Communications avec les processus

Redirections

Redirections

- La redirection d'un fichier standard d'entrée/sortie peut être effectuée de deux manières :
  - par ouverture d'un fichier et son association à un descripteur de haut niveau à l'aide de la fonction de bibliothèque `freopen()`
  - par modification du périphérique désigné par le descripteur de bas niveau en utilisant une des primitives `dup()` ou `dup2()`

`freopen()`

```
FILE *freopen(const char *pathname, const char *mode, FILE *stream);
```

- La fonction `freopen()` ouvre le fichier de nom *pathname* et l'associe au descripteur *stream*
- Le flux original est fermé s'il était valide
- *mode* est identique à celui de `fopen` :
  - `"r"`, `"r+"` en lecture
  - `"w"`, `"w+"` en écriture
  - `"a"`, `"a+"` en mode ajout

Exemple `freopen()`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    FILE *redir;
    fprintf(stdout,"La sortie standard\n");
    fprintf(stderr,"La sortie d'erreur standard\n");
    redir = freopen("/tmp/stdout.txt","w",stdout);
    if (redir == NULL) {
      perror("Echec de l'ouverture de /tmp/stdout.txt");
      return EXIT_FAILURE;
    }
    fprintf(stdout,"La sortie standard après redirection\n");
    fprintf(stderr, "La sortie d'erreur standard après redirection\n");
    return EXIT_SUCCESS;
}
```

Exemple `freopen()` : exécution

xterm

```
$ ls -l /tmp/stdout.txt
ls: impossible d'accéder à '/tmp/stdout.txt': Aucun fichier ou dossier de ce type
$ ./redirection_stdout_freopen
La sortie standard
La sortie d'erreur standard
La sortie d'erreur standard après redirection
$ ls -l /tmp/stdout.txt
-rw-r----- 1 giersch and 38 13 oct.  17:29 /tmp/stdout.txt
$ cat /tmp/stdout.txt
La sortie standard après redirection
$
```

Utilisation de la primitive `dup()/dup2()`

- C'est la méthode universelle permettant d'effectuer une redirection par duplication d'un descripteur de bas niveau valide dans un descripteur non valide
- Elle est utilisable même si le descripteur de bas niveau que l'on veut dupliquer existe déjà et ne peut être ouvert :
  - tube
  - tube nommé
  - socket
  - ...

Utilisation de la primitive `dup()`/`dup2()`

- Elle comporte 4 étapes dont deux (1 et 4) sont facultatives :

Utilisation de la primitive `dup()`/`dup2()`

- Elle comporte 4 étapes dont deux (1 et 4) sont facultatives :
  1. Création d'un descripteur en utilisant `open()`

Utilisation de la primitive `dup()`/`dup2()`

- Elle comporte 4 étapes dont deux (1 et 4) sont facultatives :
  1. Création d'un descripteur en utilisant `open()`
  2. Fermeture du descripteur à rediriger avec `close()`

Utilisation de la primitive `dup()`/`dup2()`

- Elle comporte 4 étapes dont deux (1 et 4) sont facultatives :
  1. Création d'un descripteur en utilisant `open()`
  2. Fermeture du descripteur à rediriger avec `close()`
  3. Duplication du descripteur valide à l'aide de `dup()`

Utilisation de la primitive `dup()`/`dup2()`

- Elle comporte 4 étapes dont deux (1 et 4) sont facultatives :
  1. Création d'un descripteur en utilisant `open()`
  2. Fermeture du descripteur à rediriger avec `close()`
  3. Duplication du descripteur valide à l'aide de `dup()`
  4. Fermeture du descripteur ayant été utilisé pour la redirection

Utilisation de la primitive `dup()`/`dup2()`

- Elle comporte 4 étapes dont deux (1 et 4) sont facultatives :
  1. Création d'un descripteur en utilisant `open()`
  2. Fermeture du descripteur à rediriger avec `close()`
  3. Duplication du descripteur valide à l'aide de `dup()`
  4. Fermeture du descripteur ayant été utilisé pour la redirection

- L'étape 1 est inutile si le descripteur existe déjà (tube, socket, . . .)

Utilisation de la primitive `dup()`/`dup2()`

- Elle comporte 4 étapes dont deux (1 et 4) sont facultatives :
  1. Création d'un descripteur en utilisant `open()`
  2. Fermeture du descripteur à rediriger avec `close()`
  3. Duplication du descripteur valide à l'aide de `dup()`
  4. Fermeture du descripteur ayant été utilisé pour la redirection
- L'étape 1 est inutile si le descripteur existe déjà (tube, socket, . . .)
- L'étape 4 est facultative dans tous les cas

Les primitives de duplication `dup()` et `dup2()`

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- `dup()` : duplique le descripteur passé en paramètre dans le plus petit descripteur disponible

Les primitives de duplication `dup()` et `dup2()`

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- `dup()` : duplique le descripteur passé en paramètre dans le plus petit descripteur disponible
- `dup2()` : ferme *newfd* si besoin puis duplique *oldfd* dans *newfd*, *oldfd* doit être valide. Si *newfd* = *oldfd*, rien n'est fait

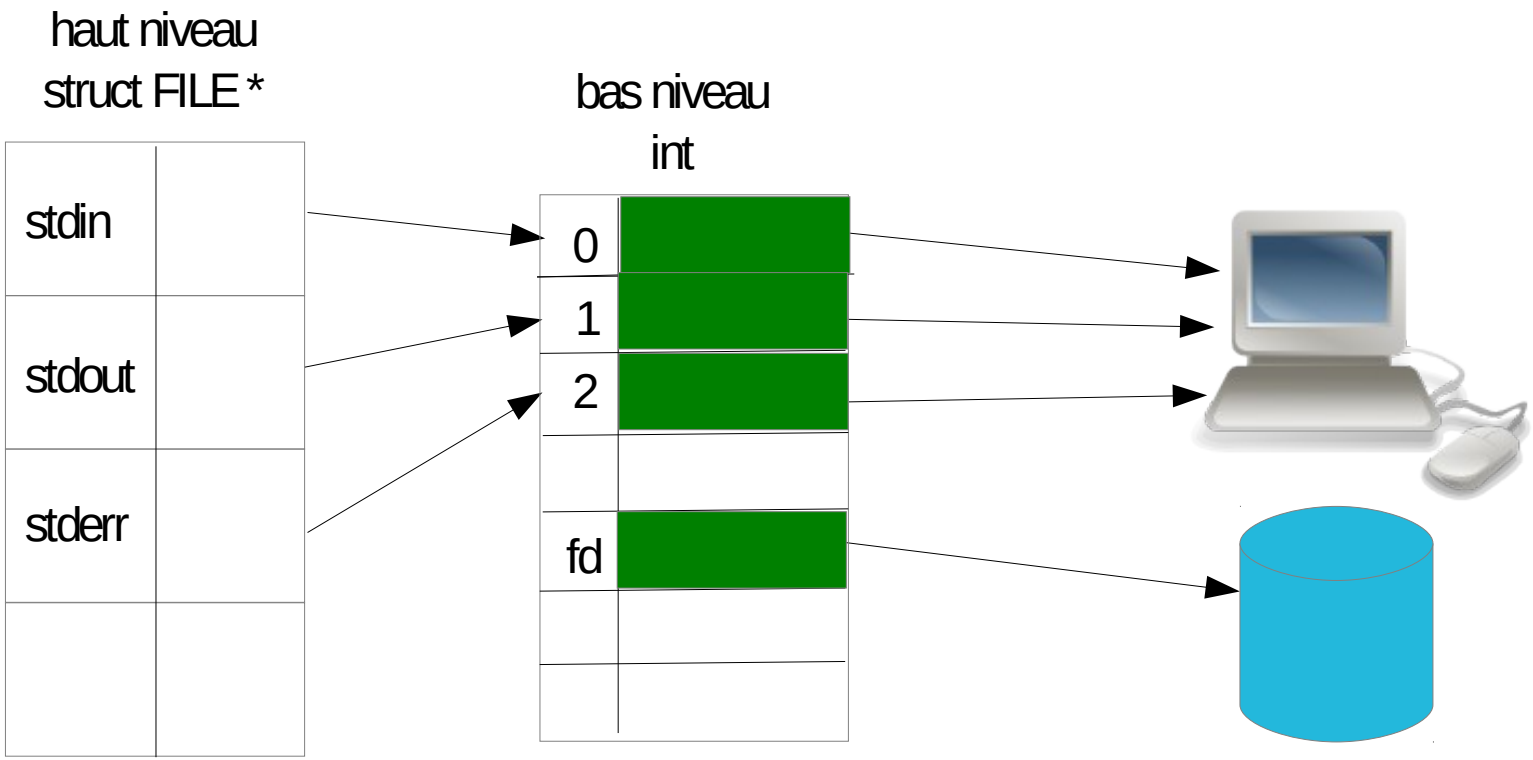
Les primitives de duplication `dup()` et `dup2()`

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- `dup()` : duplique le descripteur passé en paramètre dans le plus petit descripteur disponible
- `dup2()` : ferme *newfd* si besoin puis duplique *oldfd* dans *newfd*, *oldfd* doit être valide. Si *newfd* = *oldfd*, rien n'est fait
- Valeur retournée : -1 en cas d'erreur, *newfd* sinon

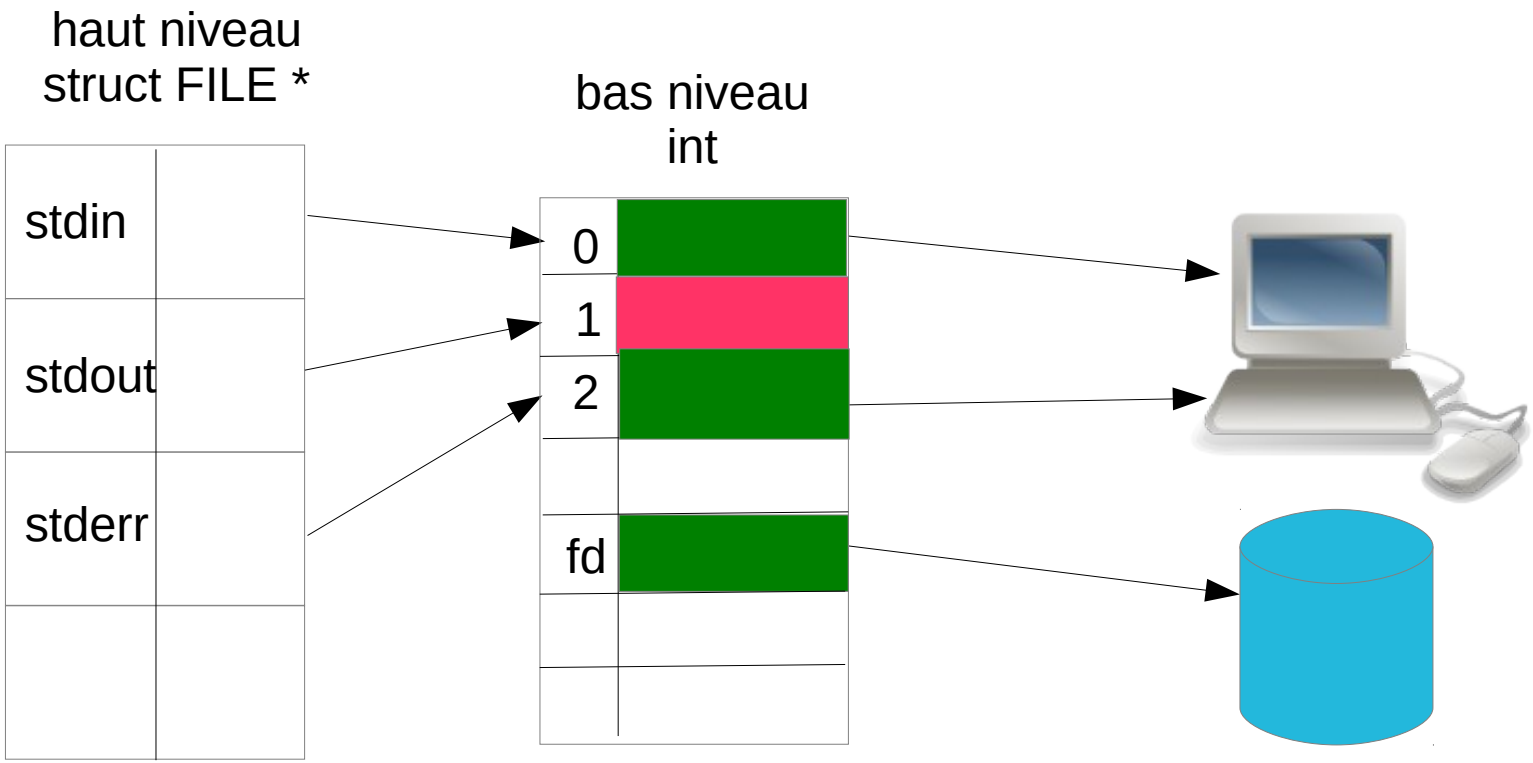
Création du descripteur

fd = open(....)



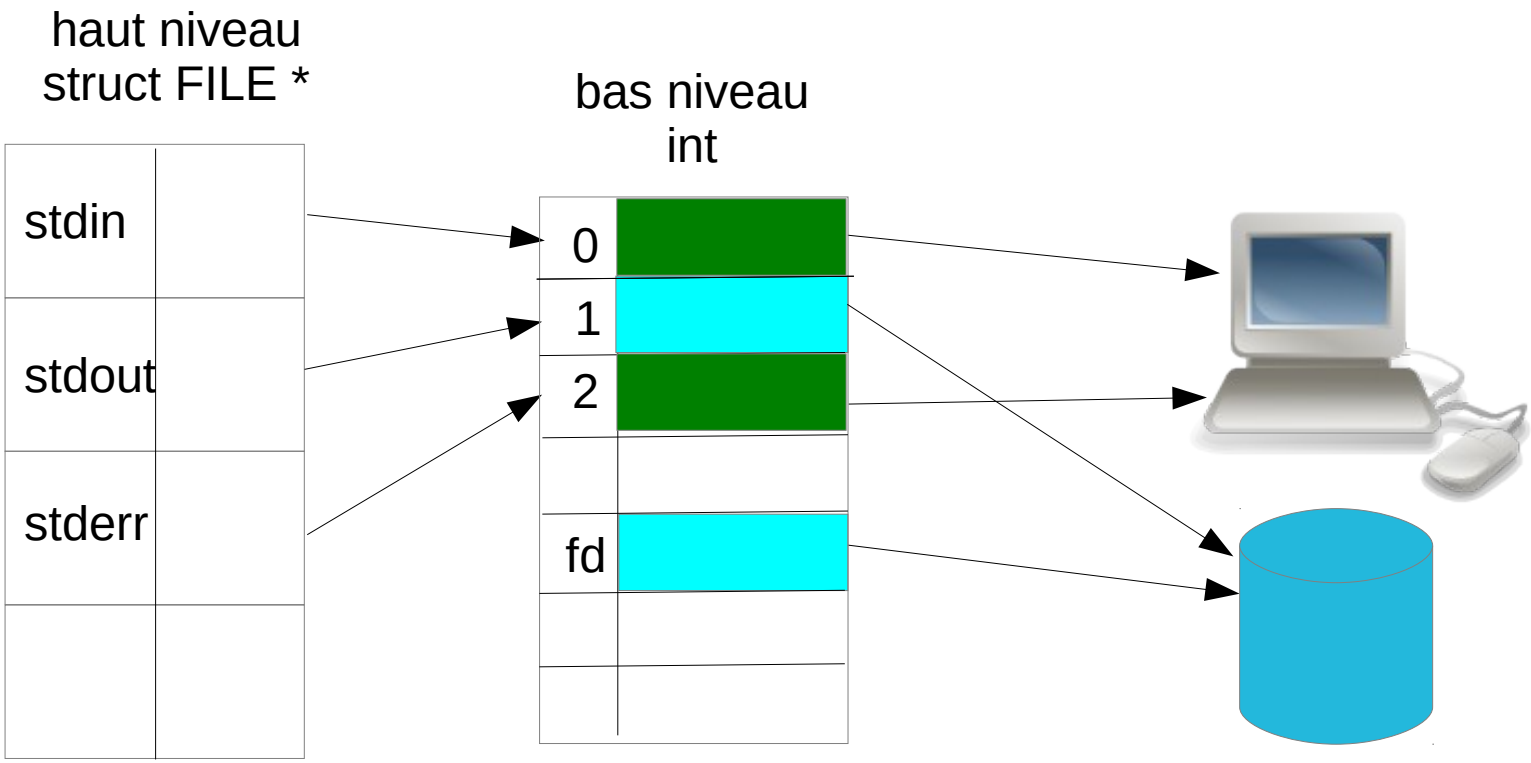
Fermeture descripteur sortie standard

close(1)



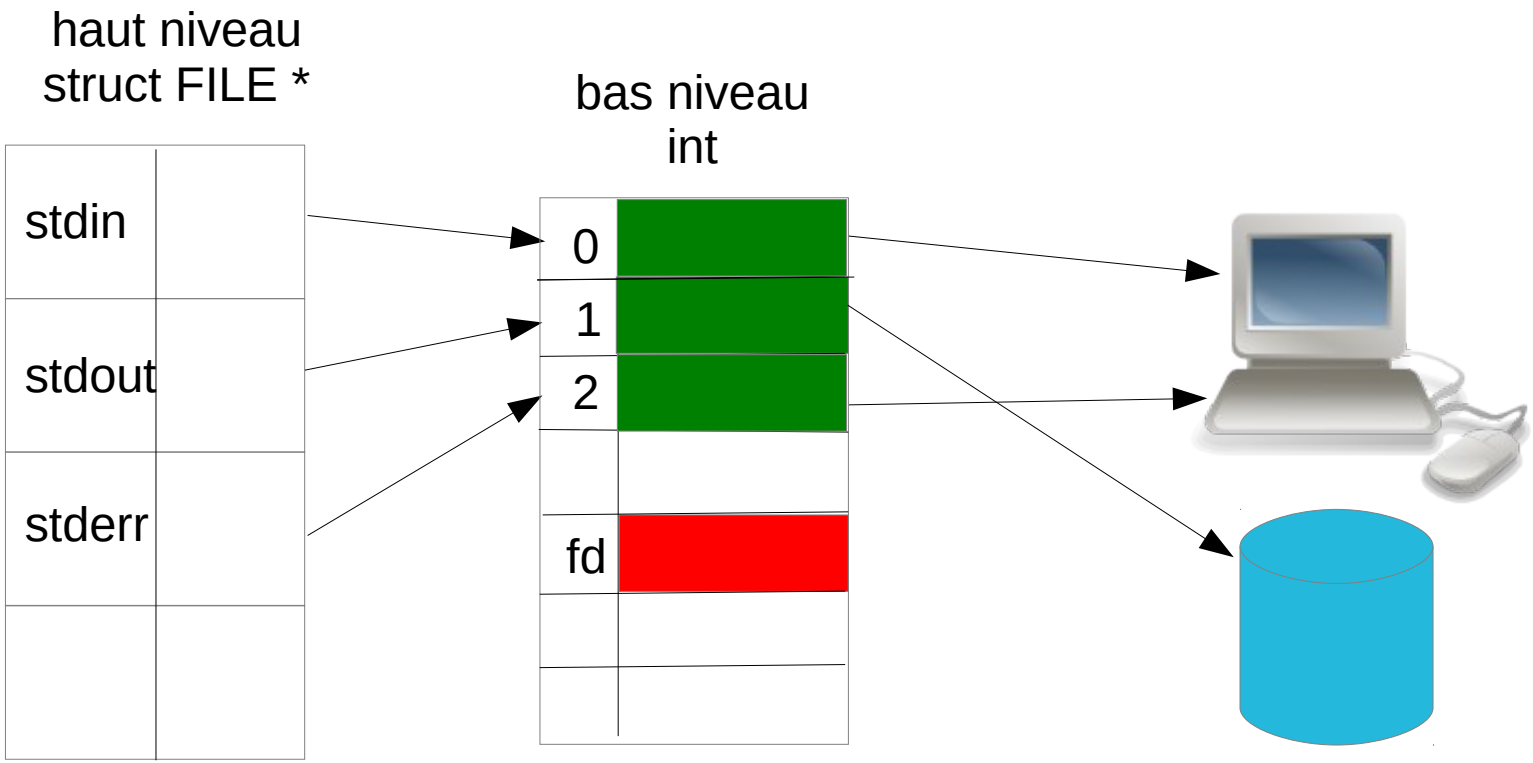
Après duplication du descripteur fd

dup(fd)



Fermeture du descripteur fd

close(fd)



Exemple : `dup()`

```
#include <unistd.h>
int main(void)
{
  fprintf(stdout,"La sortie standard\n");
  int sortie = creat("/tmp/stdout.txt", S_IRUSR | S_IWUSR);
  if (sortie < 0) {
      perror("Echec de l'ouverture de /tmp/stdout.txt");
      return EXIT_FAILURE;
  }
  close(STDOUT_FILENO); // ou bien 1
  if (dup(sortie) < 0) {
    perror("dup");
    return EXIT_FAILURE;
  }
  close(sortie);
  fprintf(stderr,"La sortie standard d'erreur après redirection\n");
  fprintf(stdout,"La sortie standard après redirection\n");
  return EXIT_SUCCESS;
}
```

Exemple : `dup2()`

```
#include <unistd.h>
int main(void)
{
  fprintf(stdout,"La sortie standard\n");
  int sortie = creat("/tmp/stdout.txt", S_IRUSR | S_IWUSR);
  if (sortie < 0) {
      perror("Echec de l'ouverture de /tmp/stdout.txt");
      return EXIT_FAILURE;
  }
  // close(STDOUT_FILENO); => inutile avec dup2
  if (dup2(sortie, STDOUT_FILENO) < 0) { // ou bien 1
    perror("dup2");
    return EXIT_FAILURE;
  }
  close(sortie);
  fprintf(stderr,"La sortie standard d'erreur après redirection\n");
  fprintf(stdout,"La sortie standard après redirection\n");
  return EXIT_SUCCESS;
}
```

Redirections des fichiers standard avant recouvrement

- Les descripteurs des fichiers ouverts font partie de l'héritage durant le `fork`
- Durant un `exec` tous ces descripteurs, sauf ceux ouverts avec l'option `O_CLOEXEC` restent valides il est donc possible de rediriger les fichiers standard juste avant l'appel à `exec`
- La même procédure peut évidemment être utilisée après `fork` pour exécuter un processus fils après avoir redirigé un ou plusieurs fichiers standards

Exemple : redirections avant recouvrement (1/2)

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    int entree = open("/etc/motd", O_RDONLY);
    if (entree < 0) {
        perror("Ouverture de /etc/motd");
        return EXIT_FAILURE;
    }
    int sortie = open("/tmp/motd", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
    if (sortie < 0) {
        perror("Ouverture de /tmp/motd");
        return EXIT_FAILURE;
    }
    /* ... */
```

Exemple : redirections avant recouvrement (2/2)

```
    /* ... */
    if (dup2(entree, STDIN_FILENO) < 0) {
        perror("Echec de la redirection de l'entrée standard");
        return EXIT_FAILURE;
    }
    if (dup2(sortie, STDOUT_FILENO) < 0){
        perror("Echec de la redirection de la sortie standard");
        return EXIT_FAILURE;
    }
    execl("/bin/cat", "cat", (char*)NULL);
    perror("Echec de la primitive execl(cat)");
    return EXIT_FAILURE;
}
```

Exemple : redirections avant recouvrement (exécution)

xterm

```
$ ls -l /{etc,tmp}/motd
ls: impossible d'accéder à '/tmp/motd': Aucun fichier ou dossier de ce type
-rw-r--r-- 1 root root 286 13 oct.  18:28 /etc/motd
$ cat /etc/motd

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
$ ./exec_cat_redirige
$ cat /tmp/motd

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
$
```

Signaux

Qu'est-ce qu'un signal ?

- Un signal est une information élémentaire reçue par un processus lui signalant l'occurrence d'un événement :
  - fin d'un fils
  - erreur arithmétique
  - demande d'arrêt
  - demande de suspension temporaire
  - reprise de l'exécution
  - ...

Envoi d'un signal

- Ce signal peut-être envoyé
  - par le système au processus qui a commis une faute

Envoi d'un signal

- Ce signal peut-être envoyé
  - par le système au processus qui a commis une faute
  - par l'utilisateur avec la commande `kill` :

> **kill** [options] PID [...]

Envoi d'un signal

- Ce signal peut-être envoyé
  - par le système au processus qui a commis une faute
  - par l'utilisateur avec la commande `kill` :

```
kill [options] PID [...]
```

  - par un autre processus exécutant la primitive :

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

Les signaux (`man 7 signal`), POSIX.1-1990

Signal	Valeur	Action	Commentaire
SIGHUP	1	Term	Déconnexion détectée sur le terminal de contrôle ou mort du processus de contrôle.
SIGINT	2	Term	Interruption depuis le clavier.
SIGQUIT	3	Core	Demande « Quitter » depuis le clavier.
SIGILL	4	Core	Instruction illégale.
SIGABRT	6	Core	Signal d'arrêt depuis abort(3).
SIGFPE	8	Core	Erreur mathématique virgule flottante.
SIGKILL	9	Term	Signal « KILL ».
SIGSEGV	11	Core	Référence mémoire invalide.
SIGPIPE	13	Term	Écriture dans un tube sans lecteur.
SIGALRM	14	Term	Temporisation alarm(2) écoulée.
SIGTERM	15	Term	Signal de fin.
SIGUSR1	30,10,16	Term	Signal utilisateur 1.
SIGUSR2	31,12,17	Term	Signal utilisateur 2.
SIGCHLD	20,17,18	Ign	Fils arrêté ou terminé.
SIGCONT	19,18,25	Cont	Continuer si arrêté.
SIGSTOP	17,19,23	Stop	Arrêt du processus.
SIGTSTP	18,20,24	Stop	Stop invoqué depuis tty.
SIGTTIN	21,21,26	Stop	Lecture sur tty en arrière-plan.
SIGTTOU	22,22,27	Stop	Écriture sur tty en arrière-plan.

Les signaux **SIGKILL** et **SIGSTOP** ne peuvent ni capturés ni ignorés.

Les signaux (`man 7 signal`), POSIX.1-2001

Signal	Valeur	Action	Commentaire
SIGBUS	10,7,10	Core	Erreur de bus (mauvais accès mémoire).
SIGPOLL		Term	Événement « pollable » (System V). Synonyme de SIGIO.
SIGPROF	27,27,29	Term	Expiration de la temporisation pour le suivi.
SIGSYS	12,-,12	Core	Mauvais argument de fonction (SVr4).
SIGTRAP	5	Core	Point d'arrêt rencontré.
SIGURG	16,23,21	Ign	Condition urgente sur socket (BSD 4.2).
SIGVTALRM	26,26,28	Term	Alarme virtuelle (BSD 4.2).
SIGXCPU	24,24,30	Core	Limite de temps CPU dépassée (BSD 4.2).
SIGXFSZ	25,25,31	Core	Taille de fichier excessive (BSD 4.2).

Les signaux (`man 7 signal`), signaux divers

Signal	Valeur	Action	Commentaire
SIGIOT	6	Core	Arrêt IOT. Un synonyme de SIGABRT.
SIGEMT	7,-,7	Term	
SIGSTKFLT	-,16,-	Term	Erreur de pile sur coprocesseur (inutilisé).
SIGIO	23,29,22	Term	E/S à nouveau possible(BSD 4.2).
SIGCLD	-,-,18	Ign	Synonyme de SIGCHLD.
SIGPWR	29,30,19	Term	Chute d'alimentation (System V).
SIGINFO	29,-,-		Synonyme de SIGPWR.
SIGLOST	-,-,-	Term	Perte de verrou de fichier.
SIGWINCH	28,28,20	Ign	Fenêtre redimensionnée (BSD 4.3, Sun).
SIGUNUSED	-,31,-	Term	Signal inutilisé (sera SIGSYS).

Les signaux (`man 7 signal`), signaux temps réel

- 32 signaux temps réel
- Définis dans POSIX.1b et POSIX.1-2001
- Numéros : SIGRTMIN=33 à SIGRTMAX=64
- Pas de signification particulière
- Deux ou trois signaux sont utilisés par la bibliothèque pthreads
- Toujours utiliser SIGRTMIN+n et pas un numéro
- Exemple : SIGRTMIN+2 = 35

Envoi d'un signal avec la commande `kill(1)`

- Syntaxe :
- `kill [options] PID [...]`
- Options :
  - `-no_signal`
  - `-s, -signal no_signal`
  - `-l, -list` : liste les signaux
  - `-L, -table` : liste les signaux mis en forme
- Par défaut le signal SIGTERM (15) est envoyé
- Attention : certains shell possèdent une commande interne kill

Envoi d'un signal avec la primitive `kill(2)`

- Synopsis :

```
int kill(pid_t pid, int sig);
```

Envoi d'un signal avec la primitive `kill(2)`

- Synopsis :

```
int kill(pid_t pid, int sig);
```

- Envoie le signal numéro *sig* à un processus ou groupe de processus.

Envoi d'un signal avec la primitive `kill(2)`

- Synopsis :

```
int kill(pid_t pid, int sig);
```

- Envoie le signal numéro *sig* à un processus ou groupe de processus.
- *pid* $>$ 0 : signal envoyé au processus *pid*

Envoi d'un signal avec la primitive `kill(2)`

- Synopsis :

```
int kill(pid_t pid, int sig);
```

- Envoie le signal numéro *sig* à un processus ou groupe de processus.
- *pid* $>$ 0 : signal envoyé au processus *pid*
- *pid* $=$ 0 : signal envoyé à tous les processus du même groupe que le processus appelant

Envoi d'un signal avec la primitive `kill(2)`

- Synopsis :

```
int kill(pid_t pid, int sig);
```

- Envoie le signal numéro *sig* à un processus ou groupe de processus.
- $pid > 0$: signal envoyé au processus *pid*
- $pid = 0$: signal envoyé à tous les processus du même groupe que le processus appelant
- $pid = -1$, signal envoyé à tous les processus sauf celui de PID 1 (init)

Envoi d'un signal avec la primitive `kill(2)`

- Synopsis :

```
int kill(pid_t pid, int sig);
```

- Envoie le signal numéro *sig* à un processus ou groupe de processus.
- $pid > 0$: signal envoyé au processus *pid*
- $pid = 0$: signal envoyé à tous les processus du même groupe que le processus appelant
- $pid = -1$, signal envoyé à tous les processus sauf celui de PID 1 (init)
- $pid < -1$, signal envoyé à tous les processus du groupe $-pid$.

Envoi d'un signal avec la primitive `kill(2)`

- Synopsis :

```
int kill(pid_t pid, int sig);
```

- Envoie le signal numéro *sig* à un processus ou groupe de processus.
- $pid > 0$: signal envoyé au processus *pid*
- $pid = 0$: signal envoyé à tous les processus du même groupe que le processus appelant
- $pid = -1$, signal envoyé à tous les processus sauf celui de PID 1 (init)
- $pid < -1$, signal envoyé à tous les processus du groupe $-pid$.
- $sig = 0$, aucun signal envoyé mais les conditions d'erreur sont vérifiées

Envoi d'un signal avec la primitive `kill(2)`

- Synopsis :

```
int kill(pid_t pid, int sig);
```

- Envoie le signal numéro *sig* à un processus ou groupe de processus.
- $pid > 0$: signal envoyé au processus *pid*
- $pid = 0$: signal envoyé à tous les processus du même groupe que le processus appelant
- $pid = -1$, signal envoyé à tous les processus sauf celui de PID 1 (init)
- $pid < -1$, signal envoyé à tous les processus du groupe $-pid$.
- $sig = 0$, aucun signal envoyé mais les conditions d'erreur sont vérifiées
- Retour : 0 en cas de succès ; -1 en cas d'échec

Comportement d'un processus à la réception d'un signal

- À la réception d'un signal le processus peut se comporter des trois manières suivantes :

Comportement d'un processus à la réception d'un signal

- À la réception d'un signal le processus peut se comporter des trois manières suivantes :
  - Ignorer le signal : rien ne se passe (**SIG_IGN**)

Comportement d'un processus à la réception d'un signal

- À la réception d'un signal le processus peut se comporter des trois manières suivantes :
  - Ignorer le signal : rien ne se passe (**SIG_IGN**)
  - Laisser le système d'exploitation appliquer l'action par défaut indiquée dans les tableaux précédents (**SIG_DFL**)

Comportement d'un processus à la réception d'un signal

- À la réception d'un signal le processus peut se comporter des trois manières suivantes :
  - Ignorer le signal : rien ne se passe (**SIG_IGN**)
  - Laisser le système d'exploitation appliquer l'action par défaut indiquée dans les tableaux précédents (**SIG_DFL**)
  - Traiter le signal en ayant auparavant associé au signal une fonction de traitement à l'aide de la primitive `signal()` ou `sigaction()`

Traitement des signaux par le processus

- Pour effectuer une action spécifique lors de la réception du signal il faut associer une fonction de traitement aux signaux que l'on désire traiter par utilisation de la primitive `signal()` ou `sigaction()`.

Traitement des signaux par le processus

- Pour effectuer une action spécifique lors de la réception du signal il faut associer une fonction de traitement aux signaux que l'on désire traiter par utilisation de la primitive `signal()` ou `sigaction()`.
- Ainsi, la réception du signal donné :

Traitement des signaux par le processus

- Pour effectuer une action spécifique lors de la réception du signal il faut associer une fonction de traitement aux signaux que l'on désire traiter par utilisation de la primitive `signal()` ou `sigaction()`.
- Ainsi, la réception du signal donné :
  - interrompra le processus quelque soit l'endroit où il en est dans son exécution ;

Traitement des signaux par le processus

- Pour effectuer une action spécifique lors de la réception du signal il faut associer une fonction de traitement aux signaux que l'on désire traiter par utilisation de la primitive `signal()` ou `sigaction()`.
- Ainsi, la réception du signal donné :
  - interrompra le processus quelque soit l'endroit où il en est dans son exécution ;
  - lancera l'exécution de la fonction associée au signal.

Traitement des signaux par le processus

- Pour effectuer une action spécifique lors de la réception du signal il faut associer une fonction de traitement aux signaux que l'on désire traiter par utilisation de la primitive `signal()` ou `sigaction()`.
- Ainsi, la réception du signal donné :
  - interrompra le processus quelque soit l'endroit où il en est dans son exécution ;
  - lancera l'exécution de la fonction associée au signal.
- La fonction exécutée lors de l'arrivée du signal reçoit en paramètre le numéro du signal et ne doit retourner aucun résultat.

Traitement des signaux par le processus

- Pour effectuer une action spécifique lors de la réception du signal il faut associer une fonction de traitement aux signaux que l'on désire traiter par utilisation de la primitive `signal()` ou `sigaction()`.
- Ainsi, la réception du signal donné :
  - interrompra le processus quelque soit l'endroit où il en est dans son exécution ;
  - lancera l'exécution de la fonction associée au signal.
- La fonction exécutée lors de l'arrivée du signal reçoit en paramètre le numéro du signal et ne doit retourner aucun résultat.
- Après traitement du signal la fonction peut :

Traitement des signaux par le processus

- Pour effectuer une action spécifique lors de la réception du signal il faut associer une fonction de traitement aux signaux que l'on désire traiter par utilisation de la primitive `signal()` ou `sigaction()`.
- Ainsi, la réception du signal donné :
  - interrompra le processus quelque soit l'endroit où il en est dans son exécution ;
  - lancera l'exécution de la fonction associée au signal.
- La fonction exécutée lors de l'arrivée du signal reçoit en paramètre le numéro du signal et ne doit retourner aucun résultat.
- Après traitement du signal la fonction peut :
  - exécuter `return` : le programme sera repris à l'endroit où il a été interrompu

Traitement des signaux par le processus

- Pour effectuer une action spécifique lors de la réception du signal il faut associer une fonction de traitement aux signaux que l'on désire traiter par utilisation de la primitive `signal()` ou `sigaction()`.
- Ainsi, la réception du signal donné :
  - interrompra le processus quelque soit l'endroit où il en est dans son exécution ;
  - lancera l'exécution de la fonction associée au signal.
- La fonction exécutée lors de l'arrivée du signal reçoit en paramètre le numéro du signal et ne doit retourner aucun résultat.
- Après traitement du signal la fonction peut :
  - exécuter `return` : le programme sera repris à l'endroit où il a été interrompu
  - appeler `exit()` : le processus est alors terminé.

Traitement des signaux par le processus

signal(2)

Historique, le plus simple utiliser, mais son usage est déconseillé [a], sauf pour SIG_DFL et SIG_IGN.

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

a. Pour un gestionnaire de signal, le mode de fonctionnement dépend de l'implémentation (version du système, options de compilation, ...).

Traitement des signaux par le processus

sigaction(2)

Définie par la norme POSIX, son usage doit être privilégié.

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

Traitement des signaux par le processus

Voir sujet de TP pour les détails. . .

Tubes

Tubes

- Un tube est une structure de communication créée et gérée par la système en mémoire

Tubes

- Un tube est une structure de communication créée et gérée par la système en mémoire
- C'est un outil de communication car il est possible d'en lire le contenu et d'y écrire des données

Tubes

- Un tube est une structure de communication créée et gérée par la système en mémoire
- C'est un outil de communication car il est possible d'en lire le contenu et d'y écrire des données
- Synchronisation :

Tubes

- Un tube est une structure de communication créée et gérée par la système en mémoire
- C'est un outil de communication car il est possible d'en lire le contenu et d'y écrire des données
- Synchronisation :
  - Le système garanti que toute opération d'une taille inférieure à PIPE_BUFF, défini dans `<limits.h>`, est réalisée de manière atomique

Tubes

- Un tube est une structure de communication créée et gérée par la système en mémoire
- C'est un outil de communication car il est possible d'en lire le contenu et d'y écrire des données
- Synchronisation :
  - Le système garanti que toute opération d'une taille inférieure à PIPE_BUFF, défini dans `<limits.h>`, est réalisée de manière atomique
  - Les données sont lues dans l'ordre où elle ont été écrites

Tubes

- Un tube est une structure de communication créée et gérée par la système en mémoire
- C'est un outil de communication car il est possible d'en lire le contenu et d'y écrire des données
- Synchronisation :
  - Le système garanti que toute opération d'une taille inférieure à PIPE_BUFF, défini dans `<limits.h>`, est réalisée de manière atomique
  - Les données sont lues dans l'ordre où elle ont été écrites
  - La lecture d'un tube vide est bloquante si il existe un écrivain au moins du tube (un processus dans lequel le descripteur d'écriture est valide)

Tubes

- Un tube est une structure de communication créée et gérée par la système en mémoire
- C'est un outil de communication car il est possible d'en lire le contenu et d'y écrire des données
- Synchronisation :
  - Le système garanti que toute opération d'une taille inférieure à PIPE_BUFF, défini dans `<limits.h>`, est réalisée de manière atomique
  - Les données sont lues dans l'ordre où elle ont été écrites
  - La lecture d'un tube vide est bloquante si il existe un écrivain au moins du tube (un processus dans lequel le descripteur d'écriture est valide)
  - L'écriture d'un tube plein est bloquante si il existe un lecteur au moins du tube (un processus dans lequel le descripteur de lecture est valide)

Tubes

- Un tube est une structure de communication créée et gérée par la système en mémoire
- C'est un outil de communication car il est possible d'en lire le contenu et d'y écrire des données
- Synchronisation :
  - Le système garanti que toute opération d'une taille inférieure à PIPE_BUFF, défini dans `<limits.h>`, est réalisée de manière atomique
  - Les données sont lues dans l'ordre où elle ont été écrites
  - La lecture d'un tube vide est bloquante si il existe un écrivain au moins du tube (un processus dans lequel le descripteur d'écriture est valide)
  - L'écriture d'un tube plein est bloquante si il existe un lecteur au moins du tube (un processus dans lequel le descripteur de lecture est valide)
- L'écriture d'un tube sans lecteur déclenche le signal SIGPIPE qui, par défaut, arrête le processus

Tube : création

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

- Crée et ouvre un tube dont les descripteurs de bas niveau sont :
  - `pipefd[0]` en lecture
  - `pipefd[1]` en écriture
- Retourne 0 en cas de succès ; -1 en cas d'échec.

Tube : utilisation

- Un tube ne peut être ouvert, la primitive pipe crée et ouvre le tube

Tube : utilisation

- Un tube ne peut être ouvert, la primitive pipe crée et ouvre le tube
- Après fermeture le tube devient inaccessible

Tube : utilisation

- Un tube ne peut être ouvert, la primitive pipe crée et ouvre le tube
- Après fermeture le tube devient inaccessible
- Pour l'utiliser à haut niveau il faut utiliser `fdopen()`

Tube : utilisation

- Un tube ne peut être ouvert, la primitive pipe crée et ouvre le tube
- Après fermeture le tube devient inaccessible
- Pour l'utiliser à haut niveau il faut utiliser `fdopen()`
- Les descripteurs étant transmis dans l'héritage durant `fork()`, l'utilisation d'un tube entre deux processus implique que le tube ait été créé par un ancêtre commun

Tubes : exemple

- Exemple : tube de communication entre deux processus

Tubes : exemple

- Exemple : tube de communication entre deux processus
- Les tubes de communication permettent à deux processus de communiquer

Tubes : exemple

- Exemple : tube de communication entre deux processus
- Les tubes de communication permettent à deux processus de communiquer
- Une utilisation classique consiste à rediriger :

Tubes : exemple

- Exemple : tube de communication entre deux processus
- Les tubes de communication permettent à deux processus de communiquer
- Une utilisation classique consiste à rediriger :
  - La sortie standard du premier dans le tube

Tubes : exemple

- Exemple : tube de communication entre deux processus
- Les tubes de communication permettent à deux processus de communiquer
- Une utilisation classique consiste à rediriger :
  - La sortie standard du premier dans le tube
  - L'entrée standard du second depuis le tube

Tubes : exemple

- Exemple : tube de communication entre deux processus
- Les tubes de communication permettent à deux processus de communiquer
- Une utilisation classique consiste à rediriger :
  - La sortie standard du premier dans le tube
  - L'entrée standard du second depuis le tube
- Par exemple :

```
ls -l -a répertoire | wc
```

`ls -l -a répertoire | wc`

- Le programme C suivant est équivalent à la commande
  `ls -l -a répertoire | wc`

`ls -l -a répertoire | wc`

- Le programme C suivant est équivalent à la commande
  `ls -l -a répertoire | wc`
- Le programme comporte deux processus

`ls -l -a répertoire | wc`

- Le programme C suivant est équivalent à la commande `ls -l -a répertoire | wc`
- Le programme comporte deux processus
- Le père exécute la commande `wc`

`ls -l -a répertoire | wc`

- Le programme C suivant est équivalent à la commande `ls -l -a répertoire | wc`
- Le programme comporte deux processus
- Le père exécute la commande `wc`
- Le fils la commande `ls -l -a répertoire`

`ls -l -a répertoire | wc`

- Le programme C suivant est équivalent à la commande `ls -l -a répertoire | wc`
- Le programme comporte deux processus
- Le père exécute la commande `wc`
- Le fils la commande `ls -l -a répertoire`
- À la terminaison du fils le processus père recevra EOF comme résultat de lecture ce qui rend inutile l'attente de la fin du fils

`ls -l -a répertoire | wc` : création du tube

```
#include <unistd.h>
#include <stdlib.h>
#include<stdio.h>

int main(int argc, char *argv[])
{
    pid_t fils;
    int tube[2];

    /* Vérification de l'existence d'un paramètre au moins */
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <répertoire>\n", argv[0]);
        return EXIT_FAILURE;
    }
    if (pipe(tube) != 0) {
        perror("Création du tube");
        return EXIT_FAILURE;
    }
```

`ls -l -a répertoire | wc` : processus fils

```
/* Création du fils exécutant ls –l –a */
fils = fork();
switch (fils) {
case –1:
    perror("Père : échec du fork");
    return 1;
case 0:
    close(tube[0]); /* lecture tube jamais utilisé */
    close(STDOUT_FILENO); /* redirection de stdout */
    dup(tube[1]);
    /* fermeture de descripteurs non utilisés devenus inutiles */
    close(tube[1]);
    /* on peut maintenant lancer l'exécution de ls –l */
    fprintf(stderr, "On lance ls -l -a %s\n", argv[1]);
    execlp("ls", "ls", "-l", "-a", argv[1], (char *)NULL);
    perror("Échec de l'exécution du programme ls");
    return EXIT_FAILURE;
}
```

`ls -l -a répertoire | wc` : processus père

```
    /* Processus père :on peut maintenant lancer l'ex'ecution de wc –l */
    close(tube[1]); /* jamais utilisé */
    close(STDIN_FILENO); /* redirection de stdin */
    dup(tube[0]);
    /* fermeture de descripteurs non utilisés devenus inutiles */
    close(tube[0]);
    execlp("wc", "wc", (char *)NULL);
    perror("Échec de l'exécution de wc");
    return EXIT_FAILURE;
}
```