Communication inter-processus

1/28

- Les tuyaux
- La redirection
- Sélection
- Projets

Se souvenir

2(1)/28

Qu'est-ce qu'un zombi ?

Différence entre fork et exec ?

Double fork : En quoi ça consiste, à quoi ça sert ?

Signaux et appels systèmes :

Interaction et conséquence ? Exemple. Quand est-on tranquille ?

Un processus peut-il compter les signaux reçus ?

Un besoin

3/28

Inventaire de la communication inter-processus

- Les signaux : une constante, communication asynchrone. Superposition possible (perte d'information).
- La valeur de retour : du père vers le fils uniquement.
- Les fichiers.
  - un processus écrit dans un fichier,
  - un autre lit dans le même fichier.

Défaut de la communication par fichier

- Lorsque le lecteur attend, il ne sait pas si l'écrivain a fini, ou calcule d'autres données.
- Une fois écrites et lues, les données sont inutiles, mais restent en mémoire.

Solution : les tuyaux

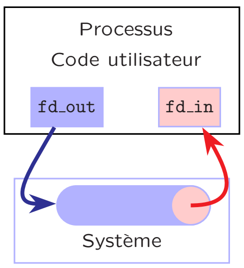
Les tuyaux (aussi appelés tubes)

4/28

Ce sont des fichiers anonymes, non persistants, de taille bornée, avec un écrivain à un bout et un lecteur à l'autre bout.

```
pipe: unit -> file_descr * file_descr
```

Retourne une paire de descripteurs `fd_in`, `fd_out`



Le tuyau est un tampon géré en file d'attente FIFO :

- Les processus en écriture remplissent le tuyau.
- Ils sont bloqués lorsque le tuyau est plein.
- Ceux en lecture vident le tuyau.
- Ils sont bloqués lorsque le tuyau est vide.

Attention !

Remarque

5/28

Les descripteurs de tuyaux, comme les descripteurs de fichiers, peuvent être mis en position non-bloquants. Pour changer l'état d'un descripteur ouvert, on utilise les appels :

```
set_nonblock : file_descr -> unit
clear_nonblock : file_descr -> unit
```

La lecture dans un tuyau vide ou l'écriture dans un tuyau plein échouent avec l'erreur `EWOULDBLOCK` (au lieu de bloquer) lorsque le descripteur associé est dans l'état non bloquant.

Intérêt

6(1)/28

Que se passe-t-il si c'est le même processus qui lit et écrit ?
— Aucun intérêt et risque de blocage s'il lit avant d'écrire.

Typiquement

- Les tuyaux sont créés avant une fourche (`fork`).
- Après le fork, le fils et le père tiennent chacun les deux bouts du tuyau (les descripteurs sont copiés).
- L'un des processus parle dans un bout et l'autre écoute dans l'autre bout.
- Chacun ferme le bout qu'il n'utilise pas, libèrant ainis le descripteur inutile.

Pour parler dans les deux sens...

On doit utiliser deux tuyaux !

Fermeture d'un tuyaux

7/28

Un tuyaux est fermé lorsque tous les descripteurs sont fermés aux deux bouts.

Lecture dans un tuyau fermé

Elle peut se poursuivre jusqu'à ce que le tuyaux soit vide. À ce moment le lecteur reçoit une fin de fichier, i.e. le système lui retourne 0 caractère en écriture. (Les lectures tamporisées lèvent l'exception `End_of_file`.)

Écriture dans un tuyau fermé

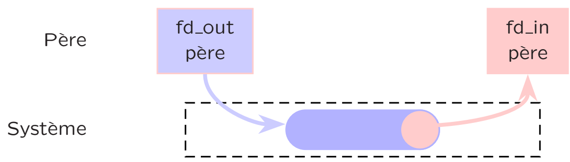
Elle n'est pas possible :

- Le processus écrivain reçoit le signal `sigpipe` si celui-ci a son comportement par défaut, qui est de tuer le processus.
- Si le signal est ignoré ou redéfini, l'écriture échoue avec une erreur `EPIPE` (`Sys_error` pour les lectures temporisées).

Typiquement

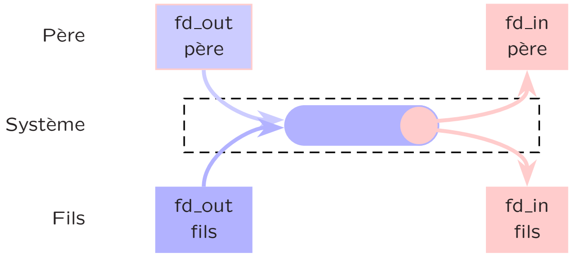
8(1)/28

```
let (fd_in, fd_out) = pipe() in
match fork() with
  0 -> close fd_in; ... write fd_out ...
| k -> close fd_out; ... read fd_in ...
```



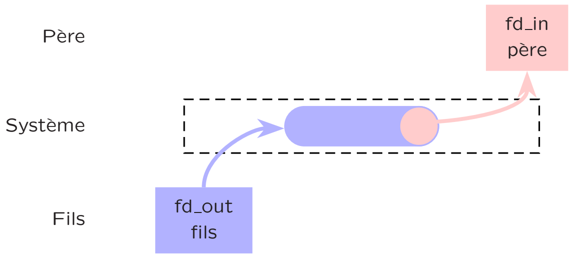
Typiquement


```
let (fd_in, fd_out) = pipe() in
match fork() with
  0 -> close fd_in; ... write fd_out ...
| k -> close fd_out; ... read fd_in ...
```

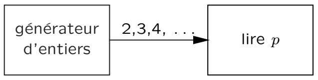


Typiquement


```
let (fd_in, fd_out) = pipe() in
match fork() with
  0 -> close fd_in; ... write fd_out ...
| k -> close fd_out; ... read fd_in ...
```

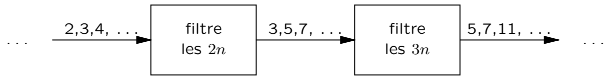


Exemple : crible d'Eratosthène parallèle



Le premier processus **filtre** lit $p = 2$, puis il crée un nouveau processus filtre, connecté à sa sortie, et il se met à filtrer les multiples de p depuis son entrée ;

Le filtre $p = 3$ affiche 3, et se met à filtrer les multiples de 3, *etc.*



En pratique, on ne lance un nouveau processus que tous les 1000 nombres premiers. Un processus implémente donc le comportement de 1000 filtres.

Les tuyaux nommés

Les tuyaux sont anonymes

Ils sont donc limités pour la communication entre un fils et son père ou plusieurs fils, ou (possible mais plus rare), entre les descendants d'une même famille.
Dans tous ces cas le tuyau est établi par un ancêtre commun.

Tuyaux nommés

Tuyaux avec un nom dans le système de fichier. Créés par

```
mkfifo: string -> file_perm -> unit
```

Ouverts comme des fichiers, se comportent comme des tuyaux : *bloquent lorsqu'ils sont vides ou pleins, ne supportent pas l'opération «seek». Partagent leur entrée dans la table des fichiers.*

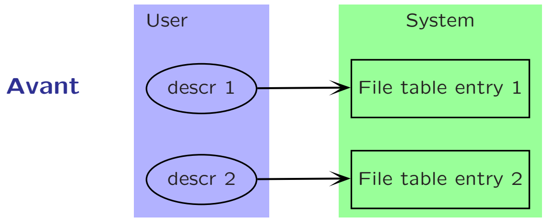
Usage : permettent la communication entre des « cousins ».

Redirection

L'opération `dup2` permet de duppliquer un descripteur de fichier en le redirigeant.

```
dup2 : file_descr -> file_descr -> unit
```

`dup2 descr1 descr2` se comporte comme « `descr2 := !descr1` ».

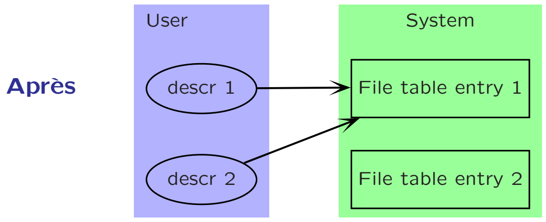


Redirection

L'opération `dup2` permet de duppliquer un descripteur de fichier en le redirigeant.

```
dup2 : file_descr -> file_descr -> unit
```

`dup2 descr1 descr2` se comporte comme « `descr2 := !descr1` ».



Propriétés

Les structures systèmes sont partagées.

La position de lecture est donc partagée entre tous les descripteurs qui pointent vers la même structure.

Les descripteurs résident dans le programme :

Un descripteur est

- un pointeur vers la structure système
- plus des drapeaux (*e.g.* `close_on_exec`), qui sont remis à zéro par `dup2`.

Le descripteur $\mathtt{descr}_1$ peut être fermé sans affecter $\mathtt{descr}_2$. Si $\mathtt{descr}_1$ et $\mathtt{descr}_2$ sont tous les deux fermés, la structure système peut être désallouée.

Création d'un descripteur

Par ouverture d'un fichier, tuyau, etc.

Les descripteurs prédéfinis `stdin`, `stdout`, `stderr`.

Attention, un descripteur, y compris `stdin`, `stdout` et `stderr` peuvent avoir été fermés !

Par duplication

```
dup : file_descr -> file_descr
```

Le descripteur retourné est un nouveau descripteur, mais partage la même entrée dans la table des fichiers.

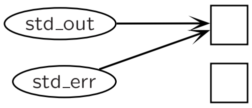
(Celui-ci peut être égal à un descripteur standard qui avait été auparavant fermé.)

Exemple

Échange de `stdout` et `stderr` :

Mauvais

```
dup2 stdout stderr ;
dup2 stderr stdout
```



Sans effet !

Exemple

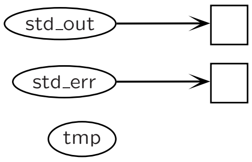
Échange de `stdout` et `stderr` :

Mauvais

```
dup2 stdout stderr ;
dup2 stderr stdout
```

Bon

```
let tmp = dup stderr in
dup2 stdout stderr ;
dup2 tmp stdout ;
close tmp ;
```



Exemple

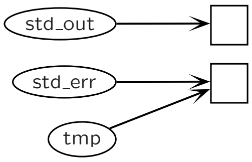
Échange de `stdout` et `stderr` :

Mauvais

```
dup2 stdout stderr ;
dup2 stderr stdout
```

Bon

```
let tmp = dup stderr in
dup2 stdout stderr ;
dup2 tmp stdout ;
close tmp ;
```



Exemple

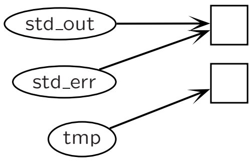
Échange de `stdout` et `stderr` :

Mauvais

```
dup2 stdout stderr ;
dup2 stderr stdout
```

Bon

```
let tmp = dup stderr in
dup2 stdout stderr ;
dup2 tmp stdout ;
close tmp ;
```



Exemple

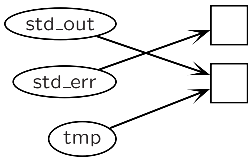
Échange de `stdout` et `stderr` :

Mauvais

```
dup2 stdout stderr ;
dup2 stderr stdout
```

Bon

```
let tmp = dup stderr in
dup2 stdout stderr ;
dup2 tmp stdout ;
close tmp ;
```



Exemple

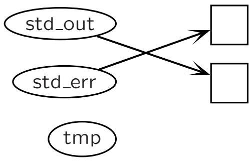
Échange de `stdout` et `stderr` :

Mauvais

```
dup2 stdout stderr ;
dup2 stderr stdout
```

Bon

```
let tmp = dup stderr in
dup2 stdout stderr ;
dup2 tmp stdout ;
close tmp ;
```



Redirection des descripteurs standards

La redirection dans les commandes shell

`cmd <file`	`cmd` lit dans `file` au lieu de `stdin`.
`cmd 1>file`	`cmd` écrit dans `file` au lieu de `stdout`.
`cmd 2>file`	`cmd` envoie sa sortie d'erreur dans `file`.
`cmd 2>&1`	`cmd` redirige sa sortie d'erreur dans `stdout`.

Redirection avant exec

```
let open_out_file cmd args out_file =
  let fd =
    openfile out_file [O_WRONLY; O_TRUNC; _O_CREAT] 0o666 in
  dup2 fd stdout;
  close fd;
  execvp cmd args;;
```

Récupérer la sortie d'une commande


```
let command_to_string com args =
  let (fd_in, fd_out) = pipe() in
  match fork() with
  | 0 ->
      close fd_in;
      dup2 fd_out stdout;
      close fd_out;
      execvp com args
  | k ->
      close fd_out;
      let chan = in_channel_of_descr fd_in in
      let after() = close_in chan  ...  in
      try_finalize input_line chan after() ;;
```

Exercices

Création de processus avec redirection

Des fonctions dérivées permettent de lancer un programme en redirigeant ses entrées/sorties standards.

Disponibles dans la librairie `Unix`

Voir `create_process` et `create_process_env`.

Ces fonctions retournent au père le numéro processus fils que le père **doit impérativement libérer** (avec `waitpid`)

Implémentation à l'aide de `fork`, `dup2` et `execvp`.

Commandes avec IO temporiéses

Similaires, mais on ouvre ensuite des canaux pour les entrées/sorties temporisées :

Avec entrées/sorties temporisés

```
open_process_in : string -> Pervasives.in_channel
open_process_out : string -> Pervasives.out_channel
```

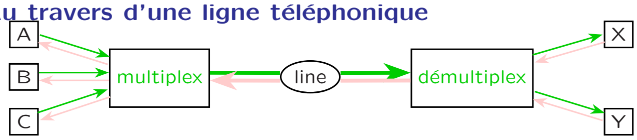
Il faut libérer les (fils des) processus alloués par :

```
close_process_in : int -> process_status
close_process_out : int -> process_status
```

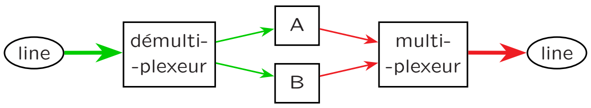
Pourquoi par directement avec `waitpid` ?
`close_process_xxx` combine `waitpid` et `close_xxx`

Multiplexage

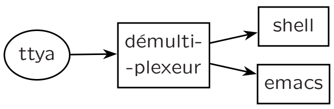
Au travers d'une ligne téléphonique



Plus retour symmétrique : problème équivalent à



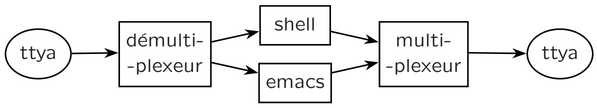
Multiplexage



Dé-multiplexage ne pose pas de problème.

- on lit le destinataire et la longeur de la transmission.
- on lit l'information et on la transmet au destinataire.

Multiplexage



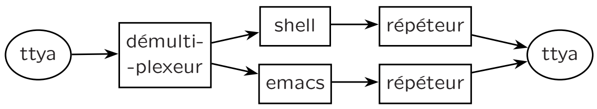
Dé-multiplexage ne pose pas de problème.

Multiplexage (Solution 1)

On redirige les sorties vers un multiplexeur.

- Lecture sur chacune des entrées
  et recopie l'information annotée sur sa sortie.
- Problème : les lectures sont bloquantes.
  - Risque de bloquer lorsque le tuyau est vide et le processus inactif.
    C'est une « situation de famine.»

Multiplexage



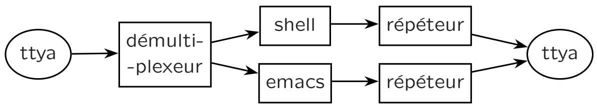
Dé-multiplexage ne pose pas de problème.

Multiplexage (Solution 2)

On met un répéteur par entrée qui

- lit son entrée et recopie l'information annotée sur la sortie.
- Problème : les écritures ne sont pas atomiques.
  - les blocs taggés risquent d'être entrelacés.
    C'est un problème « d'exclusion mutuelle » (race condition)

Multiplexage



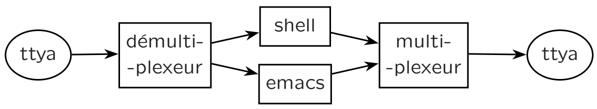
Dé-multiplexage ne pose pas de problème.

Multiplexage (Solution 3)

Répéteur + synchronisation.

- On met un verrou pendant l'écriture.
- Problème : cher + attente active
  - Il faut regarder régulièrement s'il y a eu des changements.
  - Réglage statique impossible : trop ou pas assez souvent.

Multiplexage



Dé-multiplexage ne pose pas de problème.

Multiplexage (Solution 4) : il faut une nouvelle primitive

`select`

- On se met attente sur toutes les entrées.
- Le système nous réveille et décrit les entrées qu'on peut lire sans bloquer.

Select


```
select :
  file_descr list -> file_descr list -> file_descr list ->
  float ->
  file_descr list * file_descr list * file_descr list
```

`select` ℓ_1 ℓ_2 ℓ_3 t retourne un triplet $\ell'_1, \ell'_2, \ell'_3$ de descripteurs qui sont «prêts» et parmi ℓ_1, ℓ_2, ℓ_3 :

- ℓ_1 sont des descripteurs en attente de lecture
- ℓ_2 sont des descripteurs en attente d'écriture
- ℓ_3 sont des descripteurs en attente de conditions exceptionnelles (*e.g.* contrôle de flux). Peu utilisé.
- $t > 0$ est un délai maximum d'attente.
  $t = 0$ signifie attendre indéfiniment.
  $t < 0$ signifie retourner immédiatement.

Exemple typique


```
match select [fd1;fd2] [] [] 0.5 with
  [],[],[] -> (* le delai de 0,5s est ecoule *)
| fdl,[],[] ->
    if List.mem fd1 fdl then
          (* lire depuis fd1 sans risque de bloquer *);
    if List.mem fd2 fdl then
          (* lire depuis fd2 sans risque de bloquer *)
```

Attention !

Si un descripteur d, partagé par plusieurs processus, est prêt en lecture, et est retourné par `select` dans le processus p, il se peut qu'un processus q lise les données de d avant p et que p bloque en essayant de lire d.

Le multiplex


```
let multiplex line_in line_out inputs outputs =
  let input_fds = in_line :: Array.to_list inputs in
  try while true do
    let (ready_fds, _, _) = select input_fds [] [] (-1.) in
    for i = 0 to Array.length inputs - 1 do
      if List.mem inputs.(i) ready_fds then begin
        let n = read inputs.(i) buffer 2 255 in
        buffer.[0] <- char_of_int i; (* destinataire *)
        buffer.[1] <- char_of_int n; (* longueur *)
        ignore (write line_out buffer 0 (n+2))
      end
    done;
    if List.mem line_in ready_fds then begin
      ⋮
    end
  done with End_of_file -> ();;
```

Le multiplexeur


```
let multiplex line_in line_out inputs outputs =
  let input_fds = line_in :: Array.to_list inputs in
  try while true do
    let (ready_fds, _, _) = select input_fds [] [] (-1.) in
    for i = 0 to Array.length inputs - 1 do
      ⋮
    done;
    if List.mem line_in ready_fds then begin
      really_read line_in buffer 0 2;
      let i = int_of_char(buffer.[0]) in
      match int_of_char(buffer.[1]) with
      | 0 -> close outputs.(i)
      | n -> really_read line_in buffer 0 n in
             ignore (write outputs.(i) buffer 0 n)
    end
  done with End_of_file -> ();;
```

Un timer facile et portable

Écrire `usleep` avec `select`

```
let usleep t (* microseconds *) =
  ignore (select [] [] [] (float t /. 1e6));;
```

La commande `usleep` peut être interrompue par un signal. Comment s'assurer que l'on attend au moins le temps demandé ? On utilise `gettimeofday` pour calculer le temps écoulé.

```
let usleep t (* microseconds *) =
  let s = float t /. 1e6 in let time = gettimeofday() +. s in
  let rec sleep s (* seconds *)=
    if s > 0. then
      try ignore (select [] [] [] s)
      with Unix_error (EINTR, _, _) ->
        sleep (time -. gettimeofday()) in
  sleep s
```

Appels système et signaux

Appels interruptibles

Les appels systèmes qui peuvent bloquer un temps arbitrairement long :

`waitsig`, `read` ou `write` dans un tuyau, `select`, ...

Lorsqu'un signal non ignoré est reçu pendant l'appel, l'appel échoue avec l'erreur `EINTR`. Il faut, en général relancer l'appel.

Appels non interruptibles

Les appels immédiats (pas besoin d'attendre) :

`getpid`, `umask`, `sigprocmask`, ...

Les appels dont l'attente est courte (donc bornée) :

`read`/`write` dans un fichier : si le bloc n'est pas dans le cache, on doit changer de contexte le temps de charger le bloc, mais cela prend un temps court borné.

Retenir

Communication entre processus

- par des tuyaux.
- communication établie par un processus ancêtre commun branchement par redirections
- lecture écriture comme dans un fichier, mais
  - bloquantes quand le tuyau est vide/plein
  - read/write dans des tuyaux = appel système interruptible
  - donc se protéger !

Accès à des ressources concurrentes

- par l'appel système `select`
- attente passsive (avec timer !)
- appel système interruptible (se protéger...)

Applications

- Redirections, Récupérer les résultats de comandes, Multiplexage, etc.