UE : Inter Processus Communications

L3I parcours MIAGE

Gilles Menez

Université de Nice – Sophia-Antipolis
Département d'Informatique
email : menez@unice.fr
www : www.i3s.unice.fr/~menez

3 octobre 2022: V 1.1

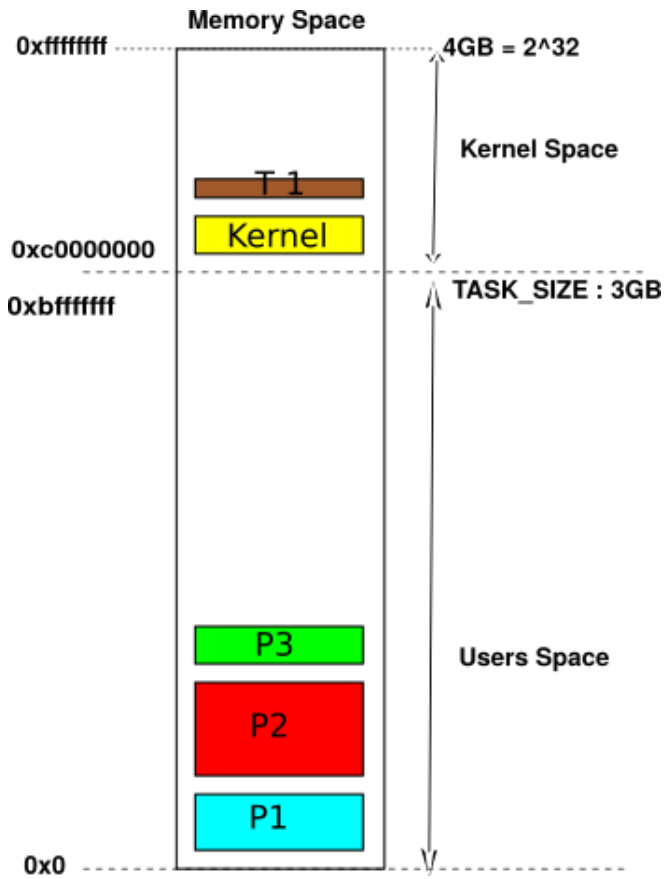
Table des Matières

Préambule : IPC

Inter Processus Communications

Description de cette partie de l'UE : Système et réseaux

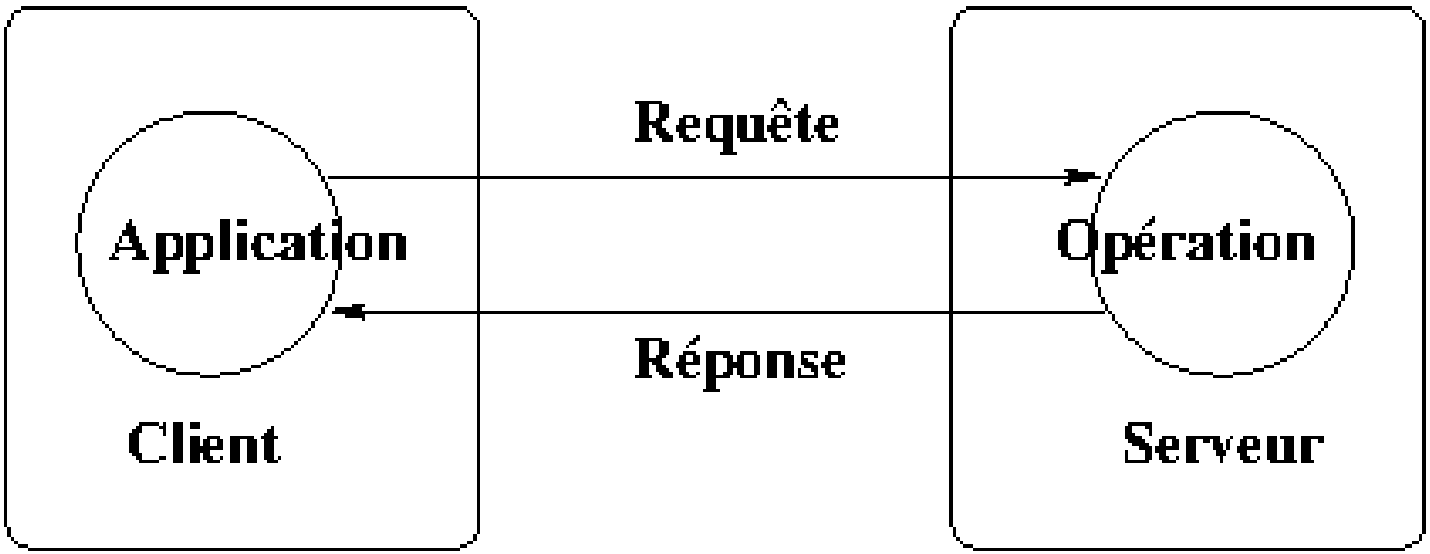
On s'est quitté sur les "processus" :



➢ Ce concept est fondamental et fondateur pour répondre au problème de la multi-utilisation (multi-utilisateurs, multi-activités) des machines.

Communication ?

Si ces activités/processus ne sont pas communicantes alors le cours peut s'arrêter là !



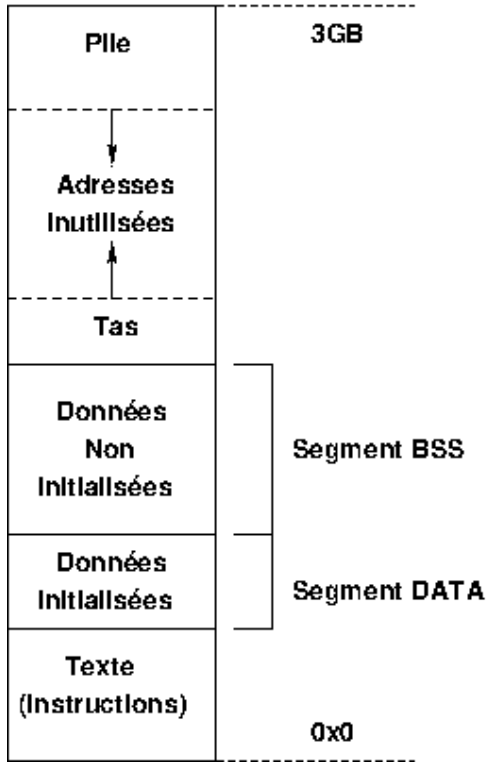
Mais par contre, **si ces activités souhaitent échanger des informations** alors il va falloir en dire plus !

- ➢ Que les activités soient situées sur la même machine,
- ➢ ou sur des machines différentes.

"Communication entre processus" **intra machine**

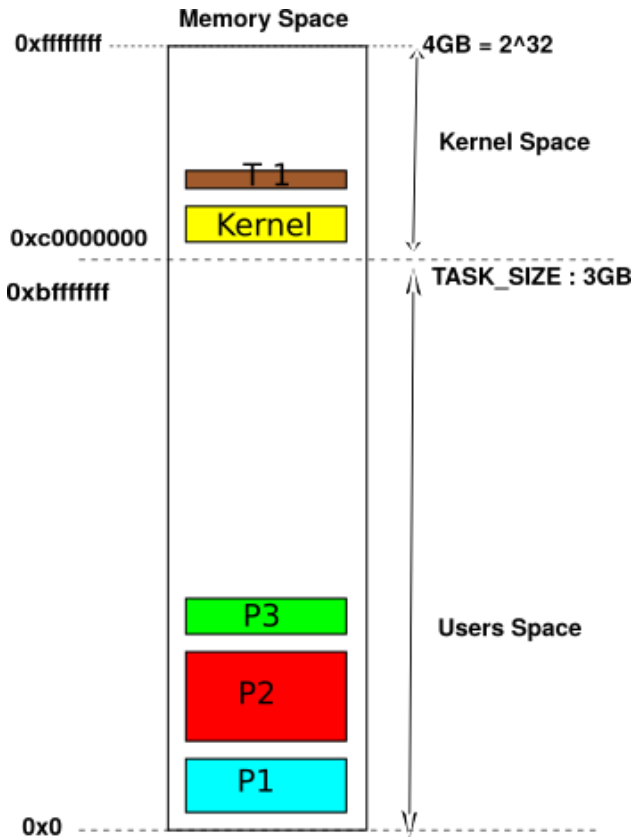
Avant même de considérer le cas de processus s'exécutant sur des machines différentes, en quoi la communication "intra machine" pose des problèmes ?

On sait que, dans sa conception, l'**espace mémoire de chaque processus lui est propre** :



Chaque processus voit la mémoire de la machine (virtuelle) **comme si il était seul** à s'exécuter !

Notamment, il n'y pas, de façon native, de zone mémoire visible / accessible / **commune** qui permettrait un partage/échange d'informations selon un scénario idéalisé :

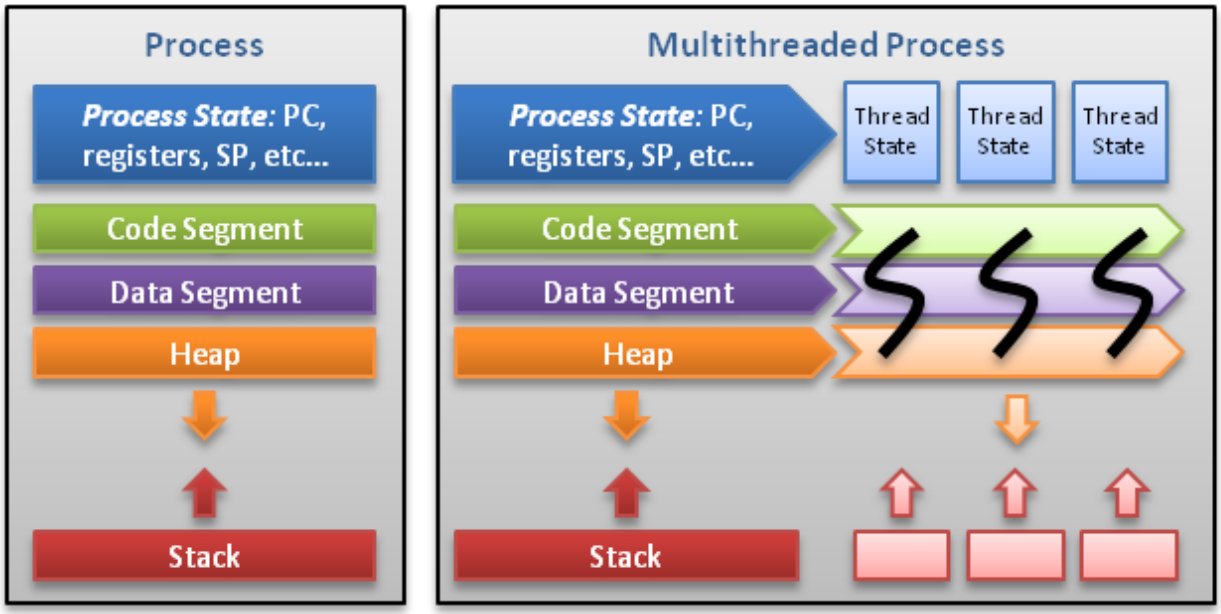


- ➢ Le processus A place une information dans cette zone,
- ➢ Le processus B récupère l'information depuis cette zone et l'utilise,
- ➢ etc

. . . utilisant ainsi cette zone pour dialoguer !

C'est tout l'intérêt des threads !

Que de partager entre les threads, la vision de l'espace mémoire du processus qui les accueille :



Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

© Alfred Park, http://randu.org/tutorials/threads

Néanmoins le thread ne répond pas à tout et notamment **il ne permet pas** le recouvrement d'image (exec()) et limite la portée des communications entre activités à la mémoire de la machine.

Quel est l'intérêt d'avoir une application multiprocessus ?

Une application moderne, si elle est de taille conséquente, aura certainement plusieurs fonctionnalités qu'elle pourrait tenter d'**architecturer en activités séparées** :

- ✔ "gestion des utilisateurs/login",
- ✔ "GUI",
- ✔ "télécommunications",
- ✔ "accès aux persistances fichiers", . . .

Discours de la méthode (Descartes) :

> . . . résoudre la complexité globale en "découpant" . . . en modularisant !

➢ Il y a donc un intérêt dans la **maîtrise de la complexité**.

Mais il peut aussi y avoir un intérêt au niveau des performances :

- ➢ En développant des processus, l'application peut profiter de la concurrence (OS) ou même du parallélisme (machine multi-coeurs+OS) pour que les différentes fonctionnalités s'exécutent "simultanément" donc (peut être) plus vite ! ? . . . ou pas ! ?

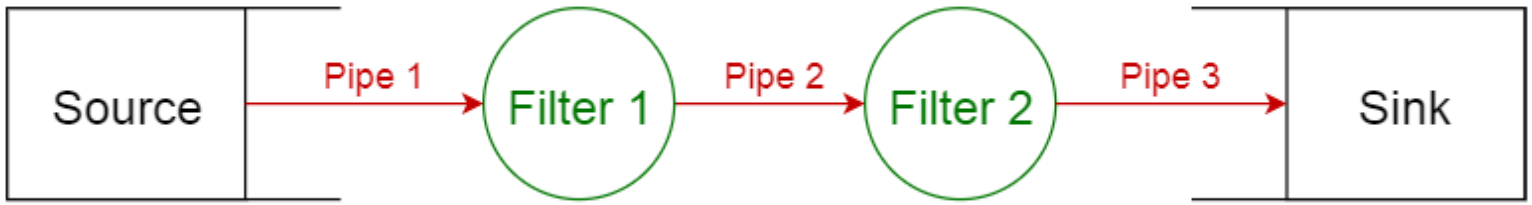
On pourrait aussi évoquer une "sureté" de fonctionnement obtenue par la duplication d'une même activité permettant la mise à l'échelle ou alors la gestion de défaillances (cas des broker MQTT . . .)

Très bien, pour ces raisons et d'autres, l'application sera donc faîte d'**"activités collaboratives"** !

Mais pour que ces activités soient **collaboratives** dans la réalisation d'une application, **il faut qu'elles puissent inter-communiquer** !

On sait que l'on peut faire communiquer deux/plusieurs processus de façon élémentaire :

➢ Le **mécanisme de pipe** présent dans tous les Shells, permet d'enchaîner deux/plusieurs processus en permettant le transfert d'informations entre eux (via la manipulation de leurs fichiers standards).



https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013

Exemple : nombre de lignes sur tous les fichiers LaTeX du répertoire.

```
wc -l *.tex | sort -nr | head -n 1
```

➢ Mais cette architecture d'application (de type "pipe filter pattern") est très restrictive tant l'organisation des activités serait contrainte par une dépendance de données "producteur/consommateur" compatible avec une approche par "filtres" . . . mais pas plus !

Alternatives de communication

Une considération supplémentaire va intervenir dans cette problématique de communication inter-processus :

La mise en réseau des machines !

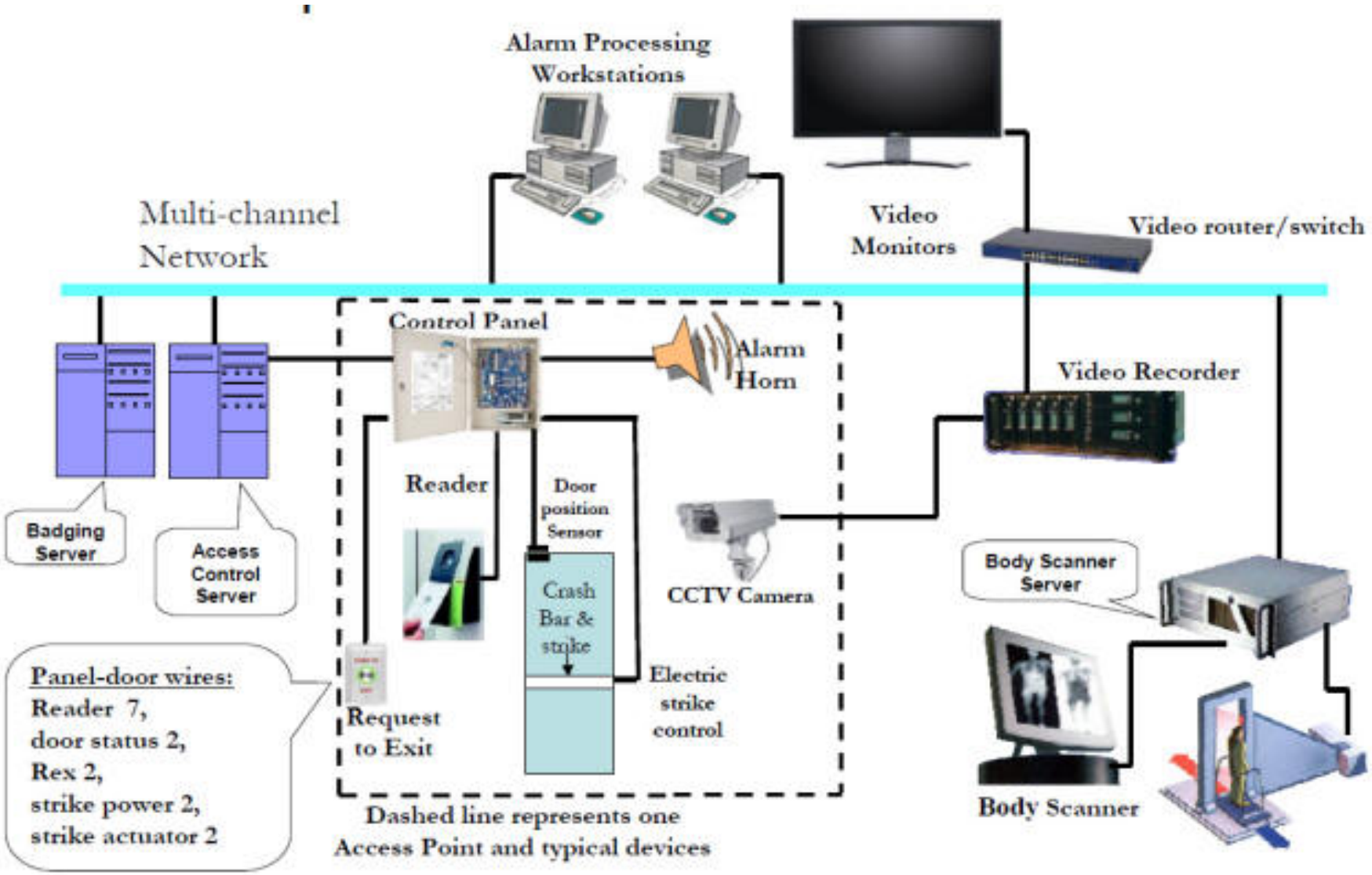
Les communications "inter-processus" pourront donc être :

- ➢ **locales / "intra-machine"** : sur la même machine,
- ➢ **distantes / "inter-machines"** : entre machines sur un réseau . . . Internet puisqu'il ne reste que lui.

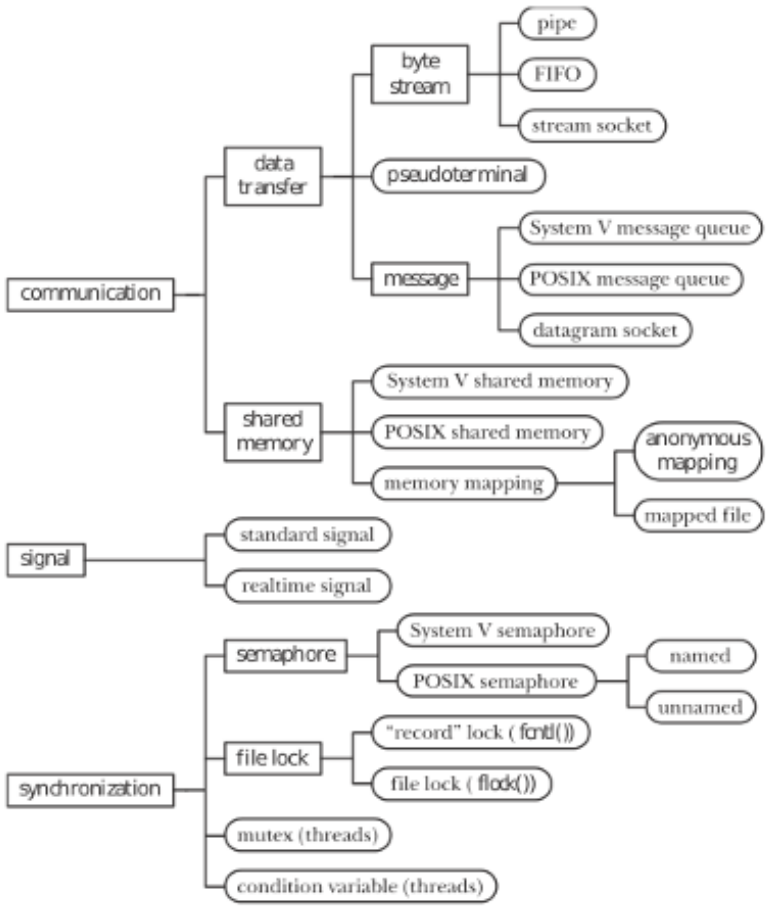
De plus, les applications ne sont plus forcément "en bijection" avec une seule machine.

- ✔ Rien n'empêche d'avoir des morceaux de cette application, les processus, sur différentes machines distantes ! , dédiées ?

Exemple : "Simplified Access Control Architecture"



La suite du cours n'ambitionne pas d'être exhaustive sur les mécanismes de communication possibles !



TLPI, Figure 43-1

Il y en a beaucoup . . . plutôt illustrer des problématiques et des solutions parmi les plus "fréquentes"/classiques.

Pour aller plus loin, les références bibliographiques du domaine sont :

① UNIX Network Programming, Volume 2 : Interprocess Communications, Second Edition.
W. Richard Stevens : 9780130810816

② Advanced Programming in the UNIX Environment.
W. Richard Stevens, Stephen A. Rago

Communications Locales :

Quelques solutions . . .

- Signaux,
- SHM,
- . . .

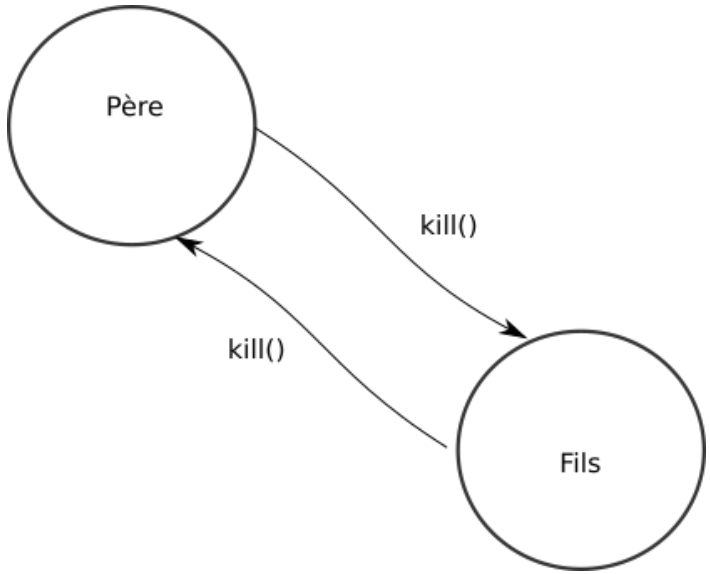
Les signaux

Les signaux peuvent permettre une communication entre processus locaux :

```
/** communication simple entre 2 processus au moyen des signaux
 *  fichier test_kill_signal.c */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
void it_fils(){
  printf("- Je suis rentre, il est sur la table ...\n") ;
  kill (getpid(),SIGINT) ;
}
void fils(){
  signal(SIGUSR1,it_fils) ;
  printf("- Maman, je vais jouer dehors  !!!\n") ;
  while(1) ;
}
int main(){
  int pid ;
  if ((pid=fork())==0)
    fils() ;
  else {
    sleep(3) ; printf("- Fils, peux-tu aller chercher le pain ?\n") ;
    sleep(2) ; kill (pid,SIGUSR1) ;
  }
}
```

C'est essentiellement la communication **d'une synchronisation et d'un entier** :

- "va chercher le pain",
- "voila c'est fait"

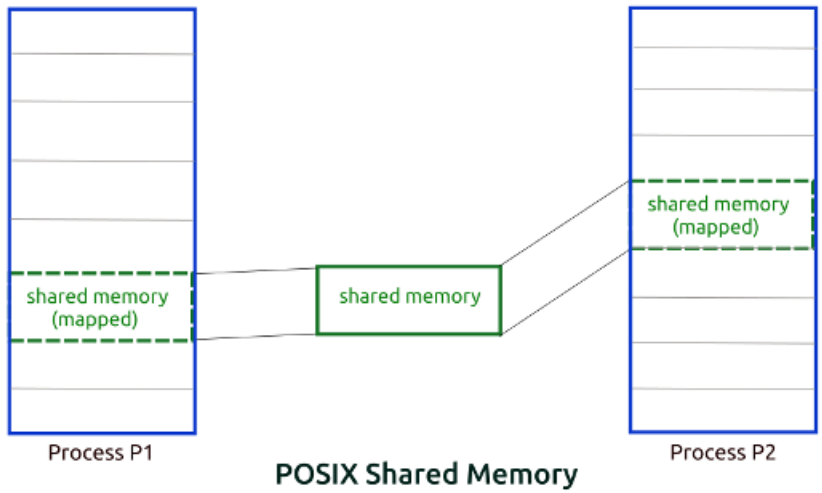



```
menez ~/EnseignementsCurrent/Cours_Ipc/Src$ : ./a.out
- Maman, je vais jouer dehors  !!!
- Fils, peux-tu aller chercher le pain ?
- Je suis rentre, il est sur la table ...
```

C'est déjà ça, . . . mais on aimerait envoyer une information avec plus de sémantique : un message !

La mémoire partagée

On a déjà rencontré cette idée de "**partage d'un segment de mémoire**" : utilisée par le noyau pour "Les bibliothèques partagées".



L'idée est de définir des segments en mémoire (un segment étant constitué d'un ensemble de pages) identifiables dans le système en tant qu'objet et de **permettre à des processus de les attacher à leur espace d'adressage**,

- ➢ c'est à dire d'**associer au premier octet du segment une adresse interprétable dans le contexte du processus**.

Ces segments sont alors des "variables globales" **aux processus** qui les partagent.

La mémoire partagée "by system V"

Comme souvent, on retrouve ce concept Unix en System V et en POSIX.

L'exemple qui suit utilise la version System V (plus simple).

- ➢ Il est très incomplet puisqu'il ne comporte **que** les mécanismes d'échanges.
- ➢ On va vite voir que si l'échange se fait sur la base du partage d'une ressource, il faut d'autres choses pour que cela fonctionne !

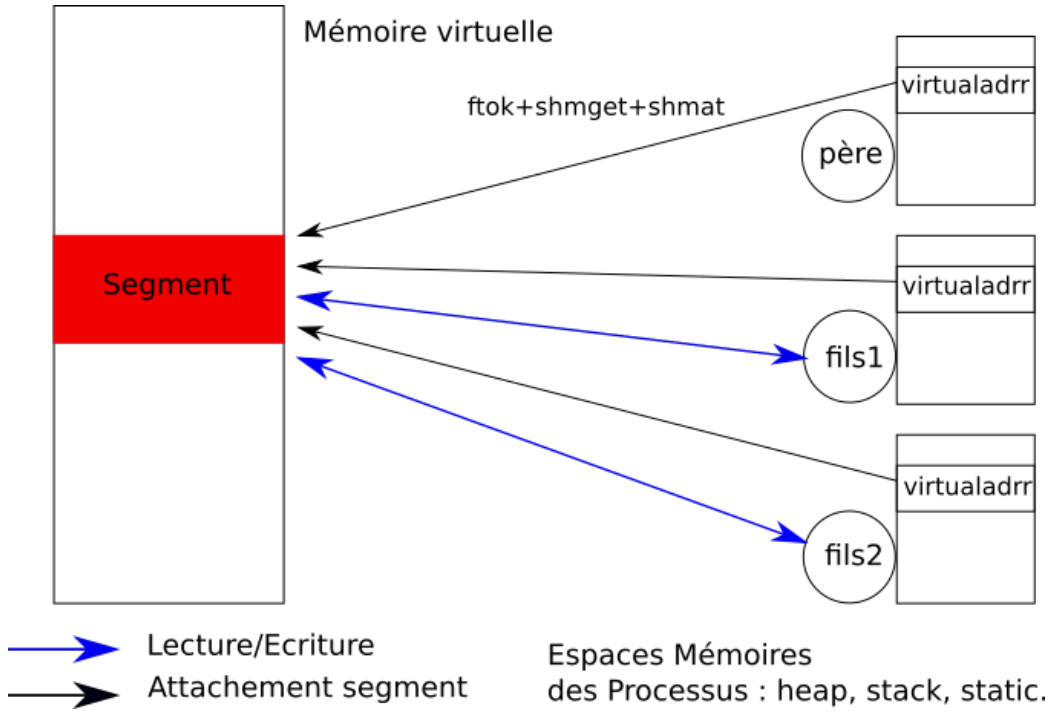
Les fonctions de l'interface System V permettant la mise en oeuvre d'une mémoire partagée sont :

shmget() :	création ou acquisition de l'identification d'un segment
shmat() :	attachement d'un segment
shmdt() :	détachement d'un segment
shmctl() :	opérations de contrôle

ou rajoutera la fonction **ftok()** qui permet de générer les clés (cf exemple).

Analyse de l'exemple qui suit ...

C'est sensé faire quoi ?



① Un processus père associe un segment mémoire à son espace d'adressage.

② Puis il instancie deux fils qui héritent de ce segment.

③ Ensuite les deux fils aimeraient s'échanger leur message (une string),

④ avant de terminer et que le père "récupére" ses fils morts.

```c
/*  Version "-1" d'un échange SHM System V entre deux fils.
    Author : GM
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/shm.h>

#define BUF_SIZE 256

int main(int argc, char *argv[]){
  int i;
  int shmid;
  key_t key;

  char *virtualaddr;   /* Cette variable héritée par tous contient,
                          apres l'appel shmat, un pointeur sur le
                          segment mémoire partagé */

```

```
25      /*-----------------------------------------------------
26       * Etape 1 : attacher la mem partagee à l'espace d'adressage du  pere
27       */
28
29      /* a) Generer une cle (un uint32_t) : Cette clef permettra aux fils
30         et/ou à d'autres processus de retrouver le segment memoire pour
31         l'utiliser */
32      key = ftok(argv[0],'R');
33      /* b) Creation du segment memoire : 1024 octets */
34      shmid = shmget(key, 1024, 0644|IPC_CREAT);
35      if (0 > shmid){ /* La creation peut echouer !!! */
36        perror("Shared Mem creation error\n");
37        exit(1);
38      }
39      /* c) Attachement à l'espace mem du processus :
40         => virtualaddr pointe sur cet espace mem
41         => virtualaddr will be available across fork !!
42      */
43      virtualaddr = shmat(shmid, (void*)0, 0);
44
45      /*-----------------------------------------------------
46       * Etape 2 : Création du fils 1
47       * => Il ecrit une information dans la shm
48       *    puis il lit une information dans la shm
```

```
49      */
50     switch (fork()){
51     case -1:
52       printf("Error forking child 1!\n"); exit(1);
53     case 0: { printf("\nChild 1 executing...\n");
54       char buf[BUF_SIZE];
55       /* Child 1 writing in shared mem : adresse héritee */
56       strcpy (virtualaddr, "Bonjour, Je suis ");
57       sleep(1); /* supposons que la fabrication du message prenne un
58                    certain temps et qu'il soit NON ATOMIC */
59       strcat (virtualaddr, "francais !");
60       printf("Message sent by child 1: %s\n", virtualaddr);
61       /* Child 1 reading from shared mem */
62       strcpy (buf, virtualaddr);
63       printf("Message received by child 1: %s\n", buf);
64       /* printf("Exiting child 1...\n"); */
65       _exit(0);
66     }
67       break;
68     default:
69       break;
70     }
71     /*-----------------------------------------------------
72      * Etape 3 : Création du fils 2
```

```
73       *           Lui lit puis ecrit
74       */
75      switch (fork()){
76      case -1:
77        printf("Error forking child 2!\n"); exit(1);
78      case 0:  { printf("\nChild 2 executing...\n");
79          char buf[BUF_SIZE];
80          /* Child 2 read from shared memory */
81          strcpy (buf, virtualaddr);
82          printf("Message received by child 2: %s\n", buf);
83          /* Child 2 writing in shared mem */
84          strcpy (virtualaddr, "Hello, I'm ");
85          sleep(1); /* supposons que la fabrication du message prenne un
86                       peu de temps et qu'il soit NON ATOMIC */
87          strcat (virtualaddr, "english !");
88          printf("Message sent by child 2: %s\n", virtualaddr);
89          /* printf("Exiting child 2...\n"); */
90          _exit(EXIT_SUCCESS);
91      }
92        break;
93      default: break;
94      }
95      /*-----------------------------------------------------
96       *  Etape 4 : Wait
```

```
97        */
98      printf("Parent waiting for children completion...\n");
99      for(i=0;i<=1;i++){
100       if (wait(NULL) == -1){
101         printf("Error waiting.\n");
102         exit(EXIT_FAILURE);
103       }
104     }
105     printf("Parent finishing.\n");
106
107     /*-----------------------------------------------------
108      * Etape 5 : Deleting Shared Memory.
109      */
110     shmctl (shmid, IPC_RMID, NULL);
111     exit(EXIT_SUCCESS);
112   }
```

Attachement/Détachement du segment mémoire I

① La fonction `ftok( char *pathname, int proj_id)` permet de fabriquer une clé à partir d'un nom de fichier (qui doit exister) et d'un caractère (proj_id).

On peut utiliser n'importe quel nom de fichier, mais il vaut mieux éviter de prendre un fichier trop commun : "/etc/hosts" => mauvais choix :-(

- ✔ D'autres applications pourraient en faire de même et générer un conflit sur l'utilisation de la ressource associée à clé.

Car **cette clé identifie cette mémoire partagée dans le système**.

- ✔ Tout processus, qui utiliserait le même nom de fichier et le même proj_id pourrait ainsi communiquer avec les processus partageant la même clé et utilisant déjà la mémoire partagée associée.

② Muni de cette clé, on peut ensuite faire l'acquisition (`shmget`) d'un segment de mémoire d'une taille spécifiée : 1024 octets dans l'exemple.

Attachement/Détachement du segment mémoire II

③ Et lorsque l'on a ce segment, venir l'attacher (`shmat`) à l'espace mémoire du processus. On peut laisser le système choisir l'adresse à laquelle se fera l'attachement : (void*)0 en second paramètre.

④ Les accés (R/W) sont ensuite réalisées comme sur une variable classique.

⑤ La fermeture de la mémoire partagée consiste à supprimer le segment associé à la clé :

```
shmctl (shmid, IPC_RMID, NULL);
```

L'échange, le partage et donc les problèmes ...

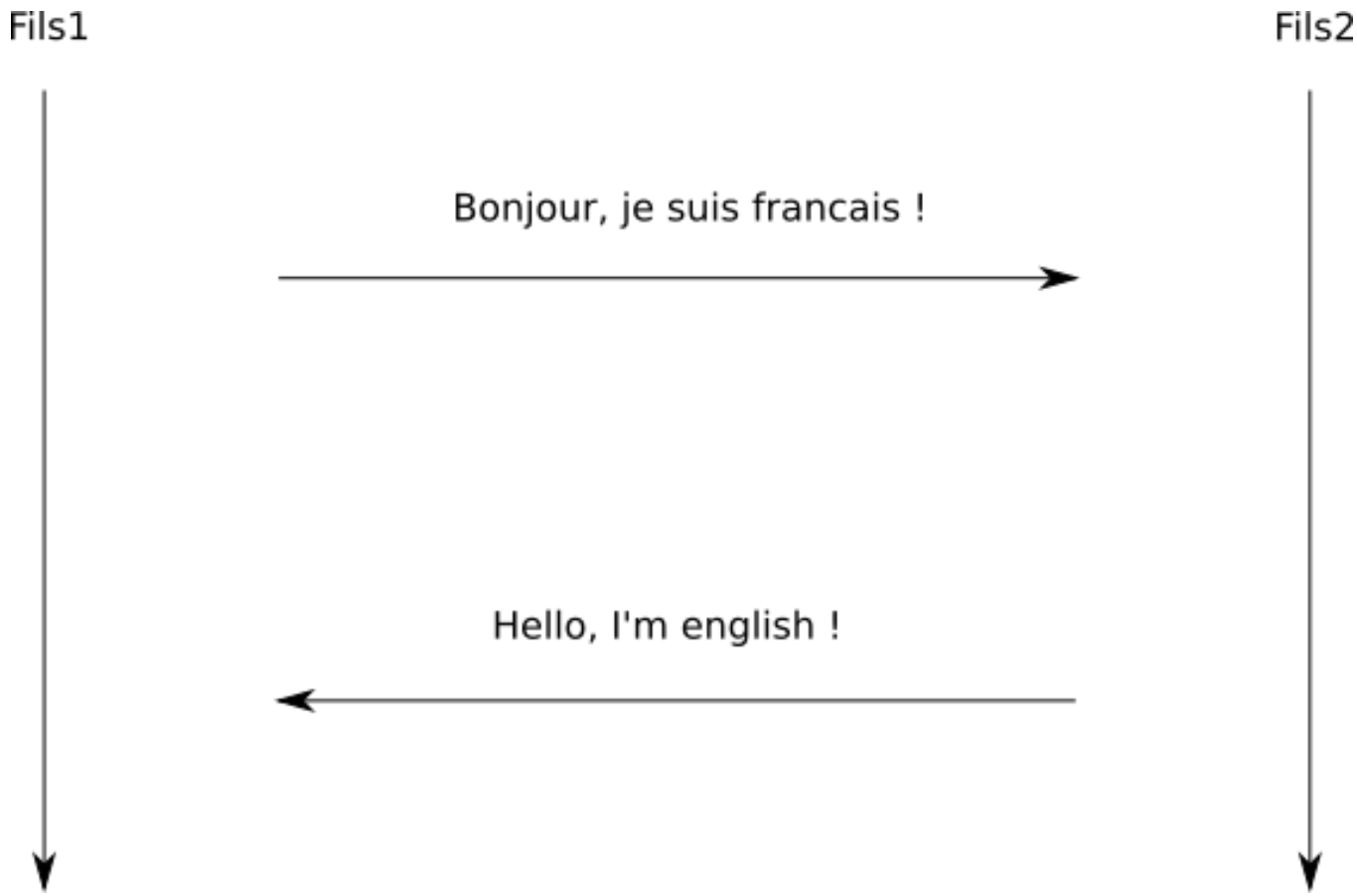
Vous avez déjà rencontré les problèmes induits par des activités concurrentes : cf TP1.

① Le premier problème est celui de l'atomicité des traitements.

Si chaque activité utilise une ressource partagée pour effectuer un traitement sur plusieurs instructions, il est fort probable que le séquencement prévu des effets de bords de ce traitement **ne supportent pas d'être "mélangés"** avec les effets de bords provoqués par une/plusieurs traitements des autres activités.

C'est le problème de la "variable globale" !

Moi je voudrais . . .



Sans atomicité, j'ai ...

```
menez ~/EnseignementsCurrent/Cours_Ipc/Src/Shm$ : gcc shm_sysv_withoutsem.c
menez ~/EnseignementsCurrent/Cours_Ipc/Src/Shm$ : ./a.out
Parent waiting for children completion...

Child 1 executing...

Child 2 executing...
Message received by child 2: Bonjour, Je suis
Message sent by child 1: Hello, I'm francais !
Message sent by child 2: Hello, I'm francais !english !
Message received by child 1: Hello, I'm francais !english !
Parent finishing.
```

Les messages reçus ne sont pas cohérents, du fait des inter-agissements "incontrôlés" des deux processus fils sur la mémoire partagée (où se trouve le message).

En fait, je l'ai un peu cherché en réalisant l'écriture en SHM en deux étapes ... : strcpy puis strcat.

➢ L'ordonnanceur risque fort de préempter le processeur entre ces deux étapes ... ce qu'il fait !

Sémaphores :

- Synchroniser . . .

Les mutex et les sémaphores

En tant qu'activité, la réponse à ce problème de partage consiste à gérer l'accès à la ressource et éviter ainsi les "mélanges" ou les contenus incohérents.

Cette gestion/contrôle peut consister à s'**approprier** puis **libérer** l'accès : c'est l'accès exclusif.

➢ La ressource est partagée . . . mais chacun y accède à son tour !

Pour réaliser cela, il y a donc des solutions :

✔ "verrou/mutex lock" "plutôt" pour les threads et

✔ le "sémaphore" qui est une généralisation des mutex.

Sémaphores POSIX

Plutôt que d'énumérer des notions et des fonctions abstraites, nous allons décrire comment on pourrait utiliser les sémaphores pour résoudre "notre problème".

➢ Le sémaphore est ici utilisé pour résoudre le problème d'exclusion mutuelle : cf sémaphore de Dijkstra (P : acquisition / V : libération).

La solution proposée utilise les "sémaphores nommés" : `/nom_sémaphore` .

Le nom du sémaphore est utilisé lors de la création et de l'ouverture du sémaphore :

✔ Il permet **aux différents processus** souhaitant l'utiliser de récupérer un pointeur vers le sémaphore ainsi nommé.

Ce nom désigne dans le système d'exploitation **"ce"** sémaphore : mécanisme de "rendez-vous".

Il existe aussi les sémaphores "non nommés".

Principe et Primitives de manipulation

Le sémaphore est une "boîte à clefs identiques" permettant d'ouvrir une porte :

- `sem_wait()` permet de récupérer une clef (i.e demande d'acquisition), à condition qu'il y en ait une. Sinon (pas de clef) la fonction est bloquante.

  Du coup, on ne peut pas "ouvrir la porte" . . . on doit attendre qu'une clé soit placée/remise dans la boîte. A ce moment, on la prendra !

- `sem_trywait()` est la version non bloquante.

- `sem_post()` permet de placer une clef dans la boite (i.e demande de libération du sémaphore).

  Cette fonction rend le nombre de clefs présentes ou manquantes (nombre de processus bloqués).

- `sem_getvalue()` permet de savoir combien il y a de clefs dans le sémaphore.

➢ `sem_unlink()` : Lorsque l'on a fini, il faut supprimer le sémaphore.

Il est important de gérer correctement cette phase :

➢ Lorsque vous nommez un sémaphore (mais c'est vrai pour tous les objets qui vont être partagés dans le système), l'OS utilise un fichier spécial pour permettre notamment le partage (en ouverture) de son existence entre père, fils et peut être d'autres.

Sous Linux, ces fichiers apparaissent dans "/dev/shm" et vous pouvez prendre note de leurs autorisations.

➢ Si un autre processus, notamment d'un autre utilisateur, venait à vouloir créer (/CREATED) ces mêmes fichiers . . . il y aurait un problème de partage !

➢ De plus, si votre application venait à sortir sans faire du ménage (unlink ou rm) au niveau de ces fichiers, ces "noms" ne pourraient plus être utilisés par d'autres.

➢ Le "exit()" ne viendra pas supprimer un fichier créé dans le système de fichier.

➢ Enfin n'oubliez pas que le système gère un fichier ouvert dans des tables et qu'il disparaît de ces tables lorsque la dernière référence au fichier se ferme . . . avec un petit délai.

```c
/*-------------------------------------------------
   Version 0 d'un échange SHM System V entre deux fils.
    Les semaphores utilisés sont POSIX !

    Author : GM
 -------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/sem.h>
#include <semaphore.h>
#include <sys/shm.h>
#include <time.h>
#define BUF_SIZE 256

int main(int argc, char *argv[]){
  int i;
  char *virtualaddr;
  int shmid;
  sem_t *s;
  key_t key;

  /*----- Attaching the shared mem to my address space  */
  key = ftok(argv[0],'R'); /* Generation de la key */
  shmid = shmget(key, 1024, 0644|IPC_CREAT); /* Creation du segment
                                                memoire : 1024 octets */
  if (0 > shmid){
    perror("Shared Mem creation error\n");
    exit(1);
  }
  /* => virtualaddr will be available across fork ! */
  virtualaddr = shmat(shmid, (void*)0, 0); /* Attachement à l'espace mem du processus */
```

```c
/*--- Create POSIX Named Semaphores, and initialising with 1 */
int init_sem_value = 1; /* Dijkstra sem */
s = sem_open("/toto", O_CREAT|O_RDWR, 0644, init_sem_value);

switch (fork()){ /*----- child 1 */
case -1:
  printf("Error forking child 1!\n");  exit(1);
case 0:
  {
    char buf[BUF_SIZE];

    /* Referring the semaphore */
    s = sem_open ("/toto", O_RDWR);

    printf("\nChild 1 executing...\n");

    /*Child 1 writing in shared mem */
    sem_wait(s);
    strcpy (virtualaddr, "Bonjour, Je suis ");
    sleep(1); /* La fabrication du message prend un peu de temps */
    strcat (virtualaddr, "francais !");
    printf("Message sent by child 1: %s\n", virtualaddr);
    sem_post(s);

    /*Child 1 reading from shared mem */
    sem_wait(s);
    strcpy (buf, virtualaddr);
    printf("Message received by child 1: %s\n", buf);
    sem_post(s);

    /*printf("Exiting child 1...\n"); */
    _exit(0);
  }
  break;
default: break;
}
```

```
switch (fork()){ /*----- child 2 */
case -1:
  printf("Error forking child 2!\n"); exit(1);
case 0:
  {
    char buf[BUF_SIZE];

    /* Referring the semaphore */
    s = sem_open ("/toto", O_RDWR);

    printf("\nChild 2 executing...\n");

    /*Child 2 reading from shared memory*/
    sem_wait(s);
    strcpy (buf, virtualaddr);
    printf("Message received by child 2: %s\n", buf);
    sem_post(s);

    /*Child 2 writing in shared mem*/
    sem_wait(s);
    strcpy (virtualaddr, "Hello, I'm ");
    sleep(1); /* La fabrication du message prend un peu de temps*/
    strcat (virtualaddr, "english !");
    printf("Message sent by child 2: %s\n", virtualaddr);
    sem_post(s);

    /*printf("Exiting child 2...\n");*/
    _exit(EXIT_SUCCESS);
  }
  break;
default:
  break;
}
```

```
  printf("Parent waiting for children completion...\n");
  for(i=0;i<=1;i++){
    if (wait(NULL) == -1){
      printf("Error waiting.\n");
      exit(EXIT_FAILURE);
    }
  }
  printf("Parent finishing.\n");

  //Deleting semaphores..
  sem_unlink ("/toto");

  //Deleting Shared Memory.
  shmctl (shmid, IPC_RMID, NULL);
  exit(EXIT_SUCCESS);

}
```

Analyse du code

Dans l'exemple, on voit bien :

① le père **créer** (option CREATE et RW) le sémaphore : "/toto"

```
sem_t *s;
int init_sem_value = 1; /* Used as mutex */
s = sem_open("/toto", O_CREAT|O_RDWR, 0644, init_sem_value);
```

Puisque le sémaphore peut être partagé, il y a des autorisations !

② chaque fils **"récuperer"** ce sémaphore dans l'OS pour pouvoir mettre en oeuvre la synchronisation.

```
s  = sem_open ("/toto", O_RDWR); /* Referring the semaphor */
```

La récupération se base sur le nom !

Synchronisation par sémaphores

On a résolu le problème de l'intégrité/atomicité des messages, mais **ce n'est pas suffisant !**

```
menez ~/EnseignementsCurrent/Cours_Ipc/Src/Shm$ : gcc shm_sysv_withsem.c -lpthread
menez ~/EnseignementsCurrent/Cours_Ipc/Src/Shm$ : ./a.out
Parent waiting for children completion...

Child 1 executing...

Child 2 executing...
Message sent by child 1: Bonjour, Je suis francais !
Message received by child 1: Bonjour, Je suis francais !
Message received by child 2: Bonjour, Je suis francais !
Message sent by child 2: Hello, I'm english !
Parent finishing.
```

On a bien essayé de placer les lectures et écritures des processus fils de façon "cohérente", mais sur la figure on constate que l'enfant 1 ne reçoit pas le bon message.

- Le séquencement des accès (désormais atomiques) n'est pas bon !
- L'enfant 1 lit trop vite . . . en tout cas avant que l'enfant 2 ait eu le temps d'écrire.

Ce n'est parce que vous avez résolu le problème de l'atomicité (synchronisation de l'accès) que le comportement global est correct.

Ce second problème est celui de la **synchronisation des échanges**.

Par exemple, même si les accès sont atomiques, échanger un message au travers d'une ressource, nécessite de respecter un "ordre" :

① Le premier dépose son message,

② Le second lit ce message,

③ Le second dépose son message,

④ Le premier lit ce message.

⑤ ...

C'est un exemple de ce qu'est une synchronisation d'échange !

Là aussi, **les sémaphores peuvent aider !**

➢ On voit cela en TD.

Tubes :

- Une solution différente des SHM pour
- Echanger des messages, ...
- Avec synchronisation !

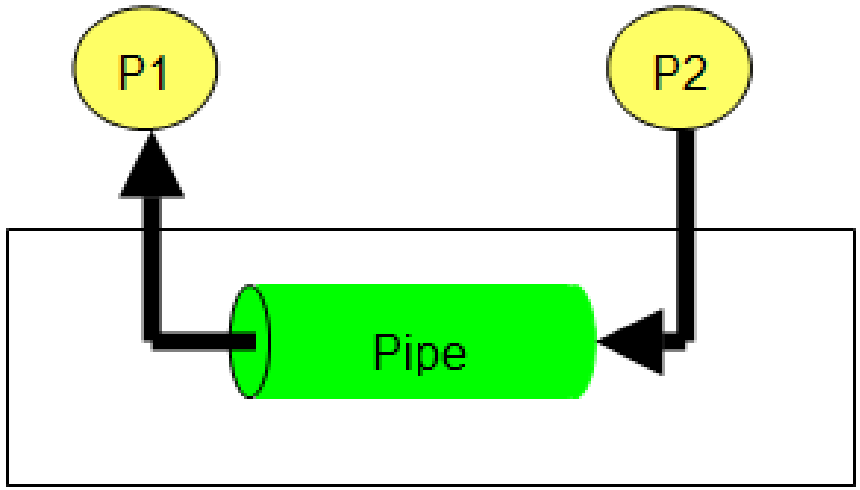
Tubes de communications / "Ordinary Pipes"

Tubes de communications / "Ordinary Pipes"

Les **pipes** (ou conduits, ou tubes) constituent un mécanisme fondamental de communication **unidirectionnelle** entre processus Unix.

Ce sont des "files" de caractères (FIFO : First In First Out) :

- ➢ les informations y sont introduites à une extrémité et en sont extraites à l'autre.



Les "conduits" sont implémentés comme les fichiers (ils possèdent un i-node), même s'ils n'ont pas de nom dans le système.

Caractéristiques des Pipes

Le pipe est le plus vieux des IPC Unix :

- ✔ C'est un mécanisme de transfert de données (byte stream) entre processus.

  Le byte stream écrit d'un coté du pipe peut être lu de l'autre côté.

- ✔ Il n'y a pas de mécanisme de "rendez vous" comme on en a rencontré avec les "keys" des SHMs.

  On demande directement un pipe au noyau ... par son nom (si il en a un).

- ✔ Une fois créé, le pipe est référencé par des descripteurs de fichiers.

- ✔ Les pipes sont "processus-persistants" et disparaissent lorsque les processus relatifs disparaissent.

- ✔ Les pipes sont portables => disponibles sur TOUTES les versions d'Unix.

"Ordinary Pipes" : pas de noms ! I

Sans nom, ce type d'objet ne peut être utilisé que par des processus qui partagent une zone mémoire ou qui ont une filiation commune permettant d'avoir hérité du pipe.

➢ Il y aussi la notion de "tube nommé" / "named pipe" . . . facilite le RDV (cf sémaphore).

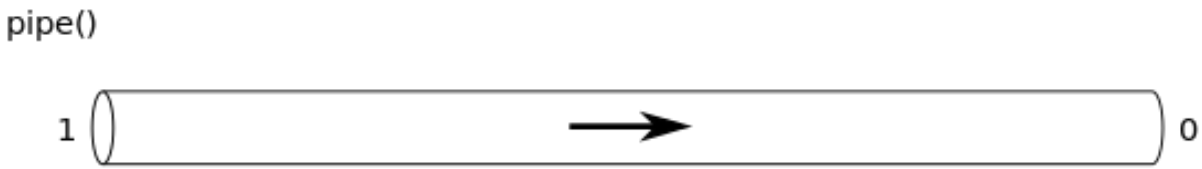
```c
/* Exemple de tube/pipe :
   Sans nommage et Processus unique (il se parle à lui même).
   Author : GM */

#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25

/* Petit pense bête : 0 pour lire et 1 pour écrire */
#define READ_END 0
#define WRITE_END 1
```

"Ordinary Pipes" : pas de noms ! II

```c
int main(int argc, char *argv[]){

  /* CREATION DU pipe */
  int fd[2]; /* 1 pipe = deux extremites : une pour lire / une pour écrire */
  if (pipe(fd) == -1) {
    fprintf(stderr,"Pipe failed");
    return 1;
  }

  char msg[BUFFER_SIZE] = "Bienvenu !";
  /* Write to the pipe */
  write(fd[WRITE_END], msg, strlen(msg)+1);
  printf("Le proc écrit : %s\n", msg);
  fflush(stdout);
  /* RAZ msg in stack */
  memset(msg, 0, BUFFER_SIZE);
  /* Read the pipe */
  read(fd[READ_END], msg, BUFFER_SIZE);
  printf("Le proc lit : %s\n",msg);
  fflush(stdout);
  /* Close the pipe */
  close(fd[READ_END]);
  close(fd[WRITE_END]);
  return 0;
}
```

Analyse du code

Le processus s'écrit à lui même : write(fd[1] . . .) puis read(fd[0] . . .)



. . . la preuve que le système joue le rôle de "buffer".

Dans ce qui suit, on reprend cet exemple mais avec deux processus : le père et son fils.

le "parent" écrit, et le "child" lit.

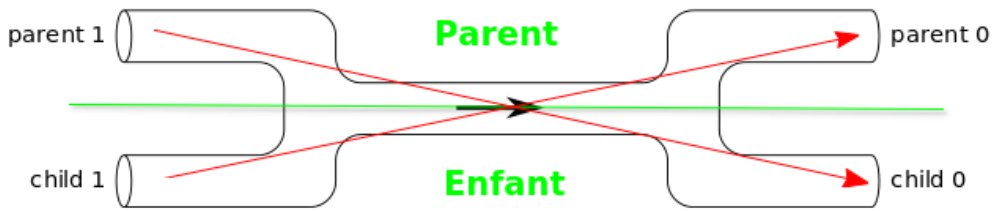
Avec deux processus

L'occurence du fork créant le fils duplique les descripteurs d'accés au pipe, mais le pipe est partagé !

Si "pipe" est la variable correspondant au tube (dans le père) alors

- ➢ $pipe[1]_{parent}$ et $pipe[1]_{child}$ permettent d'écrire dans le pipe
- ➢ $pipe[0]_{parent}$ et $pipe[0]_{child}$ permettent de lire dans le pipe

Ce qui correspond au schéma suivant :



Avec un pipe, on ne peut écrire que dans un sens . . . toujours du 1 au 0 !

- Du parent 1 au parent 0 ou au child 0
- ou du child 1 au child 0 ou au parent 0

"Ordinary Pipes" : entre deux processus. I

Le code qui suit montre comment ce schéma doit être paramétré pour permettre un transfert du père au fils.

```c
/* Exemple de tube/pipe : Le pere écrit un message à son fils.
   Author : GM
*/

#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

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

  /* CREATION DU pipe -------------------------*/
  int fd[2]; /* => deux extremites : lire/écrire */
  if (pipe(fd) == -1) {
    fprintf(stderr,"Pipe failed");
    return 1;
  }
```

"Ordinary Pipes" : entre deux processus. II

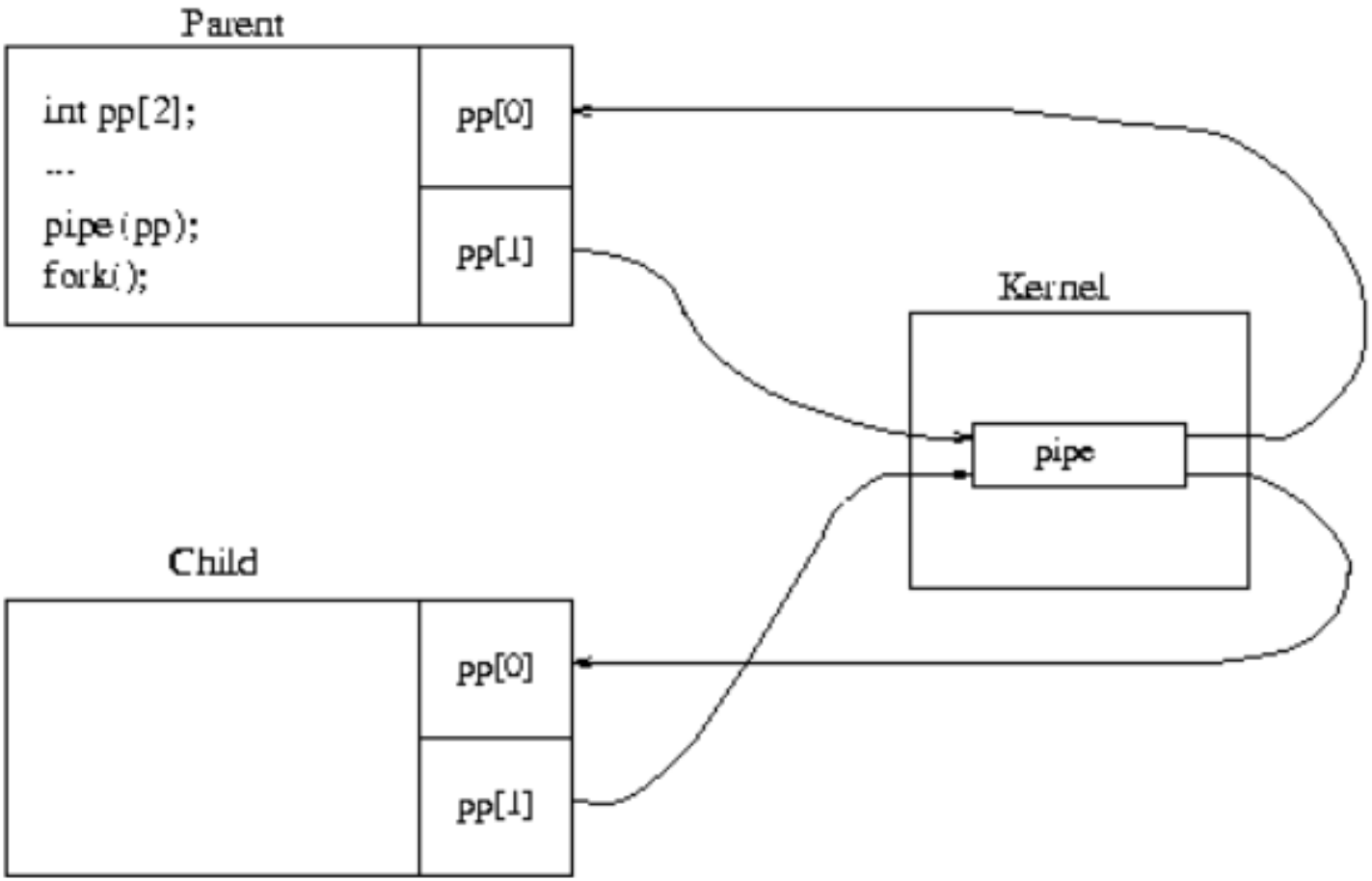
```c
/* fork a child process */
pid_t pid;
pid = fork();
if (pid < 0) { /* error occurred */
  fprintf(stderr, "Fork Failed"); return 1;
}

if (pid > 0) { /* parent process  : WRITE */
  int status;
  char msg[BUFFER_SIZE] = "Bienvenu fils !";

  /* close the unused end of the pipe */
  close(fd[READ_END]);

  /* write to the pipe */
  write(fd[WRITE_END], msg, strlen(msg)+1);
  printf("Le père écrit : %s\n", msg);
  fflush(stdout);

  /* close the write end of the pipe */
  close(fd[WRITE_END]);

  wait(&status); /* Eviter le zombie ! */
}
```

"Ordinary Pipes" : entre deux processus. III

```
  else { /* child process : READ */
    char msg[BUFFER_SIZE];

    /* close the unused end of the pipe */
    close(fd[WRITE_END]);

    /* read from the pipe */
    read(fd[READ_END], msg, BUFFER_SIZE);
    printf("Le fils lit : %s\n",msg);
    fflush(stdout);

    /* close the write end of the pipe */
    close(fd[READ_END]);
  }
  return 0;
}
```

Analyse du code

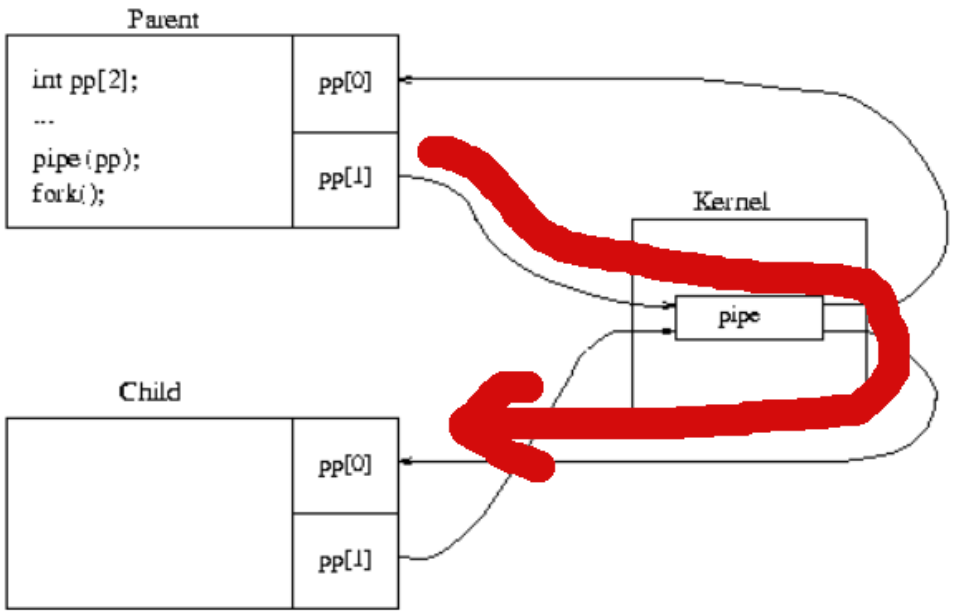
Le pipe créé par la fonction `int pipe(int pipefd[2])` résulte en un couple d'extrémités : pipefd[0] et pipefd[1].



Comme on retrouve ce pipe (et donc ces mêmes extrémités) dans le père et dans le fils (par héritage), chaque processus va utiliser une des deux extrémités. Dans cet exemple/code, le "père écrit au fils" :

① `fd[1]` l'**"extrémité pour écrire" sera utilisée par le père pour envoyer son message**.

② `fd[0]` l'**"extrémité pour lire" sera utilisée par le fils pour lire son message**.



Pour permettre la communication dans l'autre sens, il faudrait un autre pipe !

Au niveau des SHELLS

La technique du pipe est fréquemment mise en oeuvre dans le shell pour rediriger la sortie standard d'une commande sur l'entrée d'une autre (commande pipe : symbole |).

Le code qui suit montre la réalisation de la commande SHELL suivante :

ls -1 | wc -l

Vous retrouvez des instructions connues :

① Le père engendre (/fork) deux processus fils avec les images/codes des commandes `ls` et `wc` .

② **Ce qui est intéressant et nouveau**, c'est l'utilisation de la fonction `dup2` .

Grâce à cette fonction, les **extrémités du tube prennent la place des fichiers standards des processus**.

➢ Ce qui aurait du s'écrire/lire sur un fichier standard (écran ou clavier) du processus passe désormais dans le tube.

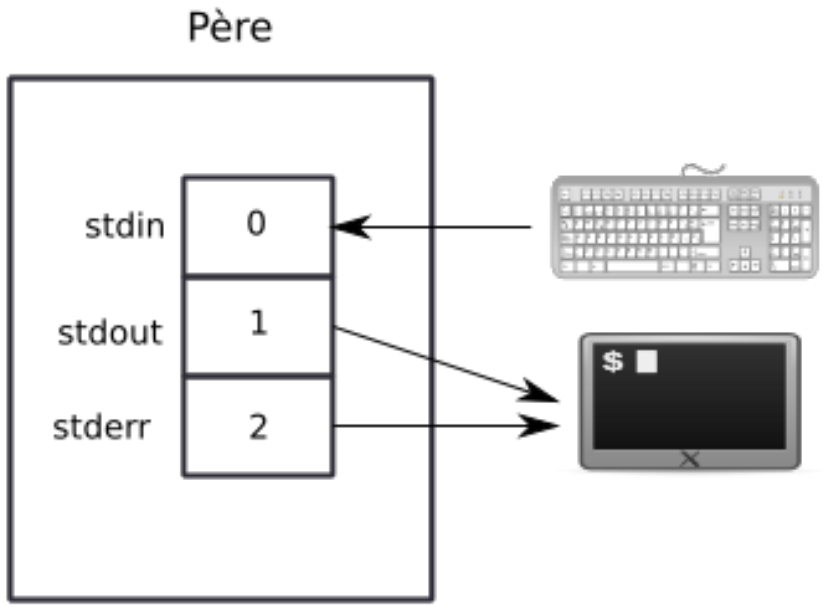
```c
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
void eliminer_zombie(int sig);  /* Handler du signal SIGCHLD */

int faire_ls(int pfd[2]){  printf("running ls %d\n",getpid());
  close(STDOUT_FILENO);
  close(pfd[0]); /* close the unused read side */
  dup2(pfd[1], STDOUT_FILENO); /* connect the write side with stdout */
  close(pfd[1]); /* close the write side
            => can be used to stop reading by next cmd in pipe : EOF */
  execlp("ls", "ls", "-1", (char *)0);   /* execute the process (ls command) */
  printf("ls failed"); /* if execlp returns, it's an error */
  return 1;
}
int faire_wc(int pfd[2]){  printf("running wc %d\n",getpid());
  close(STDIN_FILENO);
  close(pfd[1]); /* close the unused write side */
  dup2(pfd[0], STDIN_FILENO); /* connect the read side with stdin */
  close(pfd[0]); /* close the read side */
  execlp("wc", "wc", "-l", (char *) 0);   /* execute the process (wc command) */
  printf("wc failed"); /* if execlp returns, it's an error */
  return 2;
}
```

```c
int main(int argc, char *argv[]){
  int pfd[2]; /* creation du tube */
  if (pipe(pfd) == -1)  perror("impossible de creer le tube") ;

  signal(SIGCHLD, eliminer_zombie);
  printf("pere %d\n",getpid());
  if (fork() == 0){ /* Le fils ls */
    faire_ls(pfd);
  }
  else { /* Le pere */
    if (fork() == 0){ /* Le fils wc */
      faire_wc(pfd);
    }
    else { /* Le pere */
      close(pfd[0]);
      close(pfd[1]);
      sleep(1); /* Le temps que le signal .. */
      exit(0);
    }
  }
}
void eliminer_zombie(int sig){  /* Handler du signal SIGCHLD */
  int exit_cond;
  pid_t pid;
  printf("Attente de la terminaison du fils...\n");
  pid = wait(&exit_cond); /* Pid du fils */
  if (WIFEXITED(exit_cond))
    printf("%d : termine correctement : %d\n", pid, WEXITSTATUS (exit_cond));
```

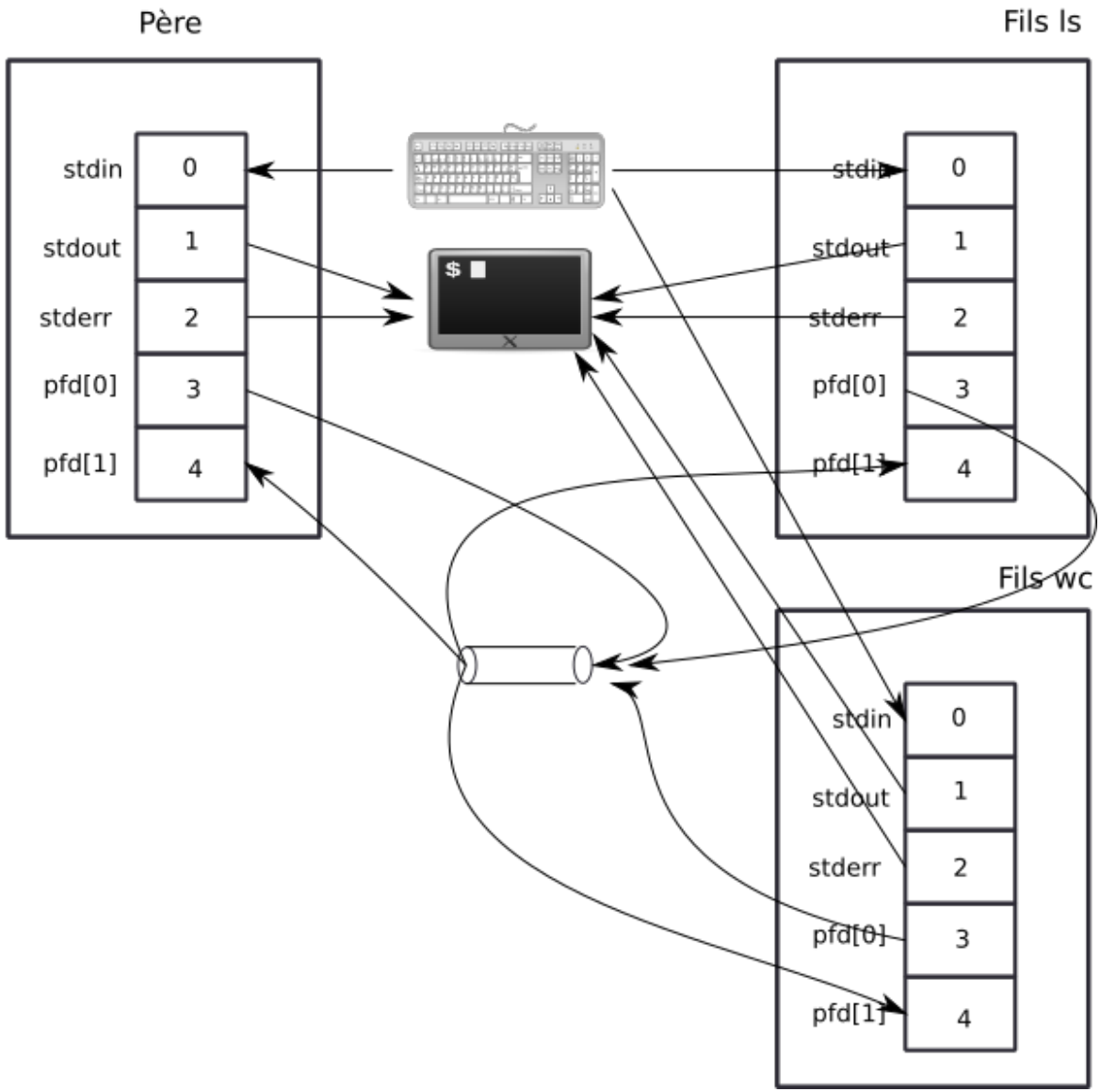
```
  else
    printf("%d : mal termine : %d\n", pid,WTERMSIG(exit_cond));
}
```

La bande dessinée qui va avec . . .

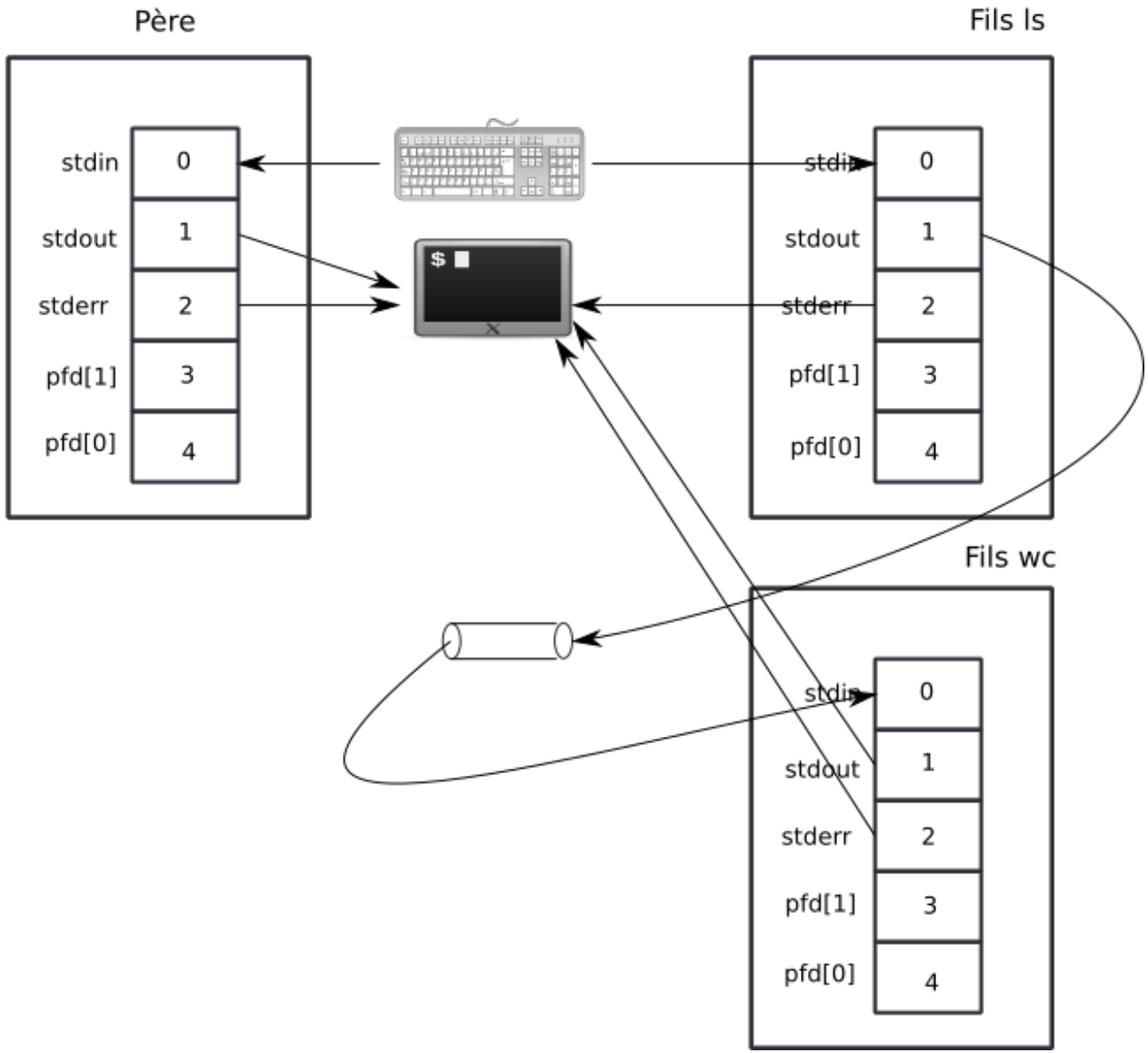
Faut pas s'emmêler les chemins . . .



Post fork :


Père
stdin
stdout
stderr
pfd[0]
pfd[1]
0
1
2
3
4
Fils ls
stdin
stdout
stderr
pfd[0]
pfd[1]
0
1
2
3
4
Fils wc
stdin
stdout
stderr
pfd[0]
pfd[1]
0
1
2
3
4

Post dup2/close :



Encore beaucoup de choses à dire . . .

① Le pipe étant unidirectionnel, pour faire dialoguer deux processus par tube, il est nécessaire d'ouvrir deux tubes, et de les utiliser l'un dans le sens contraire de l'autre.

② Il existe des pipes nommés . . . même objectif que la "key" comme pour la SHM . . . récupéré un pipe existant !

③ On verra dans le cadre des sockets la gestion des accès : read et write.

Histoire de vous sensibiliser à quelques problèmes communs aux échanges entre processus :

- ✔ Qui a dit que le nombre d'octets transférées (et rendues par les fonctions) est toujours celui donné en argument ?

  https://stackoverflow.com/questions/37355656/can-posix-read-receive-less-than-requested-4-bytes-from-a-pipe

- ✔ On "read" jusqu'à quand ?

Gestion de la fermeture

Il est important de fermer les extrémités **non utilisées** du pipe avant de commencer à l'utiliser.

Il est aussi important de gérer la terminaison des extrémités **lorsque la communication se termine** :

① Lire depuis un pipe dont l'extrémité est fermée retourne un 0.

Un peu comme un EOF ...

② Ecrire dans un pipe dont l'extrémité de lecture est fermée rend -1 avec une errno => EPIPE.

Un signal SIGPIPE est envoyé au processus : terminaison du processus ?

La suite en TP ...

Index :