Fonction **signal()**

En C, la fonction signal() permet de définir un **gestionnaire de signal**, c'est-à-dire une fonction qui sera exécutée lorsqu'un signal spécifique est reçu.

void (\*signal(int signum, void (\*handler)(int)))(int);

**Paramètres :**

* signum : le numéro du signal à capturer (ex : SIGINT, SIGTERM).
* handler : un pointeur vers la fonction qui gère le signal, ou des valeurs spéciales :
  + SIG\_IGN → pour ignorer le signal.
  + SIG\_DFL → pour restaurer le comportement par défaut du signal.

Les termes comme **SIGINT**, **SIGCHLD**, etc., sont des **signaux** utilisés par les systèmes d'exploitation de type UNIX/Linux (et aussi dans Windows avec des adaptations). Ces signaux sont des notifications envoyées à un processus pour l'avertir qu'un événement particulier est survenu.

Chaque signal est identifié par un **nom symbolique** (ex. SIGINT) et un **numéro de signal** (ex. 2 pour SIGINT), définis dans l'en-tête **<signal.h>** en C

**Liste des signaux les plus courants et leur signification :**

| **Signal** | **Numéro** | **Signification** | **Action par défaut** |
| --- | --- | --- | --- |
| **SIGINT** | 2 | Interruption (Ctrl + C) | Termine le processus |
| **SIGTERM** | 15 | Demande de terminaison propre | Termine le processus |
| **SIGKILL** | 9 | Arrêt immédiat (forcé, impossible d'ignorer) | Termine le processus |
| **SIGSTOP** | 19 | Pause du processus (Ctrl + Z) | Suspend le processus |
| **SIGCONT** | 18 | Reprise d'un processus suspendu | Reprend le processus |
| **SIGCHLD** | 17 | Notifie qu'un processus enfant s'est terminé | Ignore (par défaut) |
| **SIGSEGV** | 11 | Erreur de segmentation (accès mémoire invalide) | Termine avec un core dump |
| **SIGFPE** | 8 | Erreur arithmétique (division par zéro) | Termine avec un core dump |
| **SIGHUP** | 1 | Déconnexion du terminal (fermeture de session) | Termine le processus |
| **SIGUSR1** | 10 | Signal utilisateur défini | Personnalisable |
| **SIGUSR2** | 12 | Autre signal utilisateur défini | Personnalisable |
| **SIGALRM** | 14 | Signal d'alarme généré par alarm() | Termine le processus |

Fonction **sigemptyset()**

utilisée en C pour **initialiser un ensemble de signaux** à l'état **vide**, c'est-à-dire qu'aucun signal n'est inclus dans l'ensemble après l'initialisation.

**Prototype de sigemptyset()**

int sigemptyset(sigset\_t \*set);

**Paramètre :**

* set : Un pointeur vers une variable de type sigset\_t qui représente un ensemble de signaux.

**Valeur de retour :**

* Retourne **0** en cas de succès.
* Retourne **-1** en cas d'échec, et errno est défini pour indiquer l'erreur.

**Fonctionnement :**

La fonction **sigemptyset()** sert à s'assurer que l'ensemble de signaux est bien vide avant d'ajouter des signaux spécifiques avec d'autres fonctions comme sigaddset().

L'utilisation typique de sigemptyset() se fait avant de configurer des masques de signaux ou d'installer des gestionnaires de signaux.

**Gestion des signaux avec sigaction():**

* Avant d'installer un gestionnaire, on vide l'ensemble des signaux à bloquer pendant l'exécution du gestionnaire.

La fonction **sigaddset()**

La fonction **sigaddset()** est utilisée pour **ajouter un signal spécifique** à un ensemble de signaux (de type sigset\_t). Elle est souvent utilisée après avoir initialisé un ensemble avec sigemptyset() ou sigfillset().

**Prototype de sigaddset()**

int sigaddset(sigset\_t \*set, int signum);

**Paramètres :**

1. set : Pointeur vers un ensemble de signaux de type sigset\_t.
2. signum : Le numéro du signal à ajouter à l'ensemble (ex: SIGINT, SIGTERM, etc.).

**Valeur de retour :**

* Retourne **0** en cas de succès.
* Retourne **-1** en cas d'échec et définit errno avec :
  + EINVAL si le signal donné est invalide.
* Très utile pour :
  + Bloquer/débloquer des signaux (sigprocmask()).
  + Attendre des signaux spécifiques (sigsuspend()).
  + Gérer les signaux dans un handler (sigaction()).

La fonction **sigaction()**

La fonction **sigaction()** est utilisée pour installer un **gestionnaire de signaux** dans un programme C. Elle permet de définir comment un processus doit réagir lorsqu'il reçoit un signal spécifique, en offrant un contrôle plus fin que l'ancienne fonction signal().

**Prototype de sigaction()**

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

**Paramètres :**

1. signum : Le numéro du signal à intercepter (ex: SIGINT, SIGTERM, SIGUSR1).
2. act : Pointeur vers une structure sigaction contenant le nouveau gestionnaire de signal.
3. oldact : Pointeur vers une structure pour stocker l'ancien gestionnaire (peut être NULL si non utilisé).

**Valeur de retour :**

* Retourne **0** en cas de succès.
* Retourne **-1** en cas d'échec et définit errno (par exemple, si le signal est invalide).

**La structure sigaction**

La structure utilisée avec sigaction() est définie comme suit :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

**Champs importants :**

* **sa\_handler** : Pointeur vers la fonction qui gère le signal (ou des valeurs spéciales comme SIG\_IGN ou SIG\_DFL).
* **sa\_mask** : Ensemble de signaux à bloquer pendant l'exécution du gestionnaire (utiliser sigemptyset() et sigaddset()).
* **sa\_flags** : Options de contrôle pour le comportement du signal (ex: SA\_RESTART, SA\_NOCLDWAIT).

**Résumé rapide :**

* sigaction() est utilisé pour gérer les signaux de manière robuste et portable.
* Permet de définir des gestionnaires de signaux avec des options avancées (blocage, redémarrage, etc.).
* Toujours préférer sigaction() à signal() pour un meilleur contrôle.
* Utile dans les applications système, les serveurs et les gestionnaires de processus.

La fonction **kill()**

La fonction **kill()** est utilisée pour **envoyer un signal** à un processus ou à un groupe de processus. Elle permet de notifier un processus en lui envoyant un signal spécifique, qui peut être intercepté et traité à l'aide de gestionnaires de signaux (sigaction()), ou entraîner la terminaison du processus.

**Prototype de kill()**

int kill(pid\_t pid, int sig);

**Paramètres :**

1. **pid** : Identifiant du processus ou groupe de processus cible.
   * Valeurs spéciales possibles :
     + pid > 0 : Envoie le signal au processus avec l'ID spécifié.
     + pid == 0 : Envoie le signal à tous les processus du **même groupe** que l'appelant.
     + pid == -1 : Envoie le signal à **tous les processus accessibles** par l'utilisateur (sauf PID 1 et lui-même).
     + pid < -1 : Envoie le signal à **tous les processus appartenant au groupe** -pid.
2. **sig** : Le signal à envoyer (ex: SIGKILL, SIGTERM, SIGUSR1, 0 pour vérifier si le processus existe).

**Valeur de retour :**

* Retourne **0** en cas de succès.
* Retourne **-1** en cas d'échec et définit errno, les erreurs courantes incluent :
  + ESRCH : Le processus spécifié n'existe pas.
  + EPERM : Permission refusée (le processus appartient à un autre utilisateur).
  + EINVAL : Le numéro de signal est invalide.

**Résumé rapide :**

* kill() est utilisé pour **envoyer des signaux** à des processus.
* Il peut être utilisé pour :
  + Terminer un processus (SIGKILL, SIGTERM).
  + Vérifier si un processus est actif (kill(pid, 0)).
  + Interagir avec des processus (signaux personnalisés).
* Vérifier les erreurs avec perror() en cas d'échec.

La fonction **getpid()**

La fonction **getpid()** est utilisée pour obtenir l'**identifiant du processus actuel (PID)**. Le PID est un numéro unique attribué par le système d'exploitation à chaque processus en cours d'exécution.

**Prototype de getpid()**

pid\_t getpid(void);

**Valeur de retour :**

* Retourne l'**identifiant du processus (PID)** sous la forme d'un entier de type pid\_t (généralement un entier positif).
* Il n'y a **pas d'erreur possible**, la fonction retourne toujours un PID valide.

**Utilisation courante de getpid()**

1. **Afficher le PID du processus pour des tests de signalisation :**
   * Utile pour envoyer un signal avec kill à son propre processus.
2. **Créer des fichiers temporaires uniques (basés sur le PID) :**
   * Exemple : création d'un fichier temporaire unique avec le PID.

**La fonction sleep()**

**Description :**

* sleep() suspend l'exécution du programme pour **un nombre défini de secondes**, puis reprend automatiquement l'exécution après l'expiration de ce délai.
* Si un signal est reçu pendant le sommeil, l'exécution peut être interrompue prématurément.

**Prototype :**

unsigned int sleep(unsigned int seconds);

**Comportement :**

* Attend **un temps spécifique** (par exemple, sleep(5) attendra 5 secondes).
* Retourne avant la fin si un signal interrompt le sommeil.
* Renvoie le **temps restant** s'il est interrompu par un signal, ou 0 si le temps a été écoulé.

**La fonction pause()**

**Description :**

* pause() met le programme en **attente indéfinie** jusqu'à ce qu'un **signal** soit reçu.
* Contrairement à sleep(), il n'a **pas de durée**, et le programme ne reprend qu'après la réception d'un signal.

**Prototype :**

int pause(void);

**Comportement :**

* Attend **indéfiniment** jusqu'à ce qu'un signal soit reçu.
* Retourne toujours -1 et définit errno à EINTR (interruption par signal).
* N'a **pas d'option de délai**, donc si aucun signal n'arrive, le programme reste bloqué indéfiniment.

**Comparaison entre sleep() et pause()**

Une image contenant texte, capture d’écran, Police, reçu

Description générée automatiquement

**Quand utiliser sleep() ou pause() ?**

* **Utiliser sleep() si :**
  + Tu veux une pause pendant un certain temps.
  + L'attente doit être périodique (ex: vérifier un fichier toutes les 5 secondes).
  + Il faut continuer l'exécution après une durée.
* **Utiliser pause() si :**
  + Tu veux attendre un événement signalé par un signal.
  + Le programme ne doit rien faire tant qu'un signal n'est pas reçu.
  + Idéal pour les applications en mode "veille" en attente d'instructions.