



UNIVERSITE PIERRE ET MARIE-CURIE

PROJET PNL

PROGRAMMATION AU COEUR DU NOYAU LINUX

Compte-rendu

Auteurs:

Axel ARCHAMBAULT
Anas EZOUHRI
Bilel AFFES

Numéros étudiant:

3300807
3208760
3361270

Lundi 8 Mai 2017

Table des matières

1 Mode d'emploi	2
1.1 Contenu de l'archive fournie	2
1.2 Utilisation de l'outil	3
2 Traces d'exécution	4
2.1 List	4
2.2 Fg	5
2.3 Kill	6
2.4 Wait	7
2.5 Meminfo	8
2.6 Modinfo	9
3 Détails de fonctionnement	9
3.1 Outil	9
3.2 Module	10
4 Problèmes connus	11

1 Mode d'emploi

1.1 Contenu de l'archive fournie

L'archive fournie contient un dossier **Projet** qui lui même contient :

- **READ_ME.pdf** : ce compte-rendu.
- **Makefile** : permet de compiler l'ensemble du projet en utilisant les Makefiles des sous-dossiers.
- **patchPNL.patch** : patch à appliquer.
- Un dossier **Outil** : contient les fichiers nécessaires au fonctionnement de l'outil.
- Un dossier **Module** : contient tous les fichiers nécessaires au fonctionnement du module.

Le dossier **Outil** contient :

- **pnl_util.h** et **pnl_util.c** : le code de l'outil utilisateur.
- **Makefile** : permet de compiler l'outil et d'obtenir les .o associés.

Le dossier **Module** contient :

- **list_sig** : numéro de tous les signaux possibles et leur nom.
- **struct_module.h** : structures utilisées par le module.
- **commande.c** : code d'initialisation du module.
- **list.c** : code de la commande list.
- **fg.c** : code de la commande fg.
- **kill.c** : code de la commande kill.
- **wait.c** : code de la commande wait.
- **meminfo.c** : code de la commande meminfo.
- **modinfo.c** : code de la commande modinfo.
- **moduleTest.c** : code du module utilisé pour tester la commande modinfo.
- **Makefile** : permet de compiler le module et d'obtenir les .o et .ko associés.

1.2 Utilisation de l'outil

Voici les étapes nécessaires d'effectuer pour utiliser notre outil.

1. **Patcher puis compiler le noyau linux 4.2.3 avec le patch fourni.**
2. **Compiler l'outil et le module** fourni via la commande **make** (vous devez utiliser le makefile qui se trouvera dans le dossier **projet_pnl_Ezouhri_Archambault_Affes**, il s'occupera ensuite d'exécuter le make dans des dossiers Outil et Module.
3. **Insérer le module** via la commande **insmod exec_commande.ko**.
4. **Créer un périphérique** pour l'utilisation de l'ioctl via la commande **mknod /dev/commande c numéroMajeur** (normalement 245) 0.
5. **Lancer l'outil** via la commande **./pnl_outil**.

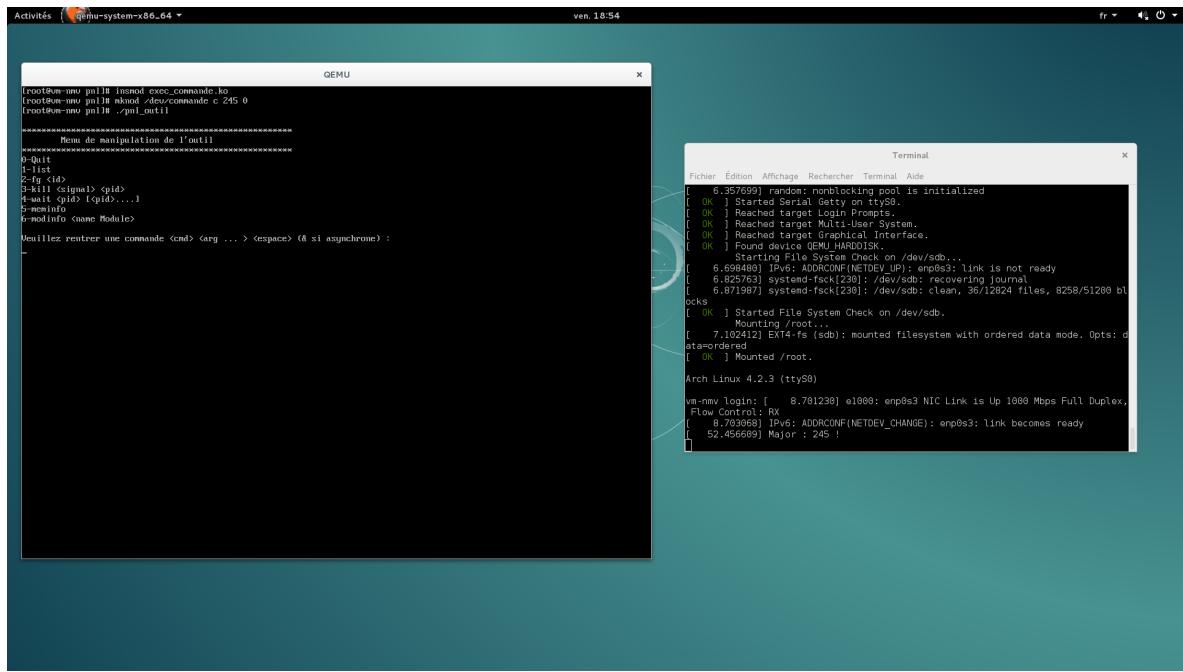
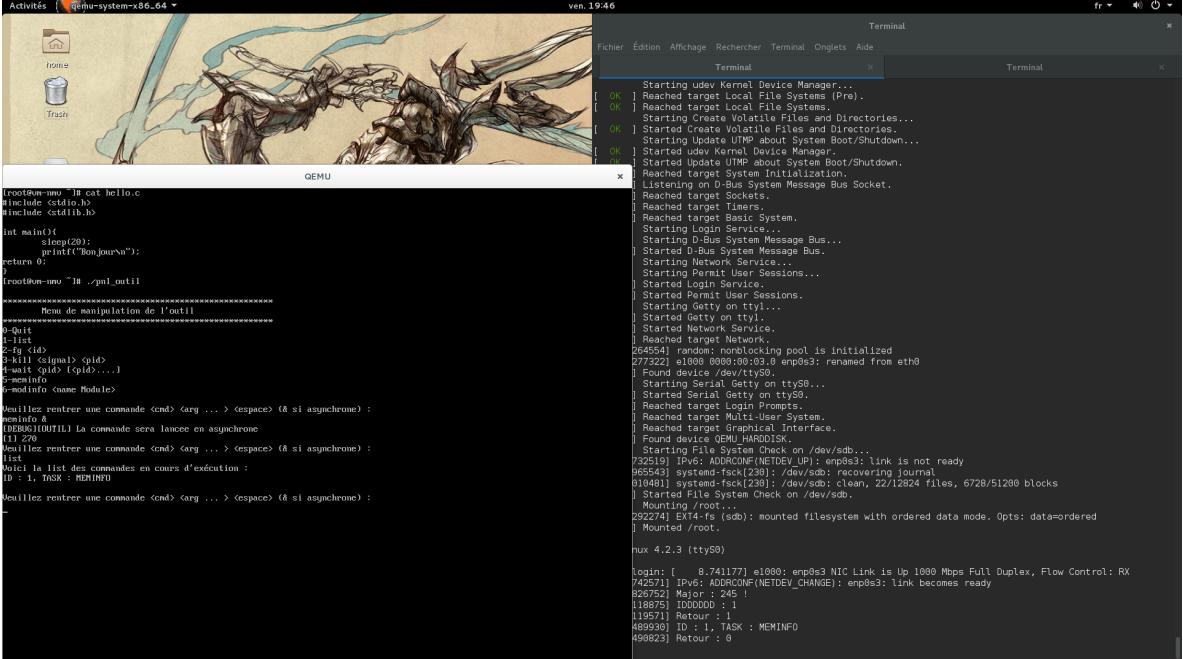


Figure 1: Lancement du module et de l'outil

2 Traces d'exécution

2.1 List



The screenshot shows a Linux desktop environment with a terminal window open. The terminal window displays a command history and a log of system boot events. The log includes entries such as:

- [OK] Reached target Local File Systems (Pre).
- [OK] Reached target Local File Systems.
- [OK] Starting Create Volatile Files and Directories...
- [OK] Started Create Volatile Files and Directories.
- [OK] Starting Update TMP about System Boot/Shutdown...
- [OK] Started Update TMP about System Boot/Shutdown...
- [OK] Started Update UTMP about System Boot/Shutdown...
- [OK] Reached target System Initialization.
- [OK] Listening on D-Bus System Message Bus Socket.
- [OK] Reached target Network.
- [OK] Reached target Basic System.
- [OK] Starting Login Service...
- [OK] Starting D-Bus System Message Bus...
- [OK] Starting Network Service...
- [OK] Starting Permit User Sessions...
- [OK] Started Login Service.
- [OK] Started Permit User Sessions.
- [OK] Started Getty on tty1...
- [OK] Started Network Service.
- [OK] Reached target Network.
- 2649541 random: nonblocking pool is initialized
- 2773234 [1.000s] /dev/eth0 renamed from eth0
- [OK] Found device /dev/tty0...
- [OK] Starting Serial Getty on ttyS0...
- [OK] Started Serial Getty on ttyS0...
- [OK] Reached target Multi-User System.
- [OK] Reached target Graphical Interface.
- [OK] Found device QEMU HARDDISK.
- [OK] Starting File System Check on /dev/sdb...
- 725254 [0.000s] /dev/sdb: link is not ready
- 965543 [0.000s] systemd-tsc(7280): /dev/sdb: recovering journal
- 910401 [0.000s] systemd-tsc(2381): /dev/sdb: clean, 22/12824 files, 6728/51200 blocks
- [OK] Started File System Check on /dev/sdb.
- [OK] Mounting /root...
- 2922741 [0.000s] ext4-fs (sdb): mounted filesystem with ordered data mode. opts: data=ordered
- [OK] Mounted /root.

The terminal also shows a command history with commands like `cat hello.c`, `gdb ./pml_outil`, and `modinfo`.

Figure 2: Commande list pendant qu'une tâche est en cours

Compte-rendu

Axel ARCHAMBAULT
 Anas EZOUHRI
 Bilel AFFES

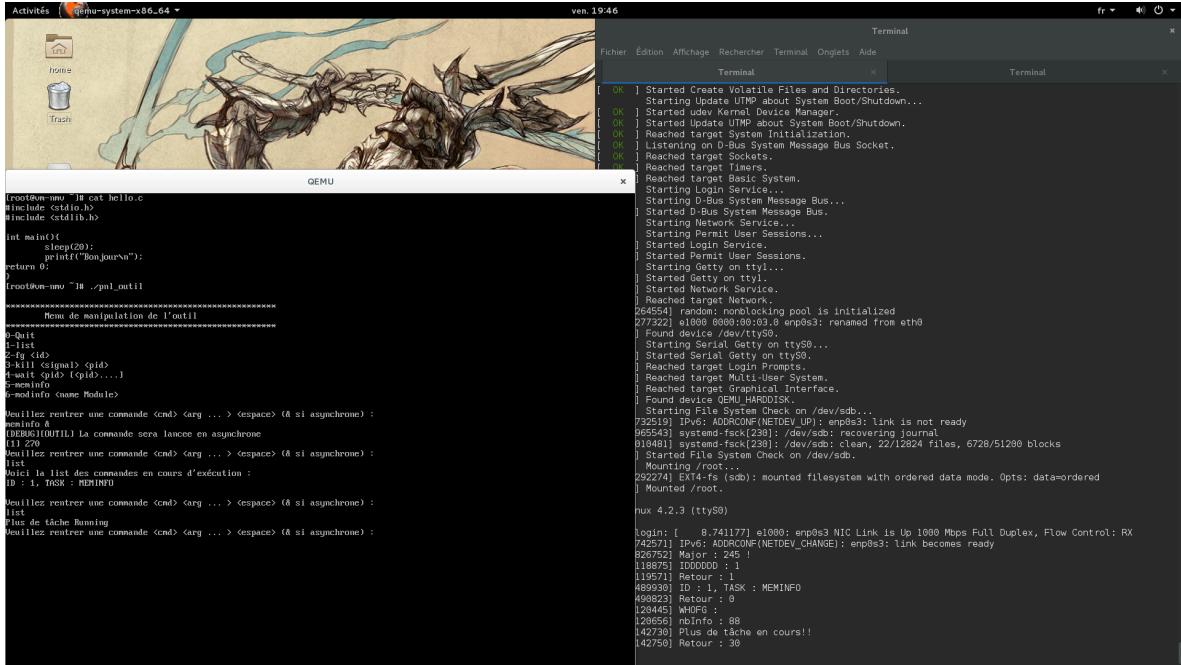


Figure 3: Commande list apres la fin de la tache

2.2 Fg

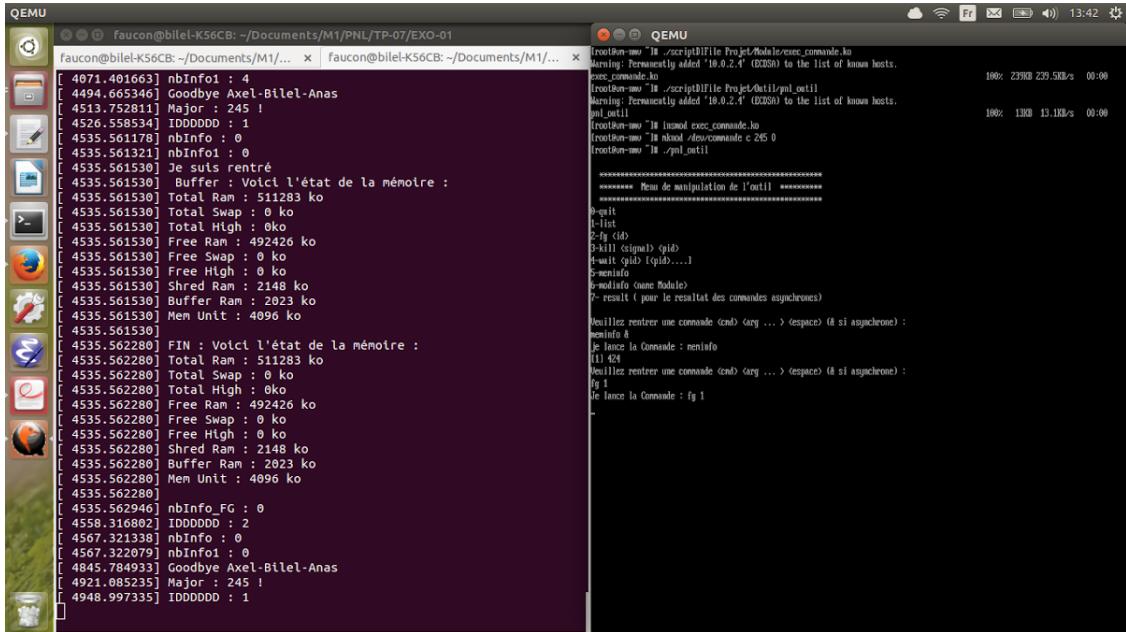


Figure 4: Commande fg d'une tache en cours

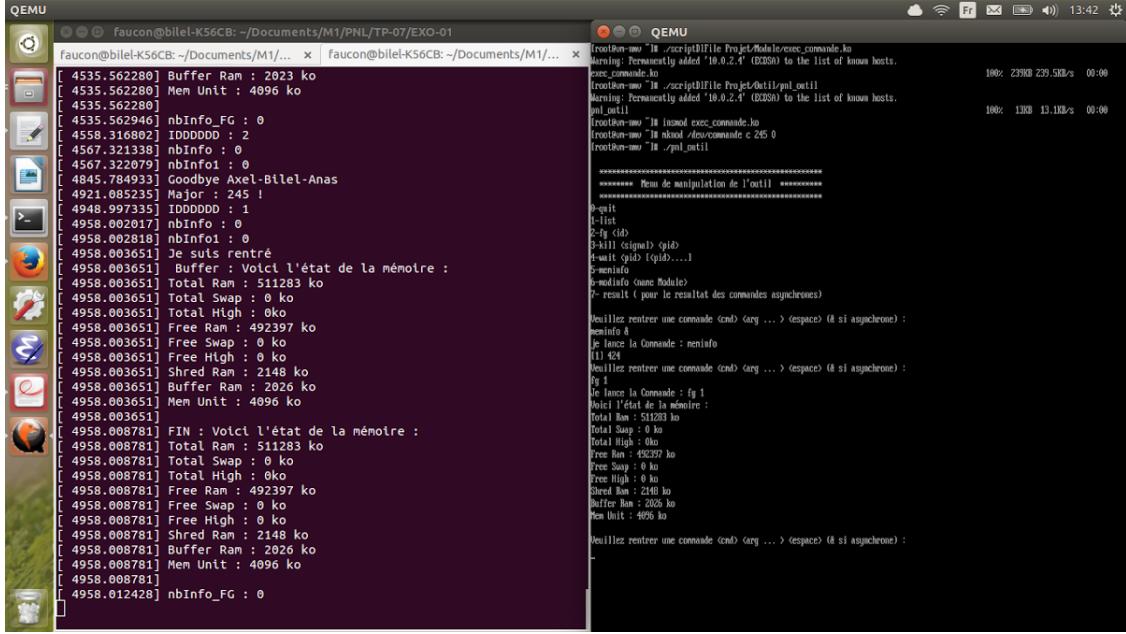


Figure 5: Fin d'une tache qu'on a fg

2.3 Kill

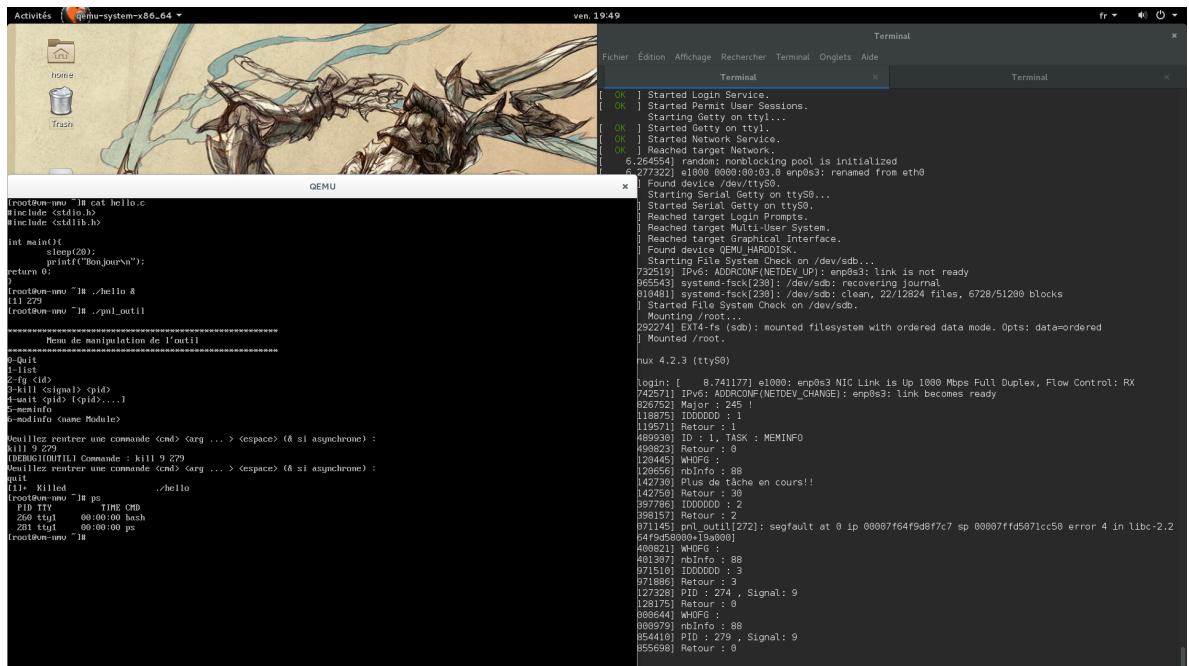


Figure 6: Commande kill

2.4 Wait

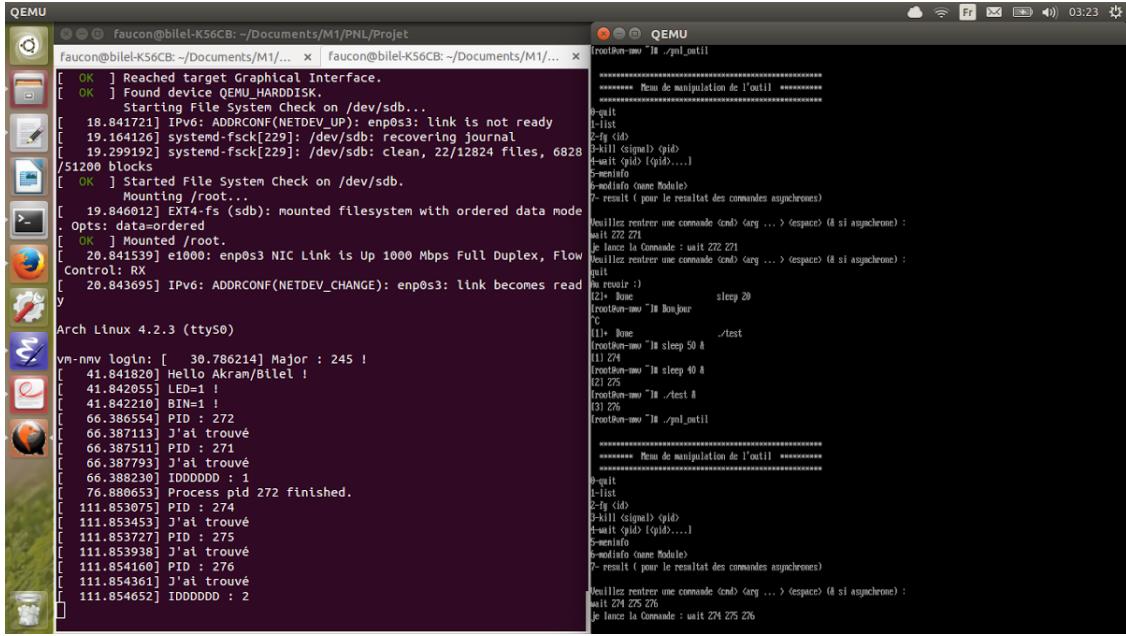


Figure 7: Commande wait sur plusieurs tâches

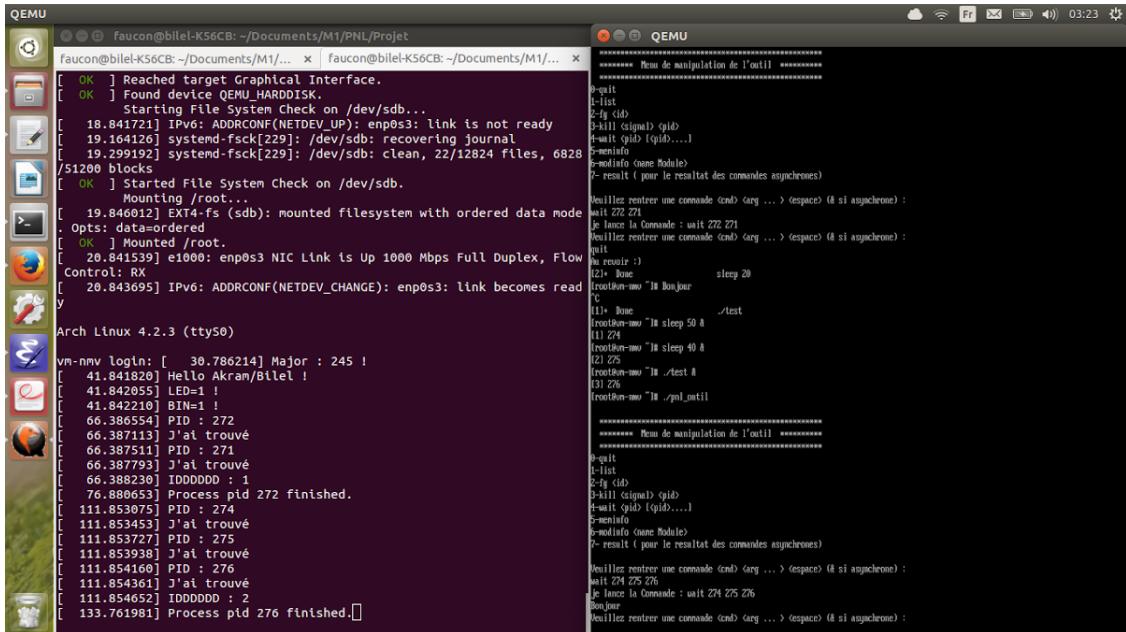


Figure 8: Fin d'une des taches qu'on a wait

2.5 Meminfo

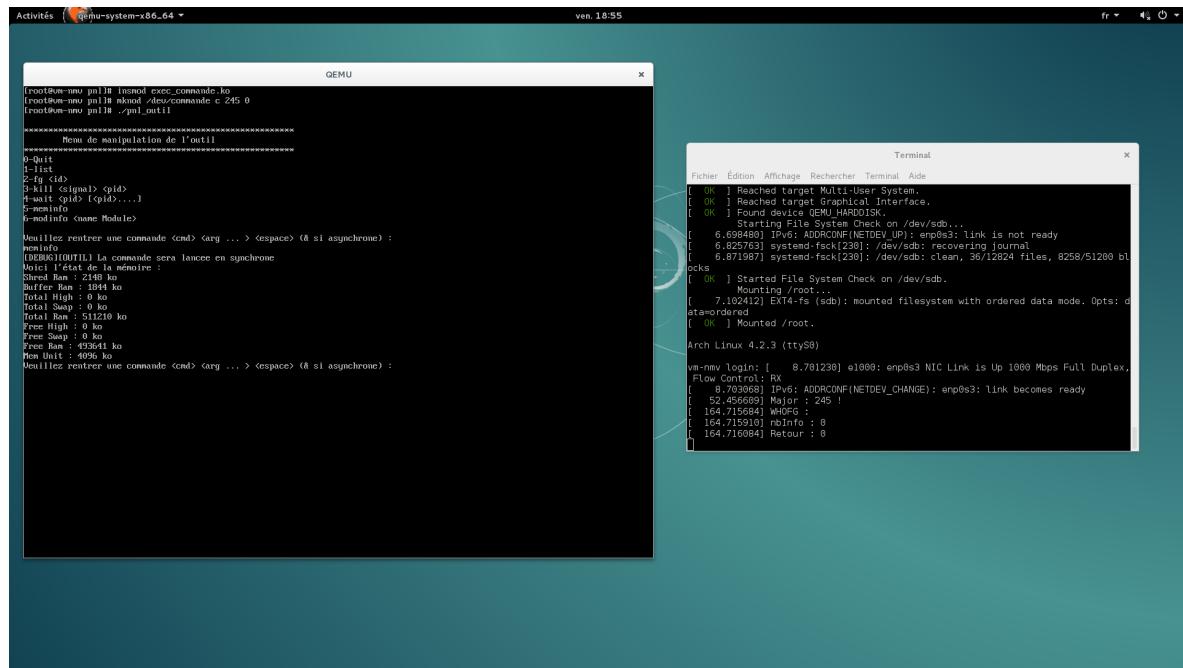


Figure 9: Commande meminfo

2.6 Modinfo

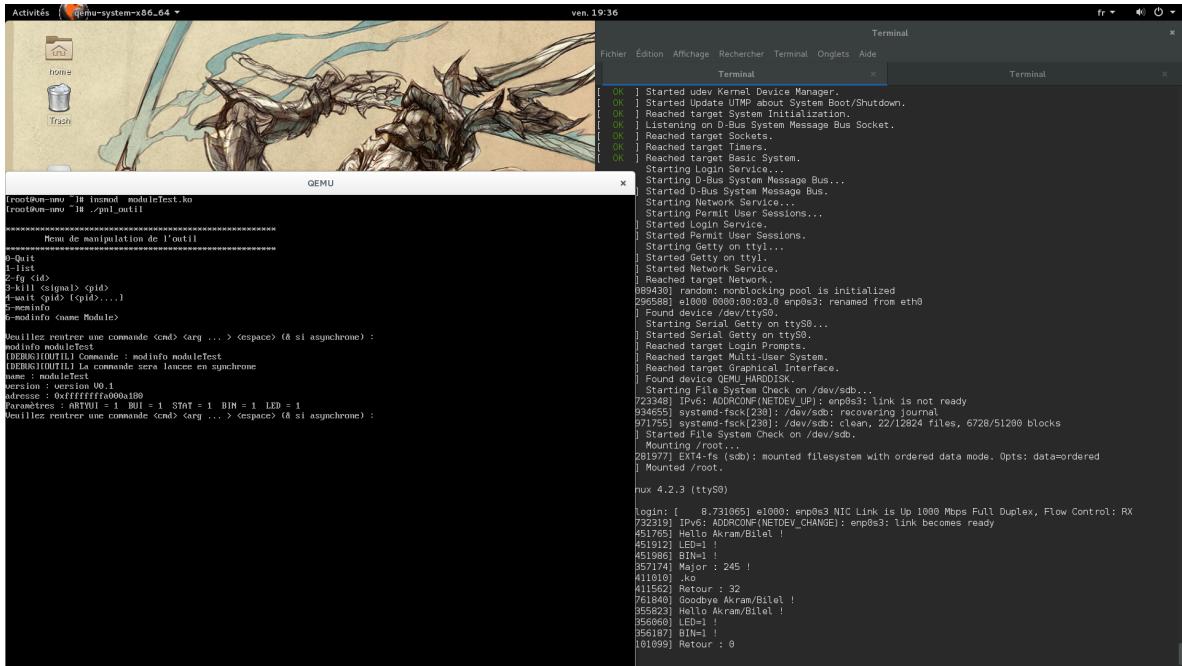


Figure 10: Commande modinfo

3 Détails de fonctionnement

Donc comme dit ci-dessus, nous avons à l'intérieur du dossier **projet_pnl_Ezouhri_Archambault_Affes** deux dossiers, Module et Outil ainsi qu'un Makefile.

3.1 Outil

Dans le dossier Outil, vous trouverez le fichier `pnl_outil.c` et `.h`. Le `.h` contiendra toutes les structures utilisées par l'utilisateur et qui permettent de récupérer le résultat de chaque commande ainsi que d'envoyer les arguments de ces derniers. Le `.c` quant'à lui contient le code source de l'outil.

En exécutant le code, vous aurez un menu qui s'affichera, il vous aidera juste à connaître la syntaxe de chaque commande ainsi que l'argument que vous devez fournir avec, la saisie de la commande se fait exactement comme dans un terminal : `<commande><arguments><espace & si asynchrone>`. Une fois la

commande saisie, nous récupérons cela, et nous découpons ensuite la commande en stockant dans un tableau le nom de la commande, ses arguments et si vous souhaitez qu'elle soit en asynchrone, l'esperluette (&).

Ensuite, nous comparons la commande avec une chaîne de caractère qui correspondra au nom d'une commande, si la comparaison réussit, alors nous entrons dans le if, et nous préparons la structure associée à cette commande en la remplaçant avec les valeurs correspondantes, puis nous appelons la méthode ioctl en lui passant le nom de la requête et ses arguments. Si la commande est exécutée en mode synchrone, alors au retour de la fonction, nous faisons un goto finish pour afficher le résultat, sinon dans le cas contraire, l'utilisateur pourra saisir une autre commande si il le souhaite.

A la fin du fichier, après tout les if else, vous trouverez un label finish, et juste après un certains nombre de if qui correspondront aux cas d'erreur ainsi que l'affichage des résultats, cela permettra d'afficher les résultats des commandes synchrone directement avec un goto, ou les asynchrones, mais dans ce cas là, vous devez saisir la chaîne "result" qui apparaît dans la 7 ème position dans le menu et qui fera juste un goto finish. Tant que le résultat n'est pas prêt, on ne pourra rentrer dans aucun if. Chaque structure a un champs finish qui sera égal à 1 si la commande est finie. Le printf qui se trouve juste après le label va permettre d'afficher l'erreur si elle surgit durant l'exécution de la commande Kill en mode asynchrone, sinon pour les commandes synchrones c'est les autres if qui s'occupent de l'affichage.

3.2 Module

Une fois que la fonction ioctl est appelée, nous rentrons alors dans le module et nous exécutons la fonction correspondante à la requête associée.

Nous avons le fichier **commande.c** qui représente le fichier principale du module, il contient les fonctions d'initialisation des tâches asynchrone, les fonctions d'ajout et de suppression des tâches dans la liste des tâches en cours, ainsi que la fonction ioctl kernel qui contient un switch(req) , chaque cases du switch correspondent à une commande associée à une requête. Enfin, les deux fonctions principales, qui seront exécutées durant l'insertion et suppression du module.

Chaque fichier hormis **commande.c**, contiendra deux fonction :

<nom_de_la_commande>_handler(argument) : cette fonction sera exécutée dans le switch case de la fonction ioctl, elle prend en argument la structure correspondante à la commande. Elle récupérera les arguments de la structure passée à la fonction, et si l'exécution doit être faites en mode asynchrone, initialise la tâche en lui attribuant

un id, le nom de la commande, la fonction à exécutée, puis l'ajoute dans la liste **taskRunning** et ensuite appelle la fonction **schedule_delayed_work()** pour lancer la tâche dans la Workqueue, Sinon, elle exécutera directement en mode synchrone la fonction qui s'occupera de la tâche avec un argument NULL.

<nom_de_la_commande>(+_process pour certain) : c'est la fonction qui sera exécutée par la Workqueue en mode asynchrone, sauf pour la commande Wait et Fg, elles seront exécutées en mode synchrone. Tout ces fonctions vérifieront d'abord les cas d'erreur, récupéreront la structure correspondante à la tâche en cours grâce à la fonction **task_asynch_of()** qui prendra en argument la variable work passée en argument, puis feront leurs traitements, à la fin, ils utiliseront la fonction **copy_to_user()** pour écrire le résultat dans l'espace utilisateur, ou les erreurs dans le cas échouant. Le champ **data** dans la structure task_asynchrone permettra de stocker l'adresse à laquelle le résultat doit être copié. Dans la plus part des fonctions nous avons deux **copy_to_user** au minimum : un pour le résultat et un autre pour dire qu'on a fini. Les conditions **if(work != NULL)** permettent de savoir si nous sommes en mode synchrone ou asynchrone. Enfin, nous avons une condition qui permet de savoir si on a été mis en avant plan ou pas. Si oui, alors on va réveiller le processus qui a exécuté Fg grâce à la fonction **wake_up()**;

les deux commandes Wait et Fg sont toujours exécutées en synchrone car vu que ces deux commandes ne rendent jamais la main que lorsqu'une tâche se termine pour le Wait ou lorsqu'une commande revient en arrière plan où se termine pour Fg, nous avons décidé de les laisser en synchrone.

4 Problèmes connus

Toutes les commandes fonctionnent parfaitement, le seul petit soucis qui existe est que, de temps à autre, la fonction **copy_to_user** n'arrive pas à copier dans l'espace utilisateur, mais très rarement, il y a un **pr_info** qui affiche l'erreur si la copie ne se passe pas bien, si ce cas se présente, ré-exécuter la commande et elle fonctionnera.

Il y a également deux lignes : **kfree(list_tmp); kfree(task_tmp);** dans la fonction **remove_list()** qui ne cause pas de soucis durant l'exécution, au contraire, l'outil fonctionnera parfaitement, mais lorsqu'on quitte l'outil et si on veut récupérer un autre fichier en ssh, il y a une exception qui se lance et donc la VM est bloqué, nous avons mis ces deux lignes en commentaire pour ne pas créer de soucis.