Cours Système n°8 : les signaux

Éric Gaudefroy

École Hexagone
Cursus Cyberdéfense - M1

1. Gestion des tâches et terminaux de contrôle

Dans cette section uniquement, le terme « tâche » est à prendre au sens du shell (cf. cours sur le shell) et non au sens de l'ordonnanceur (*cf.* cours sur l'ordonnancement). Comme expliqué dans les lignes qui suivent, une tâche logique peut ainsi comprendre une ou plusieurs tâches ordonnançables (threads et/ou processus).

1.1. Processus, groupes de processus et sessions

Un *processus* est une entité disposant d'un contexte autonome (par opposition à un thread) et capable de réaliser des actions dont notamment *fork*, *exec* et *exit* (*cf.* cours sur les processus).

Un **groupe de processus** (pgrp) est une **tâche logique** rassemblant un ou plusieurs processus réalisant collectivement un travail au profit de l'entité qui l'a initiée. Cette entité n'ayant généralement pas de visibilité sur les descendants qui seront ensuite créés, le regroupement de ces processus au sein d'un même groupe permettra une gestion collective par l'initiateur. Ainsi, l'exécution via un shell de »<cmd1> | <cmd2> » provoque la création d'un nouveau groupe avec deux processus, chacun étant ensuite libre de faire *fork* et/ou de lancer plusieurs threads.

Chaque groupe de processus possède **au plus un leader**. S'il en a un, il s'agit du processus dont le pid est numériquement égal au numéro du groupe (pgid). Lorsque l'on crée un nouveau groupe pour un processus, il en devient le leader. S'il fait *exit* avant les autres processus du groupe, le groupe n'a plus de leader.

Une **session** est **un ensemble de groupes de processus contrôlés par une même entité**. Il s'agit normalement d'un shell, attaché à un terminal de contrôle (voir la suite), et de l'ensemble des tâches lancées par ce shell et encore actives (en avant-plan ou en arrière-plan). En mode texte (localement ou lors d'un accès distant par ssh ou telnet), la notion de session correspond typiquement aux opérations lancées entre la connexion et la déconnexion d'un utilisateur. En mode graphique (localement), l'utilisateur peut exploiter plusieurs sessions simultanément à travers une connexion unique, en lançant plusieurs émulateurs de terminaux (xterm ou autre).

Chaque session possède **au plus un leader de session**. Il s'agit, s'il existe encore, du processus qui a créé la session. Typiquement, le leader d'une session est un shell.

Un groupe de processus est dit « orphelin » si aucun de ses membres n'a son père dans un groupe différent de la même session. Cette situation correspond normalement au groupe du leader de la

session (dont le père appartient à une autre session). Si le shell se termine avant les tâches qu'il a lancées, ou si ses fils directs se terminent avant leurs propres descendants, ces tâches peuvent également devenir orphelines. Dans ces deux derniers cas, la tâche orpheline n'est plus gérée en tant que tâche (soit parce qu'il n'y a plus de shell, soit parce que le shell a déjà récupéré le code de terminaison de ses descendants directs).

Nota : Tout processus du système appartient à un groupe et son groupe est inclus dans une session. Les « démons » (services) sont simplement isolés dans un groupe et une session individuels, détachés de tout terminal. Les applications graphiques lancées par l'interface graphique appartiennent généralement au groupe et à la session du client graphique (KDE, twm, etc.), également détachée de tout terminal.

1.2. Terminaux de contrôle

Les terminaux susceptibles de supporter une session sont les consoles virtuelles (*/dev/tty**, mode texte), les ports série (*/dev/ttyS**, mode texte à travers un terminal passif) et les pseudo-terminaux esclaves (*/dev/pts/** ou */dev/tty[p-za-e]**, pseudo-terminaux en mode graphique ou pour une connexion réseau), cf. la présentation des périphériques en mode caractère dans le cours sur les fichiers.
Un terminal possède toujours deux faces. La face externe, représentée par le périphérique en mode caractère, permet aux processus exploitant le terminal de récupérer des entrées (lecture de caractères sur le terminal) et de demander l'affichage d'une sortie (écriture de caractères sur le terminal). La face interne, gérée par le noyau, est chargée de fournir les entrées et d'honorer les sorties ; son fonctionnement dépend du type de terminal :

- pour une console virtuelle, les entrées et sorties correspondent respectivement aux réceptions de frappes clavier (envoyées par le noyau à la console active) et à l'écriture en mémoire (RAM vidéo affichable pour la console active) ;
- pour un port série, les entrées/sorties correspondent à des communications sur le connecteur série matériel ;
- pour un pseudo-terminal esclave, les entrées/sorties correspondent, au retraitement près, aux sorties et aux entrées du pseudo-terminal maître associé (cf. cours sur les fichiers).

Les traitements appliqués entre la face interne et la face externe du terminal dépendent de sa configuration.

Chaque terminal est associé à au plus une session, et chaque session a au plus un terminal de contrôle. L'association entre session et terminal se fait normalement lorsque le leader d'une session sans terminal ouvre (*open*, sans l'option O_NOCTTY) un terminal qui n'est pas encore associé à une session. Cette opération a en principe lieu avant le lancement de nouveaux processus dans cette session (*fork*) ; le terminal de contrôle de la session sera alors le terminal de contrôle de tous ses processus. C'est à ce terminal qu'ils auront accès en ouvrant */dev/tty*.

Parmi les groupes de processus de la session contrôlée par le terminal, au plus un est considéré d'avant-plan, les autres étant d'arrière-plan. Initialement, le groupe d'avant-plan est celui du leader de session. Un processus d'arrière-plan n'est normalement pas autorisé à lire sur son terminal de contrôle. En cas de tentative de lecture, il peut recevoir un signal (SIGTTIN) qui le gèle. L'écriture sur le terminal de contrôle peut également être régulée si le bit de configuration TOSTOP est positionné au niveau du terminal, ce qui n'est généralement pas le cas par défaut (*cf.* man tcsetattr,

hors programme). Les accès en écriture ou en gestion sont toutefois acceptés si le signal en question est ignoré ou bloqué.
Les processus du groupe d'avant-plan peuvent lire et écrire normalement sur le terminal de contrôle. Ce sont également ces processus, et eux seuls, qui reçoivent les signaux générés par le terminal, par exemple ceux résulants d'un Control+C (SIGINT) ou Control+Z (SIGTSTP) (*cf.* sections suivantes).

Nota : Les processus qui ne sont pas contrôlés par le terminal peuvent y lire et y écrire normalement, comme s'ils étaient en avant-plan. Le terminal fonctionnant en mode producteur / consommateur, les lectures concurrentes se partageront les caractères issus du terminal. Par ailleurs, un processus d'arrière-plan peut toujours lire sur son terminal de contrôle s'il prend la peine de bloquer le signal SIGTTIN.
La protection en confidentialité du terminal repose donc principalement sur les permissions affectées au fichier périphérique correspondant, plutôt que sur les mécanismes de gestion des tâches, qui visent essentiellement la fiabilité du fonctionnement du terminal (en évitant qu'une tâche d'arrière-plan ne consomme par erreur les entrées destinées à une tâche de premier-plan).
On notera en particulier qu'il n'y a pas de cloisonnement fort des entrées entre les différents processus rattachés à un même terminal de contrôle : le shell ou une tâche d'arrière-plan malveillante peuvent parfaitement lire le mot de passe saisi en réponse à une commande su lancée en avant-plan.

1.3. Commandes shell

Les commandes permettant de gérer des tâches en avant-plan ou en arrière-plan ont été présentées à l'occasion du cours sur le shell.
La commande suivante liste les processus en les regroupant par session et par groupe : **ps [-]axo sess,pgid,pid,ppid,tty,ucomm –sort sess,pgid,pid**
On pourra également utiliser, en alternative à –sort, l'option –forest de *ps* pour obtenir une représentation arborescente des processus.

2. Présentation du mécanisme de signalisation

Dans toute la suite du document, le terme « tâche » désigne une tâche au sens de l'exécution (un processus ou un thread indifféremment, du moment que les threads sont bien vus comme des fils d'exécution distincts par le noyau), et non plus au sens du shell (groupe de processus).
Les signaux constituent un mécanisme asynchrone de communication entre tâches permettant à une tâche ou au noyau de signaler à une autre tâche la survenue d'un évènement particulier, interférant, le cas échéant, avec le flot normal d'instructions exécuté par le destinataire. Un numéro de signal est associé à chaque envoi de signal. Ce numéro de signal indique la nature de l'évènement, un traitement différent pouvant être défini pour chaque type d'évènement. La réception d'un signal interrompt le flot d'exécution courant de la tâche qui le reçoit, et déclenche dans le contexte de cette tâche l'exécution d'une fonction de traitement. Au retour de cette fonction, si celle-ci n'a pas entraîné la terminaison de la tâche, le flot d'exécution reprend à l'endroit ou il avait été interrompu.

Nota : On peut rapprocher le mécanisme des signaux d'autres mécanismes asynchrones vus dans les cours précédents :

- le traitement des **interruptions** par le CPU : même logique d'enregistrement de fonctions de traitement, possibilité de masquage temporaire, même distinction entre évènements asynchrones (d'origine externe) et évènements synchrones (provoqués, volontairement ou non, par

le code courant), la principale différence étant le caractère matériel des interruptions, alors que les signaux sont purement logiciels et gérés par le noyau ;

- le traitement des requêtes d'annulation des threads.

On notera en revanche la différence fondamentale entre un mécanisme asynchrone comme les signaux et un mécanisme synchrone comme les *sockets*. Même si des données peuvent arriver dans le *buffer* de réception d'une *socket* de manière asynchrone, ces données ne seront prises en compte par la tâche destinataire que lorsque celle-ci demandera explicitement à les lire. A l'opposé, un signal qui n'est pas bloqué sera pris en compte par la tâche destinataire immédiatement au moment (arbitraire) de sa réception.

2.2. Utilisation des signaux

2.2.1. Emission d'un signal

L'émission d'un signal peut être décidée par le noyau ou par une tâche (processus ou thread) qui demande au noyau d'émettre le signal en son nom (par un appel système). Le destinataire peut être :

- un groupe de processus (prise en compte individuelle par chacun des processus du groupe) ;
- l'ensemble des processus auxquel l'émetteur a le droit d'envoyer un signal (*cf.* contrôle d'accès plus bas) ;
- un processus particulier (prise en compte par l'un quelconque de ses threads parmi ceux qui ne le masquent pas et/ou l'attendent explicitement, voir plus bas) ;
- un thread particulier d'un processus.

La gestion des signaux s'effectue en couche noyau, la liste des signaux en attente de traitement, des signaux masqués et des fonctions de traitement des signaux faisant partie du contexte associé à chaque tâche. Le noyau, chargé de l'émission des signaux (pour son compte ou en réponse à un appel système), peut modifier cette liste.

Les signaux envoyés à l'initiative du noyau servent souvent à répercuter au niveau d'une tâche une exception CPU qu'elle a provoquée (instruction illégale, accès à une zone mémoire interdite, etc.) – on parle dans ce cas de signaux synchrones. Les signaux synchrones ne peuvent normalement être reçus que par le thread qui les a provoqués. Le noyau peut également envoyer des signaux de manière asynchrone, typiquement afin de prévenir le processus qu'un certain évènement est arrivé (arrivée de données dans un tube nommé, etc.) – selon les cas, le processus doit indiquer au préalable qu'il souhaite être prévenu par un signal (configuration par *fcntl*/*ioctl*, etc.).
Les signaux envoyés à l'initiative d'autres tâches en couche utilisateur peuvent servir à indiquer que l'on souhaite que le processus se termine, à l'informer que son fichier de configuration a été modifié, etc.

2.1.2. Contrôle d'accès

Pour des raisons de sécurité, l'émission de signaux entre tâches ne fonctionne que si l'émetteur est **root** ou s'il a la même identité que le destinataire. En notant (*euid*, *uid*, *suid*) les identités effective, réelle et sauvegardée de la tâche qui demande l'émission d'un signal et (euid', uid', suid') celles de la tâche destinataire du signal, l'envoi du signal n'est autorisé que si l'une des trois conditions ci-dessous est vérifiée :

- L'émetteur et le destinataire ont une identité commune, plus précisément un au moins des identifiants de l'émetteur parmi (euid, uid) est égal à un des identifiants du destinataire parmi (uid', suid'). L'utilisation des identités réelle et sauvegardée du destinataire permet par exemple à un processus d'envoyer un signal à un de ses fils qui aurait exécuté un fichier disposant d'un bit 's', et par conséquent changé d'identité effective.
- L'émetteur dispose des privilèges root (euid == 0). Sous Linux uniquement, ce test sur euid est remplacé par un test de capacité : l'envoi est autorisé si l'émetteur dispose de CAP_KILL (laquelle est reservée à root sauf mécanisme particulier).
- Le signal SIGCONT (réveil d'une tâche en sommeil) peut par exception être envoyé sans contrôle d'identité vers n'importe quel autre processus appartenant à la même session que l'émetteur, ce qui permet typiquement au shell de réactiver un processus gelé même si il a bénéficié d'un bit « s » et qu'il a ensuite changé toutes ses identités.

En particulier, ces conditions signifient qu'une tâche doit disposer des privilèges **root** (ou du moins de CAP_KILL, sous Linux) pour envoyer un signal arbitraire à un processus sans identité commune avec la sienne.
En revanche, l'envoi d'un signal à un processus ayant disposé d'un bit *setuid* reste possible tant que ce processus n'a pas mis à jour l'ensemble de ses identités (par exemple par *setresuid*).

Naturellement, le noyau peut envoyer un signal à n'importe quelle tâche du système, sans contrôle d'accès particulier.

2.2. Traitement des signaux

Le noyau traite la réception d'un signal par une tâche de la façon suivante :

- chaque tâche dispose (dans le contexte qui lui est associé par le noyau) d'un masque représentant les signaux bloqués (un masque par thread) :
  - si le signal reçu est bloqué, il est mis en attente jusqu'à ce qu'il ne soit plus masqué, ou jusqu'à ce que la tâche se mette en attente explicite (bloquante) de ce signal (*cf.* sigwait) ;
  - sinon il est traité, même si la tâche est prise dans un appel bloquant (mais pas si elle a été gelée, *cf.* plus bas) ;
- chaque tâche dispose également d'un tableau d'actions associées au traitement des signaux (un seul tableau par processus, partagé par tous ses threads) :
  - si le traitement du signal a été configuré, le signal pourra provoquer l'exécution d'une routine de traitement préenregistrée ou être ignoré (ignorer un signal constitue une routine préenregistrée particulière) ;
  - si le traitement du signal n'a pas été configuré, l'action par défaut associée à ce signal est appliquée ;

Les actions par défaut, en l'absence de fonctions de traitement spécifiques, dépendent du signal considéré :

- ignorer le signal ;
- terminer le processus, avec ou sans création d'un core dump ;
- geler le processus, qui devient dans ce cas non-éligible du point de vue de l'ordonnanceur, et ne pourra être dégelé que par un autre signal ;
- dégeler un processus gelé.

Les actions ont pour la plupart lieu à l'échelle du processus (gel/terminaison de tous les threads) même si le destinataire est un thread particulier (avec les LinuxThreads, qui gèrent un processus par thread, le gel ne concerne que le thread visé).

On notera bien que l'exécution de la routine de traitement d'un signal intervient au moment de la réception de ce signal, ce qui peut interrompre n'importe où le flot normal d'exécution, qui reprendra après traitement si le signal n'était pas fatal. Dans le cas particulier où la tâche qui reçoit le signal est dans l'état bloqué au sens de l'ordonnanceur au moment de la réception, cette tâche est passée dans l'état éligible le temps de traiter le signal (le comportement à l'issue de ce traitement est dans ce cas variable, *cf.* paragraphe 6 sur la reprise des appels bloquants plus bas). Une tâche en état zombie ne peut pas recevoir de signal (elle n'a d'ailleurs plus de masque de signaux ni de tableau d'actions).

2.3. Cas de Linux

Le cas de Linux est particulier dans la mesure où les threads sont gérés comme des tâches différentes au niveau du noyau :

- avec les LinuxThreads, l'envoi d'un signal à un processus désigne en fait un thread particulier dans la mesure où chaque thread a un tgid propre (son « pid » virtuel, *cf.* cours sur les threads) ; par ailleurs, les effets liés aux signaux (terminaison, gel) n'affectent que le thread visé ;
- avec NPTL, l'envoi d'un signal à un processus est correctement géré, le pid utilisé comme cible pouvant indifféremment être le tgid global ou le pid individuel d'un des threads ; les effets de terminaison ou de gel sont bien propagés à l'ensemble des threads du processus.

Nota : Dans la pratique, l'utilisation de signaux en contexte multithreads soulève de nombreuses difficultés (réentrance, risque d'autoverrouillage, *cf.* plus bas) et doit autant que possible être évitée au profit de méchanismes synchrones.

3. Quelques signaux

La liste qui suit présente les principaux signaux à connaître et leur signification habituelle. Les qualificatifs sync/async s'appliquent à un contexte « usuel » d'utilisation, synchrone (exception) ou asynchrone (évenement externe).
Le lecteur est par ailleurs renvoyé à *man signal* sous Linux pour une liste plus complète des signaux standards et le détail des comportements par défaut (ignorer le signal, geler/débloquer le processus ou terminer le processus, avec ou sans « coredump »). Il peut y avoir de légères différences sur le comportement par défaut ou la liste de signaux utilisés d'un OS à l'autre.

3.1. Répercussion des exceptions CPU par le noyau

Les signaux qui suivent sont généralement envoyés par le noyau à une tâche pour signaler à cette dernière une exception levée au cours de son exécution par le CPU ou la MMU. En l'absence de masquage ou de traitement spécifique, leur réception par une tâche entraîne la terminaison de celle-ci (avec en général création d'un fichier core dump).

- SIGSEGV (sync) : Accès à une adresse mémoire incorrecte (« Segmentation Fault »), soit parce que l'adresse est invalide, soit parce que l'accès se fait selon un mode incompatible avec les protections mémoire en vigueur (segmentation, pagination, *cf.* cours sur la mémoire), ou

encore utilisation d'une instruction processeur privilégiée en couche utilisateur (ce qui revient également à une faute de segment sur une architecture x86) ;

- SIGBUS (sync) : Selon la nature de l'OS et la cause de l'exception CPU, SIGSEGV est parfois remplacé par SIGBUS (cas de FreeBSD) ;
- SIGFPE (sync) : Erreur dans un calcul à virgule flottante ou division par zéro ;
- SIGILL (sync) : Instruction machine illégale (instruction non comprise par le CPU).

3.2. Changement de l'environnement d'exécution

Les signaux suivants sont envoyés par le noyau pour signaler des évenements extérieurs au déroulement d'un programme (disponibilité ou non d'une ressource par exemple). Certains ne sont envoyés qu'aux tâches qui ont explicitement demandé la notification d'un type d'événement (SIGPOLL), d'autres le sont systématiquement (SIGCHLD). L'action par défaut associée à la plupart de ces signaux est de terminer le processus.

- SIGCHLD (async) : Changement d'état d'un processus fils (terminaison, ou éventuellement blocage ou déblocage) – utile si le père ne souhaite pas s'occuper du wait tant que le fils ne s'est pas terminé (il suffit de faire wait dans la fonction de traitement du signal ; de préférence en faire plusieurs en mode non bloquant pour gérer le cas où plusieurs fils seraient terminés mais où le signal n'aurait pas été dupliqué) ;
- SIGHUP (async) : Déconnexion du terminal de contrôle ; ce signal est par ailleurs souvent utilisé pour indiquer manuellement aux services du système (qui fonctionnent en mode démon, sans terminal de contrôle) qu'ils doivent relire leur fichier de configuration suite à une modification ;
- SIGPIPE (sync) : Tentative de production dans une ressource de type producteur / consommateur déconnectée (typiquement, un tube ou une *socket* en mode *stream* dont la connexion est fermée à l'autre bout) – ce signal accompagne alors l'erreur EPIPE ;
- SIGURG (async) : Nouvelle réception en mode urgent (transmission hors de bande sur une socket en mode *stream*) sur une *socket* "surveillée" par le processus ou le groupe (hors programme) ;
- SIGPOLL (aussi nommé SIGIO) (async) : évènement asynchrone (arrivée de données en réception, apparition d'espace libre dans un buffer d'émission, erreur, etc.) sur un descripteur de fichier « surveillé » par le processus ou le groupe (hors programme).

3.3. Signaux liés au terminal de contrôle

Les signaux qui suivent sont ceux qui sont susceptibles d'être envoyés par le terminal de contrôle par le biais de raccourcis claviers. Dans ce cas, ils sont transmis à l'ensemble des processus du groupe d'avant-plan.

- SIGINT (async) : Demande d'interruption (Ctrl-C) depuis le terminal de contrôle ;
- SIGTSTP (async) : Demande de gel (Ctrl-Z) depuis le terminal de contrôle ;
- SIGQUIT (async) : Demande d'arrêt (Ctrl-\) depuis le terminal de contrôle (avec possibilité de produire un fichier *coredump*).

Par ailleurs, le terminal de contrôle est également susceptible de déclencher l'envoi des signaux suivants à un groupe de processus d'arrière-plan qui tenterait un accès non autorisé au terminal. Ils sont également transmis à l'ensemble des processus du groupe concerné.

- SIGTTIN (sync) : Tentative de lecture sur un terminal de contrôle en tant que processus d'arrière-plan ;
- SIGTTOU (sync) : Tentative d'écriture (dépend de la configuration du terminal) ou de gestion d'un terminal de contrôle en tant que processus d'arrière-plan.

Comme évoqué plus haut, ces signaux peuvent parfaitement être bloqués ou ignorés par un programme. Leur envoi ne constitue par conséquent pas une mesure de contrôle d'accès robuste.

3.4. Contrôle d'exécution du programme

Les signaux suivants peuvent par ailleurs être envoyés, sans raccourci clavier et généralement à la demande explicite de l'utilisateur ou de l'administrateur, pour geler ou terminer l'exécution d'un programme :

- SIGTERM (async) : Demande d'arrêt (courtoise) du processus ;
- SIGKILL (async) : Arrêt brutal du processus, signal non configurable ;
- SIGSTOP (async) : Gel du processus, signal non configurable ;
- SIGCONT (async) : Reprise d'un processus gelé (peu configurable ; autorisé à être envoyé entre processus d'identités différentes s'ils sont dans une même session) ;

Les signaux « non configurables » ne peuvent être ni masqués, ni ignorés, ni redirigés vers une fonction spécifique du programme. Ainsi, SIGKILL et SIGSTOP permettront toujours de terminer ou bloquer une tâche, quels que soit son masque de signaux bloqués et les fonctions de traitement qu'elle a enregistrées. SIGCONT est « peu configurable » en ce sens que sa réception par une tâche bloquée entraînera toujours au moins le réveil de cette tâche (même si une fonction de traitement est par ailleurs définie et exécutée pour ce signal).

3.5. Signaux programmés par la tâche elle-même

- SIGALRM, SIGVTALRM et SIGPROF (async) : Expiration d'un timer déclenché par le processus ;
- SIGABRT (sync) : Arrêt anormal du processus (*cf. abort*).

Le signal SIGABRT est notamment celui qui est envoyé au processus (par lui-même) lorsqu'une corruption mémoire est détectée dans le tas (typiquement par la *glibc*, lors d'un appel à *free*) ou dans la pile (lorsque des « canaris » sont mis en place pour détecter les débordements, *cf.* cours sur l'assembleur). Même s'il est possible de définir une fonction de traitement qui « rattrape » ce signal, il faut bien tenir compte dans l'écriture de cette fonction du fait que la mémoire du processus est potentiellement corrompue ; on évitera par conséquent tout traitement complexe (et en particulier la manipulation du tas ou de la pile - par exemple afin de produire une trace de débug).

3.6. Signaux de libre emploi

- SIGUSR1 et SIGUSR2 (async) : Signaux qui n'ont pas de sens prédéfini, l'application (a priori multiprocessus) étant libre de les utiliser pour ses propres besoins de signalisation.

Par exemple, sous Linux, SIGUSR1 est souvent utilisé par le serveur graphique X11 pour signaler à son père (selon les cas, xinit ou un gestionnaire de session comme xdm) qu'il s'est bien lancé et est prêt à recevoir des connexions de clients.

3.7. Remarques

Plusieurs fonctions de langages de programmation utilisent en sous-main certains de ces signaux (via des timers, etc.) ou d'autres signaux spécifiques et non documentés (par exemple pour la synchronisation des threads LinuxThreads). De même, lors de l'écriture d'un programme faisant appel à une bibliothèque de haut niveau (en particulier de type toolkit graphique, GTK ou Qt par exemple), on ne peut pas exclure que certains signaux soient utilisés implicitement par des fonctions de la bibliothèque, et ne puissent par conséquent pas être utilisés explicitement par le programme sans perturber le fonctionnement de la bibliothèque.

Dans des conditions standard d'exécution, certains signaux (SIGFPE, SIGILL, SIGSEGV, etc.) ne sont envoyés que par le noyau en réponse à une exception causée par le code utilisateur en cours d'exécution ; d'autres (SIGALRM, SIGCHLD, etc.) sont générés par le noyau en réponse au mode de fonctionnement du processus ; d'autres encore (SIGINT, SIGKILL, SIGHUP, etc.) sont envoyés par le noyau ou par d'autres processus pour traduire certaines requêtes ou changements dans l'environnement ; enfin, SIGUSR1 et SIGUSR2 ne sont pas censés être envoyés à l'initiative du noyau ou des composants standard du système.

Du point de vue de la sécurité, il est important de retenir que n'importe quel processus autorisé à envoyer des signaux à un autre processus peut lui envoyer un signal quelconque (y compris SIGSEGV, SIGCHLD, etc.). Ainsi, un attaquant pourra par exemple envoyer un signal SIGSEGV à un programme manipulant des données sensibles, en espérant déclencher dans ce dernier un traitement d'erreur qui pourrait conduire à la compromission d'éléments sensibles.

4. Difficultés liées à la gestion des signaux

La mise en oeuvre robuste des signaux au sein d'un programme est soumise à un certain nombre de difficultés, qui découlent principalement, soit de la nature asynchrone du traitement des signaux, soit de la définition incomplète des comportements associés à certaines des fonctions permettant de gérer les signaux, qui posent en particulier des problèmes de portabilité.

4.1. Interruption d'un appel bloquant

Si une tâche bloquée sur un appel système (avec ou sans timeout) est réveillée par la réception d'un signal non fatal, qui la rend éligible du point de vue de l'ordonnanceur le temps de traiter ce signal, il peut se produire deux choses une fois le signal traité (en supposant que ce dernier n'entraîne pas la terminaison de la tâche) :

- Soit l'appel bloquant est repris, dans la mesure du possible, la tâche repassant dans l'état bloqué sans rendre la main à l'appelant. La réception du signal est dans ce cas transparente pour l'appelant. Il s'agit notamment du comportement standard sous BSD et sous Linux en l'absence de fonction spécifique de traitement d'un signal : les appels bloquants interrompus par l'action par défaut associé à un signal sont automatiquement repris.
- Soit l'appel bloquant est interrompu, la tâche reste éligible et ressort de l'appel avec un code d'erreur EINTR (à ne pas confondre avec EAGAIN, qui dénote généralement un échec lié à l'utilisation d'un mode non bloquant). La charge de relancer l'appel le cas échéant est dans ce cas laissée à l'appelant. Ce comportement est notamment celui observé sous Linux et BSD lorsqu'une fonction de traitement spécifique a été enregistrée par signal, mais cette propriété n'est pas forcément vérifiée sous d'autres UNIX.

La première approche est plus simple pour l'appelant, qui n'a pas à se soucier de la possibilité de sortie en erreur avec EINTR. Cependant, l'interruption de l'appel bloquant peut dans certains cas être souhaitée par l'appelant (le signal l'informant typiquement que l'appel n'a plus lieu d'être, ou doit être abandonné, par exemple du fait de l'expiration d'un timeout). Dans tous les cas, l'écriture d'un code robuste nécessite en général de prendre en compte la possibilité d'interruption des appels bloquants par un signal. Ainsi, on aura par exemple recours à des fonctions de lecture qui « encadrent » l'appel à *read*.

Par ailleurs, quel que soit le comportement par défaut, on notera qu'il est toujours possible de configurer finement le traitement des signaux pour choisir explicitement la stratégie à adopter (voir notamment man siginterrupt). En revanche, les différences dans le comportement par défaut d'un système d'exploitation à l'autre et d'un type d'appel à l'autre peuvent être source de difficultés dans le développement d'un code portable.

Le comportement de Linux en cas d'interruption d'un appel bloquant est particulièrement complexe : un appel bloquant interrompu est normalement repris automatiquement si aucun gestionnaire de signal spécifique n'a été appelé (i.e. le signal a été traité par le gestionnaire par défaut pour le numéro de signal considéré), ou s'il a été traité par un gestionnaire de signal installé à l'aide de la fonction sigaction avec l'option SA_RESTART, et n'est pas repris automatiquement dans les autres cas (gestionnaire installé avec sigaction sans SA_RESTART ou avec la fonction « historique » signal). Cependant, plusieurs exceptions à cette logique sont à noter :

- D'une part, un certain nombre d'appels système ne sont pas repris automatiquement (donc retournent toujours une erreur EINTR) après appel d'un gestionnaire spécifique, même lorsque ce dernier a été installé avec l'option SA_RESTART. Il s'agit notamment des appels systèmes liés aux IPC System V (msgrcv, etc.), de select et de ses dérivés, et des fonctions elles-mêmes liées à la gestion des signaux (pause, sigsuspend, etc.).
- D'autre part, un certain nombre d'appels système (dont le nombre tend à décroitre avec les versions successives du noyau) retournent systématiquement une erreur EINTR lorsqu'il sont interrompus par le traitement par défaut des signaux SIGSTOP et SIGCONT (gel et dégel du processus). Le cas de la lecture sur un descripteur de fichiers inotify constitue un exemple de cette exception.

Nota : Si un signal est reçu alors que la tâche a été gelée par un autre signal (suite à l'invocation du traitement par défaut de certains signaux comme SIGSTOP ou SIGTSTP), son traitement est généralement repoussé jusqu'au déblocage de la tâche (exceptions : SIGKILL, SIGCONT). Cet état « gelé » ne doit pas être confondu avec l'état d'une tâche simplement bloquée sur un appel système, qui est quant à elle rendue éligible pour traiter les signaux qu'elle reçoit.

4.2. Fonction de traitement des signaux reçus

Par défaut, sur un système moderne, lorsqu'une tâche est en train de traiter un signal reçu ce dernier est automatiquement masqué jusqu'à la fin de son traitement, de telle sorte que si le même signal est à nouveau reçu, on n'interrompra pas le traitement en cours. En revanche, **les autres signaux ne seront par défaut pas masqués, et la réception d'un signal différent pourra interrompre le traitement du premier signal**. Il est toutefois possible de configurer finement le traitement des signaux, notamment à l'aide de *sigaction*, de telle sorte que le signal traité ne soit pas masqué et/ou que d'autres signaux le soient pendant la durée du traitement.

Dans la mesure où une fonction de traitement peut être invoquée à tout instant et être elle-même interrompue par la réception d'un autre signal, il est souvent nécessaire d'imposer que ces fonctions et les fonctions qu'elles utilisent soient nativement réentrantes (ou plus généralement async-signal-safe, cf. cours sur les threads) et/ou de masquer les signaux dans certaines sections critiques. La difficulté vient du fait que le thread exécutant la fonction de traitement est celui qui a été interrompu, donc que la synchronisation ne peut pas se faire par des verrous : il y aurait sinon risque d'**autoblocage**. Ainsi, si un thread protège les accès à une section critique par un verrou et doit prendre ce même verrou dans le traitement d'un signal, la réception du signal alors que le thread est déjà dans la section critique bloquera complètement le thread : le traitement du signal sera bloqué faute de pouvoir obtenir le verrou, qui ne pourra être libéré que si le traitement du signal s'achève et l'exécution normale reprend... La page de man signal donne notamment une liste de fonctions qui sont garanties async-signal-safe. On notera en particulier que les fonctions d'entrée-sortie de haut-niveau comme *printf* sont susceptibles de manipuler des verrous implicites, et ne doivent donc pas être appelées depuis un gestionnaire de signal.

Nota : En plus de l'async-signal-safety, on attend souvent d'une fonction de traitement de signal qu'elle soit simple (peu d'appels imbriqués) et que son exécution se termine en temps réduit, afin de rester lisible et de perturber le moins possible (en termes de délais et de ressources) l'exécution normale du programme. En particulier, on prendra garde à ne réaliser dans une telle fonction aucun appel système susceptible de bloquer.

Lorsqu'une fonction de traitement enregistrée pour un signal a été déclenchée, le système peut au choix :

- conserver cette fonction comme fonction de traitement du signal considéré, mais masquer le signal pendant son traitement (approche moderne BSD, suivie par Linux) ;
- restaurer le comportement par défaut pour ce signal (gestion one shot, ancienne approche System V).

Là encore, une configuration fine permet de choisir explicitement la stratégie à adopter, mais il n'est pas portable de faire des hypothèses sur le comportement par défaut.

4.3. Bonne réception des signaux

La tâche qui émet un signal n'a pas de retour concernant sa bonne réception et/ou son traitement en dehors des cas d'erreur triviaux à l'émission (destinataire inexistant, privilèges insuffisants, etc.). On ne peut donc pas, sauf mécanisme particulier (envoi d'un autre signal en retour par exemple) avoir la certitude qu'un signal envoyé à été bien pris en compte par son destinataire.

Par ailleurs, selon les implémentations et les signaux considérés, la liste des signaux en attente peut être gérée par le noyau comme un simple tableau de booléens avec en indice le numéro du signal. Dans ce cas, au plus une instance d'un même signal reçu sera stockée en attendant d'être traitée, et les autres seront perdues. De plus, l'ordre dans lequel les signaux seront transmis à un processus n'est pas forcément connu (par ordre d'émission, par numéro de signal, etc.). Ces points ne doivent pas paraître choquants compte-tenu de la raison d'être de la signalisation : ainsi, un signal en attente qui indique un changement dans un fichier de configuration n'a pas besoin d'être reçu deux fois de suite par l'application ; si elle a pris du retard à la réception, elle chargera directement la dernière version en relisant ce fichier. De même, une requête d'arrêt n'a pas besoin d'être stockée en double.

Ainsi, par exemple, la réception d'un signal SIGCHLD ne signifie pas nécessairement la terminaison d'un unique processus fils : si plusieurs fils se terminent en rapide succession, le gestionnaire de signal pourra n'être appelé qu'une seule fois (cf. exemple de traitement de SIGCHLD donné dans la section précédente. Des implémentations plus riches gèrent néanmoins des queues de signaux et acceptent donc plusieurs instances d'un même signal au sein d'une queue. Ces queues peuvent d'ailleurs être classées par ordre de priorité pour les applications temps réel. Sous Linux et BSD, les signaux standard sont non duplicables, tandis que les signaux « temps réel » spécifiques à Linux (hors programme) sont gérés par queues.

Une dernière limitation des signaux concerne la fiabilité de leur origine. Les interfaces simples pour l'enregistrement de gestionnaires de signaux ne permettent pas à ces gestionnaires d'accéder à des informations sur l'émetteur du signal : noyau ou autre processus, identité de ce processus, etc. L'enregistrement des gestionnaires par sigaction permet de définir des gestionnaires plus avancés, qui se voient communiquer des informations précises sur l'origine et la cause de l'envoi du signal. Ces informations ne sont malencontreusement que rarement utilisées, et les cas de gestionnaires de signaux réalisant une opération sensible sur la base du seul numéro de signal reçu, sans s'interroger sur l'origine de ce signal, restent courants (*cf.* discussion sur les core dump plus bas). En règle générale, il vaut mieux éviter de déclencher une opération sensible (par exemple, l'affichage du contenu d'une zone mémoire) sur simple réception d'un signal, en tout cas pas sans avoir obtenu des garanties suffisantes sur son origine légitime.

4.4. Terminaison d'un processus à cause d'un signal

La réception d'un signal fatal ni masqué, ni ignoré, ni redirigé vers une routine de traitement termine immédiatement le processus concerné. Ce mode brutal de terminaison ne permet pas de gérer les routines de synchronisation / désallocation des API de haut niveau : la sortie est semblable à une sortie par *_exit* (par opposition à *exit*, qui prend le temps d'exécuter les fonctions enregistrées par atexit et de synchroniser les FILE * gérés par l'API de haut niveau). Pour cette raison, on peut chercher à intercepter les signaux de terminaison en enregistrant comme fonction de traitement une fonction chargée de désallouer proprement les ressources puis de faire exit (ou de regénérer le signal par *raise* afin de se terminer avec le bon code de terminaison).

Nota : Le code de terminaison d'un processus qui s'est terminé anormalement suite à la réception d'un signal fatal est distinguable de celui d'un processus qui est sorti volontairement par un appel à *_exit* (avec un code d'erreur ou de succès), *cf.* la description de *waitpid* et en particulier de la macro WIFEXITED dans le cours sur les processus.

Nota : Le traitement par défaut de certains signaux (notamment SIGSEGV) entraîne non seulement la terminaison brutale du processus destinataire, mais également la génération d'un core dump de ce processus, c'est-à-dire d'un fichier contenant l'image mémoire du processus au moment de sa terminaison. Ce traitement peut ainsi aboutir à l'écriture sur le disque d'éléments secrets manipulés en mémoire par le processus (mots de passe, clés cryptographiques), ce qui peut selon les cas s'apparenter à une compromission de ces éléments (voir aussi discussion sur le swap, dans le cours sur la mémoire). Le cas échéant, des précautions spécifiques peuvent être mises en œuvre pour interdire la génération de core dumps : suppression de leur support lors de la compilation du noyau ou mise à zéro de la rlimit RLIMIT_CORE pour un processus donné notamment.
On notera bien que, même si les signaux entraînant la génération de core dumps sont normalement émis par le noyau suite à une exception, rien n'interdit à une tâche malveillante d'envoyer un tel

signal à une autre tâche du même utilisateur (*cf.* remarques sur l'origine des signaux plus haut).

4.5. Conservation du masque de signaux bloqués

La définition des gestionnaires de signaux est conservée à travers un *fork*, mais est réinitialisée à l'ensemble des traitements par défaut lors d'un *execve*. En revanche, le masque des signaux bloqués par chaque *thread* est conservé non seulement lors d'un *fork* (ou *pthread_ create*), mais aussi lors d'un *execve* : le nouveau processus hérite du masque du thread qui a appelé *execve*. Un programme ne peut par conséquent pas faire d'hypothèse a priori sur la possibilité de recevoir un signal, et doit réinitialiser explicitement le masque de signaux bloqués pour obtenir les propriétés désirées. En l'absence d'une telle réinitialisation, un attaquant serait en mesure lancer le programme en masquant certain signaux afin de perturber son fonctionnement.

5. Manipulation des signaux en ligne de commande

Les commandes shell pour la manipulation de signaux sont les suivantes :

- **kill** [-s <nom_sig>] <pid> : envoi d'un signal à un processus (envoie SIGTERM si on ne précise pas le type de signal) ;
- **nohup** <cmd> : lancement d'une commande en mode autonome (ignore les signaux liés à la gestion du terminal), en redirigeant les sorties vers nohup.out ; permet à l'utilisateur de lancer une tâche de type calcul long et de se déconnecter (fermeture de « session ») ;
- **trap** <cmd> <signaux> : builtin bash pour l'enregistrement d'une routine de traitement de signaux (hors programme ; utiliser C pour les programmes nécessitant un traitement des signaux).

Sous windows PowerShell, la manipulation est réalisée avec :

- **TASKLIST** [/S système [/U utilisateur [/P [mot_de_passe]]]] : affiche une liste des processus actuellement en cours sur un ordinateur local ou un ordinateur distant ;
- **TASKKILL** [/S system [/U [/P [password]]]] [/FI filter] [/PID process id | /IM imagename] [/T][/F] : utilitaire Microsoft permettant de terminer un ou plusieurs processus ou tâche (taskkill /PID process-number /F) ;
- **Stop-Process** [-Id] <int[]> [<CommonParameters>] : similaire à TASKKILL mais dispoe d'otpions différentes.

6. Utilisation des signaux en programmation

Si elle ne crée pas de parallélisme au sens propre du terme, l'utilisation de signaux dans une application génère de l'incertitude dans le déroulement logique du programme (interruptibilité et ordre d'appel des fonctions). Il faut donc écrire les fonctions de traitement avec le même soin que lors de la programmation d'applications multithreads.

Nota : Dans une application monothreadée, le processus qui a été interrompu pour traiter un signal ne peut plus rien faire d'autre tant que la fonction de traitement ne s'est pas terminée. Par conséquent, le blocage de la fonction de traitement sur une ressource contrôlée par le processus provoque un verrou mortel.

Par ailleurs, il ne faut pas perdre de vue qu'une application qui ne se sert pas des signaux **n'est pas à l'abri d'en recevoir**.
Le TD associé à ce cours permet de découvrir la programmation des signaux sous Python.

Ce document est inspiré du « Cours Système n°8 : les Signaux » délivré par monsieur Vincent Strubel dans les cours dispensés à l'Agence nationale de la sécurité des systèmes d'information (Janvier 2013).