Cours «système d'exploitation»
2ème année
Département d'Informatique

Chapitre 4
Les Signaux

Plan

1. **Système de Gestion des Fichiers : Concepts avancés.**
2. **Création et ordonnancement de Processus.**
3. **Synchronisation de Processus.**
4. **Communication entre Processus : les Signaux.**
5. **Echange de données entre Processus.**
6. **Communication entre Processus : les IPC.**

4. Les Signaux

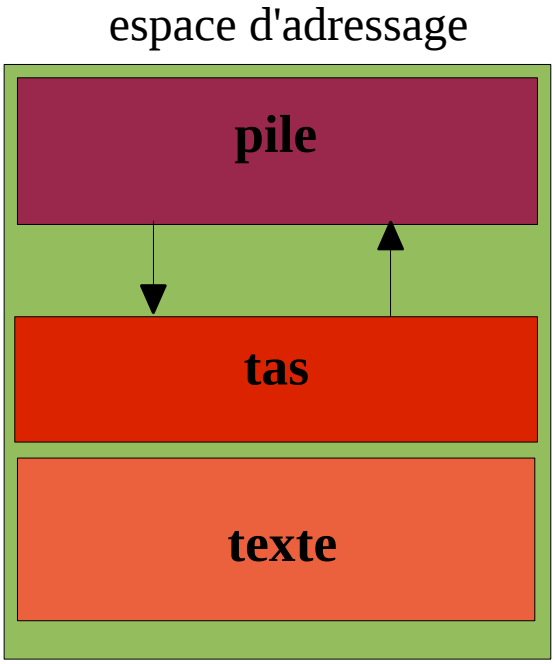
Un signal est "**un message très court**" qu'un processus peut envoyer à un autre processus, pour lui dire qu'un **événement particulier est arrivé**. Le processus pourra alors mettre en oeuvre une réponse décidée et pré-définie à l'avance (***handler***).

Ce mécanisme est implanté par un **moniteur**, qui scrute en permanence l'occurrence des signaux.

C'est par ce mécanisme que le système communique avec les processus utilisateurs en cas d'erreur (violation mémoire, erreur d'E/S) ou à la demande de l'utilisateur lui-même (caractères d'interruption ^D,^C ...).

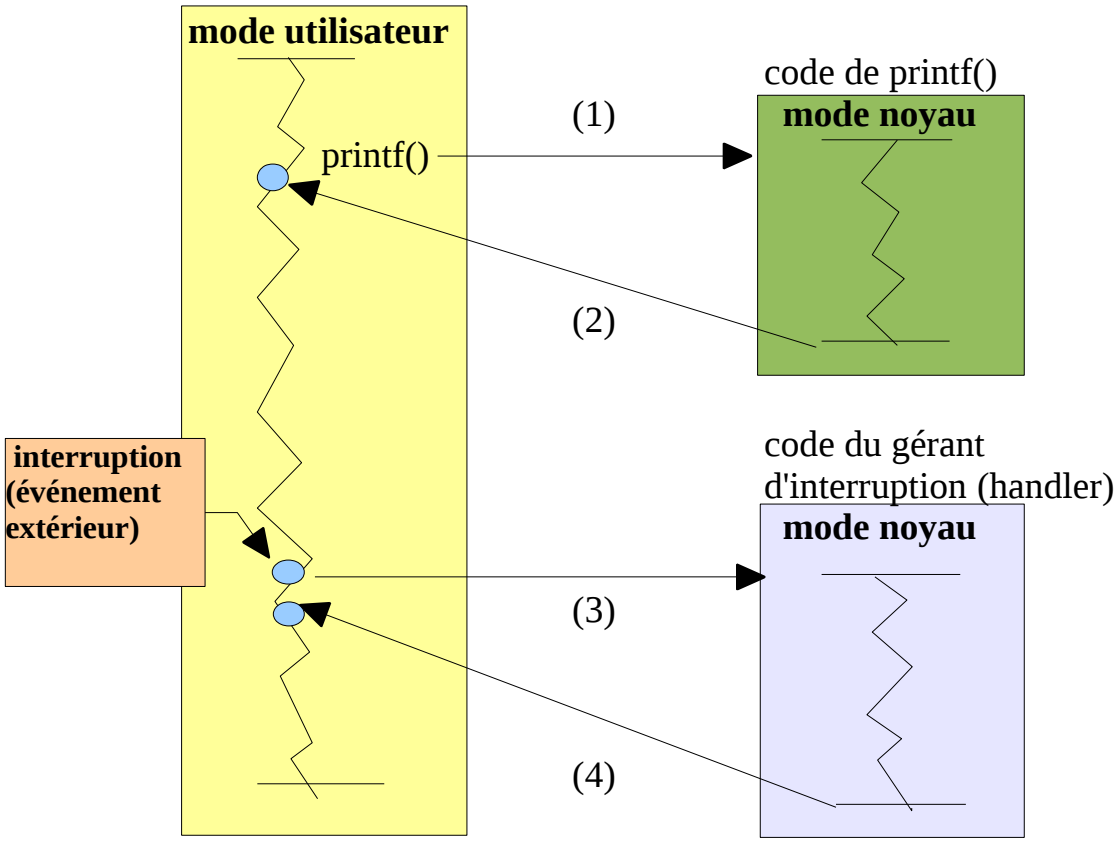
Rappels

Les processus sont composés d'un espace de travail en mémoire formé de 3 segments :



- Soit le processus accède aux objets de son espace d'adressage ⇒ **mode utilisateur**
- Soit le processus accède à des objets extérieurs (fichiers, librairies systèmes, ...) à son espace d'adressage ⇒ **mode noyau**

Le moniteur contient un certain nombre de primitives permettant l'émission de signaux asynchrones vers un processus donné et l'indication de procédure (*handler*) à exécuter à la réception d'un signal donné (ignorer un signal, provoquer une interruption ou un déroutement vers un *handler* spécifié par l'utilisateur).



En fait il existe 64 signaux différents numérotés à

partir de 1. Ces signaux portent également des noms :

SIGHUP(1), SIGINT(2), SIGQUIT(3), SIGTRAP(5), SIGFPE(8), SIGKILL(9), SIGSYS(31), SIGPIPE(13), SIGALARM(14), SIGTERM(15), SIGUSR1(10), SIGUSR2(12).

On les trouve dans "/usr/include/signal.h".

Envoyer un signal revient à envoyer ce numéro à un processus. Tout processus a la possibilité d'émettre à destination d'un autre processus un signal à condition que ses numéros de propriétaires (UID) lui en donnent le droit vis-à-vis de ceux du processus récepteur.

Métaphore de la sonnerie du téléphone

Un signal est **délivré** à un processus quand celui-ci le prend en compte (*décrochage du téléphone*).

Quand ? Au passage du mode "**actif noyau**" au mode "**actif utilisateur**" [(1)].

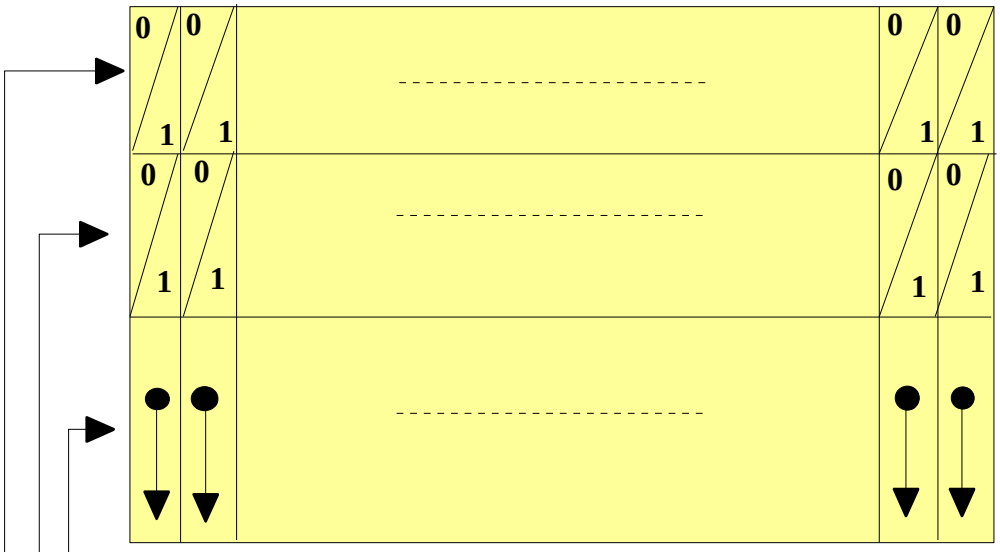
Ex. :

- sur le retour d'un gérant d'interruption matérielle,
- quand le processus vient d'être élu par l'ordonnanceur,
- sur un retour d'appel système.

Conséquence ? L'indicateur de signal pendant correspondant passe de 1 à 0.

Un processus n'est pas interruptible par un signal lorsqu'il est en mode noyau.

Signal : interruption logicielle.



- indicateurs des signaux pendants ;
- indicateurs des signaux bloqués ;
- indicateurs des comportements (incluant un pointeur vers un "handler").

Un 2ème signal identique arrive alors que le 1er est pendant (sonne), le second est perdu !

Dans BSD et dans la norme POSIX, on peut différer la délivrance des signaux : ils sont **bloqués** ou **masqués**.

L'arrivée d'un signal sur un processus endormi à un niveau de priorité interruptible(*), le réveille :

il passe à l'état prêt ; à son élection le signal sera délivré (exécuté avant le code du processus).

(*) Appels systèmes suivants :

- attente de signal (pause ...),
- wait/waitpid,
- open bloquant (tube nommé, terminal),
- pose d'un verrou (fnctl) en mode bloquant sur fichier régulier,
- read sur terminal, fichier verrouillé ou socket,
- envoi et extraction de messages (files de messages),
- opérations sur les sémaphores.

Si le **handler** du signal délivré a pour effet d'ignorer le signal, l'appel système est repris : le processus repart éventuellement à l'état endormi sur le même événement.

Les principaux signaux

SIGHUP (1)	il est envoyé lorsque la connexion physique de la ligne est interrompue ou en cas de terminaison du processus leader de la session ;
SIGINT (2)	frappe du caractère **intr** (interruption clavier) sur le clavier du terminal de contrôle ;
SIGQUIT (3)	frappe du caractère **quit** (interruption clavier avec sauvegarde de l'image mémoire dans le fichier de nom **core**) sur le clavier du terminal de contrôle ;
SIGKILL (9)	signal de terminaison non déroutable.
SIGTERM (15)	signal de terminaison, il est envoyé à tous les processus actifs par le programme **shutdown**, qui permet d'arrêter proprement un système UNIX. Terminaison normale.

D'autres signaux

SIGILL (4)	Instruction illégale.
SIGFPE (8)	Erreur arithmétique.
SIGUSR1 (10)	Signal 1 défini par l'utilisateur.
SIGSEGV (11)	Adressage mémoire invalide.
SIGUSR2 (12)	Signal 2 défini par l'utilisateur.
SIGPIPE (13)	Écriture sur un tube sans lecteur.
SIGALRM (14)	Alarme.

D'autres signaux (fin)

SIGCHLD (17)	Terminaison d'un fils.
SIGCONT (18)	Reprise du processus.
SIGSTOP (19)	Suspension du processus (non déroutable).
SIGTSTP (20)	Émission vers le terminal du caractère de suspension.
SIGTTIN (21)	Lecture du terminal pour un processus d'arrière-plan.
SIGTTOU (22)	Écriture vers le terminal pour un processus d'arrière-plan.

Lorsqu'un processus reçoit un signal, il exécute une **routine spéciale**. Cette routine peut être une routine standard du système ou bien une routine spécifique fournie par l'utilisateur. Il est également possible d'ignorer la réception d'un signal, c'est-à-dire de n'exécuter aucun traitement.

Il existe 2 routines standards :

- La première provoque purement et simplement la mort du processus qui reçoit le signal.
- La seconde copie dans un fichier nommé **core** toutes les informations concernant le processus en cours, affiche le message "***core dumped***", puis provoque la mort du processus. Le fichier **core** généré pourra être utilisé pour examiner l'état du processus au moment où il a reçu le signal.

Par défaut, la première routine est associée (entre autre) aux signaux SIGHUP, SIGINT, SIGKILL et SIGTERM, la deuxième routine est associée (entre autre) au signal SIGQUIT.

Un processus lancé en tâche de fond ne peut plus être interrompu par SIGINT ni SIGQUIT.

L'Envoi de signaux

Pour envoyer un signal à un processus, on utilise la commande appelée **kill**. Celle-ci prend en option (c'est-à-dire précédée du caractère '-') le numéro du signal à envoyer et en argument le numéro du (ou des) processus destinataire(s).

Exemple 1 :

```
$ kill 36
$
```

par défaut le signal 15 (SIGTERM) est envoyé.

Exemple 2 :

```
$ kill 0
$
```

Envoie le signal 15 à tous les processus fils, petits-fils ... tous ceux lancés depuis ce terminal.

Exemple 3 :

```
$ kill -9 36
$
```

Envoie le signal de numéro 9 (SIGKILL) au processus de numéro 36.

Comme nous l'avons vu, un événement particulier est associé à tout signal. Cependant un signal peut être adressé directement à un processus par un autre processus sans que cet événement ne se soit produit.

Par exemple, le signal de nom symbolique SIGSEGV est adressé à un processus en cas de référence mémoire erronée, mais il est possible de lui envoyer manuellement (shell) le signal par la commande :

```
$ kill -TERM pid_process
```

Ainsi lorsqu'on dit que la seule information véhiculée par un signal est son type, cela signifie qu'à la prise en compte du signal, le récepteur ne peut savoir si l'événement sous jacent s'est ou non produit et dans ce dernier cas l'identité de l'émetteur.

La commande **kill** utilisée avec l'argument '**-l**' renvoie la liste des signaux du système :

```
$ kill -l
HUP INT QUIT ILL TRAP IOT EMT
FPE KILL BUS ... TTOU
$
```

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

Un tel appel à cette primitive a pour effet d'envoyer le signal de numéro ***sig*** au(x) processus déduit(s) de la valeur de ***pid***.

Les processus destinataires sont les suivants en fonction de la valeur de ***pid*** :

<-1	tous les processus du groupe **|pid|**
-1	tous les processus du système (sauf 0 et 1)
0	tous les processus dans le même groupe que le processus
> 0	processus d'identité **pid**

La valeur du paramètre ***sig*** a l'interprétation suivante :

< 0 ou > NSIG	valeur incorrecte ;
0	pas de signal envoyé mais test d'erreur (test d'existence de processus par exemple) ;
sinon	signal de numéro ***sig***.

La primitive renvoie 0 ou 1 selon qu'elle s'est déroulée normalement (0) ou pas (1).

Il existe d'autres primitives systèmes comme par exemple **raise**. Cette fonction fait partie de l'interface standard du langage C et non de POSIX.

```
int raise (int sig);
```

Elle envoie le signal de numéro ***sig*** donné au processus courant.

Le système possède un *handler* par défaut (**SIG_DFL**) qui définit un comportement par défaut pour un processus qui reçoit un signal type.

Les différents comportements gérés par ce *handler* sont :

- terminaison du processus,
- terminaison du processus avec image mémoire (fichier core),
- signal ignoré,
- suspension (SIGSTOP) du processus,
- continuation (SIGCONT) : reprise du processus stoppé et ignoré sinon.

Mis à part pour certains signaux particuliers (**SIGKILL**, **SIGCONT** et **SIGSTOP**) pour lesquels le *handler* **SIG_DFL** par défaut est le seul possible, tout processus peut installer, pour chaque type de signal, un nouveau *handler*.

Les autres types de handler possibles sont :

- Une valeur standard désignée par la constante symbolique **SIG_IGN** indiquant que le processus doit ignorer le signal. C'est le cas des processus lancés en arrière-plan par un shell.

  signal (N_SIGNAL, SIG_IGN) ;

- Toute fonction définie par l'utilisateur qui sera exécutée à la délivrance du signal correspondant. Un signal ayant cette propriété est dit **capté** par le processus.

Les signaux permettent alors de réaliser des appels de fonctions qui ne sont pas écrits dans le programme (un peu comme si l'appel de fonction était inséré automatiquement dans le code à la délivrance du signal et supprimé immédiatement au retour de l'appel).

Les fonctions utilisables comme *handler* sont des fonctions ne renvoyant pas de valeur, c'est-à-dire dont le type du résultat est ***void***. A l'appel du *handler* correspondant à la délivrance d'un signal d'un type particulier, la fonction reçoit en paramètre le numéro du type du signal.

Un *handler* est défini de la manière suivante :

void *handler* (int *sig*);

Un processus n'est pas interruptible par un signal en mode noyau, il faut qu'il passe de l'état actif noyau à l'état actif utilisateur ; c'est le cas lorsque le processus passe à l'état élu après une interruption matérielle ou lorsqu'il revient d'un appel système.

L'arrivée d'un signal à un processus endormi à un niveau de priorité interruptible le réveille : le processus passe à l'état prêt, à son élection le signal est délivré.

Si un processus est **zombi**, les signaux sont sans effet sur lui (en effet il est terminé). Rappelons que les processus *zombi* disparaissent quand leur père prend connaissance de cette terminaison.

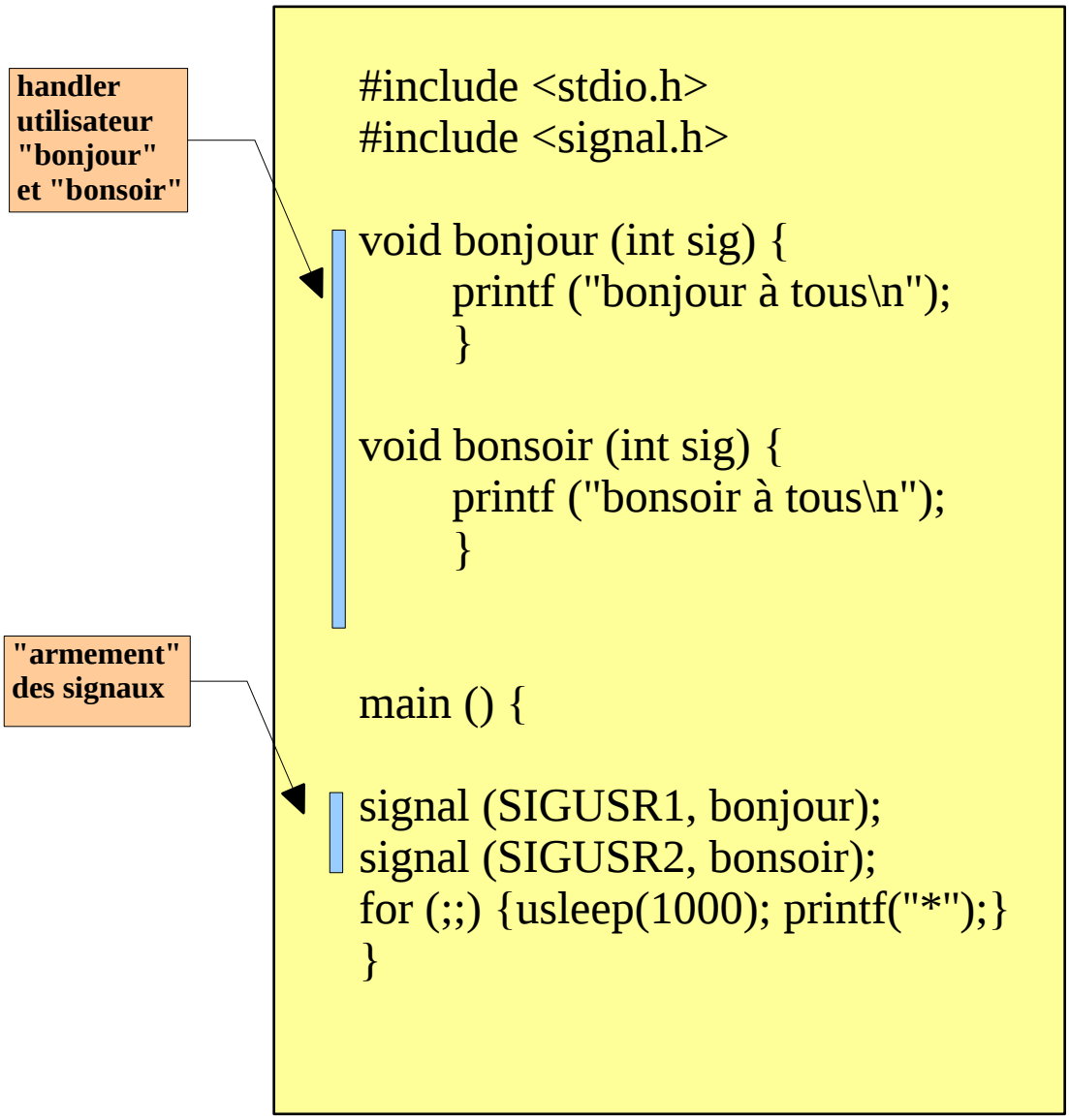
La manipulation des *handler* se fait avec la primitive ***signal*** ou la commande ***trap*** dans le shell.

A la délivrance d'un signal capté, le *handler* par défaut **SIG_DFL** est réinstallé dans les versions ATT (System V) et ne l'est pas dans les versions BSD, ce qui désactive le *handler* correspondant. Pour avoir un aspect permanent, un *handler* installé dans ces versions au moyen de la fonction signal doit donc procéder à une réinstallation.

Typiquement, le code d'un tel *handler* sera de la forme :

```
void handler (int sig) {
        ..............
        ..............
signal (sig, handler);
        ..............
        ..............}
```

Voici un exemple d'utilisation de *handler* définis par l'utilisateur :



Sur l'exemple précédent on pourra envoyer dans le shell les signaux correspondants aux processus à l'aide de la commande kill :

```
$ kill -USR1 pid
$ kill -USR2 pid
```

Au retour du *handler* appelé lors de la délivrance d'un signal capté, l'exécution du processus reprend, sauf demande contraire, au point où le processus a été interrompu.

Des reprises sur des points quelconques de sa pile préalablement repérés peuvent être demandées

Il existe d'autres primitives permettant la synchronisation des processus. C'est le cas des primitives ***sleep*** et ***pause***.

- ✿ La primitive ***sleep (int durée)*** permet de bloquer (état endormi) le processus courant sur la durée passée en paramètre.

- ✿ La primitive ***pause*** permet de mettre le processus courant en attente de l'arrivée de signaux.

```
#include <unistd.h>
int pause (void);
```

Soit le processus se termine, si le *handler* associé est **SIG_DFL** et que ce *handler* a pour action de terminer le processus,

Soit le processus exécute le *handler* correspondant au signal capté.

Cette primitive ne permet ni d'attendre l'arrivée d'un signal de type donné, ni au réveil de savoir quel signal a réveillé le processus. Pour le savoir, une solution est de récupérer cette information par l'intermédiaire du ou des handler de signaux.

On peut également manipuler les signaux directement dans le shell, en associant un traitement à un ou plusieurs signaux. Pour cela on utilisera la commande ***trap*** :

```
$ trap commande numéro_de_signal ...
```

Exemple :

```
$ trap who 2 15
$ trap "echo signal 3 recu" 3
$
```

Si le signal 2 ou 15 arrivent, la commande **who** sera lancée. Si le signal 3 arrive, c'est la commande "**echo signal 3 recu**" qui sera lancée.

Vous pouvez voir les différents traitements associés aux signaux en tapant :

```
$ trap
2:who
3:echo signal 3 recu
15:who
$
```

Lorsqu'un signal est associé à son traitement par défaut, rien n'est affiché.
Pour ignorer un signal, vous pouvez taper :

```
$ trap  " "   2
$ trap
2:
3:echo signal 3 recu
15:who
$
```

Exemple d'envoi de signaux en shell (résumé) :

```
$ trap who 2 15
$ trap «echo signal 3 recu» 3
$ trap
2:who
3:echo signal 3 recu
15:who
$ ps
PID   TTY      STAT      TIME      COMMAND
229   pp0                0:00      -bash
263   pp0      R         0:00      ps
$ kill -2 229
user_name     ttyp0      Apr 11 08:57
$ kill -15 229
user_name     ttyp0      Apr 11 08:57
$ kill -TERM 229
user_name     ttyp0      Apr 11 08:57
$ kill -3 229
signal 3 recu
$
```

Le pseudo-signal "0" est reçu par le processus lorsque celui-ci se termine. Vous pouvez donc lui associer un traitement, par exemple la destruction des fichiers temporaires :

```
$ trap "rm fich.tmp" 0
```

Les traitements associés aux signaux font partie de l'environnement d'un processus, au même titre que les fichiers ouverts ou le terminal de contrôle. Ils sont donc transmis aux processus fils. La configuration des signaux ignorés ou associés à leur traitement par défaut reste inchangée, mais les signaux associés à un traitement spécifique sont ré-associés à leur traitement par défaut.