Systèmes d'exploitation

Communication inter-processus

Thomas Ropars

thomas.ropars@univ-grenoble-alpes.fr

http://tropars.github.io/

2023

Références

Beaucoup des slides de ce cours ont été produites par d'autres personnes:

- V. Marangozova
- R. Lachaize

Contenu de ce cours

- Gestion des entrées-sorties
  - Redirections
- Les mécanismes de communication inter-processus
  - Les tubes
  - Les signaux

Agenda

Gestion des entrées sorties

Communication inter-processus

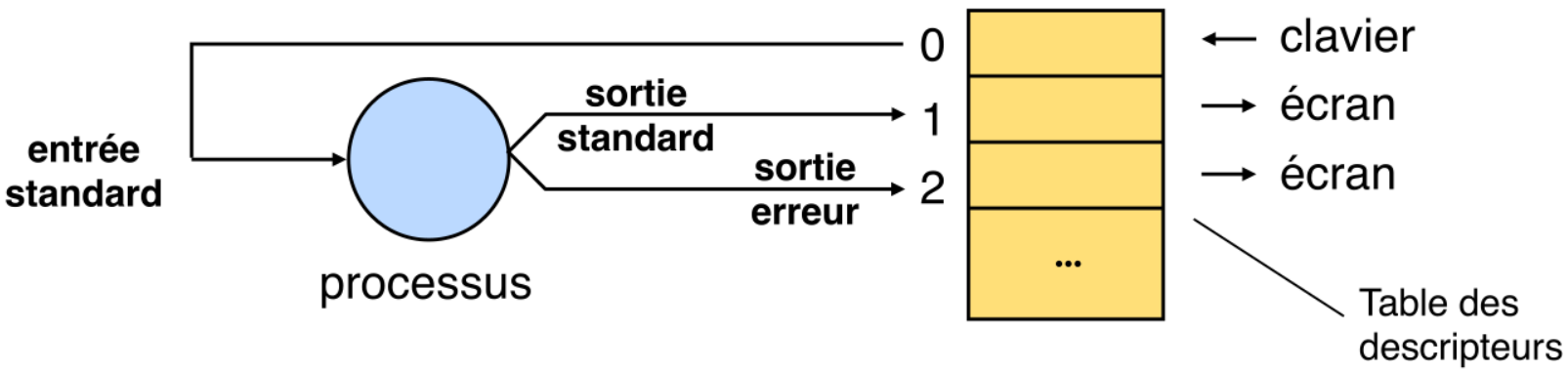
Les tubes

Les signaux

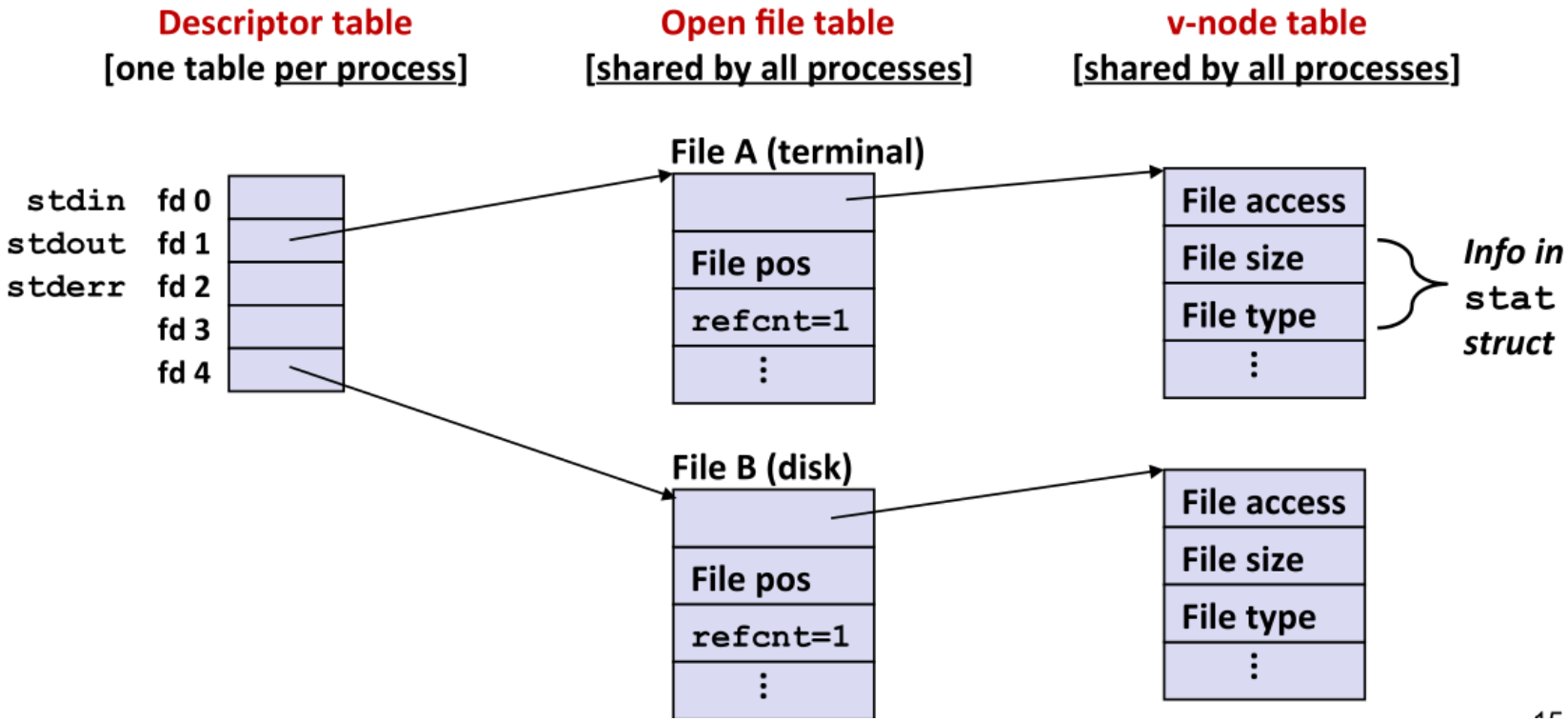
Rappel du cours précédent

Descripteur de fichiers

- Chaque fichier ouvert est représenté par un descripteur de fichier
- Tout processus a 3 descripteurs de fichiers ouverts par défaut:
  - 0: L'entrée standard
  - 1: La sortie standard
  - 2: La sortie d'erreur



Représentation des fichiers ouverts



- Un tableau de descripteur de fichiers par processus
- Des structures de données partagées entre tous les processus:

Commentaires sur la figure précédente

Des structures de données partagées entre tous les processus:

Open file table

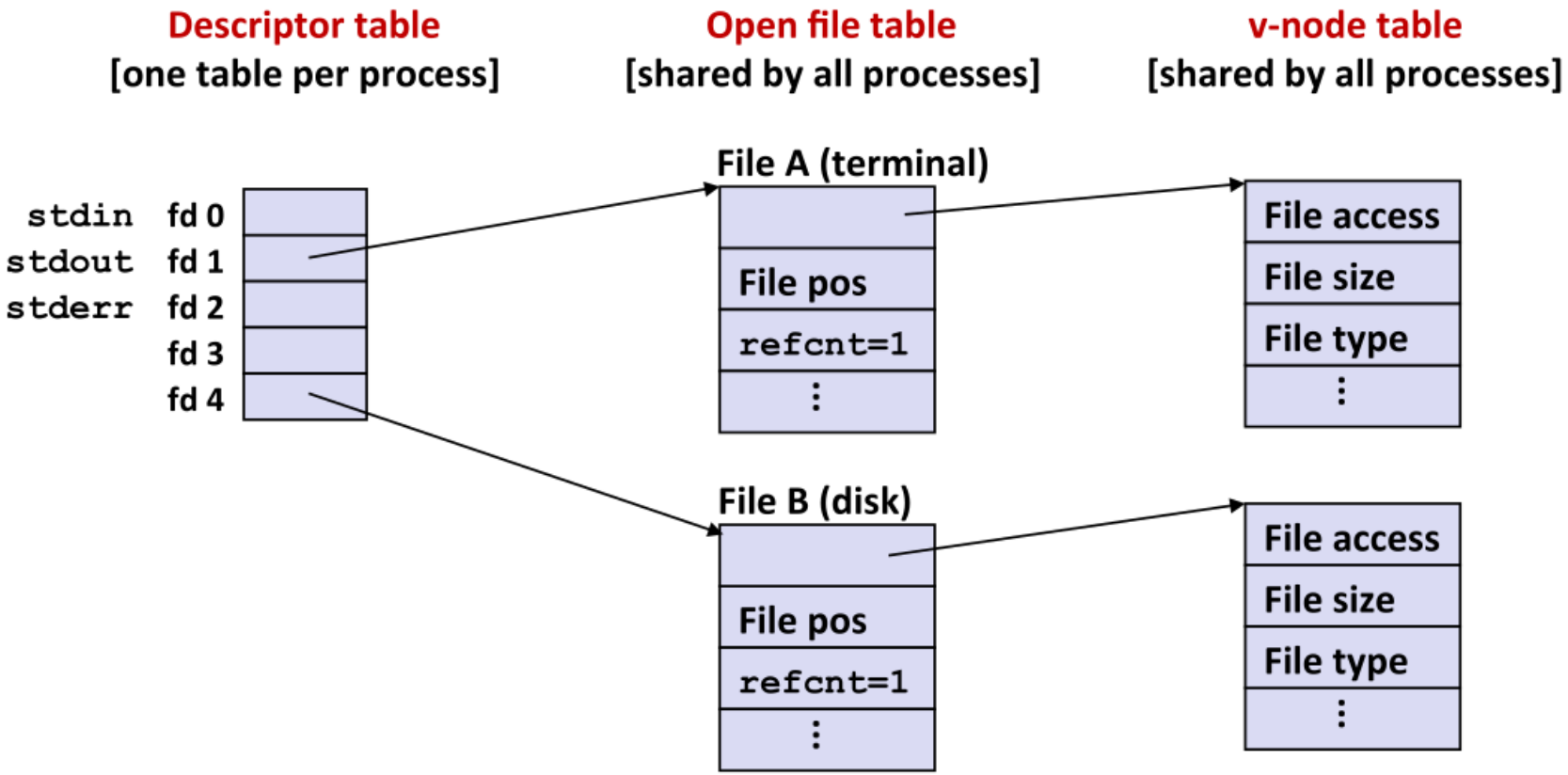
- Chaque entrée représente une *connexion* à un fichier
  - La position dans le fichier est associé à la connexion
- Plusieurs processus peuvent partager la même connexion
  - Exemple: Héritage d'un descripteur de fichier avec `fork()`
- Un fichier peut être ouvert plusieurs fois
  - Plusieurs connexions vers le même fichier

Vnode table

- Chaque entrée représente un fichier ouvert
- Contient des informations sur le fichier:
  - Droits d'accès associés
  - Taille
  - Position sur le support physique
  - Etc.

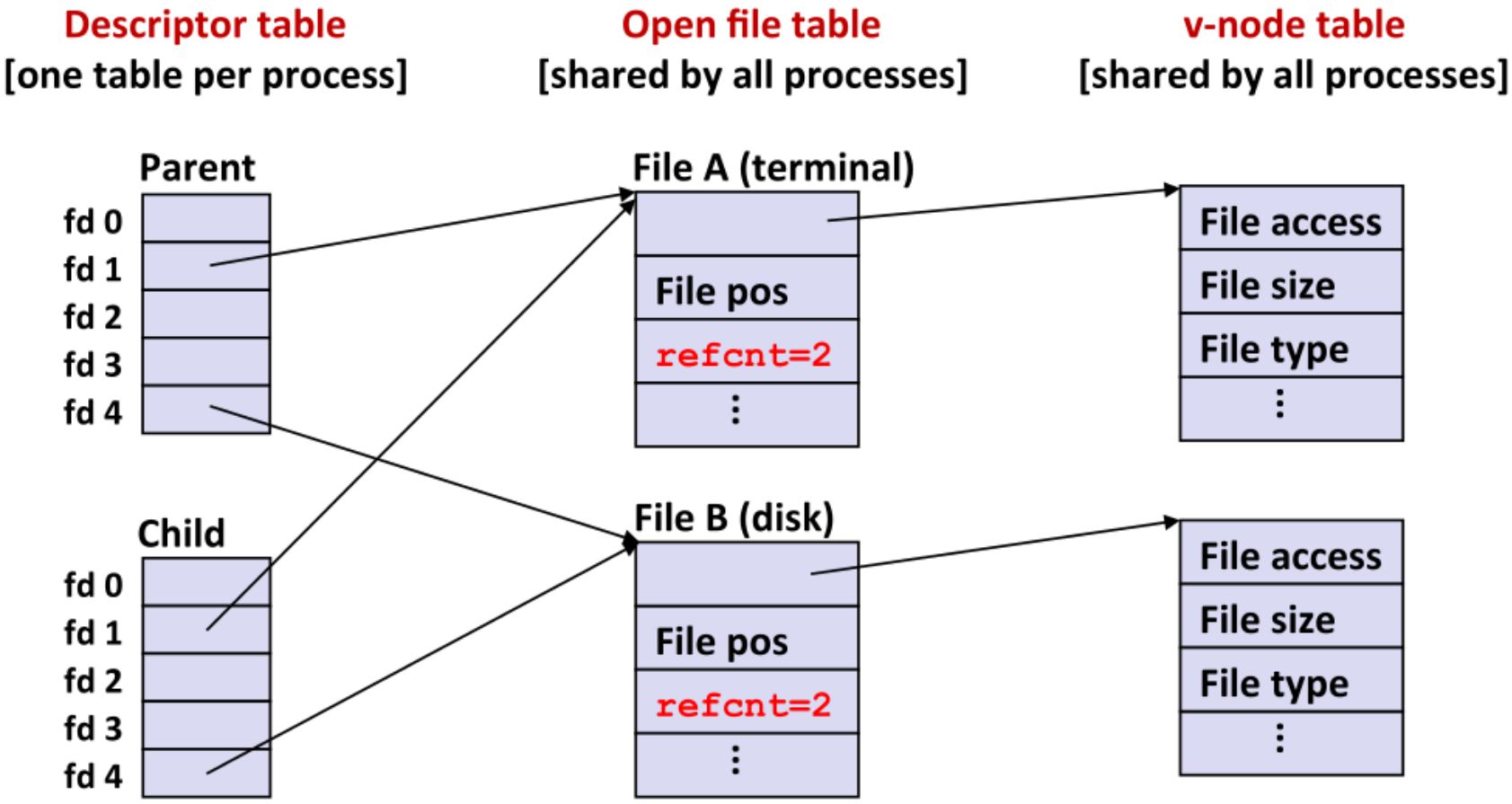
Ce qu'il se passe lors d'un appel à fork()

Avant



Ce qu'il se passe lors d'un appel à fork()

Après



- Le processus fils hérite des descripteurs de fichier ouvert
- Il partage les mêmes connexions

Redirections d'entrées-sorties

Comment est ce qu'un shell met en oeuvre la redirection de fichiers?

- `ls > file.txt`

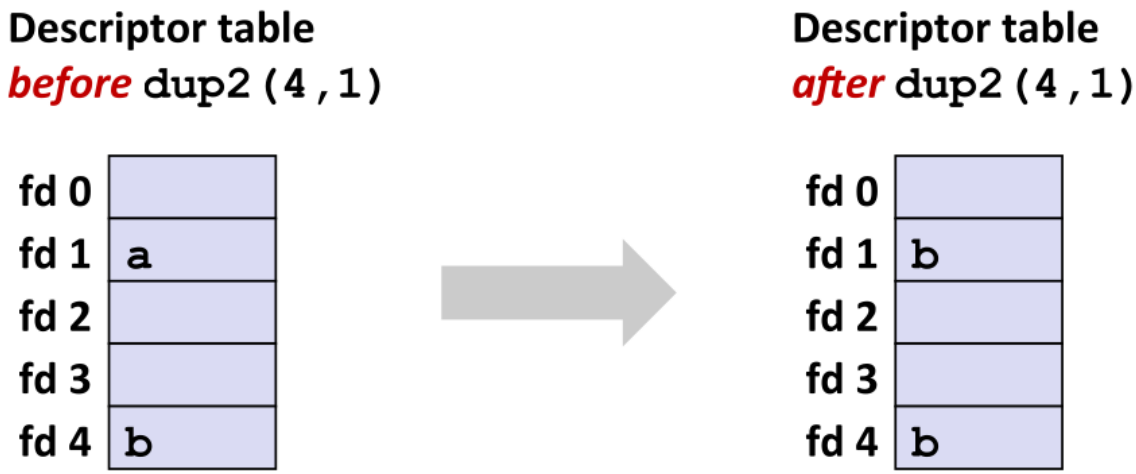
Duplication de descripteurs de fichiers

- dup(fd): Copie le descripteur de numéro `fd` dans le premier descripteur disponible
  - Copie dans la première entrée disponible dans le tableau des descripteurs de fichiers
- dup2(fd1, fd2): Copie le descripteur `fd1` dans le descripteur `fd2`.
  - Copie l'entrée correspondant à `fd1` dans l'entrée correspondant à `fd2`

Dup2()

Appel de dup2(4,1)

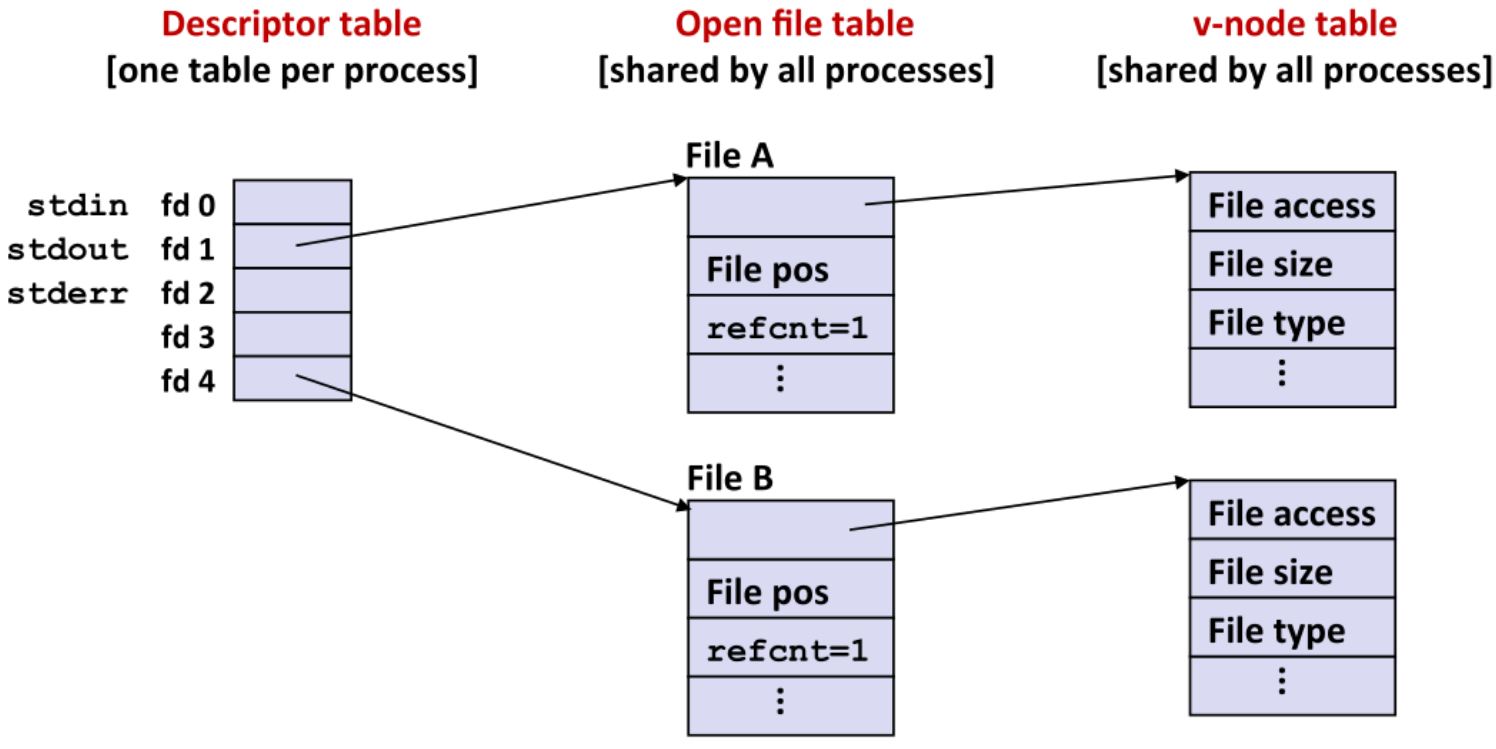
- L'écriture sur la sortie standard est maintenant redirigée vers le fichier correspondant au descripteur de fichiers 4



Étapes pour mettre en oeuvre une redirection

`ls > file.txt`

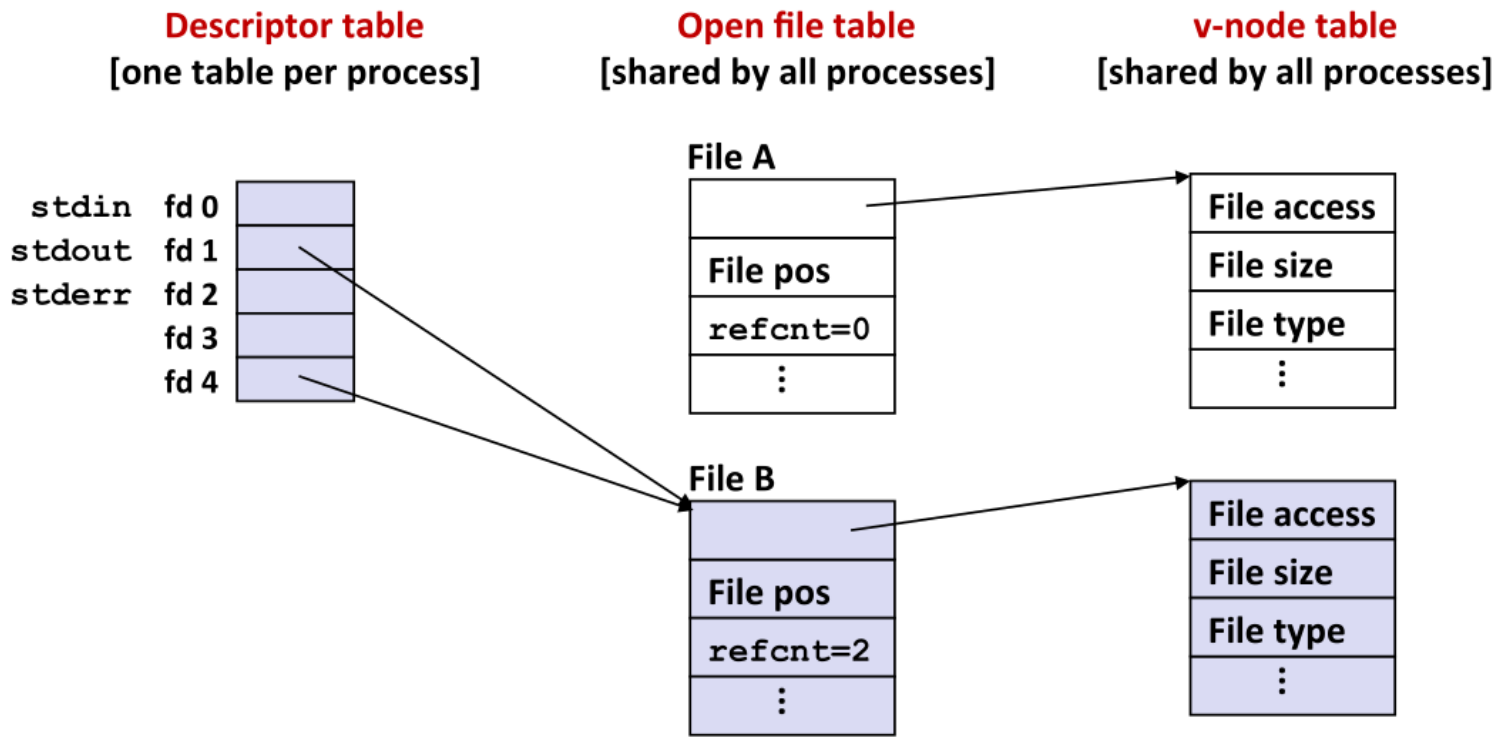
- 1) Appel à `fork()` pour créer un nouveau processus
- 2) Dans le fils, appel à `open()` pour ouvrir le fichier vers lequel rediriger la sortie standard



Étapes pour mettre en oeuvre une redirection

`ls > file.txt`

- 3) Appel à `dup2()` pour mettre en place la redirection
- 4) Appel à `exec()` pour charger le programme `ls`



Agenda

Gestion des entrées sorties

Communication inter-processus

Les tubes

Les signaux

Communication inter-processus

Inter-Process Communication (IPC)

- Un des aspects les plus importants (et aussi les plus délicats) de la programmation de systèmes
- Permet à des processus de communiquer entre eux

Mécanismes vus dans ce cours

- Les tubes
- Les signaux

Communication inter-processus

Mécanismes non couverts dans ce cours

- Les Files de messages
  - Concept de boite au lettre pour échanger des messages (pas de notion de messages avec les tubes)
- Les Segments mémoire partagée et les Sémaphores
  - Permettent d'échanger des informations en écrivant et en lisant dans une zone de mémoire partagée
  - Les sémaphores permettent la coordination des processus
- Les Sockets
  - Permettent la communication sur le réseau

Agenda

Gestion des entrées sorties

Communication inter-processus

Les tubes

Les signaux

Introduction

```
cat *.c | grep var
```

a) crée un tube et deux processus : p1 qui exécute `cat *.c`, p2 qui exécute `grep var`

b) connecte la sortie de p1 à l'entrée du tube et l'entrée de p2 à la sortie du tube

Les principes

- Un tube est un tampon qui sert de moyen de communication entre un processus producteur et un processus consommateur
  - Le tampon a une capacité finie
  - Il est unidirectionel (sous Linux)
- Les données sont transmises dans l'ordre FIFO (First In First Out)
  - Le consommateur lit les données dans l'ordre dans lequel elles ont été produites

Manipuler les flots d'entrée-sortie (primitives)

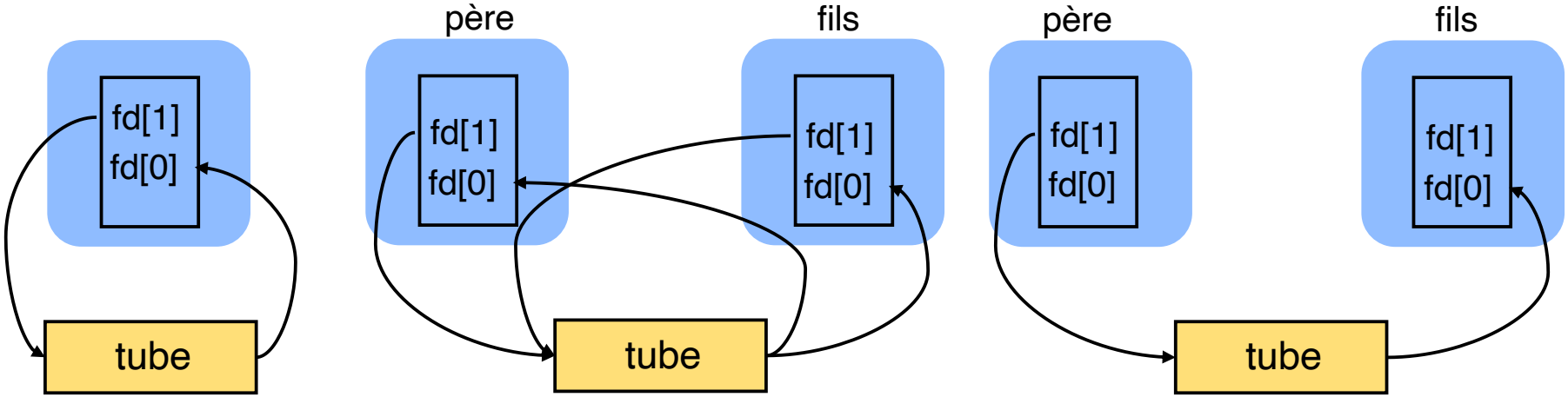
Les tubes et les flots peuvent être manipulés au niveau des appels système.

Créer un tube : la primitive `pipe()` crée un tube, dont l'entrée et la sortie sont associées à des descripteurs choisis par le système

```
int fd[2]; pipe(fd);
```

Si la primitive réussit, elle crée un tube, renvoie 0, et met à jour le tableau fd :
fd[0] = desc. sortie du tube, fd[1] = desc. entrée du tube.
Si elle échoue, elle renvoie –1

Par exemple, un père peut communiquer avec un fils à travers un tube.



après `pipe(fd)` après `fork()` (descripteurs copiés) après fermeture descripteurs inutiles

Utilisation

Communication entre un processus et son fils

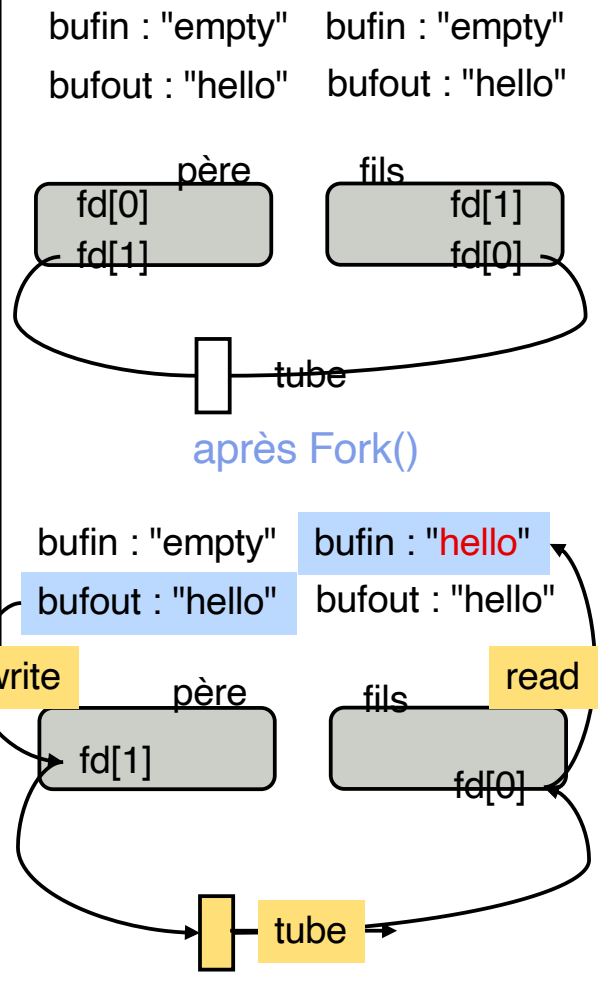
Étapes principales

1. Appel à `pipe(fd)` pour créer un tube
   - fd[1] correspond à l'entrée du tube (accès en écriture)
   - fd[0] correspond à la sortie du tube (accès en lecture)
2. Création du processus fils avec `fork()`
   - Le nouveau processus a une copie identique de la mémoire et des descripteurs de fichiers
   - Il a donc aussi accès au tube
3. Fermer les descripteurs de fichier inutiles
   - Typiquement, un seul processus accède en lecture et un seul accède un écriture
   - Sinon, risque de blocage (voir slides suivants)
4. Communiquer en utilisant `read()` et `write()`

Programmation d'un tube père -> fils

```c
#include "csapp.h"
#define BUFSIZE 10
int main(void) {
  char bufin[BUFSIZE] = "empty";
  char bufout[BUFSIZE] = "hello";
  int bytesin, bytesout;   pid_t childpid;
  int fd[2];

  pipe(fd);
  bytesin = strlen(bufin);
  childpid = Fork();
  if (childpid  != 0) {           /* père */
    Close(fd[0]); bytesout = write(fd[1], bufout, strlen(bufout)+1);
    printf("[%d]: wrote %d bytes\n", getpid(), bytesout);
  } else {                        /* fils */
    Close(fd[1]); bytesin = read(fd[0], bufin, BUFSIZE);
    printf("[%d]: read %d bytes, my bufin is {%s} \n »,
           getpid(), bytesin, bufin);
    }
    exit(0);
}
```




```
<unix>./parentwritepipe
[29196]:wrote 6 bytes
[29197]: read 6 bytes, my bufin is {hello}
<unix>
```

Un producteur-consommateur

man 7 pipe

Sémantique de l'opération d'écriture (write())

- Ecrit le nombre d'octets demandé dans le tube
- Bloque si le tampon est plein
  - L'opération `write()` ne retourne pas
  - Jusqu'à ce que des données soient lues sur le tube
    - Ce qui libère de la place pour écrire de nouvelles données
- Si tous les descripteurs de fichiers faisant référence au tube en lecture ont été fermés, le signal `SIGPIPE` est envoyé au processus essayant d'écrire.
  - Si le signal est ignoré `write()` échoue avec l'erreur `EPIPE`.
  - **Important de fermer dès que possible les descripteurs de fichier dont on n'a pas besoin**
    - Risque de blocage sinon

Un producteur-consommateur

man 7 pipe

Sémantique de l'opération de lecture (read())

- Lit le nombre d'octets demandé depuis le tube
- Bloque si le tampon est vide (ou si moins d'octets que demandé sont disponibles)
  - L'opération `read()` ne retourne pas
  - Jusqu'à ce que des données soient disponibles sur le tube
- Les données sont transmises comme un flux d'octets
  - Pas de concept de séparation entre les messages
- Si tous les descripteurs de fichiers faisant référence au tube en écriture ont été fermés, `read()` va voir EOF (*End-Of-File*) et retourner 0.
  - **Important de fermer dès que possible les descripteurs de fichier dont on n'a pas besoin**
    - Risque de blocage sinon

Tubes nommés (FIFOs)

- Un tube ne peut être utilisé qu'entre un processus et ses descendants (ou entre descendants d'un même processus).
  - Ses extrémités ne sont désignées que par des descripteurs, qui ne peuvent se transmettre qu'entre père et fils.
- Pour faire communiquer deux processus quelconques, on utilise des tubes nommés, ou FIFOs.

Tubes nommés (FIFOs)

- Un tube ne peut être utilisé qu'entre un processus et ses descendants (ou entre descendants d'un même processus).
  - Ses extrémités ne sont désignées que par des descripteurs, qui ne peuvent se transmettre qu'entre père et fils.
- Pour faire communiquer deux processus quelconques, on utilise des tubes nommés, ou FIFOs.

Les tubes nommés

- Tubes possédant un nom symbolique dans le système de fichiers
- Primitive: `int mkfifo(char *nom, mode_t mode)`
  - `mode` définit les droits d'accès au tube nommé.
- Pour pouvoir être utilisé, un FIFO doit avoir été préalablement ouvert par deux processus, l'un en mode écriture, l'autre en mode lecture.
  - Chacun des processus reste bloqué tant que l'autre n'a pas ouvert le FIFO.

Agenda

Gestion des entrées sorties

Communication inter-processus

Les tubes

Les signaux

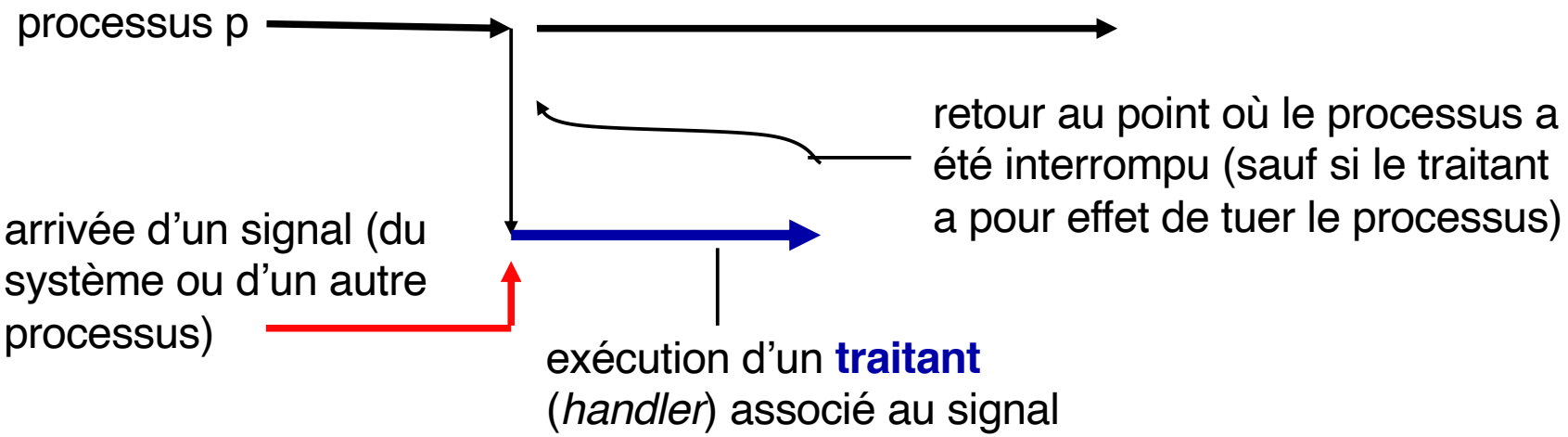
Signaux

- **Un signal est un événement asynchrone destiné à un (ou plusieurs) processus. Un signal peut être émis par un processus ou par le système d'exploitation.**

- **Un signal est analogue à une interruption : un processus destinataire réagit à un signal en exécutant un programme de traitement, ou traitant (*handler*). La différence est qu'une interruption s'adresse à un processeur alors qu'un signal s'adresse à un processus. Certains signaux traduisent d'ailleurs la réception d'une interruption (voir plus loin).**

- **Les signaux sont un mécanisme de bas niveau. Ils doivent être manipulés avec précaution car leur usage recèle des pièges (en particulier le risque de perte de signaux). Ils sont néanmoins utiles lorsqu'on doit contrôler l'exécution d'un ensemble de processus (exemple : le *shell*) ou que l'on traite des événements liés au temps.**

Fonctionnement des signaux



Points à noter (seront précisés plus loin) :

- Il existe différents signaux, chacun étant identifié par un **nom symbolique** (ce nom représente un entier)
- Chaque signal est associé à un **traitant par défaut**
- Un signal peut être **ignoré** (le traitant est vide)
- Le traitant d'un signal peut être changé (sauf pour 2 signaux particuliers)
- Un signal peut être **bloqué** (il n'aura d'effet que lorsqu'il sera débloqué)
- Les signaux **ne sont pas mémorisés** (détails plus loin)

Quelques exemples de signaux

Nom symbolique	Événement associé	Défaut
SIGINT	Frappe du caractère <control-C>	terminaison
SIGTSTP	Frappe du caractère <control-Z>	suspension
SIGKILL	Signal de terminaison	terminaison
SIGSTOP	Signal de suspension	suspension
SIGSEGV	Violation de protection mémoire	terminaison +core dump
SIGALRM	Fin de temporisation (alarm)	terminaison
SIGCHLD	Terminaison d'un fils	ignoré
SIGUSR1	Signal émis par un processus utilisateur	terminaison
SIGUSR2	Signal émis par un processus utilisateur	terminaison
SIGCONT	Continuation d'un processus stoppé	reprise

Notes. Les signaux SIGKILL et SIGSTOP ne peuvent pas être bloqués ou ignorés, et leur traitant ne peut pas être changé (pour des raisons de sécurité).
Utiliser toujours les noms symboliques, non les valeurs (exemple : SIGINT = 2, etc.) car ces valeurs peuvent changer d'un Unix à un autre. Inclure <signal.h>. Voir man 7 signal

États d'un signal

Un signal est **envoyé** à un processus destinataire et **reçu** par ce processus
Tant qu'il n'a pas été pris en compte par le destinataire, le signal est **pendant**
Lorsqu'il est pris en compte (exécution du traitant), le signal est dit **traité**

Qu'est-ce qui empêche que le signal soit immédiatement traité dès qu'il est reçu ?

- Le signal peut être bloqué, ou masqué (c'est à dire retardé) par le destinataire. Il est délivré dès qu'il est débloqué
- En particulier, un signal est bloqué pendant l'exécution du traitant d'un signal du même type ; il reste bloqué tant que ce traitant n'est pas terminé

Point important : il ne peut exister qu'un seul signal pendant d'un type donné (il n'y a qu'un bit par signal pour indiquer les signaux de ce type qui sont pendants). S'il arrive un autre signal du même type, il est perdu.

Envoi d'un signal

Un processus peut envoyer un signal à un autre processus. Pour cela, il utilise la primitive `kill` (appelée ainsi pour des raisons historiques ; un signal ne tue pas forcément son destinataire).

Utilisation : `kill(pid_t pid, int sig)`

Effet : Soit `p` le numéro du processus émetteur du signal
Le signal de numéro `sig` est envoyé au(x) processus désigné(s) par `pid` :

- si `pid > 0` le signal est envoyé au processus de numéro `pid`
- si `pid = 0` le signal est envoyé à tous les processus du même groupe que `p`
- si `pid = -1` le signal est envoyé à tous les processus
- si `pid < 0` le signal est envoyé à tous les processus du groupe `-pid`

Restrictions : un processus (sauf s'il a les droits de `root`) n'est autorisé à envoyer un signal qu'aux processus ayant le même `uid` (identité d'utilisateur) que lui.

Le processus de numéro 1 ne peut pas recevoir certains signaux (notamment SIGKILL). Étant donné son rôle particulier, il est protégé pour assurer la sécurité et la stabilité du système.

Redéfinir le traitant associé à un signal

2 appels systèmes possibles

```
#include <signal.h>
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
int sigaction(int signum, const struct sigaction *act,
        struct sigaction *oldact);
```

- `sigaction()` est plus complet et plus générique que `signal()`
  - Mais plus complexe à utiliser
- Nous nous concentrerons sur `signal()`
  - Ses paramètres:
    - Un pointeur de fonction définissant le traitant
    - Le numéro de signal concerné

Exemple 1 : traitement d'une interruption du clavier

```
void handler(int sig) { /* nouveau traitant */
  printf("signal SIGINT recu !\n");
  exit(0);
}

int main() {
  signal(SIGINT, handler); /* installe le traitant */
  pause () ; /* attend un signal*/
  exit(0);
}
```

- Redéfinition du traitant par défaut pour le signal `SIGINT`
- Résultat:

```
<unix>./test-int
        => frappe de control-C
signal SIGINT reçu !
<unix>
```

Exemple 2 : utilisation de la temporisation

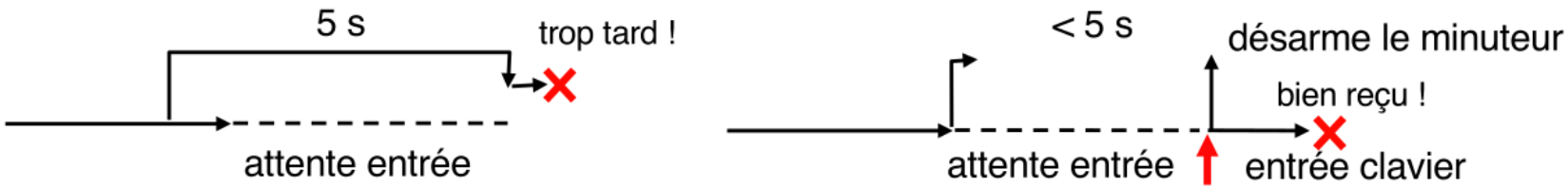
- `unsigned int alarm(unsigned int nbSec)`
  - Provoque l'envoi du signal `SIGALRM` après environ `nbSec` secondes
  - Annulation avec `nbSec=0`

```
void handler(int sig) { /* nouveau traitant */
  printf("trop tard !\n");
  exit(0);
}

int main() {
  int reponse;
  signal(SIGALRM, handler); /* installe le traitant */
  printf("Entrez un nombre avant 5s :");
  alarm(5);
  scanf("%d", &reponse);
  alarm(0); printf("bien recu !");
  exit (0),
}
```

Exemple 2 : utilisation de la temporisation

Illustration



Exemple 3: synchronisation père-fils

- Lorsqu'un processus se termine ou est suspendu, le système envoie automatiquement un signal `SIGCHLD` à son père.
  - Le traitement par défaut consiste à ignorer ce signal.
  - On peut prévoir un traitement spécifique en associant un nouveau traitant à `SIGCHLD`

Application

- Cas d'un processus qui crée un grand nombre de fils
  - Ex: Un shell (qui crée un processus pour traiter chaque commande)
- Ce processus doit prendre en compte leur fin dès que possible pour éviter une accumulation de processus zombis
  - Solution: Attente de processus dans le traitant de `SIGCHLD`

Exemple 3: synchronisation père-fils

```
void handler(int sig) { /* nouveau traitant */
  pid_t pid;
  int statut;
  pid = waitpid(−1, &statut, 0); /* attend un fils quelconque */
  return;
}

int main() {
  Signal(SIGCHLD, handler); /* installe le traitant */
  ... <creation d'un certain nombre de fils, sur demande> ...
  exit (0),
}
```

Gestion des masques de signaux

- Le masque de signaux d'un processus est automatiquement modifié lors de l'exécution d'un traitant
- Le masque définit les signaux qui sont bloqués (masqués) pendant l'exécution du traitant
  - Au moins le signal en cours de traitement (c'est le seul par défaut)
  - Les signaux ignorés dépendent des consignes spécifiées au moment de la définition des traitants

- On ne modifie pas explicitement le masque de signaux dans le code d'un traitant.
  - En revanche, on peut modifier (et consulter) le masque de signaux d'un processus dans le code extérieur aux traitants avec la primitive `sigprocmask()`

Gestion des masques de signaux

```
int main() {
  sigset_t s; // structure de donnees opaque permettant de
              // designer un ou plusieurs types de signaux
  ...
  Sigemptyset(&s); //s <- ensemble vide
  Sigaddset(&s, SIGINT); // ajout de SIGINT dans l'ensemble s
  Sigprocmask(SIG_BLOCK, &s, NULL);
  // a ce stade, le masquage de SIGINT est effectif
  // SIG_BLOCK: ajout a l'ensemble des signaux masques
  ...
  Sigprocmask(SIG_UNBLOCK, &s, NULL);
  // a ce stade, SIGINT est a nouveau démasqué
  ...
}
```

Terminaux, sessions et groupes sous Unix

- Pour bien comprendre le fonctionnement de certains signaux, il faut avoir une idée des notions de session et de groupes
- En quelques mots:
  - Une session est associée à un terminal
    - Donc au login d'un usager du système au moyen d'un shell.
    - Le processus qui exécute ce shell est le leader de la session.
  - Dans une session, on peut avoir plusieurs groupes de processus correspondant à divers travaux en cours.
    - Il existe au plus un `groupe interactif` (foreground, ou premier plan) avec lequel l'usager interagit via le terminal.
    - Il peut aussi exister plusieurs `groupes d'arrière-plan`, qui s'exécutent en travail de fond (par exemple processus lancés avec &, appelés jobs).
    - Seuls les processus du groupe interactif peuvent lire au terminal
    - les signaux SIGINT (frappe de control-C) et SIGTSTP (frappe de control-Z) s'adressent au groupe interactif

Terminaux, sessions et groupes sous Unix (2)

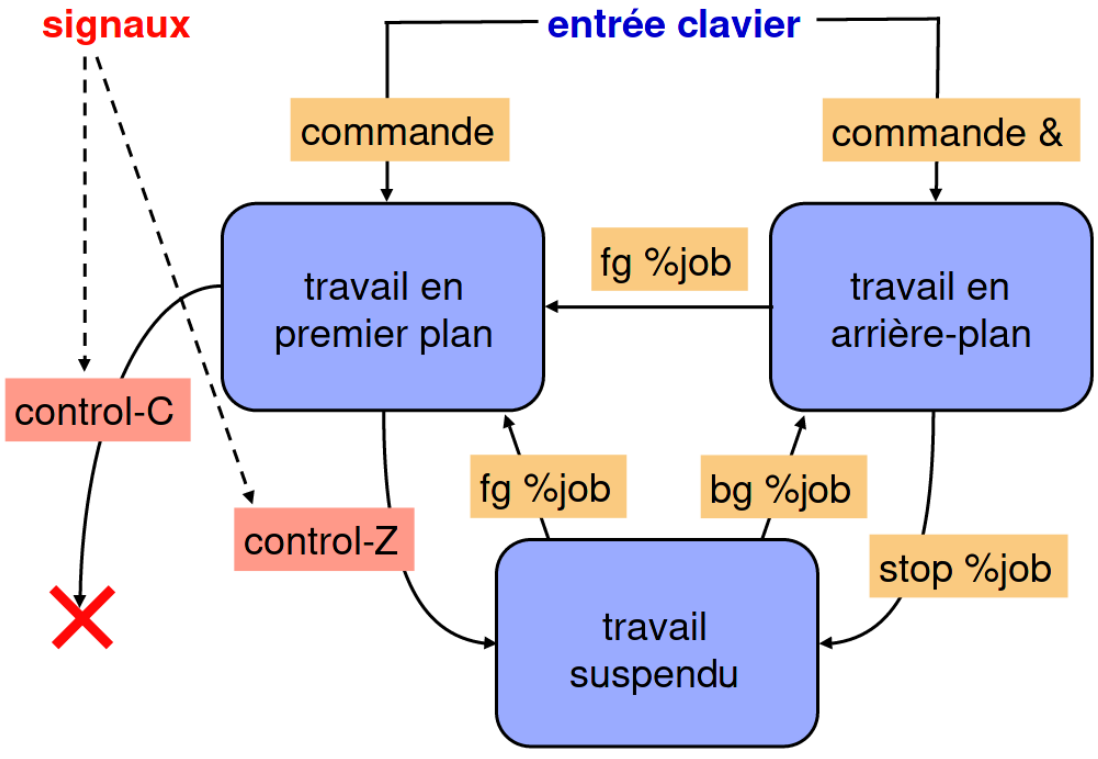
loop.c

```
int main() {
   printf"processus %d, groupe %d\n", getpid(), getpgrp());
   while(1) ;
}
```

```
<unix> loop & loop & ps
processus 10468, groupe 10468
[1]  10468
processus 10469, groupe 10469
[2] 10469
  PID TTY          TIME  CMD
 5691 pts/0    00:00:00  tcsh
10468 pts/0    00:00:00  loop
10469 pts/0    00:00:00  loop
10470 pts/0    00:00:00  ps
<unix>fg %1
loop
        =>frappe de control-Z
Suspended
<unix>jobs
[1] + Suspended          loop
[2] - Running            loop
```

```
<unix>bg %1
[1]  loop &
<unix>fg %2
loop
        => frappe de control-C
<unix>ps
  PID TTY          TIME  CMD
 5691 pts/0    00:00:00  tcsh
10468 pts/0    00:02:53  loop
10474 pts/0    00:00:00  ps
<unix>  => frappe de control-C
<unix>ps
  PID TTY          TIME  CMD
 5691 pts/0    00:00:00  tcsh
10468 pts/0    00:02:57  loop
10474 pts/0    00:00:00  ps
<unix>
```

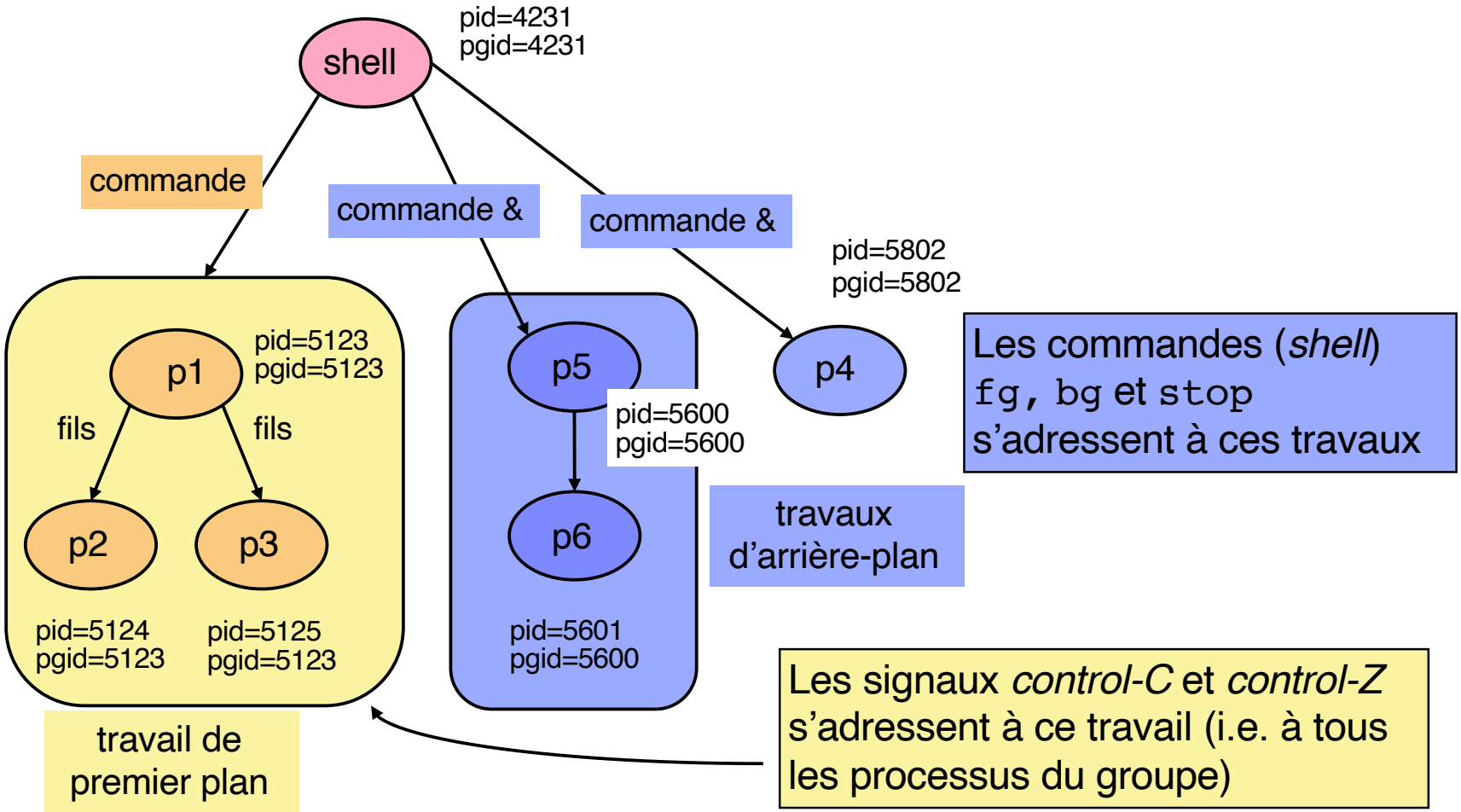
États d'un travail



Travail (*job*) = processus (ou groupe de processus) lancé par une commande au *shell*

Seuls le travail en premier plan peut recevoir des signaux du clavier. Les autres sont manipulés par des commandes

Vie des travaux



Des références en plus

Les documents suivant peuvent compléter le contenu des slides

- Les tubes (Dominique Revuz): `http://www-igm.univ-mlv.fr/~dr/NCSPDF/chapitre10.pdf`
- Les signaux (UC Louvain): `https://sites.uclouvain.be/SystInfo/notes/Theorie/html/Fichiers/fichiers-signaux.html`