Projet de Système avancé : `pignoufs` *

Fabien de Montgolfier

M1 Informatique 2025

1 Description du projet

Le but du projet est d'implémenter un système de fichier à l'intérieur d'un fichier standard de l'utilisateur, appelé le **conteneur**. Il sera accessible à travers des outils reprenant ceux de base d'un système de fichier (comme `ls` ou `cp`) au lieu d'être monté.

On ne pourra accéder au conteneur que via une projection avec `mmap` : tout accès via les primitives d'entrée/sortie comme `read` ou `write` est interdit.

Le système de fichier est *ultravérifié* au sens que tout bloc est protégé par un SHA1. Ainsi toute corruption devra être détectée. Pour tester cela, on pourra rajouter au conteneur de la corruption, à l'aide de `corrupt.c` (fourni) par exemple.

1.1 Noms de fichier

Un nom de fichier "interne", c'est-à-dire utilisé au sein du système de fichier `pignoufs`, commencera toujours par deux `'/'`, par exemple `//toto.txt`. Ce mécanisme permet de les distinguer des fichiers "externes" (existants dans l'arborescence de fichiers de la machine).

Si vous implémentez les répertoires, un nom de fichier `pignoufs` complet sera comme un nom complet Unix, en désignant par `//` la racine `pignoufs`, par exemple `//rep/sousrep/toto.txt`.

1.2 Types

— Dans le sujet tous les **entiers** sont sur 4 octets et sont au format *little endian* impérativement. Les **nombres** sont des entiers signés.

*Projet Informatique de Gestation d'un Nouvel Outil Ultravérifié de File System

— Une **adresse de bloc** est un entier signé compris entre 0, qui est l'adresse du superbloc, et $nb_b - 1$, où nb_b est le nombre de blocs. On utilise la valeur -1 pour "indéfini".
— Une **date** est le format utilise par `time` soit le nombre de secondes écoulées depuis le 1er janvier 1970 (epoch), normalement non signé [1].
— Une **taille de fichier** est un entier non signé (pour permettre des fichiers plus grands que 2Gio)

1.3 Blocs et contrôle d'erreur

L'unité d'allocation utilisée est le **bloc**. Au niveau inférieur (à l'interface avec le système d'exploitation réel), un bloc fait 4096 octets ou **4 kio** [2]. Un bloc se décompose toujours en 5 zones :
— le bloc effectif de 4 ko (4000 octets)
— un SHA1 de 20 octets, qui est le hashcode des 4000 octets précédents
— 4 octets de *type* (voir ci-dessous)
— 72 octets à disposition : verouillage et *garbage* dont la valeur est quelconque (cf section 5.3)

Cette taille de 4kio correspond à une taille de page Linux : les adresses en mémoire des blocs doivent être **alignées**, c'est-à-dire être multiples de 4096, si sur votre machine `sysconf(_SC_PAGESIZE) == 4096` (sinon, ce n'est pas grave).

Une version (très) dégradée du projet consiste à ne pas utiliser les SHA1, donc à ne faire aucun contrôle d'intégrité. Dans ce cas il faudra quand même conserver le format consistant à mettre 4000 octets par bloc de 4kio, en laissant à 0 les champs non implémentés (SHA1, type et verouillage).

Les types de blocs sont donc :

1. superbloc
2. bitmap
3. inode
4. bloc allouable libre
5. bloc de données
6. bloc d'indirection simple
7. bloc d'indirection double

1. si on est avant 2038 `https://fr.wikipedia.org/wiki/Bug_de_l'an_2038`
2. rappel, il existe une grande confusion dans le monde informatique entre les kilo-octets décimaux (1000 octets) abrégés ko ; et les kilo-octets binaires (1024 octets), normalement abrégés kio, mais on utilise "ko" de façon ambigüe. Dans ce sujet on utilisera **ko** pour 1000 octets

2 Format du système de fichier

Physiquement, le système de fichiers stocké dans le conteneur est constitué de quatre zones contigües et consécutives :

1. superbloc (un seul bloc)
2. bitmaps des blocs libres (nb_1 blocs)
3. inodes (nb_i blocs, avec une inode par bloc)
4. blocs **allouables** (nb_a blocs)

Le nombre total de blocs est $nb_b = 1 + nb_1 + nb_i + nb_a$, avec $nb_1 = \lceil \frac{nb_b}{32000} \rceil$
La taille du conteneur (codant le système de fichier) est $4096nb_b$. Un bloc allouable peut

— soit être **libre**. Son type est 4. Le contenu n'a pas à être effacé quand un bloc est libéré.
— soit être un **bloc de données** : il contient les données d'un fichier (4000 octets de données, sauf si fin de fichier dans ce bloc). Son *type* est 5.
— soit être un **bloc d'adresses** : il contient 1000 adresses de blocs (voir plus loin). Son *type* est 6 (indirection simple) ou 7 (indirection double), cf section 2.4.

2.1 Zone 1 : superbloc

Le premier bloc est le superbloc. Il obéit à un format spécial. On ne décrit que les 4000 premiers octets, les 96 derniers obéissant tous au format décrit en section 1.3. Le *type* (au sens de la section 1.3) est l'entier 1.

— un magic number (8 octets) valant "`pignoufs`"
— le nombre total nb_b de blocs du système de fichier (4 octets)
— le nombre nb_i d'inodes du système de fichier (4 octets)
— le nombre nb_a de blocs allouables (4 octets)
— le nombre nb_l de blocs allouables libres (non alloués) (4 octets)
— le nombre nb_f de fichiers du système, égal au nombre d'inodes allouées (4 octets)
— 3972 octets de 0 (sauf extension, à documenter dans le rapport)

cela correspond à une **struct pignoufs** :

```
struct pignoufs {
  char magic[8];
  int32_t nb_b;
  int32_t nb_i;
  int32_t nb_a;
  int32_t nb_l;
  int32_t nb_f;
  char zero[3972];
}
```

2.2 Zone 2 : bitmaps des blocs libres

Un bloc de *bitmap de blocs libres* est une suite de 32000 bits, chacun correspondant à un bloc. Le bit est à 0 si le bloc correspondant est alloué, à 1 s'il est libre. Ainsi le bit d'indice 123 (donc le 124ème bit) du troisième bloc de bitmaps (qui est le quatrième bloc du conteneur) correspond au bloc numéro $(2 * 32000 + 123) = 64123$. Chaque bit du bitmap vaudra

— 1 si le bloc correspondant est un bloc allouable et non alloué
— 1 si le bloc correspondant est une inode non allouée (aucun fichier)
— 0 dans les autres cas (attention **0 = non libre**).

Les bitmaps mis bout-à-bout constituent donc la **table des blocs libres**. Si le nombre de blocs nb_b n'est pas multiple de 32000 alors le contenu des derniers bits de la table est un bourrage de 0.

Après ces 4000 octets de bitmap, les 96 derniers obéissent tous au format décrit en section 1.3. Le *type* est l'entier 2.

2.3 Zone 3 : inodes

Notez que `pignoufs` utilise de grosses inodes : un bloc de 4kio par inode, et que certains attributs (propriétaire,...) ou types (fichier spécial,...) classiques peuvent manquer. Si aucun fichier n'a été alloué pour le numéro correspondant, les 4000 octets de l'inode doivent être tous à 0 (mais pas les 96 derniers du bloc "physique", pensez au SHA1 et au type). Si un fichier a été alloué, alors elle est constituée des champs suivants :

— 4 octets de flags :
  0. existence du fichier, donc vaut 1 (bit numéro 0)
  1. droit en lecture du fichier
  2. droit en écriture du fichier
  3. verrouillage en lecture du fichier.
  4. verrouillage en écriture du fichier.
  5. type répertoire. 0 si fichier régulier, 1 si répertoire (cf section 6.2)

  6-31 les autres bits (numéros 6 à 31) doivent être à 0, sauf extension (documentée dans le rapport)

— taille du fichier (4 octets)
— date de création (4 octets).
— date du dernier accès (4 octets)
— date de modification (4 octets)
— nom de fichier (256 octets). Le nom peut faire au maximum 255 octets, en évitant les 0 (`'\0'`) et les `'/'`. Les octets suivant le nom doivent être à 0. Ainsi le nom du fichier `//toto` est contitué des 4 caractères `'t' 'o' 't' 'o'` suivis de 252 zéros (`'\0'`).

— 900 adresses de blocs. La première est l'adresse du premier bloc de données, et la dernière cette du 900ème. Si le fichier fait moins de 3 600 000 (900*4ko) octets, les adresses non utilisées (les dernières) sont à -1.
— numéro du bloc d'indirection double (4 octets) si la taille dépasse 3600000 octets, cf section 2.4. Sinon on met -1.
— 120 octets d'extension, à votre disposition. Ceux non utilisés doivent être mis à 0. Bien décrire dans le rapport les extensions utilisées.

Enfin les derniers octets obéissent au format décrit en section 1.3. Le *type* est l'entier 3.

2.4 Blocs d'adresse

Un bloc de *type* 6 ou 7 (adresses) se rencontre en zone 4, mélangé à des blocs de type 4 (non alloués) et 5 (données). Il contient 1000 adresses de blocs. Si le bloc est de type 6 (indirection simple) ces adresses sont des adresses de blocs de données (de type 5). S'il est de type 7 (indirection double), ce sont des adresses de blocs d'indirection simple (de type 6). Si une adresse n'est pas utilisée, elle est à -1.

Par exemple, si le fichier `//toto` fait 8 000 010 octets, ce qui demande 2001 blocs alloués pour le stocker :

— le numéro des 900 blocs contenant les 3 600 000 premiers octets du fichier seront indiqués dans l'inode
— l'inode contient l'adresse du bloc d'indirection double B_0
— le bloc d'indirection double B_0 contient l'adresse des blocs d'indirection simples B_1 et B_2 puis 998 fois -1.
— le bloc B_1 contient les adresses de 1000 blocs de données, contenant les octets du fichier d'offset 3 600 000 à 7 599 999 inclus.
— le bloc B_2 contient les adresses de 101 blocs de données, contenant les octets du fichier d'offset 7 600 000 à 8 000 009, puis 899 fois -1.

Bien sûr si la taille d'un fichier n'est pas un multiple de 4000, comme dans cet exemple, alors on n'utilise pas les derniers octets du dernier bloc de données, mais **il faut les mettre à 0** pour que le calcul du SHA1 de ce bloc soit prévisible. Ainsi, dans l'exemple le dernier bloc de données (la 101ème adresse de B_2) contient les 10 derniers octets du fichiers, puis 3990 octets à 0.

Notez que ce système d'indirection double fait que la taille maximum d'un fichier est de 900+1000*1000 blocs soit 4003600000 octets [3].

3. cette limite est celle typique des systèmes des années 1990 comme FAT32 de Microsoft qui a 4Gio de maximum

3 Programmes à implémenter

Toutes les commandes pour accéder aux fichiers prendront la forme
`pignoufs commande fsname [autres paramètres]`
où `fsname` désigne toujours le nom du fichier conteneur du système de fichier. Notez que les option éventuelles sont à mettre après le nom du conteneur pour respecter cette syntaxe, par exemple
`pignoufs ls conteneur -l fichier`

On peut aussi utiliser le nom de la commande, par exemple
`fsck toto` au lieu de `pignoufs fsck toto` mais il faudra un seul exécutable (il récupère sa commande dans `args[0]`). On peut rajouter des extensions (options préfixée d'un tiret comme `-v` etc.) **à préciser dans le rapport !**

On prendra grand soin au **traitement** des erreurs, autant les erreurs système (l'appel système retourne -1 etc.) que les erreurs internes à l'application, telles que fichier inexistant, système de fichier plein, corruption détectée (rajoutée par `corrupt.c` ou autre) etc. Un message explicite sera affiché sur l'erreur standard. En cas de succès les commandes sont muettes, sauf explicitement demandé comme `ls` : évitez la verbosité inutile. Les commandes à implémenter sont :

3.1 `mkfs`

`pignoufs mkfs fsname nb_i nb_a` crée un nouveau système de fichier, stocké dans le conteneur (fichier régulier) `fsname` avec `nb_i` inodes (toutes vides) et `nb_a` bloc allouables (tous vides aussi).

3.2 `ls`

Liste tout le conteneur ou des fichiers. Le mieux est de reprendre le fonctionnement de `ls` : traitement de flags notemment `-l`, `-t`, tri par nom du contenu, etc. Le comportement minimum est que

`pignoufs ls fsname` affiche le nom de tous les fichiers stockés dans le filesystème `fsname`

`pignoufs ls fsname //filename` affiche s'il existe un fichier alloué nommé `filename` dans le filesystème, et si oui affiche aussi ses attributs (comme le ferait `ls -l`) sauf si vous implémentez effectivement l'option `-l` (pour coller davantage au comportement de `ls`).

3.3 `df`

`pignoufs df fsname` affiche le nombre de blocs de 4kio libres. On peut prévoir des options (à documenter) pour les inodes libres ou le nombre d'octets libres etc.

3.4 `cp`

La commande `cp` fait des copies internes, ou entre le système de fichier de la machine et `pignoufs`. Un nom de `pignoufs` sera toujours précédé par un **double slash**. Ainsi
`pignoufs cp fsname toto //tata`
copie le fichier `toto` du répertoire courant sous le nom `tata` dans `pignoufs`. Si aucun fichier nommé `tata` n'existe, une nouvelle inode est allouée pour le fichier. Sinon, il est écrasé. Des options peuvent éventuellement modifier ce comportement, à préciser dans le rapport. La copie "sortante" est réalisée par
`pignoufs cp fsname //tata tutu`
et une copie interne par
`pignoufs cp fsname //tata //titi`

3.5 `rm`

`pignoufs rm fsname file` supprime le fichier nommé `file` s'il existe.

3.6 `lock`

`pignoufs lock fsname //toto r` et `pignoufs lock fsname //toto w` tentent de verouiller le fichier nommé `//toto`, en lecture et écriture respectivement, sans rien faire d'autre et sans relâcher le verrou jusqu'à être terminé (par une touche, ou bien, il est mieux de prévoir un gestionnaire de signal SIGTERM qui relâche le verrou en cas de Control-C). Il faut afficher `verrou pris` pour distinguer une attente de verrou et une possession de verrou.

3.7 `chmod`

`pignoufs rm fsname file +r` met les droits en lecture, `-r` les supprime, et `+w -w` pour l'écriture.

3.8 Autres commandes d'accès aux fichiers

`pignoufs cat fsname //toto` affiche sur la sortie standard le fichier nommé `//toto`.

`pignoufs input fsname //tutu` lit l'entrée standard et l'écrit dans le fichier nommé `//tutu` en l'écrasant au besoin.

`pignoufs add fsname toto //tutu` ajoute le fichier régulier `toto` en queue du fichier `tata` du système de fichiers `fsname`, comme le ferait `cat toto >> tata` entre deux fichiers réguliers.

`pignoufs addinput fsname //tutu` lit l'entrée standard et l'écrit à la suite du fichier nommé `//tutu`, comme le ferait
`cat - >> tata` vers un fichiers régulier.

3.9 `fsck`

`pignoufs fsck fsname` vérifie si le système de fichiers est valide. Il faut au moins vérifier

— le magic number
— les hashs de tous les blocs
— si les inodes et blocs déclarés libres ou alloués dans le bitmap le sont effectivement (type cohérent et contenu des inodes libres à 0)

Il est mieux de faire d'autres vérifications,telles que : chaque fichier a une quantité de blocs alloués corresondant à sa taille (sauf si vous implémentez les trous,*cf* 6.3) ; ces blocs sont effectivement alloués ; les blocs alloués le sont à un seul fichier etc. Détailler dans le rapport ce qui est fait !

On peut supposer que la commande s'exécute sans accès concurrent à `fsname` (un "vrai" système de fichiers est démonté pour être vérifié).

La commande doit enlever tous les verrous éventuels qu'elle trouve (signes de crash antérieur).

3.10 `mount`

Cette commande est facultative
`pignoufs mount fsname` ne réalise pas un montage, mais peut être utilisée en préliminaire à toute autre commande concurrente pour initialiser les structures de concurrence comme les verrous et mutex. On peut aussi avoir une commande `umount` si besoin. Dites dans votre rapport si vous utilisez cette commande, qui n'est pas à implémenter si elle n'est pas nécéssaire (en cas de verouillage global par exemple)

4 Contrôle d'intégrité

Toute lecture d'un bloc doit être précédé d'un contrôle d'intégrité de son SHA1. Toute écriture d'un bloc doit être suivie de l'écriture de son SHA1. Un E/S sans contrôle d'intégrité du bloc sera pénalisée. Il est par contre possible de paralléliser : lecture et vérification peuvent être simultanée (avec un thread dédié à la vérification), et de retarder l'écriture du SHA1 (en la confiant à un thread ad hoc, *cf* section 6.4).

5 Accès concurrents et verouillage

Un aspect important du projet est la concurrence, c'est-à-dire le fait que plusieurs processus puissent tenter d'intervenir ensemble sur le système de

fichier sans créer d'incohérence. Il est **nécéssaire** d'implémenter une façon de gérer la concurrence. Vous **devez écrire dans le rapport** comment vous avez géré la concurrence. On peut utiliser :

5.1 Verrou géant (global)

C'est la façon la plus simple (donc la moins bien notée !) de faire. Un seul processus peut accéder en écriture au système de fichier, les autres attendent. On peut utiliser soit un fichier lock, soit `flock, fcntl` etc.

5.2 Verrouillage par bloc

Il est mieux de prendre des verrous bloc par bloc. Pour cela on peut utiliser des sémaphores, des verrous `fcntl` ou tout autre mécanisme. Les 72 derniers octets d'un bloc sont à votre disposition pour cela (on peut y placer des sémaphores, par exemple). On peut aussi envisager de créer des types de blocs spécifiques pour y mettre les informations de verouillage. Bien décrire dans le rapport le mécanisme utilisé !

Ainsi l'écriture d'un fichier de taille connue demande à prendre des verrous d'abord sur les blocs de bitmaps pour trouver et réserver des blocs libres pour l'inode et les données, puis à verrouiller l'inode et les blocs de données ainsi alloués le temps de les écrire. Attention avec les commandes `input` et `inputadd` on se ne sait pas à l'avance de combien de bloc de données on aura besoin, il faut dans ce cas les allouer et les écrire un par un.

5.3 Verrouillage par fichier

À plus haut niveau (au-dessus des verrous de bloc) `pignoufs` prévoit un mécanisme de verouillage par fichier, dans les inodes. Chaque inode contient deux bits de verouillage, en lecture et en écriture. Ils utilisent la sémantique usuelle qu'un verrou (lecture ou écriture) est nécéssaire pour l'opération demandée (lecture ou écriture), et que si un verrou en lecture existe, on ne peut verouiller en écriture. **Chaque commande lisant le contenu d'un fichier doit le verouiller en lecture, et chaque commande écrivant du contenu, en écriture**. Il y a plusieurs façons d'implémenter ce mécanisme, les 120 octets d'extension (protégés par SHA1) ou les 72 de fin de bloc d'inode (non protégés) peuvent être utilisés.

6 Extensions

Le sujet minimal demande à implémenter la structure de données, un mécanisme de verouillage au moins global, et le contrôle d'intégrité, plus quelques commandes créant des fichiers. On pourra envisager les extensions

suivantes. Attention, si les spécifications sont modifiées (c'est le cas des sous-répertoires en particulier) **prévoyez une compilation sans extension** afin que la conformité aux spécifications de base puisse être vérfiée.

6.1 Recherche

Il est intéressant d'implémenter des commandes pour rechercher par type, nom ou date (comme `find`) ou contenu (comme `grep`). Documentez.

6.2 Sous-répertoires

Un système de fichiers "plat" est peu utile, il est intéressant d'implémenter les répertoire. Il faut alors penser à comment ce type est codé (bien le documenter dans le rapport), implémenter `mkdir`, `rmdir`, `mv`, et modifier des commandes comme `ls`. Pensez à comment savoir si un fichier est déjà présent dans un répertoire (pour savoir si `cp` écrase ou crée, par exemple). Il n'y a pas de notion de répertoire de travail donc les commandes prendront toujours un nom interne complet en paramètre, comme `//rep/sousrep/toto`.

Vous pouvez, de surcroît, implémenter une commande `ln`. Notez que si vous décidez qu'une inode peut être liée (référencée) dans plusieurs répertoire, cela implique que `rm` ne supprime que le lien (comme sous Unix). Le choix d'avoir mis le nom dans l'inode fait que l'équivalent de `ln toto tata` n'est pas possible : le fichier régulier ne peut avoir qu'un nom, donc `toto`.

6.3 Fichiers à trou

On pourra envisager de créer des fichiers à trou. La syntaxe de la commande de haut niveau permettant de faire ça est laissée à votre disposition, elle est à détailler dans le rapport. Idéalement
`cp //fichier_a_trou fichier_a_trou`
devra restituer le trou dans le système de fichier de la machine, s'il le supporte.

6.4 Threads de vérification

On peut utiliser des **threads** pour la vérification des SHA1 en parallèle des autres opérations. Cela est spécialement intéressant pour `fsck` mais on peut imaginer les autres commandes aient deux threads (ou davantage). Une commande qui lit plusieurs blocs (comme `cat`) peut être confiée à un premier thread (pour `cat`, ce thread affiche), tandis que un ou plusieurs autres threads vérifient les blocs et affichent un message d'erreur en cas de corruption. Ce comportement est correct : il n'est pas grave que le message arrive avant ou après l'exécution de la commande, et celle-ci n'a pas à être annulée. De même le calcul et l'écriture du SHA1, pour une commande qui écrit des blocs, peut être confiée à un ou plusieurs threads dédiés.

6.5 Autres

D'autres extensions sont bienvenues et donneront une meilleures note si elles ne sont pas triviales. Encore une fois : documentez.

7 Modalités d'évaluation

7.1 Calendrier

— Choix des équipes le 14 mars. Le sujet est à faire en binôme, ou seul sur dérogation écrite à demander par mail avant le 13 mars. **Les deux membres de l'équipe doivent envoyer un mail** à `fm@irif.fr`.
— dépôts `git` créés le 23 mars
— date de rendu : dernier commit possible le 19 mai (à 23h59 anywhere on earth)
— Soutenances du 21 au 23 mai

7.2 Discussion

Il faut s'inscrire à la liste de diffusion
`https://listes.u-paris.fr/wws/info/projet.systeme-avance.m1.info`
Toute question devra être posée sur cette liste, au lieu d'être envoyée uniquement à votre enseignant. Vous pouvez poser des questions telles que "je comprends pas pourquoi ça buggue quand..." mais ne soumettez pas de gros morceaux de code.

7.3 Attendus examinés lors de la soutenance

1. Les deux membres de l'équipe peuvent expliquer le code
2. Le projet pourra être présenté et testé sur votre machine mais doit s'exécuter correctement sur lulu.
3. Les commandes font ce qu'il faut
4. Le système de fichier, après une séquence donnée, a bien l'état prévu. Cela sera vérifié avec les `ls` et `fsck` du corrigé.
5. Le verouillage sera également vérifié. Il faut donc implémenter des commandes blocante comme `lock` ou `cat` et `input` : en effet en tapant lentement le contenu ou en l'envoyant vers un *pager* lent comme `more -n 1` (affichage ligne par ligne) on pourra vérifier que l'on a le comportement attendu : blocage tant que l'autre commande a le verrou. Par exemple si l'on fait
   `pignoufs input conteneur //toto` et dans un autre terminal
   `pignoufs cat conteneur //toto`
   alors `cat` est bloqué.

7.4 git

Chaque équipe doit créer un dépôt `git` **privé** sur le gitlab de l'UFR [4] avant le 23 mars y donner accès en tant que *Reporter* à Fabien de Montgolfier (`demontgo`).

Le dépôt devra contenir un rapport **au format pdf** donnant la liste des membres de l'équipe (avec numéro étudiant et login sur le gitlab) ainsi qu'un **Makefile** tel que l'exécution de `make` à la racine du dépôt crée (dans ce même répertoire) l'exécutable `pignoufs`. En cas d'extensions modifiant la structure de données, la cible par défaut devra respecter le format du sujet, et la cible `extensions` compilera avec les extensions.

7.5 Note

Ce qui est évalué est le respect des spécifications : plus on a implémenté correctement de choses, meilleure est la note ; la robustesse du code (détection d'erreurs internes et externes) ; et la lisibilité du code et la documentation. La vitesse n'est pas notée : `pignoufs` n'est pas prévu pour être performant. Ainsi l'usage des threads poursuit un but avant tout pédagogique : montrer qu'on sait les utiliser. On visera la lisibilité et la robustesse du code plutôt que l'efficacité. Touefois, quelque chose de gravement inefficace sans raison valable sera pénalisé.

Si un membre de l'équipe a moins d'activité `git` et répond moins bien aux questions, il sera moins bien noté, jusqu'à 0 en cas d'absence de travail, même si le projet est bon.

Pour rappel, l'emprunt de code sans citer sa source est un plagiat. L'emprunt de code en citant sa source est autorisé, mais bien sûr vous n'êtes notés que sur ce que vous avez effectivement produit. Le plagiat **même partiel** entraîne une note de 0 et une éventuelle convocation de la section disciplinaire.

7.6 Portabilité

Le sujet doit compiler sous Linux avec `gcc` sans bibliothèques exotiques. `pthread` et `crypto` sont autorisées, pour une autre bibliothèque, posez la question, la valeur par défaut est *non autorisé*. Vous avez le droit d'utiliser des appels système non-POSIX seulement s'ils sont Linux. Le sujet fait comme hypothèse que la machine est *little endian* (on vérifie que `htonl(42) != 42`). Sinon, il faudra présenter sur les machines de l'UFR pour que les tests puissent bien s'effectuer.

4. `https://moule.informatique.univ-paris-diderot.fr/`