Reconfigurable & embedded Digital Systems

HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

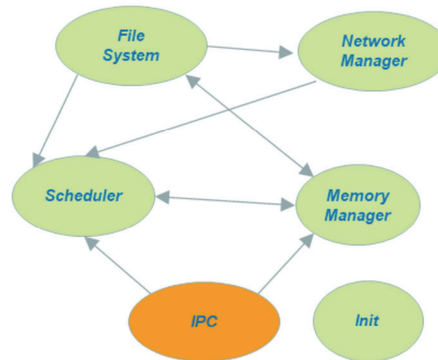
Systèmes d'exploitation
IPCs

Prof. Daniel Rossier, Prof. Alberto Dassatti

Version 3.4 (2017-2018)

Plan

- Introduction
- Tubes
- Signaux
- Fichiers mappés
- *Sockets* réseau
- Segments de mémoire partagée
- Objets de synchronisation



Introduction (1/3)

- *Inter-Process Communication*
- Ensemble de mécanismes permettant la **communication** et la **synchronisation** entre processus.
- Les IPCs ont des interfaces standardisées.
  - Standard POSIX

3 Cours SYE - Institut REDS/HEIG-VD

On se rappelle que les processus disposent de leur propre espace d'adressage et qu'ils ne peuvent pas *s'interférer* mutuellement (par principe de sécurité). Lorsque l'on utilise des *threads* à l'intérieur d'un processus, ces derniers ayant accès au même espace d'adressage, ils peuvent sans autre se synchroniser et accéder à des variables partagées. Pour que les processus puissent s'échanger des données ou se synchroniser (opération copier-coller entre *Word* et *Powerpoint* par exemple), il est nécessaire de recourir à des mécanismes de communication implémentés dans le noyau. Ces mécanismes sont appelés IPCs (*Inter-Process Communication*).

Les IPCs sont généralement coûteux en terme de temps CPU et de ressources mémoire. C'est pourquoi ils font constamment l'objet d'amélioration et d'optimisation au sein des OS et au niveau des librairies utilisateur. Bien entendu, POSIX propose toute une gamme de spécifications dédiées aux IPCs.

Introduction (2/3)

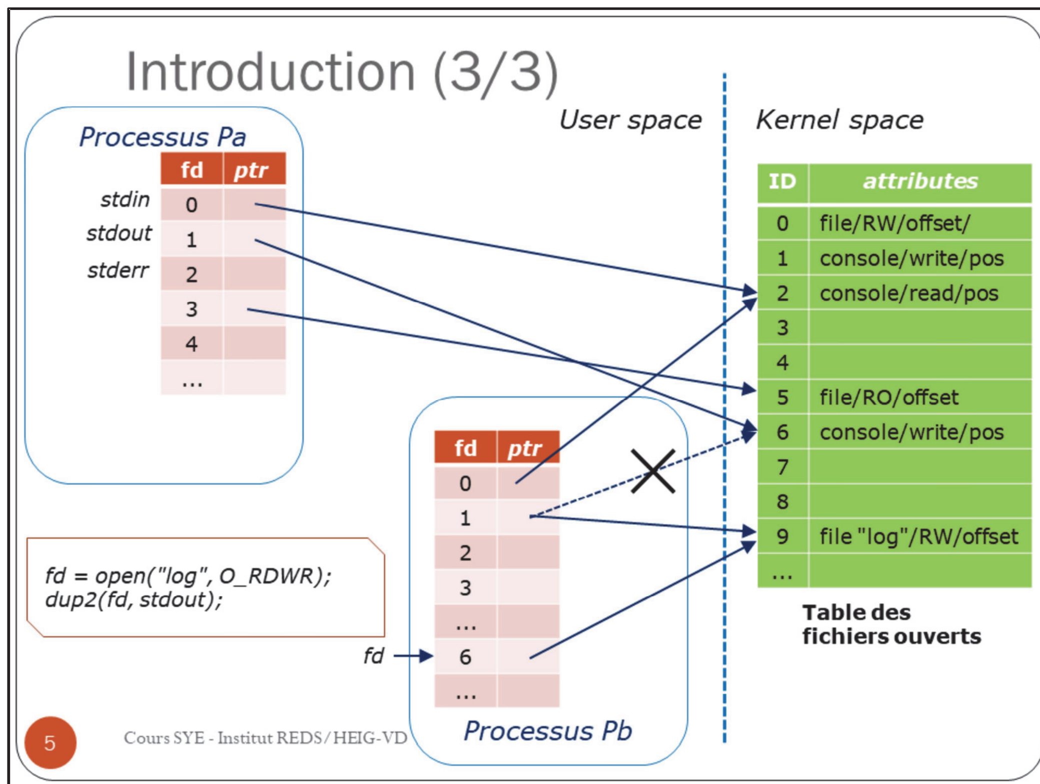
- Descripteurs de fichier (*file descriptor*)
  - **Identifiant** (nombre entier) **unique** vers une ressource de type fichier typiquement
  - Un tableau de descripteurs **par processus**
  - Référence vers **une table des *fichiers ouverts*** dans l'espace noyau
- Redirection (*synonyme*) d'un descripteur
  - Changement de référence vers la table des fichiers ouverts
  - Appel système ***dup2(int orig, int copy)***

4 Cours SYE - Institut REDS/HEIG-VD

Lors de l'ouverture d'un fichier (en lecture ou en écriture), l'appel système *open()*, typiquement utilisé à cette fin, retourne un entier appelé *descripteur de fichier* ou ***file descriptor*** en anglais. Il s'agit d'un identifiant **unique** au niveau du processus qui sera utilisé à chaque fois qu'il sera nécessaire de référencer cette ressource de type fichier.

Chaque processus dispose d'une table de descripteurs contenant l'ensemble des références vers les fichiers ouverts. Comme nous le verrons dans ce chapitre, il peut s'agir de références vers des ressources autres que des fichiers, comme des tubes par exemple.
Cette table de descripteurs est utilisée pour stocker les références vers une autre table entièrement gérée dans l'espace noyau : la ***table des fichiers ouverts***. Dans cette table, le noyau conserve l'état des ressources (comme la position courante dans un fichier par exemple) qui sont manipulées par l'espace utilisateur au travers des descripteurs de fichier. L'utilité de cette table est facile à comprendre; si deux processus manipulent le même fichier (en lecture), la lecture effectuée dans le premier processus n'aura, fort heureusement, **aucune incidence** sur la lecture effectuée par le second processus.

L'appel système ***dup2()*** permet de manipuler la référence d'un descripteur vers la table des fichiers ouverts de telle sorte que ce descripteur pointe vers une entrée utilisée par un autre descripteur. On parle alors de ***redirection*** (ou ***synonyme***) de descripteur. La page suivante montre un tel scénario.

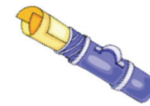


Ainsi, chaque processus dispose de sa propre table de descripteurs de fichier. Dans cette table, on trouve notamment trois descripteurs (les trois premiers) qui ont une signification particulière : il s'agit de ***stdin*** (associé au descripteur no. 0), ***stdout*** (associé au no. 1) et ***stderr*** (associé au no. 3). Ces trois descripteurs sont utilisés respectivement pour *l'entrée standard*, la *sortie standard* et une sortie réservée pour les messages d'erreur (à l'origine identique à *stdout*).

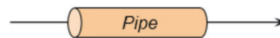
Dans l'exemple ci-dessus, l'entrée et la sortie standard des deux processus sont identiques. C'est le cas si les deux processus "tournent" dans une même fenêtre comme l'application ***ls*** lancé à partir d'un ***shell***.

L'illustration ci-dessus montre également une redirection du descripteur *stdout* dans le processus *Pb.* Dans ce cas, le descripteur référence l'entrée no. 9 dans la table des fichiers ouverts, entrée utilisée par le descripteur no. 6 dans l'espace utilisateur. Or, le descripteur no. 6 est celui obtenu à l'ouverture du fichier "log", selon l'exemple. Cela signifie que toute écriture utilisant le descripteur no. 1 (donc *stdout*) sera équivalent à une écriture utilisant le descripteur no. 6. Nous avons **donc redirigé la sortie standard dans le fichier "log"**. Le descripteur no. 1 est devenu synonyme du descripteur no. 6.

Tubes (1/6)



- **Tubes** (*pipes*)
- Communication (généralement) **unidirectionnelle**.
- Lecture **destructive**
  - Gestion de **type FIFO**



- **Capacité limitée** (généralement de 4 Ko)
- Fichier **spécial**
  - Utilisation des appels systèmes classiques *write()* et *read()*

6 Cours SYE - Institut REDS/HEIG-VD

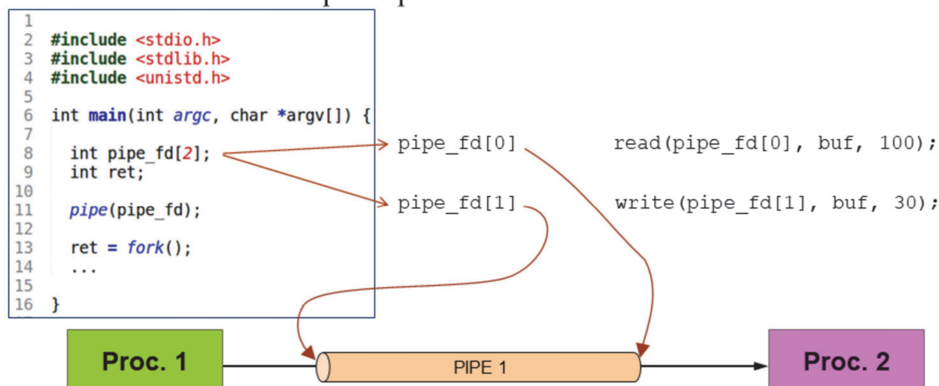
Les tubes sont utilisés avant tout pour du transfert rapide de messages courts entre processus et permettent une écriture/lecture de type FIFO selon un modèle producteur/consommateur. Un message déposé dans un tube restera présent dans celui-ci tant qu'aucun processus n'aura effectué la lecture. Le message sera automatiquement retiré du tube aussitôt que le processus consommateur aura effectué la lecture.

Les tubes ont une capacité limitée, ce qui signifie qu'ils peuvent se trouver dans un état *vide* ou *plein*. Dans les deux cas, le comportement *par défaut* consistera à **suspendre** le processus qui tentera d'effectuer une lecture d'un tube vide, et pour le processus qui tentera d'effectuer une écriture dans un tube plein (le processus passera dans l'état *waiting*).

La communication s'effectue en utilisant les appels système conventionnels utilisés pour la manipulation de fichiers, à savoir *write(), read(), close()*. C'est dire que les tubes sont intimement liés aux mécanismes de gestion des systèmes de fichiers; un tube sera considéré comme un fichier *spécial*.

Tubes (2/6)

- Un tube possède **deux descripteurs de fichier** car il y a deux extrémités.
  - Un descripteur pour écrire dans le tube
  - Un autre descripteur pour lire le tube



7 Cours SYE - Institut REDS/HEIG-VD

La création d'un tube (anonyme) s'effectue à l'aide de l'appel système *pipe*(), qui retournera un tableau de deux entiers (deux descripteurs). En effet, c'est en utilisant ces descripteurs comme premier argument des fonctions *write(), read(), etc.* que le développeur pourra différencier les deux extrémités.

La norme POSIX impose toutefois que le premier descripteur (*pipe_fd[0]* dans l'exemple ci-dessus) soit utilisé pour la **lecture** du tube, et le second descripteur (*pipe_fd[1]*) soit utilisé pour **l'écriture**. Par analogie, on se rappellera que le descripteur no. 0 de tout processus correspond à l'entrée standard (*stdin*), et le descripteur no. 1 correspond à la sortie standard (*stdout*).

Dans l'exemple ci-dessus, la communication est bien établi entre deux processus. Il faut se rappeler que l'appel système *fork()* permet la création d'un nouveau processus et que ce dernier hérite de toutes les ressources du parent. Grâce à ce mécanisme, l'appel au *fork()* après l'appel à *pipe()* aura pour effet de **dupliquer** les descripteurs de fichiers dans le processus fils et ces deux descripteurs permettront de référer le même tube.

Par la suite, et dans cet exemple, une des deux extrémités du tube (celle non utilisée) **devra être fermée** dans chacun des processus (parent/enfant).

Tubes (3/6)

- Tube **nommé**
  - Création d'un tube avec l'appel système *mkfifo()*
  - Connexion au tube avec l'appel système *open()*
  - Existence (virtuelle) dans le système de fichier
  - Persistance (doit être détruit explicitement)
- Tube **anonyme**
  - Tube créé avec l'appel système *pipe()*
  - Utilisation conjointe avec la création de processus
  - Pas de persistance (destruction automatique à la terminaison des processus)

8 Cours SYE - Institut REDS/HEIG-VD

Il existe deux types de tube décrit ci-après.

Les **tubes nommés** sont créés avec l'appel système *mkfifo()* et apparaissent par la suite comme un fichier (virtuel) au niveau de l'espace utilisateur. C'est pourquoi, ils peuvent être manipulés avec les appels système *open(), read(), write(), close()*. De tels tubes peuvent être référencés par n'importe quel processus désirant échanger des données au travers de ce mécanisme. Plusieurs processus peuvent se connecter sur un même tube. Un tube nommé reste persistent au niveau du système de fichiers et doit être explicitement détruit (appel système *unlink()* par exemple).

Les tubes **anonymes** sont particuliers et très simple d'utilisation grâce à l'appel système *fork().* Ils ne peuvent pas être référencés par n'importe quel processus, mais seulement par la descendance du processus parent qui a créer le tube. Un exemple fréquent de l'utilisation de tube anonyme apparaît au niveau d'un *shell* en utilisant la commande suivante : ***ls | more*** qui ura pour effet de transférer la **sortie** du processus *ls* vers **l'entrée** du processus *more* (sous Windows, c'est pareil avec la commande "***dir | more***".

Le symbole **|** (barre verticale) indique au *shell* qu'il faut créer un tube anonyme entre les deux applications. Le mécanisme nécessite également de connecter la sortie standard du processus *ls* vers l'entrée du *pipe*, et de connecter l'extrémité du *pipe* sur l'entrée standard du processus *more*. Bien sûr, cette *redirection* est possible grâce à l'appel système *dup2()*.

Tubes (4/6)

- Exemple avec un tube nommé

```c
8  int main(int argc, char *argv[]) {
9
10     int fd_pipe;
11     int ret;
12     char *pipeName = "essai.fifo";
13     char *str = "Bonjour";
14
15
16     if (mkfifo(pipeName, O_CREAT | 0644) != 0) {
17       printf("Pipe creation failed.\n");
18       exit(EXIT_FAILURE);
19     }
20
21     fd_pipe = open(pipeName, O_WRONLY);
22     if (fd_pipe == -1) {
23       printf("Failed at open.\n");
24       exit(EXIT_FAILURE);
25     }
26
27     write(fd_pipe, str, strlen(str) + 1);
28
29     close(fd_pipe);
30
31     return EXIT_SUCCESS;
32 }
```

```c
1  #include <fcntl.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <string.h>
6
7  #define BUFFER_SIZE   80
8
9  int main(int argc, char *argv[]) {
10
11     int fd_pipe;
12     char *pipeName = "essai.fifo";
13     char str[BUFFER_SIZE];
14
15     fd_pipe = open(pipeName, O_RDONLY);
16     if (fd_pipe == -1) {
17       printf("Failed at open.\n");
18       exit(EXIT_FAILURE);
19     }
20
21     read(fd_pipe, str, BUFFER_SIZE);
22     printf("%s\n", str);
23
24     close(fd_pipe);
25     unlink(pipeName);
26
27     return EXIT_SUCCESS;
28 }
```

9 Cours SYE - Institut REDS/HEIG-VD

L'exemple ci-dessus montre deux processus, l'un créant un tube nommé et se connectant à l'extrémité de celui-ci en écriture, l'autre se connectant à l'extrémité en lecture. La *connexion* au tube s'effectue à l'aide de l'appel système *open()*. Le nom du tube est stocké dans la variable *pipeName*.

Dans cet exemple, le premier processus pourrait être considéré comme un "serveur" et le second comme un "client".

Avec le comportement par défaut, le premier processus qui se connecte au tube se bloque lors de l'appel à *open()* jusqu'à ce que le second processus se connecte à l'autre extrémité.

Tubes (5/6)

- Exemple avec un tube anonyme

- **Deux processus**
- Le fils écrit dans le tube.
- Le parent lit le tube.

```c
 7 int main(int argc, char *argv[]) {
 8
 9   int pipe_fd[2];
10   char buffer[80];
11   int pid;
12
13   /* Pipe creation */
14   pipe(pipe_fd);
15
16   /* Now, we fork a child */
17   pid = fork();
18
19   if (!pid) {
20
21     /* The child will send a message */
22     close(pipe_fd[0]);
23     write(pipe_fd[1], "Hello, I am your child\n", 24);
24     close(pipe_fd[1]);
25
26     exit(0);
27
28   } else {
29
30     /* The parent will read something from the pipe */
31     close(pipe_fd[1]);
32
33     read(pipe_fd[0], buffer, 80);
34
35     printf("I got from the child: %s\n", buffer);
36     close(pipe_fd[0]);
37     waitpid(pid, NULL, 0);
38
39   }
40   exit(0);
41 }
```

10 Cours SYE - Institut REDS/HEIG-VD

La création d'un tube anonyme s'effectue avec l'appel système *pipe()*. Celui-ci réserve deux descripteurs de fichier, l'un pour permettre l'écriture dans le tube, et l'autre l'écriture.

L'utilisation d'un tube anonyme n'a de sens que si le processus qui l'obtient crée un ou plusieurs processus enfants. Par l'effet de duplication de l'appel système *fork()*, le processus fils recevra une copie conforme du tableau des descripteurs associé au processus parent. De ce fait, le processus fils disposera des descripteurs liés au tube créé dans le processus parent. En revanche, il faut relever que le tube lui-même constitue une ressource du noyau, utilisée par les processus, et **n'est pas dupliqué**.
Ce mécanisme permet bien aux deux processus (parent et enfant) de manipuler le même tube.

Tubes (6/6)

- Utilisation du tube dans un *shell* : "**ls | more**"

```
2
3   /* ... */
4
5   int pipe_fd[2];
6
7   pipe(pipe_fd);
8
9   if (!fork()) {
10
11      close(pipe_fd[0]);          /* Extrémité pas utilisée dans ce processus */
12      dup2(pipe_fd[1], 1);        /* stdout devient synonyme de pipe_fd[1] */
13
14      execve("/bin/ls", ...)      /* Remplacement de l'image binaire */
15
16  } else {
17
18      close(pipe_fd[1]);          /* Extrémité pas utilisée dans ce processus */
19      dup2(pipe_fd[0], 0);        /* stdin devient synonyme de pipe_fd[0] */
20
21      execve("/bin/more", ...);   /* Remplacement de l'image binaire */
22
23  }
24
25  /* ... */
26
```

11 Cours SYE - Institut REDS/HEIG-VD

Comme nous l'avons vu au début de ce chapitre, l'appel système ***dup2(orig, copy)*** permet de créer une redirection du descripteur *copy* vers le descripteur *orig* (*copy* devient synonyme de *orig*).

L'exemple ci-dessus est très instructif. On remarque qu'une redirection est effectuée dans les deux processus : dans le processus parent, c'est l'entrée standard qui devient synonyme de l'extrémité en lecture du tube. Dans le processus fils, c'est la sortie standard qui devient synonyme de l'extrémité en écriture du tube.

Dans les deux processus, la redirection est suivie par l'appel à *exec()*. Dans ce cas, le remplacement de l'image binaire **n'affecte pas** le tableau des descripteurs associé au processus. Autrement dit, lorsque la nouvelle image (le nouveau programme) s'exécute, le tableau des descripteurs **reste inchangé** (les références sont préservées) et les redirections précédentes restent valides : dans le processus fils, toute écriture vers *stdout* amènera une écriture dans le tube, et dans le processus enfant, toute lecture sur *stdin* amènera une lecture du tube.

En marge, il est utile de noter une autre utilisation de l'appel système *dup2()*. Le *shell* peut l'utiliser pour rediriger la sortie d'une application vers un fichier. Exemple d'une telle utilisation : `ls > liste.txt`

Cette "commande" aura pour effet de rediriger la sortie du processus *`ls`* dans le fichier *`liste.txt`* (à noter que dans ce cas, aucun tube n'intervient).

Signaux (1/5)

- Les signaux sont des informations de type événement qui peuvent être transmis à un processus d'une manière **asynchrone**.
  - Analogie à la notion d'interruption
  - Le **noyau** ou un **processus** peut envoyer des signaux.

- Un processus peut **réagir** à la réception d'un signal.
  - Il existe un comportement par défaut associé à chaque signal.
- Le signal provoque **l'exécution d'un traitant** (*handler*) **dans l'espace utilisateur**.
  - Proche de la notion de *routine de service (ISR)* mais dans l'espace utilisateur.

12 Cours SYE - Institut REDS/HEIG-VD

Les signaux sont utilisés pour envoyer des événements de différents types aux processus. Dès qu'un processus reçoit un signal (il faut bien entendu que le processus soit dans l'état *running*), il le traite immédiatement, c-à-d que l'exécution est déroutée sur une fonction particulière appelée *traitant* (*handler*). Cette fonction est aussi une fonction de type *callback* car elle n'est pas invoqué explicitement par le processus, mais par le noyau et à n'importe quel moment.

Les signaux peuvent jouer le rôle de sonnette d'alarme: lorsque l'utilisateur appuie sur une séquence de touche comme *CTRL/C* par exemple, le noyau envoie un signal au processus qui peut décider d'une action particulière en réponse à cet événement. Si le développeur ne souhaite pas implémenter une action particulière, un *comportement par défaut* est alors déclenché (chaque signal en possède un). Dans le cas du *CTRL/C*, le processus terminera son exécution aussitôt.

Signaux (2/5)

- Un seul exemplaire de signal en attente
- Le signal est pris en compte dès que le processus se trouve dans l'état running.
- Possibilité de masquer un signal
  - Le signal est temporisé et délivré lors du démasquage de celui-ci.
- Possibilité d'ignorer un signal
  - Le signal est perdu.

13 Cours SYE - Institut REDS/HEIG-VD

Chaque processus (et la plupart du temps chaque *thread*) dispose de sa propre file de signaux en attente. Quelques règles simples permettent de comprendre le fonctionnement des signaux :

a) Un seul exemplaire de signal peut se retrouver dans la file d'attente (appuyer plusieurs fois sur *CTRL/C* alors que le processus ne réagit pas encore aura pour effet de temporiser qu'un seul exemplaire de *CTRL/C*).

b) Un signal peut être masqué. Dans ce cas il est temporisé dans la file d'attente, même lorsque le processus s'exécute. Dès que le signal est démasqué, le *traitant* correspondant sera automatiquement appelé.

c) La fonction associée au traitant peut être implémentée dans le processus ou une fonction par défaut peut être utilisée. On peut également décider d'ignorer simplement le signal, dans ce cas il n'y a plus aucune fonction associée. Cela n'est cependant pas possible pour tout l'ensemble des signaux prédéfinis: quelques signaux critiques ne peuvent pas voir leur traitant redéfini ou ignoré (signal permettant la terminaison abrupte d'un processus par exemple).

Signaux (3/5)

- Exemples de signaux prédéfinis

Nom du signal	Evénement associé
SIGHUP	Terminaison du processus leader de session
SIGINT	Frappe du caractère **intr** **(CTRL/C)** sur le clavier du terminal de contrôle
SIGQUIT	Frappe du caractère **quit** **(CTRL/D)** sur le clavier du terminal de contrôle
SIGILL	Détection d'une instruction illégale
SIGABRT	Terminaison anormale provoquée par l'exécution de la fonction *abort*
SIGFPE	Erreur arithmétique (division par zéro)
SIGKILL	Signal de terminaison
SIGSEGV	Violation mémoire (segmentation fault)
SIGPIPE	Ecriture dans un tube sans lecteur
SIGALRM	Fin de temporisation (fonction *alarm*)
SIGTERM	Signal de terminaison
SIGUSR1	Signal émis par un processus utilisateur
SIGUSR2	Signal émis par un processus utilisateur
SIGCHLD	Terminaison d'un fils
SIGSTOP	Signal de suspension
SIGTSTP	Frappe du caractère **susp** **(CTRL/Z)** sur le clavier du terminal de contrôle
SIGCONT	Signal de continuation d'un processus stoppé
SIGTTIN	lecture par un processus en arrière-plan
SIGTTOU	Ecriture par un processus en arrière-plan (terminal en mode **tostop**)

Cours SYE - Institut REDS/HEIG-VD

14

Il existe toute une série de signaux prédéfinis par l'OS. Le tableau ci-dessus montre les signaux les plus courant (à chaque nom de signal correspond une constante numérique). La commande "***kill -l***" permet d'obtenir une liste complète des signaux.

Le signal ***SIGINT*** correspond au *CTRL/C*.

Le signal **SIGKILL** est un signal envoyé lors de la commande "***kill -9 <pid>***" et termine de manière abrupte le processus *<pid>*. A noter que l'envoi d'un signal peut se faire à un processus se trouvant dans l'état *waiting*. Dans ce cas, le processus passera alors dans l'état *ready* avant de pouvoir se retrouver dans l'état *running*, d'exécuter le traitant correspondant et repasser dans l'état *waiting*.

Le signal ***SIGCHLD*** est toujours envoyé par le processus fils à son processus parent lorsque celui-là termine son exécution. Le parent peut ainsi réagir (en exécutant un *waitpid()* par exemple pour récupérer l'état du fils).

Les signaux ***SIGUSR1*** et ***SIGUSR2*** sont deux signaux sans signification particulière au niveau du noyau. Ils sont libres d'utilisation par les processus.

Signaux (4/5)

- **Association** du signal avec son traitant
  - Appel système ***signal()***
- **Envoi** d'un signal depuis un processus
  - Appel système ***kill()***

Processus A

```c
/* Prototype */
int kill(pid_t pid, int sig);

int main(int argc, char **argv) {
  int pid_dest;

  pid_dest = atoi(argv[1]);

  kill(pid_dest, SIGUSR2);

}
```

Envoi d'un signal →

Processus B

```c
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

void process_sig(int sig) {

  switch (sig) {
    case SIGUSR1:
      printf("Received SIGUSR1\n");
      break;
    case SIGUSR2:
      printf("Received SIGUSR2\n");
      break;
  }
}

int main(int argc, char **argv) {

  signal(SIGUSR1, process_sig);
  signal(SIGUSR2, process_sig);

  while (1);
}
```

15 Cours SYE - Institut REDS/HEIG-VD

L'association d'un signal avec son traitant s'effectue avec l'appel système *signal()* – ou de préférence ***sigaction***().

L'envoi de signal depuis un processus s'effectue avec l'appel système *kill().* A noter qu'il existe également l'application *kill*, permettant d'envoyer un signal à n'importe quel processus. Par défaut, l'application envoie le signal *TERM* au processus correspondant au *pid* passé en argument.

Signaux (5/5)

- Le traitant prédéfini *SIG_IGN* permet d'ignorer le signal
- Le traitant prédéfini *SIG_DFL* de restituer le comportement par défaut.
- Un processus peut être *tué* avec *kill -9 <pid>*

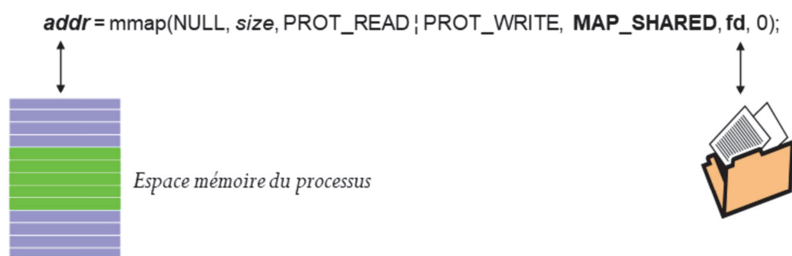
```c
1
2  /* L'exemple suivant montre comment ignorer les CTRL/Z (suspension). */
3  /* Le corps du programme ne fait rien (boucle infinie) */
4
5  void fonctionDuSignal(int sig)
6  {
7    if (sig == SIGTSTP)
8      printf("Suspension impossible\n");
9  }
10
11 int main(int argc, char **argv)
12 {
13   signal(SIGTSTP, fonctionDuSignal);
14
15   while (1);
16 }
```

```c
1
2  int main(int argc, char **argv)
3  {
4    signal(SIGINT,SIG_IGN);
5    signal(SIGQUIT,SIG_IGN);
6
7    while (1);
8  }
```


Chaque signal possède un comportement par défaut; à tout moment, le traitement d'un signal peut être réinitialisé de telle sorte à ce que le comportement par défaut soit réactivé (signal *SIGN_DFL*). Si l'on souhaite ignorer un signal, il suffit d'utiliser l'argument *SIG_IGN* à la place du traitant.

Fichiers mappés (1/1)

- L'appel système *mmap()* permet de **projeter** le contenu d'un fichier en mémoire.
  - **Synchronisation automatiquement** du contenu entre la mémoire et le fichier projeté.
- Un fichier mappé (en mémoire) peut être **partagé** par plusieurs processus, d'où son utilisation en tant qu'IPC.



17 Cours SYE - Institut REDS/HEIG-VD

Le mécanisme de fichier mappé est très puissant et est utilisé non seulement comme IPC, mais également dans le cadre de chargement d'une image binaire (exécutable) ou généralement d'un fichier en mémoire. Nous comprendrons plus l'intérêt de l'appel système *mmap()* lorsque nous aborderons le chapitre consacré à la gestion mémoire virtuelle. Dans le contexte IPC, les fichiers mappés peuvent être utilisés pour un partage efficace et *synchronisé* entre plusieurs processus. L'appel système *mmap()* charge le contenu dans l'espace d'adressage du processus (donc dans sa mémoire, et dans l'espace utilisateur) de telle sorte que le processus puisse rapidement accéder au contenu et effectuer des modifications le cas échéant. Une particularité importante des fichiers mappés est que toute modification en mémoire peut être automatiquement reportée dans le fichier même (rôle de la constante *MAP_SHARED* ci-dessus).

Sockets réseau (1/9)

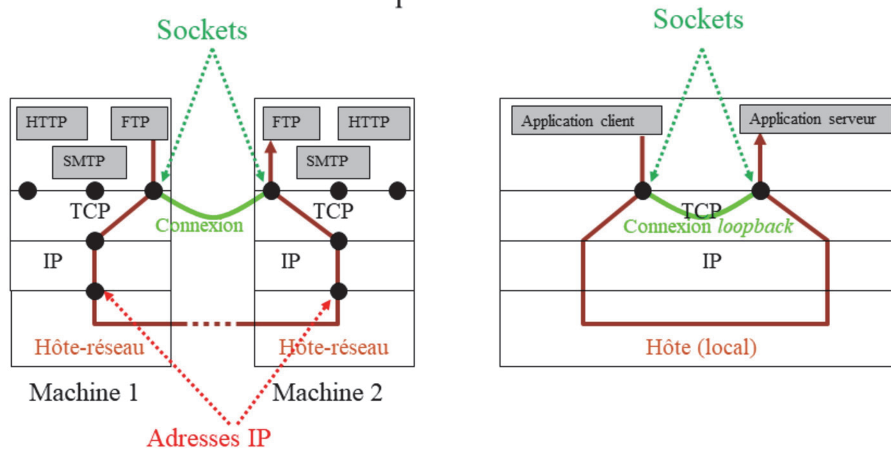
- ***Socket***
  - Extrémité d'une communication
  - Sockets de type TCP, UDP
  - API standardisé au niveau POSIX
  - Utilisation similaire à la manipulation de fichier

18 Cours SYE - Institut REDS/HEIG-VD

Les *sockets* constituent un élément-clé dans la communication entre applications locales et distantes. Ils ont été créés comme interface de communication dans les systèmes *Unix*, puis se sont répandus dans pratiquement tous les systèmes d'exploitation (sous *Windows*, c'est la notion de *Winsocks* qui s'apparente aux *sockets*).
Il existe différents types de *socket*, en fonction de l'usage que l'on en fait dans l'application utilisateur : les *socket TCP/IP* et *socket UDP/IP* sont certainement les plus utilisés. Il existe aussi des *sockets* de type *"raw"* permettant de manipuler les paquets IP de la couche réseau, et les *sockets* de type "*device socket*" permettant de gérer directement les paquets de la couche de liaison (couche 2 du modèle OSI).

Aujourd'hui, l'interface de programmation des *sockets* est spécifiée dans les normes *POSIX* (on parle aussi de *socket BSD* puisqu'ils ont d'abord été définis par l'Université de *Berkeley*. Sous Unix/Linux/MacOSX, l'interface permet d'utiliser certains appels systèmes dédiés à la manipulation de fichier (*open(), read(), write(), etc.*). L'approche est ainsi compatible avec ce type de fonction, et la notion de *descripteur de fichier* est également reprise : on parlera, dans le contexte des *sockets,* de ***descripteur de socket*** pour permettre aux appels systèmes de référencer le bon *socket*.

En revanche, la gestion des *sockets* au niveau noyau n'a rien à voir avec celle d'un fichier. C'est pourquoi, le noyau dispose d'un sous-système réseau à part entière qui gère le transit des paquets réseau traversant les différentes couches de protocole, entre l'application (espace utilisateur) et la carte réseau (matériel).



Dans ce chapitre, nous considèrerons uniquement des *sockets* de type TCP/IP (probablement les plus utilisés). La figure ci-dessus montre une architecture typique en couche intervenant dans la communication entre deux machines distantes (reliées par une connexion filaire de type *Ethernet* par exemple, ou sans fil de type *Wifi*), ainsi qu'entre processus tournant sur une même machine. On remarque sur cette figure que les *sockets* constituent le point d'entrée sur la couche TCP (dans notre cas) : les *sockets* permettent de définir les **extrémités** d'une connexion.

Dans le premier cas (deux machines distantes), deux adresses IPs sont requises afin d'identifier les machines, alors que dans le second cas (une seule machine), une seule adresse IP est requise (l'adresse locale est aussi définie comme ***localhost***).

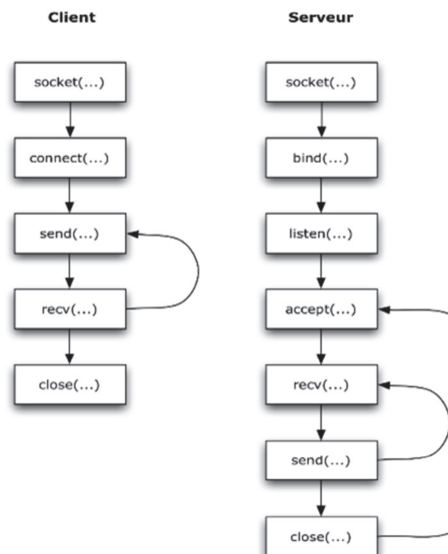
Les adresses IP ne suffisent donc pas à identifier l'application émettrice ou réceptrice. La notion de ***port*** doit également être associée. Cette notion est propre au protocole du niveau supérieur à la couche IP. Le protocole TCP permet de gérer un ensemble de ports associés à une adresse IP identique. Chaque port peut ainsi être associé à une application spécifique (exemple d'une adresse TCP/IP avec référence au port : **192.168.1.34:8080**). Le port no. 8080 est généralement utilisé par une application de type *navigateur*.

La communication entre deux processus locaux est un cas particulier et conduit à l'utilisation d'une connexion de type ***loopback*** (boucle fermée) : elle est plus simple à gérer aux niveaux des couches de protocole.

Sockets réseau (3/9)

- **Appels systèmes** dédiés aux *sockets*
  - *socket()*
  - *close()*
  - *connect()* → ***Spécifique au client***
  - ***send()***
  - ***recv()***
  - *bind()*
  - *listen()*
  - *accept()*

  Spécifique au serveur (bind(), listen(), accept())



20 Cours SYE - Institut REDS/HEIG-VD

Dans la plupart des cas, une communication TCP/IP est établie selon une approche de type *client-serveur*. Le serveur s'initialise en se mettant à l'écoute sur un port de communication dédié. Le client démarre la communication en envoyant une requête au serveur qui, ce dernier, établit le flux de communication bidirectionnelle avec le client. Le mécanisme est détaillé plus loin.

Le scénario d'établissement d'une communication tel que décrit ci-dessus fait intervenir plusieurs appels systèmes qui doivent être utilisés judicieusement au niveau client et au niveau serveur.

Les principaux appels systèmes sont énoncés ci-dessus.

Sockets réseau (4/9)

- Structure utilisée pour la définition d'une adresse IP
  - #include <sys/socket.h>

- Structure *sockaddr* et *sockaddr_in*

```
1
2  struct sockaddr {
3    sa_family_t     sa_family;      /* address family, AF_xxx       */
4    char            sa_data[14];    /* 14 bytes of protocol address */
5  };
6
7  struct sockaddr_in {
8    short   sin_family;       /* address family */
9    u_short sin_port;         /* TCP/UDP port number */
10   struct  in_addr sin_addr; /* IP address */
11   char    sin_zero[8];      /* not used, must be zero */
12 };
```

21 Cours SYE - Institut REDS/HEIG-VD

Avant d'entrer dans le détail des appels systèmes, il est nécessaire de mentionner l'existence de quelques structures de données qui interviennent dans les arguments de ces fonctions. Ces structures et autres types de données concernés découlent directement de l'API décrit dans les spécifications POSIX.

La structure générique ***sockaddr*** permet de décrire une adresse IP. Le caractère générique de cette structure apparaît au niveau du champ *sa_data* qui réserve 14 octets contenant l'adresse de protocole (pas forcément IP à priori). Ainsi, le mécanisme de transtypage (*cast*) du langage C permettra de convertir ce champ vers le type que l'on souhaite utiliser.

La structure ***sockaddr_in*** est un exemple de structure spécifique pour décrire une adresse IP. On retrouvera ainsi les champs suivants (ces 3 champs font 14 octets):

- *sin_family* Famille d'adresses AF_INET pour la famille de protocoles PF_INET
- *sin_port* Numéro de port en format *big-endian* (utilisé dans les réseaux)
- *sin_addr* Adresse IP en format numérique

La conversion d'une adresse IP en format "aaa.bbb.ccc.ddd" vers le format numérique (de type *struct in_addr*) et vice-versa s'effectue respectivement à l'aide des fonctions ***inet_pton()*** et ***pton_inet()***.
De même la conversion du numéro de port dans le format ad-hoc s'effectue à l'aide des fonctions ***ntohs()*** et ***htons()***.

Sockets réseau (5/9)

- **Appels systèmes** (1/2)
  - Gestion des *sockets*

```
1
2  /* Création d'un socket */
3  int socket(int domain, int type, int protocol);
4
5  /* Fermeture d'un socket */
6  int close(int fd);
7
8  /* Connexion d'un socket à une adresse de destination */
9  int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
10
11 /* Spécification de l'adresse locale */
12 int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
13
14 /* Mise du socket en mode passif & création d'une queue de connexions */
15 int listen(int sockfd, int backlog);
16
17 /* Attente d'une connexion client / Traitement de connexions en attente */
18 int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
19
```

22 Cours SYE - Institut REDS/HEIG-VD

Les appels système ci-dessus permettent la gestion des *sockets* en terme d'ouverture (initialisation), préparation à la mise en écoute (*bind()* + *listen()* + *accept()*) pour un serveur et l'établissement de connexion côté client *(connect()).*

Comme d'habitude, des informations détaillés sur l'utilisation de ces fonctions peuvent être obtenues dans les pages *man*.

Sockets réseau (6/9)

- **Appels systèmes** (2/2)
  - Gestion du flux (continu) de données
  - Fonctionnement identique à celle d'un fichier
  - Attention à la signification de l'argument `<len>/<count>`

```c
1
2  /* Envoi de données */
3  ssize_t send(int sockfd, const void *buf, size_t len, int flags);
4  ssize_t write(int fd, const void *buf, size_t count);
5
6  /* Réception de données */
7  ssize_t recv(int sockfd, void *buf, size_t len, int flags);
8  ssize_t read(int fd, void *buf, size_t count);
9
```

23 Cours SYE - Institut REDS/HEIG-VD

L'envoi de données au travers des *sockets* se fait aisément à l'aide des appels systèmes *send()* ou *write()*. La différence entre les deux réside dans l'apparition d'un quatrième argument *<flags>* permettant de contrôler le comportement des *sockets* (opérations bloquantes ou non par exemple).

Ces appels systèmes sont bloquant (ou *synchrones*) dans leur exécution, et se termine au retour des fonctions. On peut utiliser les appels systèmes ***ioctl()*** ou *fcntl()* afin de contrôler certaines caractéristiques des *sockets*, telles que rendre les appels systèmes *read()/write()* non bloquants.

Tout comme pour les fichiers, il faut faire très attention dans l'interprétation de l'argument *<len>/<count>* de ces fonctions: dans le cas d'un *write()/send()*, l'argument spécifie le nombre d'octets **qui doivent** être transférés, c-à-d le nombre d'octets contenus dans le *buffer (<buf>)*. Le retour de ces fonctions indiquera le nombre d'octets réellement écrits; dans le cas d'une utilisation en mode bloquant (mode par défaut), la valeur de retour correspond toujours à la valeur de cet argument, sauf s'il y a eu un problème de transfert. En revanche, dans le cas d'un *read()/recv()*, l'argument spécifie le nombre maximum d'octets **qui peut** être reçus (et non qui doit être reçus). Cette valeur correspond généralement à la **capacité maximale du *buffer***. Le retour des fonctions indiquera le nombre d'octets réellement reçus, qui peut être inférieur à l'argument (c'est très souvent le cas).

Sockets réseau (7/9)

- Connexion de type *client-serveur*

Client

```
fd = socket();

connect(fd, addr, port);

send(fd, buf, size, 0);
len = recv(fd, buf, maxlen, 0);

close(fd);
```

Serveur

```
fd = socket();
bind(fd, addr, addrlen);
listen(fd, max_connections);
newfd = accept(fd, &addr, &addrlen);
len = recv(newfd, buf, maxlen, 0);
send(newfd, buf, size, 0);
close(newfd);

close(fd);
```

Création du socket et liaison avec une adresse/port

demi-fermeture

24 Cours SYE - Institut REDS/HEIG-VD

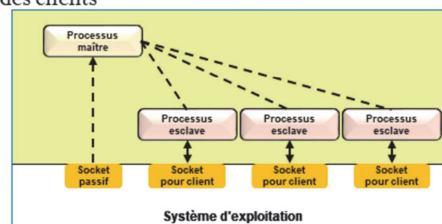
L'exemple ci-dessus schématise l'établissement d'une connexion de type *client-serveur*. Le serveur commence par déclarer un *socket* (référencée par le descripteur de fichier *fd*). Ce *socket* permettra d'écouter les demandes de connexions en provenance des clients. Un client devra se connecter au serveur en utilisant une **adresse *IP*** ainsi qu'un **numéro de *port***. Ces informations d'adresse (*IP + port*) sont définies dans la structure de type *sockaddr* et doivent être définies par le serveur; c'est la raison de l'appel à la fonction *bind()*. Puis le serveur demande au noyau d'initialiser une **file d'attente** liée à cette adresse afin que les demandes puissent être ***temporisées*** jusqu'à leur prise en compte. L'appel système *listen()* permet de faire cela en spécifiant le nombre maximum de requêtes pouvant être temporisées. L'appel système *accept()* permettra de traiter une requête en attente, de récupérer l'adresse (et numéro de port) du client et d'associer un nouveau socket qui sera réservé à cette connexion. Celui-ci contiendra également un **port local** déterminé par le sous-système réseau et qui sera **dédié à cette connexion**; il n'y aura par conséquent aucune interférence avec le canal réservé pour l'écoute des requêtes.

Le client quant à lui déclare un socket et effectue la requête de connexion au serveur à l'aide de l'appel système *connect()*. Dès que la connexion est établie, l'envoi et la réception de données peut alors commencer.

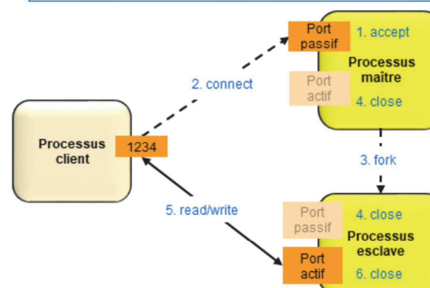
La fermeture du *socket* s'effectue de manière indépendante entre le client et le serveur. C'est dire qu'il n'y a pas (forcément) simultanéité; l'état transitoire est appelé *demi-fermeture* et sera discuté plus loin.

Sockets réseau (8/9)

- Serveur **itératif et concurrent**
  - Traite plusieurs communications en parallèle
  - Evite des délais d'attente avant le traitement des clients
  - Implémentation par ***threads***
  - Implémentation par **processus**



- Serveur **TCP** avec connexion
  - Communication fiable (robuste) point-à-point
  - Etablissement fiable de connexion
  - Contrôle de flux
  - Transmission *full-duplex*



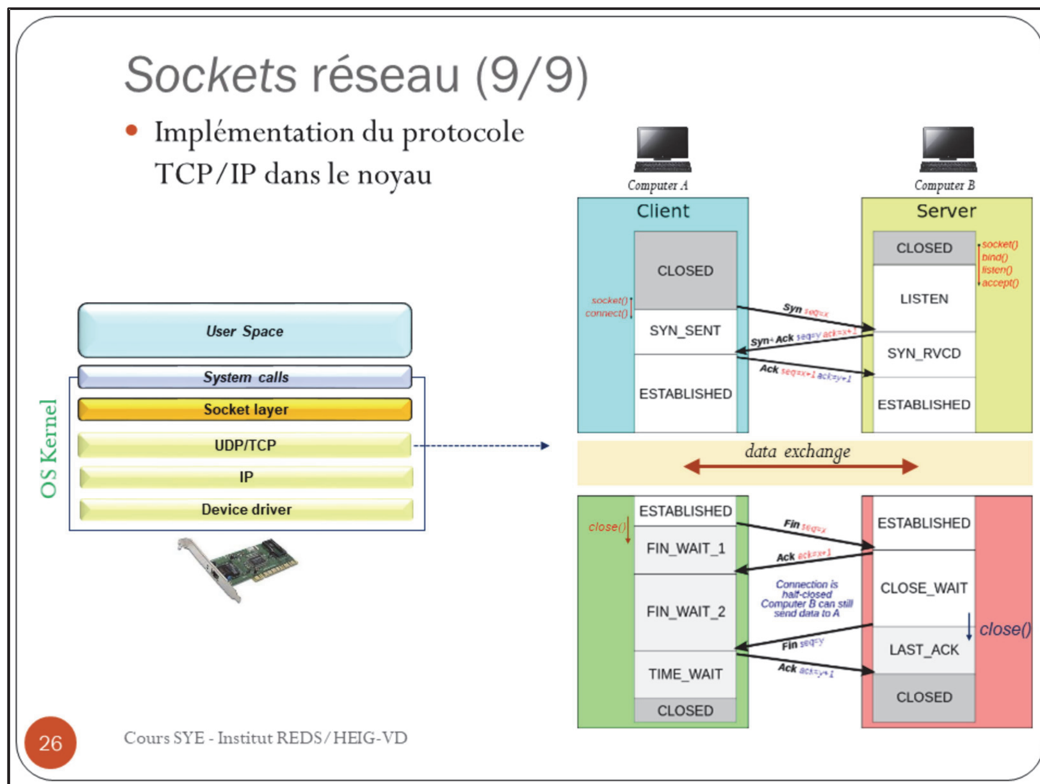
25

Cours SYE - Institut REDS/HEIG-VD

Il existe plusieurs approches pour implémenter une architecture de type *client-serveur*. Nous nous concentrerons sur une approche de type *itérative* et *concurrente*, avec une communication de type TCP/IP.

Dans une architecture de ce type, le serveur se met à l'écoute des requêtes-client sur un port dédié, et démarre un nouveau ***processus*** ou un ***thread***, qui prendra en charge la communication avec ce client particulier exclusivement. C'est pourquoi la communication avec chaque client nécessitera l'utilisation d'un port particulier (côté serveur). La paire de *socket* client-serveur suffira à l'établissement d'une transmission bidirectionnelle (*full-duplex*). Dès que le serveur a démarré le nouveau processus (ou *thread*), il se remet immédiatement à l'écoute de nouvelles requêtes, donnant ainsi au serveur le moyen de réagir rapidement aux demandes des clients.

Dès que la communication entre le client et le serveur se termine, le processus (ou *thread*) meurt et les ports sont libérés.



L'établissement d'une communication TCP/IP entre client et serveur passe par un protocole bien défini qui fait partie intégrante de la couche de transport (en l'occurrence TCP). La figure de gauche ci-dessus met en évidence la présence d'un sous-système réseau au sein du noyau d'OS qui gère précisément l'architecture en couche du modèle OSI.

La couche TCP est responsable notamment d'établir la connexion entre deux *sockets* à l'aide du protocole de type *handshake*, telle qu'il est décrit sur la figure de droite. L'exemple montre la connexion de deux clients sur un serveur.

TCP définit différents types de messages (*SYN/ACK/SYN+ACK/FIN*) ainsi qu'un numéro de séquence transmis avec le message. Un message de type *SYN* signifie une demande de connexion. Le serveur répond avec un message *SYN+ACK* et le client à son tour doit répondre au serveur par le message *ACK*.
La terminaison s'effectue d'une manière similaire. L'appel système *close()* déclenche l'envoi d'un message *FIN* qui devra être acquitté par l'autre extrémité, qui devra à son tour effectuer un *close()* (au niveau applicatif).
Le délai entre le traitement du message *FIN* et l'exécution du *close()* dépend de l'application; c'est un cas intéressant qui nécessite un état particulier au niveau *socket* appelé *"demi-fermeture" (half-closed)* où seule la transmission du serveur vers le client est possible.

Segments de mémoire partagée (1/1)

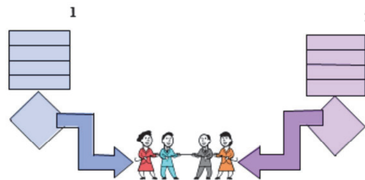
- Les processus peuvent se partager des segments de mémoire partagée.
  - Pas de structure organisationnelle au niveau du contenu
- *Appels systèmes*
  - *int **shmget(key_t** cle, int taille, int options);*
    - *Création d'un segment mémoire*
  - *void ***shmat(int** dipc, const void *adr, int option);*
    - *Attachement du segment identifié par <dipc> à l'adresse <adr> du processus exécutant l'appel système.*
  - *int **shmdt** (const void *adr);*
    - *Détachement du segment*


```c
/* shm_server.c */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ     27

main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    /*
     * We'll name our shared memory segment "5678".
    */
    key = 5678;

    /*
     * Create the segment.
     */
    if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /*
     * Now we attach the segment to our address space.
     */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /*
     * Now put some things into the memory for the
     * other process to read.
     */
    s = shm;

    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = NULL;

    /*
     * Finally, we wait until the other process changes the first
       character of our memory to '*', indicating that
       it has read what we put there.
     */
    while (*shm != '*')
        sleep(1);

    exit(0);
}
```

```c
 * shm-client - client program to demonstrate shared memory.
 */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ     27

main()
{
    int shmid;
    key_t key;
    char *shm, *s;

    /*
     * We need to get the segment named
     * "5678", created by the server.
     */
    key = 5678;

    /*
     * Locate the segment.
     */
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /*
     * Now we attach the segment to our data space.
     */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /*
     * Now read what the server put in the memory.
     */
    for (s = shm; *s != NULL; s++)
        putchar(*s);
    putchar('\n');

    /*
     * Finally, change the first character of the
     * segment to '*', indicating we have read
     * the segment.
     */
    *shm = '*';

    exit(0);
}
```

Objets de synchronisation (1/1)

- Le système d'exploitation fournit les mécanismes pour contrôler **l'accès concurrent** par différents processus.



- **Sémaphore**
  - Utilisation d'un compteur, d'une liste de processus, méthodes P() et V(), etc.

28 Cours SYE - Institut REDS/HEIG-VD

Les mécanismes de synchronisation interprocessus sont vastes et ne sont pas traités dans ce cours: en effet, ils seront étudiés en détail dans le cadre d'un cours de programmation concurrente.

Références

- A. Silberschatz et al.:
  "**Operating Systems Concepts**", 8th edition, Wiley

- Andrew Tanenbaum
  "**Systèmes d'exploitation**", 3ème édition