Cours de Réseau et communication Unix n°1

Edouard THIEL

Faculté des Sciences

Université d'Aix-Marseille (AMU)

Septembre 2014

Les transparents de ce cours sont téléchargeables ici :
http://pageperso.lif.univ-mrs.fr/~edouard.thiel/ens/rezo/

Lien court : http://j.mp/rezocom

Présentation de l'UE

- 10 séances CM/TD/TP
- Rendre les TPs par mail à la fin de la séance, 3 TP notés
- Examen final : droit aux notes de CM/TD/TP, pas aux annales ni aux livres
- Note finale $=$ max (examen ; $0.7\,$examen $+\,0.3\,$TPs)
- Contenu :
  - Rappels boîte à outil Unix en C.
  - Points de communication du système de fichiers.
  - Modèles de client-serveur.
  - Sockets des domaines Unix et Internet.
  - Modèle en couche, protocoles, architecture réseau.
  - Réalisation de nombreux clients-serveurs, en particulier un aspirateur web et un serveur web multi-clients en C.

Ressources

- page web du cours : transparents, annales corrigées, liens, etc.
- Prise de notes : noter ce qui semble important, renvoyer au numéro de transparent quand beaucoup d'informations.

Pour aller plus loin :

- `man`
- C. Blaess, *Développement système sous Linux*, → contient à peu près tout notre cours et bien plus !
- JP. Gourret, *Programmation système* → début de notre cours
- ouvrages externes de réseau : Pujolle, Tanenbaum . . .

Plan du cours n°1

1. Processus Unix
2. Terminaison des processus
3. Synchronisation
4. Recouvrement de processus

5. Les signaux Unix
6. Envoi de signaux
7. Réception des signaux
8. Signaux usuels

1 - Processus Unix

Un processus =

- programme en cours d'exécution
- contexte d'exécution
- données du programme

Le système d'exploitation (SE) maintient une table des processus, où chaque processus a une entrée unique : le PID

Afficher la table des processus : `ps`, `pstree`, `top`

Système multi-tâche

- temps partagé (petits intervalles de temps)
- préemptif

Orchestré par l'ordonnanceur (uk: scheduler) :

- module du noyau
- chargé de l'avancement de l'exécution des processus
- calcul dynamique des priorités

Arbre des processus

- Chaque processus
  - a un parent, et possède 0, 1 ou + fils
  - est identifié par son PID
  - connaît le PPID de son parent
- Le processus racine est `init`, de PID 1
  - `init` adopte automatiquement les orphelins (processus dont le père est terminé)
  - Sur Ubuntu, les orphelins d'un user son adoptés par `init --user`, cf `pstree`

Obtenir le PID :

```
#include <unistd.h>      // Fonctions Unix standard
pid_t getpid  (void);    // pid_t = type entier
pid_t getppid (void);
```

Mécanisme de création

Mécanisme unique de création de processus :

Un processus (le père) demande, en appelant `fork()`, la création dynamique d'un nouveau processus (le fils).

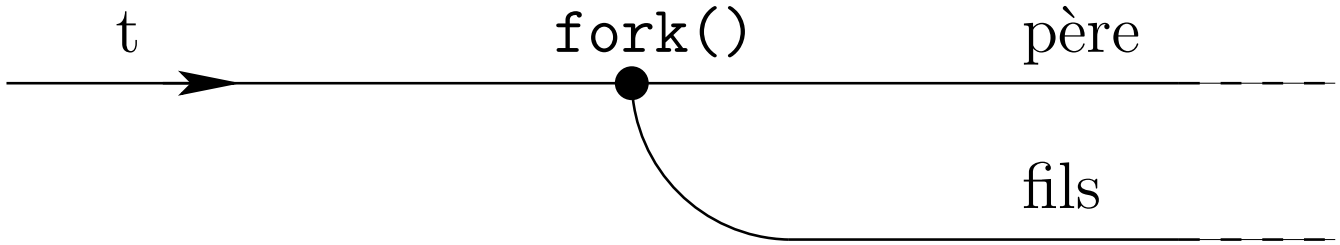
Le fils s'exécute ensuite de façon concurrente avec le père.

Tous les processus Unix sont créés de cette façon
(excepté le processus originel de PID 0).

Spécification :

```
#include <unistd.h>
pid_t fork (void);
```

Création par duplication



Avant `fork` :
un seul processus, qui va faire l'appel système.

Après `fork` :
2 processus reviennent de l'appel, et continuent le même programme.

Le fils est un clone du père : presque tout ce qui concerne le père est dupliqué (par *copy on write*), sauf : `man fork`

- le PID et PPID
- les temps d'exécutions
- alarmes et signaux en attente

→ Pas de partage des données : chaque processus travaille sur ses propres données, n'a pas d'accès aux autres.

Premier exemple

```
#include <stdio.h>          // printf
#include <unistd.h>         // fork, getpid

int main () {
  printf ("Je suis le futur père %d\n", (int) getpid());
  fork();
  printf ("Je suis %d\n", (int) getpid());
  return 0;
}
```

Exécution :
```
Je suis le futur père 436
Je suis 440
Je suis 436
```

Les 2 processus font bien le 2^e `printf`.
L'ordre d'affichage est aléatoire : dépend de l'ordonnanceur.

Comportement différent

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

int main () {
  pid_t p = getpid();
  printf ("Je suis le futur père %d\n", (int) p);
  fork();
  if (getpid() == p)
       printf ("Je suis le père %d mais ignore le "
               "PID de mon fils\n", (int) p);
  else printf ("Je suis %d fils de %d\n",
               (int) getpid(), p);
  return 0;
}
```

Exécution :

```
Je suis le futur père 444
Je suis le père 444 mais ignore le PID de mon fils
Je suis 448 fils de 444
```

Utilisation du retour de `fork`

`fork()` renvoie une valeur différente selon le processus :

`-1`	Échec création processus (seul le père revient)
`0`	On est dans le fils
`>0`	On est dans le père ; PID du fils

En cas d'erreur système, on affiche la cause avec :

```
#include <stdio.h>
void perror(const char *s);
```

Exemple à retenir

```
#include <stdio.h>      // printf, perror
#include <stdlib.h>     // exit
#include <unistd.h>     // fork, getpid, getppid

int main () {
  pid_t p;
  p = fork();
  if (p < 0) { perror("fork"); exit (1); }

  if (p == 0) {  /* Fils */
    printf ("Je suis %d fils de %d\n",
            (int) getpid(), (int) getppid());
    exit (0);
  } /* Fin fils */

  /* Suite du père */
  printf ("je suis %d père de %d\n", (int) getpid(), (int) p);
  exit (0);
}
```

2 - Terminaison des processus

La terminaison d'un processus survient

- à réception d'un signal Unix ;
- à la demande du programme lui-même.

À tout endroit du programme :

```
#include <stdlib.h>
void exit (int status);
```

→ Terminaison immédiate avec code de terminaison `status`

Seul l'octet de poids faible est significatif : 0 succès, $\neq 0$ échec.

Constantes `stdlib.h` : `EXIT_SUCCESS`, `EXIT_FAILURE`

Retour de `main`

Appel de `main` par le système :

```
_start() {

  int status = main (...);

  exit (status);
}
```

Dans `main` :

`return status;` est équivalent à `exit (status);`

Différence : dans un appel récursif de `main`, on revient d'un niveau.

Fonctions d'avant-terminaison

= fonctions de nettoyage appelées par le système juste avant la terminaison.

Enregistrer une fonction :

```
#include <stdlib.h>
int atexit ( void (*fonction)(void) );
```

On peut enregistrer plusieurs fonctions, elles seront appelées dans l'ordre inverse.

Nombre max : `ATEXIT_MAX` dans `<limits.h>`

Exemple

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

void au_revoir (void) { printf ("Au revoir\n"); }
void bye_bye   (void) { printf ("Bye bye\n"); }

int main () {
  atexit (au_revoir);
  atexit (bye_bye);
  printf ("Avant exit ...\n");
  exit (0);
  printf ("Après exit\n");
}
```

Exécution :

```
Avant exit ...
Bye bye
Au revoir
```

Durant la terminaison

Lorsqu'on appelle `exit()` :

- les fonctions d'avant-terminaison sont appelées ;
- les buffers sont flushés ;
- les fichiers ouverts sont fermés ;
- la mémoire est rendue ;
- le processus passe à l'état de Zombie.

Appel plus bas niveau : `_exit()` :

- les fonctions d'avant-terminaison ne sont pas appelées ;
- flush des buffers : système-dépendant ;
- (le reste est idem).

3 - Synchronisation

Un processus terminé passe à l'état de Zombie :

- tout ce qui le concerne est déchargé de la mémoire ;
- ne subsiste qu'une entrée dans la table des processus (état `DEFUNCTED`) avec son exit status.

Utilité du mécanisme :

permettre au père de connaître le exit status de son fils, même si le fils est mort depuis un moment.

Inconvénient :

Il faut nettoyer la table des processus des zombies, sinon vite saturée.

Attente de terminaison

Mécanisme unique pour relever le exit status et nettoyer la table :

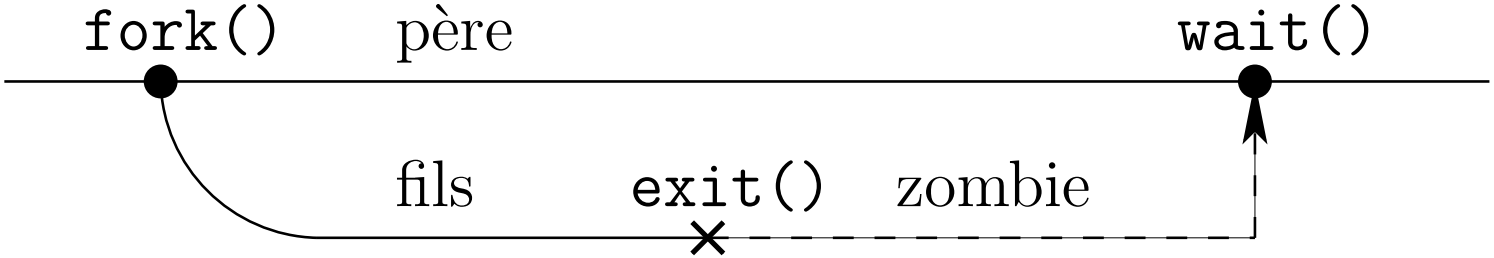
```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait (int *stat_infos);
pid_t waitpid (pid_t pid, int *stat_infos, int options);
```

`wait` attend la fin de n'importe quel fils et renvoie son PID ;
`waitpid` attend la fin du fils ayant le PID `pid`

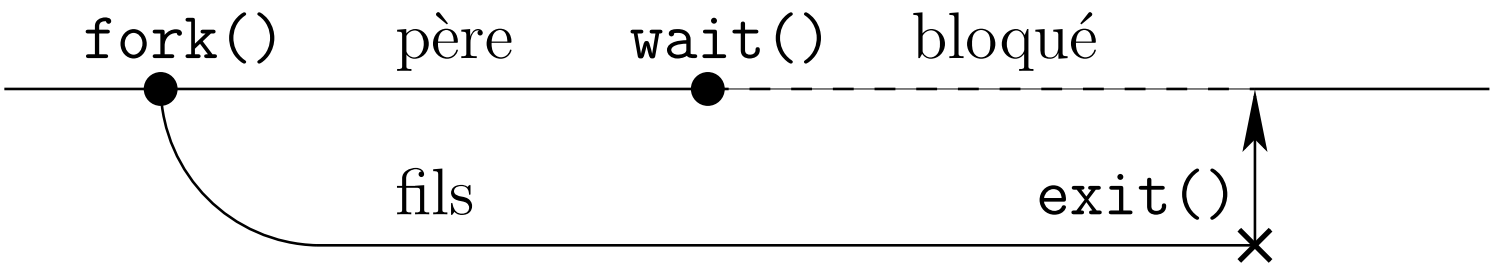
Appel de `wait`

A. Si le processus appelant ne possède aucun fils : retour immédiat, renvoie `-1`.

B. Si le processus appelant possède des fils zombies : retour immédiat, renvoie l'un des PID ; ce zombie est supprimé de la table des processus.



C. Si le processus appelant possède des fils mais aucun zombie : bloqué jusqu'à terminaison de l'un des fils, puis **B**.



Récupération du status

 `wait();` → `SIGSEGV`

`wait(NULL);` → attente du zombie sans récupération

Pour récupérer le status :

```
int status, stat_infos;
wait (&stat_infos);
status = WEXITSTATUS(stat_infos);
```

`stat_infos` contient le `exit status` du fils (décalé de 8 bits) plus d'autres informations sur la terminaison (`man wait`).

Attente d'un fils particulier

`waitpid (pid, &stat_infos, options);`

permet d'attendre la fin du fils `pid > 0`, ou de n'importe quel fils (`pid = -1`).

`options = 0` : attente bloquante

`options = WNOHANG` : retour immédiat

Remarque : à chaque terminaison de processus, un signal Unix `SIGCHLD` est envoyé au père $\rightarrow$ S'il le capte, il peut lever le zombie avec `waitpid (-1, NULL, WNOHANG);`

Mais que fait `init` ?

Lorsque le père A d'un fils B se termine, B est adopté par `init` (ou `init --user`)

- Pour B, `getppid() = 1` (ou le PID de `init --user`)
- Quand B se terminera, `init` recevra un signal `SIGCHLD`, puis supprimera le zombie B.

4 - Recouvrement de processus

Recouvrement = chargement d'un nouveau programme binaire ou script en mémoire.

- l'espace d'adressage du processus est entièrement écrasé : perte irréversible de l'ancien programme et des données ;
- retour d'un recouvrement uniquement s'il n'a pu être fait ;
- pas de création de nouveau processus : on conserve le PID, le PPID, la Table des Descripteurs (redirections) ;
- le nouveau programme démarre au début (`_start` puis `main`) avec de nouveaux arguments.

Recouvrement en bash

```
#! /bin/bash
exec ls -l /home
echo "ls: échec recouvrement" >&2
exit 1
```

Le script `bash` se recouvre avec la commande `ls` et les arguments `ls -l /home`.

$\rightarrow$ `ls` aura le même PID, PPID et redirections que le script original.

$\rightarrow$ Lorsque `ls` sera terminée, pas de retour dans le script (qui n'existe plus), c'est la fin du processus.

Recouvrement en C

```
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
            char *const envp[]);
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg,
              ..., char * const envp[]);
```

`execv*` : arguments par tableau `argv` terminé par `NULL`

`execl*` : arguments en liste `arg1, arg2, .., NULL`

Recouvrement en précisant le chemin

```
int execv(const char *path, char *const argv[]);
int execl(const char *path, const char *arg, ...);
```

Le programme est cherché dans le chemin path :

```
char *myargs[] = { "ls", "-l", "/home", NULL};
execv ("/bin/ls", myargs);
perror ("exec ls");
exit (1);
```

ou de manière équivalente :

```
execl ("/bin/ls", "ls", "-l", "/home", NULL);
perror ("exec ls");
exit (1);
```

Recouvrement avec le PATH

```
int execvp(const char *file, char *const argv[]);
int execlp(const char *file, const char *arg, ...);
```

Le programme `file` est cherché dans `$PATH` :

```
char *myargs[] = { "ls", "-l", "/home", NULL};
execvp ("ls", myargs);
perror ("exec ls");
exit (1);
```

ou de manière équivalente :

```
execlp ("ls", "ls", "-l", "/home", NULL);
perror ("exec ls");
exit (1);
```

Méthode alternative

Pour lancer une commande dans un programme C, au lieu de faire `fork` / `exec`, on peut appeler

```
#include <stdlib.h>
int system(const char *command);
```

Crée un fils et un petit-fils :

- Le fils se recouvre par le shell `/bin/sh` , qui va découper les arguments ;
- le petit-fils se recouvre avec la commande et les arguments.

Renvoie le `status` de la commande terminée.

Deux processus, impossibilité de choisir le shell, certains signaux bloqués, etc : à éviter !

5 - Les signaux Unix

Forme limitée de communication inter-processus (IPC) :

- notification asynchrone d'un évènement,
- envoyée à un processus ou à un groupe de processus.
- Seule information envoyée : le nom du signal.

Lorsqu'un processus reçoit un signal :

- son flot d'exécution est interrompu par le système ;
- puis un *handler* de signal est appelé ;
- enfin le flot d'exécution est éventuellement repris.

Un signal = une interruption logicielle.

handler de signal = fonction appelée pendant l'interruption.

Vu en TP

- `^C` provoque l'envoi du signal `SIGINT` au processus en avant-plan, ce qui l'INTerrompt.
- `^Z` provoque l'envoi du signal `SIGTSTP` au processus en avant plan, ce qui Temporairement le SToPe.
- `fg` ou `bg` provoque l'envoi du signal `SIGCONT` ce qui réveille (CONTinue) le processus.
- `kill pid` envoie le signal `SIGTERM` au processus `pid`, ce qui le TERMine.
- `kill -9 pid` ou `kill -KILL pid` envoie le signal `SIGKILL` au processus `pid`, ce qui le tue.
- Une violation d'adresse mémoire (sortie de tableau) provoque l'envoi de `SIGSEGV` (SEGment Violation) $\rightarrow$ core dump.
- Une division par zero provoque l'envoi de `SIGFPE` (Floating Point Exception) $\rightarrow$ core dump.

Bilan

Des signaux Unix peuvent être envoyés par

- le terminal (`^C` ou `^Z`)
- le shell (`fg` ou `bg`)
- l'utilisateur avec la commande `kill`
- le système (erreurs matérielles ou logicielles)
- etc

À réception d'un signal, un processus peut

- se terminer
- ignorer le signal
- être tué, endormi ou réveillé
- générer une image mémoire (core dump)
- capter le signal et décider quoi faire.

Portabilité

Différentes implémentations :

- BSD (implémentation originale, années 1970)
- SysV (début années 1980)
- POSIX.1 (normalisations 1990, 2001)

→ Différents signaux et comportements.

Chaque système implémente :

- les signaux standards POSIX
- éventuellement des signaux temps réel
- des signaux particuliers au système

→ portabilité limitée.

Identification des signaux

Chaque signal a un nom et un numéro.

Définis dans `signal.h`, voir `man 7 signal` :

`NSIG` nombre de signaux admis sur le système, numérotés `1 .. NSIG-1`

`SIG`*name* constante désignant le signal *name* :

```
SIGHUP  SIGINT  SIGQUIT  ...
  1       2       3
```

Connaître le numéro d'un signal :

```
<> kill -l TERM        # ou SIGTERM
15
<> kill -l 15
TERM
```

Afficher les signaux

```
<> kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS
34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7
42) SIGRTMIN+8  43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11
46) SIGRTMIN+12 47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15
50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11
54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3
62) SIGRTMAX-2  63) SIGRTMAX-1  64) SIGRTMAX
```

6 - Envoi de signaux

Permission :

un processus A peut envoyer un signal à un processus B si

- ils ont le même user
- ou si A est privilégié.

Envoi de signaux en bash

- `kill -`*num* *pid*
  `kill -`*name* *pid*
  `kill -SIG`*name* *pid*

Envoie le signal au processus *pid*.

```
<> kill -INT 1234
```

- Si on ne précise pas le signal, `SIGTERM` est envoyé par défaut.

```
<> kill 1234
```

- Si le pid est `-1`, le signal est envoyé à tous les processus permis (sauf `init` et lui-même).

```
<> kill -9 -1
```

Envoi de signaux avec `killall`

- Pour connaître les instances d'un programme :

```
<> pidof bash
3779 3759 3705 3639 3630 3575 2182
```

On peut donc envoyer un signal à toutes les instances :

```
<> kill -QUIT $(pidof bash)
```

- La commande `killall` le fait directement :

```
<> killall -QUIT bash
```

`SIGTERM` est envoyé par défaut :

```
<> killall bash
```

- On peut utiliser des regexp :

```
<> killall -QUIT -r 'gnome-(calc|sys).*or'
```

Envoi de signaux en C

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

Envoie le signal `sig` au processus `pid`
(si `-1` : à tous les processus permis sauf `init` et lui-même).

Renvoie 0 succès, -1 erreur.

Exemple :

```
if (kill (1234, SIGHUP) < 0) perror ("kill");
```

- ```
  #include <signal.h>
  int raise (int sig);
  ```

Envoie le signal `sig` au processus appelant ;
équivalent à `kill (getpid(), sig)`

Tester l'existence d'un processus

```
    kill -0 pid      (en bash)
ou  kill (pid, 0)    (en C)
```

n'envoie pas de signal,

mais réussit / renvoie 0 si le processus `pid` existe
(même à l'état de zombie),

sinon échoue / renvoie -1.

7 - Réception des signaux

Vie d'un signal :

1. Un signal est généré par `kill` ou `sigqueue` ;
2. il est en attente d'être délivré par le système ;
3. il est délivré au processus cible.

Un signal généré peut être bloqué par le système = il reste en attente.

Un processus peut placer un masque de signaux = une liste de signaux à bloquer.

Masque enlevé $\rightarrow$ les signaux en attente sont délivrés.

Atomicité

- Règle pour les signaux standards :

  une seule occurrence d'un signal X peut être en attente.

  $\rightarrow$ si un signal X est généré alors qu'un signal X est déjà en attente, il est perdu.

- Signaux temps réels (`SIGRTMIN .. SIGRTMAX`) :

  plusieurs occurrences d'un signal peuvent être en attente, voir `man getrlimit` et `man sigqueue`

Handler de signal

Fonction appelée par le système lorsqu'il délivre un signal `sig` pour que le processus capte ce signal.

Prototype : `void handler (int sig);`

Handlers fournis :

- `SIG_DFL` : handler par défaut
- `SIG_IGN` : pour ignorer un signal

Comportement de `SIG_DFL` selon `sig` :

(a) Terminaison du processus
(b) Terminaison avec image mémoire
(c) Ignorer le signal
(d) Processus suspendu
(e) Processus repris

Installer un handler avec `sigaction()`

Complètement normalisé par POSIX, remplace `signal()`

```
#include <signal.h>

int sigaction (int sig, const struct sigaction *act,
                                    struct sigaction *old);
struct sigaction {
  void     (*sa_handler)(int);
  sigset_t   sa_mask;
  int        sa_flags;
};
```

Installe le handler `act->sa_handler` pour le signal `sig`.

Si `old` $\neq$ `NULL`, récupère l'ancien handler dans `old->sa_handler`.

Renvoie 0 succès, -1 erreur.

Options

De nombreux paramétrages sont possibles.

```
act.sa_flags = flag1 | flag2 | ...;
```

- Par défaut, le handler est maintenu après délivrance.

  On peut demander la réinstallation de `SIG_DFL` :

  ```
  act.sa_flags |= SA_RESET_HAND;
  ```

- Par défaut, tout appel bloquant est interrompu par un signal, puis échoue avec `errno = EINTR`.

  On peut demander que les appels bloquants soient silencieusement repris : `act.sa_flags |= SA_RESTART;`

  → plus facile à gérer : aucun code supplémentaire

Une fonction utilitaire

```
int bor_signal (int sig, void (*h)(int), int options)
{
  int r; struct sigaction s;
  s.sa_handler = h; sigemptyset (&s.sa_mask);
  s.sa_flags = options;
  r = sigaction (sig, &s, NULL);
  if (r < 0) perror ("bor_signal");
  return r;
}
```

Exemple d'usage :

```
void capter (int sig)
{
  printf ("Reçu signal %d\n", sig);
}
int main ()
{
  bor_signal (SIGTERM, capter, SA_RESTART);
  bor_signal (SIGQUIT, capter, SA_RESTART);
  bor_signal (SIGHUP, SIG_IGN, SA_RESTART);
  while (1) pause ();
}
```

Délivrance des signaux

- L'appel d'un handler peut avoir lieu lorsque le processus
  - revient d'une interruption matérielle ;
  - vient d'être élu par l'ordonnanceur ;
  - revient d'un appel système.

- Un signal est sans effet sur un zombie, même `SIGKILL`.

Signaux captables

- Tous les signaux sont captables sauf :

  `SIGKILL`, `SIGSTOP`, `SIGCONT`

- La différence entre `SIGSTOP` et `SIGTSTP` :

  `SIGTSTP` peut être capté.

- Si le processus est stoppé (il a reçu `SIGSTOP` ou `SIGTSTP`)
  - un signal `SIGTERM` ou `SIGCONT` le réveille ;
  - les autres signaux seront délivrés au réveil.

Duplication et recouvrement

- Lors de la duplication avec `fork` :
  - les handlers sont conservés ;
  - les signaux en attente sont supprimés dans le fils.

- Lors d'un recouvrement avec `exec*` :
  - les handlers sont remis à `SIG_DFL` ;
  - sauf pour les signaux ignorés, qui le restent.

8 - Signaux usuels

- `SIGHUP` : Hang UP

adressé à tous les processus d'une session lorsque le processus leader d'une session se termine.

Effet : terminaison

Protéger une commande : `nohup commande`

- `SIGINT` :

adressé à tous les processus d'un groupe en premier plan sous le contrôle d'un terminal lorsque frappe `^C`.

Effet : terminaison

Signaux avec image mémoire

- `SIGSEGV` : SEGment Violation

Violation de page mémoire

Effet : terminaison + core dump

- `SIGFPE` : Floating Point Exception

Division par zéro

Effet : terminaison + core dump

Signaux courants

- `SIGUSR1`, `SIGUSR2`

Sans signification absolue

Effet : terminaison

- `SIGCHLD` : fin d'un CHiLD

Envoyé au père lorsqu'un fils se termine

Effet : aucun (signale un zombie à lever)

Signal retardé

- `SIGALRM` : ALaRMe

Signal émis à un processus à sa propre demande, au bout d'un délai fixé par :

```
#include <unistd.h>
unsigned int alarm (unsigned int seconds);
```

Chaque nouvel appel annule et remplace le précédent.

Si `seconds` est 0, le décompte est annulé.

Renvoie le nombre de secondes restant avant la fin prévue du précédent appel, sinon 0.

Effet de `SIGALRM` : termine le processus.

Ne pas utiliser en même temps `alarm` et `sleep`