TP3-UTC502

Gestion des volumes et sauvegarde

Groupe 1 :

BIGAUD Emilie

DOUÇOT Kévin

LE COZ Julien

PARIS Loic

SARAHANABHAVAN Saranya

Manipulation 1 : Gestion des modules du noyau

1. Les modules

Afin de lister les modules en cours d'utilisation, nous utilisons la commande *lsmod*. Cela affiche également la mémoire utilisée qui est consultable sous /proc/modules :

```
┌──(root㉿Kali)-[/home/user]
└─# lsmod
Module                  Size  Used by
rfkill                 36864  2
qrtr                   49152  4
vboxsf                 45056  0
sunrpc                692224  1
binfmt_misc            24576  1
joydev                 28672  0
snd_intel8x0           49152  2
intel_rapl_msr         20480  0
hid_generic            16384  0
intel_rapl_common      32768  1 intel_rapl_msr
snd_ac97_codec        176128  1 snd_intel8x0
usbhid                 65536  0
ac97_bus               16384  1 snd_ac97_codec
```

Pour trouver et lister les noms des modules qui prennent en charge la carte réseau et carte son, nous utilisons la commande *lspci* avec l'option *-k*. Cette option permet d'afficher les pilotes noyau gérant chaque périphérique. Pour affiner la recherche, on rajoute *grep* puis *-A* pour afficher les x lignes suivant après le mot recherché :

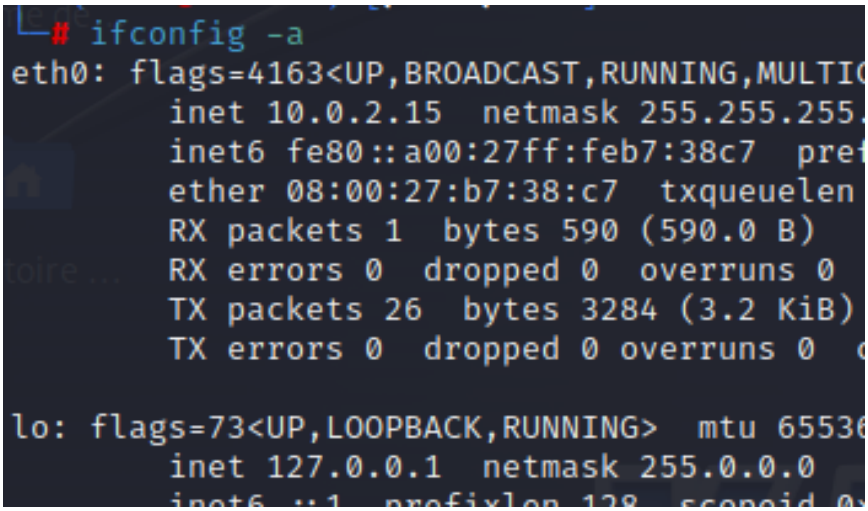
```
└─# lspci -v | grep -i ether -A 8
00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Contr
oller (rev 02)
        Subsystem: Intel Corporation PRO/1000 MT Desktop Adapter
        Flags: bus master, 66MHz, medium devsel, latency 64, IRQ 19
        Memory at f0200000 (32-bit, non-prefetchable) [size=128K]
        I/O ports at d020 [size=8]
        Capabilities: [dc] Power Management version 2
        Capabilities: [e4] PCI-X non-bridge device
        Kernel driver in use: e1000
        Kernel modules: e1000
```

```
└─# lspci -v | grep -i audio -A 5
00:05.0 Multimedia audio controller: Intel Corporation 82801AA AC'97 Audio Co
ntroller (rev 01)
        Subsystem: Dell 82801AA AC'97 Audio Controller
        Flags: bus master, medium devsel, latency 64, IRQ 21
        I/O ports at d100 [size=256]
        I/O ports at d200 [size=64]
        Kernel driver in use: snd_intel8x0
        Kernel modules: snd_intel8x0
```

Si l'option -A n'est pas mis, il y aurait seulement les lignes avec le mot recherché d'affiché.

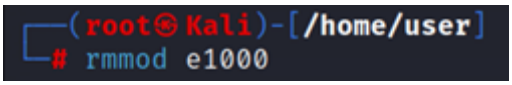
2. Module Ethernet

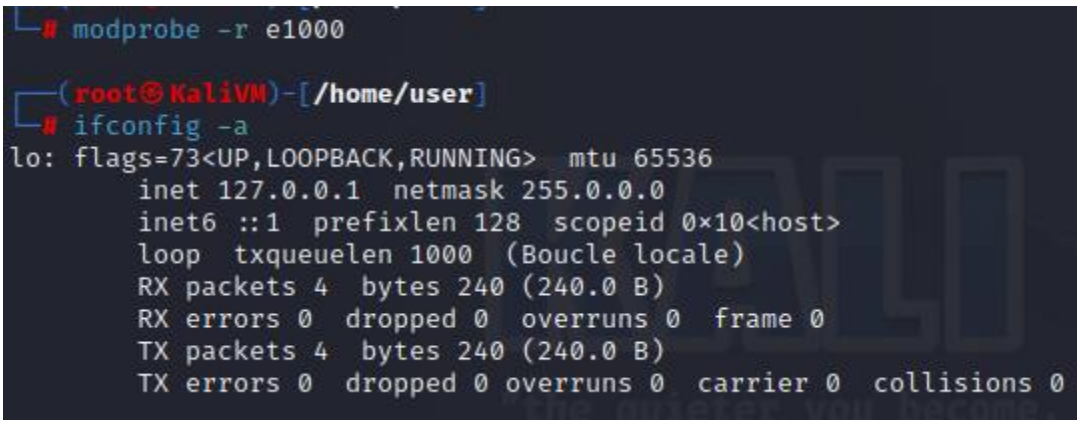
Pour lister les interfaces réseaux, nous utilisons la commande *ifconfig* avec l'option *-a*, qui permet de tout afficher, y compris les interfaces désactivées :



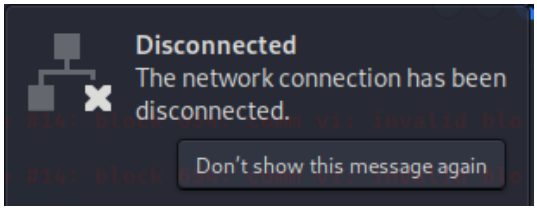
Nous pouvons voir qu'il y a 2 interfaces : *l'eth0* qui permet de communiquer avec les autres machines du réseau ou d'aller vers l'extérieur avec une gateway. La *lo* qui correspond à la loopback.

Pour retirer le module ethernet, nous avons 2 commande possible *rmmod* et *modprobe* avec l'option *-r*. Il faut renseigner le numéro du module associé trouvé précédemment, ici e1000 :





Les deux commandes mènent au même résultat, la suppression de eth0, seul l'interface loopback est présente. Un message pop-up nous indique que nous sommes déconnectés du réseau, la navigation sur internet ou vers d'autres PC est impossible :



Pour remonter le module eth0, nous utilisons la commande *modprobe* suivi du numéro du module :

```
┌──(root㉿Kali)-[/home/user]
└─# modprobe e1000
```

```
┌──(root㉿Kali)-[/home/user]
└─# ifconfig -a
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
        inet6 fe80::a00:27ff:fece:59c2  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:ce:59:c2  txqueuelen 1000  (Ethernet)
        RX packets 1  bytes 590 (590.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 11  bytes 1080 (1.0 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Boucle locale)
        RX packets 4  bytes 240 (240.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4  bytes 240 (240.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Lorsque nous retirons la module eth0 il n'y a pas de log qui apparait. En remettant le module, nous avons les logs suivants :

```
[ 9584.879405] e1000: Intel(R) PRO/1000 Network Driver
[ 9584.879408] e1000: Copyright (c) 1999-2006 Intel Corporation.
[ 9585.240367] e1000 0000:00:03.0 eth0: (PCI:33MHz:32-bit) 08:00:27:ce:59:c2
[ 9585.240382] e1000 0000:00:03.0 eth0: Intel(R) PRO/1000 Network Connection
[ 9587.328637] e1000: eth0 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: RX
[ 9587.328903] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
```

3. Cœur du noyau

Afin de connaitre la taille du cœur, nous utilisons la commande *du* avec les options *-sh*. L'option *-h* qui sert à afficher de façon plus lisible en Ko, Mo etc... Sinon cela s'afficherai par block de 1024. L'option *-s* sert à compiler tous les répertoire et sous-répertoire à partir du dossier renseigné. Le répertoire cible est */boot/*nom_VM :

```
└─# du -sh /boot/vmlinuz-6.1.0-kali5-amd64
8,2M    /boot/vmlinuz-6.1.0-kali5-amd64
```

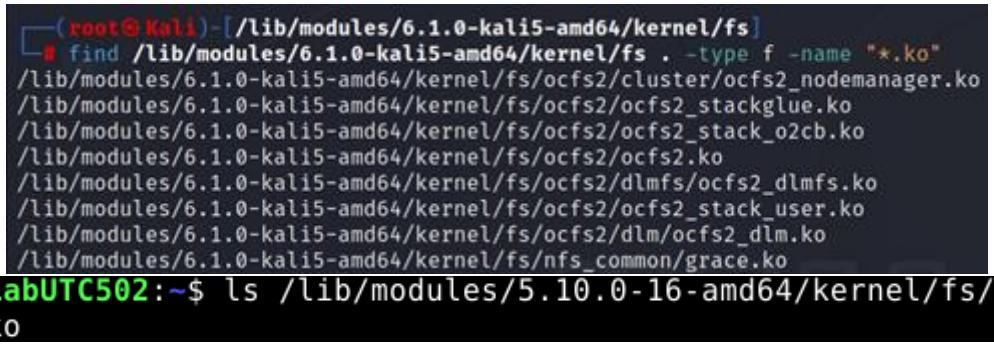
Pour afficher le nombre de module en cours, nous utilisons la commande *find* puis | *wc* avec l'option *-l* sur le même répertoire que précédement :

```
┌──(root㉿Kali)-[/boot]
└─# find /boot vmlinuz-6.1.0-kali5-amd64 | wc -l
353
```

Ici nous avons 353 modules présent. Cela dépend des VM, car sur une autre VM nous avions 4023 modules.

Pour afficher la taille de l'ensemble de module, nous nous positionnons dans le répertoire */lib*, puis nous ré-utilisons la commande *du* :

Afin de lister les modules présents dans l'arborescence ci-dessous, nous utilisons la commande *find* avec les options suivantes :



Le format ext3 a été remplacé par ext4 qui est plus moderne et adapté aux systèmes actuels. On voit que seul le module ext4 est présent dans le dossier contenant les modules associés aux systèmes de fichiers. Nous pouvons en déduite que le format ext3 peut également être inclus dans le cœur du noyau.

Synthèse

Afin de rendre les manipulations liées au noyau plus faciles et éviter d'avoir à le recompiler dès qu'il y a un changement de pilote, ou d'une autre fonctionnalité, les concepteurs du noyau l'ont divisée en modules. Ces modules sont facilement chargeables et retirables, et toutes les informations concernant leur arborescence est centralisée afin d'en simplifier les manipulations.

L'utilisation de la commande *modprobe* permet, contrairement à *rmmode* et *insmod*, de gérer les dépendances entre modules ainsi que de s'affranchir du chemin complet des modules.

Manipulation 2 : L'arborescence temporaire

1. Observation

Copie du fichier */boot/initrd/version_noyau* dans un dossier initrd, préalablement créé, avec la commande *cp*. Nous l'avons renommé avec la commande *mv* et décompressé avec *zcat* :

```
root@LabUTC502:/# mkdir initrd
root@LabUTC502:/# cd /initrd
root@LabUTC502:/initrd# cp /boot/initrd.img-5.10.0-16-amd64 .
root@LabUTC502:/initrd# mv initrd.img-5.10.0-16-amd64 initrd.cpio.gz
root@LabUTC502:/initrd# zcat initrd.cpio.gz | cpio -idv
.
bin
conf
conf/arch.conf
conf/conf.d
conf/conf.d/resume
conf/initramfs.conf
etc
etc/default
etc/default/keyboard
etc/fonts
etc/fonts/conf.d
etc/fonts/conf.d/60-latin.conf
```

Pour lister le nombre de module embarqué, nous regardons avec la commande *find* :

```
root@LabUTC502:/initrd# find . -iname "*ko"
./usr/lib/modules/5.10.0-16-amd64/misc/vboxvideo.ko
./usr/lib/modules/5.10.0-16-amd64/kernel/drivers/spi/spi-lm70llp.ko
./usr/lib/modules/5.10.0-16-amd64/kernel/drivers/spi/spi-butterfly.ko
./usr/lib/modules/5.10.0-16-amd64/kernel/drivers/spi/spi-bitbang.ko
./usr/lib/modules/5.10.0-16-amd64/kernel/drivers/spi/spi-pxa2xx-platform.ko
./usr/lib/modules/5.10.0-16-amd64/kernel/drivers/spi/spi-pxa2xx-pci.ko
./usr/lib/modules/5.10.0-16-amd64/kernel/drivers/firewire/firewire-ohci.ko
./usr/lib/modules/5.10.0-16-amd64/kernel/drivers/firewire/firewire-core.ko
./usr/lib/modules/5.10.0-16-amd64/kernel/drivers/firewire/firewire-sbp2.ko
./usr/lib/modules/5.10.0-16-amd64/kernel/drivers/infiniband/hw/mlx5/mlx5_ib.ko
./usr/lib/modules/5.10.0-16-amd64/kernel/drivers/infiniband/hw/mlx4/mlx4_ib.ko
./usr/lib/modules/5.10.0-16-amd64/kernel/drivers/infiniband/core/ib_core.ko
./usr/lib/modules/5.10.0-16-amd64/kernel/drivers/infiniband/core/ib_uverbs.ko
./usr/lib/modules/5.10.0-16-amd64/kernel/drivers/infiniband/core/rdma_cm.ko
./usr/lib/modules/5.10.0-16-amd64/kernel/drivers/infiniband/core/ib_cm.ko
./usr/lib/modules/5.10.0-16-amd64/kernel/drivers/infiniband/core/iw_cm.ko
./usr/lib/modules/5.10.0-16-amd64/kernel/drivers/ptp/ptp.ko
./usr/lib/modules/5.10.0-16-amd64/kernel/drivers/bus/mhi/core/mhi.ko
```

Nous observons qu'il y a un grand nombre de module, pour les compter le faire un à un n'est pas la meilleure solution. Pour afficher le nombre de module automatiquement, on ajout | *wc* avec l'option *-l* :

```
root@LabUTC502:/initrd# find . -iname "*ko" | wc -l
711
```

2. Création d'une nouvelle image de démarrage

Nous éditons le fichier */etc/initramfs-tools/initramfs.conf* et modifions le nombre maximal de module :

```
root@LabUTC502:/# vi /etc/initramfs-tools/initramfs.conf
#
# MODULES: [ most | netboot | dep | list ]
#
# most - Add most filesystem and all harddrive drivers.
#
# dep - Try and guess which modules to load.
#
# netboot - Add the base modules, network modules, but skip block devices.
#
# list - Only include modules from the 'additional modules' list
#

MODULES=dep
```

Pour que la modification soit prise en compte, mise à jour de l'image :

```
root@LabUTC502:/# update-initramfs -u
update-initramfs: Generating /boot/initrd.img-5.10.0-16-amd64
```

Nous vérifions que la modification à fonctionné. Nous remarquons que le nombre de module a grandement diminué :

```
root@LabUTC502:/initrd# find . -iname "*ko" | wc -l
27
```

Synthèse

Un driver est nécessaire pour lire la partition principale. Or ce driver peut être présent sur cette même partition. Dans ce cas il faut contourner ce paradoxe en utilisant une image *initrd* qui est une sorte de noyau de démarrage. Cette image contient des modules du noyau et peut être mise à jour grâce au fichier de configuration *initramfs.conf* et à la commande *update-initramfs*.

Tous les modules du noyau ne sont pas nécessaires pour que cette image soit fonctionnelle, mais certains sont incontournables pour que cette image puisse remplir son rôle.

Manipulation 3 : Recompilation du noyau

1. Préparation

Pour les besoins de cette manipulation nous installons l'ensemble des paquets renseigné sur le TP. La version linux source 3.2 est une ancienne version, nous installerons sur nos VM les version 6.1 pour certains et 5.10 pour d'autres :

```
└─# apt-get install linux-source-6.1
```

```
┌──(root㉿KaliVM)-[/usr/src]
└─# ls
linux-config-6.1  linux-source-6.1.tar.xz
```

Le paquet *kernel-package* n'existe pas en tant que tel. Nous installons un à un la liste cité ci-dessous :

```
└─# apt-get install kernel
Completing package
kernelshark   kerneltop     kernel-wedge
```

Nous copions le paquet *linux-source* dans le répertoire */home* :

```
┌──(root㉿KaliVM)-[/usr/src]
└─# cp linux-source-6.1.tar.xz /home
```

Avant de dézipper le fichier, il faut le décompressé comme le montre l'extension *.xz*. Nous utiliserons l'option *-d* pour forcer la décompression :

```
┌──(root㉿Kali)-[/home]
└─# xz -d /usr/src/linux-source-6.1.tar.xz
```

Une fois décompressé, nous le dézippons avec les options *-xvf* :

```
┌──(root㉿KaliVM)-[/home]
└─# tar -xvf linux-source-6.1.tar
linux-source-6.1/
linux-source-6.1/.clang-format
linux-source-6.1/.cocciconfig
linux-source-6.1/.get_maintainer.ignore
linux-source-6.1/.gitattributes
linux-source-6.1/.gitignore
linux-source-6.1/.mailmap
linux-source-6.1/.rustfmt.toml
linux-source-6.1/COPYING
linux-source-6.1/CREDITS
```

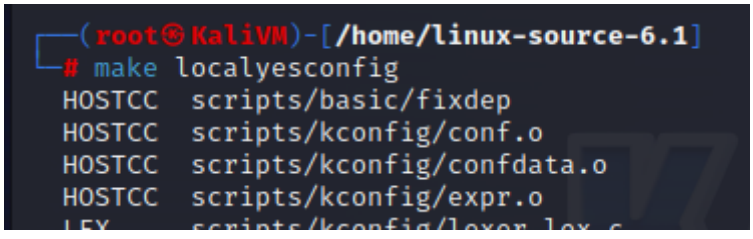
2. Configuration du noyau

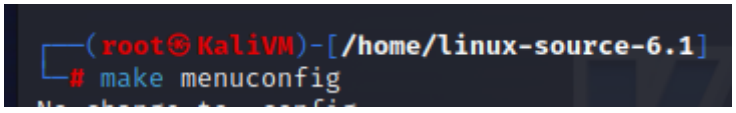
Nous copions le fichier de config se trouvant dans */boot* dans le répertoire */home*, que nous renommons en *saveconfig*, qui nous sert de fichier de sauvegarde :

Les différentes cibles permettant la configuration sont les suivantes à l'aide de la commande *make --help* :

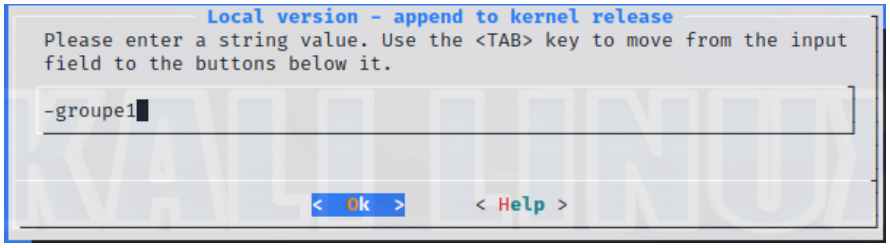
```
Configuration targets:
  config          - Update current config utilising a line-oriented program
  nconfig         - Update current config utilising a ncurses menu based program
  menuconfig      - Update current config utilising a menu based program
  xconfig         - Update current config utilising a Qt based front-end
  gconfig         - Update current config utilising a GTK+ based front-end
  oldconfig       - Update current config utilising a provided .config as base
  localmodconfig  - Update current config disabling modules not loaded
                    except those preserved by LMC_KEEP environment variable
  localyesconfig  - Update current config converting local mods to core
                    except those preserved by LMC_KEEP environment variable
  defconfig       - New config with default from ARCH supplied defconfig
  savedefconfig   - Save current config as ./defconfig (minimal config)
  allnoconfig     - New config where all options are answered with no
  allyesconfig    - New config where all options are accepted with yes
  allmodconfig    - New config selecting modules when possible
  alldefconfig    - New config with all symbols set to default
  randconfig      - New config with random answer to all options
  yes2modconfig   - Change answers from yes to mod if possible
  mod2yesconfig   - Change answers from mod to yes if possible
  listnewconfig   - List new options
  helpnewconfig   - List new options and help text
  olddefconfig    - Same as oldconfig but sets new symbols to their
                    default value without prompting
  tinyconfig      - Configure the tiniest possible kernel
  testconfig      - Run Kconfig unit tests (requires python3 and pytest)
```

Pour notre configuration nous utilisons les commandes *make localyesconfig* et *make menuconfig* :

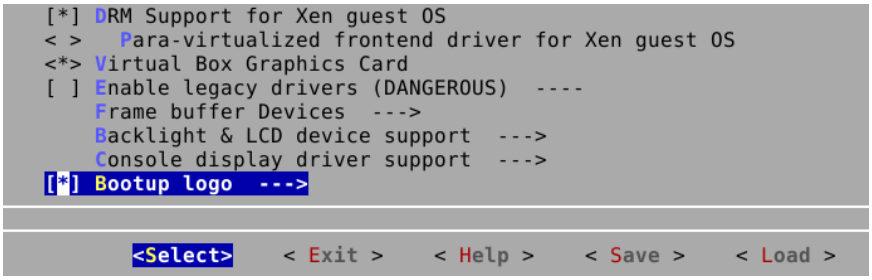


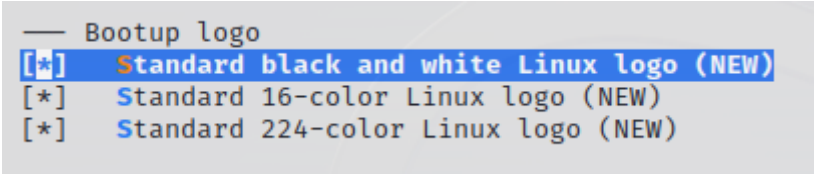


Nous configurons la version locale en renseignant notre groupe :

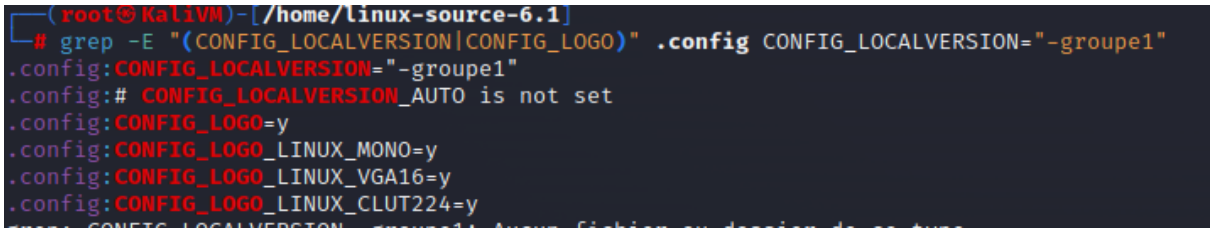


Puis nous configurons le bootup logo. Par défaut ça n'affiche rien, une fois sur le choix il faut appuyer sur la barre d'espace pour l'activer (* indique que c'est activé). A ce moment nous avons eu 3 choix. Même procédé pour activer ou non les choix qui s'offre à nous :





Nous vérifions que la configuration est prise en compte, ce qui est le cas :



Malgré avoir installé les diverses options de kernel ou *makepkg*, nous n'avons pas réussi à compiler le noyau, la commande *make-kpkg* ne fonctionne pas ni le sustitu *makepkg*.

Synthèse

Etant open source, les noyaux Linux sont disponibles. Plusieurs outils permettent de compiler ses sources et ainsi d'autoriser une personnalisation très fine de cet élément essentiel du système d'exploitation. Il est d'abord nécessaire de disposer du code du noyau, puis de configurer la compilation à venir. C'est une opération qui prend un certain temps et qui mobilise l'appareil mais l'administrateur a ainsi un contrôle total sur le fonctionnement de la machine.

Manipulation 4 : Compiler à partir des sources

1. Récupération source

Nous récupérons le code source *sthttpd* indiqué à l'adresse renqeigné sur le TP. Nous décompressons l'archive avec le commande *tar* et les options *-xvzf* :

```
└─# tar -xvzf sthttpd-2.26.4.tar.gz
sthttpd-2.26.4/
sthttpd-2.26.4/src/
sthttpd-2.26.4/src/thttpd.h
sthttpd-2.26.4/src/fdwatch.c
sthttpd-2.26.4/src/libhttpd.h
```

```
┌──(root㉿KaliVM)-[/media/user]
└─# ls
sthttpd-2.26.4  sthttpd-2.26.4.tar.gz

┌──(root㉿KaliVM)-[/media/user]
└─# ls sthttpd-2.26.4
aclocal.m4   configure.ac  extras       Makefile.in  scripts  www
config.h.in  depcomp       install-sh   missing      src
configure    docs          Makefile.am  README       TODO
```

2. Configuration

Nous créons le répertoire *thttpd-pkg* qui nous servira de répertoire de travail pour la manipulation :

```
user@LabUTC502:~$ cd sthttpd-2.26.4/
user@LabUTC502:~/sthttpd-2.26.4$ mkdir thttpd-pkg
user@LabUTC502:~/sthttpd-2.26.4$ WEBDIR="/home/user/sthttpd-2.26.4/thttpd-pkg/var/www/" WEBGROUP="root" ./configure --prefix=/home/user/sthttpd-2.26.4/thttpd-pkg/usr/
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /usr/bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
```

3. Compilation

Le répertoire créé et la configuration effectué, nous compilons le logiciel avec la commande *make* :

```
user@LabUTC502:~/sthttpd-2.26.4$ make
make  all-recursive
make[1]: Entering directory '/home/user/sthttpd-2.26.4'
Making all in src
make[2]: Entering directory '/home/user/sthttpd-2.26.4/src'
gcc -DHAVE_CONFIG_H -I. -I..     -g -O2 -MT match.o -MD -MP -MF .deps/match.Tpo -c -o match.o match.c
mv -f .deps/match.Tpo .deps/match.Po
rm -f libmatch.a
ar cru libmatch.a match.o
ar: `u' modifier ignored since `D' is the default (see `U')
ranlib libmatch.a
gcc -DHAVE_CONFIG_H -I. -I..     -g -O2 -MT thttpd.o -MD -MP -MF .deps/thttpd.Tpo -c -o thttpd.o thtt
```

4. Installation

Pour installer le logiciel précédemment préparé, et éviter que le script échoue, nous utilisons les commandes *fakeroot* et *make* :

```
user@LabUTC502:~/sthttpd-2.26.4$ fakeroot make install
Making install in src
make[1]: Entering directory '/home/user/sthttpd-2.26.4/src'
make[2]: Entering directory '/home/user/sthttpd-2.26.4/src'
test -z "/home/user/sthttpd-2.26.4/thttpd-pkg/usr/sbin" || /usr/bin/mkdir -p "/home/user/sthttpd-2.26.4/thttpd-pkg/usr/sbin"
  /usr/bin/install -c thttpd '/home/user/sthttpd-2.26.4/thttpd-pkg/usr/sbin'
make[2]: Nothing to be done for 'install-data-am'.
make[2]: Leaving directory '/home/user/sthttpd-2.26.4/src'
make[1]: Leaving directory '/home/user/sthttpd-2.26.4/src'
Making install in extras
make[1]: Entering directory '/home/user/sthttpd-2.26.4/extras'
make[2]: Entering directory '/home/user/sthttpd-2.26.4/extras'
```

Nous pouvons observer l'arborescence du répertoire *thttpd-pkg* avec la commande *tree* :

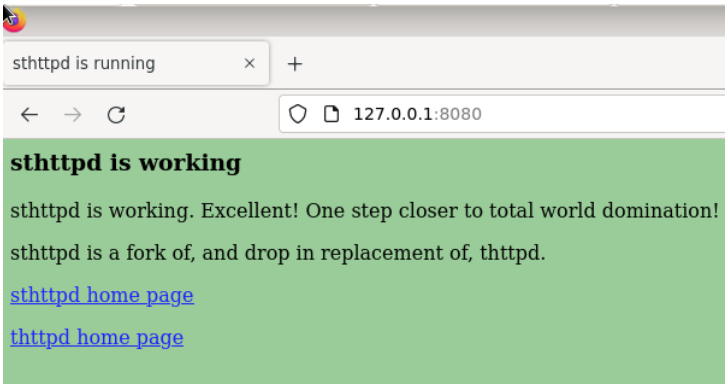
```
user@LabUTC502:~/sthttpd-2.26.4$ tree thttpd-pkg/
thttpd-pkg/
├── usr
│   ├── sbin
│   │   ├── makeweb
│   │   ├── syslogtocern
│   │   ├── th_htpasswd
│   │   └── thttpd
│   └── share
│       └── man
│           ├── man1
│           │   ├── makeweb.1
│           │   └── th_htpasswd.1
│           └── man8
│               ├── redirect.8
│               ├── ssi.8
│               ├── syslogtocern.8
│               └── thttpd.8
└── var
    └── www
        ├── cgi-bin
        │   ├── phf
        │   ├── printenv
        │   ├── redirect
        │   └── ssi
        └── index.html

9 directories, 15 files
```

5. Test du programme

Nous testons le programme en le lançant depuis le dossier *thttpd-pkg* :

```
user@LabUTC502:~/sthttpd-2.26.4$ cd thttpd-pkg/
user@LabUTC502:~/sthttpd-2.26.4/thttpd-pkg$ ./usr/sbin/thttpd -p 8080 -d ./var/www/
```

Nous vérifions que cela fonctionne depuis un navigateur en renseignant l'adresse de loopback sur le port 8080 :



L’essai est concluant. Pour faire le lien avec la manip 1, si le module *eth0* est retiré, ce même test fonctionnerai car le logiciel est en local, et l’adresse de loopback sur trouve sur le module *L0*.

Pour arrêter le logiciel, il faut récupérer son PID, qui est l’identifiant du processus unique, puis utiliser la commande *kill* pour terminer le processus :

```
user@LabUTC502:~/sthttpd-2.26.4/thttpd-pkg$ ps -C thttpd
    PID TTY          TIME CMD
  39181 ?        00:00:00 thttpd
user@LabUTC502:~/sthttpd-2.26.4/thttpd-pkg$ kill 39181
```

Synthèse

Le noyau n’est pas le seul composant logiciel qui peut être compilé, n’importe quel programme peut l’être à partir du moment où l’on dispose des sources. Là aussi plusieurs outils sont mis à notre disposition pour pouvoir configurer et compiler des sources afin d’obtenir un exécutable.

Manipulation 5 : Script de démarrage

Nous créons le répertoire dans lequel se trouvera le script :

```
user@LabUTC502:~/sthttpd-2.26.4$ mkdir -p thttpd-pkg/etc/init.d
user@LabUTC502:~/sthttpd-2.26.4$ vi thttpd-pkg/etc/init.d/thttpd
user@LabUTC502:~/sthttpd-2.26.4$ ls thttpd-pkg/etc/init.d/thttpd
```

Création du script :

```
#!/bin/bash
#THHTTPD_BIN="/usr/sbin/thttpd"
THHTTPD_BIN="/home/user/sthttpd-2.26.4/thttpd-pkg/usr/sbin/thttpd"
PID_FILE="/var/run/thttpd.pid"

# Démarrage
start() {
    if [ -f "$PID_FILE" ]
    then
        # Serveur déjà en cours
        echo "Le serveur thttpd est déjà en cours d'exécution"
        exit 1
    else
        $THHTTPD_BIN -p 8080 -d ./var/www/ -i "$PID_FILE"
        echo "Le serveur a été démarré"
    fi

}

# Arrêt
stop() {
    if [ ! -f "$PID_FILE" ]
    then
        # Serveur déjà à l'arrêt
        echo "Le serveur n'est pas en cours d'exécution"
        exit 1
    else
        # On récupère le PID
        PID=$(cat "$PID_FILE")
        # Arrêt du serveur
        kill "$PID"
        echo "Le serveur a été arrêté"
        rm "$PID_FILE"
    fi

}
# Vérifier l'utilisateur qui exécute le script
if [ "$(id -u)" != "0" ]
then
    echo "L'exécution de ce script nécéssite des privilèges root"
    exit 1
fi

# Ligne de commande
case "$1" in
    "start")
        start
        ;;
    "stop")
        stop
        ;;
    "restart")
        stop
        start
        ;;
    *)
        echo "Utilisation : $0 {start|stop|restart}"
        exit 1
        ;;
esac

exit 0
```

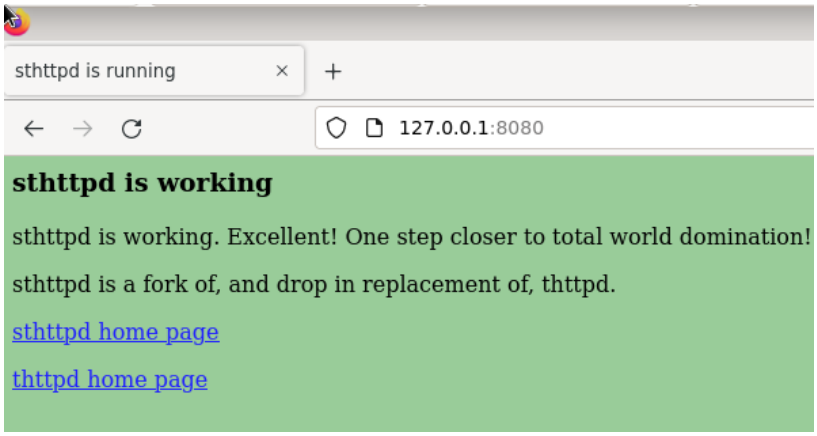
Afin de rendre le script exécutable, il faut modifier les droits :

```
user@LabUTC502:~/sthttpd-2.26.4$ cd thttpd-pkg/
user@LabUTC502:~/sthttpd-2.26.4/thttpd-pkg$ sudo chmod 755 etc/init.d/thttpd
user@LabUTC502:~/sthttpd-2.26.4/thttpd-pkg$ ls -la etc/init.d/
total 12
drwxr-xr-x 2 user user 4096 May 12 08:46 .
drwxr-xr-x 3 user user 4096 May 12 08:43 ..
-rwxr-xr-x 1 user user 1139 May 12 08:44 thttpd
```

Nous effectuons le test du script pour le démarrage et redémarrage :

```
user@LabUTC502:~/sthttpd-2.26.4/thttpd-pkg$ sudo ./etc/init.d/thttpd start
Le serveur a été démarré
user@LabUTC502:~/sthttpd-2.26.4/thttpd-pkg$ sudo ./etc/init.d/thttpd restart
Le serveur a été arrêté
Le serveur a été démarré
user@LabUTC502:~/sthttpd-2.26.4/thttpd-pkg$ sudo ./etc/init.d/thttpd start
Le serveur thttpd est déjà en cours d'exécution
```

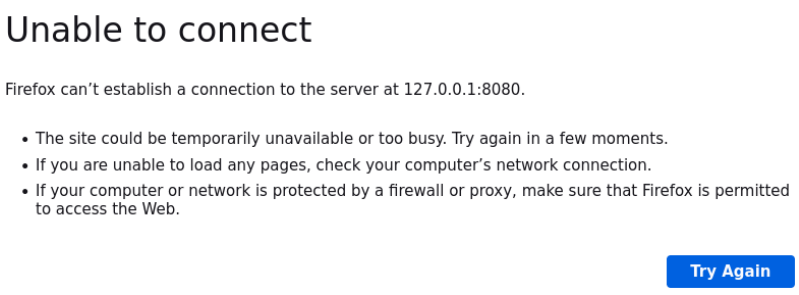
Sur la page web, le service est disponible :



Arrêt du script :

```
user@LabUTC502:~/sthttpd-2.26.4/thttpd-pkg$ sudo ./etc/init.d/thttpd stop
Le serveur a été arrêté
user@LabUTC502:~/sthttpd-2.26.4/thttpd-pkg$ sudo ./etc/init.d/thttpd stop
Le serveur n'est pas en cours d'exécution
```

Le service n’est plus disponible sur la page web :



Que ce soit pour le démarrage ou l’arrêt, le script a fonctionné. Nous avons eu les informations renseignées dans le script lors de son utilisation.

Synthèse

Un programme fonctionnant en tant que service doit rendre la main après son lancement, il s'exécute donc en tâche de fond. Arrêter un tel service nécessite de connaître son PID et d'exécuter la commande *kill*, ce qui n'est pas le plus "user friendly".

Il est utile de disposer d'un script de démarrage pour simplifier les interactions utilisateur avec le programme. Un tel script peut prendre en paramètre les chaînes de caractères start/stop et parfois restart. Ainsi le script va vérifier l'état actuel du service et le lancer, ou le stopper, selon les circonstances.

Manipulation 6 : Création de paquet

Nous créons le fichier de control puis affichons le contenu :

```
user@LabUTC502:~/sthttpd-2.26.4$ mkdir thttpd-pkg/DEBIAN
user@LabUTC502:~/sthttpd-2.26.4$ vi thttpd-pkg/DEBIAN/control
```

```
Package: thttpd
Version: 1.0
Section: base
Priority: optional
Architecture: all
Depends: bash
Maintainer: Le Coz J.
Description: Lance un serveur web
Homepage: http://127.0.0.1:8080
```

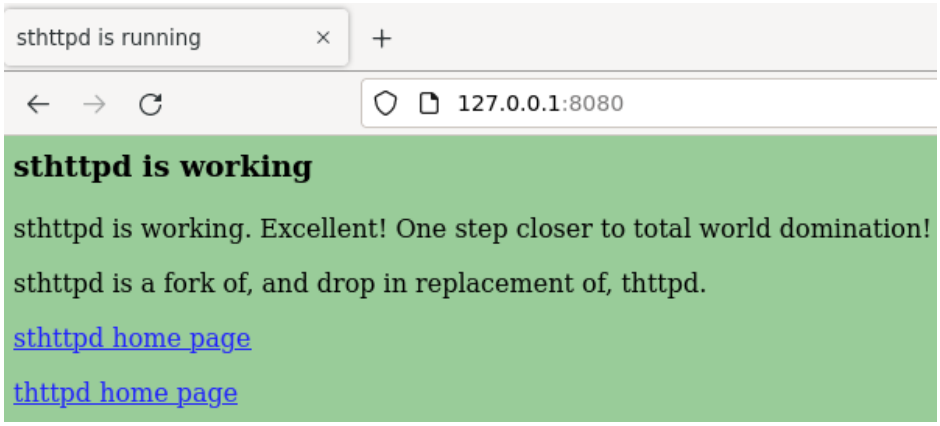
Nous créons le paquet avec la commande *dpkg-deb* :

```
user@LabUTC502:~/sthttpd-2.26.4$ dpkg-deb --build thttpd-pkg
dpkg-deb: building package 'thttpd' in 'thttpd-pkg.deb'.
user@LabUTC502:~/sthttpd-2.26.4$ ls thttpd-pkg.deb
thttpd-pkg.deb
```

Nous installons le paquet précédemment créé :

```
user@LabUTC502:~/sthttpd-2.26.4$ sudo dpkg -i thttpd-pkg.deb
Selecting previously unselected package thttpd.
(Reading database ... 157063 files and directories currently installed.)
Preparing to unpack thttpd-pkg.deb ...
Unpacking thttpd (1.0) ...
Setting up thttpd (1.0) ...
Processing triggers for man-db (2.9.4-2) ...
```

Nous testons le scritp afin de voir si nous pouvons démarrer et arrêter le serveur. Le test est concluant, le serveur est joignable :



Synthèse

Une fois correctement configuré et compilé, disposant d'un script de démarrage, notre programme est prêt à remplir le rôle auquel il est destiné. Cependant ces différentes manipulations nécessitent des connaissances spécifiques et sont sources potentielles d'erreurs. Grâce à l'utilitaire *dpkg*, il est possible d'empaqueter ce programme en tant que produit fini, de le partager et de l'installer facilement sur d'autres systèmes. On s'économise ainsi les étapes de téléchargement des sources, de configuration, de compilation et d'installation.

Synthèse globale

L'écosystème Linux est open-source. Les systèmes d'exploitation associés, ainsi que les programmes recommandés, sont souvent embarqués dans le noyau ou dans l'environnement de bureau proposé par le système. Ils sont donc également open-source. A la différence d'un logiciel propriétaire, les sources d'un logiciel open-source sont disponibles, consultables et même souvent modifiables ainsi que distribuables sous certaines conditions.

Le fait que le code source du noyau, et des applications, soit libre implique plusieurs choses. C'est un matériau gratuit, de conception transparente et de fait très hautement personnalisable. En effet, il est possible de compiler les sources du noyau, mais rien ne nous empêche de modifier ces sources avant la compilation, à condition d'avoir les compétences requises. Il est donc possible de prendre complètement la main sur le système que ce soit au niveau du code, de la configuration ou de l'installation des éléments présents sur la machine.

Comme son nom l'indique le noyau est un composant essentiel du système d'exploitation. Cet élément permet principalement l'utilisation du matériel de la machine, il possède donc de nombreuses fonctionnalités. En cas de modifications très précises, par-exemple le changement d'un pilote, la mise à jour de cet élément peut impliquer la compilation des sources ce qui est une opération sensible. Plusieurs types de noyaux existent, avec pour chacun ses forces et ses faiblesses (monolithique, micro-noyau ou encore monolithique modulaire). Depuis la version 2.0 le noyau Linux est un monolithe modulaire.

La conception modulaire d'un noyau permet de grandement en simplifier la manipulation. En effet, grâce à ce concept on peut facilement charger, ou retirer, des modules afin de n'avoir à recompiler le noyau que dans certains cas particuliers. Ainsi seul la modification des modules directement intégrés dans le monolithe nécessite une recompilation. L'arborescence de ces différents modules est centralisée pour en faciliter la gestion, il est ainsi facile de retrouver les différents modules utilisés. Tous les fichiers du noyau se trouvent dans un répertoire dédié à la racine du système de fichiers.

Il est malgré tout important de bien faire la part des choses entre les fonctionnalités directement intégrées dans le noyau et les modules. Dans l'exemple de ce TP, le système de fichiers ext3 est inclus dans le noyau contrairement au système de fichiers ext4. Ce dernier est donc facilement retirable pour obtenir un noyau plus léger, là où la suppression de ext3 nécessiterait une compilation. Afin de suivre l'historique des différentes manipulations, il est possible d'utiliser la commande *dmesg* pour consulter la mémoire tampon des messages du noyau.

Au démarrage, les modules du noyau sont stockés dans la partition racine. Cependant cette partition est inaccessible, il est nécessaire de charger certains modules du noyau avant de pouvoir y accéder. On se retrouve donc dans une impasse car : les modules permettant d'accéder à la partition ne sont pas accessibles, à contrario tant qu'ils ne sont pas accessibles, on ne peut pas les charger. On se retrouve dans un cercle vicieux. Les images *initrd* permettent de charger les modules minimaux afin de monter cette partition racine et ainsi permettre le démarrage du noyau. En effet, tout comme le noyau réel, cette image contient des modules et son contenu permet donc de contourner ce problème de démarrage.

Comme beaucoup d'autres choses sur un système Linux, cette image *initrd* peut être générée par le système avec le fichier de configuration *initramfs.conf* et la commande *update-initramfs*. Cette image peut contenir tous les modules du noyau, mais seuls certains d'entre eux sont indispensables au bon démarrage du système. Le fait d'intégrer seulement certains modules permet de diminuer la taille de l'image. D'autres particularités sont paramétrables comme la compression ou la connexion au réseau par une interface en particulier.

Du fait que le code source du noyau est disponible et compilable, toute une suite d'outils est disponible pour pouvoir exécuter ces opérations. Une personnalisation très fine du système est permise par cette possibilité. Pour se faire il faut déjà avoir téléchargé, et éventuellement modifié, le code du noyau, puis il faut configurer la future compilation. L'opération est coûteuse en temps et en ressources, mais permet une totale prise en main du système. Pour simplifier ces opérations délicates et réservées à un public averti, la commande *make menuconfig* met à notre disposition une interface graphique permettant de prendre la main sur les nombreuses options de compilation de ce noyau.

Tout comme pour le noyau, tous les programmes dont le code source est disponible sont compilables. Dans ce cas également, différentes suites d'outils sont disponibles pour produire des exécutables à partir des sources. Les programmes peuvent s'exécuter au premier plan ou en arrière-plan.

Lorsque le programme s'exécute en arrière-plan en tant que service, ce dernier doit rendre la main une fois qu'il est démarré. Bien que le lancement d'un tel service soit souvent plutôt classique, interrompre ce programme implique l'utilisation de la commande *kill* avec le **PID** associé au processus ce qui implique plusieurs manipulations un peu fastidieuses.

Heureusement, il est possible de mettre en place un script de démarrage, souvent écrit en bash, pour simplifier les opérations de démarrage et d'arrêt. Ce script permet de vérifier l'état actuel du service (le service est-il déjà actif ou non ?) et de le lancer ou d'arrêter selon le cas. Le script va gérer la création et la suppression d'un fichier PID pour pouvoir vérifier si le service est déjà actif ou non et éventuellement l'arrêter. Un tel script prend au moins deux chaînes de caractères éventuelles en paramètres, les chaînes start et stop, ainsi que souvent la chaine restart.

Obtenir un exécutable depuis les sources demande donc plusieurs opérations impliquant des compétences spécialisées. Chacune de ces opérations est bien sûr source d'erreurs, et certaines d'entre elles sont fastidieuses. Il faut télécharger le code source, configurer la compilation, compiler, ce qui peut durer plus ou moins longtemps selon la taille de l'exécutable et éventuellement créer le script de démarrage. Heureusement, il est possible d'empaqueter le fruit de toutes ces étapes pour simplifier le déploiement de l'exécutable. C'est ce que propose de faire, entre-autres, la commande *dpkg*. Une fois l'exécutable empaqueté, il est beaucoup plus facile et rapide de le partager ainsi que de l'installer sur d'autres systèmes. Même pour un usage strictement personnel, une telle opération facilite la réinstallation d'un logiciel. Un fichier de contrôle est requis pour décrire ce paquet et ses modalités d'installation. Une installation avec un tel utilitaire permet la création automatique des répertoires requis pour les fichiers du logiciel et l'organisation de l'arborescence initialement prévue.