SIGNAUX POSIX

2016 – 2017

Florent Gluck – Florent.Gluck@hesge.ch

Version 0.61

h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Hes·so GENÈVE
Haute Ecole Spécialisée
de Suisse occidentale

Qu'est-ce qu'un signal ?

- Un signal est **une interruption logicielle** délivrée à un processus.
- Le système d'exploitation utilise les signaux pour signaler des situations exceptionnelles à un programme en exécution ; typiquement :
  - accès mémoire invalide ;
  - terminaison de processus ;
  - pause de processus ;
  - etc.
- **Un signal notifie le processus d'un événement :**
  - **Le processus est interrompu et exécute la routine associée au signal.**
- Forme primitive de communication inter-processus (IPC).
- Mécanisme annulation threads implémenté avec signaux (GNU Linux).
- Utilisés par les shells pour le contrôle de jobs.

Signaux et synchronicité

- Les signaux peuvent être générés de manière **synchrone** ou **asynchrone**.
- Un **signal synchrone** correspond à une action spécifique est délivré durant cette action.
  - La plupart des erreurs génèrent des signaux de manière synchrones.
- Un **signal asynchrone** est généré par un événement externe (en dehors du processus le recevant).
  - Ces signaux arrivent de manière non prévisible durant l'exécution.
  - Typiquement, les événements externes génèrent des signaux de manière asynchrone.
  - De même, un signal envoyé d'un processus à un autre.

Standardisation POSIX

- L'implémentation des signaux est ancienne dans l'histoire d'UNIX → une des premières méthodes d'IPC.
- Historiquement différences de comportements entre systèmes UNIX → incompatibilités parfois subtiles.
- Les signaux ont évolués en fiabilité et fonctionnalités au cours du temps.
- POSIX.1 spécifie la gestion des signaux en définissant une interface et un comportement à respecter.
- Présentons ici les signaux selon POSIX.1-2008.
- L'ancienne API des signaux UNIX est **obsolète** et **à éviter** (fonctions C `signal`, `sigmask`, `sigsetmask`, etc.).

Noms et identification des signaux

- NSIG signaux existent, identifiés par un numéro de 1 à NSIG et un nom symbolique commençant par le préfix SIG
- Liste affichable avec la commande `kill -l`
- Liste des signaux dans `/usr/include/x86_64-linux-gnu/bits/signum.h`
- L'API des signaux est déclarée dans le header `signal.h`

Envoyer un signal depuis un shell

- La commande `kill`* permet d'envoyer un signal à un processus :

```
> kill -STOP 3180   # envoie SIGSTOP au processus 3180
```

  Si aucun signal n'est spécifié, le signal envoyé par défaut est `SIGTERM`.

- La commande `killall` permet d'envoyer un signal en spécifiant un nom de commande plutôt qu'un *pid*; le signal est envoyé à toutes les commandes portant le même nom que celui spécifié.

- La commande `pkill` est similaire à `killall`, sauf qu'il ne s'agit pas d'un match exact du nom spécifié, mais d'un match partiel → **attention**, surtout en tant que *root* !

* Historiquement, `kill` permettait uniquement de tuer un processus, d'où le nom « kill ».

Shell et signaux

- Lorsqu'une application est lancée dans un terminal, bash permet d'envoyer certains signaux à l'aide des touches de contrôle suivantes :
  - CTRL-C : termine le processus en envoyant le signal SIGINT
  - CTRL-Z : stoppe le processus en envoyant le signal SIGTSTP
  - CTRL-\ : termine le processus (*core dump*) en envoyant le signal SIGQUIT

- **Attention** : SIGTSTP signifie « TTY stop » et n'est pas le même signal que SIGSTOP ; l'action de ce dernier ne peut être changée et le signal ne peut être ignoré (tout comme SIGKILL).
- Le processus stoppé par SIGTSTP peut être continué avec le signal SIGCONT.

Source des signaux

- Un signal peut-être généré par :
  - Le kernel → ex. processus accédant à une zone mémoire non autorisée (SIGSEGV), division par zéro (SIGFPE), etc.
  - Un événement externe → utilisateur génère un caractère de contrôle dans un terminal ; ex. : CTRL-C (SIGINT).
  - Un autre processus → appel système `kill` (envoie un signal à un processus ou un groupe de processus).
  - Processus lui-même → fonction `abort`, appels systèmes `alarm`, `kill`, etc.

Action d'un signal

- **A tout signal est associé une action** qui détermine le comportement du processus à la réception du signal.

- Par défaut, chaque signal est associé à une des actions suivantes :

Action	Description
Term	Termine le processus
Ign	Ignore le signal
Core	Termine le processus et créé un fichier "core dump"
Stop	Stop le processus
Cont	Continue le processus (si stoppé)

Comment changer l'action d'un signal ?

- Un processus peut changer l'action d'un signal avec la fonction C `sigaction` afin de définir le comportement à adopter au moment de la réception du signal :
  - Exécuter son action par défaut (Term, Ign, Core, Stop, Cont).
  - Ignorer le signal.
  - Exécuter une fonction personnelle, appelée **handler** de signal.
- Les actions des signaux SIGKILL et SIGSTOP **ne peuvent pas** être changées.
  - Pourquoi ?

Comment changer l'action d'un signal ?

- Un processus peut changer l'action d'un signal avec la fonction C `sigaction` afin de définir le comportement à adopter au moment de la réception du signal :
  - Exécuter son action par défaut (Term, Ign, Core, Stop, Cont).
  - Ignorer le signal.
  - Exécuter une fonction personnelle, appelée **handler** de signal.
- Les actions des signaux SIGKILL et SIGSTOP **ne peuvent pas** être changées.
  - Pourquoi ? Car l'administrateur système doit pouvoir tuer ou stopper un processus pouvant compromettre la stabilité du système.
- Dans le cadre d'une application multi-threadée, l'action pour un signal donné **est la même pour tous les threads !**

Principaux signaux et actions associées

Nom du signal	Evénement associé	Action par défaut
SIGHUP	Terminaison du processus leader de session	Term
SIGINT	Frappe du caractère CTRL-C sur terminal de contrôle	Term
SIGQUIT	Frappe du caractère CTRL-\ sur le terminal de contrôle	Core
SIGILL	Détection d'une instruction illégale	Core
SIGABRT	Terminaison provoquée en exécutant abort	Term
SIGFPE	Erreur arithmétique (division par zéro, ...)	Term
SIGKILL*	Terminaison forcée immédiate	Term
SIGSEGV	Violation mémoire	Core
SIGPIPE	Ecriture dans un pipe sans lecteur	Term
SIGALRM	Fin de temporisation (fonction alarm)	Term
SIGTERM	Terminaison	Term
SIGUSR1	Signal émis par un processus utilisateur	Term
SIGUSR2	Signal émis par un processus utilisateur	Term
SIGCHLD	Terminaison d'un fils	Ign
SIGSTOP*	Suspension du processus	Stop
SIGTSTP	Frappe du caractère CTRL-Z sur le terminal de contrôle	Stop
SIGCONT	Continuation du processus (si stoppé)	Cont

Handler de signal

Lorsque le handler d'un signal est appelé suite à la réception du signal, le **processus/thread courant est interrompu** et le CPU exécute le code du handler.

Le code d'un handler de signal devrait respecter les critères suivants :

- Code concis et exécution rapide, afin de réduire le temps pendant lequel le processus ou thread courant est bloqué :
  - Typiquement, mise à jour de variables d'état et/ou terminer l'application.

- Le code doit uniquement appeler des fonctions réentrantes, aussi appelées "async-signal safe" (cf. slide suivante).
  - Pourquoi ?

Handler de signal

Lorsque le handler d'un signal est appelé suite à la réception du signal, le **processus/thread courant est interrompu** et le CPU exécute le code du handler.

Le code d'un handler de signal devrait respecter les critères suivants :

- Code concis et exécution rapide, afin de réduire le temps pendant lequel le processus ou thread courant est bloqué :
  - Typiquement, mise à jour de variables d'état et/ou terminer l'application.

- Le code doit uniquement appeler des fonctions réentrantes, aussi appelées "async-signal safe" (cf. slide suivante).
  - Pourquoi ?

    Car un signal peut potentiellement être déclenché alors que l'exécution de son handler n'est pas encore terminée !

Fonctions async-signal safe

POSIX.1-2008 défini les fonctions suivantes comme **async-signal safe** (cf. `man 7 signal`):

_Exit()	faccessat()	linkat()	select()	socketpair()
_exit()	fchmod()	listen()	sem_post()	stat()
abort()	fchmodat()	lseek()	send()	symlink()
accept()	fchown()	lstat()	sendmsg()	symlinkat()
access()	fchownat()	mkdir()	sendto()	tcdrain()
aio_error()	fcntl()	mkdirat()	setgid()	tcflow()
aio_return()	fdatasync()	mkfifo()	setpgid()	tcflush()
aio_suspend()	fexecve()	mkfifoat()	setsid()	tcgetattr()
alarm()	fork()	mknod()	setsockopt()	tcgetpgrp()
bind()	fstat()	mknodat()	setuid()	tcsendbreak()
cfgetispeed()	fstatat()	open()	shutdown()	tcsetattr()
cfgetospeed()	fsync()	openat()	sigaction()	tcsetpgrp()
cfsetispeed()	ftruncate()	pause()	sigaddset()	time()
cfsetospeed()	futimens()	pipe()	sigdelset()	timer_getoverrun()
chdir()	getegid()	poll()	sigemptyset()	timer_gettime()
chmod()	geteuid()	posix_trace_event()	sigfillset()	timer_settime()
chown()	getgid()	pselect()	sigismember()	times()
clock_gettime()	getgroups()	raise()	signal()	umask()
close()	getpeername()	read()	sigpause()	uname()
connect()	getpgrp()	readlink()	sigpending()	unlink()
creat()	getpid()	readlinkat()	sigprocmask()	unlinkat()
dup()	getppid()	recv()	sigqueue()	utime()
dup2()	getsockname()	recvfrom()	sigset()	utimensat()
execl()	getsockopt()	recvmsg()	sigsuspend()	utimes()
execle()	getuid()	rename()	sleep()	wait()
execv()	kill()	renameat()	sockatmark()	waitpid()
execve()	link()	rmdir()	socket()	write()

Asynchronisme et atomicité

- Les signaux sont de nature **asynchrone** → lorsque la routine d'un signal est exécutée, l'exécution du processus/thread en cours est interrompue et la routine du signal est exécutée.
  - L'interruption par un signal peut se produire n'importe où dans le code !

- A la fin de l'exécution du handler de signal, le processus ou thread en cours reprend là où il avait été interrompu, tout comme lors d'un changement de contexte.

- Tout handler de signal manipulant des variables globales devrait les déclarer `volatile sig_atomic_t` → accès atomique garantit en lecture/écriture.

Etats d'un signal

- Un signal peut être **Ignoré** ou **bloqué** :
  - **Ignoré** : le signal est perdu.
  - **Bloqué** : le signal n'est pas reçu → **il sera seulement reçu lorsqu'il sera débloqué.**

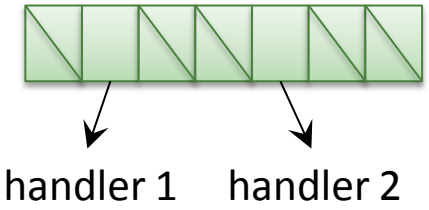
- Un signal **pendant** (*pending*) est un signal envoyé à un processus (ou thread) mais pas encore reçu ; le bit "pendant" est mis à 1 ; **si un signal du même type arrive alors qu'il en existe déjà un de pendant, le dernier est perdu !**

- Un signal est **reçu** (*delivered*) lorsque le processus le prend en compte (le traite) ; le bit "pendant" passe alors à 0.

- Un signal est dit **bloqué** (*blocked*) (ou masqué) lorsqu'il est mis en attente pour être traité ultérieurement.

Signaux, processus et threads

- Chaque processus possède une table de pointeurs où chaque entrée pointe sur un handler de signal :



- Chaque thread (y compris le thread `main`) possède :
  - Un tableau de bits indiquant **les signaux *pendants :***

  - Un tableau de bits indiquant **les signaux *bloqués :***

Détermination des signaux pendants

La fonction `sigpending` permet d'examiner quels signaux bloqués seront déclenchés :

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

- Les signaux masqués qui seront déclenchés lors d'un démasquage sont retournés dans l'ensemble **set**;
- Renvoie 0 en cas de succès.

Signaux et héritage

Que se passe-t-il après un appel à `fork`?

- Les handlers définis dans le père sont hérités par le processus fils.
- Les signaux bloqués dans le père sont hérités chez le fils.
- Chez le fils, la table des signaux pendants est initialisée à zéro.
- A la terminaison d'un processus, le signal SIGCHLD est toujours envoyé au processus père.

Que se passe-t-il après un appel à `pthread_create`?

- Les signaux bloqués dans le père sont hérités par le thread fils.
- Chez le fils, la table des signaux pendants est initialisée à zéro.

Dans le cas d'un appel à une fonction `exec`, la table de pointeurs de signaux est ré-initialisée aux handlers par défaut.

Envoyer un signal à un processus

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

- Envoie le signal **sig** à un processus ou groupe de processus **pid**
- Signification du paramètre **pid** :

> 0	le processus dénoté par l'ID **pid**
0	tous les processus dans le même groupe que le processus
-1	Tous les processus pour lesquels le processus peut envoyer des signaux
< -1	tous les processus du groupe de processus dénoté par le groupe ID |**pid**|

- **sig** est le numéro du signal entre 1 et NSIG
- Renvoie 0 si succès et -1 en cas d'échec
- Un processus peut seulement envoyer un signal à un autre processus s'ils possèdent le même propriétaire !

Envoyer un signal à un thread

```
int pthread_kill(pthread_t thread, int sig);
```

- Similair à `kill` mais pour les threads.
- Envoie le signal **sig** à **thread** du processus courant
- **sig** est le numéro du signal entre 1 et NSIG
- Renvoie 0 en cas de succès
- Nécessite de spécifier la librairie pthread à l'édition des liens (option `-lpthread` passée à gcc)

Envoi au thread ou processus appelant

- La fonction `raise` permet d'envoyer un signal au processus ou thread appelant :

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

- Dans le cas d'un programme séquentiel, équivalent à :

```
kill(getpid(), sig);
```

- Dans le cas d'un programme multi-threadé, équivalent à :

```
pthread_kill(pthread_self(), sig);
```

Réception des signaux et threads

- Si un signal est envoyé (avec la commande `kill` ou la fonction C du même nom) à une application multi-threadée, celui-ci sera **seulement** reçu par **un thread** n'ayant pas bloqué le signal.

- Si plusieurs threads sont eligibles, alors le kernel en choisira **arbitrairement un** comme destinataire (non déterminisme) !

Masque de signaux

- Un masque de signal empêche la réception d'un ou plusieurs signaux.

- Il existe deux types de masques de signaux :

  1. Les signaux masqués durant l'exécution du processus/thread :

     - Fonctions `sigprocmask` et `pthread_sigmask`

  2. Les signaux masqués **temporairement**, uniquement pendant l'exécution de la routine de handler :

     - Champ **`sa_mask`** de la structure `struct sigaction` passée à la fonction `sigaction`

Manipulation du masque

- Le masque de signaux est un ensemble de bits **indiquant si un signal est bloqué ou pas** (bit à 1 ou 0).
- Le masque se manipule avec les fonctions :
  - `int` **`sigemptyset`**`(sigset_t *)` : créé un masque où aucun des signaux n'est bloqué.
  - `int` **`sigfillset`**`(sigset_t *)` : créé un masque où tous les signaux sont bloqués.
  - `int` **`sigaddset`**`(sigset_t *, int signum)` : ajoute le signal spécifié au masque.
  - `int` **`sigdelset`**`(sigset_t *, int signum)` : supprime le signal spécifié du masque.
  - `int` **`sigismember`**`(sigset_t *, int signum)` : teste l'état du signal spécifié ; renvoie 1 si `signum` fait partie du masque.

Changer l'action d'un signal

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

- Le signal est spécifié par **signum** (n'importe quel signal, sauf SIGKILL et SIGSTOP qui sont ignorés).
- La nouvelle action est définie dans **act** (voir slide suivante), sauf si **act** est `null`.
- L'ancienne action est sauvée dans **oldact**, sauf si **oldact** est `null`.
- Renvoie 0 en cas de succès et -1 en cas d'erreur.
- Remarques :
  - Pour récupérer le handler d'un signal, le 2ème argument est mis à `null`.
  - Pour savoir si un signal est valide pour l'architecture utilisée, `sigaction` peut-être appelé avec les 2 derniers arguments à `null`.

Structure sigaction

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

- **sa_handler spécifie l'action pour le signal signum :**
  - SIG_DFL → action par défaut
  - SIG_IGN → ignorer le signal
  - Pointeur sur la fonction de handler (reçoit le n° de signal comme arg.)
- **sa_flags** → permet de modifier le comportement du signal (cf. slide 27)
- **sa_mask** → signaux à bloquer lors de l'exécution du handler ; **le signal ayant déclenché le handler est toujours bloqué**, sauf si SA_NODEFER est spécifié dans **sa_flags**.

Exemple

Mise en place d'un handler pour le signal SIGINT (CTRL-C) :

```
static void handler(int signum) {
        char msg[] = "SIGINT received! Aborted.\n";
        write(STDOUT_FILENO, msg, strlen(msg));
        _exit(EXIT_SUCCESS);
}

int main() {
        struct sigaction act;
        act.sa_handler = handler;     // new handler
        act.sa_flags = 0;             // no flags
        sigemptyset(&act.sa_mask);    // all signals unblocked
        if (sigaction(SIGINT, &act, NULL) == -1) {
                perror("sigaction");
                exit(EXIT_FAILURE);
        }
        int count = 0;
        while(1) {
                printf("Line %d\n", ++count);
                sleep(1);
        }
        return EXIT_SUCCESS;
}
```

sigaction : champs sa_flags

Le champs **sa_flags** de la structure `sigaction` permet de spécifier le comportement du handler :

- **SA_NODEFER** permet au signal d'être reçu lorsqu'il est dans son propre handler (handler réentrant) ; si cette option n'est pas spécifiée, le handler est bloqué en cas d'appel réentrant.
- **SA_RESTART** tente de redémarrer automatiquement un appel système ou un appel de librairie interrompu par un signal ; deux possibilités dépendantes de l'implémentation (cf. `man 7 signal`) :
  1. Appel redémarré automatiquement dès le handler de signal terminé.
  2. Appel échoué avec le code d'erreur EINTR.
- **SA_RESETHAND** restaure l'action par défaut au moment de l'entrée dans le handler de signal.
- Plusieurs options peuvent être combinées avec l'opérateur |
- `man sigaction` liste et décrit toutes les options disponibles.

Exemple

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

static void handler(int signum) {}

int main() {
        struct sigaction act;
        act.sa_handler = handler;     // new handler
        act.sa_flags = 0;             // no flags
        sigemptyset(&act.sa_mask);    // all signals unblocked
        if (sigaction(SIGINT, &act, NULL) == -1) {
                perror("sigaction");
                exit(EXIT_FAILURE);
        }

        char name[256];
        printf("Your name: ");
        fgets(name, sizeof(name), stdin);
        printf("Hello %s", name);

        return EXIT_SUCCESS;
}
```

Exemple

Que se passe-t-il si le processus reçoit le signal (SIGINT) pendant l'exécution de la fonction `fgets` ?

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

static void handler(int signum) {}

int main() {
        struct sigaction act;
        act.sa_handler = handler;     // new handler
        act.sa_flags = 0;             // no flags
        sigemptyset(&act.sa_mask);    // all signals unblocked
        if (sigaction(SIGINT, &act, NULL) == -1) {
                perror("sigaction");
                exit(EXIT_FAILURE);
        }

        char name[256];
        printf("Your name: ");
        fgets(name, sizeof(name), stdin);
        printf("Hello %s", name);

        return EXIT_SUCCESS;
}
```

Exemple

Que se passe-t-il si le processus reçoit le signal (SIGINT) pendant l'exécution de la fonction `fgets` ?

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

static void handler(int signum) {}

int main() {
        struct sigaction act;
        act.sa_handler = handler;     // new handler
        act.sa_flags = SA_RESTART;
        sigemptyset(&act.sa_mask);    // all signals unblocked
        if (sigaction(SIGINT, &act, NULL) == -1) {
                perror("sigaction");
                exit(EXIT_FAILURE);
        }

        char name[256];
        printf("Your name: ");
        fgets(name, sizeof(name), stdin);
        printf("Hello %s", name);

        return EXIT_SUCCESS;
}
```

Masque de signaux (processus)

Pour lire ou modifier le masque des signaux bloqués du **processus** appelant :

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

how peut prendre 3 valeurs :

SIG_BLOCK	Le masque de signaux bloqués est l'union du masque courant et celui passé en argument.
SIG_UNBLOCK	Les signaux spécifiés dans le masque sont supprimés du masque courant.
SIG_SETMASK	Le masque spécifié remplace l'ancien masque.

- Si **set** est NULL → masque de signaux inchangé
- L'ancien masque est retourné dans **oldset**
- Renvoie 0 en cas de succès
- Tout processus ou thread enfant **hérite** du masque de signaux de son père.

Masque de signaux (thread)

La fonction `sigprocmask` n'est pas adaptée aux applications multi-threadées. Dans le cas d'un code multi-threadé, la fonction permettant de définir les signaux bloqués pour le thread appelant est :

```
int pthread_sigmask(int how, const sigset_t *set,
                    sigset_t *oldset);
```

- Arguments et utilisation sont identiques à la fonction `sigprocmask`
- Renvoie 0 en cas de succès
- Rappel: un thread enfant **hérite** du masque de signaux de son père.

Exemple : pthread_sigmask

Blocage de tous les signaux, puis déblocage de SIGINT :

```
static void handler(int signum) {
    char msg[] = "SIGINT received! Aborted.\n";
    write(STDOUT_FILENO, msg, strlen(msg));
    _exit(EXIT_SUCCESS);
}

int main() {
    sigset_t mask;
    sigfillset(&mask);
    pthread_sigmask(SIG_SETMASK, &mask, NULL); // block all signals

    struct sigaction act;
    memset(&act, 0, sizeof(act));
    act.sa_handler = handler;
    sigaction(SIGINT, &act, NULL);

    int count = 0;
    while (1) {
        printf("Line %d\n", ++count);
        sleep(1);
        if (count == 4) {  // unblock SIGINT after 4 seconds
            sigdelset(&mask, SIGINT);
            pthread_sigmask(SIG_SETMASK, &mask, NULL);
        }
    }
    return EXIT_SUCCESS;
}
```

Attente sur un signal (1)

Un mécanisme de synchronisation simple est l'attente sur un signal. Trois mécanismes permettent d'attendre **passivement** sur un ou plusieurs signaux :

1) `pause` : suspend l'exécution du processus/thread courant jusqu'à ce que n'importe quel signal soit reçu. Le handler de signal est exécuté.

2) `sigsuspend` : change temporairement le masque de signaux, puis suspend l'exécution du processus/thread courant jusqu'à ce qu'un signal non bloqué soit reçu. Le handler de signal est exécuté.

3) `sigwait` : suspend l'exécution du processus/thread courant jusqu'à ce qu'un ou plusieurs signaux spécifiés en argument soient reçus. Le handler de signal **n'est pas exécuté**.

Attente sur un signal avec pause

```
int pause(void);
```

- `pause()` se termine seulement lorsque l'exécution du handler de signal est terminée.
- Renvoie toujours -1 mais en cas d'erreur `errno` contient le numéro d'erreur.

Exemple : attente avec pause

Quel est le danger dans le code suivant ?

```
// done is set to 1 in SIGUSR1's handler
static volatile sig_atomic_t done = 0;

...

// execution is suspended until SIGUSR1 is received
if (!done)
      pause();

// resuming execution (once the signal is received)
...
```

Exemple : attente avec pause

Quel est le danger dans le code suivant ?

```
// done is set to 1 in SIGUSR1's handler
static volatile sig_atomic_t done = 0;

...

// execution is suspended until SIGUSR1 is received
if (!done)
        pause();

// resuming execution (once the signal is received)
...
```

Problème : opération non atomique !

Le signal peut arriver **après** que la variable done soit testée, mais avant l'appel à `pause`. Si aucun autre signal n'arrive, le processus ne sera jamais réveillé → deadlock !

Attente sur un signal

- Evitez la fonction `pause`, sauf dans les situations triviales.
- Une manière plus fiable permettant d'éviter les problèmes de concurrence est de bloquer tous les signaux, puis d'attendre seulement sur ceux qui nous intéressent.
  - C'est le but de la fonction `sigsuspend` qui est utilisée conjointement avec un masquage approprié des signaux.
  - Contrairement à `pause`, `sigsuspend` permet d'attendre seulement sur des signaux spécifiques.

Attente sur un signal avec sigsuspend

```
int sigsuspend(sigset_t *mask);
```

- Remplace **temporairement** le masque de signaux bloqués par **mask** et suspend le processus jusqu'à réception d'un signal non bloqué (*i.e.* absent du masque), le tout de **manière atomique**.
- `sigsuspend()` se termine seulement lorsque l'exécution du handler de signal est terminée.
- Le masque spécifié reste en effet tant que `sigsuspend` est en attente. A la sortie de `sigsuspend`, le masque précédent est rétabli.
- Renvoie toujours -1 mais en cas d'erreur `errno` contient le numéro d'erreur.

Exemple : attente avec sigsuspend

```
// done is set to 1 in SIGUSR1's handler
static volatile sig_atomic_t done = 0;

...

sigset_t maskall, maskmost, maskold;
// create a mask for all signals
sigfillset(&maskall);
// create a mask for all signals except SIGUSR1
sigfillset(&maskmost);
sigdelset(&maskmost, SIGUSR1);
// block all signals
pthread_sigmask(SIG_SETMASK, &maskall, &maskold);

// suspend the thread until SIGUSR1 is received
if (!done)
      sigsuspend(&maskmost);

// restore the previous mask
pthread_sigmask(SIG_SETMASK, &maskold, NULL);

...
```

Attente sur un signal avec sigwait

Suspend le thread appelant jusqu'à ce qu'un des signaux de **set** deviennent pendant. Le signal reçu est ensuite écrit dans **sig**.

```
int sigwait(const sigset_t *set, int *sig);
```

- La fonction enlève ensuite le bit pendant du signal reçu.
- La fonction **n'exécute pas** le handler du signal reçu, **contrairement à `pause` et `sigsuspend` !**
- Le(s) signal(aux) sur lequel attendre **doit être péalablement bloqué.**
- Ne modifie pas le masque du processus/thread courant, contrairement à `siguspsend`, qui elle le remplace temporairement.
- Renvoie 0 en cas de succès.

Exemple 1: sigwait

Le code ci-dessous bloque SIGINT et SIGUSR1, puis attend sur ceux-ci. Si SIGINT est reçu, un message est affiché. Le programme se termine dès réception de SIGUSR1.

```
int main() {
    sigset_t mask, maskold;
    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGUSR1);
    pthread_sigmask(SIG_SETMASK, &mask, &maskold);
    puts("SIGINT and SIGUSR1 blocked.");
    puts("Use SIGINT to print a message and SIGUSR1 to quit.");
    int sig;
    do {
        sigwait(&mask, &sig);
        if (sig == SIGINT)
            puts("Be strong, I whispered to my wifi signal!");
    } while (sig != SIGUSR1);
    puts("Program terminated.");
    return EXIT_SUCCESS;
}
```

Exemple 1: sigwait

Le code ci-dessous bloque SIGINT et SIGUSR1, puis attend sur ceux-ci. Si SIGINT est reçu, un message est affiché. Le programme se termine dès réception de SIGUSR1.

```
int main() {
    sigset_t mask, maskold;
    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGUSR1);
    pthread_sigmask(SIG_SETMASK, &mask, &maskold);
    puts("SIGINT and SIGUSR1 blocked.");
    puts("Use SIGINT to print a message and SIGUSR1 to quit.");
    int sig;
    do {
        sigwait(&mask, &sig);
        if (sig == SIGINT)
            puts("Be strong, I whispered to my wifi signal!");
    } while (sig != SIGUSR1);
    puts("Program terminated.");
    return EXIT_SUCCESS;
}
```

Que se passe-t-il si SIGUSR2 est reçu ?

Exemple 2: sigwait

Le code ci-dessous bloque tous les signaux puis attend sur SIGINT et SIGUSR1. Si SIGINT est reçu, un message est affiché. Le programme se termine dès réception de SIGUSR1.

```
int main() {
    sigset_t mask, maskold;
    sigfillset(&mask);
    pthread_sigmask(SIG_SETMASK, &mask, &maskold);
    puts("All signals blocked.");
    puts("Use SIGINT to print a message and SIGUSR1 to quit.");
    int sig;
    do {
        sigwait(&mask, &sig);
        if (sig == SIGINT)
            puts("Be strong, I whispered to my wifi signal!");
    } while (sig != SIGUSR1);
    puts("Program terminated.");
    return EXIT_SUCCESS;
}
```

Exemple 2: sigwait

Le code ci-dessous bloque tous les signaux puis attend sur SIGINT et SIGUSR1. Si SIGINT est reçu, un message est affiché. Le programme se termine dès réception de SIGUSR1.

Que se passe-t-il si SIGUSR2 est reçu ?

```
int main() {
    sigset_t mask, maskold;
    sigfillset(&mask);
    pthread_sigmask(SIG_SETMASK, &mask, &maskold);
    puts("All signals blocked.");
    puts("Use SIGINT to print a message and SIGUSR1 to quit.");
    int sig;
    do {
        sigwait(&mask, &sig);
        if (sig == SIGINT)
            puts("Be strong, I whispered to my wifi signal!");
    } while (sig != SIGUSR1);
    puts("Program terminated.");
    return EXIT_SUCCESS;
}
```

Alarm

L'appel système `alarm` permet d'envoyer le signal SIGALRM au processus appelant après un nombre de **secondes** donné :

```
unsigned int alarm(unsigned int seconds);
```

- Renvoie le nombre de secondes restantes jusqu'au déclenchement d'une alarme spécifiée précédemment (ou 0 si aucune).
- Tout appel précédent à `alarm` est annulé.
- `alarm(0)` annule l'alarme courante.
- Si davantage de précision et/ou de souplesse est nécessaire, préférer la fonction `setitimer` .
- **Attention** : les fonctions `setitimer` et `alarm` partagent le même timer, donc risques d'intérférences.

Threads et signaux

- Tout comme pour les applications séquentielles, les applications multi-threadées devraient correctement gérer les signaux : au minimum une terminaison *propre* lors de la réception de SIGINT.

- Dans tout code multi-threadé, il est très fortement recommandé de **réserver un thread pour la gestion des signaux** et de les gérer avec **`sigwait`**.

- Sauf besoins spécifiques, éviter d'utiliser les signaux pour réaliser de la synchronisation, préférez l'utilisation de sémaphores ou variables de condition.

Ressources

Signal handling (GNU C library manual)

- http://www.gnu.org/software/libc/manual/html_node/Signal-Handling.html#Signal-Handling

All about Linux signals

- http://www.linuxprogrammingblog.com/all-about-linux-signals?page=show

Atomic types

- http://www.gnu.org/software/libc/manual/html_node/Atomic-Types.html

Use reentrant functions for safer signal handling

- http://www.ibm.com/developerworks/linux/library/l-reent/index.html