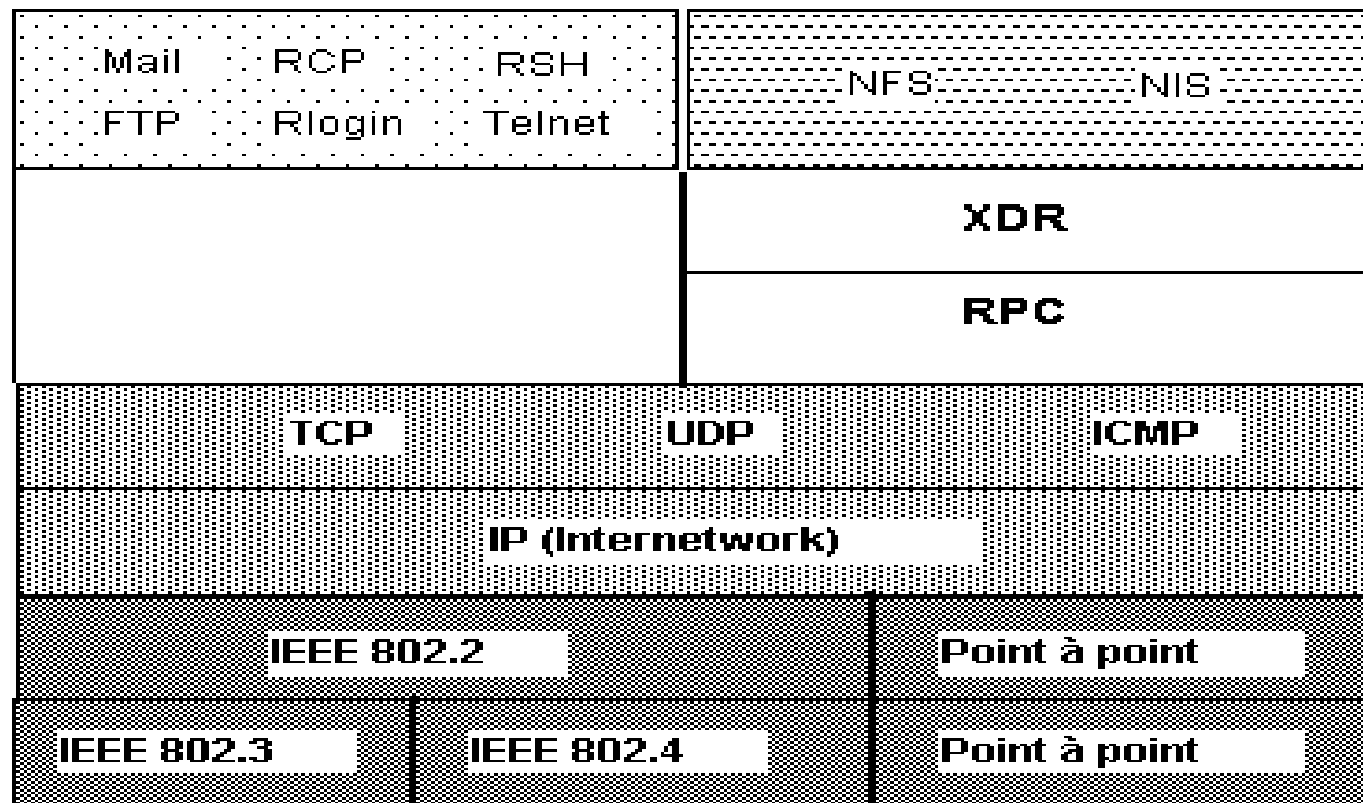
```
fcntl(sd, F_SETFL, &optSock); /* Applique options */
```

Remote Procedure Call

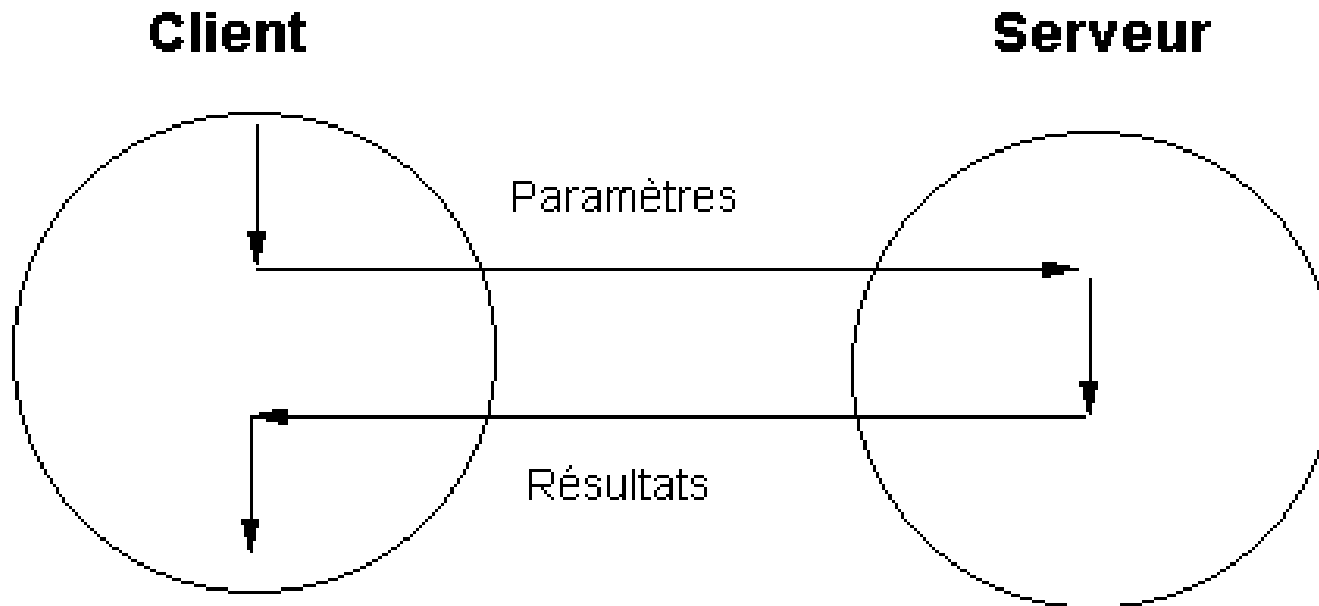
Généralités sur les RPC

- Concepts introduit par SUN MicroSystem pour l'implémentation de la couche NFS (**N**etwork **F**ile **S**ystem).
- Objectif: une application cliente doit pouvoir exécuter une fonction sur une machine distante.
- Le programmeur est déchargé de la gestion des sockets.
- Ce concept d'informatique distribuée a servi de base à la gestion des RMI (**R**emote **M**ethod **I**nvocation), EJB (**E**nterprise **J**ava **B**ean) de Java et à CORBA (**C**ommon **O**bject **R**equest **B**roker **A**rchitecture).

Généralités sur les RPC

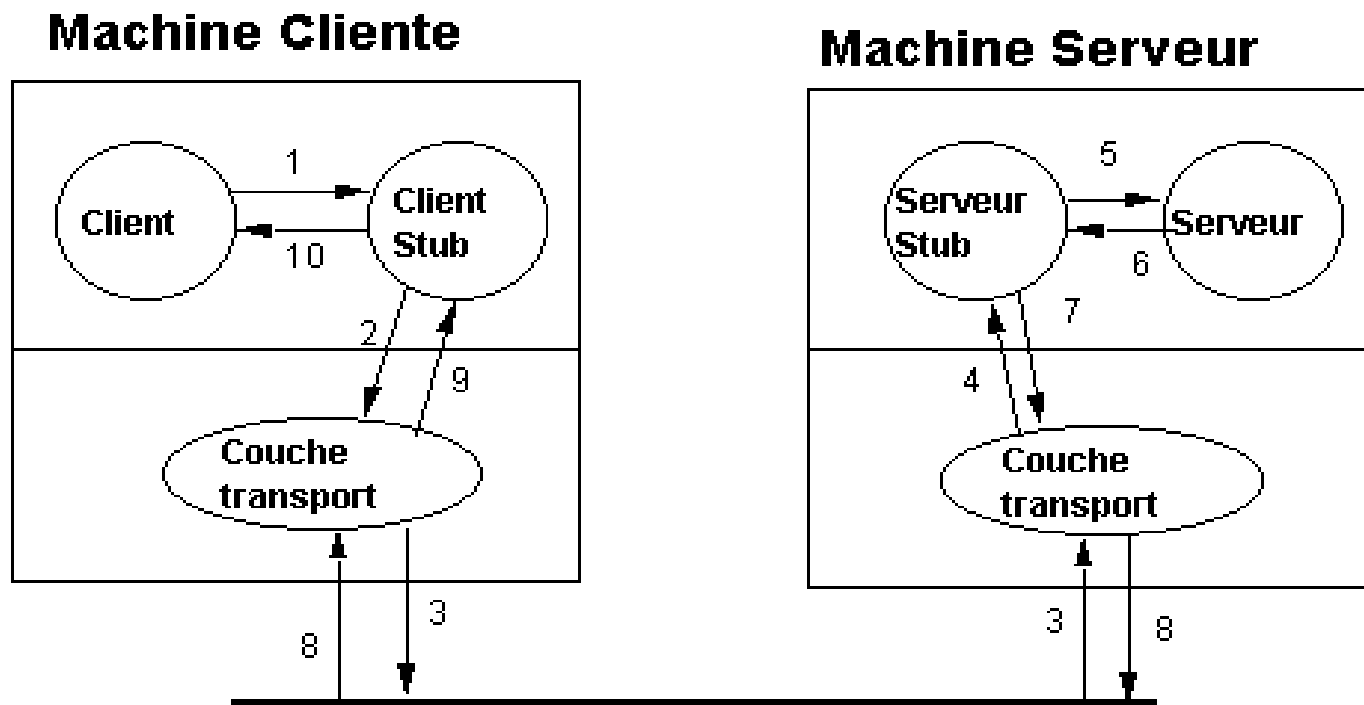


Vision des RPC par le programmeur



result = f(arg)

Vision des RPC par le concepteur



RPC: côté serveur

- Les fonctions sont enregistrées auprès d'un processus portmap (TCP/111).
- Une fonction est caractérisée par un entier long, un numéro de version et un nom.
- Les entiers longs sont répertoriés ainsi (liste dans /etc/rpc sous Unix):

Intervalle	Description
0x00000000 à 0x1fffffff	Réservé (défini par Sun)
0x20000000 à 0x3fffffff	Non réservé
0x40000000 à 0x5fffffff	Réservé (Transient
0x60000000 à 0xffffffff	Réservé

Enregistrement des fonctions

```
int registerrpc (  
    unsigned long prog,  
    unsigned long vers,  
    unsigned long proc,  
    char *( *f ) (),  
    bool_t (*xdr_arg) (),  
    bool_t (*xdr_res)()  
);
```

Utilisation d'une fonction distante

```
int callrpc (  
    char *machine,  
    unsigned long prog,  
    unsigned long vers,  
    unsigned long proc,  
    bool_t (*xdr_arg)(),  
    char *arg,  
    bool_t (*xdr_res)()  
    char *res  
  
    );
```

XDR: Codage des types de données

- Problème principal des rpc: trouver un codage commun de tous les types de données indépendant de la plate forme.
- Une bibliothèque de codage a été créée: XDR (**eXternal Data Representation**).
- Nécessite le fichier d'entête: *#include <rpc/xdr.h>*
- Chaque type scalaire de base dispose d'une fonction de codage de la forme:

bool_t xdr_<type> (XDR xdr, <type>argresp)

Codage XDR des types de base

TYPE C	Fonction XDR associée
char	xdr_char
int	xdr_int
unsigned int	xdr_u_int
short int	xdr_short
unsigned short	xdr_u_short
long int	xdr_long
unsigned long	xdr_u_long
float	xdr_float
double	xdr_double

Codage XDR d'une structure

```
struct complexe
```

```
{
```

```
    float reel,imaginaire ;
```

```
};
```

```
bool_t xdr_complexe (XDR *xdr,struct complexe *ptr)
```

```
{
```

```
    return xdr_float (xdr,&ptr->reel) && xdr_float (xdr,&ptr->imaginaire) ;
```

```
}
```

Exemple RPC: le fichier d'entête

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

```
#include <rpc/xdr.h>
```

```
struct couple
```

```
{
```

```
    float e1,e2 ;
```

```
};
```

```
int xdr_couple () ;
```

```
struct couple *mult (struct couple *) ;
```

Exemple RPC: le serveur

```
#include "exemple.h"
main ()
{
    int rep ;

    rep = registerrpc (0x33333333, 1, 2, mult, xdr_couple, xdr_couple) ;
    if (rep < 0)
    {
        perror ("registerrpc") ;
        exit (1) ;
    }
    svc_run () ;
}
```

Exemple RPC: le client

```
#include <stdio.h>
#include "exemple.h"
int main (int argc, char **argv)
{
    struct couple don, res ;
    if (argc != 2)
    {
        fprintf (stderr, "Usage: call machine\n");
        exit (1);
    }
    for (;;)
    {
        printf ("Entrer 2 reels : "); scanf ("%f%f", &don.e1, &don.e2);
        if (callrpc (argv [1], 0x33333333, 1, 2, xdr_couple, &don, xdr_couple, &res) == 0)
        {
            printf (" %f * %f = %f\n", don.e1, don.e2, res.e1);
            printf (" %f / %f = %f\n", don.e1, don.e2, res.e2);
        }
        else fprintf (stderr, "Erreur\n");
    }
}
```

Exemple RPC: la fonction mult ()

```
#include "exemple.h"  
  
char *mult (struct couple *p)  
{  
    static struct couple res ;  
    res.e1 = p->e1 * p->e2 ;  
    res.e2 = p->e1 / p->e2 ;  
    return ( (char *) &res) ;  
}
```

Exemple RPC: la conversion XDR

```
#include "exemple.h"
```

```
bool_t xdr_couple (XDR *xdrp, struct couple *p)
```

```
{
```

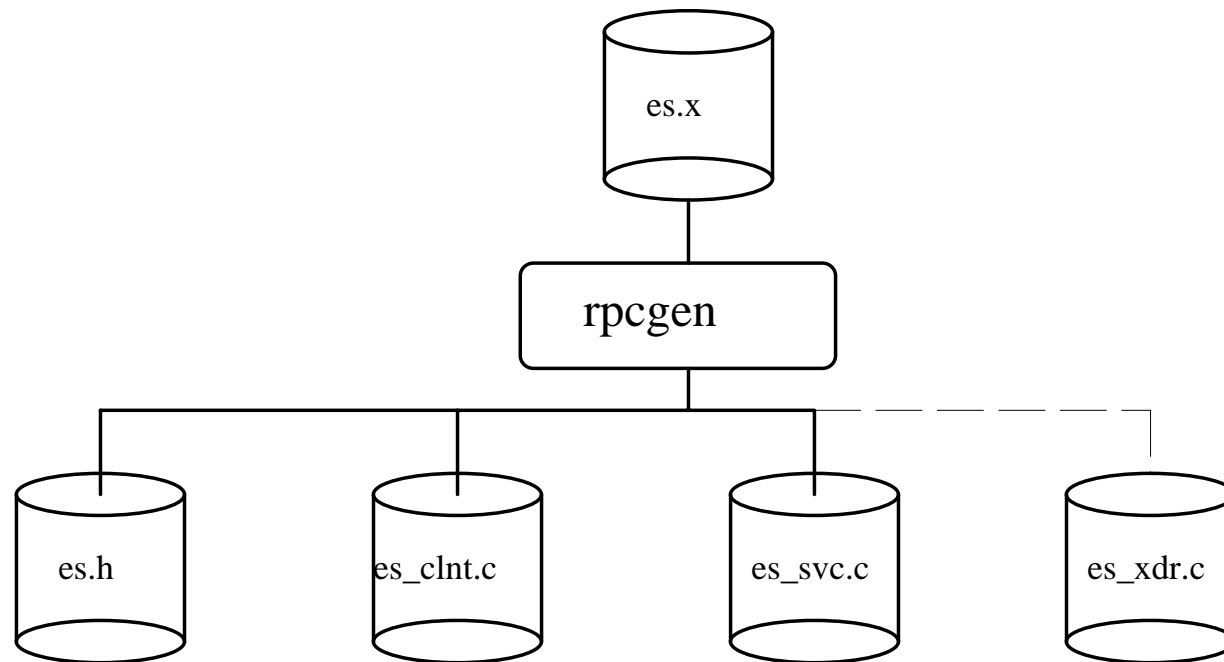
```
    return ( xdr_float (xdrp, &p->e1) && xdr_float (xdrp, &p->e2) );
```

```
}
```

Le langage RPCL

- Création d'un "Remote Procedure Call Language" (RPCL) pour simplifier l'écriture des fonctions XDR.
- C'est un langage de description des types de données à utiliser.
- La commande *rpcgen* va ensuite générer:
 - Toutes les fonctions xdr nécessaires.
 - Les squelettes (côté client et côté serveur) qui vont rendre transparent au programmeur l'aspect réseau et les conversions xdr.

Principe de fonctionnement



Syntaxe RPCL

- Le langage RPCL consiste en une série de définitions pouvant être de 6 types différents :
 - déclaration de constantes
 - déclaration de types énumérés
 - déclaration de structures (identique au C)
 - déclaration d'unions
 - déclaration de nouveaux types (typedef, identique au C)
 - déclaration de programme

Écriture du serveur

- Les règles à observer pour l'écriture des fonctions du serveur sont les suivantes :
 - Les noms de fonctions (définies dans *es.x*) doivent être suivies du caractère "_" et du numéro de version; Si par exemple on a défini dans *es.x* une fonction *gettime* correspondant à la version 1, le nom de cette fonction dans le fichier *proc.c* sera *gettime_1*.
 - Les fonctions n'admettent qu'un seul paramètre qui est un *pointeur* sur le type déclaré dans le fichier *es.x*.
 - La valeur de retour des fonctions est également un *pointeur* sur le type déclaré dans le fichier *es.x*.

Écriture du client

```
#include <rpc/rpc.h>
#include "es.h"
void main (int argc, char **argv)
{
    CLIENT *cl ;
    /* server est une chaîne de caractère contenant le nom de la machine distante
    Le protocole est soit "udp", soit "tcp" */
    cl = clnt_create (server,....PROG,....VERS,protocole) ;
    if (cl == NULL)
    {
        clnt_pcreateerror (server) ;
        exit (1) ;
    }
    Appel de la fonction distante avec 2 paramètres :
    - adresse sur le type déclaré
    - variable cl
}
```

Syntaxe RPCL

Constantes

const <identificateur> = <integer> ;

Exemple : const MAX=8192 ;

Types énumérés

enum <identificateur>

{

enum-value-list

}

où enum-value-list représente :

enum-value , **enum-value-list**

où enum-value représente

identificateur = valeur

Syntaxe RPCL: la clause *program*

```
program <identificateur>
{
    version <identificateur>
    {
        type nom_fonction (type) = <valeur> ;
        ...
    } = valeur ;
} = valeur ;
```

Exemple :

```
program TIMEPROG
{
    version TIMEVERS
    {
        unsigned int TIMEGET (void) = 1 ;
        void TIMESET (unsigned) = 2 ;
    } = 1 ;
} = 0x2000000 ;
```

Exemple: le fichier RPCL: add.x

```
struct data
{
    unsigned int arg1;
    unsigned int arg2;
};
typedef struct data data;
struct reponse
{
    unsigned int somme;
    int errno;
};
typedef struct reponse reponse;
program CALCUL
{
    version VERSION_UN
    {
        void CALCUL_NULL(void) = 0;
        reponse CALCUL_ADDITION(data) = 1;
    } = 1;
} = 0x20000001;
```

Génération des différents fichiers

- La commande "*rpcgen -a add.x*" génère les fichiers:
 - *add.h* fichier d'entête à laisser inchangé
 - *add_client.c* programme client, **à compléter**
 - *add_clnt.c* squelette client à laisser inchangé
 - *add_server.c* fonctions distantes, **à compléter**
 - *add_svc.c* squelette serveur, à laisser inchangé
 - *add_xdr.c* codage xdr des types utilisés, à laisser inchangé

Exemple d'écriture d'une fonction

```
reponse *calcul_addition_1(data *argp, struct svc_req *rqstp)
{
    static reponse result;
    result.somme = argp->arg1 + argp->arg2 ;
    return (&result);
}
```

Exemple: la fonction à enregistrer

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

```
char **essai_1 (char **chaine)
```

```
{
```

```
    static char *buffer ;
```

```
    if (buffer == NULL)
```

```
    {
```

```
        buffer = calloc (10,sizeof (char)) ;
```

```
    }
```

```
    strncat (buffer,"Bonjour\n",8) ;
```

```
    return &buffer ;
```

```
}
```

Exemple: le client

```
#include <rpc/rpc.h>
#include "es.h"
void main (int argc, char **argv)
{
    CLIENT *cl ;
    char *buffer ;
    char **result ;
    buffer = (char *) calloc (20, sizeof (char)) ;
    cl = clnt_create ("e3000", MESSAGEPROG, MESSAGEVERS, "tcp") ;
    if (cl == NULL)
    {
        perror ("Connexion client") ;
        exit (1) ;
    }
    strncpy (buffer, "essai ", 5) ;
    result = essai_1(&buffer) ;
    printf ("%s\n", *result) ;
}
```

La bibliothèque libsdl

Bibliothèque SDL

- Tentative d'uniformiser les API Windows/Linux sur les thèmes:
 - audio
 - vidéo
 - threads
 - timers
 - ...
- Logiciels domaine public: <http://www.libsdl.org>

Exemple programmation thread

```
#include <stdio.h>
#include "SDL.h"
#include "SDL_thread.h"
#include "SDL_mutex.h"
int potty = 0;
int fin;
SDL_mutex *lock;
int thread_func(void *data)
{
    int num = (int) data;
    int compteur = 0;
    while ( fin == 0 ) {
        printf ("Thread %d, je prends le mutex\n",num) ;
        SDL_mutexP(lock);
        printf ("Thread %d, j'ai pris le mutex\n",num) ;
        SDL_Delay (1000) ;
        printf ("Thread %d, je rends le mutex\n",num) ;
        SDL_mutexV(lock); compteur += 1 ;
    }
    printf("Fin Thread %d\n",num);
    return(compteur);
}

int main (int argc,char **argv)
{
    const int progeny = 5;
    SDL_Thread *kids[progeny];
    int i, lots;
    lock = SDL_CreateMutex();
    for ( i=0; i<progeny; ++i ) {
        kids[i] = SDL_CreateThread(thread_func, (void *)i);
    }
    SDL_Delay(5*1000);
    SDL_mutexP(lock);
    printf("On arrete les threads\n");  fin = 1;
    SDL_mutexV(lock);
    for ( i=0; i<progeny; ++i ) {
        SDL_WaitThread(kids[i], &lots);
        printf("Thread %d a utilise le mutex %d fois\n", i,
lots+1);
    }
    SDL_DestroyMutex(lock);
    return 0 ;
}
```

Exemple d'utilisation d'un mutex

```
#include <stdio.h>
#include "SDL.h"
#include "SDL_thread.h"
#include "SDL_mutex.h"
int potty = 0;
int gotta_go;
int thread_func(void *data)
{
    SDL_mutex *lock = (SDL_mutex *)data;
    int times_went = 0 ;
    while ( gotta_go ) {
        SDL_mutexP(lock); ++potty;
        printf("Thread %d using the potty\n", SDL_ThreadID());
        if ( potty > 1 ) {
            printf("potty already used!\n");
        }
        -- potty; SDL_mutexV(lock); ++times_went;
    }
    printf("ok\n");
    return(times_went);
}

int main (int argc,char **argv)
{
    const int progeny = 5;
    SDL_Thread *kids[progeny];
    SDL_mutex *lock;
    int i, lots;
    lock = SDL_CreateMutex();
    gotta_go = 1;
    for ( i=0; i<progeny; ++i ) {
        kids[i] = SDL_CreateThread(thread_func, lock);
    }
    SDL_Delay(5*1000); SDL_mutexP(lock);
    printf("Everybody done?\n"); gotta_go = 0;
    SDL_mutexV(lock);
    for ( i=0; i<progeny; ++i ) {
        SDL_WaitThread(kids[i], &lots);
        printf("Thread %d used the potty %d times\n", i+1,
lots);
    }
    SDL_DestroyMutex(lock); return 0 ;
}
```

Programmation réseau IPv6

La programmation IPv6

- L'interface de programmation IPv6 est définie par les RFC 2553 et 2292.
- Ce qui a changé:
 - Les structures d'adresse.
 - L'interface socket.
 - Les fonctions de conversions entre noms et adresses.
 - Les fonctions de conversion d'adresses.

Les structures d'adresses

- Une nouvelle famille d'adresses *AF_INET6* et une nouvelle famille de protocole *PF_INET6* ont été définies dans *<sys/socket.h>*.

- Pour des raisons de compatibilité, ces deux constantes sont égales:

```
#define PF_INET6 AF_INET6
```

- Une adresse IPv6 est contenue dans la structure:

```
struct in6_addr  
{  
    uint8_t s6_addr[16];    /* IPv6 address */  
};
```

Les structures d'adresses

- La structure *in6_addr* peut également être définie par:

```
struct in6_addr
{
    union
    {
        uint8_t _S6_u8[16];
        uint32_t _S6_u32[4];
        uint64_t _S6_u64[2];
    } _S6_un;
};

#define s6_addr _S6_un._S6_u8
```

La structure *sockaddr_in6*

- Structure analogue à la structure *sockaddr_in* d'IPv4.
- Il existe deux implémentations de cette structure:
 - Implémentation d'Unix 4.3BSD:

```
struct sockaddr_in6
{
    sa_family_t sin6_family;    /* AF_INET6 (champ sur 16 bits) */
    in_port_t sin6_port;       /* transport layer port # */
    uint32_t sin6_flowinfo;    /* IPv6 traffic class & flow info */
    struct in6_addr sin6_addr;  /* IPv6 address */
    uint32_t sin6_scope_id;    /* set of interfaces for a scope */
};
```

La structure *sockaddr_in6*

- Implémentation d'Unix 4.4BSD:

```
struct sockaddr_in6
{
    uint8_t sin6_len; /* length of this struct */
    sa_family_t sin6_family; /* AF_INET6 (champ sur 8 bits)*/
    in_port_t sin6_port; /* transport layer port # */
    uint32_t sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr; /* IPv6 address */
    uint32_t sin6_scope_id; /* set of interfaces for a scope */
};
```

- Les systèmes qui implémentent la version 4.4BSD définissent une constante *SIN6_LEN* dans le fichier d'entête *<netinet/in.h>*.

L'adresse "wildcard"

- En IPv4, la constante `INADDR_ANY` est utilisée par une application pour laisser au système le choix de l'adresse source.
- En IPv6, il y a deux possibilités:
 - Utilisation d'une constante au moment de la déclaration:

```
struct in6_addr anyaddr = IN6ADDR_ANY_INIT;
```
 - Utilisation d'une variable externe:

```
extern const struct in6_addr in6addr_any;  
sin6.sin6_addr = in6addr_any;
```

L'adresse de "loopback"

- En IPv4, l'adresse de "loopback" est définie par la constante `INADDR_LOOPBACK`.

- En IPv6, il y a deux possibilités:

- Utilisation d'une constante au moment de la déclaration:

```
struct in6_addr loopbackaddr = IN6ADDR_LOOPBACK_INIT;
```

- Utilisation d'une variable externe:

```
extern const struct in6_addr in6addr_loopback;  
sin6.sin6_addr = in6addr_loopback;
```

L'interface socket

- La fonction `socket ()` utilise le même schéma qu'en IPv4:
`sock = socket (PF_INET6,SOCK_DGRAM,0) ;`
`sock = socket (PF_INET6,SOCK_STREAM,0) ;`
- Les autres fonctions de l'interface socket sont inchangées, moyennant le transfert d'une adresse de type *struct sockaddr_in6*.
- Principales fonctions inchangées:
`bind ()`, `connect ()`, `send ()`, `sendto ()`, `accept ()`, `recv ()`, `recvfrom ()`,
`getsockname ()`, `getpeername ()`.

Les fonctions de conversion entre noms et adresses

- Les fonctions *gethostbyname ()*, *gethostbyaddr ()*, *getservbyname ()*, *getservbyport ()* ont été remplacées par deux fonctions: *getaddrinfo ()* et *getnameinfo ()*:

```
int getaddrinfo ( const char *nodename, const char *servname,  
                 const struct addrinfo *hints, struct addrinfo **res);
```

```
int getnameinfo ( const struct sockaddr *sa, socklen_t salen,  
                 char *host, size_t hostlen, char *serv, size_t servlen, int flags);
```

- Les structures *addrinfo* sont allouées dynamiquement; il faut les désallouer après utilisation:

```
void freeaddrinfo(struct addrinfo *ai);
```

La structure "addrinfo"

```
struct addrinfo
{
    int ai_flags; /* AI_PASSIVE, AI_CANONNAME, AI_NUMERICHOST */
    int ai_family; /* PF_xxx */
    int ai_socktype; /* SOCK_xxx */
    int ai_protocol; /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
    size_t ai_addrlen; /* length of ai_addr */
    char *ai_canonname; /* canonical name for nodename */
    struct sockaddr *ai_addr; /* binary address */
    struct addrinfo *ai_next; /* next structure in linked list */
};
```

Les fonctions de conversion numériques d'adresses

- Les deux fonctions *inet_addr ()* et *inet_ntoa ()* convertissent des adresses IPv4 entre une représentation binaire et une représentation textuelle.
- Les fonctions équivalentes pour IPv6/IPv4 sont:

```
int inet_pton(int af, const char *src, void *dst);
```

```
const char *inet_ntop(int af, const void *src, char *dst, size_t size);
```

- Ces fonctions ne sont pas supportées par winsock; on peut utiliser en remplacement *WSAAddressToString()* et *WSAStringToAddress ()*.

Exemple de conversion d'adresse

```
#include <stdio.h>
#include <sys/socket.h>
#include <netdb.h>

int main (int argc, char **argv)
{
    struct addrinfo hints ;
    struct addrinfo *res ;
    char str_addr [INET6_ADDRSTRLEN] ;

    memset (&hints, 0, sizeof (struct addrinfo)) ;
    hints.ai_family = PF_INET6 ;
    if (getaddrinfo ("www.crihan.fr", NULL, &hints, &res) == 0)
    {
        struct in6_addr *src = &((struct sockaddr_in6 *) res->ai_addr)->sin6_addr ;
        if (inet_ntop (AF_INET6, src, str_addr, sizeof (str_addr)) != 0)
        {
            printf ("Adresse IPv6 = %s\n", str_addr) ;
        }
        freeaddrinfo (res) ;
    }
    else printf ("Adresse non resolue\n") ;
}
```

Résultat affiché:

Adresse IPv6 = ::2001:660:7401:211:0:0

Même exemple sous Windows

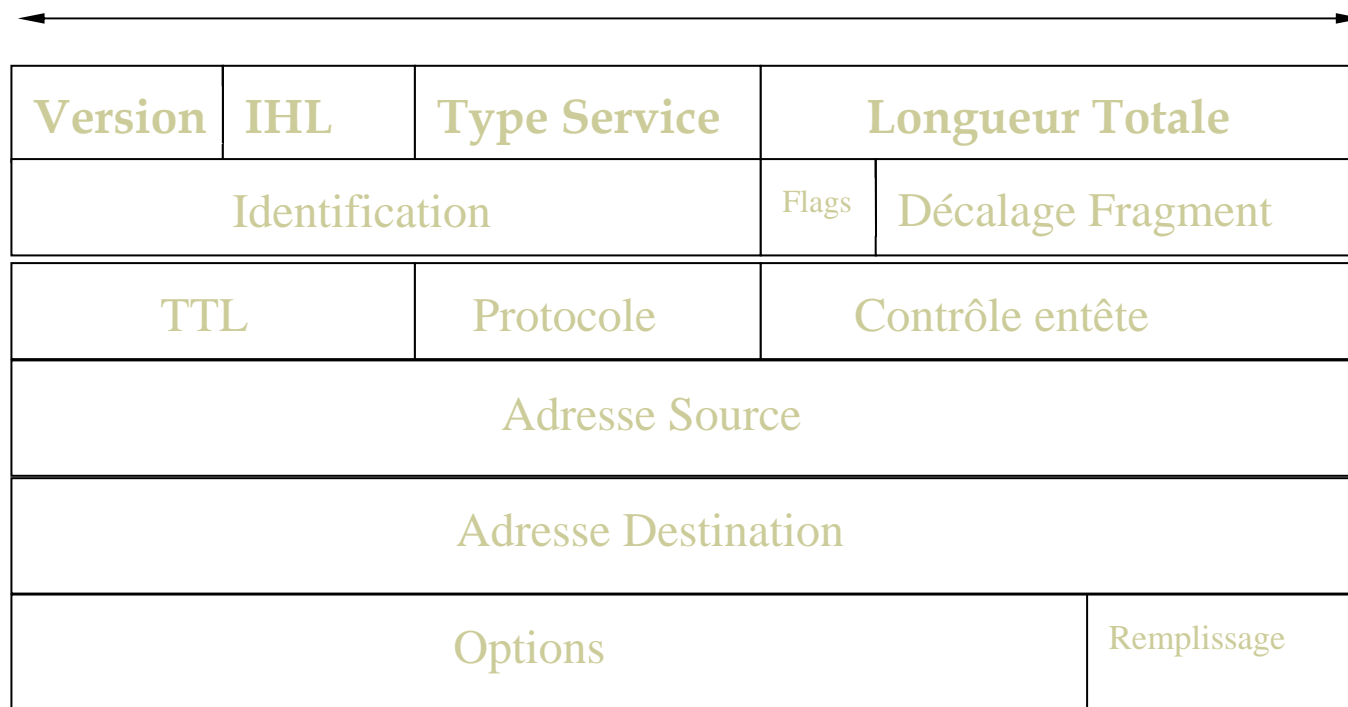
```
#include <stdio.h>
#include <winsock2.h>
#include <ws2tcpip.h>
int main (int argc, char **argv [])
{
    WSADATA wsadata ;

    if (WSAStartup (MAKEDWORD (2,2),&wsadata) != 0) { fprintf (stderr,"Erreur dll\n") ; return 1 ; }
    struct addrinfo hints ; struct addrinfo *res ;
    memset (&hints,0,sizeof (struct addrinfo)) ;
    hints.ai_family = PF_INET6 ; hints.ai_socktype = SOCK_STREAM ;
    if (getaddrinfo ("www.crihan.fr",NULL,&hints,&res) == 0)
    {
        char dest [INET6_ADDRSTRLEN] ;
        DWORD lg_dest = INET6_ADDRSTRLEN ;
        if (WSAAddressToString (res->ai_addr,res->ai_addrlen,NULL,dest,&lg_dest) == 0)
        {
            printf ("adresse IPv6 = %s\n",dest) ;
        }
        freeaddrinfo (res) ;
    }
    WSACleanup () ;
}
```

Annexes

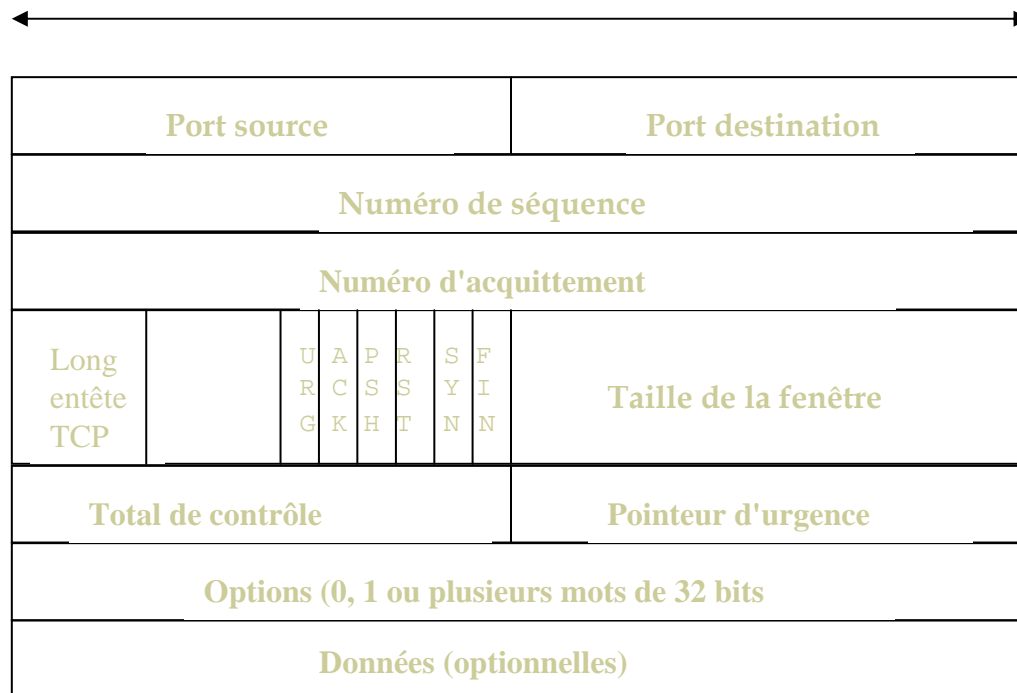
Entête IP

32 bits



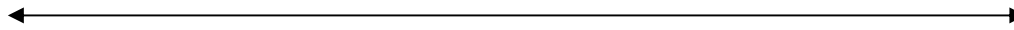
Entête TCP

32 bits



Entête UDP

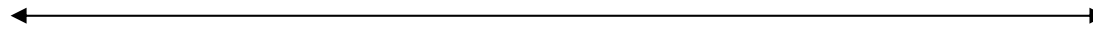
32 bits



Port source	Port destination
Longueur UDP	Total de contrôle UDP

Entête IPv6

32 bits



Ver sion	Pri	Etiquette de flot		
Longueur de charge utile		En-tête suivant	Nb max de sauts	
Adresse source (16 octets)				
Adresse destination (16 octets)				