umsignal

EPITA - SRS 2005

Thomas Tychensky
David Brochoire
Nicolas Vigier

Conception et implémentation d'une bibliothèque de transmission de messages d'un processus à un autre par une interface proche de celle utilisée pour l'émission et la réception de signaux.

Il s'agit d'un nouveau type de signaux en User-Land permettant de lui rattacher des données.

Septembre-Octobre 2004

Ce document peut être traduit et distribué librement. Il est soumis aux termes de la licence 'GNU Free Documentation License' (FDL) disponible sur http://www.gnu.org/copyleft/fdl.html.

Sommaire

Introduction

Dans le cadre de nos études à l'EPITA (École Pour l'Informatique et les Techniques Avancées), et de notre spécialisation SRS (Systèmes Réseaux et Sécurité), nous devions réaliser un système de signaux comparable au système de signaux sous Unix.

A la différence de ces derniers, nos signaux (les umsignaux) doivent fonctionner en mode User-Land, supporter l'envoi de données rattachées aux signaux, fournir une gestion de droit plus complexe et assurer la bonne réception des signaux.

La durée du travail s'étalait sur environ un mois et demi et la date du rendu des livrables était le 12 octobre 2004.

Chapter 1

Description du projet

1.1 Rappel des signaux Unix

1.1.1 Présentation

Un signal est une notification logicielle d'un événement faite à un processus. Le signal est généré lorsque son événement déclencheur se produit. Les signaux sont alors délivrés lorsque le processus lance une action correpondant à l'arrivée des signaux. Avant cela ils sont dit en attente (pending). Le temps entre la génération du signal et le moment ou il est délivré peut varier.
Lorsque que le signal est transmis au processus, celui-ci execute un handler spécifique affecté au signal. Le handler peut être éventuellement une fonction écrite par un utilisateur. Il est possible de bloquer les signaux pour empêcher d'eventuelle interruption asynchrone, ou alors d'ignorer certains signaux.
Un utilisateur (ou un processus utilisateur) peut envoyer un signal à un processus seulement si le processus tourne sous le compte de l'utilisateur. Root peut envoyer des signaux à tous.

1.1.2 Limitation

Dans le système de signaux décrits précédemments il existe un certains nombres de limitations.

Les droits des signaux

La limitation des signaux à des processus tournant sous le même utilisateurs peut apparaître tout à fait normal pour des signaux du genre de SIGKILL. Cependant pour des signaux tel que SIGUSR1 ou SIGUSR2 il pourrait être intéressant d'utiliser les signaux dans un système de notification en permettant à des processus tournant sous d'autres utilisateurs d'envoyer des signaux.

Les données transmises

L'envoi d'un signal se résume à une notification car les signaux Unix ne permettent pas de transmettre des informations en même temps que la notification.

La réception des signaux

Avec les signaux Unix il est possible de perdre certains signaux. Ainsi lorsque les signaux sont bloqués, qui il y ai un signal en attente ou plusieurs, lorsque le processus débloquera le signal correpondant il n'en recevra qu'un seul. De même si plusieurs signaux arrivent en même temps, l'ordre dans lequel ils sont délivrés au processus peut ne pas correspondre à leur ordre d'émission au processus.

1.2 Présentation du sujet

1.2.1 Le but du projet

Le but du projet est de développer un système de signaux en User-Land sous la forme d'une bibliothèque C. Le système de signaux appelé umsignaux doit être le plus portable possible.
L'interface de l'api est fournie avec le sujet ainsi que son principe de fonctionnement. Son implémentation est libre et laisse les choix techniques aux développeurs.
Les spécifications de base de la bibliothéques sont les suivantes :

- Il existe 255 umsignaux numérotés de 1 à 255, sans signification particulière. Le umsignal numéro 0, comme le signal numéro 0, n'existe pas.
- Par défaut, tout umsignal est ignoré.
- Un processus récepteur peut choisir de recevoir un umsignal particulier, en lui affectant un handler. Cet handler sera alors appelé dès que le processus recevra le umsignal qu'il écoute, interrompant le cours normal de l'exécution et la reprenant une fois le handler terminé.

Ces différentes fonctionnalités reprennent en partie le principe du fonctionnement des signaux Unix.

1.2.2 Les apports par rapport aux signaux Unix

Les améliorations sont en rapport avec les limitations précédentes.

- Les umsignaux peuvent s'empiler. L'ordre dans lequel ils sont délivrés au processus correspond à leur ordre d'émission, à l'exception des umsignaux qui sont délivrés lorsqu'ils sont débloqués.
- Un umsignal s'accompagne d'un message, qui sera passé en argument au handler. Ce message est une chaîne de caractères de taille variable limitée.
- Un processus peut recevoir des umsignaux provenant de processus du même utilisateur, mais aussi de processus ayant les mêmes privilèges de groupe, ou encore de processus sans privilège particulier. Il peut contrôler qui (utilisateur, groupe, autres) a le droit de lui envoyer quel umsignal. L'implémentation sur la gestion des droits doit être sécurisée.

Chapter 2

Gestion de projet

2.1 Organisation de l'équipe

- Thomas Tychensky (chef de projet)
- David Brochoire
- Nicolas Vigier

2.2 Outils de développement

2.2.1 SubVersion

SubVersion est un outil permettant de gérer l'évolution dans le temps d'un ensemble de fichiers. Il se veut le remplaçant de CVS. Ce dernier est très connu mais connait des manques de fonctionnalité que SubVersion souhaite combler. SVN permet d'extraire des sources, de les modifier, de soumettre ses modifications, de garder l'historique des modifications et de restaurer n'importe quelle version précédente. Mais l'atout majeur de SVN est de permettre à plusieurs développeurs de travailler sur le même groupe de fichiers : chacun travaille de manière indépendante dans son environnement personnel et SVN gère l'ensemble.

Le serveur SubVersion du projet est hébergé sur le domaine "wober.mars-attacks.org". Chacun des membres du groupe dispose d'un login et d'un mot de passe pour pouvoir accéder aux sources via SSH.

2.2.2 Éditeur et compilateur

Tout ce projet a été développé avec l'éditeur emacs et le compilateur C de GNU (gcc), tous les deux disponibles sur le PIE (Parc Informatique de l'EPITA). La

taille du code source à fournir n'étant pas très importante, l'utilisation d'outils plus imposant ne s'est pas fait ressentir.

2.2.3 Plateforme de développement

Une des contraintes de ce projet est qu'il doit au moins fonctionner sur NetBSD 1.6.1 (les machines de l'école) et sur Linux 2.x (les machines du lab SRS). En vue de porter la bibliothéque sur d'autres architectures, nous avons travaillé chacun sur différentes architectures tout au long du projet.
Ainsi parmis celles disponibles et en dehors des architectures de base qui doivent être supportées, nous avons développé sur FreeBSD 5.x et Mac OS X 10.3.

Chapter 3

Développement

3.1 Architecture et fonctionnement de umsignal

La conception de l´architecture des umsignaux pris une place importante dans le projet. Ceci est du au fait, que l´on emet des principes de fonctionnement qui ne sont pas toujours implémentable sur les plateformes de développement visées.
La conception s´est déroulé en plusieurs étapes. Nous avons réalisé une première architecture en groupe en ne prenant en compte que les besoins définis dans le sujet. Par la suite nous avons énuméré les éventuelles problèmes d´implémentation ou de fonctionnement (problèmes de sécurité, etc). Nous avons ensuite corrigé l´architecture et nous nous sommes fixés certaines limites dans notre implémentation. Deux autres architectures ont suivis la première pour aboutir à une architecture finale proche de la première.
Les points suivants reprennent cela en détaillant chaque étape.

3.1.1 Premier choix d´architecture: envoi par socket unix

Choix

Dû au fait que les signaux unix ne peuvent pas être envoyé entre processus tournant sous un utilisateur différent nous avons de suite écarté ce mode de communication car trop limité.
Nous avons donc pensé au pipe nommé, ou eventuellement au socket unix. La communication par pipe nommé aurait nécessité l´ouverture de 2 pipes et nous avons repéré des problemes de gestion d´envoi de signaux du à la sorte de multiplexage des informations réalisé par le pipe (pas de distinction claire des envoyeurs de signaux, pour une eventuelle gestion de signaux incorrect). Il nous restait donc les sockets unix qui repondait à tout les problèmes (un socket par envoyeur / emetteur de signaux).

Au niveau du fonctionnement, un processus qui veut recevoir crée un socket unix à un endroit precis du système avec un nom se réferant à son pid. Le socket est alors accessibles à tous les processus voulant lui envoyer des signaux. Ceci suppose donc l´existence d´un repertoire ayant des droits permettant à tous les signaux de créer leur socket.
Pour éviter de bloquer le processus récepteur de signaux, nous avons pensé à le faire forker afin de laisser son fils gérer les attentes du à la reception de signaux. La communication entre le fils et le père se faisant au moyen de 2 pipes et de signaux. Dés que le fils a reçu un umsignal, il interrompt son père avec un signal (SIGUSR1) et lui envoie les données sur le pipe. Le fils envoie l´aquittement nécessaire à l´envoyeur. Le fils peut gérer plusieurs réception en paralelle en faisant un select sur les différents sockets des envoyeurs.
Pour envoyer un signal il suffit de parcourir le répertoire ou se situe les sockets afin de trouver le socket correspondant au pid recherché. On ouvre le socket et on lui transmet les informations en attendant l´acquittement.

Problèmes

Les problèmes avec cette modélisation sont nombreux.

Problèmes de droit Nous avons relevé des problèmes de droit à différents endroits.
Le processus de création suppose l´existence d´un répertoire avec des droits assez permissifs. Un système ne peut peut-être pas se permettre de réaliser ce genre de répertoire et garantir une disponibilité pour la création d´autres socket. Il existe un risque de flood de l endroit ou le socket doit se créer. Ceci entraine un risque de création de fichier arbitraire

Problème d´identification D´autre part ceci peut entrainer un problèmes d´identification, lors de la recherche du socket correspondant à un pid voulu. Un processus usurpateur peut essayer de se faire passer en changeant le nom de socket afin de se faire passer pour un autre processus, ce qui peut bloquer completement l´utilisation des umsignaux.
Le problème peut se poser si les processus ne détruisent pas leur socket lorsqu´il termine leur execution.

Impossibilité de gérer le pgid Notre architecture révèle une de ces limites car il n´est pas possible de gérer le pgid avec la modélisation actuelle a moins de tenter l´envoi sur tous les sockets disponibles.

3.1.2 Second choix d´architecture: le démon

Ayant réalisé les problèmes qui pouvaient subvenir avec cette modélisation, nous avons cherché comment la complémenter et l'améliorer. Pour cela nous avons pensé à un démon.

Choix

Le démon se charge de relayer l'information entre processus. Lorsque que l'un d'entre eux décide d'utiliser notre api, il commence par ouvrir une socket unix qui servira à la réception d'umsignaux dans un path quelconque. Ensuite, il se connecte de manière transparente au démon via une autre socket unix et lui envoie le path de la première socket qu'il a ouvert. Le démon stocke par la même occasion le pid, pgid et uid de ce processus qui vient de le contacter.

Quand un processus décide d'envoyer un umsignaux, ne connaissant pas le path de le socket à ouvrir, il commence par contacter le démon en lui donnant le pid qu'il cherche à contacter, celui-ci lui donne en réponse le path de la socket.

Ensuite le processus d'échange d'umsignaux est le même que précédement.

Cette méthode a pour avantage de résoudre les problèmes de création de socket déjà utilisée, car le path de celles-ci peut-être complétement aléatoire. Il permet aussi de gérer la connexion vers un pgid au lieu d'un pid, pour cela le démon regarde la table des processus enregistrés chez lui, et renvoie la liste des sockets qui correspondent à ce pgid.

Problèmes

Comme toujours, rien n'est parfait, et cette modélisation possède aussi ses défauts. Tout d'abord le fait de tout centraliser par le démon enlève de la stabilité à l'architecture, car cela rend le démon fortement vulnèrable à des attaques par flood. Ou plus simplement, une erreur d'execution dans le démon rend l'api inutilisable pour tout le monde.

Ensuite le nombre d'ouverture de socket et de connexion est nettement plus important dans cette modélisation, ce qui gérère l'utilisation de ressources systèmes.

3.1.3 Dernier Choix d´architecture: le binaire suid

Nous avons finalement opté pour une troisième modélisation qui s'inspire des deux précedentes, en cherchant à simplifier le code à fournir et à améliorer les performances et la stabilité. Nous avons pour cela réussi à nous passer d'un démon.

Choix

Pour expliquer clairement notre choix, nous allons faire l'exemple de la configuration par défaut de l'api. Cela veut dire que nous nous retrouvons sur un os possédant un user ¨umsignal¨, un groupe ¨umsignal¨, un binaire ums_receive dans le $PATH, ce binaire étant suid selon le user ¨umsignal¨. Enfin il existe deux répertoires /var/lib/umsignal/pid et /var/lib/umsignal/pgid tous les deux appartenant à umsignal:umsignal avec un mask a 711.

Lorsqu'un processus commence à utiliser notre api, il commence par ouvrir deux pipes puis forke. Le processus fils execute le binaire suid dont nous avons parlé. Celui-ci créé le socket unix /var/lib/umsignal/pid/$pid_du_processus_pere ainsi qu´un lien symbolique /var/lib/umsignal/pgid/$pgid/$pid -> /var/lib/umsignal/pid/$pid.

Pour l'envoi de signaux, il suffit de connaitre le pid d'un processus pour en déduire le path de la socket à contacter. Et dans le cas d'un pgid, il suffit d'ouvrir toutes les sockets se trouvant dans le répertoire correspondant.

Pour gérer le problème de socket issu d'executions précédentes, le binaire suid commence par essayer de les effacer avant de la recréer.

Nous corrigeons le problème de path pouvant être pris auparavant par un processus mal intentionné car seul le user "umsignal" a le droit d'écrire dans les répertoires réservés.

Enfin, dans ce cas nous diminuons le nombre d'ouverture de sockets par rapport au démon car le path de celles-ci peut être déterminé sans connaissance extérieure.

Limites

Nous connaissons pour l'instant une limite à ce système, si jamais le processus en attente de signaux change de pgid après avoir fait le premier appel à notre api, nous ne changeons jamais la socket référencée dans le répertoire pgid ce qui rend le processus joignable que par son pid ou son ancien pgid.

3.2 Les choix d'implémentation

3.2.1 L´envoi des umsignaux

L'envoi des umsignaux est une partie assez simple à implémenter du fait de notre modélisation. Pour trouver un destinataire unique, il suffit d'aller chercher la socket portant le nom du pid du processus destinataire dans le répertoire réservé à ça. Ensuite nous ouvrons notre socket unix en mode connecté, et nous envoyons les différentes informations qui sont l'umsignal, la longueur du message associé et les données de ce message.

Par exemple, dans la configuration par défaut, pour contacter le processus de pid 39, nous faisons un connect(¨/var/lib/umsignal/pid/39¨) pour lui envoyer les données.

Ensuite nous nous mettons en attente d'un acquittement qui nous précisera si le signal a bien été reçu ou si des problèmes de droit se sont posés.

Dans le cas d'un envoi groupé à un pgid, il suffit de parcourir toutes les sockets ouvertes dans le répertoire correspondant (ex: /var/lib/umsignal/pgid/$pgid/*).

3.2.2 La réception des umsignaux

La reception des umsignaux se fait donc sur un socket unix, placé à un endroit precis du systeme de fichier. Lorsqu'un umsignal est envoyé sur le socket, le programme destinataire doit lire sur son socket les informations sur cet umsignal. Mais les umsignaux etants un moyen de communications asynchrone entre les processus, le programme qui recoit les signaux doit pouvoir etre interompu dans une de ses tache, sans avoir besoin d'aller lire regulierement si de nouvelles donnees sont presentes dans son socket de reception. Nous avons donc pris la decision de forker notre processus lors du premier appel a la fonction de definition d'un umsignal. Le processus forké est alors chargé de créer le socket unix en ecoute, puis d'ecouter sur ce socket afin de receptionner les umsignaux qui y seront envoyés. Ce processus passe donc la plus grande partie de son temps en attente dans un select. Le socket en ecoute est (par defaut) le fichier /var/lib/umsignal/pid/PID (PID devant etre remplace par le pid pere du processus créant le socket). L'envoi des umsignaux étant possible pour un process group complet, un repertoir au nom du process group est crée et contient des liens vers les sockets correspondants au pid membres de ce process group.

Pour eviter que n'importe qui ai la possibilité de créer n'importe quel socket dans le repertoire de facon par exemple a recevoir les signaux pour un autre processus, ce repertoir n'est pas en ecriture sauf pour son proprietaire qui est le compte umsignal. Comme vu plus haut, la partie du programme chargee de creer le socket puis de receptionner les umsignaux qui arrivent dessus est en fait un programme suid appartenant au compte umsignal. Une fois le socket crée (le pid du père étant récupéré par getppid()), le programme de réception n'a plus besoin des droits du compte umsignal et s'en debarrasse donc en faisaint appel à setuid(getuid()).

Le processus forké est donc pret à recevoir les signaux qui lui sont envoyé sur son socket. Lorsqu'il recoit un umsignal, il doit tout d'abord verifier les permitions afin de verifier si l'envoi à son père est autorisé. Pour cela il doit tenir à jour une table des droits de chacun des signaux. Lorsque dans le processus pere un appel est fait a la fonction de modification des droits pour un umsignal, il faut que cette modification soit communiquée au fils. Cela se fait en utilisant un pipe entre les 2 processus. Le processus fils passant son

temps à attendre dans un select, il nous a suffit de rajouter le file descriptor correspondant au pipe dans ce select. De cette facon le processus fils est avertit automatiquement de l'arrivée d'une definition de nouveaux droits sur le pipe, et peu donc les appliquer rapidement. Pour verifier les droits, le processus doit connaitre l'identitée du processus qui se connecte au socket. Sur les OS possèdant l'appel système getpeereid ou un equivalent (FreeBSD, OpenBSD, Linux), cette verification est possible (nous ne l'avons neamoins tésté avec notre programme à l'heure actuelle que sur Linux). Pour les autres, une verification de droits n'est pas possible. Sur certains OS tels que NetBSD, une authentification particulière est possible. Nous avons tenté de la mettre en oeuvre, mais cela ne fonctionnait qu'aléatoirement, et nous avons donc préféré (par manque de temps pour trouver d'ou venait le probleme) désactiver la gestion des droits sur NetBSD.

Lorsque les droits pour un signal recu sont corrects, il faut que l'umsignal soit ensuite retransmis au processus père. Cela se fait à l'aide d'un 2ème pipe. Mais le processus père n'étant pas forcement en attente sur le pipe lui permettant de recevoir les umsignaux, il faut le prevenir que quelquechose est arrivé pour lui. Nous lui envoyons donc le signal USR1, qui va interrompre le processus père. La fonction de gestion du signal USR1 chez le processus père est une fonction qui va lire sur le pipe afin de récuperer l'umsignal qui lui est déstiné, puis le traiter si un handler pour ce signal a été definit.

3.2.3 La gestion des umsignaux

La gestion des signaux comprend, la gestion des handlers et des droits, l'éxecution des handlers associés aux signaux reçus ainsi que la récupération des umsignaux depuis le fils.
La gestion des handlers se fait à l´aide d´une structure dans une liste. Lors d´une modification dans la liste (setmode. sethandler, sethandlermode) les signaux sont bloqués (surtout SIGUSR1) pour eviter toute sollication externe de la liste alors qu´elle est en cours de modification.
La gestion des signaux en attente se fait dans une liste. Suffisemmnet dínformation sont gardé en mémoire afin d´executer handler au on moment lors de déblocage de signaux et de réception en même temps.
Les droits sur chaque umsignal sont stockés chez le processus père de même que le stockage des handler et leur execution, mais la vérification des droits s´effectue chez le processus fils afin de ne pas déreanger le père inutilement. Les droits sont envoyés au fils via le pipe. Les modifications de droit deviennent donc valide lors que le fils en a pris connaissance.

Conclusion

En résumé, nous avons trouvé le projet umsignal plus abordable que le sujet précédent de notre spécialisation SRS (cf : alink). De plus, la part de conception y était plus importante et c'est une chose qui nous a intéressé et motivé. Ce point est d'ailleurs visible avec les différentes modélisations par lesquelles nous sommes passés.

Sans avoir réussi à implémenter toutes les fonctionnalités, notamment la sécurisation des connexions qui a posé des problèmes de portage, nous sommes assez satisfait de notre modélistion finale qui ne nécessite pas de démon et qui nous semble moins contraignante.