NOM

signal - Panorama des signaux.

Linux prend en charge à la fois les signaux POSIX classiques (« signaux standards ») et les signaux POSIX temps-réel.

Dispositions de signaux

Chaque signal a une disposition courante, qui détermine le comportement du processus lorsqu'il reçoit ce signal.

Les symboles de la colonne « Action » indiquent l'action par défaut pour chaque signal, avec la signification suivante :

Term Par défaut, terminer le processus.

Ign Par défaut, ignorer le signal.

Core Par défaut, créer un fichier core et terminer le processus (voir **core**(5)).

Stop Par défaut, arrêter le processus.

Cont Par défaut, continuer le processus s'il est actuellement arrêté.

Un processus peut changer la disposition d'un signal avec **sigaction**(2) ou (de façon moins portable) **signal**(2). Avec ces appels système, un processus peut choisir de se comporter de l'une des façons suivantes lorsqu'il reçoit ce signal : effectuer l'action par défaut, ignorer le signal, ou rattraper le signal avec un gestionnaire de signal, c'est-à-dire une fonction définie par le programme, qui est invoquée automatiquement lorsque le signal est distribué. (Par défaut, le gestionnaire de signaux est appelé sur la pile normale des processus. Il est possible de s'arranger pour que le gestionnaire de signaux utilise une autre pile ; consultez **sigaltstack**(2) pour une discussion sur comment faire ceci et quand ça peut être utile.)

La disposition d'un signal est un attribut du processus : dans une application multithreadée, la disposition d'un signal particulier est la même pour tous les threads.

Un fils créé par **fork**(2) hérite d'une copie des dispositions de signaux de son père. Lors d'un **execve**(2), les dispositions des signaux pris en charge sont remises aux valeurs par défaut ; les dispositions des signaux ignorés ne sont pas modifiées.

Envoyer un signal

Les appels système suivants permettent à l'appelant d'envoyer un signal :

raise(3)	Envoie un signal au thread appelant.
kill(2)	Envoie un signal au processus indiqué, à tous les membres du groupe de processus indiqué, ou à tous les processus du système.
killpg(2)	Envoie un signal à tous les membres du groupe de processus indiqué.
pthread_kill(3)	Envoie un signal au thread POSIX indiqué, dans le même processus que l'appelant.
tgkill(2)	Envoie un signal au thread indiqué, à l'intérieur d'un processus donné. (C'est l'appel système qui était utilisé pour implémenter **pthread_kill**(3))
sigqueue(2)	Envoie un signal temps-réel, avec ses données jointes, au processus indiqué.

Attente de la capture d'un signal

Les appels système suivants suspendent l'exécution du processus ou du thread appelant jusqu'à ce qu'un signal soit attrapé (ou qu'un signal non pris en charge termine le processus) :

pause(2) Suspend l'exécution jusqu'à ce que n'importe quel signal soit reçu.

sigsuspend(2) Change temporairement le masque de signaux (voir ci-dessous) et suspend l'exécution jusqu'à ce qu'un des signaux masqué soit reçu.

Accepter un signal de façon synchrone

Au lieu de rattraper un signal de façon asynchrone avec un gestionnaire de signal, il est possible accepter un signal de façon synchrone, c'est-à-dire de bloquer l'exécution jusqu'à ce qu'un signal soit distribué. À ce moment, le noyau renvoie des informations concernant le signal à l'appelant. Il y a deux façon générale pour faire cela :

* **sigwaitinfo**(2), **sigtimedwait**(2) et **sigwait**(3) suspendent l'exécution jusqu'à ce qu'un des signaux dans l'ensemble indiqué soit distribué. Chacun de ces appels renvoie des informations concernant le signal distribué.

* **signalfd**(2) renvoie un descripteur de fichier qui peut être utilisé pour lire des informations concernant les signaux qui sont distribué à l'appelant. Chaque **read**(2) dans ce descripteur de fichier est bloquant jusqu'à ce que des signaux de l'ensemble fournit à **signalfd**(2) soit distribué à l'appelant. Le tampon renvoyé par **read**(2) contient une structure qui décrit le signal.

Masque de signaux et signaux en attente

Un signal peut être bloqu, ce qui signifie qu'il ne sera pas reçu par le processus avant d'être débloqué. Entre sa création et sa réception, le signal est dit en attente.

Chaque thread d'un processus a un masque de signaux indépendant, qui indique l'ensemble des signaux bloqués par le thread. Un thread peut modifier son masque de signaux avec **pthread_sigmask**(3). Dans une application traditionnelle, à un seul thread, **sigprocmask**(2) peut être utilisée pour modifier le masque de signaux.

Un fils créé avec **fork**(2) hérite d'une copie du masque de signaux de son père ; le masque de signaux est préservé au travers d'un **execve**(2).

Un signal peut être créé (et donc mis en attente) pour un processus dans son ensemble (par exemple avec **kill**(2)), ou pour un thread en particulier (par exemple, certains signaux comme **SIGSEGV** et **SIGFPE** sont générés suite à une instruction particulière en langage machine, et sont dirigés vers un thread, de même que les signaux envoyés avec **pthread_kill**(3)). Un signal envoyé à un processus peut être traité par n'importe lequel des threads qui ne le bloquent pas. Si plus d'un thread ne bloque pas le signal, le noyau choisit l'un de ces threads arbitrairement, et lui envoie le signal.

Un thread peut obtenir l'ensemble des signaux en attente avec **sigpending**(2). Cet ensemble est l'union des signaux en attente envoyés au processus, et de ceux en attente pour le thread appelant.

Un fils créé avec **fork**(2) démarre avec un ensemble de signaux en attente vide ; l'ensemble de signaux en attente est préservé au travers d'un **execve**(2).

Signaux standards

Linux prend en charge les signaux standards indiqués ci-dessous. Plusieurs d'entre eux dépendent de l'architecture, comme on le voit dans la colonne «Valeur ». Lorsque trois valeurs sont indiquées, la première correspond normalement aux architectures Alpha et Sparc, la seconde aux ix86, ia64, ppc, s390, arm et sh, et la dernière aux Mips. Un « - » dénote un signal absent pour l'architecture correspondante.

Voici tout d'abord les signaux décrits dans le standard POSIX.1-1990 original :

Signal	Valeur	Action	Commentaire
SIGHUP	1	Term	Déconnexion détectée sur le terminal de contrôle ou mort du processus de contrôle.
SIGINT	2	Term	Interruption depuis le clavier.
SIGQUIT	3	Core	Demande « Quitter » depuis le clavier.
SIGILL	4	Core	Instruction illégale.
SIGABRT	6	Core	Signal d'arrêt depuis **abort**(3).
SIGFPE	8	Core	Erreur mathématique virgule flottante.
SIGKILL	9	Term	Signal « KILL ».
SIGSEGV	11	Core	Référence mémoire invalide.
SIGPIPE	13	Term	Écriture dans un tube sans lecteur.
SIGALRM	14	Term	Temporisation **alarm**(2) écoulée.
SIGTERM	15	Term	Signal de fin.
SIGUSR1	30,10,16	Term	Signal utilisateur 1.
SIGUSR2	31,12,17	Term	Signal utilisateur 2.
SIGCHLD	20,17,18	Ign	Fils arrêté ou terminé.
SIGCONT	19,18,25	Cont	Continuer si arrêté.
SIGSTOP	17,19,23	Stop	Arrêt du processus.
SIGTSTP	18,20,24	Stop	Stop invoqué depuis tty.
SIGTTIN	21,21,26	Stop	Lecture sur tty en arrière-plan.
SIGTTOU	22,22,27	Stop	Écriture sur tty en arrière-plan.

Les signaux **SIGKILL** et **SIGSTOP** ne peuvent ni capturés ni ignorés.

Ensuite, les signaux non décrits par POSIX.1-1990, mais présents dans les spécifications SUSv2 et POSIX.1-2001 :

Signal	Valeur	Action	Commentaire
SIGBUS	10,7,10	Core	Erreur de bus (mauvais accès mémoire).
SIGPOLL		Term	Événement « pollable » (System V). Synonyme de **SIGIO**.
SIGPROF	27,27,29	Term	Expiration de la temporisation pour le suivi.
SIGSYS	12,-,12	Core	Mauvais argument de fonction (SVr4).
SIGTRAP	5	Core	Point d'arrêt rencontré.
SIGURG	16,23,21	Ign	Condition urgente sur socket (BSD 4.2).
SIGVTALRM	26,26,28	Term	Alarme virtuelle (BSD 4.2).
SIGXCPU	24,24,30	Core	Limite de temps CPU dépassée (BSD 4.2).
SIGXFSZ	25,25,31	Core	Taille de fichier excessive (BSD 4.2).

Jusqu'à Linux 2.2 inclus, l'action par défaut pour **SIGSYS**, **SIGXCPU**, **SIGXFSZ** et (sur les architectures autres que Sparc ou Mips) **SIGBUS** était de terminer simplement le processus, sans fichier core. (Sur certains Unix, l'action par défaut pour **SIGXCPU** et **SIGXFSZ** est de finir le processus sans fichier core). Linux 2.4 se conforme à POSIX.1-2001 pour ces signaux et termine le processus avec un fichier core.

Puis quelques signaux divers :

Signal	Valeur	Action	Commentaire
SIGIOT	6	Core	Arrêt IOT. Un synonyme de **SIGABRT**.
SIGEMT	7,-,7	Term	
SIGSTKFLT	-,16,-	Term	Erreur de pile sur coprocesseur (inutilisé).
SIGIO	23,29,22	Term	E/S à nouveau possible(BSD 4.2).
SIGCLD	-,-,18	Ign	Synonyme de **SIGCHLD**.
SIGPWR	29,30,19	Term	Chute d'alimentation (System V).
SIGINFO	29,-,-		Synonyme de **SIGPWR**.
SIGLOST	-,-,-	Term	Perte de verrou de fichier.
SIGWINCH	28,28,20	Ign	Fenêtre redimensionnée (BSD 4.3, Sun).
SIGUNUSED	-,31,-	Term	Signal inutilisé (sera **SIGSYS**).

(Le signal 29 est **SIGINFO** / **SIGPWR** sur Alpha mais **SIGLOST** sur Sparc).

SIGEMT n'est pas spécifié par POSIX.1-2001 mais apparaît néanmoins sur

la plupart des Unix, avec une action par défaut typique correspondant à une fin du processus avec fichier core.

SIGPWR (non spécifié dans POSIX.1-2001) est typiquement ignoré sur les autres Unix où il apparaît.

SIGIO (non sécifié par POSIX.1-2001) est ignoré par défaut sur plusieurs autres systèmes Unix.

Signaux temps-réel

Linux prend en charge les signaux temps-réel tels qu'ils ont été définis à l'origine dans les extensions temps-réel POSIX.1b (et inclus à présent dans POSIX.1-2001). L'intervalle des signaux temps-réels gérés est défini par les macros **SIGRTMIN** et **SIGRTMAX**. POSIX.1-2001 exige qu'une implémentation gère au moins **_POSIX_RTSIG_MAX** (8) signaux temps-réels.

Le noyau Linux gère une gamme de 32 signaux temps-réel, numérotés de 33 à 64. Cependant, l'implémentation des threads POSIX de la glibc utilise en interne deux (pour l'implémentation NPTL) ou trois (pour l'implémentation LinuxThreads) signaux temps-réel (voir **pthreads**(7)) et ajuste la valeur de **SIGRTMIN** en conséquence (à 34 ou 35). Comme la gamme de signaux temps-réel varie en fonction de l'implémentation des threads par la glibc (et cette implémentation peut changer à l'exécution en fonction du noyau et de la glibc) et que la gamme de signaux temps-réel varie bien sûr également suivant les systèmes Unix, les programmes ne devraient jamais faire rfrence des signaux temps rel en utilisant des numros, mais devraient toujours à la place utiliser des signaux temps-réel avec la notation **SIGRTMIN**+n en vérifiant à l'exécution que **SIGRTMIN**+n ne dépasse pas **SIGRTMAX**.

Contrairement aux signaux standards, les signaux temps-réel n'ont pas de signification prédéfinie : l'ensemble complet de ces signaux peut être utilisée à des fins spécifiques à l'application. (Notez quand même que l'implémentation LinuxThreads utilise les trois premiers signaux temps-réel).

L'action par défaut pour un signal temps-réel non capturé est de terminer le processus récepteur.

Les signaux temps-réel se distinguent de leurs homologues classiques ainsi:

1. Plusieurs instances d'un signal temps-réel peuvent être empilées. Au contraire, si plusieurs instances d'un signal standard arrivent alors qu'il est bloqué, une seule instance sera mémorisée.

2. Si le signal est envoyé en utilisant **sigqueue**(2), il peut être accompagné d'une valeur (un entier ou un pointeur). Si le processus récepteur positionne un gestionnaire en utilisant l'attribut **SA_SIGINFO** de l'appel **sigaction**(2) alors il peut accéder à la valeur transmise dans le champ si_value de la structure siginfo_t passée en second argument au gestionnaire. De plus, les champs si_pid et si_uid de cette structure fournissent le PID et l'UID réel du processus émetteur.

3. Les signaux temps-réel sont délivrés dans un ordre précis. Les divers signaux temps-réel du même type sont délivrés dans l'ordre où ils ont été émis. Si différents signaux temps-réel sont envoyés au processus, ils sont délivrés en commençant par le signal de numéro le moins élevé (le signal de plus fort numéro est celui de priorité la plus faible). Par contre, si plusieurs signaux standards sont en attente dans un processus, l'ordre dans lequel ils sont délivrés n'est pas défini.

Si des signaux standards et des signaux temps-réel sont simultanément en attente pour un processus, Posix ne précise pas d'ordre de délivrance. Linux, comme beaucoup d'autres implémentations, donne priorité aux signaux temps-réel dans ce cas.

D'après POSIX, une implémentation doit permettre l'empilement d'au moins **_POSIX_SIGQUEUE_MAX** (32) signaux temps-réel pour un processus.

Néanmoins, Linux fonctionne différemment. Jusqu'au noyau 2.6.7 inclus, Linux impose une limite pour l'ensemble des signaux empilés sur le système pour tous les processus. Cette limite peut être consultée, et modifiée (avec les privilèges adéquats) grâce au fichier /proc/sys/kernel/rtsig-max. Un fichier associé, /proc/sys/kernel/rtsig-nr, indique combien de signaux temps-réel sont actuellement empilés. Dans Linux 2.6.8, ces interfaces /proc ont été remplacées par la limite de ressources **RLIMIT_SIGPENDING**, qui spécifie une limite par utilisateur pour les signaux empilés ; voir **setrlimit**(2) pour plus de détails.

Fonctions pour signaux sûr asynchrones

Une fonction configurée par **sigaction**(2) ou **signal**(2) pour la gestion d'un signal doit prendre beaucoup de précautions, puisqu'elle peut interrompre à n'importe quel endroit l'exécution du programme. POSIX possède la notion de « fonctions sûres ». Si un signal interrompt l'exécution d'une fonction non sûre, et que le gestionnaire appelle une fonction non sûre, alors le comportement du programme n'est pas défini.

POSIX.1-2004 (également appelée « POSIX.1-2001 Technical Corrigendum 2 ») impose qu'une implémentation garantisse que les fonctions suivantes puissent être appelée sans risque à l'intérieur d'un gestionnaire de signal :

```
_Exit()
_exit()
abort()
accept()
access()
aio_error()
aio_return()
aio_suspend()
alarm()
bind()
cfgetispeed()
cfgetospeed()
cfsetispeed()
cfsetospeed()
chdir()
chmod()
chown()
clock_gettime()
close()
connect()
creat()
dup()
dup2()
execle()
execve()
fchmod()
fchown()
fcntl()
fdatasync()
fork()
fpathconf()
fstat()
fsync()
ftruncate()
getegid()
geteuid()
getgid()
getgroups()
getpeername()
getpgrp()
getpid()
getppid()
getsockname()
getsockopt()
getuid()
kill()
link()
listen()
```

```
       lseek()
       lstat()
       mkdir()
       mkfifo()
       open()
       pathconf()
       pause()
       pipe()
       poll()
       posix_trace_event()
       pselect()
       raise()
       read()
       readlink()
       recv()
       recvfrom()
       recvmsg()
       rename()
       rmdir()
       select()
       sem_post()
       send()
       sendmsg()
       sendto()
       setgid()
       setpgid()
       setsid()
       setsockopt()
       setuid()
       shutdown()
       sigaction()
       sigaddset()
       sigdelset()
       sigemptyset()
       sigfillset()
       sigismember()
       signal()
       sigpause()
       sigpending()
       sigprocmask()
       sigqueue()
       sigset()
       sigsuspend()
       sleep()
       sockatmark()
       socket()
       socketpair()
       stat()
       symlink()
       sysconf()
       tcdrain()
       tcflow()
       tcflush()
       tcgetattr()
       tcgetpgrp()
       tcsendbreak()
       tcsetattr()
       tcsetpgrp()
       time()
       timer_getoverrun()
       timer_gettime()
       timer_settime()
       times()
       umask()
       uname()
       unlink()
       utime()
       wait()
       waitpid()
       write()

   POSIX.1-2008 supprime fpathconf(), pathconf() et sysconf() de la  liste
```

ci-dessus et ajoute les fonctions suivantes :

execl()
execv()
faccessat()
fchmodat()
fchownat()
fexecve()
fstatat()
futimens()
linkat()
mkdirat()
mkfifoat()
mknod()
mknodat()
openat()
readlinkat()
renameat()
symlinkat()
unlinkat()
utimensat()
utimes()

Interruption des appels système et des fonctions de bibliothèque par des gestionnaires de signal

Si un gestionnaire de signal est invoqué pendant qu'un appel système ou une fonction de bibliothèque est bloqué, alors :

* soit l'appel est automatiquement redémarré après le retour du gestionnaire de signal ;

* soit l'appel échoue avec l'erreur **EINTR**.

Lequel de ces deux comportements se produira dépend de l'interface et de si le gestionnaire de signal a été mis en place avec l'attribut **SA_RESTART** (voir **sigaction**(2)). Les détails varient selon les systèmes Unix ; voici ceux pour Linux.

Si un appel bloqué à l'une des interfaces suivantes est interrompu par un gestionnaire de signal, l'appel sera automatiquement redémarré après le retour du gestionnaire de signal si l'attribut **SA_RESTART** a été indiqué ; autrement, l'appel échouera avec l'erreur **EINTR** :

* Les appels **read**(2), **readv**(2), **write**(2), **writev**(2) et **ioctl**(2) sur des périphériques « lents ». Un périphérique « lent » est un périphérique où un appel d'entrées-sorties peut bloquer pendant un temps infini, par exemple un terminal, un tube ou une socket. (Selon cette définition, un disque n'est pas un périphérique lent.) Si un appel d'entrées-sorties sur un périphérique lent a déjà transféré des données au moment où il est interrompu par un gestionnaire de signal, l'appel renverra un code de succès (normalement, le nombre d'octets transférés).

* **open** (2), s'il peut bloquer (par exemple, lors de l'ouverture d'une FIFO ; voir **fifo**(7)).

* **wait**(2), **wait3**(2), **wait4**(2), **waitid**(2), et **waitpid**(2).

* Interfaces de sockets : **accept**(2), **connect**(2), **recv**(2), **recvfrom**(2), **recvmsg**(2), **send**(2), **sendto**(2) et **sendmsg**(2), à moins qu'une temporisation n'ai été placée sur la socket (voir ci-dessous).

* Interfaces de verrouillage de fichiers : opération **F_SETLKW** de **flock**(2) et **fcntl**(2).

* Interfaces de files de messages POSIX : **mq_receive**(3), **mq_timedreceive**(3), **mq_send**(3) et **mq_timedsend**(3).

* Opération **FUTEX_WAIT** de **futex**(2) (depuis Linux 2.6.22 ; auparavant, échouait toujours avec l'erreur **EINTR**).

* Interfaces de sémaphores POSIX : sem_wait(3) et sem_timedwait(3) (depuis Linux 2.6.22 ; auparavant, échouait toujours avec l'erreur **EINTR**).

Les interfaces suivantes ne sont jamais relancées après avoir été interrompues par un gestionnaire de signal, quelle que soit l'utilisation de **SA_RESTART** ; elles échouent toujours avec l'erreur **EINTR** lorsqu'elles sont interrompues par un gestionnaire de signal :

* Les interfaces de socket, quand une temporisation a été définie sur la socket en utilisant setsockopt(2) ; accept(2), recv(2), recvfrom(2) et recvmsg(2), si un délai de réception (**SO_RCVTIMEO**) a été défini ; connect(2), send(2), sendto(2) et sendmsg(2), si un délai de transmission (**SO_SNDTIMEO**) a été défini.

* Interfaces utilisées pour attendre des signaux : pause(2), sigsuspend(2), sigtimedwait(2) et sigwaitinfo(2).

* Interfaces de multiplexage de descripteurs de fichier : epoll_wait(2), epoll_pwait(2), poll(2), ppoll(2), select(2) et pselect(2).

* Interfaces IPC de System V : msgrcv(2), msgsnd(2), semop(2) et semtimedop(2).

* Interfaces de sommeil : clock_nanosleep(2), nanosleep(2) et usleep(3).

* read(2) sur un descripteur de fichier inotify(7).

* io_getevents(2).

La fonction sleep(3) n'est également jamais relancée si elle est interrompue par un gestionnaire, mais elle renvoie un code de retour de succès, le nombre de secondes restantes pour le sommeil.

Interruption des appels système et des fonctions de bibliothèque par des signaux d'arrêt

Sous Linux, même en l'absence de gestionnaires de signal, certaines interfaces en mode bloquant peuvent échouer avec l'erreur **EINTR** après que le processus ait été arrêté par l'un des signaux d'arrêt et relancé avec le signal **SIGCONT**. Ce comportement n'est pas ratifié par POSIX.1 et n'apparaît pas sur d'autres systèmes.

Les interfaces Linux qui affichent ce comportement sont :

* Les interfaces de socket, quand une temporisation a été définie sur la socket en utilisant setsockopt(2) ; accept(2), recv(2), recvfrom(2) et recvmsg(2), si un délai de réception (**SO_RCVTIMEO**) a été défini ; connect(2), send(2), sendto(2) et sendmsg(2), si un délai de transmission (**SO_SNDTIMEO**) a été défini.

* epoll_wait(2), epoll_pwait(2).

* semop(2), semtimedop(2).

* sigtimedwait(2), sigwaitinfo(2).

* read(2) sur un descripteur de fichier inotify(7).

* Linux 2.6.21 et antérieurs : opération **FUTEX_WAIT** de futex(2), sem_timedwait(3), sem_wait(3).

* Linux 2.6.8 et antérieurs : msgrcv(2), msgsnd(2).

* Linux 2.4 et antérieurs : nanosleep(2).

CONFORMITÉ

POSIX.1, sauf indication contraire.

BOGUES

SIGIO et **SIGLOST** ont la même valeur, le dernier est mis en commentaire dans les sources du noyau, mais certaines applications considèrent encore que le signal 29 est **SIGLOST**.

VOIR AUSSI

kill(1), **getrlimit**(2), **kill**(2), **killpg**(2), **setitimer**(2), **setrlimit**(2), **sgetmask**(2), **sigaction**(2), **sigaltstack**(2), **signal**(2), **signalfd**(2), **sigpending**(2), **sigprocmask**(2), **sigqueue**(2), **sigsuspend**(2), **sigwaitinfo**(2), **abort**(3), **bsd_signal**(3), **longjmp**(3), **raise**(3), **sigset**(3), **sigsetops**(3), **sigvec**(3), **sigwait**(3), **strsignal**(3), **sysv_signal**(3), **core**(5), **proc**(5), **pthreads**(7)

COLOPHON

Cette page fait partie de la publication 3.23 du projet man-pages Linux. Une description du projet et des instructions pour signaler des anomalies peuvent être trouvées à l'adresse http://www.kernel.org/doc/man-pages/.

TRADUCTION

Cette page de manuel a été traduite et mise à jour par Christophe Blaess <http://www.blaess.fr/christophe/> entre 1996 et 2003, puis par Alain Portal <aportal AT univ-montp2 DOT fr> jusqu'en 2006, et mise à disposition sur **http://manpagesfr.free.fr/**.

Les mises à jour et corrections de la version présente dans Debian sont directement gérées par Julien Cristau <jcristau@debian.org> et l'équipe francophone de traduction de Debian.

Veuillez signaler toute erreur de traduction en écrivant à <debian-l10n-french@lists.debian.org> ou par un rapport de bogue sur le paquet manpages-fr.

Vous pouvez toujours avoir accès à la version anglaise de ce document en utilisant la commande « **man -L C** <section> <page_de_man> ».