

TD 2: Premiers pas avec Git

ED6 — Licence 3 — Université Paris Diderot

Séances du 17 et 24 février 2014

Lors de la précédente séance de TP, nous avons travaillé sur la manipulation de fichiers sources. Cette séance, nous allons nous initier à la maintenance d'un répertoire de travail de manière structurée et ordonnée. Nous en profiterons également pour effectuer quelques révisions de shell-scripting, et commencerons à travailler sur l'écriture de programmes C en plusieurs fichiers.

1 Premiers pas avec Git

Git, UN LOGICIEL DE CONTRÔLE DE VERSIONS



Git est un logiciel de contrôle de versions, il permet de sauvegarder l'historique du contenu d'un répertoire de travail. Pour ce faire l'utilisateur doit régulièrement enregistrer (en créant une révision ou *commit*) les modifications apportées au répertoire, il pourra ensuite accéder à l'historique de toutes les modifications et inspecter l'état du dossier à chaque révision.

Git a la particularité de permettre de créer une copie d'un répertoire de travail et de synchroniser entre eux plusieurs copies du même répertoire, permettant décentraliser le travail.

Lors du devoir pour aujourd'hui, dont le sujet est disponible aussi sur

<https://upsilon.cc/~zack/teaching/1314/ed6/tp-1-devoir.pdf> (1)

vous avez eu accès à la plate-forme d'hébergement GitLab

<http://moule.informatique.univ-paris-diderot.fr:8080/> (2)

en utilisant le protocole `ssh` (authentification au moyen de votre clef publique), puis vous avez créé un projet sur GitLab (c'est-à-dire un dépôt Git) et vous avez récupéré une copie de ce projet sur votre répertoire personnel. Assurez-vous d'avoir accompli toutes les tâches du devoir pour aujourd'hui (1).

Pour chaque séance de TP, vous devez créer un nouveau projet sur GitLab nommé `tp1`, `tp2`, etc. selon le numéro de séance, ou bien vous pouvez créer un seul projet pour tout le cours nommé `ed6` et organisé avec des répertoires `tp1`, `tp2`, etc.

Question 1.1. Création et récupération d'un projet. Créez un nouveau projet sur GitLab (2) nommé `tp2`.

Récupérez une copie de ce projet sur votre répertoire personnel de votre machine au moyen de la commande

```
$ git clone git@moule.informatique.univ-paris-diderot.fr:votre_identif/votre_projet.git
```

où `votre_identif` est votre nom utilisateur dans le compte LDAP et `votre_projet` est le nom que vous avez donné à votre projet, donc `tp2` dans ce cas. Quel répertoire a été créé ? Que contient-il (y compris les répertoires cachés) ?

Question 1.2. Une première révision. Placez-vous dans le répertoire créé par la commande `git clone` ci-dessus (c'est le répertoire de travail de votre dépôt). Créez un fichier `description.txt`. Éditez-le pour y ajouter une courte description du fait que vous allez utiliser ce dépôt pour le TP2 du cours de ED6. Ajoutez ce fichier à l'index du répertoire au moyen de la commande

```
$ git add nom_fichier
```

où `nom_fichier` est le nom du fichier que vous venez de créer, donc `description.txt` dans ce cas.

Commitez ce changement au moyen de la commande

```
$ git commit -m 'premiere revision'
```

Allez sur la page de votre projet `tp2` dans GitLab (2). Le fichier `description.txt` est-il présent ? En cliquant sur l'onglet Files dans la page de votre projet, vous pouvez voir les fichiers qui ont été envoyés sur le serveur de GitLab.

Question 1.3. Modifications. Éditez le fichier `description` pour y préciser votre nom et prénom.

Depuis le répertoire de votre dépôt, lancez `Gitg` (ou lancez `Gitg` puis ouvrez votre dépôt) au moyen de la commande `$ gitg` sur le shell. `Gitg` a deux onglets `History` et `Commit`. Dans l'onglet `Commit`, on remarque 4 cadres :

- `Unstaged` qui contient la liste des modifications qui ont été apportées dans le dépôt et qui n'ont pas été sélectionnées pour être committées,
- `Staged` qui contient la liste des modifications qui ont été apportées et qui ont été sélectionnées pour être committées,
- `Changes` qui affiche une modification,
- `Commit message` qui contient le message du commit courant.

Question 1.4. Une seconde révision. Commit à l'aide de Gitg. À l'aide de `Gitg`, commitez votre modification. Que se passe-t-il si vous commitez sans donner un message de commit ? Commitez avec le message "deuxieme revision". Que s'est-il passé sur `GitLab` (2) ? Quels fichiers sont présents sur `GitLab` ?

Question 1.5. Seconde copie de travail. Dans un nouveau répertoire `/tmp`, exécutez à nouveau la commande

```
$ git clone git@moule.informatique.univ-paris-diderot.fr:votre_identif/tp2.git
```

Que contient le répertoire créé ? Pourquoi ?

Question 1.6. Envoi des données sur le serveur de GitLab. Dans la copie de travail où vous avez créé et commité le fichier `description.txt`, exécutez la commande

```
$ git push origin master
```

pour envoyer pour la première fois au serveur `GitLab` les modification au projet que vous avez commité.¹ Cette fois, que s'est-il passé sur `GitLab` (2) ?

Question 1.7. Mise à jour de la deuxième copie de travail. Exécutez dans `/tmp/tp2` la commande

```
$ git pull
```

Que s'est-il passé dans le répertoire `/tmp/tp2` ? Dans le répertoire `/tmp/tp2`, supprimez le fichier `description.txt` : comment pouvez-vous le récupérer depuis `GitLab` ? Sinon, essayez la commande `git checkout -- description.txt`.

Question 1.8. Ajout de répertoire. Retournez dans la première copie de travail, téléchargez au moyen de la commande

```
$ wget http://upsilon.cc/~zack/teaching/1314/ed6/tp1-files.tar
```

les fichiers sur lesquels vous aviez travaillé pendant la première séance de TP.

Décompressez l'archive (idéalement au moyen de l'invocation idoine de `tar`). Commitez l'ensemble des fichiers créés, à l'exception de l'archive (question 1.4). Envoyez ces modifications sur `GitLab` (question 1.6).

Question 1.9. Contenu du dépôt. Dans la copie de travail `/tmp/tp2`, exécutez la commande

```
$ git pull
```

Quel est le contenu du répertoire ? Pourquoi ?

Question 1.10. Ajout du travail de la première séance. Retournez dans le répertoire de votre première copie de travail : copiez-y, en écrasant les fichiers que vous venez de décompresser (question 1.8), les fichiers sources que vous avez obtenus en fin de première séance de TP.

Question 1.11. Résumé des modifications. Exécutez la commande

```
$ git status
```

Comment interprétez-vous le résultat ?

Question 1.12. Commit en ligne de commande. Exécutez la commande

```
$ git commit
```

Que se passe-t-il ? Essayez les commandes `git commit --interactive` et `git add`.²

Question 1.13. Historique du répertoire. Lancez `Gitg`. Comment pouvez-vous accéder dans `Gitg` au contenu du fichier `chunk.c` juste après que vous ayez décompressé l'archive ? Et dans `GitLab` ?

Est-il possible de voir les différences entre les deux versions du fichier `chunk.c` dans `Gitg` ? Et dans `GitLab` ?

Dorénavant, vous travaillerez systématiquement dans votre dépôt, en commitant (avec des messages explicatifs) de temps en temps afin de conserver un historique de votre travail. N'oubliez pas d'envoyer vos modifications sur `GitLab`.

1. Cette syntaxe longue est nécessaire la première fois que vous faites `push` pour indiquer ou, de préférence, vous voulez envoyer vos changements. Pour les fois successives, la syntaxe plus courte `git push` sera suffisante.

2. En général, c'est la commande `git add nom_fichier1 nom_fichier2 nom_fichier3` qui sélectionne, parmi tous les fichiers ajoutés ou modifiés dans votre copie de travail du dépôt Git, les fichiers `nom_fichier1`, `nom_fichier2` et `nom_fichier3` pour qu'ils soient committés lors de la prochaine commande `git commit`.

2 Un peu de shell-scripting

Question 2.1. Dans le répertoire `tp2`, créez un répertoire `scripts`, dans lequel nous écrirons quelques scripts.

Lorsqu'on programme en C, il est bon d'avoir le réflexe de systématiquement créer pour chaque fichier `.c`, un fichier `.h` correspondant, et de l'inclure dans le fichier `.c`.

Question 2.2. Écrire un script `check_headers.sh` qui vérifie que dans le répertoire courant, à chaque fichier `.c`, il existe un fichier `.h` correspondant. Vous pourrez utiliser la commande `basename`.

Connaissez-vous une manière de se passer de l'utilisation de `basename` ? Commitez ce script.

Question 2.3. Modifiez votre script pour qu'en plus il teste si dans chaque fichier `.c` pour lequel le fichier `.h` existe, il vérifie que le fichier `.c` contient bien le `#include` correspondant. Vous pourrez utiliser la commande `grep`. Commitez vos modifications en mettant un message de commit qui décrive brièvement la modification apportée à votre script.

Question 2.4. Pour aller plus loin. Créez un script `create_headers.sh`, qui pour chaque fichier `.c` qui n'a pas de fichier `.h` associé, crée le fichier `.h` correspondant, et ajoute au fichier `.c` l'entête `#include "machin.h"` correspondante. Idéalement, cette inclusion doit être la dernière du fichier `.c`.

Question 2.5. Écrire un script `headers.sh` qui invoqué sans option appelle le script `check_headers.sh`, et qui avec l'option `-c` crée les fichiers d'entête manquants (avec un appel à `create_headers.sh`).

3 Quelques fonctions de la Glibc



LA BIBLIOTHÈQUE STANDARD C

La bibliothèque standard C est une composante essentielle d'un système d'exploitation de type UNIX. C'est cette bibliothèque qui est censée fournir tous les appels système, i.e. toutes les fonctions nécessaires pour interagir avec le système d'exploitation (manipulation de fichiers, envoi de signaux entre processus, etc.). La Glibc, bibliothèque standard C de GNU/Linux présente ainsi plus d'un millier de fonctions.

1. On vous donne le code C suivant :

```
void skel(char *buf) {
    char *p = buf;
    while (1) {
        printf("%d\n", atoi(p));
        if ((p = index(p, ' ')) == NULL)
            break;
        else
            p++;
    }
}

int main(void) {
    skel("1 2 3 4 5 6 7 8 9 10");
    exit(0);
}
```

Que fait-il ? (vous pourrez vous aider de la documentation des fonctions de la bibliothèque standard `atoi` et `index`, accessible avec `man 3 atoi` et `man 3 index`.)

- À l'aide du code qui vous a été donné, dans le répertoire `tp2/exosC`, écrire dans un fichier `statistiques.c` une fonction `int* eval_string(char* buf)` qui prend en argument une chaîne de caractères pour laquelle on supposera qu'elle contient une liste d'entiers écrits en base 10 et séparés par des espaces. Cette fonction renverra un tableau contenant dans sa première case le nombre d'entiers lus et dans les suivantes chacun de ces entiers.
- Définir une constante `MAX_LINE_LENGTH` égale à 200. Lire la documentation de `fgets(3)`. Écrire une fonction `main` qui lit une ligne de l'entrée standard (on supposera qu'elle ne contient que des entiers séparés par des espaces) et affiche la moyenne des entiers présents sur celle-ci. Si la ligne a plus de `MAX_LINE_LENGTH` caractères, le programme doit s'arrêter et renvoyer un code d'erreur -1.
- Préprocessez votre fichier `statistiques.c` avec `gcc -E statistiques.c -o statistiques.i`. Où se trouve le fichier `stdio.h` ? Quel est le code de retour de la commande `grep MAX_LINE_LENGTH statistiques.i` et de `grep MAX_LINE_LENGTH statistiques.c` ?

- Poursuivez la compilation de votre programme avec `gcc -Wall statistiques.i -o statistiques`. Testez votre programme.



Pour que les programmes puissent exécuter des fonctions de la bibliothèque standard, il existe une version compilée de toutes les fonctions de la bibliothèque standard (habituellement dans le fichier `/lib/libc.so.6`). Lors de la création d'un exécutable, `gcc` effectue automatiquement une liaison dynamique avec cette version compilée de la bibliothèque C standard.

- Modifiez le `main` pour qu'il lise indéfiniment des lignes, en ayant le même comportement. Compilez, testez.
- Si l'on veut que le programme lise une seule ligne ou qu'au contraire, il en lise indéfiniment, il faut modifier le fichier source, puis le recompiler. Nous allons utiliser les flags d'exécution pour pouvoir faire ce choix non plus à la compilation, mais à l'exécution.

Ainsi, on veut que `./statistiques` lise indéfiniment des lignes, mais `./statistiques -1` ne lira qu'une seule ligne. Si une autre option est lue, le programme doit afficher un message d'erreur explicitant l'utilisation de ce programme. Lire la documentation de `getopt(3)`, puis implémentez ce comportement. A-t-on besoin d'écrire `#include <getopt.h>`? Compilez, testez.

- Modifiez votre programme pour qu'il affiche la moyenne et/ou l'écart type pour chaque ligne selon que le programme ait un flag `-m` et/ou `-e`. On rappelle que l'écart type d'un ensemble de n éléments (x_1, \dots, x_n) dont

la moyenne est \bar{x} est donnée par la formule
$$\sqrt{\sum_{i=1}^n \frac{(x_i - \bar{x})^2}{n}}$$
.

Compilez, testez, libérez-vous la mémoire que vous allouez ?



Bien que la Glibc comporte de nombreuses fonctions mathématiques (`sqrt`, `cosl`, `exp`, etc.), elles n'ont pas été compilées dans `/lib/libc.so.6`, mais dans `/lib/libm.so.6` (avec un `m` comme dans "maths"). Comment demander à `gcc` de lier votre exécutable vis-à-vis de ce fichier ?

- Compilez statiquement vis-à-vis de la bibliothèque `math`³. Quelle est la taille de l'exécutable obtenu ? En quoi les sorties de `ldd` et `nm` diffèrent-elles sur les exécutables statiques et dynamiques ?

4 Utilisation de la bibliothèque `readline`

READLINE, POUR LES INTERFACES EN LIGNE DE COMMANDE



Sous les systèmes de type UNIX, lorsqu'un programme est exécuté, trois flux sont automatiquement créés : l'entrée standard `stdin` (en lecture seule pour le programme) et les sorties standard `stdout` et d'erreur `stderr` (en écriture seule pour le programme). Lorsque le programme est lancé depuis un terminal (avec un interpréteur de commande ou shell), les entrées clavier sont connectées à l'entrée standard (et bien souvent aussi au display afin que l'utilisateur dispose d'un contrôle visuel sur ses entrées clavier), et les sorties standard et d'erreur sont connectés au display. Comme le clavier est relié directement à l'entrée standard, il n'y a pas d'édition possible de la part de l'utilisateur. GNU/readline permet d'ajouter des fonctions d'édition et d'historique à ces interfaces en ligne de commande.

- Lire la documentation de la fonction `char* readline (const char *prompt)` dans la page de manuel `readline(3)`. Modifiez `statistiques.c` pour utiliser `readline` au lieu de `fgets` pour lire sur l'entrée standard. Compilez, testez.
- Quels sont les fichiers supplémentaires nécessaires pour compiler votre programme ? Quels sont les fichiers supplémentaires nécessaires pour exécuter votre programme ?
- Utilisez une macro préprocesseur `USE_READLINE`, qui fait que votre code n'utilise `readline` que si elle est définie (votre code utilise `fgets` lorsqu'elle n'est pas définie).
- Consulter la documentation — à l'aide de `info readline` dans un terminal ou dans Emacs en cherchant `readline` dans le buffer créé par `C-h i`—et déduisez de la section 2.1 comment utiliser la fonction `add_history` pour activer l'historique. Compilez et testez cette nouvelle fonctionnalité

3. avec l'option `-static` qui doit précéder les `-l` de chaque bibliothèque que l'on souhaite statiquement lier.

5 Utilisation de la bibliothèque ncurses

NCURSES, POUR DES INTERFACES EN MODE TEXTE



La bibliothèque `ncurses` est une bibliothèque du projet GNU (donc libre) fournissant une API pour le développement d'interfaces en mode texte.

Une interface en mode texte se démarque d'une interface en ligne de commande (comme celle fournie par `readline`), car l'interface utilisateur occupe la totalité de l'écran, et n'est donc pas limité au traitement ligne par ligne.

Elle offre notamment la possibilité d'afficher un caractère en tout point de l'écran, ce que cet exercice se propose d'exploiter.

1. Consulter la page de manuel de `ncurses(3ncurses)`. Comme beaucoup de bibliothèques, pour être utilisée elle nécessite un appel à une fonction d'initialisation et doit effectuer un appel à une fonction de nettoyage avant que le programme ne termine.
2. À l'aide de la fonction `mvaddnstr` écrire à une position aléatoire sur l'écran une chaîne de caractères.
Indice : `refresh(3ncurses)`, `rand(3)`
3. Modifiez votre programme pour qu'il effectue 100 fois l'opération suivante : afficher la chaîne à une position aléatoire, attendre 500ms, effacer l'écran.
Indice : `erase(3ncurses)`, `usleep(3)`
4. Modifiez votre programme pour qu'il boucle jusqu'à ce que l'utilisateur appuie sur une touche
Indice : `cbreak(3ncurses)`, `nodelay(3ncurses)`, `noecho(3ncurses)`, `getch(3ncurses)`

6 Une (toute) petite bibliothèque partagée

1. Dans un répertoire `libhello`, écrire dans un fichier `hello.c` une fonction `void hello()` qui affiche une salutation⁴ sur la sortie standard. Exportez l'entête de cette fonction dans `hello.h`.
2. Nous allons maintenant créer à partir de ce fichier C une bibliothèque dynamique. Compilez avec la commande `gcc -shared hello.c -o libhello.so`⁵
Quelles sont les sorties de `ldd` et `nm` sur `libhello.so` ?
3. **Dans un autre répertoire** créez un fichier `test-hello.c` qui appelle la fonction `hello` dans le `main`. Quel(s) fichier(s) `.h` faut-il inclure ? Quel type d'`#include` faut-il utiliser ?
4. Préprocessez votre fichier (avec l'option `-E` de `gcc`), est-il normal que le compilateur ne trouve pas `hello.h`. Utilisez l'option `-I` de `gcc` pour corriger ce problème.
5. Compilez maintenant votre fichier `test-hello.c`, à quelle étape votre compilation échoue-t-elle ?
Il faut montrer au compilateur où se trouve notre bibliothèque : utilisez les options `-l` et `-L` pour lui indiquer `libhello.so`.
6. Que se passe-t-il lorsqu'on essaie de lancer l'exécutable produit ? Que renvoie `ldd` ?
7. Exécutez votre programme (supposément nommé `test-hello`) avec l'invocation shell suivante :
`LD_LIBRARY_PATH=/home/moi/ed6/tp2/libhello ./test-hello`

4. En panne d'inspiration ? Essayez-donc `C-h C-h` dans Emacs.

5. L'option `-shared` demande à `gcc` de créer un "shared object", cet objet est écrit dans le fichier `libhello.so` : le `.so` indique qu'il s'agit d'une bibliothèque partagée, et le préfixe `lib` permettra à `gcc` de lier vis-à-vis de cet objet.