Système d'exploitation

V. Parallélisme et synchronisation

Kévin Perrot

Aix-Marseille Université

2014

Ce cours est inspiré (entre autres) des supports de Fabien Rico et du livre de Jean-Marie Rifflet.

Table des matières

1 Introduction

Ce cours présente comment un processus peut en créer d'autres, et comment ils peuvent partager des ressources et communiquer. Rappelons que les processus sont de

deux types : lourds (processus) et légers (threads). Les idées présentées s'appliqueront aussi bien aux processus lourds que légers, nous indiquerons lors des présentations des solutions techniques si elles s'appliquent aux processus ou aux threads. Presque toutes les implémentations en C qui seront présentées font partie de la norme POSIX (*Portable Operating System Interface*).

La simultanéité dans l'exécution de plusieurs processus (lourds ou léger) peut être *réelle* (sur plusieurs processeurs), ou réalisée par *commutation* (alternance sur un seul processeur). Les problèmes de *synchronisation* apparaissent sous deux formes :

— lors du partage de ressources (comment assurer que deux processus ne modifient pas les mêmes données en même temps ? et si une ressources peut être utilisée par au plus n processus en même temps ?),

— lors de la communication (comment un processus peut transmettre le résultat de son exécution à un autre ? comment faire en sorte que l'autre soit prêt à recevoir l'information ?).

2 Processus `init`

Rappelons rapidement la séquence de démarrage d'une machine :

1. BIOS : POST et recherche du MBR sur un périphérique (disques, lecteur cd, usb).
2. Le Master Boot Record pointe vers le chargeur d'amorçage (*boot loader*), qui propose à l'utilisateur de choisir le système d'exploitation à démarrer.
3. La main est donnée au système d'exploitation.

Le premier processus lancé au démarrage du système d'exploitation (par le noyau) est `init`, il a l'identifiant 1. C'est un processus démon (*deamon*) qui continue de s'exécuter jusqu'à l'extinction du système. Il est l'ancêtre (direct ou indirect) de tous les processus. En effet, tous les processus suivants seront créés avec l'appel système `fork` (ou par d'autres commandes qui utilisent `fork`). Comme nous le verrons, `fork` a pour effet de créer un nouveau processus, fils du processus courant. L'ensemble des processus du système forme ainsi une *arborescence*, dont `init` est la racine.

3 Création de processus : `fork`

Chaque processus a un identifiant unique, le `pid` (*process identifier*). Sur tous les systèmes de type Unix il s'agit d'un entier, mais la norme POSIX définit tout de même le type `pid_t` pour les `pid` (au cas où ce soit une chaîne de caractères par exemple), pour nous il se manipulera donc comme un entier long (`long int`). Un processus peut accéder à son `pid` et à celui de son père (`ppid`) grâce aux deux fonctions suivantes.

```
#include <unistd.h>
pid_t getpid(void); // identité du processus
pid_t getppid(void); // identité du processus père
```

Le type `pid_t` est déclaré dans la librairie `sys/types.h`.

3.1 `Fork`

La primitive `fork` permet la création dynamique d'un nouveau processus, qui s'exécute de façon concurrente avec le processus qui l'a créé. Son prototype est le suivant.

```
#include <unistd.h>
pid_t fork(void);
```

L'appel à cette fonction créé un nouveau *processus fils*. La réalisation de cette opération (en mode noyau sous Unix), attribue un bloc de contrôle (*control bloc*) et initialise le nouveau processus. C'est une copie exacte du *processus père* pour une grande part de ses attributs. Le processus fils exécutera le même code que le processus père sur une copie des données de ce dernier au moment de l'appel. Une fois le nouveau processus créé, tout se passe comme si les processus père et fils avaient tous les deux réalisé un appel à `fork` : chacun reprendra son exécution au moment du retour de l'appel à `fork` et continuera à exécuter le même code, mais sur des données indépendantes. Pour que les comportements des deux processus ne soit pas identiques, il est nécessaire de pouvoir les distinguer, cela est réalisé par un test sur la valeur retournée par la primitive `fork`, qui vaut

— `0`, dans le processus fils,

— le `pid` du processus fils créé, dans le processus père.

En cas d'échec de création d'un nouveau processus, la valeur `-1` est retournée.

3.2 Ce qui n'est pas hérité

Le processus fils hérite de tous les attributs du processus père, sauf

— le `pid` et le `ppid`,

— les temps d'exécutions sont initialisés à la valeur nulle,

— les signaux pendants ne sont pas hérités (voir Section 7.1),

— la priorité du fils est initialisée à une valeur par défaut,

— les verrous détenus par le père ne sont pas hérités (voir Section 6.2).

Les processus fils reçoivent en particulier une copie des descripteurs de fichiers du processus père, qui pointent sur les mêmes entrées de la table des descripteurs de fichiers (ouverts). Ainsi, toute opération sur ces fichiers réalisée par l'un des deux aura des répercutions sur l'autre (par exemple le déplacement de la position courante, qui est mémorisée dans la table des descripteurs de fichiers).

3.3 *Copy on write*

Le processus fils travail sur une copie des données du père, mais ces données (les *pages* correspondantes) ne sont effectivement copiées qu'à partir de l'instant où elles sont modifiées, pour alléger le mécanisme de création d'un processus : c'est le *copy on write*.

3.4 Zombis et orphelins

Tout processus qui se termine passe dans l'état *zombi*, où il reste tant que son père n'a pas pris connaissance de sa terminaison. Le but de cet état est de permettre au père de récupérer le code de retour (*exit status*) du fils, qui attend donc dans l'état zombi que son

père le lui demande. Ce mécanisme est très similaire à un appel de fonction classique, qui retourne une valeur, sauf qu'ici l'appel n'est pas bloquant : le père continue de s'exécuter en parallèle de son fils. Pour récupérer le code de retour de ce dernier, et donc effectuer une *synchronisation*, deux primitives sont offertes aux pères.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int* pointeur_status);
pid_t waitpid(pid_t pid, int* pointeur_status, int options);
```

Sémantique de `wait` :

— si le processus appelant n'a aucun fils, alors la valeur `-1` est retournée ;

— si le processus appelant a au moins un fils zombi, alors le `pid` d'un de ceux-ci est retournée (il est au passage tué), et si l'adresses `pointeur_status` est différente de `NULL` alors la valeur `*pointeur_status` fournit des informations sur la terminaison du processus zombi ;

— si le processus appelant a des fils mais aucun fils zombi, il est bloqué jusqu'à ce que l'un de ses fils devienne zombi (il peut aussi y avoir interruption de l'appel par un signal « non mortel », dans ce cas la valeur de retour est `-1`).

Pour choisir quel zombi on souhaite tuer, il y a la primitive `waitpid`, dont le paramètre `pid` permet de sélectionner le processus attendu de la manière suivante :

pid	Interprétation
`<-1`	tout processus fils dans le groupe `|pid|`
`-1`	tout processus fils
`0`	tout processus fils du même groupe que l'appelant
`>0`	processus fils d'identité `pid`

Le paramètre `option` permet par exemple d'indiquer que l'appel n'est pas bloquant (avec la valeur `WNOHANG`). La fonction peut maintenant renvoyer `0` si l'appel est non bloquant et que le processus demandé existe mais n'est pas un zombi.

Si un père meurt (se termine) avant son fils, ce dernier restera zombi pour toujours ? Non, le processus `init` est un tueur de zombis : il adopte automatiquement tous les orphelins et dès leur terminaison il attend le zombi correspondant.

3.5 Avec recouvrement : `execvp`

L'appel système `execvp` ne créé pas de nouveau processus : il permet de remplacer le code du processus courant, on parle de *recouvrement*. Cet appel permet d'exécuter un autre code (une commande Bash ou un autre programme), précisé en argument.

```
#include <unistd.h>
execvp(const char* commande, const char* arguments[]);
```

Exemple d'utilisation : `execvp("ls",{"ls","-l","-a",NULL})`;.

Il existe des variantes de `execvp`, la particularité de cette dernière est qu'elle va chercher la commande dans la liste des dossiers contenus dans la variable `PATH` (comme le ferait l'interprète de commande).

4 Processus et threads

Lors d'un appel à `fork`, un nouveau processus fils est créé, qui mène une existence indépendante : la seule façon de remarquer si l'on est dans le processus père ou fils est de tester la valeur de retour de la commande `fork`, car toutes les données du processus sont copiées.

En revanche, lorsqu'un processus créé un thread, il partage sa mémoire virtuelle avec le processus qui l'a créé, ce qui se traduira pour nous par le fait que les variables globales (qui ne sont pas définies à l'intérieur d'une fonction) sont partagées. La pile d'appel n'est pas partagée, ce qui veut dire pour nous que le thread et son processus parent peuvent faire des appels de fonctions sans interférer l'un avec l'autre.

Important : un thread exécute une fonction (voir Section 5).

Un processus est composé de *threads*, qui sont également appelés *processus légers*, car la commutation de contexte entre threads d'un même processus est plus rapide que la commutation de contexte entre processus.

Autre différence notoire : lorsqu'on tue un processus, on tue en même temps tous les threads qu'il a créés, alors qu'on peut tuer indépendamment le père et le fils issus d'un appel à `fork`.

5 Threads (librairie `pthread`)

Un processus créé un thread avec l'appel à la fonction suivante.

```
#include <pthread.h>
int pthread_create(
  pthread_t* thread, // résultat : le thread créé
  pthread_attr_t* attr, // les attributs de création ou NULL
  void* (*start_routine)(void*), //la fonction que le thread exécute
  void* arg); // l'argument de la fonction
```

Le nouveau thread va alors exécuter la fonction `start_routine` sur l'argument `arg` (si l'on veut passer plusieurs arguments, il faut utiliser une structure). Nous utiliserons toujours les attributs par défaut : `NULL`. Cet appel retourne `0` en cas de succès, et un code d'erreur en cas d'échec. Pour préciser la fonction, il faudra que celle-ci ait le type de retour `void*` et un unique paramètre de type `void*`, et si elle s'appelle `fonction_exemple` alors nous donnerons l'argument `&fonction_exemple` (son adresse) pour l'appel à `pthread_create` (ou l'on transtype : `(void*(*)(void*))fonction_exemple`).

Un thread peut se terminer de son propre gré si il appel la fonction

```
#include <pthread.h>
void pthread_exit(void* retval);
```

où `retval` est la valeur de retour (à laquelle le processus père peut accéder, pour retourner plusieurs variables il faut passer par une structure). Sa terminaison peut également être forcée par son processus père si ce dernier appel la fonction

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
```

où `thread` indique le thread à terminer. Cet appel retourne `0` en cas de succès, et un code d'erreur en cas d'échec.

Pour récupérer la valeur retournée par un thread, le processus père a à sa disposition la fonction

```
#include <pthread.h>
int pthread_join(
  pthread_t thread, // le thread à attendre
  void** retval); // valeur de retour
```

dont l'appel est bloquant. `thread` indique le thread à attendre. Encore une fois, pour retourner plusieurs variables il faut passer par une structure. Cet appel retourne `0` en cas de succès, et un code d'erreur en cas d'échec.

En ce qui concerne les arguments qui sont de type `void*`, il est nécessaire de les transtyper (caster) pour les exploiter. Par exemple, si `retval` était de type `int*` (il est maintenant de type `void*`), alors on le placera dans la variable `mon_entier` de type `int*` avec l'instruction

```
mon_entier = (int*) retval.
```

6 Exclusion mutuelle et section critique

Lorsque plusieurs processus accèdent à une même donnée, par exemple une variable globale partagée par deux threads, il faut mettre en place un mécanisme d'*exclusion mutuelle* pour éviter les comportements indéterminés du type :

— deux threads écrivent en même temps dans une même variable,

— un thread écrit dans une variable en même temps qu'un autre thread lit sa valeur.

Une ressource non partageable est dite *critique*. Une séquence de programme qui utilise une ressource critique est appelée *section critique*. L'exclusion mutuelle consiste à empêcher deux processus (ou thread) d'être simultanément en section critique. On supposera toujours qu'un processus qui entre en section critique en sort au bout d'un temps fini (pas de blocage).

La mise en place de l'exclusion mutuelle passe par trois sections de code : initialisation (prépare les variables qui indiquent si la section critique est libre), prologue (demande bloquante d'entrée en section critique), épilogue (sortie de la section critique, en autorisant l'accès aux autres).

⟨`initialisation`⟩

⟨`prologue`⟩
⟨`section critique`⟩
⟨`épilogue`⟩

6.1 Atomicité

Pour réaliser naïvement l'exclusion mutuelle, une première solution consisterait à utiliser une variable booléenne partagée indiquant si la ressource critique est libre ou non.

```
⟨initialisation⟩        libre := vrai

       ⟨prologue⟩  (1)  tant que (libre == faux) faire
                   (2)  fin faire
                   (3)  libre := faux
⟨section critique⟩      ...
       ⟨épilogue⟩       libre := faux
```

Dans ce cas, il peut y avoir un problème lorsque

1. un processus `A` est en section critique
2. deux processus `B` et `C` attendent pour entrer en section critique (boucle `(1)(2)`)
3. le processus `A` sort de la section critique
4. `B`, qui attendait, passe la boucle (test `(1)` faux)
5. `C` passe également la boucle (test `(1)` faux), avant que
6. `B` met la variable `libre` à faux (instruction `(3)`)
7. `C` met aussi la variable `libre` à faux (instruction `(3)`)
8. `B` et `C` sont tous les deux en section critique !!

Le problème vient du fait que le test de la variable `libre` et sa modification sont deux opérations séparées.

Il est possible de remédier à ce problème avec un algorithme un peu plus complexe, comme l'*algorithme de Peterson* (1981) qui utilise une variable partagée indiquant à quel processus c'est le tour (chacun tente de laisser le tour aux autres) et un tableau de booléens (une case pour chaque processus, indiquant s'il demande d'entrer en section critique ou non).

La solution que nous retiendrons, qui est également la plus répandue car la plus efficace, consiste en l'introduction au niveau matériel d'une instruction *Test and Set* (TAS), qui rend *atomique* le test d'une variable et sa modification : elle ne peut pas être interrompue, ces deux opérations n'en forment qu'une. C'est le principe des *verrous*.

6.2 Thread `mutex`

Un *mutex* est un *verrou* de synchronisation, qui permet

— de vérifier qu'on est autorisé à passer ;

— d'interdire le passage aux autres.

Ces deux actions forment *une seule instruction atomique*.

Les verrous sont implémentées dans la bibliothèque `pthread.h` avec le type `pthread_mutex_t`. Le terme *verrou* indique qu'au plus un processus peut entrer en section critique à la fois, et le terme *sémaphore* désigne une compteur utilisé dans le cas plus général où au plus n processus peuvent entrer simultanément en section critique (la lecture/modification du sémaphore est protégée par un verrou).

6.2.1 Verrou

Nous allons voir l'implémentations des verrous pour les threads avec les `mutex` de la bibliothèque `pthread.h`. Il y a deux façons d'initialiser un `mutex` (verrou), mais l'idée est toujours la suivante : le processus déclare une variable `mutex` globale et l'initialise dans le `main` avant la création des threads ; de cette façon tous les threads partagerons le `mutex`.

```
#include <pthread.h>
//statique
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
//dynamique
int pthread_mutex_init(
  pthread_mutex_t* mutex, // le mutex à initialiser
  const pthread_mutex_attr_t* mutexattr); // attributs spéciaux ou NULL
int pthread_mutex_destroy(pthread_mutex_t* mutex);
```

Ces fonctions retournent `0` en cas de succès, et un code d'erreur en cas d'échec. La méthode statique permet de définir rapidement un mutex avec un comportement standard, qui est équivalent à l'attribut `NULL`. Les autres attributs servant à décrire le comportement d'un thread demandant un mutex qu'il possède déjà, par défaut l'appel suspend le thread pour toujours.

Deux façons de prendre un `mutex` : demande bloquante (*attente passive*) ou non bloquante (*attente active*) ; et une façon de rendre un `mutex`.
Important : un thread en section critique bloque les autres, il faut donc minimiser la durée du séjour.

```
#include <pthread.h>
// prendre le verrou (appel bloquant si le verrou n'est pas libre)
int pthread_mutex_lock(pthread_mutex_t* mutex);
// prendre le verrou (appel non bloquant)
int pthread_mutex_trylock(pthread_mutex_t* mutex);
// rendre le verrou
int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

Ces fonctions renvoient `0` en cas de succès, et un code d'erreur en cas d'échec. La valeur de retour a une importance particulière pour `pthread_mutex_trylock` car elle indique si le `mutex` était libre et a été pris : si et si seulement si l'appel retourne `0`.

Si plusieurs threads sont en attente, qui est réveillé ? L'OS choisit aléatoirement le thread a qui sera attribué le `mutex`.

Si un thread qui possède déjà un mutex souhaite en prendre un autre qui n'est pas encore libre, peut-on arriver à une situation interbloquée ? Oui ! Pour éviter cela, on introduit les *variables de condition* : si je suis en section critique et que j'attends une condition, alors durant mon attente je libère le `mutex` que je détiens. Les variables de condition résolvent des problèmes d'atomicité similaires à ceux résolus par les `mutex`. Attention, les conditions sont abstraites : un thread dit « j'attends la condition `toto` » et un autre thread va dire « la condition `toto` est vérifiée », mais la condition en question n'est pas précisée : c'est une *condition abstraite* qui est gérée par les thread, et non par un test. Elles ont le type `pthread_cond_t` et son initialisées comme suit.

```
#include <pthread.h>
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Un thread en section critique attend une condition avec les appels aux fonctions suivantes.

```
#include <pthread.h>
int pthread_cond_wait(
  pthread_cond_t* cond, // la condition à attendre
  pthread_mutex_t* mutex); // le mutex libéré
int pthread_cond_timedwait(...) // idem avec un délai maximum
```

Le `mutex` libéré est repris (avec possiblement une attente) lorsque la condition est vérifiée. L'appel retourne `0` en cas de succès, et un code d'erreur en cas d'échec. Pour réveiller un thread en attente, on appellera les fonctions ci-dessous.

```
#include <pthread.h>
// réveil d'un thread
int pthread_cond_signal(pthread_cond_t* cond);
// réveil de tous les threads (un par un)
int pthread_cond_broadcast(pthread_cond_t* cond);
```

6.2.2 Sémaphore

Le *sémaphore* est un compteur utilisé dans le cas plus général où au plus n processus peuvent entrer simultanément en section critique. Il est

- initialisé avec une valeur strictement positive ;
- décrémenté lors d'une entrée en section critique ;
- bloquant quand sa valeur est nulle ;
- incrémenté lors d'une sortie de section critique.

Sa lecture et sa modification sont protégées par un `mutex` (verrou). Il est utile pour une ressource que n processus peuvent utiliser simultanément.

Le nombre n peut également être infini, le compteur indique alors combien de processus sont en section critique (l'utilisation du compteur est inversée). Ce cas est par exemple utile au problème de lecteur/rédacteur : une ressource peut être lue par un nombre indéfini de processus en même temps, mais ne peut être modifiée que par un seul processus à la fois et si personne n'est en train de la lire.

6.2.3 Moniteur

Le moniteur généralise le concept de primitives de synchronisation : il reprend les concepts de la programmation orientée objet et encapsule l'utilisation des verrous et des conditions. Dans le cas du sémaphore, on le déclare puis on l'utilise à l'aide des méthodes :

- `debut_lecture(moniteur);`
- `fin_lecture(moniteur);`
- `debut_ecriture(moniteur);`
- `fin_ecriture(moniteur);`

Une fois le code écrit, on peut ainsi se resservir du moniteur très facilement !

6.3 Famine et interblocage

Attention : interblocage, famine et favoritisme (non équité) sont possibles si l'on ne prend pas garde ! Dans la conception des programmes, il faut essayer de réduire au minimum nécessaire la taille/durée des zones d'exclusion mutuelle.

7 Communication

On peut imaginer des communications entre thread à l'aide des section critiques : par exemple avec un tableau dont l'accès est protégé par un mutex, que chacun peut lire/écrire pour communiquer avec les autres. Voyons d'autres façons plus simples de communiquer, qui fonctionnent également entre processus.

7.1 Signaux

Les signaux sont le mode de communication inter-processus le plus rudimentaire. On a une vingtaine de *signaux* différents, chacun ayant une signification particulière. Quelques exemples :

Nom du signal	Evénement associé
`SIGINT`	frappe du caractère `intr` au clavier (`Ctrl-c`)
`SIGFPE`	erreur arithmétique (division par zéro, ...)
`SIGKILL`	signal de terminaison (imposée)
`SIGSEGV`	violation mémoire
`SIGTERM`	signal de terminaison (demandée)
`SIGCHLD`	terminaison d'un fils
`SIGSTOP`	signal de suspension
`SIGCONT`	signal de continuation d'un processus stoppé

Un processus peut également envoyer des signaux à lui-même. Le comportement par défaut à la réception d'un signal est la plupart du temps la terminaison du processus (c'est notamment le comportement non modifiable lors de la réception d'un signal `SIGKILL`, en revanche par défaut le signal `SIGCHLD` est ignoré).

Il y a trois phases dans la transmission d'un signal :

1. le signal est *envoyé* par un processus,
2. le signal est *pendant* tant qu'il n'a pas été pris en compte par le destinataire,
3. le signal est *délivré* lorsqu'il est pris en compte.

La commande Bash pour envoyer des signaux est `kill <pid>` (par défaut c'est le signal `SIGTERM` qui est envoyé, des options permettent de préciser le signal).

En C, la commande s'appelle également `kill` :

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

où `pid` précise l'identifiant du processus destinataire et `sig` le type de signal.

Pour changer le comportement d'un processus à la réception d'un signal, on utilise un *handler* qui précise la fonction à exécuter en cas de réception d'un signal donné.

7.2 Tubes

Les *tubes* sont des fichiers gérés en mode FIFO, destinés à la communication entre processus ou threads. Ils ont deux entrées dans la table des fichiers ouverts : une extrémité pour écrire, l'autre pour lire (chaque processus peut alors avoir un descripteur qui pointe vers chacune des entrées de cette table). La création d'un tube avant un appel à `fork` permet ainsi à un père et son fils de communiquer !

Attention les tubes ont une capacité finie, et l'opération de lecture est destructive. On utilise les fonctions suivantes pour créer, écrire et lire dans un tube.

```
#include <unistd.h>
// création
int pipe(int p[2]); // p[1] est l'entrée et p[0] la sortie
// écriture
ssize_t write(int entree_tube, const void* chaine, size_t nb_octets);
// lecture
ssize_t read(int sortie_tube, voir* chaine_lue, size_t nb_octets);
```

où les `void*` seront pour nous des chaines de caractères (`char*` ou `char[]`), et nb_octets indique le nombre maximum de caractères à écrire ou lire. Les fonctions `read` et `write` renvoient le nombre d'octets (de caractères) lus.

Il est important de fermer le côté du tube non-utilisé (`p[0]` pour l'écrivain, `p[1]` pour le lecteur), afin que le système sache quand envoyer un caractère de fin de fichier (`EOF`).

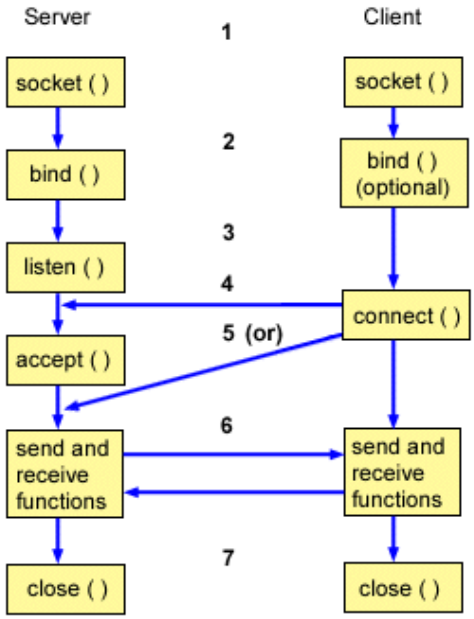
```
#include <unistd.h>
int close(int descripteur);
```

7.3 Sockets

Les tubes permettent aux processus d'une même machine de communiquer. Pour communiquer d'une machine à l'autre, on utilise les *sockets*, qui permettent de passer par le réseau. Il y a plusieurs façons d'utiliser les sockets (qui sont créées avec la fonction `socket` de la librairie `sys/socket.h`) :

- Mode connecté ou non (la connexion est maintenue ou réétablie entre chaque message),
- Mode paquet ou flux (les messages sont envoyés par morceaux ou comme un seul message en continu).

Le schéma ci-contre présente le mécanisme général de communication par socket, avec les (nombreux) appels de fonction nécessaires.



(source : http://www-01.ibm.com/support/knowledgecenter/ssw_ibm_i_71/rzab6/howdosockets.htm)