ptar – un extracteur d'archives tar *durable* et parallèle

RS : Réseaux et Systèmes
Deuxième année

L'objectif de ce projet est de réaliser un extracteur d'archives *tar* garantissant l'écriture sur le disque des fichiers extraits. Afin d'assurer des performances raisonnables, l'extracteur utilisera plusieurs *threads* pour paralléliser les écritures sur le disque dur.

Objectifs pédagogiques :

- Lecture de fichiers dans un format binaire assez complexe (tar) ;
- Création et manipulation de répertoires et fichiers avec l'API POSIX ;
- Manipulation de threads ;
- Utilisation de Git.

1 Logistique

1.1 Comment travailler

Apprendre à travailler en groupe fait partie des objectifs de ce projet. Il vous est donc demandé de travailler en binôme. Il est **interdit** de travailler seul (sauf pour au plus un étudiant, si vous êtes un nombre impair). Un ingénieur travaille rarement seul.

1.2 Évaluation de ce projet

Tests automatiques. Une part importante de l'évaluation du projet sera faite à l'aide d'une batterie de tests. Il est donc très important, au cours de votre travail, d'accorder une large part à vos propres tests. Plusieurs ≪tests blancs≫ seront également organisés au cours du projet : votre code sera récupéré, compilé, et testé sur des cas de tests qui feront partie de ceux utilisés pour l'évaluation finale. Il est donc indispensable de commencer à travailler tôt pour bénéficier de ces ≪tests blancs≫.

Rapport. Vous devez rendre un mini-rapport de projet (5 pages maximum hors annexes et page de garde, format pdf). Vous y détaillerez vos choix de conception, les difficultés auxquelles vous avez été confronté, et comment vous les avez résolues. Vous indiquerez également le nombre d'heures passées sur les différentes étapes de ce projet (conception, codage, tests, rédaction du rapport) par membre du groupe.

Soutenances. Des soutenances pourront être organisées (éventuellement seulement pour certains groupes, par exemple s'il y a des doutes sur l'originalité du travail rendu). Vous devrez nous faire une démonstration de votre projet et être prêts à répondre à toutes les questions techniques sur le codage de l'application.

Section ≪Remerciements≫ du rapport. Votre rapport doit contenir quelque chose comme la mention suivante : ≪Nous avons réalisé ce projet sans aucune forme d'aide extérieure≫ ou bien ≪Nous avons réalisé ce projet en nous aidant des sites webs suivants (avec une liste de sites, et les informations que vous avez obtenues de chaque source)≫ ou encore ≪Nous avons réalisé ce projet avec l'aide de (nommez les personnes qui vous ont aidé, et indiquez ce qu'ils ont fait pour vous)≫. Si la liste est trop longue, vous pouvez la déplacer en annexe (pour ne pas empiéter sur vos 5 pages de rapports). Rien ne vous empêche de vous faire aider, mais il vous est demandé un minimum d'honnêteté académique en listant vos aides. Tout manquement à cette règle sera sanctionné comme il se doit.

La triche sera sévèrement punie. Par « se faire aider », on entend : avoir une discussion sur le design du code, y compris sur des détails techniques et/ou les structures de données à mettre en œuvre. Par tricher, on entend : copier ou recopier du code ou encore lire le code de quelqu'un d'autre pour s'en inspirer. Nous testerons l'originalité de votre travail par rapport aux autres projets rendus [1], et il est difficile de défendre en soutenance un projet que l'on n'a pas écrit entièrement.

1. `http://theory.stanford.edu/~aiken/moss/`

≪Rendu≫ du projet. Vous devez utiliser un dépôt Git privé sur GitHub, dans l'organisation `ln-projetu`, dont le nom commence par `rs2016-`. Pour cela :

1. Chaque membre du binôme crée un compte GitHub, s'il n'en a pas déjà un.
2. L'un des membres du binôme m'envoie un mail (à `lucas.nussbaum@loria.fr`), avec pour sujet ≪ *demande ajout ln-projetu pour RS2016* ≫, et comme contenu ≪ *mon login GitHub est monlogin-github* ≫.
3. Attendez que je vous ajoute à l'organisation `ln-projetu`.
4. Un des membres du binôme crée un nouveau *repository* sur `https://github.com/new`, en précisant bien *ln-projetu* comme *Owner*. Le nom du *repository* doit commencer par `rs2016-`, puis être suivi des deux noms des membres du binôme. Une fois le *repository* créé, l'autre membre du binôme peut être ajouté (Settings - Collaborators & Teams - Collaborators).

Le projet sera récupéré automatiquement sur votre dépôt GitHub. Les tests blancs vous permettront de vérifier que tout fonctionne bien. Il n'y a pas d'action particulière à effectuer pour ≪ rendre ≫ votre projet. À la date et heure de rendu, l'état de votre dépôt sera simplement récupéré.

Votre dépôt Git doit contenir, à la racine du dépôt :

— Un ficher `AUTHORS` listant les noms et prénoms des membres du groupe (une personne par ligne) ;
— Un `Makefile` compilant votre projet en créant un fichier exécutable nommé `ptar` ;
— Un fichier `rapport.pdf` contenant votre rapport au format PDF.

Les fichiers `AUTHORS` et `Makefile` doivent être rajoutés rapidement (ils sont indispensables pour que le projet soit testé lors des tests blancs). Le fichier `rapport.pdf` peut n'être ajouté qu'à la fin du projet.

Des informations complémentaires pourront être fournies sur `http://members.loria.fr/lnussbaum/RS/`. Ces informations complémentaires doivent être considérées comme faisant partie du sujet. Il est donc conseillé de surveiller cette page régulièrement.

Vos questions éventuelles peuvent être adressées à `lucas.nussbaum@loria.fr`. Les réponses (et les questions correspondantes) pourront être publiées sur la page du projet.

2 Description du projet

Le projet de cette année vise à développer un extracteur d'archives *tar* garantissant l'écriture sur le disque des fichiers extraits. Afin d'assurer des performances raisonnables, l'extracteur utilisera plusieurs *threads* pour paralléliser les écritures sur le disque dur.

2.1 Archives TAR

tar est à la fois un format de fichiers, et le nom de l'outil standard dans le monde Unix pour manipuler ces archives. Les fichiers *tar* sont utilisés pour rassembler un ensemble de fichiers (et de répertoires) dans un seul fichier (*tar* signifie ≪ goudron ≫). Les archives *tar* ne sont pas compressées. On utilise en général *tar* avec un compresseur (*gzip*, *bzip2*, etc.) pour obtenir des archives compressées (`tgz = tar.gz = tar + compression gzip`).

Pour plus de détails sur *tar*, consultez :

— `http://en.wikipedia.org/wiki/Tar_(file_format)`
— La page de manuel `tar(5)` (à ne pas confondre avec `tar(1)`, qui décrit la commande `tar`), dont une copie en ligne est disponible sur `https://members.loria.fr/lnussbaum/RS/tar.pdf`

2.2 Durabilité des écritures

Dans les bases de données, la durabilité (*durability*) est la propriété qui garantit qu'une transaction a été enregistrée de manière à survivre à une panne. En pratique, cela nécessite de n'indiquer la transaction comme terminée qu'une fois que les données ont été écrites sur un support durable (disque dur par exemple). La plupart des outils usuels n'offrent pas cette propriété : lorsqu'une copie de fichiers avec `cp` se termine, les données sont peut-être encore en cours d'écriture sur le disque dur par le système d'exploitation. Forcer la durabilité pour toutes les actions sur les fichiers est possible, mais cela ralentirait très fortement le fonctionnement du système, puisqu'il est nécessaire d'attendre l'écriture effective des données sur le disque dur après chaque action. L'utilisation d'écritures durables est donc limitée aux

logiciels qui le nécessitent vraiment, comme les gestionnaires de bases de données ou certains serveurs réseaux [2].

D'un point de vue pratique, une séquence `open(); write(); close();` ne garantit pas l'écriture des données sur le disque dur au moment du `close();`. Pour attendre l'écriture effective des données, il faut utiliser l'appel système `fsync()` : `open(); write(); fsync(); close();`.

2.3 Multiplexage des écritures avec des threads

L'utilisation de `fsync();` lors de l'écriture de chaque fichier limite très fortement les performances. Ainsi, pour une archive dont l'extraction prend 2 secondes avec la commande `tar`, l'extraction avec un *tar durable* prend 90 secondes. Afin de limiter ce problème, il est proposé d'utiliser des threads POSIX pour réaliser plusieurs écritures simultanément. Le système d'exploitation pourra ainsi réordonner les écritures sur le disque dur d'une manière plus efficace.

Sur l'archive mentionnée précédemment, utiliser des threads permet de diviser par trois le temps d'exécution.

Plusieurs stratégies sont possibles pour paralléliser l'extraction. Une stratégie simple est de constater que les descripteurs de fichiers sont partagés par l'ensemble des threads, et qu'il est donc possible de lire l'archive à extraire de n'importe quel thread. Chaque thread peut alors, alternativement, lire un fichier dans l'archive, avant de passer la main au thread suivant pendant qu'il écrit les données sur le disque dur.

2.4 Interface utilisateur

Une fois compilé, votre projet se présentera sous la forme d'un fichier exécutable `ptar`, décrit par sa page de manuel ci-dessous.

```
PTAR(1)                         Commandes                         PTAR(1)

NOM
       ptar - Listing et extraction d'archives TAR

SYNOPSIS
       ptar [OPTION]... [FICHIER]

DESCRIPTION
       Affiche le contenu (si aucun paramètre n'est fourni) ou, plus utile,
       extrait l'archive TAR nommée FICHIER, à partir du répertoire courant.

OPTIONS
       Aucune option n'est obligatoire.

       -x
            En plus de lister le contenu de l'archive TAR passée en paramètre,
            extrait son contenu (crée les répertoires, fichiers, définit les
            permissions, les propriétaires, les dates de modification, etc.)

       -l
            Listing détaillé. Par défaut, ptar n'affiche que les noms des
            répertoires et fichiers. Avec l'option -l, ptar affiche, séparés
            par un espace à chaque fois:
             - les permissions sous la forme drwxr-xr-x (comme dans la sortie
               de ls -l)
             - l'uid et le gid du fichier, sous la forme 1001/1001
             - la taille du fichier (en décimal)
```

2. Par exemple, un serveur SMTP (email) ne répond qu'il *accepte* un mail qu'une fois qu'il l'a réellement écrit sur le disque dur. La RFC 2821 indique (section 6.1) : *When the receiver-SMTP accepts a piece of mail (by sending a "250 OK" message in response to DATA), it is accepting responsibility for delivering or relaying the message. It must take this responsibility seriously. It MUST NOT lose the message for frivolous reasons, such as because the host later crashes or because of a predictable resource shortage.*

```
            - la date de modification, sous la forme 2016-10-13 16:44:12
            - le nom du fichier ou du répertoire (suivi d'un / si répertoire)
            - si le fichier est un lien symbolique, "-> destination"

       -p NBTHREADS
            Utilise NBTHREADS threads pour réaliser les écritures en parallèle.

       -z
            L'archive passée en paramètre est compressée avec gzip; demande à
            ptar de la décompresser avant de la traiter. ptar chargera
            dynamiquement la bibliothèque zlib (avec dlopen) pour
            effectuer la décompression.

       Toutes les options peuvent être combinées.

VALEUR DE RETOUR
       ptar s'arrête en renvoyant la valeur de retour 0 si tous les traitements
       se sont déroulés avec succès; 1 sinon.

EXEMPLE
       ptar -lxzp 8 fichier.tar.gz
            Liste le contenu détaillé de fichier.tar.gz (une archive tar
            compressée avec gzip) et extrait son contenu dans le répertoire
            courant. L'extraction se fait en utilisant 8 threads.

VOIR AUSSI
       tar(1), tar(5), zlib(3)
```

2.5 Conseils de réalisation

La quantité de code à produire est relativement faible (quelques centaines de lignes de code tout au plus). Mais le niveau de difficulté du code à produire est assez important. Il est crucial de programmer de manière prudente, réfléchie, claire et progressive, en testant bien les différents cas d'erreur.

Il est conseillé (mais pas obligatoire) de réaliser votre projet dans l'ordre qui suit.

Étape 1 : développement d'un *listeur* basique d'archives tar (sans extraction).
Pour cette première étape, il faut bien lire la page de manuel de `tar(5)`, et éventuellement compléter cette lecture par d'autres documents. Il est conseillé d'utiliser le format *POSIX ustar archives* ou *GNU tar archives*.

Il faut faire attention au fait qu'une archive *tar* est une succession de blocs de 512 octets, qu'il est préférable de lire l'un après l'autre dans leur ensemble. Pour lire les en-têtes, il suffit de faire un `read()` dont la destination est une structure (`struct header_posix_ustar ma_struct; read(fd, &ma_struct, 512);`). Faites bien attention à la description des différents champs : certaines valeurs numériques sont en octal. Dans un premier temps, ne considérez que les répertoires et fichiers, sans prendre en compte les champs tels que les permissions, propriétaire, date/heure de modification, etc.

Étape 2 : ajout du traitement des options de ligne de commande.
Il est très fortement conseillé d'utiliser `getopt(3)` pour gérer facilement tous les cas (ordre des paramètres, etc.).

Étape 3 : développement d'un extracteur de fichiers *tar* (option `-x`).
Cette étape consiste à créer les répertoires et fichiers contenus dans l'archive à l'aide des différents appels système de manipulation de répertoires et fichiers.

Étape 4 : traitement des champs de l'archive non considérés jusque là (option `-l`).
(mode (= permissions), uid/gid (= propriétaire/groupe), taille, mtime, liens symboliques). Ces attributs doivent aussi être appliqués lors de l'extraction (pas uniquement lors du listing). Bien sûr, l'utilisation de `system()` n'est pas autorisée.

Étape 5 : ajout de la *durabilité* et parallélisation en utilisant les threads (option `-p`).
Cette étape consiste à s'assurer que les fichiers sont bien écrits sur le disque dur en ajoutant les appels à `fsync()`, puis à utiliser plusieurs threads pour écrire les données.

Étape 6 : gestion des archives compressées (option `-z`).
chargement dynamique (avec `dlopen`, vu en cours) de la bibliothèque zlib (documentation sur `http://www.zlib.net/`).

Étape 7 : vérification de la cohérence des données avec le champ `checksum`.
Si l'archive a été corrompue, sortir avec une erreur (code de retour 1).

Ces étapes sont assez indépendantes, et peuvent être utilisées pour répartir le travail dans le binôme. Toutefois, il est conseillé de terminer les étapes 1 à 4 avant de passer à la suite.

Barème indicatif :
- Binôme réalisant parfaitement les étapes 1 à 3 : 9/20
- Étape 4 complète : +4 points
- Étape 5 complète : +4 points
- Étape 6 complète : +2 points
- Étape 7 complète : +1 point

3 Calendrier

- La version finale de votre projet sera récupérée le **vendredi 16/12/2016 à 16h00**. Il n'y aura aucune extension, ni possibilité de corriger votre code après cette date.
- Il est très vivement conseillé de commencer tôt, pour profiter au maximum des tests blancs qui vous permettront d'avoir un retour sur le fonctionnement de votre programme. Le premier test blanc sera lancé le **jeudi 27/10 à 8h**. Il y aura 3 à 4 tests blancs répartis pendant la durée du projet, à des dates précisées ultérieurement sur la page web du module RS.