24 août 2008

Découvrir UML, ou comment mettre des Linux dans son Linux

Catégorie : Administration système Tags : lmhs

~~Retrouvez cet article dans :~~ Linux Magazine Hors série 17

Il n'est pas rare de vouloir compiler et tester les derniers noyaux sortis. Toutefois, il est difficile de faire confiance d'emblée à de tels noyaux, et faire tourner sa machine dessus peut paraître hasardeux. En outre, recompiler et rebooter à chaque nouvelle modification peut vite devenir pénible. User Mode Linux (UML)[1] offre une alternative à tous ces problèmes.

UML est un projet libre qui permet de faire tourner un Linux dans une machine virtuelle depuis n'importe quelle distribution Linux. Dans les nouveaux noyaux 2.6, UML fait même parti intégrante des sources du noyau (arch/um). Nous allons donc présenter ici comment tester son nouveau noyau avec UML et quelles utilisations sont possibles.

Compiler son UML pour un noyau 2.6

Le noyau 2.6 étant encore en phase de tests, certains bogues persistent encore, notamment en ce qui concerne l'architecture UML. On se basera tout au long de cet article plus particulièrement sur un noyau 2.6-test3 et on verra les problèmes encore existants et comment y faire face. Nul doute que lors de la sortie du noyau, ces problèmes seront résolus. On remarquera également que l'intérêt d'UML est également de pouvoir être utilisé par n'importe quel utilisateur sans privilège particulier, et que par conséquent il n'est pas nécessaire (donc inutile :), de faire les manipulations suivantes en tant que ~~root~~. Les compilations ont été effectuées avec gcc-3.2 (le 3-3 semblant poser quelques problèmes).
La première étape consiste à décompresser son noyau :

```
tom@hote:tar -zxf linux-2.6.0-test3.tar.gz
```

En se plaçant dans le répertoire des sources, on remarque alors la présence du répertoire arch/um qui décrit l'architecture UML. Pour préciser donc que c'est à cette architecture qu'on s'intéresse, il faudra ajouter ARCH=um pour chaque étape de la compilation. Par exemple, on peut configurer les options de son noyau grâce à la commande :

```
tom@hote:make menuconfig ARCH=um
```

Il est recommandé d'appliquer le dernier patch UML disponible sur le site aux sources du kernel, celles-ci n'ayant pas l'air d'avoir pris en compte toutes les modifications. On applique alors ce patch grâce à la commande :
tom@hote:bzcat uml-patch-2.6.0-test3-1.bz2 | patch -p1
La partie réseau comporte encore un bogue dans le noyau 2.6-test3 qu'on va tout de suite corriger. Dans le fichier ~~arch/um/drivers/net_kern.c~~, il suffit de commenter la ligne ~~dev->hard_header = uml_net_hard_header;~~ (ligne 339). En effet, sans cette modification, les headers des paquets réseaux ne seront pas complets et il ne sera donc pas possible de communiquer avec d'autres machines.
Il ne nous reste alors plus qu'à compiler en sélectionnant les différentes options que l'on souhaite tester. Il faut cependant remarquer que certains composants ne marchent pas : le support pour les modules (CONFIG_MODULES), SCSI et MTD (MTD_BLKMTD). Il ne faut donc pas choisir ces composants, sinon le noyau ne pourra pas être compilé.
Sachant tout ca, il nous est maintenant possible de compiler notre noyau sans difficulté. Tout d'abord, on nettoie nos sources :

```
tom@hote:make mrproper
```

(si on oublie cette commande, le linkage final risque de ne pas marcher après plusieurs compilations). On prend les options par défaut :

```
tom@hote:make defconfig ARCH=um
```

Toutefois, certains des modules qui ne fonctionnent pas sont sélectionnés. Il suffit de les commenter dans le .config (CONFIG_MODULES=y, CONFIG_SCSI=y, CONFIG_MTD_BLKMTD=m), ou de les modifier avec un ~~make menuconfig ARCH=um~~.
La phase de compilation proprement dite peut alors être lancée grâce à la commande :

```
tom@hote:make linux ARCH=um
```

Après un petit moment d'attente, on obtient l'exécutable linux.

Lancement

Tout d'abord il faut récupérer un rootfs. Plusieurs sont disponibles sur le site de UML. Nous avons choisi Debian-3.0r0.ext2. Pour lancer UML, il suffit de taper :

```
tom@hote:./linux root=/dev/ubd/0 ubd0=Debian-3.0r0.ext2 devfs=mount
```

(on remarquera le ~~root=/dev/ubd/0~~ qu'on peut remplacer par root=6200, obligatoire sur la ligne, contrairement aux versions précédentes, et qui évitera une erreur au lancement : Kernel panic: VFS: Unable to mount root fs on unknown-block(0,0)). Ça y est, on a enfin un UML qui fonctionne :)

```
tom@uml:uname -a
Linux uml 2.6.0-test3-1um #1 Fri Sep 26 18:35:56 CEST 2003 i686 unknown
```

Réseau

Pour lancer le réseau, il suffit d'ajouter à la ligne de lancement de l'UML :
~~eth0=tuntap,,,192.168.0.2~~. Il est également nécessaire d'avoir les ~~uml_utilities d'installées~~, et que le binaire ~~uml_net~~ se trouve bien dans le path et ait les bonnes permissions.
Le réseau se configure ensuite comme sous n'importe quel Linux :

```
tom@uml:ifconfig eth0 192.168.0.3 up
```

Les 2 machines peuvent alors communiquer : (l'hôte a pour IP 192.168.0.4)

```
tom@uml:ping 192.168.0.4
PING 192.168.0.4 (192.168.0.4): 56 data bytes
64 bytes from 192.168.0.4: icmp_seq=0 ttl=64 time=0.9 ms
64 bytes from 192.168.0.4: icmp_seq=1 ttl=64 time=0.6 ms
```

Par défaut, l'hôte joue le rôle de proxy arp pour l'UML, mais on peut bien entendu ensuite le configurer pour qu'il serve de passerelle ou de bridge.

Utiliser UML

UML est un simple programme, exécutable par un simple utilisateur sans privilèges.
Une fois compilé, on a l'exécutable linux. Celui-ci possède plusieurs arguments possibles, dont voici quelques-uns :

- **root** : comme pour un vrai noyau, on indique sur quelle partition on va monter la racine. /dev/ubd0 correspond au majeur 98 (62 en Hexa) et au mineur 0 (00 en Hexa) ; donc on peut l'écrire root=6200 ;
- **ubd0** : indique le filesystem à monter comme racine, si on a choisi /dev/ubd0 dans l'option précédente ;
- **debug** : pour déboguer le noyau UML ;
- **devfs=mount** : pour monter une partition virtuelle contenant des devices par défaut contenant par exemple /dev/ubd0 ;
- **umid** : on donne un nom à chaque système UML pour les différencier si on en élève plusieurs ;
- **mode=tt** : si le mode skas est présent sur l'hôte mais qu'on ne veut pas l'utiliser ;
- **jail** : pour activer la protection mémoire du mode jail ;
- **honeypot** : pour permettre à certains exploits de fonctionner et de faire croire qu'on serait bien sur un vrai noyau ;
- **mem** : pour indiquer la taille mémoire allouée au système UML ;
- **eth0** : pour configurer un élément réseau, les arguments indiquent à quoi est relié sur l'hôte cet élément.

Comment ça marche ?

Le fonctionnement habituel d'un processus sous Linux se fait par l'intermédiaire du noyau, qui est

le lien avec le matériel. L'idée avec User-Mode-Linux, c'est de placer le noyau en tant que simple programme. Les processus virtuels feront donc leurs appels systèmes vers ce programme émulant un noyau.
La différence avec un noyau habituel n'est pourtant pas très grande, puisque tout d'abord, on l'a vu, il se construit à partir des mêmes sources communes du noyau linux. La différence tient juste dans l'architecture. Avec UML on aura un matériel hardware émulé. Une fois compilé et lancé, UML se comporte donc comme un simple programme, qui fonctionne comme un noyau, mais qui n'a pas un réel accès à l'hardware. Etant lancé en mode utilisateur, cela permettra que ce qui s'y passe à l'intérieur n'ait pas de répercussions sur le matériel.
Ainsi le noyau UML, au lieu de s'adresser directement au matériel, s'adresse au noyau réel comme un programme normal (voir Figure 1).

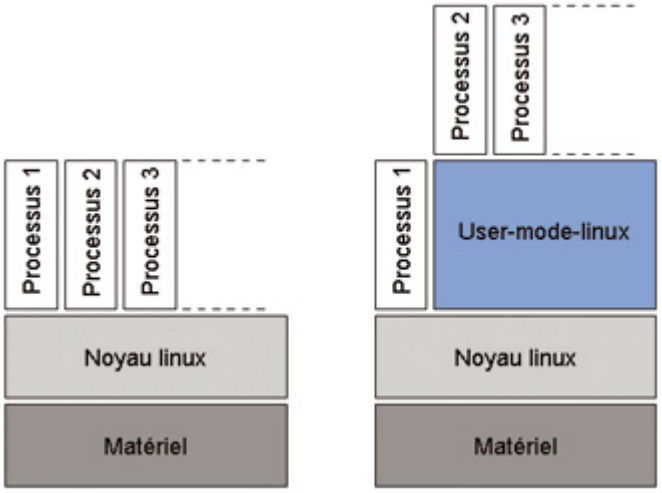

Figure 1 UML n'a pas de lien direct avec le matériel.

Linux supporte la surveillance des "Threads" en redirigeant les appels systèmes. UML se sert donc de cette fonctionnalité. Ainsi les appels systèmes des processus sous UML sont surveillés pour être traités par le noyau UML (voir Figure 2). UML fonctionne donc avec IPC, pour la communication inter-processus et se sert donc de l'appel système ptrace.

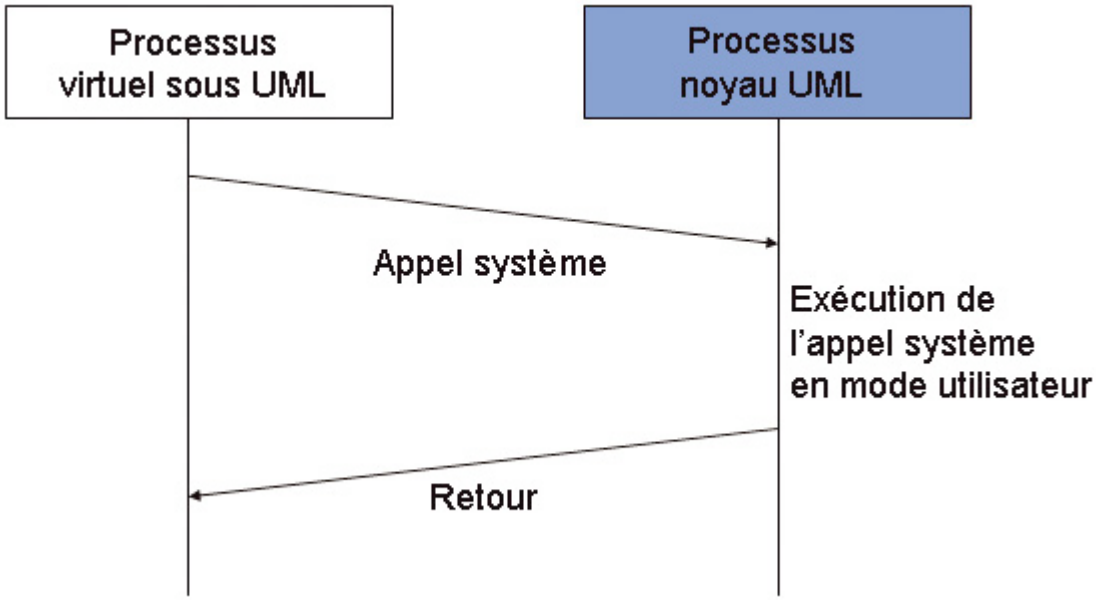

Figure 2 Interception des appels systèmes.

ptrace

Ptrace fournit au processus parent un moyen de contrôler l'exécution d'un autre processus et d'éditer son image mémoire. L'utilisation primordiale de cette fonction est l'implémentation de points d'arrêt pour le debugging. Un processus suivi se déroule jusqu'à l'arrivée d'un signal. Ensuite il s'interrompt, même si le signal est ignoré, et son père sera averti grâce à la fonction wait.
Il peut inspecter et modifier le processus fils pendant son arrêt. Le parent peut également faire continuer l'exécution de son fils. Quand le père a fini le suivi, il peut terminer le fils ou le faire continuer normalement.
Ainsi, UML peut prendre la place du vrai noyau.

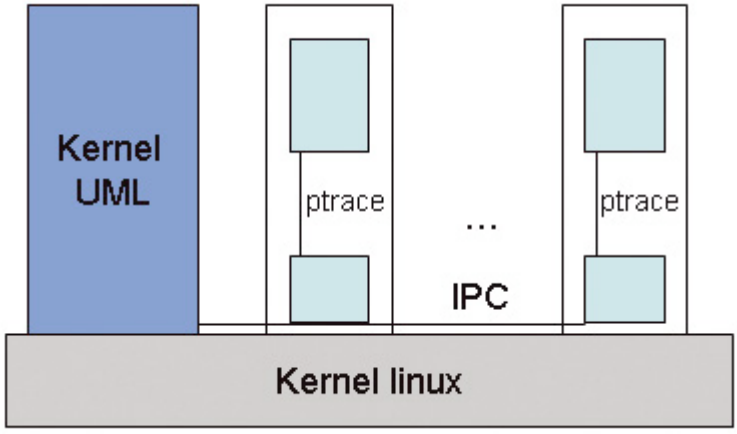

Figure 3 Interception des appels systèmes.

Deux modes d'exécution sont disponibles sous UML, le mode tt et le mode skas.

Le mode tt

C'est le mode par défaut. ~~tt~~ signifie Tracing Thread. Le fonctionnement d'UML dans ce mode est simple, chaque processus virtuel sous UML a un processus correspondant sur l'hôte. Il y a alors une Thread spéciale qui permet de relier les appels systèmes au processus UML noyau. On peut lister les processus sur l'hôte :

```
kl@hote:ps -f -C linux
UID        PID  PPID  C STIME TTY          TIME CMD
kl         309   305  1 15:51 ttyp1    00:00:04 linux  [(tracing thread)]
kl         314   309  0 15:51 ttyp1    00:00:00 linux  [(kernel thread)]
kl         321     1  0 15:51 ttyp1    00:00:00 linux  [(kernel thread)]
kl         323     1  0 15:51 ttyp1    00:00:00 linux  [(kernel thread)]
kl         325     1  0 15:51 ttyp1    00:00:00 linux  [(kernel thread)]
kl         327     1  0 15:51 ttyp1    00:00:00 linux  [(kernel thread)]
kl         333     1  0 15:51 ttyp1    00:00:00 linux  [(Unknown)]
kl         647     1  0 15:51 ttyp1    00:00:00 linux  [(Unknown)]
...
```

Remarque : Si on tue un de ces processus sur l'hôte, on peut s'attendre à un blocage du système UML. En effet, le processus noyau le surveillant attendra indéfiniment que celui-ci se manifeste. Le noyau UML est présent dans l'espace adresse de chacun de ses processus qui sont accessibles en écriture. C'est évidemment un problème de sécurité, puisque, avec l'accès en écriture aux données du kernel, un processus pourrait ainsi aller jusqu'à l'hôte. Le mode jail (ce mode est détaillé plus loin dans l'article) permet de résoudre ce problème, mais dans le cadre d'un honeypot, cela est visible.
De plus, l'aller/retour du signal est lent dans le cadre du mode tt.

Le mode skas

skas signifie Separate Kernel Address Space.
Pour utiliser ce mode, il faut patcher [5] le noyau de l'hôte.
Ensuite, au lancement d'UML, celui-ci détectera automatiquement la présence de l'hôte patché ou non. S'il ne trouve pas le support ~~skas~~ de l'hôte, il passera en mode ~~tt~~.
Le mode ~~skas~~ est plus sûr que le mode ~~tt~~ et il est aussi plus rapide. Pour le système UML vu de l'intérieur, il n'apporte aucun changement. Pour l'hôte, les différences sont que désormais il n'y a plus que 4 processus en tout par UML. Il permet donc de moins polluer en présence de processus l'hôte, surtout si on place plusieurs systèmes UML dessus.

```
tom@hote:ps -f -C linux
UID        PID  PPID  C STIME TTY          TIME CMD
tom       1469   578 43 15:15 pts/1    00:00:02 linux [(Unknown)]
tom       1471  1469  5 15:15 pts/1    00:00:00 [linux]
tom       1477  1469  1 15:15 pts/1    00:00:00 linux [(Unknown)]
tom       1478  1469  0 15:15 pts/1    00:00:00 linux [(Unknown)]
```

- **kernel thread**, qui fonctionne dans l'espace adresse séparé du noyau, exécute le code du noyau et fait l'interception des appels systèmes des processus UML.
- **userspace thread**, qui exécute tous les codes des processus UML, passe d'un contexte de processus UML à un autre.
- **ubd driver asynchronous IO thread** est le driver entrée/sortie d'ubd pour la gestion du

filesystem virtuel.
- **write SIGIO emulation thread** gère l'émulation pour l'écriture des signaux d'entrée/sortie.

Enfin, le mode skas permet de diminuer la charge lors de l'exécution de plusieurs systèmes UML.

A quoi ça sert ?

Nous allons présenter ici différentes utilisations d'UML. Celles-ci sont diverses, mais elles reposent toutes sur le principe voulant donner le moindre privilège à une utilisation quelconque.

Faire des tests

Vous voulez tester une nouvelle version du noyau Linux, une version encore en test ou patchée, ou votre version du noyau modifiée par vos soins.
Vous voulez tester une distribution, voir le comportement après un rm -R /, créer un système prenant le moins de place, un noyau le plus petit possible...
Autant de challenges ou de problèmes que vous voulez résoudre, avec la possibilité de faire ces tests facilement et de façon plus agréable (i.e. pas besoin de rebooter après chaque test et chaque kernel panic) et sans craindre pour son filesystem ou son matériel.
UML permet aussi de tester facilement des processus dangereux dans un système isolé.

Debugging

Puisque UML représente un noyau Linux et qu'il se comporte comme un simple programme, eh bien, on va pouvoir déboguer un noyau comme un programme habituel.
Pour cela, rien de plus simple, il suffit d'ajouter au lancement d'UML l'option ~~debug~~, ce qui permet d'avoir un accès à un terminal contenant ~~gdb~~, qui pourra suivre le fonctionnement du noyau virtuel. Ainsi, quand on voudra déboguer un noyau, on pourra penser à utiliser UML.
Pour la version d'UML du noyau 2.6.0, le noyau est automatiquement prêt pour le débogage. Si vous utilisiez une ancienne version d'UML, veillez à activer l'option CONFIG_DEBUGSYM au moment de la compilation (elle permet de compiler le noyau avec l'option -g de gcc).
Il faut signaler qu'un mécanisme de débogage a été ajouté à UML, étant donné que les processus sont déjà "ptracés" et que gdb ne pourrait donc le faire (il faut bien penser à activer l'option CONFIG_PT_PROXY lors de la compilation du noyau UML pour permettre à gdb de pouvoir lui même ptracer les processus).
Pour le mode skas, le débogage est sensiblement identique, hormis le fait que l'option debug n'est pas nécessaire. En effet, il suffit de lancer gdb sur le binaire d'UML.

```
kl@hote:gdb linux
```

Puis ensuite, on lance l'exécution et on inhibe la prise en compte des signaux SIGSEGV et SIGUSR1 qui sont à l'usage propre d'UML.

```
(gdb) r root=6200 ubd0=/uml/FileSystem/Debian-3.0r0.ext2 devfs=mount
(gdb) handle SIGSEGV pass nostop noprint
(gdb) handle SIGUSR1 pass nostop noprint
```

Ensuite, le débogage se fait comme à l'habitude avec les commandes att (pour attacher un programme au débogage), b (pour déposer un point d'arrêt), c (pour continuer l'exécution), n (pour

exécuter la ligne suivante)...

Jail

Tout le monde connaît chroot, qui permet de lancer un service dans une partie restreinte du système de fichiers. Toutefois, si une personne malveillante arrive à pirater un service "chrooté", il aura les droits de l'utilisateur qui a lancé le service. En outre, différentes techniques permettent de sortir de ce chroot.
UML propose une solution intéressante à ce problème. Comme nous l'avons dit, UML peut être lancé par un utilisateur quelconque du système. En le lançant par un utilisateur qui a un minimum de droit, on va ainsi limiter les risques, puisque même si un pirate arrive à devenir root sur notre UML, il ne sera finalement que l'utilisateur qui a lancé le processus sur la machine hôte.
En outre, UML propose une option sur sa ligne de commande : ~~jail~~. Ceci permet un meilleur contrôle de la mémoire, mais bien entendu ralentit l'exécution des processus. Pour utiliser cette option, il suffit de la préciser sur la ligne de commande. Toutefois, cela nécessite que certaines options du noyau soient désactivées, ce qui semble logique du point de vue de la sécurité :

```
'jail' may not used used in a kernel with CONFIG_SMP enabled
'jail' may not used used in a kernel with CONFIG_HOSTFS enabled
'jail' may not used used in a kernel with CONFIG_MODULES enabled
```

Pour utiliser cette option, on recompile donc le kernel en enlevant ces options. Il semble également logique d'utiliser le mode skas, plus sûr, d'UML. Pour ce faire, il suffit juste d'appliquer le patch correspondant au noyau de l'hôte :

```
tom@hote:patch -p1 <host-skas3.patch
```

Une fois tout cela fait, on peut démarrer un rootfs dans une ~~jail~~. Toutefois, pour faire une bonne prison, il est inutile de mettre des binaires qui ne servent à rien. La meilleure solution consiste alors à se construire soi-même son propre rootfs. Le plus simple est d'utiliser les loopdevices (il faut avoir CONFIG_BLK_DEV_LOOP=y pour le noyau de l'hôte).
On crée et monte alors simplement le filesystem ~~jail_apachefs.ext2~~ (10 Mo) :

```
tom@hote:dd if=/dev/zero of=jail_apachefs.ext2 seek=$[10*1024] count=1 bs=1k
tom@hote:losetup /dev/loop0 jail_apachefs.ext2
tom@hote:mkfs.ext2 /dev/loop0
tom@hote:mount -o loop /dev/loop0 /mnt/uml/
```

Il ne reste plus qu'à copier les fichiers nécessaires au lancement de notre Linux virtuel (un script de démarrage /sbin/init, les fichiers /etc/passwd, /etc/groups...), ainsi que les binaires et bibliothèques nécessaires (ldd le binaire pour connaître les bibliothèques utiles) au lancement de l'application que l'on souhaite sécuriser. Un exemple de jail DNS est disponible sur le site d'UML.

Copy On Write

UML propose autre chose de très intéressant : Copy On Write ou Cow. En effet, il peut être intéressant de lancer plusieurs UML différents avec des filesystems très voisins. Cow permet alors des gains de place importants. En effet, il est possible de lancer plusieurs UML simultanément et de travailler sur un filesystem identique, et de ne finalement sauvegarder que les modifications de chacun séparément.
Pour utiliser cette technique, il faut lancer l'UML en passant sur la ligne de commande

ubd0=fichier.cow,Debian-3.0r0.ext2. Le fichier fichier.cow sera alors créé et il contiendra les différentes modifications apportées. Lorsqu'on voudra le réutiliser, il suffira de passer sur la ligne de commande ubd0=fichier.cow, le chemin du rootfs de départ étant en fait écrit dans le fichier cow.

```
tom@hote:ls -lsh
 61M -rwxr-xr-x    1 tom      tom           60M 2003-09-26 17:37 Debian-3.0r0.ext2
572K -rw-r--r--    1 tom      tom           60M 2003-09-26 17:39 root.cow
```

Le fichier cow ne fait donc que 572K après quelques modifications, au lieu des 60 Mo du rootfs de départ. L'intérêt peut donc être énorme.
Le rootfs de départ constitue alors un fichier de backup et ne doit plus être relancé directement. Pour s'assurer de ne pas l'endommager, il parait intéressant de changer les droits qui lui sont associés :

```
tom@hote:chmod 444 Debian-3.0r0.ext2
```

Il se peut toutefois qu'une imprudence nous oblige à modifier ou recopier ce fichier de backup. L'UML ne veut alors plus se lancer : 0mtime mismatch (1064590679 vs 1064591338) of COW header vs backing file. Il suffit de corriger cela grâce à la commande :

```
mtime=1064590679 ; unset LC_ALL ;
touch --date="`date -d 1970-01-01\ UTC\ $mtime\ seconds`" Debian-3.0r0.ext2
```

Il est également possible de réassembler le fichier cow et le fichier de backup pour faire une sauvegarde par exemple. Ceci se fait avec la commande (uml_moo fait parti des uml_utilities) :

```
tom@hote:uml_moo root.cow Debian-3.0r0.ext2.backup
```

On peut alors ensuite relancer ce rootfs comme un rootfs normal.

Pour finir

L'avantage est avant tout que si UML a un problème, le système principal est encore en pleine santé. On peut donc très bien l'utiliser comme honeypot [6] avec tous les avantages de surveillance par l'hôte qu'il propose.

Les projets en cours

Aujourd'hui, les évolutions d'UML sont surtout au niveau du portage. D'une part, pour l'instant, UML permet d'émuler une machine de type i386 ; il pourrait être intéressant d'émuler pour différentes architectures.
Une autre direction concerne le portage au niveau de l'hôte. Il faudrait porter UML sur différents OS comme Unix ou même Windows.
D'autres projets concernent des améliorations d'hostfs (le filesystem permettant de remonter le / de l'hôte sous l'UML).
On pourrait aussi avoir un UML émulant un système multiprocesseur.
Un projet concerne la standardisation d'UML en tant que bibliothèque afin de permettre aux programmes qui y serait liés d'accéder à toutes les fonctionnalités du noyau.
Enfin, les projets relatifs à UML concernant toutes les utilisations possibles qu'on pourrait en faire.

Références

[1] http://user-mode-linux.sourceforge.net
[2] http://usermodelinux.org
[3] http://hints.linuxfromscratch.co.uk/hints/uml.txt
[4] http://www.netwinder.org/~tpot/nano-linux.html
[5] http://user-mode-linux.sourceforge.net/dl-sf.html#Host%20patches
[6] Misc 8, spécial Honeypot.

~~Retrouvez cet article dans :~~ Linux Magazine Hors série 17

Posté par (La rédaction) | Signature : Thomas Meurisse, Michaël Hervieux |

Laissez une réponse

Vous devez avoir ouvert une session pour écrire un commentaire.

« Précédent Aller au contenu »

Identifiez-vous
Inscription
S'abonner à UNIX Garden

Articles de 1ère page

- Inkscape 0.45 : plus rapide, plus facile !
- KTorrent : le client BitTorrent pour KDE
- La légalité du spam : une question ouverte
- Pear et les librairies PHP
- FreeDOS
- Mise en œuvre d'une passerelle Internet sous Linux
- Les pseudo-classes en CSS
- GNU/Linux Magazine N°108 - Septembre 2008 - Chez votre marchand de journaux
- Linux Pratique N°49 - Septembre/Octobre 2008 - Chez votre marchand de journaux
- Donner du style à son texte : utiliser les lettrines

Actuellement en kiosque :

- # Il y a actuellement

- 721 articles/billets en ligne.

Recherche

- # Catégories

- - Administration réseau
  - Administration système
  - Agenda-Interview
  - Audio-vidéo
  - Bureautique
  - Comprendre
  - Distribution
  - Embarqué
  - Environnement de bureau
  - Graphisme
  - Jeux
  - Matériel
  - News
  - Programmation
  - Réfléchir
  - Sécurité
  - Utilitaires
  - Web

- # Archives

- - septembre 2008
  - août 2008

- juillet 2008
- juin 2008
- mai 2008
- avril 2008
- mars 2008
- février 2008
- janvier 2008
- décembre 2007
- novembre 2007
- février 2007

GNU/Linux Magazine

- GNU/Linux Magazine 108 - Septembre 2008 - Chez votre marchand de journaux
- Edito : GNU/Linux Magazine 108
- GNU/Linux Magazine HS 38 - Septembre/Octobre 2008 - Chez votre marchand de journaux
- Edito : GNU/Linux Magazine HS 38
- GNU/Linux Magazine 107 - Juillet/Août 2008 - Chez votre marchand de journaux

GNU/Linux Pratique

- Linux Pratique N°49 -Septembre/Octobre 2008 - Chez votre marchand de journaux
- Edito : Linux Pratique N°49
- À télécharger : Les fichiers du Cahier Web de Linux Pratique n°49
- Linux Pratique Essentiel N°3 - Août/Septembre 2008 - Chez votre marchand de journaux
- Edito : Linux Pratique Essentiel N°3

MISC Magazine

- Misc 38 : Codes Malicieux, quoi de neuf ? - Juillet/Août 2008 - Chez votre marchand de journaux
- Edito : Misc 38
- Références de l'article « Détection de malware par analyse système » d'Arnaud Pilon paru dans MISC 38
- Références de l'article « La sécurité des communications vocales (3) : techniques numériques » d'Éric Filiol paru dans MISC 38
- Misc 37 : Déni de service - Mai/Juin 2008 - Chez votre marchand de journaux