

Signaux

Présentation: Stéphane Lavirotte

Auteurs: ... et al*

**(*) Cours réalisé grâce aux documents de :
Stéphane Lavirotte, David Picard, Marc Pouzet,
Jean-Paul Rigault**

Mail: Stephane.Lavirotte@unice.fr

Web: <http://stephane.lavirotte.com/>

Université de Nice - Sophia Antipolis

Transition: Communication entre Processus

- ✓ **Communication entre un père et son fils**
 - Zone de données après `fork()` (mais à sens unique)
 - Paramètres après `exec()`
 - Tube (`pipe()`)

- ✓ **Communication entre deux processus sans lien**
 - Pipe nommé (fichier spécial `FIFO`)
 - Communication via « vrais » Fichiers... (à bannir)

- ✓ **Problèmes:**
 - Limitations (ne permet pas de tout faire)
 - Certaines pas souhaitables (communication par fichier...)



Gestion des Signaux

« Un sourire ça change tout »

Définition des Signaux

- ✓ **Moyen simple de communication**
 - Entre processus
 - Entre le noyau et les processus
- ✓ **Signaux:**
 - Événements externes qui changent le déroulement d'un processus de manière **asynchrone**
 - Un signal peut être bloqué par le processus
 - Différentes origines des signaux (logiciel, matériel)
- ✓ **Notion de signal**
 - interruption terminal (^C, ^Z, ^\)
 - terminaison d'un processus fils
 - erreur arithmétique (division par 0)
 - violation de protection mémoire

Etat, action associée

✓ État d'un signal ; action associée

- Ignoré (et donc perdu !)
- Associé à son action par défaut
 - dépend du signal (rien, suspension, reprise, terminaison avec ou sans core...)
- Associé à une action définie par l'utilisateur (piégé, capturé, « trappé »)
 - « *handler* » de signal (fonction utilisateur)

✓ Signal différé (masqué, bloqué)

- Le signal est mémorisé
- L'action sera effectuée lors du déblocage (démasquage)

Propriétés Signaux

- ✓ **Aucune priorité entre les différents signaux**
- ✓ **L'ordre de délivrance de plusieurs signaux « simultanés » n'est pas garantie**
- ✓ **Les signaux sont traités lorsque le processus passe du mode noyau au mode utilisateur**
 - **Au moment d'un changement de contexte**
 - **Au moment du retour d'un appel système**
- ✓ **Donc traitement asynchrone:**
 - **Envoyer la commande `kill -9 pid` ne signifie pas que le processus est interrompu immédiatement**
 - **Mais seulement qu'il sera interrompu dès lors que celui-ci écoute !**

Liste de Signaux Prédéfinis sous Unix

Signal	Val	Signification	ANSI Posix	core	Action déf.	Remarque
SIUGHUP	1	Coupure ligne terminal	P		Fin	
SIGINT	2	Interruption	A/P		Fin	^C au terminal
SIGQUIT	3	Quitter	P	✓	Fin	^\ au terminal
SIGILL	4	Instruction illégale	A/P	✓	Fin	Exception
SIGTRAP	5	Trace trap	P			
SIGABRT	6	Abort	A/P	✓	Fin	
SIGFPE	8	Exception calcul flottant	A/P	✓	Fin	Exception
SIGKILL	9	Terminaison forcée	P		Fin	ni ignorable, ni piégeable, ni blocable
SIGUSR1	10	Signal utilisateur	P		-	
SIGSEGV	11	Violation mémoire	A/P		Fin	Exception
SIGUSR2	12	Signal utilisateur	P		-	

Liste de Signaux Prédéfinis sous Unix

Signal	Val	Signification	ANSI Posix	core	Action déf.	Remarque
SIGPIPE	13	Écriture sur pipe sans lecteur	P		Fin	Exception
SIGALARM	14	Alarme (time-out)	P		-	
SIGTERM	15	Terminaison douce	A/P			
SIGCHLD	17	Changement état d'un fils	P		-	
SIGCONT	18	Reprise	P		Reprise	(fg et bg)
SIGSTOP	19	Suspension forcée	P		Suspension	ni ignorable, ni piégeable, ni blocable
SIGTSTP	20	Suspension douce	P		Suspension	^Z au terminal
SIGTTIN	21	Lecture terminal depuis un processus background	P		Suspension	Exception
SIGTTOU	22	Écriture terminal depuis un processus background	P		Suspension	Exception

Délivrance d'un Signal à un Processus

✓ Caractères spéciaux au terminal

- ^C= INT, ^Z= TSTP, ^\,,= ABRT ...

✓ Fonctions spéciales du Shell

- `kill -signal pid`
 - Envoie le *signal* au processus *pid*
- `fg (foreground)`
 - Reprend l'exécution au premier plan d'un processus suspendu
- `bg (background)`
 - Reprend l'exécution en arrière plan d'un processus suspendu

✓ Primitive Posix : fonction `kill()`

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

```
#include <signal.h>
```

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



Signaux et ANSI C

Les signaux définis dans la norme de
programmation C

Etat d'un Signal an ANSI C

Fonction `signal()` 1/3

```
#include <signal.h>
```

```
void (*signal(int sig, void (*ph)(int)))(int);
```

✓ **ou, si l'on préfère**

```
typedef void (*sighandler_t)(int);
```

```
Sighandler_t signal(int sig, sighandler_t ph);
```

- **Positionne l'action associée à la réception du signal `sig`**
- **L'action associée est `ph` (« *pointer to handler* »)**
 - **`SIG_IGN` : signal ignoré**
 - **`SIG_DFL` : action par défaut**
 - **une fonction utilisateur (paramètre `int`, retour `void`) : piégé**
- **Retourne l'ancienne action associée**

Etat d'un Signal an ANSI C

Fonction `signal()` 2/3

```
#include <signal.h>
void on_signal(int sig) {
    printf("*** signal %d\n", sig);
}

main() {
    void (*ph)(int);

    signal(SIGQUIT, SIG_IGN);
    ph = signal(SIGINT, SIG_IGN);
    printf("INT et QUIT ignorés\n");
    sleep(5);

    signal(SIGQUIT, on_signal);
    signal(SIGINT, on_signal);
    printf("INT et QUIT trappés\n");
    sleep(5);

    signal(SIGQUIT, SIG_DFL);
    signal(SIGINT, ph);
    printf("INT restauré "QUIT défaut\n");
    sleep(5);
}
```

Etat d'un Signal en ANSI C

Fonction `signal()` 3/3

```
$ test-signal  
INT et QUIT ignorés  
^ \ ^C INT et QUIT trappés  
^ \ *** signal 3  
^ \ *** signal 3  
^C *** signal 2  
INT restauré QUIT défaut  
^C  
$
```



Signaux et Posix 1

Les signaux définis dans la norme Posix

Etat d'un Signal en Posix

- ✓ **Inconvénients des signaux d'Ansi C**
 - Seulement 6 signaux
 - Impossibilité de consulter l'action/état courant(e)
 - Impossibilité de bloquer (masquer) d'autres signaux pendant l'exécution du *handler*
 - Pas de possibilité d'extension

- ✓ **Posix introduit de nouveaux mécanismes**
 - Fonction `sigaction()` comme remplacement de `signal()`
 - Blocage (masquage) de signaux (emprunté à BSD)
 - Permet de positionner le masque et le *handler* à l'appel

- ✓ **Attention à ne pas mélanger les deux appels systèmes**
 - Interagissent de manière étrange
 - Recommandation de plutôt utiliser `sigaction()`

Action Associée en Posix

	Bloqué	Débloqué
Ignoré	Rien	Rien
Associé à l'action par défaut	Action différée au déblocage	Action immédiate
Piégé	Exécution du <i>handler</i> différée ou déblocage	Exécution du <i>handler</i> immédiate

Masque des Signaux Posix

1/2

```
#include <signal.h>
int sigprocmask(int how,
                const sigset_t *set,
                sigset_t *old_set);
```

✓ **set** **contient l'ensemble des signaux à masquer ou démasquer**

✓ **how** **détermine la fonction à effectuer**

- SIG_BLOCK : **bloque les signaux de** set
- SIG_UNBLOCK : **débloque les signaux de** set
- SIG_SETMASK : **positionne le masque du processus à** set

✓ **old_set** **contient l'ancien masque**

```
int sigpending(sigset_t *set);
```

✓ **sigpending** **retourne les signaux bloqués en attente**

Masque des Signaux Posix

2/2

- ✓ **Ensemble de signaux**
 - Ensemble de bits, 1 bit par signal

- ✓ **Fonctions de manipulation**

```
#include <signal.h>
```

```
int sigemptyset(sigset_t *set);  
int sigfillset(sigset_t *set);  
int sigaddset(sigset_t *set, int sig);  
int sigdelset(sigset_t *set, int sig);  
int sigismember(const sigset_t *set, int sig);
```

Fonction `sigaction()`

```
#include <signal.h>

int sigaction(int sig,
               const struct sigaction *actp,
               struct sigaction *old_actp);
```

✓ Champs de `struct sigaction`

- `void (*sa_handler)(int)`
 - **fonction de capture (identique à `signal()`)**
- `sigset_t sa_mask`
 - **masque des signaux à bloquer lors de l'exécution du *handler***
- `int sa_flags`
 - **utile seulement pour `SIGCHLD`**

Exemple sigaction()

1/2

```

#include <signal.h>

void on_signal(int sig) {
    printf("*** signal %d\n", sig);
    sleep(5);
    printf("*** fin handler\n");
}

main() {
    struct sigaction sigact;
    sigset_t msk_int, msk_quit;

    sigemptyset(&msk_int);
    sigaddset(&msk_int, SIGINT);
    sigemptyset(&msk_quit);
    sigaddset(&msk_quit, SIGQUIT);
  
```

```

    sigact.sa_handler = on_signal;
    sigact.sa_mask = msk_quit;
    sigaction(SIGINT, &sigact, NULL);
    sigact.sa_mask = msk_int;
    sigaction(SIGQUIT, &sigact, NULL);
    printf("INT et QUIT trappés\n");
    sleep(10);
}
  
```

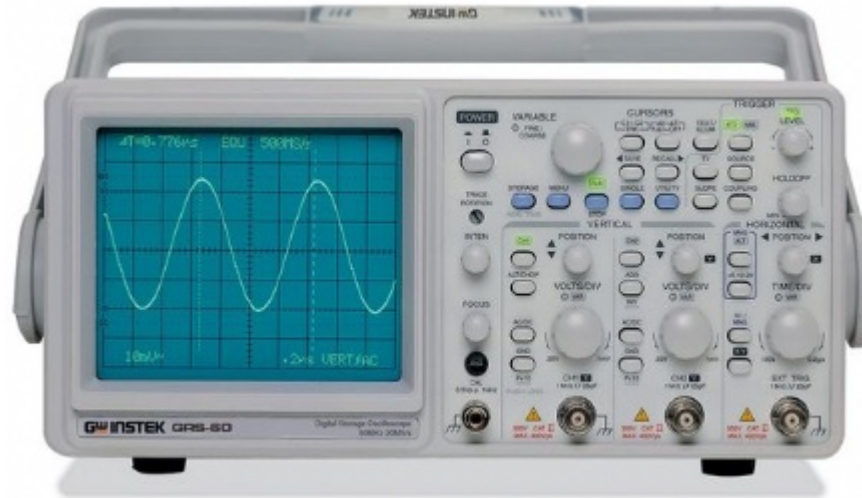
```

$ test-sigaction
INT et QUIT trappés
^C*** signal 2
^\\^\\*** fin handler
*** signal 3
*** fin handler
$
  
```

Durée de Vie du *Handler*

- ✓ Lorsque la fonction `sigaction()` est utilisée pour trapper un signal
 - le *handler* est valide jusqu'à ce qu'un prochain `sigaction()` l'invalide

- ✓ En revanche la durée de vie du *handler* établi par `signal()` est dépendante de l'implémentation
 - après réception du signal, l'action par défaut est rétablie
 - on est donc souvent conduit à réarmer le *handler* dans le *handler* lui-même (cas d'Unix SVR4, de Solaris...)



Signaux et autres Fonctions

Interaction entre les Signaux et les primitives `fork()` et `exec()`

- ✓ Attributs de processus
 - État des signaux (ignoré, action par défaut, trappé)
 - Masque des signaux bloqués
- ✓ Héritage de l'état et du masque lors d'un `fork()`
- ✓ Transmission de l'état et du masque à travers un `exec()`
 - sauf pour les **signaux piégés** qui sont rétablis à l'**action par défaut**

Autres Fonctions liées au Signaux

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

```
int sleep(unsigned int seconds);
```

- **Suspend le processus pendant le nombre de secondes indiqués, ou jusqu'à ce qu'un signal (non ignoré) arrive**

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

```
int pause();
```

- **Suspend le processus jusqu'à ce qu'un signal (non ignoré) arrive**
- **Cette fonction retourne après que le *handler* (éventuel) ait été exécuté**

Point de Reprise

✓ Motivation

- Permettre d'abandonner une fonction pour reprendre le traitement à un niveau plus élevé dans la chaîne (dynamique) d'appel
- Mécanisme primitif d'exception
- Goto non local

```
main() {  
    while (...) {  
        cmd = read_cmd();  
        execute(cmd);  
    }  
}
```

```
void execute(char *cmd) {  
    decode(cmd);  
    expand(cmd);  
    run(cmd);  
}
```

```
void decode(char *cmd) {  
    if (bad(cmd)) ????
```

Points de reprise en Ansi C

`setjmp()` et `longjmp()` 1/2

```
#include <setjmp.h>
int setjmp(jmp_buf env);
int longjmp(jmp_buf env, int v);
```

- ✓ `setjmp()` **positionne un point de reprise**
 - les informations sont mémorisées dans `env`
 - `env` doit être une variable globale
 - lors du premier passage, `setjmp()` retourne 0
- ✓ `longjmp()` **se branche au point de reprise** `env`, cad au `setjmp()` correspondant
 - la valeur `v` est alors retournée par `setjmp()` (ou 1 si `v = 0`)
 - l'environnement `env` doit être actif

Points de reprise en Ansi C

setjmp() et longjmp() 1/2

```
jmp_buf env;
```

```
void execute(char *cmd) {  
    decode(cmd);  
    expand(cmd);  
    run(cmd);  
}
```

```
main() {  
  
    while (...) {  
        if (setjmp(env) > 0)  
            puts("Again!");  
        cmd = read_cmd();  
        execute(cmd);  
    }  
}
```

```
void decode(char *cmd) {  
    if (bad(cmd))  
        longjmp(env, 1);  
}
```

Points de reprise en Ansi C

setjmp() et longjmp() 1/2

```
jmp_buf env;
```

```
void on_int(int sig) {  
    longjmp(env, 2);  
}
```

```
main() {  
    signal(SIGINT, on_int);  
    while (...) {  
        if (setjmp(env) > 0)  
            puts("Again!");  
        cmd = read_cmd();  
        execute(cmd);  
    }  
}
```

```
void execute(char *cmd) {  
    decode(cmd);  
    expand(cmd);  
    run(cmd);  
}
```

```
void decode(char *cmd) {  
    if (bad(cmd))  
        longjmp(env, 1);  
}
```

Points de reprise Posix

`sigsetjmp()` et `siglongjmp()`

- ✓ Les fonctions `setjmp()` et `longjmp()` ne sauvegardent pas (et donc ne restaurent pas) le masque des signaux bloqués
- ✓ Posix a donc introduit `sigsetjmp()` et `siglongjmp()`
 - utilisation identique (`sigsetjmp` a 2 arguments)
 - à préférer aux versions Ansi C
- ✓ Dans les deux cas , les variables locales sont dans un état indéfini après un `longjmp()`
 - à moins d'avoir été déclarées `volatile`

Variable volatile ?

✓ Volatile

- Qualifier un type particulier
 - Prévient le compilateur que cette variable peut être modifiée de manière extérieure au flot normal du programme
 - `longjmp()` **ou** `siglongjmp()`
 - Entrée/Sortie
 - Threads
 - Interruption ou signal
 - ...
 - Aucune optimisation du compilateur ne doit être appliquée à cette variable
 - Relue en mémoire à chaque accès
 - Pas mise dans un registre temporaire
 - Ralentit le code, mais le protège
- ## ✓ Existe dans les langages modernes
- C, C++, C# et Java



Bilan d'étape

... Avant la suite ... et fin

Synthèse de ce qui a été vu jusqu'à présent

- ✓ **Nous avons vu:**
 - Exécution des programmes
 - Debugging de programmes
 - Et chargement dynamique de bibliothèques
 - Gestion des Processus
 - Pour aller sur une première abstraction qui est le processus
 - Puis la notion de processus léger (thread)
 - Gestion des entrées-sorties
 - Accès aux fichiers
 - Redirection
 - Tubes anonymes et nommés
 - Puis les signaux
 - Moyen de communiquer entre les processus
- ✓ **Pour avoir une vue complète des fonctionnalités d'un système d'exploitation, il nous manque...**
 - La Gestion Mémoire (objet du prochain cours)

InterProcess Communication (IPC)

- ✓ **Méthodes de communication Inter-Processus**
 - Fichier (beuk!... pas efficace)
 - Signal (communication asynchrone)
 - Tube (communication unidirectionnelle)
 - Tubes nommés (différent d'un système à l'autre)
 - Socket (communication entre machines distantes)

- ✓ **Mais aussi (que nous ne verrons pas dans ce cours)**
 - Message Queue
 - Message Passing
 - Mémoire partagée
 - Fichier en mémoire