LEPL1503

Version 2021

O. Bonaventure, G. Detal, C. Paasch

févr. 16, 2022

Table des matières

CHAPITRE 1

Introduction

Outre des compétences théoriques qui sont abordées dans d'autres parties de ce document, la maîtrise d'une système informatique implique également une excellente connaissance des outils informatiques qui sont inclus dans ce système. Cette connaissance se construit bien entendu en utilisant activement ces logiciels. Cette section comprend les informations de bases sur quelques logiciels importants qui sont utilisés dans le cadre du cours. Elle est volontairement réduite car de nombreux logiciels pourraient faire l'objet de livres complets. Les étudiants sont invités à proposer des améliorations à cette section sous la forme de pull-requests via https://github.com/obonaventure/SyllabusC.

CHAPITRE 2

Éditeurs

De nombreux éditeurs sont utilisables pour manipuler efficacement du code source en langage C. Chaque étudiant doit choisir l'éditeur qui lui convient le mieux. En voici quelques uns :

- vi(1) est un des premiers éditeurs à avoir été écrit pour Unix. Il a fortement évolué, et reste un outil de choix pour de nombreux administrateurs systèmes. De nombreux tutoriels permettent d'apprendre rapidement vi(1), dont https://developer.ibm.com/tutorials/l-vi/
- emacs est un autre éditeur fréquemment utilisé sous Unix. Il existe de très nombreuses extensions à emacs qui lui permettent de faire tout ou presque, y compris de jouer à des jeux comme Tetris. Son extensibilité peut rebuter certains utilisateurs. De nombreux tutoriels sont disponibles sur Internet, dont https://www.gnu.org/software/emacs/tour/
- gedit est l'éditeur de base dans l'environnement graphique GNOME utilisé dans les distributions Linux.
- Sublime Text est un éditeur de texte léger et facile d'utilisation. La version complète est payante, mais il est possible d'utiliser la version gratuite à volonté. Cet éditeur de texte est un bon choix pour les débutants qui n'auraient jamais touché à la programmation.
- Atom est un éditeur de texte facile d'utilisation et proposant de nombreuses extensions. Il a été développé par l'équipe de GitHub, et profite donc d'une interface simple pour le version control des projets en utilisant Git (voir la section consacrée à Git).
- Visual Studio Code est un éditeur de texte proposant également de nombreuses extensions, et régulièrement utilisé dans l'entreprise. Il permet, entre autres, de profiter d'un terminal de commandes sur la même fenêtre que l'éditeur de texte.
- eclipse est un environnement complet de développement écrit en Java pour supporter initialement ce langage. De nombreuses extensions à eclipse existent, dont CDT qui permet la manipulation efficace de code source en langages C et C++.

Pour les débutants, les éditeurs conseillés sont Sublime Text pour sa simplicité, Atom pour son intégration de Git (qui sera largement utilisé lors du projet), ou Visual Studio Code pour sa popularité et son accès direct à un terminal.

CHAPITRE 3

Gestionnaires de code partagé

Dans de nombreux projets informatiques, il est nécessaire d'utiliser des outils qui permettent d'organiser efficacement le partage du code source entre plusieurs développeurs. On parle de **version control**. Les plus anciens gestionnaires de code sont cvs ou rcs. Ces logiciels ont été créés lorsque Unix était utilisé sur des mini-ordinateurs qui servaient tout un département. Aujourd'hui, les logiciels de gestion de code source s'utilisent en combinaison avec des serveurs Internet pour permettre un partage large du code source. Les plus connus sont :

- Git qui est décrit dans ce syllabus. Il s'agit de l'outil de version control le plus utilisé, et donc également celui que vous utiliserez. Il est aussi décrit dans plein d'autres ressources comme git-scm contenant le livre Pro Git, une vidéo faite par le créateur de ce sites, ou encore la documentation de GitHub.
- subversion qui est décrit dans ce syllabus également.
- mercurial
- bazaar

CHAPITRE 4

Compilateurs

Le compilateur C utilisé dans de nombreuses distributions Linux est gcc(1). C'est un compilateur open-source développé activement dans le cadre du projet GNU par la Free Software Foundation. Nous utiliserons principalement gcc(1) dans le cadre de ce cours.

Il existe des alternatives à gcc(1) comme llvm que nous utiliserons lorsque nous analyserons le code assembleur généré par un compilateur C. Les variantes commerciales de Unix utilisent généralement des compilateurs propriétaires, dont par exemple Oracle Studio ou la suite de compilateurs développée par intel.

CHAPITRE 5

Disposer d'une machine Linux

Pour les cours LEPL1503 et LINFO1252, chaque étudiant doit avoir accès à une machine tournant sous un système d'exploitation Linux. Ceci est nécessaire pour les exercices et projets de ces cours, qui utilisent le langage C, et savoir utiliser une machine Linux, notamment en ligne de commande, est un acquis d'apprentissage. Ce document va donc détailler 4 manières d'avoir accès à une machine Linux, chacune avec un niveau différent de compromis entre facilité d'installation et confort d'utilisation. Ces 4 possibilités sont les suivantes, depuis la plus simple d'installation (mais la moins confortable d'utilisation) à la plus complexe :

- Accéder à l'image d'une machine des salles informatiques de l'université depuis le navigateur.
- Utiliser Windows Subsystem for Linux
- Installer une machine virtuelle Linux
- Installer un dual-boot Linux

Ces solutions sont toutes possibles pour un ordinateur utilisant Windows, mais seules les première et troisième sont disponibles pour MacOS (du moins, de manière simple).

5.1 Accéder à UDS depuis le navigateur

UDS est un service en ligne proposé par l'université, qui permet d'accéder, grâce à une connexion internet, à une machine équivalente à celles des machines Linux des salles informatiques, depuis son navigateur. Cette solution a l'avantage de ne nécessiter aucune installation, mais elle nécessite une bonne connexion Internet pour tourner de manière stable et fluide. Elle est donc conseillée pour les étudiants ayant une bonne connexion Internet. Pour les autres, les solutions suivantes sont plus indiquées.

Pour accéder à UDS, commençons par se rendre sur l'URL https://uds.siws.ucl.ac.be. Connectez-vous en indiquant votre **adresse email UCLouvain**, et votre mot de passe :

Ensuite, sélectionnez la machine Didac-ingi (pas Didac-Générique !) :

Connectez-vous à la machine en indiquant votre **identifiant UCLouvain** (pas l'adresse email !) et votre mot de passe :

Vous avez désormais accès à une machine Linux (pour être plus précis, Fedora) !

Note : Sauvegarder des fichiers sur UDS

Lorsqu'on utilise UDS, la machine est réinitialisée entre chaque connexion. Cela signifie qu'à chaque connexion, la machine est remise à son état initial, et tous les fichiers créés sont supprimés. Pour sauvegarder les fichiers, il faut donc les enregistrer sur son espace personnel UCLouvain. Cet espace est disponible depuis toutes les machines de l'UCLouvain, et son contenu est sauvegardé entre chaque connexion. Il se trouve à l'emplacement `/home/user/oasis` sur UDS.

Fig. 1 – Page de connexion d'UDS

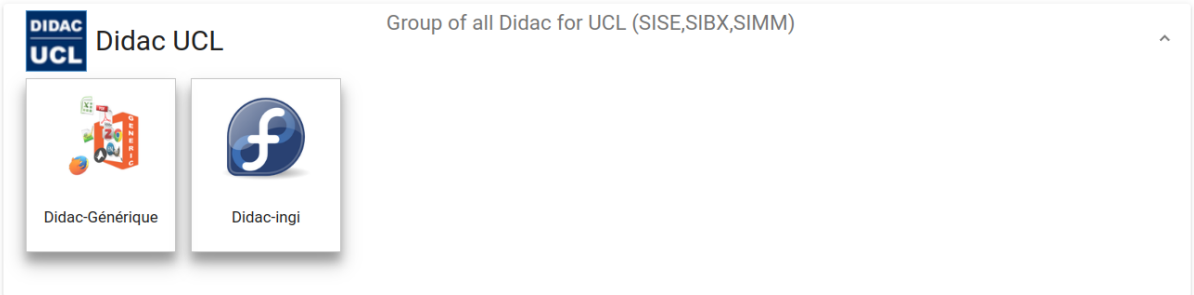


Fig. 2 – Choix de la machine UDS

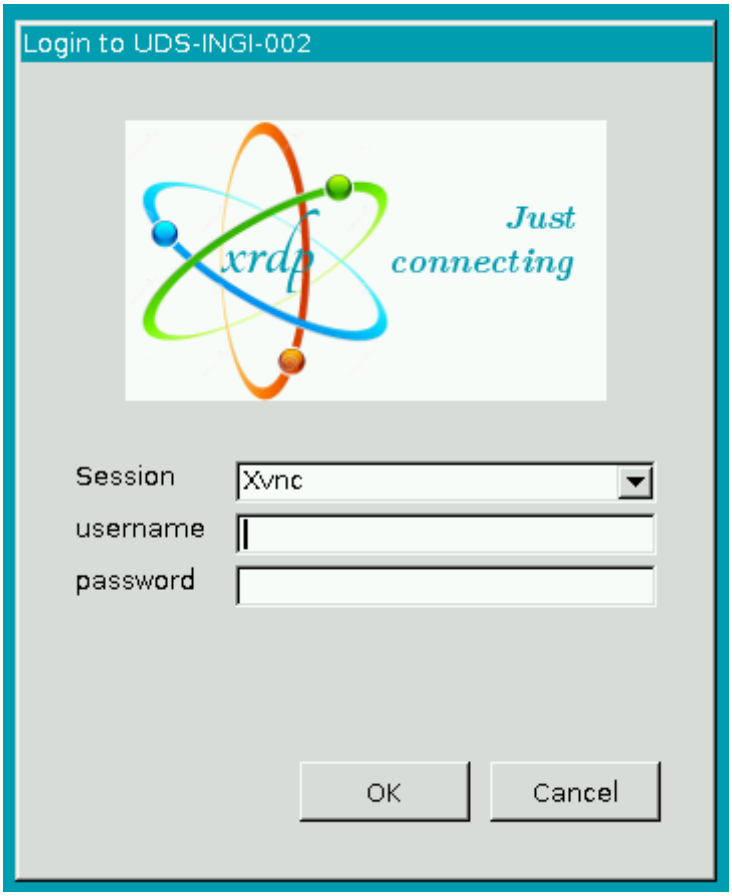


Fig. 3 – Connexion à la machine UDS

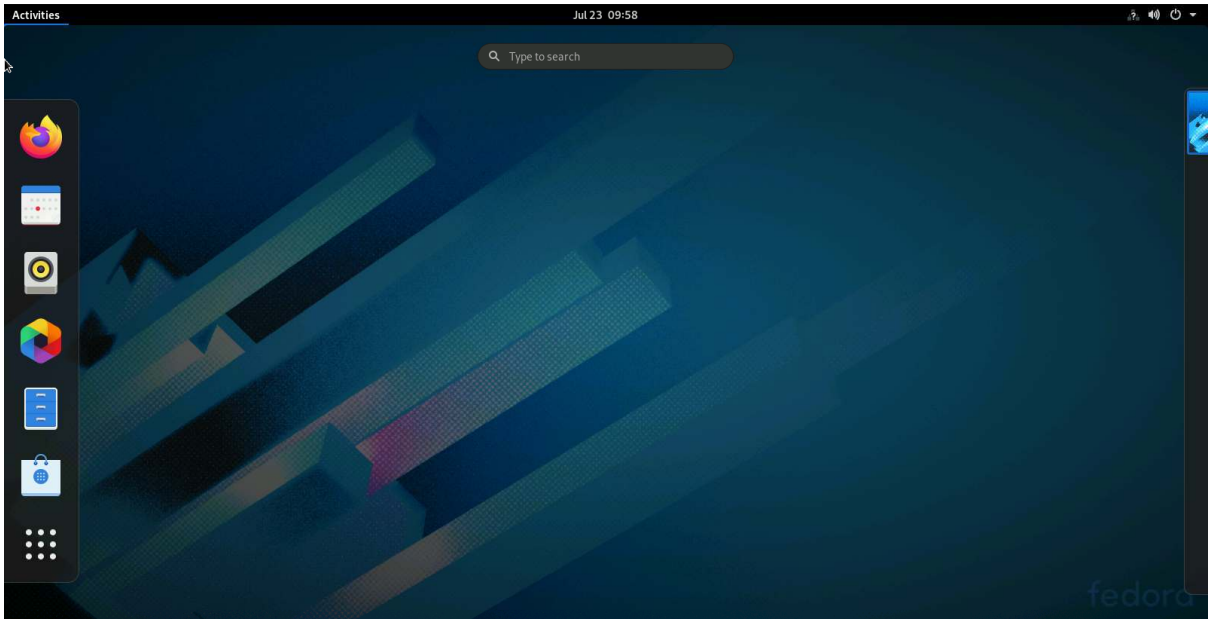


Fig. 4 – Machine Linux (Fedora) depuis le navigateur

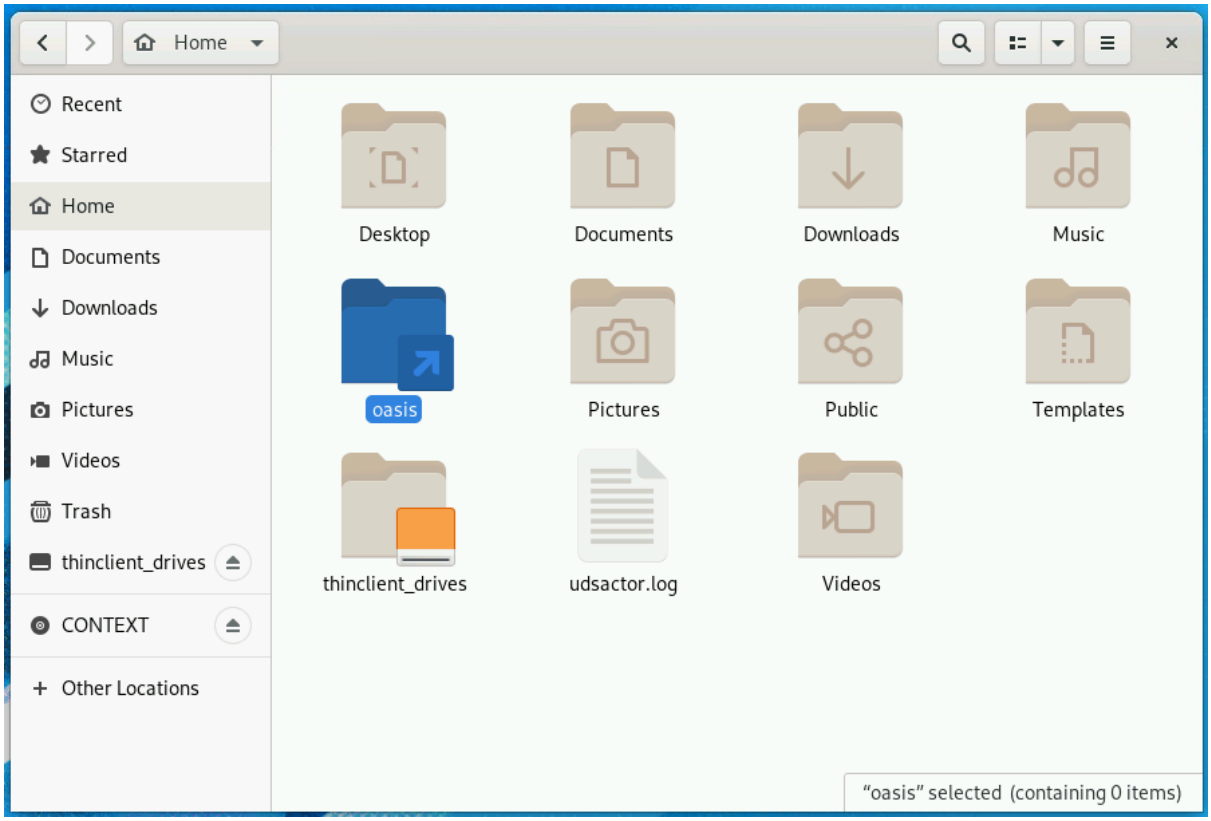


Fig. 5 – Dossier `oasis` sur la machine UDS

5.2 Utiliser Windows Subsystem for Linux

Sur les ordinateurs Windows, il existe un moyen d'obtenir un terminal Linux léger, sur lequel on peut exécuter des commandes bash comme sur une machine Linux classique. Ce terminal, qui s'appelle Windows Subsystem for Linux (WSL), ne propose cependant pas d'interface graphique, on ne peut donc interagir qu'en ligne de commande. Cela peut rebuter certains utilisateurs, mais cela permet d'avoir un système très léger et donc très rapide, et dont l'installation est très facile. De plus, cette solution ne fonctionne que sur les machines Windows. Les instructions d'installation mentionnées ci-après proviennent de https://lecrabeinfo.net/installer-wsl-windows-subsystem-for-linux-sur-windows-10.html

Pour installer WSL, il faut d'abord activer la fonctionnalité dans les paramètres Windows. Pour ce faire, commencez par ouvrir l'outil *Fonctionnalités de Windows*, depuis *Paramètres* > *Applications* > *Applications et fonctionnalités* > *Fonctionnalités facultatives* > *Plus de fonctionnalités Windows*. Cochez la case « Sous-système Windows pour Linux » puis cliquez sur *OK*. Vous devrez redémarrer pour finaliser l'installation de la fonctionnalité.

Il est également recommandé de définir WSL 2 comme version par défaut. Pour ce faire, dans PowerShell, entrez la commande suivante :

```
PS > wsl --set-default-version 2
```

Il ne reste plus qu'à installer la distribution Linux voulue depuis le Windows Store. Il est conseillé de choisir « Ubuntu », car c'est une distribution de Linux très répandue et facile d'utilisation. Depuis le Windows Store, rechercher « Ubuntu », et l'installer.

Une fois l'installation de l'application terminée, Ubuntu doit encore installer toutes les librairies nécessaires, et ne peut donc pas être utilisé directement. Un peu de patience !

Une fois l'installation réellement terminée, vous aurez accès à un terminal Ubuntu, comme si vous étiez sur une machine Linux. Toutes les commandes bash vous seront donc accessibles, notamment les commandes de gestion

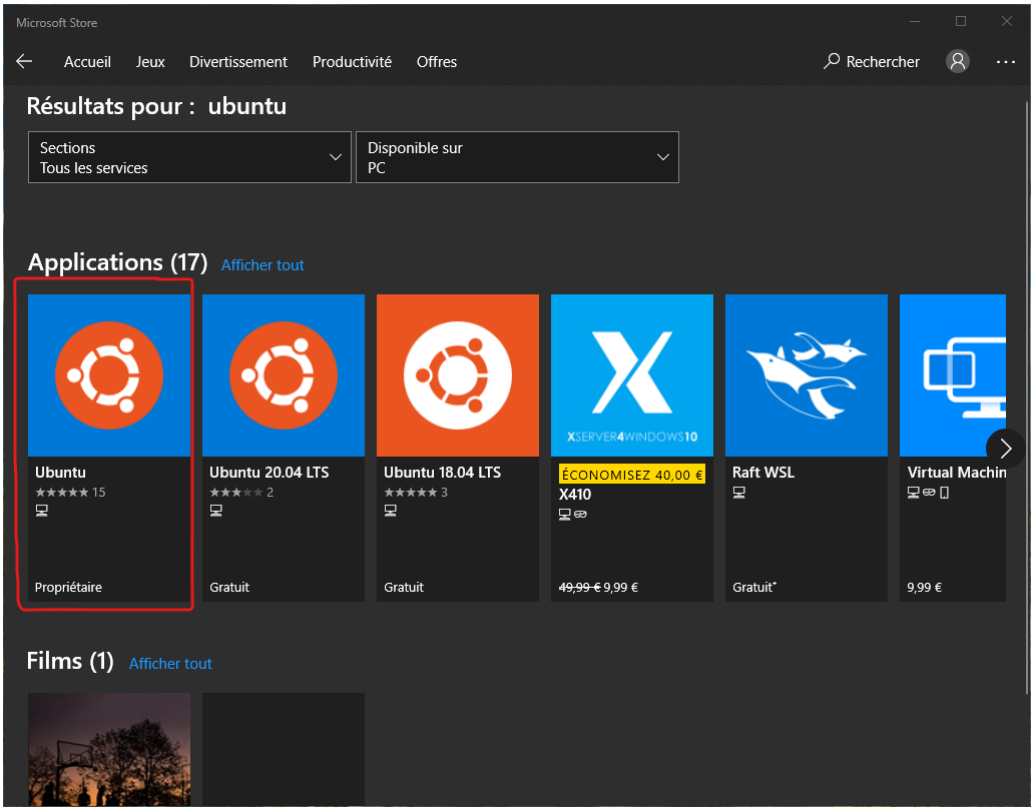


Fig. 6 – Application « Ubuntu » depuis le Windows Store

des dossiers et de compilation de programmes C.

Note : Accéder aux fichiers de Windows depuis WSL

Lorsqu'on utilise WSL, il n'y a pas d'interface graphique pour, par exemple, utiliser un éditeur de texte tel que Atom pour coder. On peut cependant produire les fichiers textes nécessaires depuis Windows directement, puis y accéder depuis WSL pour la compilation et l'exécution. Or, puisque WSL possède son propre système de fichiers Linux, il faut donc creuser un peu pour retrouver les fichiers Windows. Les disques Windows sont accessibles depuis le dossier `/mnt`, suivi de la lettre du disque. Par exemple, le disque C est accessible depuis `/mnt/c`. Depuis ces disques, vous pouvez retrouver tous les fichiers disponibles sur Windows. Pour plus de facilité, il est conseillé de créer un raccourci depuis le dossier d'accueil de WSL vers les dossiers Windows désirés. Pour ce faire, on utilise la commande ln(1) :

```
$ ln -s TARGET_FILE LINK_NAME
```

Dans ce cas, `TARGET_FILE` sera le fichier ou dossier Windows voulu (accessible à partir de `/mnt`), et `LINK_NAME` sera l'emplacement voulu du raccourci (pour le mettre sur le dossier d'accueil de WSL, ce sera `~/target_file`).

5.3 Machine virtuelle Linux

Une des façons les plus populaires d'avoir accès à une autre machine depuis sa propre machine, est d'utiliser une machine virtuelle, ou VM (pour *virtual machine*). Il s'agit d'une machine émulée, grâce à un logiciel dédié, sur une machine physique. Cela permet d'avoir accès à certains systèmes d'exploitations ou certaines machines sans devoir les installer de manière physique. Ce genre d'installation est donc plus facile, mais puisque la machine est émulée, la performance est moins élevée. Il s'agit d'un bon compromis entre performance et facilité d'installation, si on ne veut pas installer complètement un nouveau système d'exploitation sur sa machine.

Pour disposer d'une machine virtuelle, il faut tout d'abord obtenir le logiciel d'émulation. Le plus populaire est VirtualBox. Il est open-source, et disponible sur Windows et MacOS. La première étape est donc d'aller sur la page *Downloads* du site de VirtualBox, puis de télécharger et installer la dernière version du logiciel. Ensuite, sur la même page, il faut télécharger le *VirtualBox Extension Pack*, qui permet d'étendre les capacités des VMs, et sera nécessaire pour celle que nous utiliserons. Une fois téléchargé, il suffit de double-cliquer sur le fichier, pour que VirtualBox l'installe automatiquement. Plus d'informations sur l'installation de VirtualBox sont disponibles à l'adresse suivante : https://wiki.student.info.ucl.ac.be/Logiciels/VirtualBox.

La prochaine étape est de télécharger la machine virtuelle en elle-même. Nous fournissons l'image des machines Fedora disponibles en salles informatiques, qui peut être installée sur VirtualBox. Cette image est la même que celle accessible via UDS, comme décrit plus tôt. Elle peut être téléchargée avec le lien suivant : https://wiki.student.info.ucl.ac.be/uploads/Mat%C3%A9riel/Mat%C3%A9riel/fedora32.ova (**Attention**, le fichier fait plus de 5 GB !). Une fois téléchargée, il suffit de double-cliquer sur le fichier, et VirtualBox ouvrira le fichier pour l'installation. Sur la fenêtre s'étant ouverte, les paramètres de base peuvent être laissés. Cliquez sur *Importer* pour créer la machine virtuelle. La procédure d'importation prend un certain temps, patience !

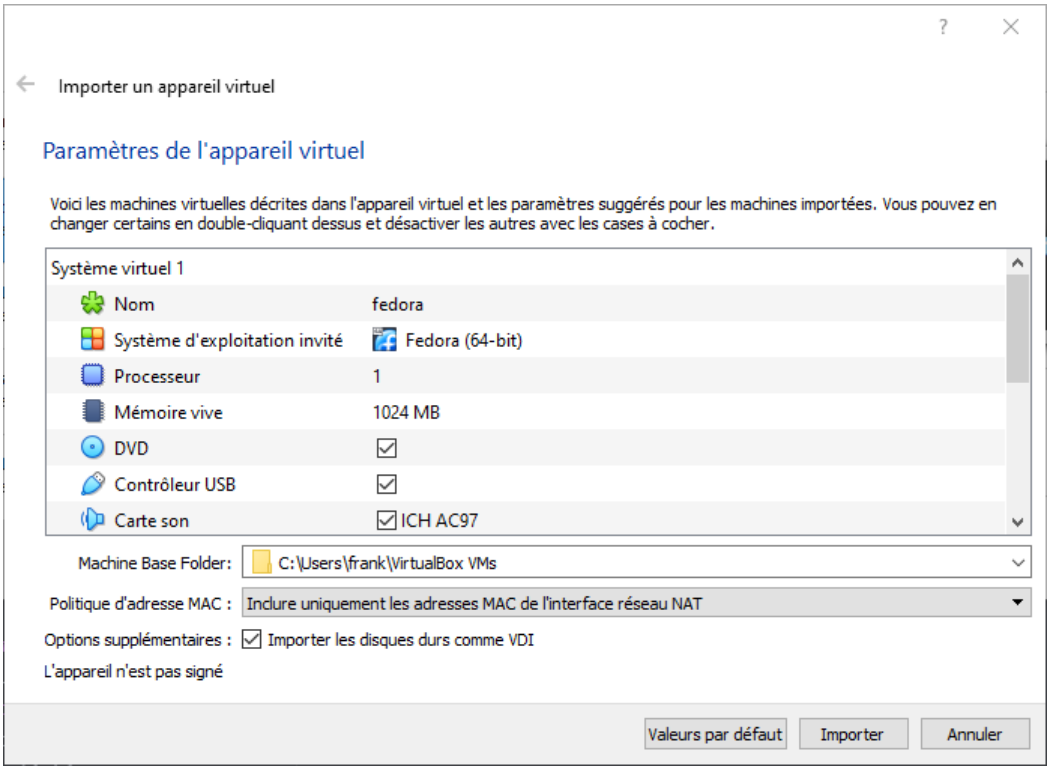


Fig. 7 – Importation d'une VM dans VirtualBox

Une fois que la VM est importée, démarrez la. Sur l'écran noir vous demandant de choisir entre 2 versions de Fedora, choisissez la première. Ensuite, sélectionnez l'utilisateur « Tux », qui est l'utilisateur par défaut. Vous avez désormais une machine virtuelle Linux fonctionnelle, à l'intérieur de votre machine !

5.4 Dual-boot Linux

La dernière option, qui est la plus radicale, est d'installer directement Linux en tant que système d'exploitation sur sa machine, de manière à pouvoir choisir au démarrage entre le système d'exploitation natif et Linux. Ceci s'appelle un *dual-boot*, car on a le choix entre deux systèmes d'exploitation (*dual*) lors du démarrage (*boot*). Cette possibilité est la plus compliquée en terme d'installation, mais permet le plus de confort d'utilisation et de performance une fois l'installation effectuée. Elle est conseillée pour les étudiants en filière informatique (car Linux pourra être utilisé par la suite dans d'autres cours), mais peut sembler trop lourde pour les étudiants qui n'utiliseront Linux que pour le cours LEPL1503.

Il existe de nombreuses manières d'installer un dual-boot sur sa machine. Le kot à projet Louvain-li-Nux propose généralement une « Install Party », où ils peuvent vous aider à installer Linux en dual-boot. De nombreux tutoriels

sont également disponibles en ligne. L'installation dépend évidemment de la version de Linux que vous souhaitez. Un tutoriel pour l'installation d'Ubuntu, qui est une des versions les plus populaires de Linux, est disponible à l'URL https://lecrabeinfo.net/installer-ubuntu-20-04-lts-le-guide-complet.html. Pour Fedora, la version de Linux disponible sur les machines des salles informatiques (et sur UDS, comme expliqué précédemment), un tutoriel est disponible à l'URL https://www.tecmint.com/install-fedora-with-windows-dual-boot/.

CHAPITRE 6

Shell

L'interpréteur de commande, ou shell, est l'interface de communication entre l'utilisateur et le système d'exploitation. C'est un exécutable chargé d'interpréter les commandes écrites par l'utilisateur et de les exécuter.

Dans le cadre de ce cours nous utiliserons l'interpréteur bash(1). Cet exécutable est généralement placé dans le fichier `/bin/bash`.

Le shell est un outil très puissant. Il permet d'effectuer de nombreuses opérations qui peuvent difficilement être réalisées manuellement ou via une interface graphique.

Note : Astuces pour utiliser le shell rapidement

Lors de la frappe d'un nom de fichier, d'un chemin d'accès ou même d'une commande tapez sur la touche `<tab>` pour « compléter » le mot que vous écrivez. Cela vous évite de devoir écrire les noms des commandes ou des fichiers en entier à chaque fois. Si rien ne se passe tapez deux fois <tab> pour obtenir la liste des possibilités.

De plus, vous pouvez accéder à l'historique des commandes en utilisant la flèche du haut.

Note : Chemin absolu et relatif

Pour écrire la position d'un fichier (son chemin), il y a deux manières de faire :

- Le chemin absolu : il fait référence au chemin qu'il faut parcourir dans le système de fichier en démarrant de la racine, représentée par le caractère `/`.
- Le chemin relatif : il fait référence au chemin à parcourir depuis le dossier courant.

6.1 Expressions régulières

Avant de commencer à voir les commandes utiles avec le shell, il est important de définir ce qu'est une expression régulière (*regex(3)*). Les expressions régulières caractérisent des chaînes de caractères et elles sont utiles pour de nombreuses commandes. Nous l'utiliserons notamment pour faire une recherche dans un fichier.

Dans une regex, certains caractères ont une signification particulière :

Expression	Explication	Exemple
`\`	Caractère d'échappement	`[\.]` contient un « . »
`^`	Début de ligne	`^b` commence par b
`.`	N'importe quel caractère	`^.$` contient un seul caractère
`$`	Fin de ligne	`er$` finit par « er »
`|`	Alternative	`^(a|A)` commence par a ou A
`( )`	Groupement	`^((a)|(er))` commence par a ou er
`-`	Intervalle de caractères	`^[a-d]` commence par a,b,c ou d
`[ ]`	Ensemble de caractères	`[0-9]` contient un chiffre
`[^]`	Tout sauf un ensemble de caractères	`^[^a]` ne commence pas par a
`+`	1 fois ou plus	`^(a)+` commence par un ou plusieurs a
`?`	0 ou 1 fois	`^(a)?` commence ou non par un a
`*`	0 fois ou plus	`^(a)*` peut ou non commencer par a
`{x}`	x fois exactement	`a{2}` deux fois « a »
`{x,}`	x fois au moins	`a{2,}` deux fois « a » au moins
`{x, y}`	x fois minimum, y maximum	`a{2,4}` deux, trois ou quatre fois « a »

Notes :

— `^b$` = contient uniquement le caractère `b`

— `^$` = la ligne est vide

Nous verrons plus en détail leur utilisation avec les commandes plus complexes.

6.2 Manipulation des répertoires

Chaque processus travaille dans un répertoire dit courant. C'est le répertoire dans lequel le processus accède pour lire ou écrire des fichiers lorsque le processus utilise un nom relatif. La commande pwd(1) affiche le chemin du répertoire courant.

Il est possible de changer le répertoire courant du processus ou du shell en utilisant la commande cd(1posix). Exemples :

— cd(1posix) *chemin* : change le répertoire courant par celui de « chemin ».

— cd(1posix) : change le répertoire courant par le répertoire de login de l'utilisateur courant.

— cd(1posix) .. : remonte dans le répertoire prédécesseur dans l'arborescence des fichiers.

La commande mkdir(1) permet de créer un répertoire. Elle prend comme argument le nom du répertoire à créer. La commande rmdir(1) supprime un répertoire qui doit être vide. Pour effacer un répertoire et tous les fichiers qu'il contient, il faut utiliser la commande rm(1) avec l'option `-r`. Ainsi, `rm -r /tmp/t` supprime le répertoire `/tmp/t` ainsi que tous les fichiers et sous-répertoires se trouvant dans ce répertoire.

La commande ls(1) permet de connaître l'ensemble des fichiers et répertoires contenus dans le répertoire courant. Elle supporte plusieurs options dont les plus utiles sont :

— `-F` : Positionne à la fin des noms de fichier un `/` pour les répertoires et `*` pour les fichiers exécutables

— `-a` : Affiche tous les fichiers, y compris les fichiers cachés (ceux qui commencent par le caractère `.`)

— `-d` : Ne liste pas le contenu d'un répertoire : si *rep* est un répertoire, `ls -l` *rep* listera le contenu du répertoire *rep*, alors que `ls -ld` *rep* listera la description du répertoire

— `-l` : Description complète du contenu d'un répertoire (une ligne par fichier)

Avec l'option `-l`, le premier caractère de la ligne indique le type du fichier. Le caractère `-` correspond à un fichier standard et `d` à un répertoire. Il est aussi possible de connaître le contenu d'un autre répertoire que le répertoire courant en fournissant le nom de ce répertoire comme argument à la commande `ls`.

```
/repertoiretest $ ls
file.txt  repertoiresoustest/

/repertoiretest $ l repertoiresoustest/
first.txt  log.log  second.txt
```

6.3 Manipulation de fichiers

6.3.1 Créer et détruire

- > filename crée un fichier vide.
- touch(1) filename crée un fichier vide.
- echo(1) mon_texte > filename crée un fichier avec « mon_texte » dedans.

rm(1) [-irf] files efface les fichiers
- -i : interactif, demande une confirmation sur chaque fichier
- -f : force la suppression du fichier
- -r : efface un répertoire et son contenu

6.3.2 Visualiser

nl(1) [-opt] file affiche le contenu d'un fichier et en numérote les lignes.
- **-bt** : numérote les lignes non vides (par défaut)
- **-ba** : numérote toutes les lignes
- **-bpXXX** : numérote seulement les lignes qui contiennent la chaîne de caractères XXX
- -sX : supprime le décalage dû à la numérotation et utilise le séparateur X
- -s'XXX" : supprime le décalage dû à la numérotation et utilise la chaîne "XXX"

paste(1) [-opt] f1 f2 concatène horizontalement et affiche les deux fichiers.
- -s : copie les lignes d'un fichier sur une ligne

more(1) file visualise le contenu du ou des fichiers par page.

Si il contient plus d'une page :
- q ou Q : pour terminer la visualisation
- RETURN : pour visualiser une ligne supplémentaire
- ESPACE : pour visualiser la page suivante
- h : pour obtenir de l'aide

6.3.3 Modifier

touch(1) filename met à jour les dates d'accès et de modification du fichier. Crée le fichier si il n'existe pas.

- -c : empêche la création du fichier si celui ci n'existe pas
- -m : change uniquement la date de modification du fichier
- -a : change uniquement la date d'accès du fichier

split(1) [-opt] file [out] coupe le fichier en plusieurs petites parties
- -b nbr : découpe selon un nombre d'octets
- -n nbr : découpe selon un nombre de lignes

6.3.4 Extraction de données

grep(1) [-opt] regex file recherche l'expression dans les fichiers.
- -i : ignore la casse
- -v : affiche les lignes ne contenant pas l'expression.
- -c : compte les lignes ne contenant pas la chaîne
- -n : numérote chaque ligne contenant la chaîne
- -x : affiche les lignes correspondant exactement à la chaîne

uniq(1) [-opt] filename affiche le fichier en supprimant les lignes qui se répètent successivement.
- -u : Affiche seulement les lignes n'apparaissant qu'une seule fois
- -d : Affiche seulement les lignes répétées
- **-c** [En plus de l'affichage standard, chaque ligne est précédée du nombre de répétitions] Si cette option est utilisée, alors les options -u et -d sont ignorées.
- -i : ignore la casse
- -s N : ne compare pas les N premiers caractères de chaque ligne
- -w N : ne compare pas plus de N caractères de chaque ligne

sort(1) [-opt] filename trie les lignes par ordre alphabétique.

— -f : ignore la casse
— -r : inverse l'ordre de tri
— -o : modifie la sortie standard
— -t : modifie le caractère séparateur. Par défaut c'est une chaîne de blancs
— -n : compare selon la valeur arithmétique
— -k : spécifie la colonne utilisée pour le tri

uniq(1) et sort(1) sont souvent utilisés ensemble. Par exemple, cette commande trie les lignes de *file.txt* selon leur nombre d'apparitions.

```
$ cat file.txt
une fois
deux fois
deux fois
trois fois
encore une fois
trois fois
toujours une fois
trois fois

$ sort file.txt | uniq -c | sort -n
        1 encore une fois
        1 toujours une fois
        1 une fois
        2 deux fois
        3 trois fois
```

Une autre utilisation possible est de pouvoir trier un fichier, par exemple CSV, sur une colonne particulière. Tout d'abord, il faut modifier le séparateur de colonne avec -t, puis spécifier la colonne

```
$ cat file.txt
pcr,01,3
pcr,1,3
pcr,04,5
pcr,03,6
alex,03,6
zorro,01,20
zorro,5,4

$ cat file.txt | sort -t, -k2n
zorro,01,20
pcr,01,3
pcr,1,3
alex,03,6
pcr,03,6
pcr,04,5
zorro,5,4
```

diff(1) [-opt] f1 f2 compare le contenu de deux fichiers.

— -i : ignore la casse
— -c : rapport plus clair
— -q : indique uniquement si les fichiers sont différents
— -b : ignore les différences dues à des espaces blancs
— -B : ignore les différences dues à des lignes blanches

```
$ cat test.txt
premiere ligne similaire

deuxieme differente
et moi pareil
troisieme comme la deuxieme
```

(suite sur la page suivante)

(suite de la page précédente)

```
et enfin la quatrieme est la meme!
$ cat testbis.txt
premiere ligne similaire
en effet, je ne lui ressemble pas..
et moi pareil
moi non plus, tres cher.

et enfin la quatrieme est la meme!

$ diff test.txt testbis.txt
2,3c2                                   =  Les lignes 2,3 du premier fichier␣
↪et 2 du second sont différentes
<                                       _
< deuxieme differente                    \
---                                        >  Affichage des lignes différentes
> en effet, je ne lui ressemble pas..    _/
5c4                                     _
< troisieme comme la deuxieme             \
---                                        > Même réflexion
> moi non plus, tres cher.               _/
```

6.3.5 Obtenir des informations

wc(1) [-opt] filename donne sur stdout des informations au sujet de l'entrée standard ou d'une liste de fichiers.

Première colonne est le nombre de lignes, deuxième le nombre de mots et en dernier le nombre d'octets.

- -l : nombre de lignes
- -c : nombre d'octets
- -m : nombre de caractères
- -L : la longueur de la plus longue ligne
- -w : le nombre de mots

6.4 Manipulations communes aux répertoires et fichiers

6.4.1 Copier

cp(1) [-opt] src dst copie la src dans le fichier dst.

Si dst n'existe pas, il est créé. Sinon, si c'est un fichier, son contenu est écrasé.

- -r : spécifie la copie d'un répertoire
- -u : copie uniquement si src est plus récent que dst ou si il est manquant dans dst

Note : Si la destination est un répertoire, alors la source peut être une liste de fichiers.

```
$ cp test.txt ./testbis/
$ cp test.txt btest.txt ../
$ cp -r repertoire ../repertoirebis
```

6.4.2 Déplacer ou renommer

mv(1) [-opt] src dst renomme ou déplace src en dst.

- -f : écrase les fichiers existants
- -i : demande confirmation avant d'écraser un fichier existant
- -n : n'écrase aucun fichier déjà existant

Note : Si la destination est un répertoire, alors la source peut être une liste de fichiers.

```
$ mv test.txt testrename.txt
$ mv test.txt ./testbis/
$ mv repertoire ./repertoirebis
```

6.4.3 Rechercher

Pour les critères de recherche :

- critère1 critère2 = et logique
- !critère = non logique
- critère1 -a critère2 = ou logique

find(1) chemin regex recherche les fichiers/répertoires caractérisés par nom, à partir du répertoire *rep* et affiche le résult

- **-name** : sur le nom du fichier
- **-perm** : sur les droits d'accès du fichier
- -links : sur le nombre de liens du fichier
- **-user** : sur le propriétaire du fichier
- -group : sur le groupe auquel appartient le fichier
- **-type** : sur le type (d=répertoire, c=caractère, f=fichier normal)
- **-size** : sur la taille du fichier en nombre de blocs (1 bloc=512octets)
- -atime : par date de dernier accès en lecture du fichier
- -mtime : par date de dernière modification du fichier
- -ctime : par date de création du fichier
- -print : affiche les fichiers sur stdout

```
$ find ./ -name "*fi*" -print          = contenant fi
$ find ./ -mtime "3" -print            = modifié dans les trois derniers␣
↪jours
$ find ./ -name "*s*" -a -name "f*"    = contenant s et commençant par f
```

Note : « ./ » représente le répertoire courant

Il y a trois remarques à faire sur la commande find :

- Il est parfois nécessaire de mettre -print dans la commande pour afficher le résultat
- Lors de larges recherches, il peut y avoir un message d'erreur pour chaque tentative d'accès à un fichier où vous n'avez pas d'autorisation d'accès, par exemple des fichiers système. Pour éviter que ces messages d'erreur ne polluent la recherche, il faut rediriger la sortie d'erreur standard dans « un puits sans fond ». Pour cela, rajoutez 2>/dev/null
- Il est parfois très utile de pouvoir exécuter une commande sur les fichiers trouvés. La solution la plus légère est de rediriger la sortie et de lui attribuer une commande. Pour cela, il faut faire : « find rep -name expr| xargs commande ». Cette commande est expliquée dans la section « Commandes plus complexes ».

Pour cet exemple, le résultat est tous les fichiers dont le nom contient « mon test », et donc le fichier contient « supertab ». .. code-block : : console

$ find /testdirectory -name *mon test* -type f | xargs grep supertab

6.4.4 Création de lien

ln(1) [-opt] src dst création d'un lien (raccourci) sur un fichier ou un répertoire. Attention un lien n'est pas une copie.

Il existe deux sortes de liens :

- le lien physique : uniquement des fichiers
- le lien symbolique (avec l'option -s) : fichiers et répertoires

« SHEMA »

Dans le cas de lien physique, on supprime le fichier en supprimant tous les liens qui pointent sur ce fichier. Par contre pour des liens symboliques, vous pouvez effacer le fichier sans effacer les liens, mais alors ceux-ci seront invalides.

6.4.5 Archivage et compression

Il est important de noter qu'une archive n'est pas forcément compressée.

tar(1) [-opt] tarname.tar files crée une archive à partir d'une liste de fichiers ou de répertoires.

- f : argument obligatoire, sauf si l'on veut lire ou écrire vers/depuis un lecteur de bande

— c : crée une archive
— z : compresse l'archive créée, en utilisant gzip. (Attention, l'extension doit être « tar.gz »)
— j : compresse mieux l'archive mais prend plus de temps. (Attention, l'extension doit être « tar.bz2 »)
— x : désarchive
— t : inspection de l'archive

```
$ tar cf monarchive.tar firstfile.c  secondfile.c        =      crée une␣
↪archive contenant deux fichiers
$ tar cfz monarchive.tar.gz firstfile.c  secondfile.c    =      crée une␣
↪archive compressée
$ tar tf monarchive.tar                                  =      inspecte l
↪'archive créée
firstfile.c
secondfile.c
$ tar xf monarchive.tar.gz                               =      désarchive
$ tar xf monarchive.tar -C /home                         =      désarchive␣
↪monarchive.tar dans /home
```

gzip(1) file compresse un fichier ou une archive
— **-c** : la compression est effectuée sur la sortie standard au lieu du fichier lui-même
— -c1 : compression plus rapide
— -c9 : meilleur compression

```
$ gzip secondfile.c              =      compresse un fichier et produit un␣
↪fichier .gz
$ gzip monarchive.tar            =      compresse une archive

$ ls
monarchive.tar                   =      compresse monarchive.tar vers␣
↪monarchive.tar.gz
$ gzip monarchive.tar
ls
monarchive.tar  monarchive.tar.gz
```

6.4.6 Permissions

Pour chaque fichier, il y a trois classes d'utilisateurs
— user : le propriétaire du fichier
— groupe : le groupe auquel appartient le fichier
— autre : tous les autres

Les permissions accordées à ces trois classes sont :
— r : lecture
— w : écriture
— x : exécution (Un fichier peut être exécuté et un répertoire peut devenir répertoire courant)

chmod(1) mode files change les permissions du ou des fichiers/répertoires.

```
                              user    group   other
mode désiré : rwxr-xr--       rwx      r-x     r--
                              111      101     100     (en binaire)
                               7        5       4      (en hexadecimal)

d'où la commande ``chmod 754 fichier``
```

chown(1) owner files change le propriétaire du fichier.

chgrp(1) grp files change le groupe du fichier.

6.4.7 Obtenir des informations

stat(1) [-opt] filename donne des informations sur les métadonnées associées au fichier

— -f : affiche l'état du système de fichiers plutôt que celui du fichier
— -L : suit les liens du fichier
— -t : affiche les informations de façon concise
— –format=FORMAT : affiche les informations selon le format choisi

```
Séquences de format valables pour les fichiers :
        %a droits d'accès en octal
        %A droits d'accès dans un format lisible par un humain
        %b nombre de blocs alloués (voir << %B >>)
        %B taille, en octets, de chaque bloc rapporté par %b
        %d numéro de péripherique en décimal
        %D numéro de péripherique en hexadécimal
        %f mode brut en hexadécimal
        %F type de fichier
        %g identifiant de groupe du propriétaire
        %G nom de groupe du propriétaire
        %h nombre de liens directs (<< hard >>)
        %i numéro d'inode
        %m point de montage
        %n nom de fichier
        %N nom du fichier cité, déréférencé s'il s'agit d'un lien symbolique
        %o taille de bloc d'entrée/sortie
        %s taille totale, en octets
        %u identifiant du propriétaire
        %U nom d'utilisateur du propriétaire
        %w date de création au format lisible, ou << - >> si elle n'est pas␣
↪connue
        %x date du dernier accés au format lisible
        %y date de la dernière modification au format lisible
        %z date du dernier changement au format lisible

 Séquences de format valables pour les systèmes de fichiers :
        %a nombre de blocs libres disponibles pour les utilisateurs normaux
        %b nombre total de blocs de données dans le système de fichiers
        %c nombre total d'inodes dans le système de fichiers
        %d nombre d'inodes libres dans le système de fichiers
        %f nombre de blocs libres dans le système de fichiers
        %i identifiant du système de fichier en hexadécimal
        %l longueur maximale des noms de fichier
        %n nom de fichier
        %s taille des blocs (pour des transferts plus rapides)
        %S taille fondamentale des blocs (pour le décompte des blocs)
        %t type en hexadecimal
        %T type dans un format lisible par un humain
```

6.5 Gestion des processus

top(1) affiche les processus en cours d'exécution. pstree(1) affiche l'arbre des processus.

strace(1) [-opt] cmd trace les appels systèmes et la création de signaux effectués par une commande
— -c : collecte quelques statistiques de base concernant les appels système tracés
— -o : redirige la sortie standard
— -p : avec cette option, cmd est remplacé par le PID d'un processus, et celui ci est tracé
— -T : indique le temps passé dans chaque appel système
— -t : indique l'heure au début de chaque ligne. -tt comprend les microsecondes
— -r : donne le temps entre deux appels systèmes successifs

```
$ strace -c ./monexecutable -o fichierRecoltantLesInformations.log
```

lsof(8) [-opt] affiche les fichiers ouverts.
— -p PID : uniquement les fichiers ouverts du processus
— -i : affiche les connexions réseau ouvertes

```
$ lsof -i -p 2735          =  Les connexions ouvertes ET les fichiers ouverts␣
↪par le processus 2735
$ lsof -i -a -p 2735       =  Les connexions ouvertes par le processus 2735
```

kill(1) pid supprime le processus spécifié. Si malgré la commande, le processus n'est pas détruit, essayez kill -9 pid.

6.6 Symboles utiles

6.6.1 Redirection de l'entrée, sortie et erreur standard

Lors de l'exécution d'une commande, un processus est créé et celui-ci va ouvrir trois flux : l'entrée, la sortie et l'erreur standard. Par défaut lorsque l'on exécute un programme, les données sont donc lues à partir du clavier et le programme envoie sa sortie et ses erreurs sur l'écran. toutefois, il est possible de rediriger ces flux.

- < l'entrée standard est lue à partir d'un fichier
- > La sortie standard est redirigée dans un fichier. Si le fichier existe, il est vidé avant d'écrire.
- >> La sortie standard est redirigée dans un fichier. Si le fichier existe, la sortie standard est ajoutée à la fin de celui ci.
- 2> La sortie d'erreur standard est redirigée
- cmd1 | cmd2 La sortie standard de cmd1 devient l'entrée standard de cmd2

6.6.2 Symboles pour les commandes

- `?` caractère joker remplaçant un seul caractère
- `!` inverse le sens d'un test ou l'état de sortie d'une commande.
- `*` caractère joker remplaçant une chaîne de caractères
- `&` exécute une commande en arrière-plan
- `;` sépare des instructions sur une seule ligne
- `cmd1 && cmd 2` cmd2 n'est exécuté que si cmd1 réussit
- `cmd1 || cmd 2` cmd2 n'est exécuté que si cmd1 échoue
- \ annule l'effet du caractère spécial suivant
- " " annule l'effet de tous les caractères spéciaux entre les guillemets, sauf `$` et \

6.7 Commandes utiles

6.7.1 Pour effectuer des chaînes

xargs(1) permet d'appliquer une commande à l'entrée standard.

Pour cet exemple, le résultat est tous les fichiers dont le nom contient « mon test », et dont le fichier contient « supertab ».

```
$ find /testdirectory -name *mon test* -type f | xargs grep supertab
```

tee(1) file lit depuis l'entrée standard, écrit dans la sortie standard et dans le fichier. Elle est utilisée pour continuer une chaîne tout en faisant une sauvegarde des informations.

```
% echo "Les tubes sont un mécanisme puissant." | tee fichier.txt | wc
         1       6      39
% cat fichier.txt
Les tubes sont un mécanisme puissant.
```

On peut voir que le texte a bien été relayé vers la commande « wc » et qu'en même temps, ce texte a été écrit dans fichier.txt

6.7.2 Informations générales

su(1) passe en mode « root », c'est à dire administrateur

whatis(1) cmd explique brièvement l'utilité d'une commande

apropos(1) [-opt] mot-clé recherche dans les man pages les commandes correspondants aux mots clés.
- **-a** [Affiche seulement les résultats répondant à tout les mots clés.] L'inverse est le fonctionnement par défaut

date(1) donne l'heure, selon l'horloge de votre ordinateur

cal(1) affiche un calendrier du mois courant

halt(8) éteint l'ordinateur.

reboot(8) redémarre l'ordinateur

6.7.3 Informations système

time(1posix) programme permet de calculer le temps d'exécution d'un programme

df(1) [-opt] [file] indique l'espace disque utilisé et disponible sur tous les systèmes de fichiers. Si des fichiers sont passés en argument, seul les systèmes de fichiers contenant un des fichiers sont montrés.

—	**-h**	Imprime les dimensions dans un format lisible par l'utilisateur
—	**-H**	Idem que -h, mais il utilise des puissances de 1000 au lieu de 1024
—	**-i**	Affiche l'information i-node au lieu de l'utilisation des blocs
—	**-l**	Limite l'affichage aux systèmes de fichiers locaux
—	**-P**	Utilise le format de résultat POSIX
—	**-T**	Imprime le type de système de fichiers

6.7.4 Maniement des jobs

La plupart des commandes en console sont exécutées rapidement, mais ce n'est pas le cas de toutes. Certaines commandes, que l'on va appeler *jobs*, prennent plus de temps (comme par exemple copier un très gros fichier), et d'autres encore tournent indéfiniment.

Évidemment, quand un job est en cours d'exécution à la console, plus aucune action ne peut être faite sur celle-ci. Unix nous vient en aide dans ce cas-là avec le raccourci `Ctrl+z` et les commandes jobs(1), bg(1) et fg(1).
- `Ctrl+z` : Le job passe dans l'état `suspended`. Il est en pause, et placé en background.
- `jobs` : Affiche à la console la liste des jobs présents en background
- `bg` : Passe un job mis en background de l'état `suspended` à l'état `running`. Le job reste en background, mais il continue à s'exécuter
- `fg` : Passe un job du background à l'avant-plan

Exemples :

```
$ yes > \dev\null
#nous lançons la commande yes

^Z
#nous la suspendons avec Ctrl+z
[1]+  Stopped                 yes > \dev\null
#elle est placée en arrière-plan

$ jobs
#nous regardons la liste des jobs en arrière plan
[1]+  Stopped                 yes > \dev\null
#chaque job à un numéro qui lui est attribué. ici 1

$ bg 1
#nous relançons yes en arrière-plan. On peut utiliser son nom comme son␣
↪numéro avec la commande bg et fg
[1]+ yes > \dev\null &
#yes s'est remis en route

$ jobs
#nous vérifions le statut de yes avec jobs
[1]+  Running                 yes > \dev\null &
#il est en cours d'exécution
```

(suite sur la page suivante)

(suite de la page précédente)

```
$ fg yes
#nous remettons yes en avant-plan
yes > \dev\null

^Z
#nous le suspendons à nouveau
[1]+  Stopped                 yes > \dev\null

$ kill %1
#nous terminons yes avec la commande kill %[numJob]
[1]+  Stopped                 yes > \dev\null

$ jobs
#nous vérifions les jobs
[1]+  Terminated: 15          yes > \dev\null
#yes est marqué Terminated

$ jobs
#un deuxième appel à jobs nous affiche une liste vide
```

6.8 Commandes complexes

6.8.1 Modification d'un fichier

sed(1) [-n] [-e "prog"] [-f cmdfile] [file] applique des commandes de "prog" sur un fichier

- -n : n'affiche aucune ligne, sauf celle spécifiée avec la commande p
- **-e** [specifie les commandes à appliquer sur le fichier] Note : Il vaut mieux encadrer la commande avec des " ou des «
- -f : les commandes sont lues à partir d'un fichier

Pour bien comprendre la puissance de sed, il est important de comprendre son fonctionnement. sed fonctionne en 4 étapes :

- Lecture d'une ligne sur le flux d'entrée, et stockage dans l'espace de travail
- Exécute les commandes sur l'espace de travail
- Envoie la ligne au flux de sortie en lui rajoutant un "n"
- Recommence avec la ligne suivante ...

Une commande d'un "prog" est constituée d'un adressage, c-à-d les lignes sur lesquelles la commande est appliquée, et de l'action à exécuter.

1. L'adressage est décomposé en deux catégories.
   - : toutes les lignes
   - num : la ligne « num ». La dernière ligne est symbolisée par $
   - num1, num2 : les lignes entre num1 et num2
   - /regex/ : les lignes correspondant à l'expression régulière regex
   - **/regex1/, /regex2/** [les lignes entre la première ligne correspondant à regex1 et la première ligne correspondant à regex2] Si regex2 est vide, la commande sera appliquée jusqu'à la fin du fichier.

   Note : Le ! représente la négation. Mettez le après votre spécification des lignes pour prendre la négation

   RAPPEL sur les regex :

Expression	Explication	Exemple
\	Caractère d'échappement	[\.] contient un « . »
^	Début de ligne	^b commence par b
.	N'importe quel caractère	^.$ contient un seul caractère
$	Fin de ligne	er$ finit par « er »
|	Alternative	^(a|A) commence par a ou A
( )	Groupement	^((a)|(er)) commence par a ou er
-	Intervalle de caractères	^[a-d] commence par a,b,c ou d
[ ]	Ensemble de caractères	[0-9] contient un chiffre
[^]	Tout sauf un ensemble de caractères	^[^a] ne commence pas par a
+	1 fois ou plus	^(a)+ commence par un ou plusieurs a
?	0 ou 1 fois	^(a)? commence ou non par un a
*	0 fois ou plus	^(a)* peut ou non commencer par a
{x}	x fois exactement	a{2} deux fois « a »
{x,}	x fois au moins	a{2,} deux fois « a » au moins
{x, y}	x fois minimum, y maximum	a{2,4} deux, trois ou quatre fois « a »

Notes :
— ^b$ = contient uniquement le caractère b
— ^$ = la ligne est vide

2. Les actions
   — p : affiche les lignes
   — d : supprime les lignes
   — y/l1/l2 : remplace les caractères de la première liste par les caractères de la seconde
   — **s/mtf/sbst/** [substitue le mtf par le sbst]

     Note [Par défaut seule la première occurrence est remplacée.]
     — Pour toutes les remplacer : /s/motif/substitut/g
     — Pour en remplacer 4 : /s/motif/substitut/4
   — N : charge une ligne supplémentaire dans l'espace de travail
   — D : efface l'espace de travail jusqu'au premier saut de ligne incorporé
   — b : revient

   Pour faire des commandes groupées, placez vos commandes entre { } séparées par « ; ».
   Quelques illustrations basiques :

```
$ sed '' test.txt                          = Le script est vide, il␣
↪renvoie simplement le fichier

$ sed -n '/Ici/p' test.txt                 = Affiche les lignes contenant␣
↪Ici
$ sed 'p' test.txt                         = Double toutes les lignes

$ sed -e '4d; 7d' test.txt                 = Supprime les lignes 4 et 7
$ sed -e '4,7d' test.txt                   = Supprime les lignes entre 4␣
↪et 7

$ sed '/^#/ d' test.txt                    = Supprime les lignes␣
↪commencant par #
$ sed '/e$/ d' test.txt                    = Supprime les lignes se␣
↪terminant par e
$ sed '/#/,/@/d' test.txt                  = Supprime les lignes comprises␣
↪entre le premier # et le premier @
```

(suite sur la page suivante)

(suite de la page précédente)

```
$ sed -e 's/^#//' test.txt              = Supprime le commentaire en␣
↪début de ligne, puisqu'il
                                          est remplacé par ''

$ sed -e 'y/éèê/eee/' test.txt          = Retire les accents, puisqu
↪'ils sont remplacés par 'e'

$ sed -e ' 4,7 {y/éèê/eee/;s/e/[]/} test.txt    = Remplace les accents,␣
↪puis remplace les "e" par "[]"

$ sed -e '/^$/ {N; D}' test.txt         = Supprime les sauts de ligne
```

Explication : Pour les lignes vides, on charge la ligne suivante, on envoie ce qui se trouve dans l'espace de travail jusqu'au premier "n", puis on continue le traitement du texte. Pour continuer le traitement, une nouvelle ligne est chargée et va donc « écraser » les "n" qui sont toujours présents dans l'espace de travail.

Lors du remplacement d'un mot par un autre, il peut survenir un problème de taille. En effet, le remplacement n'est effectué que sur le premier mot de la ligne trouvé.

```
$ sed -e ' s/[oe]/[/' test.txt
B[njour,

C[ci est un fichier de test.
Ici la lign[ numéro 4.

# c[ci pourrait être un commentaire
Ici la lign[ numéro 7.I

Au r[voir
```

On remarque que tout les "e" et "o" n'ont pas été remplacés...

Pour contrecarrer ce problème, il est possible de placer dans le script un label et de revenir dessus, comme un goto en C. Pour effectuer ce retour utilisez la commande "b".

```
$ sed -re ':start {s/[eo]/[/g; /[eo]/ b start}' test.txt
B[nj[ur,

C[ci [st un fichi[r d[ t[st.
Ici la lign[ numér[ 4.

# c[ci p[urrait êtr[ un c[mm[ntair[
Ici la lign[ numér[ 7.I

Au r[v[ir
```

Explication : Un label est placé au début des commandes. La première commande remplace le premier [eo] trouvé. La seconde retourne au label si il reste encore un [eo] dans la ligne. Une fois qu'il n'y a plus de [eo], la ligne suivante est chargée.

6.8.2 Appliquer des actions à un fichier

awk(1) [-Fs] [-v variable] [-f fichier de commandes] "program" fichier

- -F : Spécifie les séparateurs de champ
- -v : Définit une variable utilisée à l'intérieur du programme.
- -f : Les commandes sont lues à partir d'un fichier.

Note : awk est une commande extrêmement puissante, elle permet d'effectuer une multitude d'opérations. Son utilisation est complexe et elle est bien détaillée sur ce site : https://www.shellunix.com/awk.html. Je vous encourage à le lire.

6.8.3 Redirection nommée

mkfifo(1) nom crée un tube nommé

```
ls | less est donc similaire à  mkfifo /tmp/tempfifo
                                ls > /tmp/tempfifo
                                less < /tmp/tempfifo
```

6.9 Bash

Taper des commandes dans la console est inévitable lors d'opérations avancées sur un système Unix, et peut devenir très vite répétitif et fastidieux pour l'utilisateur. Le Bash est justement là pour éviter ces répétitions et automatiser certaines tâches à l'aide de scripts, qui sont des fichiers texte composés de différentes commandes Unix, lus, interprétés et exécutés par Bash.

6.9.1 Premier script

Nous allons écrire un premier script bash pour présenter la manière générale de procéder avec un tel outil. Les scripts commencent toujours par la ligne `#!/bin/bash` qui indique à l'exécution qu'il s'agit d'un script et avec quel interpréteur le lire (ici bash).

```
#!/bin/bash
echo "Hello, 1252"
```

Nous allons enregistrer ce texte sous le nom hello.sh, puis changer ses permissions pour le rendre exécutable.

```
$ chmod 700 hello.sh
```

Après il ne reste plus qu'à l'exécuter et observer le résultat.

```
$ ./hello.sh
Hello, 1252
```

6.9.2 Les variables

Bash permet l'utilisation de variables dans les scripts. Il peut s'agir de simples variables ou de tableaux. Bash n'est pas un langage typé, les entiers ou les String n'existent pas, toutes les variables sont traitées de la même façon. Pour illustrer ceci nous allons écrire le script variables.sh

```
#!/bin/bash

bonjour='Hello, '
#il est important de ne pas mettre d'espaces autour du =
nombre[0]=12
nombre[1]=52

echo $bonjour${nombre[0]}${nombre[1]}
#on accède à une variable simple avec un $ devant son nom
#on accède à un élément d'un tableau avec un $ devant et des {} autour
echo $bonjour${nombre[*]}
#le caractère * indique qu'on veut utiliser tous les éléments du tableau␣
↪(séparés
#par un espace à chaque fois)
```

Ce script produit comme résultat

```
$ ./variables.sh
Hello,1252
Hello,12 52
```

6.9.3 Les structures de contrôle

Comme dans chaque langage de programmation, bash offre les structures de contrôle habituelles telles que les boucles if, for ou encore while que nous allons démontrer maintenant.

Comme dit précédemment, il n'y a pas de type en bash, true et false n'existent pas. Les conditions que les boucles vont utiliser seront les valeurs renvoyées par l'exécution d'une commande. Un 0 renvoyé correspond à un true, tandis que tout le reste est considéré comme un false.

Dans le but de tester ces boucles nous utiliserons un petit programme en C, return.c, qui va renvoyer la valeur qu'il reçoit en argument. Le script de test est structures.sh.

```
#!/bin/bash

if ./return 0; then
#la valeur de renvoi sera 0 quand la boucle aura été exécutée
echo "Hello"
fi

if ./return 1; then
#ici c'est la condition else qui sera remplie
echo "Hello"
else
echo "Bye"
fi

for i in 1 2 5 2
#les boucles for peuvent s'écrire de cette façon
do
echo $i
done

echo Hello again!

for (( j=1; j<=5; j++))
#ou encore utiliser la synthaxe classique comme en C ou Java
do
echo $j
done

k=4
while ((k>0))
do
echo $k
k=$((k-1))
done
```

Le résultat à l'exécution est

```
$ ./structures.sh
Hello
Bye
1
2
5
2
Hello again!
1
2
3
4
5
4
```

(suite sur la page suivante)

(suite de la page précédente)

```
3
2
1
```

CHAPITRE 7

Gestion des processus

Les systèmes d'exploitation de type Unix sont multitâches et multi-utilisateurs. Cela signifie qu'il est possible d'exécuter simultanément plusieurs programmes qui appartiennent potentiellement à différents utilisateurs. Sous Unix, l'unité d'exécution d'un programme est appelée un processus. Lorsque vous exécutez un programme C que vous avez compilé depuis la ligne de commande, le shell lance un nouveau processus. Chaque processus est identifié par le système d'exploitation via son pid ou process identifier. Ce pid est alloué par le système d'exploitation au moment de la création du processus. À tout instant, le système d'exploitation maintient une table des processus qui contient la liste de tous les processus qui sont en cours d'exécution. Comme nous aurons l'occasion de nous en rendre compte plus tard, cette table contient énormément d'informations qui sont utiles au système. À ce stade, l'information importante qui se trouve dans la table des processus est le pid de chaque processus et l'utilisateur qui possède le processus. La commande ps(1) permet de consulter de façon détaillée la table des processus sur un système Unix. Voici un exemple d'utilisation de cette commande sur un système Linux.

```
$ ps u
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
obo      16272  0.0  0.0 110464  1884 pts/1    Ss   11:35   0:00 -bash
obo      16353  0.0  0.0 110184  1136 pts/1    R+   11:43   0:00 ps u
```

Dans cet exemple, l'utilisateur `obo` possède actuellement deux processus. Le premier est l'interpréteur de commande bash(1) et le second le processus ps(1). L'interpréteur de commande a `16272` comme pid tandis que le pid de ps(1) est 16353.

ps(1) n'est pas la seule commande permettant de consulter la table des processus. Parmi les autres commandes utiles, on peut mentionner top(1) qui permet de visualiser les processus qui s'exécutent actuellement et le temps CPU qu'ils consomment ou pstree(1) qui présente les processus sous la forme d'un arbre. Sous Linux, le répertoire `/proc`, qui est documenté dans proc(5) contient de nombreux pseudos fichiers avec énormément d'informations relatives aux processus qui sont en cours d'exécution. Parcourir le répertoire `/proc` et visualiser avec `less(1)` les fichiers qui s'y trouvent est une autre façon de consulter la table des processus sous Linux.

Pour comprendre le fonctionnement des processus, il est intéressant d'expérimenter avec le processus ci-dessous. Celui-ci utilise l'appel système getpid(2) pour récupérer son pid, l'affiche et utilise la fonction sleep(3) de la librairie pour se mettre en veille pendant trente secondes avant de se terminer.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
  unsigned int sec=30;
  int pid=(int) getpid();

  printf("Processus : %d\n",pid);
  printf("[pid=%d] Sleep : %d secondes\n",pid, sec);
  sec=sleep(sec);
  if(sec==0) {
    printf("[pid=%d] Fin du processus\n",pid );
    return(EXIT_SUCCESS);
  }
  else {
    printf("[pid=%d] Interrompu alors qu'il restait %d secondes\n",pid,sec);
    return(EXIT_FAILURE);
  }
}
```

Ce programme peut être compilé avec gcc(1) pour produire un exécutable.

```
$ ls -l getpid*
-rw-r--r--  1 obo  obo   608 10 fév 12:11 getpid.c
$ gcc -Wall -o getpid getpid.c
$ ls -l getpid*
-rwxr-xr-x  1 obo  obo  8800 10 fév 12:12 getpid
-rw-r--r--  1 obo  obo   608 10 fév 12:11 getpid.c
```

Cet exemple utilise la commande ls(1) pour lister le contenu d'un répertoire. L'argument `-l` permet de d'obtenir pour chaque fichier son nom, sa date de modification, sa taille, l'utilisateur et le groupe auquel il appartient ainsi que ses permissions. Sous Unix, les permissions associées à un fichier sont divisées en trois blocs. Le premier bloc correspond aux permissions qui sont applicables à l'utilisateur qui possède le fichier. Pour l'exécutable `getpid`, les permissions du propriétaire sont `rwx`. Elles indiquent que le propriétaire peut lire le fichier (permission `r`), l'écrire ou l'effacer (permission `w`) et l'exécuter (permission `x`). Sous Unix, seuls les fichiers qui possèdent la permission à l'exécution peuvent être lancés depuis l'interpréteur. Ces permissions peuvent être modifiées en utilisant la commande chmod(1). Les deux autres blocs de permissions sont relatifs aux membres du même groupe que le propriétaire et à un utilisateur quelconque. Nous y reviendrons plus en détail lorsque nous abordons les systèmes de fichiers. En pratique, il est important de savoir qu'un fichier shell ou un fichier compilé qui n'a pas le bit de permission `x` ne peut pas être exécuté par le système. Ceci est illustré par l'exemple ci-dessous.

```
$ chmod -x getpid
$ ls -l getpid*
-rw-r--r--  1 obo  obo  8800 10 fév 12:12 getpid
-rwxr-xr-x  1 obo  obo  8800 10 fév 12:11 getpid.o
$ ./getpid
-bash: ./getpid: Permission denied
$ chmod +x getpid
$ ./getpid
Processus : 11147
```

L'interpréteur de commande bash(1) permet lancer plusieurs processus en tâche de fond. Cela se fait en suffixant la commande avec `&`. Des détails complémentaires sont disponibles dans la section `JOB CONTROL` du manuel de bash(1). Lorsqu'un processus est lancé en tâche de fond, il est détaché et n'a plus accès à l'entrée standard. Par contre, il continue à pouvoir écrire sur la sortie standard et la sortie d'erreur standard. L'exemple ci-dessous illustre l'exécution de deux instances de `getpid`.

```
$ ./getpid &
[1] 10975
$ Processus : 10975
[pid=10975] Sleep : 30 secondes
$ ./getpid &
[2] 10976
$ Processus : 10976
```

(suite sur la page suivante)

(suite de la page précédente)

```
[pid=10976] Sleep : 30 secondes
ps u
USER   PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
obo   8361   0,0  0,0  2435548    208 s003  S+    9:24     0:00.14 -bash
obo  10975   0,0  0,0  2434832    340 s000  S    12:05     0:00.00 ./getpid
obo  10976   0,0  0,0  2434832    340 s000  S    12:05     0:00.00 ./getpid
[pid=10975] Fin du processus
[pid=10976] Fin du processus
[1]-  Done                    ./getpid
[2]+  Done                    ./getpid
```

Ces deux instances partagent la même sortie standard. En pratique, lorsque l'on lance un processus en tâche de fond, il est préférable de rediriger sa sortie et son erreur standard. Lorsque l'on développe de premiers programmes en C, il arrive que celui-ci se lance dans une boucle infinie. Deux techniques sont possibles pour interrompre un tel processus qui consomme inutilement les ressources de la machine et peut dans certains cas la surcharger fortement.

Si le programme a été lancé depuis un shell, il suffit généralement de taper sur *Ctrl-C* pour interrompre son exécution, comme dans l'exemple ci-dessous.

```
$ ./getpid
Processus : 11281
[pid=11281] Sleep : 30 secondes
^C
```

Parfois cependant *Ctrl-C* n'est pas suffisant. C'est le cas notamment lorsqu'un processus a été lancé en tâche de fond. Dans ce cas, la meilleure technique est d'utiliser ps(1) pour trouver l'identifiant du processus et l'interrompre via la commande kill(1). Cette commande permet d'envoyer un signal au processus. Nous verrons plus tard le fonctionnement des signaux sous Unix. À ce stade, le signal permettant de terminer avec certitude un processus est le signal `KILL`. C'est celui qui est utilisé dans l'exemple ci-dessous.

```
$ ./getpid &
[1] 11285
$ Processus : 11285
[pid=11285] Sleep : 30 secondes
ps
PID TTY           TIME CMD
384 ttys000      0:00.32 -bash
11285 ttys000    0:00.00 ./getpid
$ kill -KILL 11285
$ ps
 PID TTY           TIME CMD
 384 ttys000    0:00.33 -bash
 [1]+  Terminated              ./getpid
```

CHAPITRE 8

GCC

Le compilateur gcc(1) est un compilateur largement utilisé pour les programmes écrits en C. Il permet de produire un fichier *exécutable*, qui pourra être exécuté pour effectuer ses opérations, à partir des fichiers *source*, qui contiennent le programme écrit en C. Les fichiers *source* ont en général l'extension `.c`, alors que le fichier *exécutable* n'a généralement pas d'extension (du moins dans le cadre de ce cours).

Soit un fichier *source* `helloworld.c` contenant le code C suivant :

```c
#include <stdlib.h>
#include <stdio.h>

// Main function, prints "Hello world !"
int main(int argc, char const *argv[]) {
    printf("Hello world !\n");

    return EXIT_SUCCESS;
}
```

Pour compiler ce programme et produire l'exécutable `prog`, il suffit d'utiliser gcc(1) :

```
$ gcc helloworld.c -o prog
```

Discutons de chacune des parties de cette commande :

- `gcc` indique l'outil shell utilisé, ici gcc(1)
- `helloworld.c` indique le fichier source, en C
- l'option `-o` permet de spécifier le nom du fichier produit, qui dans ce cas est le fichier exécutable
- `prog` est le nom du fichier exécutable

On peut maintenant exécuter le programme `prog` :

```
$ ./prog
Hello world !
```

Cette description basique de gcc(1) est suffisante pour le début de ce cours. Pour savoir comment compiler des programmes constitués de plusieurs fichiers sources, veuillez vous référer à la partie De grands programmes en C de la partie Théorie du syllabus.

8.1 Compléments

De manière technique, le travail du compilateur peut être découpé selon 4 étapes distinctes :

- Appel du préprocesseur `cpp` : Supprime les commentaires, inclus les *#include* et évalue les macros
- Appel du compilateur `cc1` : Génère un fichier assembleur (.as)
- Appel de l'assembleur `as` : Génère le fichier objet (.o)
- Appel du de l'éditeur de liens `ld` : Génère l'exécutable

Différentes options peuvent être utilisé avec gcc :

- **-E** : Appelle uniquement le préprocesseur
- **-S** : Appelle uniquement le préprocesseur et le compilateur
- **-C** : Appelle le préprocesseur, le compilateur et l'assembleur
- **-o nom** : Détermine le nom du fichier de sortie
- **-g** : Option nécessaire pour générer les informations symboliques de débogage avec gdb
- **-On** : Indique le niveau d'optimisation où n est compris entre 0 et 3
- **-Wall** : Active tout les warnings
- **-Werror** : Considère tout les warnings comme des erreurs
- **--help** : Messages d'aide

Notons que les trois premières options ne présentent pas d'intérêt pour ce cours.

CHAPITRE 9

GDB

9.1 Liste des commandes

gdb(1) permet de débugger vos programmes plus facilement en permettant d'analyser l'état durant l'exécution. Pour pouvoir analyser votre exécutable avec gdb(1), vous devez ajouter les symboles de débuggage lors de la compilation en utilisant l'option `-g` de gcc(1) :

```
gcc -g gdb.c -o my_program
```

L'option -g de gcc(1) place dans l'exécutable les informations sur les noms de variables, mais aussi tout le code source.

Lancez gdb avec la commande `gdb my_program`. Ceci va vous ouvrir la console de `gdb` qui vous permet de lancer, le programme et de l'analyser. Pour démarrer le programme, tapez `run`. gdb va arrêter l'exécution au premier problème trouvé. Votre programme tourne encore pour l'instant. Arrêtez-le avec la commande `kill`.

9.1.1 Breakpoint

Pour analyser un programme, vous pouvez y placer des breakpoints. Un breakpoint permet de mettre en pause l'exécution d'un programme à un endroit donné pour pouvoir afficher l'état des variables et faire une exécution pas-à-pas. Pour mettre un breakpoint, vous avez plusieurs choix :

- `break [function]` met en pause l'exécution à l'appel de la fonction passée en argument à la commande
- `break [filename:linenumber]` spécifie le fichier du code source et la ligne à laquelle l'exécution doit s'arrêter
- `delete [numberbreakpoint]` supprime le breakpoint spécifié

Note : Chaque breakpoint est caractérisé par un numéro. Pour obtenir la liste des breakpoints utilisés `info break`

9.1.2 Informations à extraire

Une fois un breakpoint placé, plusieurs informations peuvent être extraites via gdb(1) :

- `print [variablename]` affiche la valeur de la variable dans son format de base. Il est possible de connaître la valeur pointée en utilisant `*` ainsi que l'adresse de la variable avec `&`.

```
Il est aussi possible de modifier une variable avec ``set variable [nom_
↪variable] = [valeur]``.
De façon similaire avec ``print [nom_variable] = [valeur]``.
```

— `info reg [registre]` affiche les informations sur tous les registres si aucun registre n'est explicitement spécifié. `info reg eax` donne le même résultat que `print $eax`.

```
Il est intéressant de noter qu'il est possible d'afficher une variable␣
↪sous le format spécifié. Pour cela, remplacer ``print`` par :
* ``p/x`` - affiche en format hexadécimal la variable spécifiée
* ``p/d`` - en format entier signé
* ``p/f`` - en format floating point
* ``p/c`` - affiche un caractère.
```

— `backtrace` ou `bt` affiche la pile des appels de fonctions.

```
Il est possible de naviguer dans la pile des appels à l'aide de ``up`` et␣
↪``down``. Ces deux commandes montent et descendent respectivement dans␣
↪la pile. C'est très utile car il est possible de modifier le contexte␣
↪dans lequel on se trouve pour afficher les variables.
```

— `info frame` donne des informations sur la frame actuelle.
— `list` affiche les lignes de codes entourant le break. On peut donc facilement voir le code posant un problème ou analyser le code avant de faire une avancée pas à pas.
— `show args` affiche les arguments passés au programme.
— `info breakpoints` affiche les breakpoints
— `info diplays` affiche les displays
— `info func [fonctionname]` affiche le prototype d'une fonction

9.1.3 Avancement de l'exécution

Quand vous avez acquis suffisamment d'informations sur le programme, vous avez plusieurs choix pour continuer son exécution :

— `next` exécute la prochaine instruction de votre code source, mais sans rentrer dans des fonctions externes.
— `step` exécute la prochaine instruction de votre code source, mais en entrant dans le code des fonctions appelées.
— `continue` continue le reste de l'exécution jusqu'au prochain breakpoint.

9.1.4 Automatisation

Lors d'un débuggage long et fastidieux, il est parfois nécessaire d'exécuter certaines commandes à chaque breakpoint.

— `commands [numerobreakpoint]` définit une liste de commandes associées à un breakpoint. Celles ci seront exécutées quand on s'arrêtera sur ce breakpoint. Il suffit de taper les commandes à effectuer les unes après les autres et de terminer par `end`. Si vous ne fournissez pas de numéro, les commandes sont assignées au dernier breakpoint créé.
— `display [variablename]` affiche la variable à chaque breakpoint.

9.1.5 Gestion des Signaux

En plus des breakpoints, gdb(1) interrompt l'exécution du programme en cours lorsqu'il intercepte certains signaux d'erreurs comme les signaux `SIGSEGV` et `SIGINT`. gdb(1) permettra alors de corriger plus facilement certaines erreurs comme les erreurs de segmentation ou les problèmes de deadlocks.

Il est possible de gérer le comportement de gdb(1) lorsque des signaux sont interceptés. Tout d'abord, la commande `info signals` permet d'afficher la liste des signaux reconnus par gdb(1) ainsi que la façon dont il les traite (par exemple interrompre le programme en cours ou non). On peut changer la façon de traiter un signal avec la commande `handle [SIGNAL] [HANDLING...]` où `[SIGNAL]` est le signal à intercepter (son numéro ou son nom complet) et `[HANDLING]` la façon de traiter ce signal par gdb(1) [1]. Par exemple, la commande `handle SIGALRM stop print` permet d'interrompre le programme et d'afficher un message quand gdb intercepte le signal `SIGALRM`.

1. Une liste plus complète des mots-clés utilisables pour modifier le comportement de gestion des signaux peut-être consultée ici : ftp://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_38.html .

Localiser un signal

Avec gdb(1), il est possible de localiser un signal et de débugger certaines erreurs comme une erreur de segmentation. En effet, lorsque gdb(1) interrompt le programme en cours après l'interception d'un signal d'erreur comme `SIGSEGV`, il est possible de trouver la ligne du programme à laquelle le signal a été intercepté en tapant le mot-clé `where` une fois le programme interrompu (il est cependant nécessaire d'avoir compilé le programme avec l'option `-g` de `gcc` pour trouver la ligne précise). Ensuite, grâce aux commandes expliquées plus tôt, il est possible de vérifier les valeurs des variables lors de l'interception du signal pour trouver l'origine du problème.

En plus de localiser facilement les erreurs de segmentation dans un programme, vous pourrez analyser plus aisément les problèmes de deadlock des threads. En effet, lorsque le programme est lancé sur le shell et que vous remarquez un deadlock, vous pouvez appuyer sur `CTRL + C` pour lancer le signal `SIGINT` au programme. Cela permettra de trouver les endroits où bloquent les différents threads du programme à l'aide des commandes décrites dans la section de débuggage des threads ci-dessous.

9.1.6 Extraction de code assembleur

— `disas` affiche le code assembleur
— `disas /m blah` met en correspondance le code assembleur et le code source

Pour arrêter la console de gdb, tapez `quit`.

9.2 Illustration avec des exemples

9.2.1 Premier programme

Le premier programme est `src/calc.c`. Compilez-le et exécutez le pour vous apercevoir que le programme est erroné. A priori vous avez peu, ou pas, d'informations sur l'erreur. Lancez donc gdb à l'aide de `gdb calc` puis lancez le programme avec `run`.

```
Program received signal SIGFPE, Arithmetic exception.   => Exception␣
↪arithmetique
0x0000000000400553 in calc (a=165, b=4) at calc.c:10    => Dans la␣
↪fonction calc du fichier calc.c à la ligne 10

10                      res = (a*5 -10) / (b-i);        => Affichage de␣
↪la ligne problématique
```

Le premier réflexe doit être `list` pour observer le code. Puisque le problème vient de la ligne 10 dans la boucle, nous allons nous arrêter à la ligne 10 avec `break 10` et relancer le programme. Le programme va s'arrêter avant le début de la boucle. Utilisez `print a` et `print b` pour connaître les arguments reçus par calc.

```
Il est intéressant de noter une particularité du language C par rapport à␣
↪java : une variable déclarée n'est pas initialisée à 0 par défaut, elle␣
↪reprend juste la valeur de la mémoire avant son affectation. ``print␣
↪i`` et ``print res`` vous donneront donc des résultats aléatoires.
```

Puisque le problème vient du calcul arithmétique, placez un break sur cette ligne pour pouvoir observer à chaque itération les variables. `break 9` puis `commands` qui permet d'automatiser des commandes. Nous rajouterons comme commandes :

```
* ``echo i : ``
* ``print i``
* ``echo b : ``
* ``print b``
* ``echo numerateur : ``
* ``print a*5 -10``
* ``echo denominateur : ``
* ``print b-i``
* et enfin ``end`` pour terminer la liste de commandes.
```

Il ne reste plus qu'à avancer avec `continue` pour aller de breakpoint en breakpoint et d'observer les variables pour comprendre le problème. On va pouvoir deviner que le problème vient d'un dénominateur nul. Pour résoudre ce problème, il faut passer une valeur plus grande que 6 à calc lors de son appel depuis la fonction main. `list main` suivi de plusieurs `list` permet de visualiser la main. On peut repérer l'appel de la fonction calc à la ligne 18.

Supprimez les anciens break avec `delete [numerobreakpoint]` le numéro du breakpoint est connu via `info break`. Rajoutez un break à la ligne 18, `break 18` et lancez le programme. `set variable m = 10` pour assigner la valeur 10 à la variable m. Puis continuez l'exécution du programme. Celui se terminera normalement puisque il n'y a plus de division par zéro.

9.2.2 Deuxième programme

Le deuxième programme est appelé `src/recursive.c`. Celui ne présente aucun bug et se déroulera normalement. Toutefois, il est intéressant d'utiliser gdb(1) pour bien comprendre les différents contextes au sein d'un programme. Mettez un break sur la fonction `factTmp` avec `break factTmp` et ajoutez automatiquement à ce breakpoint la commande `backtrace`, via `commands`. Ensuite, lancez le programme. `backtrace` vous permet de visualiser les appels de fonction effectués. Nous pouvons voir que la fonction `factTmp` a été appelée par `factTerminal`, elle même appelée par la fonction `main`.

```
#0  factTmp (acc=1, nbr=6) at recursive.c:8
#1  0x0000000000040057d in factTerminal (a=6) at recursive.c:17
#2  0x0000000000400598 in main (argc=1, argv=0x7fffffffe1b8) at recursive.
↪c:23
```

Essayez d'afficher les variable `globalVar` puis `localVar`. Vous remarquerez qu'il n'est pas possible d'afficher `localVar` puisque cette variable ne fait pas partie de l'environnement contextuel de factTmp. Pour afficher cette variable, il faut remonter la liste des appels. `up` permettra de remonter les appels pour pouvoir afficher `localVar`. Une fois la variable affichée, redescendez avec `down` et continuez 4 fois le programme après le breakpoint. Vous remarquerez que la liste des appels s'allonge à chaque appel récursif, ce qui est tout à fait normal.

Naviguez dans les appels récursifs de factTmp en affichant les valeur de `globalTmp`, `tmp`, `acc` et `nbr`. Il est important de bien comprendre que la variable statique `globalTmp` est commune à tous les appels de la fonction `factTmp` et un changement de cette variable dans un des appels récursifs modifie la variable des autres appels. A contrario, la variable local ainsi que les arguments sont propres à chaque appel.

Vous pouvez maintenant terminer le programme.

9.2.3 Troisième programme

Le troisième programme est `src/tab.c`. Compilez-le. Ce programme s'exécute correctement, et pourtant, il y contient une erreur. Lancez le programme avec gdb et mettez un breakpoint sur la première instruction, à savoir la ligne 9. Pour comprendre un problème sans savoir où commencer, il est utile de suivre l'évolution des variables.

```
Il est important de savoir que ``print``, ainsi que ``display``,␣
↪supportent les expressions telles que :
        * tab[1], tab[i],...
        * &i, *i,...
```

Avancez instruction par instruction, avec `step` ou `next` et portez attention aux valeurs de tab[i] par rapport à i. Une fois le problème trouvé avec gdb, solutionnez le.

Plus d'informations sur gdb(1) peuvent être trouvées sur :

- https://www.cprogramming.com/gdb.html
- https://developer.ibm.com/articles/l-gdb/
- https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS_C/gdb.html

9.3 Débuggage des threads avec GDB

gdb(1) est aussi utile pour débugger des programmes avec des threads. Il permet de faire les opérations suivantes sur les threads :

- Recevoir une notification lors de la création d'un nouveau thread.
- Afficher la liste complète des threads avec `info threads`.
- Placer un breakpoint dans un thread. En effet, si vous placez un breakpoint dans une certaine fonction, et un thread passe lors de son exécution à travers ce breakpoint, `gdb` va mettre l'exécution de tous les threads en pause et changer le contexte de la console gdb(1) vers ce thread.
- Lorsque les threads sont en pause, vous pouvez manuellement donner la main à un thread en faisant `thread [thread_no]` avec `thread_no` étant l'indice du thread comme indiqué par `info threads`

D'autres commandes pour utiliser gdb(1) avec les threads :

- https://sourceware.org/gdb/current/onlinedocs/gdb/Threads.html

CHAPITRE 10

Valgrind

Plusieurs outils en informatique peuvent vous aider à localiser des bugs dans vos programmes. Parmi ceux-ci, voici deux outils particulièrement utiles pour les problèmes liés à la mémoire :

— valgrind(1) vous permet de détecter des erreurs liés à la gestion de la mémoire (malloc(3), free(3),...)
— gdb(1) permet de voir ce qui se passe « à l'intérieur » de votre programme et comment les variables évoluent.

Le site https://www.cprogramming.com/debugging/ vous donne des techniques de débuggage plus détaillées et explique valgrind(1) et gdb(1) plus en détails.

Valgrind permet de détecter des erreurs liées à la gestion de la mémoire dans vos programmes. Pour utiliser valgrind, lancez la commande valgrind(1) avec votre exécutable comme argument :

Parmi les erreurs que valgrind est capable de détecter nous avons :

— Mémoire non-désallouée : Lors d'un appel à malloc(3), vous obtenez un pointeur vers une zone de mémoire allouée. Si vous « perdez » la valeur de ce pointeur, vous n'avez plus le moyen de libérer cette zone de mémoire. Essayez valgrind(1) avec le petit programme `src/nofree.c`
— Désallouer deux fois la même zone de mémoire : Si vous appelez deux fois free(3) sur la même zone de mémoire, valgrind(1) va détecter cette erreur. Essayez-le avec le petit programme `src/twofree.c`
— Accès en dehors des limites d'une zone mémoire : Si vous allouez une zone de mémoire d'une certaine taille (par exemple un table de 10 chars) et que vous accédez à une adresse qui excède cette zone (par exemple vous accédez au 11ième élément) vous aurez probablement une `Segmentation fault`. Valgrind permet de détecter ces erreurs et indique l'endroit dans votre code où vous faites cet accès. Essayez-le avec le petit programme `src/outofbounds.c`

On vous encourage à lancer valgrind(1) sur votre projet pour vérifier que vous n'avez pas introduit de memory-leaks sans le vouloir. valgrind(1) ne remplace pas une écriture attentive du code mais peut permettre de détecter rapidement certaines erreurs courantes. Vous trouverez plus de détails sur les liens suivants :

— https://www.cprogramming.com/debugging/valgrind.html
— https://valgrind.org

10.1 Les bases de `valgrind`

Commençons par le programme le plus simple possible que nous allons tester à l'aide de valgrind(1) :

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
```

(suite sur la page suivante)

(suite de la page précédente)

```
{
   printf("Hello, 1252 !\n");
   return EXIT_SUCCESS;
}
```

Après compilation et l'exécutions avec valgrind(1) nous obtenons :

```
$ gcc -o hello hello.c
$ ./hello
Hello, 1252 !
$ valgrind ./hello
==13415== Memcheck, a memory error detector
==13415== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==13415== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright␣
↪info
==13415== Command: ./hello
==13415==
Hello, 1252 !
==13415==
==13415== HEAP SUMMARY:
==13415==     in use at exit: 0 bytes in 0 blocks
==13415==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==13415==
==13415== All heap blocks were freed -- no leaks are possible
==13415==
==13415== For counts of detected and suppressed errors, rerun with: -v
==13415== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```

Nous pouvons lire dans ce rapport plusieurs informations importante comme le `HEAP SUMMARY` qui résume l'utilisation du tas. Dans notre cas particulier, on peut voir que rien n'a été alloué (en effet, il n'y a pas eu de malloc) et rien n'a été libéré.

L'`ERROR SUMMARY` indique le nombre d'erreurs détectées.

La phrase que nous voulons voir après chaque exécution de valgrind(1) est :

```
All heap blocks were freed -- no leaks are possible
```

Ce qui indique qu'aucun memory leak ne peut avoir lieu dans notre programme.

10.2 Détecter les memory leaks

A présent nous allons montrer comment détecter des fuites de mémoire dans un programme à l'aide de valgrind(1). Testons le programme `src/nofree.c` :

```
$ gcc -o nofree nofree.c
$ valgrind ./nofree
==13791== Memcheck, a memory error detector
==13791== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==13791== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright␣
↪info
==13791== Command: ./nofree
==13791==
==13791==
==13791== HEAP SUMMARY:
==13791==     in use at exit: 6 bytes in 1 blocks
==13791==   total heap usage: 1 allocs, 0 frees, 6 bytes allocated
==13791==
==13791== LEAK SUMMARY:
==13791==    definitely lost: 6 bytes in 1 blocks
==13791==    indirectly lost: 0 bytes in 0 blocks
```

(suite sur la page suivante)

(suite de la page précédente)

```
==13791==      possibly lost: 0 bytes in 0 blocks
==13791==    still reachable: 0 bytes in 0 blocks
==13791==         suppressed: 0 bytes in 0 blocks
==13791== Rerun with --leak-check=full to see details of leaked memory
==13791==
==13791== For counts of detected and suppressed errors, rerun with: -v
==13791== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```

Nous remarquons directement que cette fois ci des leaks ont été trouvés par valgrind(1). Celui ci indique en effet la perte de 6 bytes de mémoire sur le tas qui ont été alloués par 1 malloc(3) et qui n'ont pas été libérés avant le `return`.

Maintenant nous savons que nous avons un memory leak, mais valgrind(1) peut faire plus que seulement les détecter, il peut aussi trouver où ont ils lieu. Nous remarquons dans le rapport qu'il conseil de relancer le test avec cette fois ci l'option `--leak-check=full` pour avoir plus de détails sur notre fuite. Nous avons dés lors de nouvelles informations dans `HEAP SUMMARY` :

```
==13818== 6 bytes in 1 blocks are definitely lost in loss record 1 of 1
==13818==    at 0x4A05FDE: malloc (vg_replace_malloc.c:236)
==13818==    by 0x4004DC: main (nofree.c:5)
```

La fuite a donc lieu à la ligne 5 de notre programme qui correspond à :

```
char *ptrChars = (char *)malloc(6 * sizeof(char));
```

On sait maintenant quel est le malloc(3) responsable du leak, et il est facile de l'éviter en écrivant `free(ptrChars);` avant le `return`.

10.3 Double free

valgrind(1) ne se contente pas seulement de trouver des memory leaks, il est aussi capable de détecter des doubles free qui peuvent engendrer des corruptions de mémoire. Pour montrer cette fonction de valgrind(1) nous utilisons le petit programme `src/twofree.c`.

```
$ valgrind ./twofree
==13962== Memcheck, a memory error detector
==13962== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==13962== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright␣
↪info
==13962== Command: ./twofree
==13962==
==13962== Invalid free() / delete / delete[]
==13962==    at 0x4A0595D: free (vg_replace_malloc.c:366)
==13962==    by 0x40053F: main (in twofree.c:8)
==13962==  Address 0x4c2d040 is 0 bytes inside a block of size 6 free'd
==13962==    at 0x4A0595D: free (vg_replace_malloc.c:366)
==13962==    by 0x400533: main (in twofree.c:8)
==13962==
==13962==
==13962== HEAP SUMMARY:
==13962==     in use at exit: 0 bytes in 0 blocks
==13962==   total heap usage: 1 allocs, 2 frees, 6 bytes allocated
==13962==
==13962== All heap blocks were freed -- no leaks are possible
==13962==
==13962== For counts of detected and suppressed errors, rerun with: -v
==13962== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

Ici valgrind(1) nous indique qu'il a trouver une erreur et qu'il s'agit d'un `Invalid free()` à la ligne 8 de notre programme. Facilement trouvé et corrigé !

10.4 Segmentation Fault

Les segmentation faults sont des erreurs courantes lors de la programmation en C/C++. Elles ont lieu lors de l'accès à des zones de mémoire non-allouées. valgrind(1) permet de facilement trouver l'origine des segfaults et de les corriger. Démonstration avec `src/outofbounds.c` :

```
$ gcc -g -o outofbounds outofbounds.c
```

Il est important de compiler avec le drapeau -g pour dire au compilateur de garder les informations de débuggage.

```
$ ./outofbounds
Segmentation fault
$ gcc -g -o outofbounds outofbounds.c
$ ./outofbounds
$ valgrind ./outofbounds
==14236== Memcheck, a memory error detector
==14236== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==14236== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright␣
↪info
==14236== Command: ./outofbounds
==14236==
==14236== Invalid write of size 1
==14236==    at 0x400530: main (outofbounds.c:7)
==14236==  Address 0x4c2d04c is 6 bytes after a block of size 6 alloc'd
==14236==    at 0x4A05FDE: malloc (vg_replace_malloc.c:236)
==14236==    by 0x40051C: main (outofbounds.c:5)
==14236==
==14236==
==14236== HEAP SUMMARY:
==14236==     in use at exit: 0 bytes in 0 blocks
==14236==   total heap usage: 1 allocs, 1 frees, 6 bytes allocated
==14236==
==14236== All heap blocks were freed -- no leaks are possible
==14236==
==14236== For counts of detected and suppressed errors, rerun with: -v
==14236== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

valgrind(1) trouve bien une erreur, à la ligne 7 de notre petit programme.

10.5 Détecter les deadlocks avec `valgrind`

valgrind(1) contient des outils qui vont au-delà des simples tests de l'allocation de la mémoire. Notamment l'outil `helgrind` permet de détecter des deadlocks. Utilisez `helgrind` sur le petit programme `./src/thread_crash.c` en faisant :

```
$ valgrind --tool=helgrind [my binary]

==24314== Helgrind, a thread error detector
==24314== Copyright (C) 2007-2010, and GNU GPL'd, by OpenWorks LLP et al.
==24314== Using Valgrind-3.6.1-Debian and LibVEX; rerun with -h for␣
↪copyright info
==24314== Command: ./thread_crash
==24314==
==24314== Thread #2 was created
==24314==    at 0x512E85E: clone (clone.S:77)
==24314==    by 0x4E36E7F: do_clone.constprop.3 (createthread.c:75)
==24314==    by 0x4E38604: pthread_create@@GLIBC_2.2.5 (createthread.
↪c:256)
==24314==    by 0x4C29B23: pthread_create_WRK (hg_intercepts.c:257)
==24314==    by 0x4C29CA7: pthread_create@* (hg_intercepts.c:288)
==24314==    by 0x400715: main (in /home/christoph/workspace/SINF1252/
↪SINF1252/2012/S6/src/thread_crash)
```

(suite sur la page suivante)

(suite de la page précédente)

```
==24314==
==24314== Thread #2: Exiting thread still holds 1 lock
==24314==    at 0x4E37FB6: start_thread (pthread_create.c:430)
==24314==    by 0x512E89C: clone (clone.S:112)
```

Plus d'informations sur :

— https://valgrind.org/docs/manual/hg-manual.html

CHAPITRE 11

Introduction aux Makefiles

Les Makefiles sont des fichiers utilisés par le programme make(1) afin d'automatiser un ensemble d'actions permettant la génération de fichiers, la plupart du temps résultant d'une compilation.

Un Makefile est composé d'un ensemble de règles de la forme :

```
target [target ...]: [component ...]
    [command]
    ...
    [command]
```

Chaque règle commence par une ligne de dépendance qui définit une ou plusieurs cibles (`target`) suivies par le caractère **:** et éventuellement une liste de composants (`components`) dont dépend la cible. Une cible ou un composant peut être un fichier ou un simple label. Chacune des commandes (`command`) est simplement une ligne de commande shell. Les commandes seront exécutées dans l'ordre de la cible du Makefile. Les commandes peuvent donc être n'importe quelle ligne de commande shell, mais les Makefiles sont en général utilisés pour automatiser la compilation de projets informatiques, et dans ce cas les lignes de commandes s'occuperont de la compilation (par exemple avec la commande `` `gcc ``).

Il est important de se rendre compte que l'espacement derrière les `command` doit impérativement commencer par une *tabulation*. Ça ne peut pas commencer par des espaces. Il ne faut pas non plus confondre la touche tabulation du clavier qui est souvent interprétée par les éditeurs de texte par une indentation et le caractère de tabulation (souvent écrit `\t` comme en C ou en bash) qui sont souvent affichés avec 2, 3, 4 ou 8 espacements en fonction des préférences de l'utilisateur. On parle bien ici du caractère de tabulation. Heureusement, bien que beaucoup de gens configurent leur éditeur de texte pour indenter avec des espaces, la plupart des bons éditeurs reconnaissent que c'est un Makefile et indentent avec des tabulations.

Le fichier suivant reprend un exemple de règle où la cible et le composant sont des fichiers.

```
text.txt: name.txt
    echo "Salut, " > text.txt
    cat name.txt >> text.txt
```

Pour exécuter les commandes fournies dans un Makefile, il suffit d'appeler la commande shell `make` dans le dossier où se situe le Makefile. Lorsque `make` est exécuté en utilisant ce Makefile, on obtient :

```
$ make
make: *** No rule to make target `name.txt', needed by `text.txt'.  Stop.
```

Comme `text.txt` dépend de `name.txt`, il faut que ce dernier soit défini comme cible dans le Makefile ou existe en tant que fichier. Si nous créons le fichier `name.txt` contenant `Tintin` et que `make` est ré-exécuté, on obtient la sortie suivante :

```
$ make
echo "Salut, " > text.txt
cat name.txt >> text.txt
$ cat text.txt
Salut,
Tintin
```

Si les fichiers de dépendance n'ont pas été modifiés, une prochaine exécution de la commande `make` ne fera rien, comme montré ci-dessous. Puisque les Makefiles sont en général utiliser pour compiler des projets, cela permet de ne pas recompiler un projet qui n'aurait pas été modifié.

```
$ make
make: `text.txt' is up to date.
```

Lorsqu'une dépendance change, `make` le détecte et ré-exécute les commandes associées à la cible. Dans le cas suivant, le fichier `name.txt` est modifié, ce qui force une nouvelle génération du fichier `text.txt`.

```
$ echo Milou > name.txt
$ make
echo "Salut, " > text.txt
cat name.txt >> text.txt
$ cat text.txt
Salut,
Milou
```

Comme spécifié précédemment, les Makefiles sont principalement utilisés pour automatiser la compilation de projets. Si un projet dépend d'un fichier source `test.c`, le Makefile permettant d'automatiser sa compilation peut s'écrire de la façon suivante :

```
test: test.c
    gcc -o test test.c
```

Ce Makefile permettra de générer un binaire `test` à chaque fois que le fichier source aura changé.

11.1 Les cibles (targets)

Comme indiqué ci-dessus, une règle d'un Makefile commence par une **cible** ou **target**. Cette cible peut indiquer le nom du fichier qui sera créé par la règle, ou simplement un nom simple pour la règle.

Soit un fichier Makefile contenant 2 règles :

```
target1:
    echo "Target 1"

target2:
    echo "Target 2"
```

En utilisant la commande `make` sans préciser de cible, c'est la première cible du Makefile qui est exécutée :

```
$ make
echo "Target 1"
Target 1
```

Il est également possible de préciser quelle cible exécuter, en donnant la cible en argument lorsqu'on appelle `make` :

```
$ make target1
echo "Target 1"
Target 1
$ make target2
echo "Target 2"
Target 2
```

11.2 Les variables

Les fichiers `Makefile` permettent d'utiliser des variables, qui permettent de stocker potentiellement n'importe quelle valeur. Ces variables peuvent être de deux types différents :

— Les **variables personnalisées**, définies par l'utilisateur, et qui peuvent prendre n'importe quelle valeur.
— Les **variables automatiques**, qui sont des raccourcis pour des valeurs déjà présentes dans le fichier.

Les deux types de variable seront présentés ci-après.

11.2.1 Variables personnalisées

Les variables personnalisées permettent d'associer un nom à potentiellement n'importe quelle valeur. Elles permettent de faciliter l'évolution du fichier, car si une valeur doit changer, on peut se contenter de modifier la variable associée, au lieu de devoir modifier toutes les règles. Celles-ci sont généralement définies au début du fichier, une par ligne comme :

```
CC = GCC
OPT = -ansi
VARIABLE_AU_NOM_TRES_LONG = 1
```

Notez que les noms sont écrits en majuscule par convention. Leur appel est semblable à celui en script shell (bash) excepté les parenthèses après le symbole $. On écrit par exemple `$(CC)`, `$(CFLAGS)`, `$(VARIABLE_AU_NOM_TRES_LONG)`. Make autorise de remplacer les parenthèses par des accolades mais cette pratique est moins répandue.

```
CC = GCC
CFLAGS = -ansi

build:
    $(CC) $(CFLAGS) foo.c -o foo
```

Vous aurez compris qu'ici, la cible `build` effectue la commande `gcc -ansi foo.c -o foo`. Il est très intéressant de savoir que toutes les variables d'environnement présentes lors de l'appel au Makefile sont également disponibles avec la même notation. Vous pouvez donc très bien utiliser la variable `$(HOME)` indiquant le répertoire attribué à l'utilisateur sans la définir.

Il existe six différentes manières d'assigner une valeur à une variable. Nous ne nous intéresserons qu'à quatre d'entre elles.

— La première permet de lier la variable à une valeur (ici value). Mais celle-ci ne sera évaluée qu'à son appel.
— La seconde permet de déclarer une variable et de l'évaluer directement en même temps.
— La troisième permet d'assigner une valeur à la variable uniquement si celle-ci n'en a pas encore.
— La quatrième permet d'ajouter une valeur à une autre déjà déclarée.

Une description détaillée de ces méthodes d'assignation et des deux autres restantes se trouve à l'adresse suivante https://www.gnu.org/software/make/manual/make.html#Setting

11.2.2 Variables automatiques

Les variables automatiques sont des raccourcis, propres à la syntaxe des fichiers `Makefile`, qui permettent d'exprimer succinctement des valeurs déjà présentes dans le fichier. Elles sont utilisées dans les commandes formant les différentes règles. Elles sont en général formées de deux caractères spéciaux, le premier étant toujours `$`. Les plus utilisées seront présentées dans cette section.

La variable `$@` référence le nom de la cible. Par exemple, pour compiler un exécutable `prog`, on peut utiliser la règle suivante :

```
prog: src.c
  gcc -o $@ src.c
```

La variable `$<` référence le nom de la première dépendance. Par exemple, pour compiler un exécutable `prog`, on peut utiliser la règle suivante :

```
prog: src.c
  gcc -o prog $<
```

La variable `$^` référence la liste des dépendances. Par exemple, pour compiler un exécutable `prog` basé sur deux fichiers objets, on peut utiliser la règle suivante :

```
prog: src_1.o src_2.o
  gcc -o prog $^
```

D'autres variables existent, mais sont moins utilisées en pratique. Plus d'informations sont disponibles à l'adresse suivante : https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html.

11.3 Les conditions

Les variables ne servent pas uniquement à éviter la redondance d'écriture dans votre fichier. On peut aussi les utiliser pour réaliser des opérations conditionnelles comme :

```
DEBUG = 1

build:
ifeq ($(DEBUG), 1)
    gcc -Wall -Werror -o foo foo.c
else
    gcc -o foo foo.c
endif
```

Ici `ifeq` permet de tester un « si égal ». Il existe aussi l'opération opposée `ifneq` pour « si non-égal ». Remarquez que les conditions ne doivent pas être tabulées au risque d'obtenir une erreur de syntaxe incompréhensible. Les conditions peuvent avoir différentes syntaxes. Vous pouvez les trouver sur cette page https://www.gnu.org/software/make/manual/make.html#Conditional-Syntax

Avec les sections précédentes et la suivante nous allons pouvoir nous aventurer dans la création de Makefiles plus complexes. On peut vouloir effectuer des compilations différentes suivant l'environnement de l'utilisateur comme son OS, son matériel ou juste son nom. Encore une fois Make nous gâte en nous offrant la possibilité d'exécuter des commandes shell dans nos Makefiles. Imaginez avoir besoin d'options de compilation supplémentaires à cause de votre OS que seul vous avez besoin. Vous pouvez effectuer une compilation conditionnelle sur votre nom.

```
USER := $(shell whoami)

build:
ifeq ($(USER), sfeldman)
    gcc -I($HOME)/local/include -o foo foo.c
else
    gcc -o foo foo.c
endif
```

Ici `$(shell whoami)` est un appel à la fonction shell (de Make) qui nous permet d'assigner à la variable `USER`, en évaluant immédiatement l'appel, le résultat de la commande shell (bash) `whoami` renvoyant le nom de l'utilisateur actuel. Cela ne fonctionnera que si la commande `whoami` est disponible dans le shell évidemment.

11.4 La cible .PHONY

Make compare les dates de modification des fichiers produits avec les dates de leur(s) source(s) pour savoir si celles-ci ont été modifiées depuis leur dernière compilation. Cela lui permet de ne pas devoir recompiler des

fichiers qui n'auraient pas changé d'un appel à l'autre. Malheureusement ce comportement qui peut sembler avantageux amène aussi des problèmes, en l'occurrence pour des règles ne produisant aucun fichier. Une solution pour pallier le problème consiste à indiquer que la règle ne crée rien. Pour faire cela il existe une cible spéciale `.PHONY` permettant de définir quelles règles doivent toujours être exécutées à nouveau. Ainsi une règle `.PHONY` ne rencontrera jamais le problème d'être déjà à jour. Une bonne pratique est de déclarer dans `.PHONY` toutes les règles de nettoyage de votre projet.

```
build:
    gcc -o foo foo.c

.PHONY: clean

clean:
    rm -f *.o
```

Cela est aussi pratique pour forcer une nouvelle compilation.

```
build:
    gcc -o foo foo.c

.PHONY: clean rebuild

clean:
    rm -f *.o foo

rebuild: clean build
```

11.5 Compléments

Cette section propose quelques compléments, utiles pour la création de fichiers `Makefile` plus complexes.

11.5.1 Règles d'inférence

Il est possible de définir des règles génériques, qui fonctionneront pour tous les fichiers qui correspondent à un *pattern*. Le pattern est alors exprimé avec le caractère `%`. Par exemple, pour compiler tous les fichiers sources, possédant l'extension `.c`, en fichiers objets correspondant, on peut utiliser la règle suivante :

```
%.o: %.c
    gcc -o $@ -c $^
```

Remarquez que cette règle utilise les **variables automatiques**, décrites plus haut.

11.5.2 Commentaires

Afin de rendre vos Makefiles plus lisibles, vous pouvez y insérer des commentaires en plaçant un croisillon en début de ligne. Cette syntaxe est semblable au script shell.

```
# Commentaire sur
# plusieurs lignes
build:
    gcc -o foo foo.c # commentaire en fin de ligne
```

11.5.3 Commandes silencieuses

Corriger les erreurs de vos Makefiles peut sembler difficile lorsque vous êtes baignés dans un flux d'instructions. Vous pouvez néanmoins régler leur verbosité. Il est possible de rendre silencieuse une commande en plaçant une arobase devant. Ceci indique juste à Make de ne pas imprimer la ligne de commande. La sortie standard de cette commande restera visible.

```
build:
    @echo "Building foo"
    @gcc -o foo foo.c
```

Pour plus d'informations en français sur l'écriture ou utilisation des Makefiles voir *[DeveloppezMake]*.

Documentation complète en anglais sur le site officiel *[GNUMake]*.

CHAPITRE 12

CUnit : librairie de tests

CUnit est une librairie de tests unitaires en C. Cette librairie vous sera utile lors de développement de projets en C.

12.1 Installation

CUnit n'est pas installé par défaut sur les machines des salles. Vous devez donc l'installer par vous-même. Le reste de cette section a pour but de vous aider dans l'installation de celle-ci.

La première étape consiste à récupérer les sources de CUnit sur https://sourceforge.net/projects/cunit/files/. Les sources se trouvent dans une archive `CUnit-*-src.tar.bz2` et la dernière version devrait se nommer `CUnit-2.1-3-src.tar.bz2`. Une fois l'archive téléchargée, ouvrez un terminal et placez-vous dans le dossier où se trouve celle-ci. Exécutez :

```
$ tar xjvf CUnit-2.1.-3.tar.bz2
$ cd CUnit-2.1-3
$ ./bootstrap
$ ./configure --prefix=$HOME/local
$ make
$ make install
```

Une fois ces commandes exécutées, la librairie ainsi que ses fichiers d'entête sont installés dans le dossier `$HOME/local` (`$HOME` est en fait une variable bash qui définit votre répertoire principal). Comme vous n'avez pas les droits administrateur, vous ne pouvez pas installer d'application ni de librairie dans les chemins classiques (c.-à-d., par exemple dans `/usr/lib`, `/usr/include`, `/usr/bin`). C'est pour cela que nous installons la librairie dans un dossier local.

12.2 Compilation, édition des liens et exécution

Comme la librairie n'est pas installée dans les chemins classiques, il faut pouvoir dire à gcc où se trouvent les fichiers d'entête ainsi que la librairie afin d'éviter les erreurs de compilation. Pour cela, il faut spécifier à la compilation l'argument `-I${HOME}/local/include` afin de lui dire qu'il doit également aller chercher des fichiers d'entête dans le dossier `$HOME/local/include` en plus des chemins classiques tels que `/usr/include` et `/usr/local/include`.

Lors de l'édition des liens avec le linker, il faut spécifier où se trouve la librairie dynamique afin de résoudre les symboles. Pour cela, il faut passer l'argument `-lcunit` pour effectuer la liaison avec la librairie CUnit ainsi

que lui spécifier `-L${HOME}/local/lib` afin qu'il cherche également des librairies dans le dossier `$HOME/local/lib`.

Lors de l'exécution, il faut également spécifier où se trouvent les librairies. Par exemple pour un binaire `test` qui utilise la librairie CUnit, on peut exécuter :

```
$ export LD_LIBRARY_PATH=$HOME/local/lib:$LD_LIBRARY_PATH
$ ./test
```

12.3 Utilisation

Dans CUnit, on retrouve toutes les fonctions nécessaire pour gérer un ensemble de suites de tests. Cet ensemble forme un catalogue ; il est composé d'une ou plusieurs suite(s) de tests ; chaque suite est composée d'un ou plusieurs tests.

Pour pouvoir concrètement exécuter un ensemble de tests, il est nécessaire de réaliser les différentes étapes suivantes :

1. Programmer les tests
2. Initialiser le catalogue
3. Ajouter les suites de tests dans le catalogue
4. Ajouter les tests dans les suites de tests
5. Exécuter les tests
6. Terminer proprement l'exécution des tests

La suite de la section détaille chacune de ces étapes.

Tout d'abord, il est nécessaire d'écrire les tests. Aucune libraire ne peut les écrire pour vous. Toutefois, CUnit vient avec un certain nombre de macros permettant de vérifier les propriétés qui nous intéressent. Pour pouvoir utiliser ces macros, il est nécessaire d'importer `CUnit.h`. La table suivante récapitule les principales macros. Il est important d'appeler ces macros lorsque l'on rédige les tests, ce sont ces appels qui détermineront si oui ou non, le test est fructueux.

Assertion	Définition
`CU_ASSERT(int expression)`	Vérifie que la valeur est non-nulle (`true`).
`CU_ASSERT_TRUE(value)`	Vérifie que la valeur est non-nulle (`true`).
`CU_ASSERT_FALSE(value)`	Vérifie que la valeur est nulle (`false`).
`CU_ASSERT_EQUAL(actual, expected)`	Vérifie que `actual` est égal à `expected`.
`CU_ASSERT_NOT_EQUAL(actual, expected))`	Vérifie que `actual` n'est pas égal à `expected`.
`CU_ASSERT_PTR_EQUAL(actual, expected)`	Vérifie que le pointeur `actual` est égal au pointeur `expected`.
`CU_ASSERT_PTR_NOT_EQUAL(actual, expected)`	Vérifie que le pointeur `actual` est différent du pointeur `expected`.
`CU_ASSERT_PTR_NULL(value)`	Vérifie que le pointeur est `NULL`.
`CU_ASSERT_PTR_NOT_NULL(value)`	Vérifie que le pointeur n'est pas `NULL`.
`CU_ASSERT_STRING_EQUAL(actual, expected)`	Vérifie que la chaîne de caractère `actual` est égale à la chaîne de caractère `expected`.
`CU_ASSERT_STRING_NOT_EQUAL(actual, expected)`	Vérifie que la chaîne de caractère `actual` n'est pas égale à la chaîne de caractère `expected`.
`CU_ASSERT_NSTRING_EQUAL(actual, expected, count)`	Vérifie que les `count` premiers caractères de la chaîne `actual` sont égaux aux `count` premiers caractères de la chaîne `expected`.
`CU_ASSERT_NSTRING_NOT_EQUAL(actual, expected, count)`	Vérifie que les `count` premiers caractères de la chaîne `actual` ne sont pas égaux aux `count` premiers caractères de la chaîne `expected`.
`CU_ASSERT_DOUBLE_EQUAL(actual, expected, granularity)`	Vérifie que `actual` et `expected` ne diffèrent pas plus que `granularity` ($\|actual - expected\| \leq \|granularity\|$)
`CU_ASSERT_DOUBLE_NOT_EQUAL(actual, expected, granularity)`	Vérifie que `actual` et `expected` diffèrent de plus que `granularity` ($\|actual - expected\| > \|granularity\|$)
`CU_PASS(message)`	Ne vérifie rien mais notifie que le test est réussis
`CU_FAIL(message)`	Ne vérifie rien mais notifie que le test est raté

Par exemple, les méthodes ci-dessous vérifie chacune certaines propriétés.

```
void test_assert_true(void)
{
  CU_ASSERT(true);
}

void test_assert_2_not_equal_minus_1(void)
{
  CU_ASSERT_NOT_EQUAL(2, -1);
}

void test_string_equals(void)
{
  CU_ASSERT_STRING_EQUAL("string #1", "string #1");
}

void test_failure(void)
{
  CU_ASSERT(false);
}

void test_string_equals_failure(void)
{
  CU_ASSERT_STRING_EQUAL("string #1", "string #2");
}
```

Une fois les tests écrits, il faut initialiser le catalogue (et donc l'infrastructure de tests) en appelant la méthode

`CU_initialize_registry()`. Cette méthode retourne un code d'erreur qu'il est impératif de vérifier pour s'assurer du bon fonctionnement de la vérification des tests. Par exemple,

```
if (CUE_SUCCESS != CU_initialize_registry())
  return CU_get_error();
```

Pour ajouter les suites de tests au catalogue, il faut faire appel à la méthode `CU_add_suite(const char* strName, CU_InitializeFunc pInit, CU_CleanupFunc pClean)`. Comme on peut le voir, cette méthode demande un nom (qui doit être unique pour un catalogue) ainsi que deux pointeurs de fonction. Ces pointeurs de fonction permettent d'exécuter du code avant (typiquement appelé *setup*) ou après (typiquement *teardown*) l'exécution des tests de la suite. Ces méthodes sont utiles pour initialiser un environnement d'exécution pour des tests le nécessitants (par exemple, s'assurer de la présence de fichier, initialiser certaines variables, etc.). Ces méthodes sont bien sûr optionnelles, si aucune n'est nécessaire, il suffit alors de passer `NULL` en paramètre. Par ailleurs, notons que ces méthodes doivent retourner 0 si tout c'est bien passé, un chiffre positif dans le cas contraire. Comme pour l'initialisation du catalogue, il est bien entendu nécessaire de vérifier le code retourné par la méthode. La table suivante décrit les codes d'erreurs.

Code d'erreur	Définition
`CUE_SUCCESS`	Aucune erreur
`CUE_NOREGISTRY`	Erreur d'initialisation
`CUE_NO_SUITENAME`	Nom manquant
`CUE_DUP_SUITE`	Nom non unique
`CUE_NOMEMORY`	Pas de mémoire disponible

Par exemple, le code suivant crée une nouvelle suite de test nommée *ma_suite*, avec une fonction d'initialisation et une fonction de terminaison.

```
int setup(void)  { return 0; }
int teardown(void) { return 0; }
// ...
CU_pSuite pSuite = NULL;
// ...
pSuite = CU_add_suite("ma_suite", setup, teardown);
if (NULL == pSuite) {
  CU_cleanup_registry();
  return CU_get_error();
}
```

Les tests peuvent ensuite être ajoutés à la suite de test. Pour cela, il faut faire appel à la méthode `CU_add_test(CU_pSuite pSuite, const char* strName, CU_TestFunc pTestFunc)`. Comme pour une suite de tests, il est nécessaire de préciser un nom. Ce nom doit être unique pour la suite de test. Le second paramètre est un pointeur vers la fonction de test. A nouveau, il est important de vérifier la valeur de retour de la méthode.

Code d'erreur	Définition
`CUE_SUCCESS`	Aucune erreur
`CUE_NOSUITE`	Suite de tests `NULL`
`CUE_NO_TESTNAME`	Nom manquant
`CUE_DUP_TEST`	Nom non unique
`CUE_NO_TEST`	Pointeur de fonction `NULL` ou invalide
`CUE_NOMEMORY`	Pas de mémoire disponible

Le code suivant ajoute les tests décrits ci-dessus à la suite de test que nous avons créé juste avant.

```
if ((NULL == CU_add_test(pSuite, "Test assert true", test_assert_true)) ||
    (NULL == CU_add_test(pSuite, "Test assert 2 not equal -1", test_
↪assert_2_not_equal_minus_1)) ||
    (NULL == CU_add_test(pSuite, "Test string equals", test_string_
↪equals)) ||
```

(suite sur la page suivante)

(suite de la page précédente)

```
    (NULL == CU_add_test(pSuite, "Test failure", test_failure)) ||
    (NULL == CU_add_test(pSuite, "Test string equals failure", test_
↪string_equals_failure)))
{
  CU_cleanup_registry();
  return CU_get_error();
}
```

Maintenant que le catalogue est initialisé, qu'il contient des suites de tests et que les tests ont été ajoutés à ces suites, il nous est possible d'exécuter ces tests. Il existe plusieurs moyens d'exécuter les tests CUnit, nous présentons uniquement le mode de base, non interactif. Pour les autres modes, referez-vous à la documentation. Pour faire tourner les tests, il suffit d'appeler la méthode `CU_basic_run_tests()` qui appellera tous les tests dans toutes les suites des catalogues référencés. Ensuite, on peut afficher le rapport à l'aide de `CU_basic_show_failures(CU_pFailureRecord pFailure)` et `CU_get_failure_list()`.

```
CU_basic_run_tests();
CU_basic_show_failures(CU_get_failure_list());
```

Avec le programme illustré ci-dessous, la console nous affiche les messages suivants :

```
     CUnit - A unit testing framework for C - Version 2.1-2
     http://cunit.sourceforge.net/


Suite ma_suite, Test Test failure had failures:
    1. cunit.c:24  - false
Suite ma_suite, Test Test string equals failure had failures:
    1. cunit.c:29  - CU_ASSERT_STRING_EQUAL("string #1","string #2")

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      5      5      3      2        0
             asserts      5      5      3      2      n/a

Elapsed time =    0.000 seconds


  1. cunit.c:24  - false
  2. cunit.c:29  - CU_ASSERT_STRING_EQUAL("string #1","string #2")
```

Enfin, il est nécessaire de libérer les ressources en appelant `CU_cleanup_registry()`.

CHAPITRE 13

Introduction à Git

git(1) est un outil de **version control**, c'est-à-dire de gestion de code sous formes de versions. Il est utilisé pour sauvegarder chaque version d'un code source, depuis sa création jusqu'à la dernière version. Il est également très utile dans le cas où plusieurs programmeurs travaillent sur le même projet, car il permet de fusionner les modifications apportées par chaque membre de l'équipe et d'éviter les conflits de code, par exemple un fichier qui aurait été modifié conjointement par 2 personnes. Il est extrêmement utilisé en pratique, et permet de faciliter grandement la gestion du code source lors de projets de taille relativement grande, ou comprenant plusieurs programmeurs.

git(1) a été développé initialement pour la gestion du code source du kernel Linux. Il est aussi utilisé pour la gestion des sources de ce document depuis https://github.com/obonaventure/SyllabusC. On l'utilise le plus souvent à l'aide de l'utilitaire git(1) mais il existe aussi des applications graphiques.

Le code source est sauvegardé dans un *dépôt*, ou *repository*, qui est simplement un dossier contenant le code, et auquel chaque développeur a accès. Ce repository contient un historique de toutes les versions du code, depuis sa création. Chacune des différentes versions est enregistrée dans un *commit*, qui représente les modifications apportées aux différents fichiers du projet depuis la version précédente. On sait ainsi facilement voir ce qui a changé entre deux versions (pas spécialement une version et la suivante) et même restaurer l'état de certains fichiers à une version sauvegardée dans un commit. Du coup, si vous utilisez git(1) pour un projet, vous ne pouvez jamais perdre plus que les changements que vous n'avez pas encore committé. Toutes les versions du codes déjà committées sont sauvegardées et facilement accessibles. Cette garantie est extrêmement précieuse et constitue à elle seule une raison suffisante d'utiliser git(1) pour tous vos projets.

git(1) est souvent utilisé conjointement avec une plateforme en ligne, comme GitHub ou GitLab (via la Forge UCLouvain), qui permet de centraliser le code source, pour que tous les collaborateurs puissent récupérer la dernière version, et qui permet également de consulter le code source en ligne.

13.1 Configuration de l'utilisateur

Lorsqu'on utilise git(1), chaque commit est documenté en fournissant le nom de l'auteur, son email, un commentaire et une description (optionnelle). Pour ne pas devoir spécifier le nom et l'email à chaque fois, il est possible de configurer git(1) pour qu'il les stocke et les utilise pour chaque commit. Pour ce faire, il suffit d'utiliser la commande git-config(1) :

```
$ git config --global user.name "Your name"
$ git config --global user.email name@domain.com
```

L'option `--global` signifie que ces données seront utilisées pour chaque repository sur la machine. En enlevant l'option, les données seront donc uniquement utilisées pour le repository courant.

13.2 Création d'un repository

La première étape pour profiter des capacités de git(1) pour un projet, est de créer un repository qui contiendra le code source du projet. La façon la plus simple de faire est de créer le repository depuis la plateforme en ligne (GitHub ou GitLab), puis le cloner en local. Pour ce faire, la première étape est de créer le repository sur la plateforme en ligne. Cela est relativement simple et ne sera pas décrit dans ce document. Ce repository sera appelé *remote*, car il n'est pas situé en local, mais sur un serveur distant accessible depuis l'Internet, ce qui permet à chaque utilisateur de le consulter pour obtenir la dernière version du code source. Une fois créé, il faut récupérer le lien du repository sur la page web du projet. Le lien peut être sous forme HTTPS ou SSH. Le premier est le choix de base, et le second est choisi pour utiliser une clé ssh pour s'identifier (voir la section *SSH* du syllabus pour plus d'informations). Ensuite, il faut *cloner* le repository en local, avec la commande git-clone(1) :

```
$ git clone LIEN_DU_REPOSITORY
Cloning into 'NOM_DU_REPOSITORY'...
```

Cette commande va cloner le repository dans le dossier courant, de manière à avoir une copie locale du code source sur laquelle travailler.

13.3 Utilisation linéaire de Git

La manière la plus simple d'utiliser git(1) est de façon linéaire, c'est-à-dire que chaque version du code (chaque commit) sera une modification de la précédente, par addition, modification, ou suppression de fichiers. Dans ce cas, après la création du repository contenant le projet, le travail sur le code source suit un schéma, qui est répété pour chaque modification, et qui est le suivant :

- Récupération du dernier commit (`git pull`)
- Modification du code source
- Ajout des modifications au commit (`git add`)
- Sauvegarde du commit (`git commit`)
- Publication des changements sur le remote (`git push`)

Chacune de ces étapes sera décrite ci-après.

13.3.1 Récupération du dernier commit

Avant de travailler sur le code, il faut récupérer en local toutes les modifications qui auraient été apportées au remote entre temps. En effet, si on ne récupère pas ces modifications, des conflits peuvent apparaître, car des fichiers auraient été modifiées en même temps dans deux copies du repository.

Pour récupérer la dernière version du remote, il suffit d'exécuter la commande git-pull(1) :

```
$ git pull
```

Cette commande va appliquer les derniers commits du remote à la copie locale du repository. Ensuite, on peux travailler sur le code et modifier les fichiers.

13.3.2 Ajout des modifications au commit

Lorsque des modifications ont été apportées au code, et qu'on veut les publier sur le remote pour que tous les développeurs aient accès à la dernière version, la première étape est de créer un commit contenant ces modifications.

Imaginons que le repository contient un fichier `main.c` (qui calcule la somme des entiers de 0 à n) qui a été modifié. On peut voir les fichiers qui ont été modifiés avec la commande git-status(1) :

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#     modified:   main.c
```

(suite sur la page suivante)

(suite de la page précédente)

```
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Avec git-diff(1), on peut voir quelles sont les lignes qui ont été retirées (elles commencent par un −) et celles qui ont été ajoutées (elles commencent par un +).

```
$ git diff
diff --git a/main.c b/main.c
index 86601ed..a9e4c4a 100644
--- a/main.c
+++ b/main.c
@@ -2,7 +2,12 @@
 #include <stdlib.h>

 int main (int argc, char *argv[]) {
-  long int sum = 0, i, n = 42;
+  long int sum = 0, i, n;
+  char *end = NULL;
+  n = strtol(argv[1], &end, 10);
+  if (*end != '\0') {
+    return EXIT_FAILURE;
+  }
   for (i = 1; i <= n; i++) {
     sum += i;
   }
```

Si les modifications nous conviennent, il suffit ensuite d'ajouter les fichiers modifiés au commit, avec la commande git-add(1) :

```
$ git add main.c
```

Il est également possible d'ajouter d'un coup tous les fichiers modifiés au commit en utilisant l'option `--all` de git-add(1) :

```
$ git add --all
```

Le commit a été crée, il faut maintenant le sauvegarder, puis le publier sur le remote.

13.3.3 Sauvegarde du commit

Une fois que le commit a été crée, il faut le sauvegarder, pour indiquer au repository qu'on est passé à une nouvelle version. Pour ce faire, on utilise la commande git-commit(1) :

```
$ git commit
```

Cette commande va ouvrir un éditeur de texte pour indiquer un message décrivant le commit. Par défaut, l'éditeur est vim(1). Il s'agit d'un éditeur en ligne de commande, puissant mais très compliqué à utiliser pour les débutants. Il est possible de modifier l'éditeur par défaut en utilisant la commande git-config(1), déjà mentionnée plus haut. Un autre éditeur en ligne de commande, plus simple d'utilisation, est nano(1). Pour choisir nano(1) comme éditeur par défaut, il suffit d'exécuter la commande suivante :

```
$ git config --global core.editor nano
```

Cependant, ouvrir un éditeur de texte à chaque commit peut vite devenir laborieux. En utilisant l'option `-m` de git-commit(1), il est possible de spécifier le message décrivant le commit directement lors de l'appel à la commande git-commit(1) :

```
$ git commit -m "Commit message"
[master 56ce59c] Commit message
 1 file changed, 6 insertions(+), 1 deletion(-)
```

Parmi les options de git-commit(1), il existe aussi l'option `-a` qui peut s'avérer très utile. Cette option permet d'ajouter directement, lors de l'appel à git-commit(1), toutes les modifications qui auraient été apportées à des fichiers **déjà enregistrés dans le repository**. Si de nouveaux fichiers ont été créés, l'option `-a` ne les prendra pas en compte, et il faudra alors passer par la commande git-add(1).

Il est finalement possible de combiner les options `-m` et `-a`, en utilisant l'option `-am`. Cette option permet donc, en une seule commande, d'ajouter toutes les modifications apportées aux fichiers déjà suivis, et de préciser le message du commit, de la façon suivante :

```
$ git commit -am "Commit message"
[master 56ce59c] Commit message
 1 file changed, 6 insertions(+), 1 deletion(-)
```

Il est alors possible de voir le nouveau commit dans l'historique du repository, en utilisant la commande git-log(1) :

```
$ git log
commit 56ce59c54726399c18b3f87ee23a45cf0d7f015d
Author: Benoît Legat <benoit.legat@gmail.com>
Date:   Sun Aug 25 15:37:51 2013 +0200

    Commit message

commit 3d18efe4df441ebe7eb2b8d0b78832a3861dc05f
Author: Benoît Legat <benoit.legat@gmail.com>
Date:   Sun Aug 25 15:32:42 2013 +0200

    First commit
```

Une fois que le commit a été enregistré, il reste à le publier sur le remote, pour que tous les développeurs du projet y aient accès.

13.3.4 Publication du commit sur le remote

Pour que tous les développeurs soient en mesure de voir les dernières modifications qui auraient été apportées en local, il faut que chaque développeur, après avoir créé et enregistré un commit, le publie sur le remote, qui est accessible par tous les développeurs via l'Internet. Pour ce faire, on utilise la commande git-push(1) :

```
$ git push
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 291 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To github.com:user/projectname.git
   80507e3..205842a  master -> master
```

De cette manière, chaque développeur qui voudrait à son tour apporter des modifications au projet, peut appliquer les mêmes étapes, et le remote contiendra toujours la dernière version du code. En résumé, les étapes sont :

- `git pull`
- Modification du code
- `git add`
- `git commit`
- `git push`

13.3.5 Résolution de conflits

Lorsque plusieurs développeurs travaillent sur un même projet, il est possible qu'il apportent des modifications au code en même temps. Dans ce cas, pour le second développeur voulant *push* ses modifications, le *push* sera rejeté :

```
$ git push
To github.com:user/projectname.git
```

(suite sur la page suivante)

(suite de la page précédente)

```
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'github.com:user/projectname.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Cela est dû a fait que le remote a été modifié entre temps par un autre développeur, et donc que le dernier commit n'est pas le même sur le repository local et le remote.

Pour régler ce problème, on commence par faire un `git pull`. Deux cas de figure peuvent alors apparaître. Le premier cas, le plus simple, arrive lorsque les deux développeurs ont modifié des fichiers différents. Dans ce cas, le `pull` va réussir à fusionner les deux versions du repository, et produire un *merge* (une fusion). Un éditeur de texte s'ouvrira pour indiquer un message relatif au merge, et une fois ce message écrit, le *merge* sera effectué :

```
$ git pull
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Total 4 (delta 3), reused 4 (delta 3), pack-reused 0
Unpacking objects: 100% (4/4), done.
From github.com:user/projectname
   4d38eb9..617618b  master     -> origin/master
Merge made by the 'recursive' strategy.
 main.c | 3 +++
 1 file changed, 3 insertions(+)
```

Il ne reste plus qu'à faire un `git push` pour que le *merge* soit publié sur le remote.

Le deuxième cas possible arrive lorsque les deux développeurs ont modifié le même fichier (par exemple `main.c`). Dans ce cas, le `git pull` n'arrivera pas à *merge* automatiquement :

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 1), reused 3 (delta 1)
Unpacking objects: 100% (3/3), done.
From github.com:user/projectname
   80507e3..205842a  master     -> origin/master
Auto-merging main.c
CONFLICT (content): Merge conflict in main.c
Automatic merge failed; fix conflicts and then commit the result.
```

git(1) marque alors dans le fichier `main.c` la ligne en conflit et ce qu'elle vaut dans les deux commits :

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
<<<<<<< HEAD
  return EXIT_SUCCESS;
=======
  return 0;
>>>>>>> 205842aa400e4b95413ff0ed21cfb1b090a9ef28
}
```

La ligne située entre le marqueur `HEAD` et la ligne de séparation est la version présente en local, tandis que la ligne située après la ligne de séparation est celle présente sur le remote. Il est possible de retrouver quels sont les fichiers en conflit en utilisant git-status(1) :

```
$ git status
# On branch master
```

(suite sur la page suivante)

(suite de la page précédente)

```
# You have unmerged paths.
#   (fix conflicts and run "git commit")
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#
#    both modified:      main.c
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Il suffit alors d'éditer le fichier en question, et de ne garder que le contenu voulu dans le fichier :

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
  return EXIT_SUCCESS;
}
```

Il faut ensuite commit et push les modifications pour sauvegarder la fusion :

```
$ git commit -am "Merge conflict"
[master eede1c8] Merge conflict
$ git push
Counting objects: 8, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 478 bytes, done.
Total 4 (delta 2), reused 0 (delta 0)
To github.com:user/projectname.git
   205842a..eede1c8  master -> master
```

Le conflit sera alors résolu, et la dernière version du code sera disponible sur le remote.

13.4 Utilisation non-linéaire de Git

La puissance de git(1) vient du fait qu'il est possible de créer des historiques non-linéaires, plus complexes que l'historique linéaire simple décrit jusqu'à présent. Pour cela, on utilise le concept de *branches*, qui représentent différentes modifications en parallèle du code source.

13.4.1 Branches

Un repository git(1) est divisé en *branches*, qui représentent des évolutions différentes en parallèle du repository. Chaque commit est appliqué sur une seule branche. De cette manière, les branches sont une bonne manière de développer de nouvelles fonctionnalités, sans compromettre une version fonctionnelle du code.

Lors de l'utilisation linéaire de git(1) décrite ci-dessus, toutes les modifications apportées au code se faisaient sur une seule branche, la branche `master`. Il s'agit de la branche de base, sur laquelle toutes les modifications sont apportées, si on ne créé pas explicitement de nouvelle branche. De base, l'historique d'un repository est donc le suivant :

Les commits sont représentés en bleu, et les branches en rouge. L'indication `HEAD` représente l'état actuel du repository sur la copie locale.

Pour créer une nouvelle branche, on utilise la commande git-branch(1), en spécifiant le nom de la nouvelle branche. On peut aussi utiliser cette commande sans argument pour montrer toutes les branches existantes, avec un symbole `*` pour indiquer la branche active (donc là où est situé le marqueur `HEAD`) :

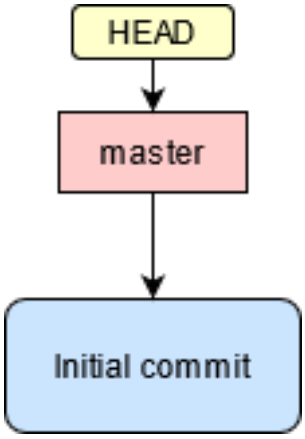


Fig. 1 – Historique initial d'un repository

```
$ git branch branch_1
$ git branch
  branch_1
* master
```

L'historique est désormais le suivant :

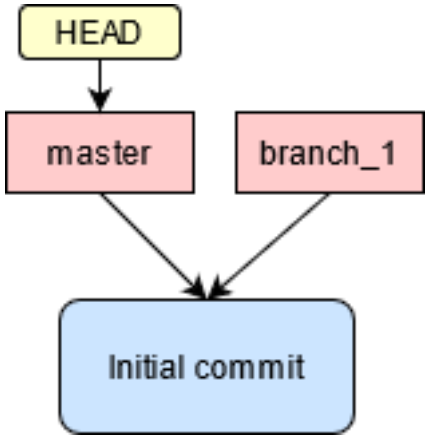


Fig. 2 – Historique après création de `branch_1`

Pour supprimer une branche, on utilise la commande git-branch(1), avec l'option `-d`, et en spécifiant le nom de la branche à supprimer :

```
$ git branch -d branch_1
$ git branch
* master
```

La création d'une branche ne change pas la branche active, ce qui signifie que les modifications apportées au code le seront toujours sur la branche `master`. Pour changer de branche active, il faut utiliser la commande git-checkout(1) :

```
$ git checkout branch_1
Switched to branch 'branch_1'
$ git branch
* branch_1
  master
```

Avec cette commande, le pointeur `HEAD` a été modifié, et pointe maintenant vers la branche `branch_1`. Désormais, les modifications seront bien apportées sur la branche `branch_1`.

Attention, lorsqu'on travaille sur une branche autre que `master`, les simples commandes `git push` ou `git pull` ne fonctionneront pas. A la place, il faut utiliser les commandes suivantes :

- `git push origin branch`
- `git pull origin branch`

Ces commandes fonctionnent également avec la branche `master`, en remplaçant le nom de la branche par `master`.

13.4.2 Fusionner des branches

En général, on utilise les branches pour développer de nouvelles fonctionnalités sans risquer de compromettre la base fonctionnelle du code. Lorsque la fonctionnalité est finie et est fonctionnelle, on veut pouvoir fusionner la branche de base (`master`) avec la branche utilisée pour développer la fonctionnalité (soit `branch`), en appliquant un *merge*. Pour ce faire, il y a deux possibilités :

- Utiliser l'interface web de la plateforme (GitLab ou GitHub). Cette possibilité est la plus simple.
- Utiliser la ligne de commande.

Fusionner des branches depuis l'interface web

La première possibilité est très simple. Un exemple sera donné ici avec GitLab, et est très similaire avec GitHub. Tout d'abord, depuis la page du repository, aller sur la page « *Merge requests* » (« *Pull requests* » sur GitHub) :

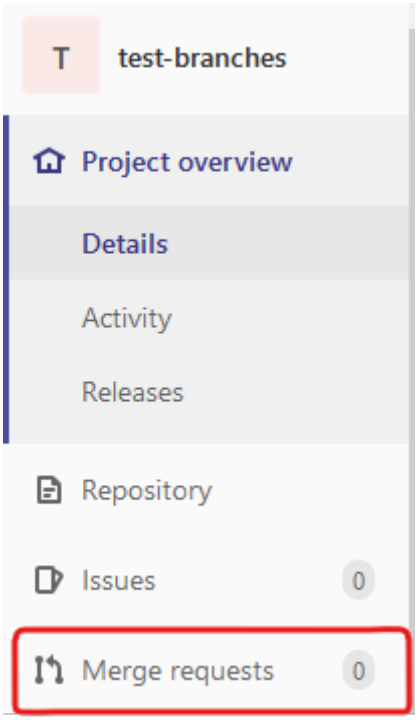


Fig. 3 – Menu du repository sur GitLab

Créez une nouvelle *merge/pull request*. Il faut ensuite choisir les branches *source* et *cible* (*target*). La branche source sera celle avec la nouvelle fonctionnalité, dans notre cas la branche `branch`, tandis que la branche *cible* sera la branche de base, dans notre cas la branche `master`.

Il est possible d'inclure une description à la *merge request*, et de configurer plusieurs options, comme la personne qui doit s'occuper de la *merge request*, ou le fait que la branche source sera supprimée ou pas après la fusion. Une fois la *merge request* créée, s'il n'y a pas de conflit, les branches peuvent être fusionnées automatiquement :

Si les mêmes fichiers ont été modifiés sur les deux branches, il y a conflit, et il est donc impossible de fusionner les branches automatiquement :

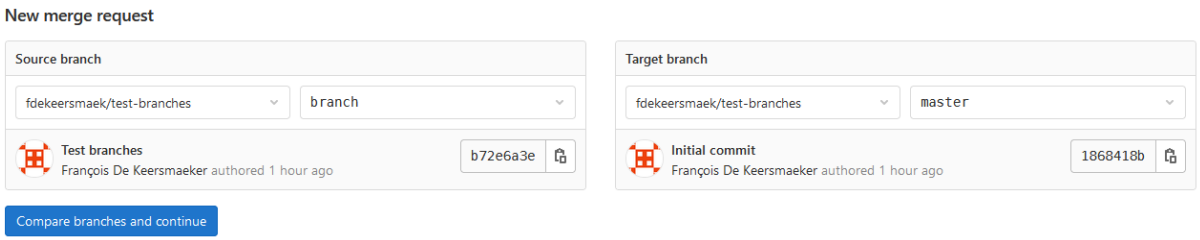


Fig. 4 – Sélection des branches *source* et *cible*

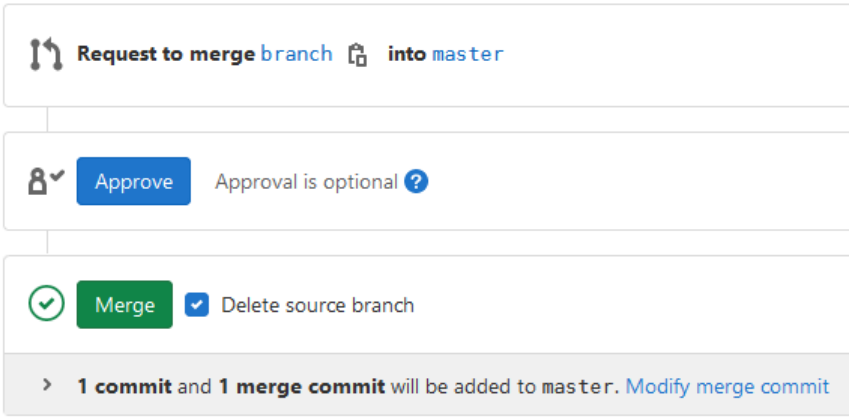


Fig. 5 – Fusion automatique

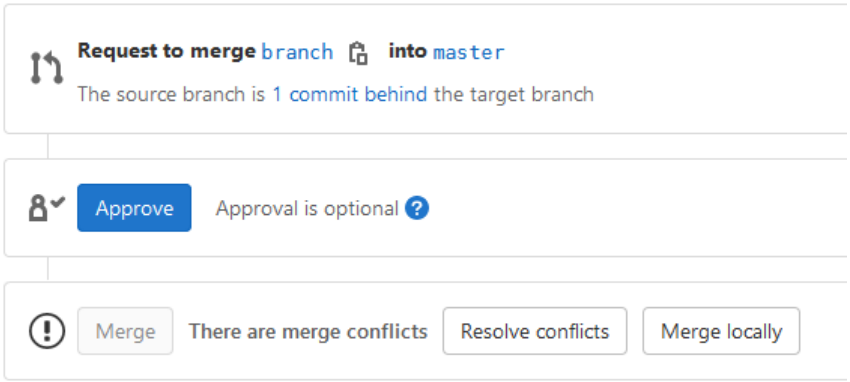


Fig. 6 – Conflits lors de la fusion

Ces conflits peuvent être résolus directement depuis l'interface web, ou en fusionnant les branches localement, puis en réglant les conflits comme expliqué précédemment.

Fusionner des branches localement

Il est également possible de fusionner des branches localement, en utilisant la ligne de commande. Pour cela, on va utiliser la commande git-merge(1), depuis la branche cible (`master`), pour la fusionner avec la branche source (`branch`). Si il n'y a pas de conflit, la fusion est automatique :

```
$ git merge branch
Updating 1f939f3..62cf363
Fast-forward
 branch.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Si les mêmes fichiers ont été modifiés sur les deux branches, il y a conflit, et il faut donc résoudre ces conflits comme expliqué précédemment. Ici, il y a conflit sur le fichier `branch.txt` :

```
$ git merge branch
Auto-merging branch.txt
CONFLICT (content): Merge conflict in branch.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Le fichier `branch.txt` a donc été marqué pour la résolution :

```
<<<<<<< HEAD
Test 1
=======
Test 2
>>>>>>> branch
```

13.5 Autres commandes utiles

Cette section présente d'autres commandes plus avancées de git(1), qui peuvent s'avérer utile.

13.5.1 Afficher l'historique

Pour afficher l'historique, outre l'outil utilisé pour faire les illustrations de ce cours que vous pouvez retrouver https://github.com/blegat/git-dot, il existe la commande git-log(1). Elle est très flexible comme on va le voir. `git log` affiche simplement l'historique à partir de `HEAD`

```
$ git log
commit 0dd6cd7e6ecf01b638cd631697bf9690baedcf20
Merge: eda36d7 6fd2e9b
Author: Benoît Legat <benoit.legat@gmail.com>
Date:   Sun Aug 18 15:29:53 2013 +0200

    Merge branch 'universal'

    Conflicts:
        main.c

commit 6fd2e9bfa199fc3dbca4df87d225e35553d6cd79
Author: Benoît Legat <benoit.legat@gmail.com>
Date:   Sun Aug 18 15:06:14 2013 +0200

    Fix SIGSEV without args

commit eda36d79fd48561dce781328290d40990e74a758
Author: Benoît Legat <benoit.legat@gmail.com>
```

(suite sur la page suivante)

(suite de la page précédente)

```
Date:   Sun Aug 18 14:58:29 2013 +0200

    Add pid/ppid info

...
```

Mais on peut aussi demander d'afficher les modifications pour chaque commit avec l'option `-p`

```
$ git log -p
commit 0dd6cd7e6ecf01b638cd631697bf9690baedcf20
Merge: eda36d7 6fd2e9b
Author: Benoît Legat <benoit.legat@gmail.com>
Date:   Sun Aug 18 15:29:53 2013 +0200

    Merge branch 'universal'

    Conflicts:
        main.c

commit 6fd2e9bfa199fc3dbca4df87d225e35553d6cd79
Author: Benoît Legat <benoit.legat@gmail.com>
Date:   Sun Aug 18 15:06:14 2013 +0200

    Fix SIGSEV without args

diff --git a/main.c b/main.c
index 8ccfa11..f90b795 100644
--- a/main.c
+++ b/main.c
@@ -9,7 +9,7 @@

 // main function
 int main (int argc, char *argv[]) {

 // main function
 int main (int argc, char *argv[]) {
-  if (strncmp(argv[1], "--alien", 8) == 0) {
+  if (argc > 1 && strncmp(argv[1], "--alien", 8) == 0) {
     printf("Hello universe!\n");
   } else {
     printf("Hello world!\n");

commit eda36d79fd48561dce781328290d40990e74a758
Author: Benoît Legat <benoit.legat@gmail.com>
Date:   Sun Aug 18 14:58:29 2013 +0200

    Add pid/ppid info

diff --git a/main.c b/main.c
index 8381ce0..b9043af 100644
--- a/main.c
+++ b/main.c
@@ -5,9 +5,11 @@
 // includes
 #include <stdio.h>
 #include <stdlib.h>
+#include <unistd.h>

 // main function
 int main () {
+  printf("pid: %u, ppid: %u\n", getpid(), getppid());
```

(suite sur la page suivante)

(suite de la page précédente)

```
   printf("Hello world!\n");
   return EXIT_SUCCESS;
 }
```

Il existe encore plein d'autres options comme `--stat` qui se contente de lister les fichiers qui ont changés. En les combinant on peut obtenir des résultats intéressants comme ci-dessous

```
$ git log  --graph --decorate --oneline
*   0dd6cd7 (HEAD, master) Merge branch 'universal'
|\
| * 6fd2e9b Fix SIGSEV without args
| *   88d2c61 Merge branch 'master' into universal
| |\
| * | e0c317a Make it universal
* | | eda36d7 Add pid/ppid info
| |/
|/|
* | c35a8c3 Add Makefile
|/
* c1f2163 Add intro
* b14855e Add .gitignore
* bc620ce Add return
* 76c1677 First commit
```

On ajoute d'ailleurs souvent un raccourci pour avoir ce graphe avec `git lol`.

```
$ git config --global alias.lol "log --graph --decorate --oneline"
```

13.5.2 Sauvegarder des modifications hors de l'historique

On a vu que certaines opérations comme git-checkout(1) nécessitent de ne pas avoir de modifications en conflit avec l'opération.

git-stash(1) permet de sauvegarder ces modifications pour qu'elles ne soient plus dans le *working directory* mais qu'elles ne soient pas perdues. On peut ensuite les appliquer à nouveau avec `git stash apply` puis les effacer avec `git stash drop`.

Reprenons notre exemple de *Changer la branche active* illustré par la figure suivante

```
$ git checkout pid
Switched to branch 'pid'
$ echo "42" >> main.c
$ echo "42" >> .gitignore
$ git stash
Saved working directory and index state WIP on pid: b14855e Add .gitignore
HEAD is now at b14855e Add .gitignore
$ git checkout master
Switched to branch 'master'
$ git stash apply
Auto-merging main.c
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#    modified:   .gitignore
#    modified:   main.c
#
no changes added to commit (use "git add" and/or "git commit -a")
```

On voit que les changements on été appliqués

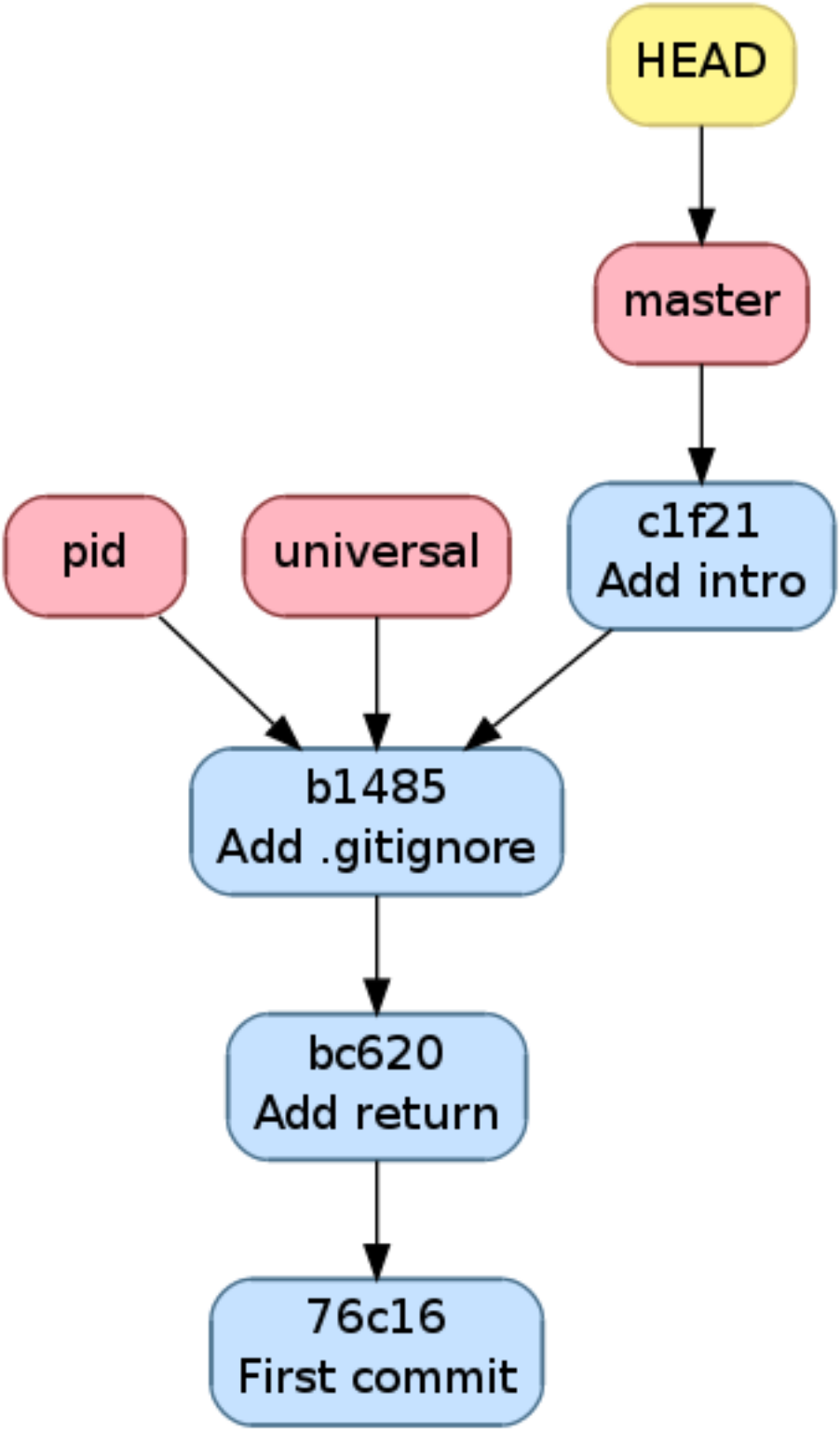


Fig. 7 – Historique après avoir ajouté un commentaire d'introduction

```
$ git diff
diff --git a/.gitignore b/.gitignore
index cba7efc..5df1452 100644
--- a/.gitignore
+++ b/.gitignore
@@ -1 +1,2 @@
 a.out
+42
diff --git a/main.c b/main.c
index 8381ce0..eefabd7 100644
--- a/main.c
+++ b/main.c
@@ -11,3 +11,4 @@ int main () {
   printf("Hello world!\n");
   return EXIT_SUCCESS;
 }
+42
```

On peut alors supprimer le *stash*

```
$ git stash drop
Dropped refs/stash@{0} (ae5b4fdeb8bd751449d73f955f7727f660708225)
```

13.5.3 Modifier un commit récent

Si on a oublié d'ajouter des modifications dans le dernier commit et qu'on ne l'a pas encore *pushé*, on peut facilement les rajouter. Il suffit de donner l'option `--amend` à git-commit(1). Il ajoutera alors les modifications au commit actuel au lieu d'en créer un nouveau.

On peut aussi annuler le dernier commit avec `git reset HEAD^`. git(1) permet aussi de construire un commit qui a l'effet inverse d'un autre avec git-revert(1). Ce dernier construit un commit qui annulera l'effet d'un autre commit. Voyons tout ça par un exemple qui pourrait être le code de *Deep Thought*.

On a un fichier `main.c` contenant

```c
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
  int *n = (int*) malloc(sizeof(int));
  *n = 42;
  printf("%d\n", *n);
  return EXIT_SUCCESS;
}
```

un `Makefile` contenant

```
run: answer
    echo "The answer is `./answer`"

answer: main.c
    gcc -o answer main.c
```

si bien qu'on a

```
$ make
gcc -o answer main.c
echo "The answer is `./answer`"
The answer is 42
$ make
echo "The answer is `./answer`"
The answer is 42
```

(suite sur la page suivante)

(suite de la page précédente)

```
$ touch main.c
$ make
gcc -o answer main.c
echo "The answer is `./answer`"
The answer is 42
```

et un fichier `.gitignore` avec comme seul ligne `answer`.

Commençons par committer `main.c` et `.gitignore` en oubliant le `Makefile`.

```
$ git init
Initialized empty Git repository in /path/to_project/.git/
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#    .gitignore
#    Makefile
#    main.c
nothing added to commit but untracked files present (use "git add" to track)
$ git add .gitignore main.c
$ git commit -m "First commit"
[master (root-commit) 54e48c9] First commit
 2 files changed, 10 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 main.c
$ git log --stat --oneline
54e48c9 First commit
 .gitignore | 1 +
 main.c     | 9 +++++++++
 2 files changed, 10 insertions(+)
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#    Makefile
nothing added to commit but untracked files present (use "git add" to track)
```

On pourrait très bien faire un nouveau commit contenant le `Makefile` mais si, pour une quelconque raison, on aimerait l'ajouter dans le commit précédent, on peut le faire comme suit

```
$ git add Makefile
$ git commit --amend
[master 1712853] First commit
 3 files changed, 15 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 Makefile
 create mode 100644 main.c
$ git log --stat --oneline
1712853 First commit
 .gitignore | 1 +
 Makefile   | 5 +++++
 main.c     | 9 +++++++++
 3 files changed, 15 insertions(+)
```

On voit qu'aucun commit n'a été créé mais c'est le commit précédent qui a été modifié. Ajoutons maintenant un check de la valeur retournée par malloc(3) pour gérer les cas limites

```
$ git diff
diff --git a/main.c b/main.c
index 39d64ac..4864e60 100644
--- a/main.c
+++ b/main.c
@@ -3,6 +3,10 @@

 int main (int argc, char *argv[]) {
   int *n = (int*) malloc(sizeof(int));
+  if (*n == NULL) {
+    perror("malloc");
+    return EXIT_FAILURE;
+  }
   *n = 42;
   printf("%d\n", *n);
   return EXIT_SUCCESS;
```

et committons le

```
$ git add main.c
$ git commit -m "Check malloc output"
[master 9e45e79] Check malloc output
 1 file changed, 4 insertions(+)
$ git log --stat --oneline
9e45e79 Check malloc output
 main.c | 4 ++++
 1 file changed, 4 insertions(+)
1712853 First commit
 .gitignore | 1 +
 Makefile   | 5 +++++
 main.c     | 9 +++++++++
 3 files changed, 15 insertions(+)
```

Essayons maintenant de construire un commit qui retire les lignes qu'on vient d'ajouter avec git-revert(1)

```
$ git revert 9e45e79
[master 6c0f33e] Revert "Check malloc output"
 1 file changed, 4 deletions(-)
$ git log --stat --oneline
6c0f33e Revert "Check malloc output"
 main.c | 4 ----
 1 file changed, 4 deletions(-)
9e45e79 Check malloc output
 main.c | 4 ++++
 1 file changed, 4 insertions(+)
1712853 First commit
 .gitignore | 1 +
 Makefile   | 5 +++++
 main.c     | 9 +++++++++
 3 files changed, 15 insertions(+)
```

Le contenu de `main.c` est alors

```c
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
  int *n = (int*) malloc(sizeof(int));
  *n = 42;
  printf("%d\n", *n);
  return EXIT_SUCCESS;
}
```

Comme c'est une bonne pratique de vérifier la valeur de retour de malloc(3), supprimons ce dernier commit

```
$ git reset HEAD^
Unstaged changes after reset:
M    main.c
$ git log --oneline
9e45e79 Check malloc output
1712853 First commit
```

13.6 Corriger des bugs grâce à Git

git(1) permet de garder des traces des nombreux changements qui ont été effectué au cours de l'évolution d'un programme. Il contient d'ailleurs un outil très puissant vous permettant de retrouver la source de certaines erreurs, pourvu que les changements soient faits par petits commits : git-bisect(1).

Supposez que vous ayez introduit une fonctionnalité dans votre programme. Tout allait alors pour le mieux. Quelques semaines plus tard, à votre grand dam, vous vous rendez compte qu'elle ne fonctionne plus. Vous sillonnez tous les fichiers qui pourraient interagir avec cette fonction, en vain. Dans le désespoir, à l'approche de la deadline, vous succombez au nihilisme.

Avant de tout abandonner, pourtant, vous réalisez quelque chose de très important. Ce que vous cherchez, c'est la source de l'erreur ; cela fait, la corriger sera sans l'ombre d'un doute une tâche aisée. Si seulement il était possible de voir à partir de quel changement le bug a été introduit...

C'est là que vous repensez à git(1) ! git(1) connaît tous les changements qui ont été effectués, et vous permet facilement de revenir dans le passé pour vérifier si le bug était présent à un moment donné. En outre, vous vous rappelez vos cours d'algorithmiques et vous rendez compte que, puisque vous connaissez un point où le bug était présent et un autre ou il ne l'était pas, vous pouvez à l'aide d'une recherche binaire déterminer en un temps logarithmique (par rapport aux nombres de révisions comprises dans l'intervalle) quelle révision a introduit l'erreur.

C'est exactement l'idée derrière *git-bisect(1)* : vous donnez un intervalle de commits dans lequel vous êtes certains de pouvoir trouver le vilain commit responsable de tous vos maux, pour ensuite le corriger. Vous pouvez même entièrement automatiser cette tâche si vous pouvez, excellent programmeur que vous êtes, écrire un script qui renvoie 1 si le bug est présent et 0 si tout va bien.

Pour vous montrez comme utiliser cette fonctionnalité, et vous convaincre que cela marche vraiment, et pas seulement dans des exemples fabriqués uniquement dans un but de démonstration, nous allons l'appliquer à un vrai programme C : `mruby`, une implémentation d'un langage correspondant à un sous-ensemble de Ruby.

Intéressons nous à un des problèmes qui a été rapporté par un utilisateur. Si vous lisez cette page, vous verrez qu'en plus de décrire le problème, il mentionne le commit à partir duquel il rencontre l'erreur. Si vous regardez aussi le commit qui l'a corrigée, vous verrez que le développeur a bien dû changer une ligne introduite dans le commit qui avait été accusé par l'utilisateur.

Mettons nous dans la peau de l'utilisateur qui a trouvé le bug, et tentons nous aussi d'en trouver la cause, en utilisant git(1) . D'abord, il nous faut obtenir le dépôt sur notre machine (vous aurez besoin de Ruby afin de pouvoir tester), et revenir dans le passé puisque, depuis, l'erreur a été corrigée.

```
$ git clone git@github.com:mruby/mruby.git
(...)
$ cd mruby
$ git checkout 5b51b119ca16fe42d63896da8395a5d05bfa9877~1
(...)
```

Sauvegardons aussi le fichier de test proposé, par exemple dans `~/code/rb/test.rb` :

```
class A
  def a
    b
  end
  def b
    c
```

(suite sur la page suivante)

(suite de la page précédente)

```
  end
  def c
    d
  end
end
x = A.new.a
```

Vous devriez maintenant être capable de vérifier que la méthode `A.a` n'est pas incluse dans la backtrace :

```
$ make && ./bin/mruby ~/code/rb/test.rb
(...)
trace:
        [3] /home/kilian/code/rb/test.rb:9:in A.c
        [2] /home/kilian/code/rb/test.rb:6:in A.b
        [0] /home/kilian/code/rb/test.rb:13
/home/kilian/code/rb/test.rb:9: undefined method 'd' for #<A:0xdf1000>␣
↪(NoMethodError)
```

C'est le moment de commencer. Il faut d'abord dire à git(1) que nous désirons démarrer une bissection et que le commit actuel est « mauvais », c'est à dire que le bug est présent. Ceci est fait en utilisant les deux lignes suivantes, dans l'ordre :

```
$ git bisect start
$ git bisect bad
```

Regardons ce qu'il en était quelque mois auparavant (remarquez qu'il faut utiliser `make clean` pour s'assurer de tout recompiler ici) :

```
$ git checkout 3a27e9189aba3336a563f1d29d95ab53a034a6f5
Previous HEAD position was 7ca2763... write_debug_record should dump info␣
↪recursively; close #1581
HEAD is now at 3a27e91... move (void) cast after declarations
$ make clean && make && ./bin/mruby ~/code/test.rb
(...)
trace:
        [3] /home/kilian/code/rb/test.rb:9:in A.c
        [2] /home/kilian/code/rb/test.rb:6:in A.b
        [1] /home/kilian/code/rb/test.rb:3:in A.a
        [0] /home/kilian/code/rb/test.rb:13
/home/kilian/code/rb/test.rb:9: undefined method 'd' for #<A:0x165d2c0>␣
↪(NoMethodError)
```

Cette fois-ci, tout va bien. Nous pouvons donc en informer git(1) :

```
$ git bisect good
Bisecting: 116 revisions left to test after this (roughly 7 steps)
[fe1f121640fbe94ad2e7fabf0b9cb8fdd4ae0e02] Merge pull request #1512 from␣
↪wasabiz/eliminate-mrb-intern
```

Ici, git(1) nous dit combien de révisions il reste à vérifier dans l'intervalle en plus de nous donner une estimation du nombre d'étapes que cela prendra. Il nous informe aussi de la révision vers laquelle il nous a déplacé. Nous pouvons donc réitérer notre test et en communiquer le résultat à git(1) :

```
$ make clean && make && ./bin/mruby ~/code/test.rb
(...)
trace:
        [3] /home/kilian/code/rb/test.rb:9:in A.c
        [2] /home/kilian/code/rb/test.rb:6:in A.b
        [1] /home/kilian/code/rb/test.rb:3:in A.a
        [0] /home/kilian/code/rb/test.rb:13
/home/kilian/code/rb/test.rb:9: undefined method 'd' for #<A:0x165d2c0>␣
↪(NoMethodError)
```

(suite sur la page suivante)

(suite de la page précédente)

```
$ git bisect good
Bisecting: 58 revisions left to test after this (roughly 6 steps)
[af03812877c914de787e70735eb89084434b21f1] add mrb_ary_modify(mrb,a); you␣
↪have to ensure mrb_value a to be an array; ref #1554
```

Si nous réessayons, nous allons nous rendre compte que notre teste échoue à présent (il manque la ligne `[1]`) : nous somme allés trop loin dans le futur. Il nous faudra donc dire à git(1) que la révision est mauvaise.

```
$ make clean && make && ./bin/mruby ~/code/test.rb
(...)
trace:
        [3] /home/kilian/code/rb/test.rb:9:in A.c
        [2] /home/kilian/code/rb/test.rb:6:in A.b
        [0] /home/kilian/code/rb/test.rb:13
/home/kilian/code/rb/test.rb:9: undefined method 'd' for #<A:0x165d2c0>␣
↪(NoMethodError)
$ git bisect bad
Bisecting: 28 revisions left to test after this (roughly 5 steps)
[9b2f4c4423ed11f12d6393ae1f0dd4fe3e51ffa0] move declarations to the␣
↪beginning of blocks
```

Si vous continuez à appliquer cette procédure, vous allez finir par trouver la révision fautive, et git(1) nous donnera l'information que nous recherchions, comme par magie :

```
$ git bisect bad
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[a7c9a71684fccf8121f16803f8e3d758f0dea001] better error position display
$ make clean && make && ./bin/mruby ~/code/rb/test.rb
(...)
trace:
        [3] /home/kilian/code/rb/test.rb:9:in A.c
        [2] /home/kilian/code/rb/test.rb:6:in A.b
        [0] /home/kilian/code/rb/test.rb:13
/home/kilian/code/rb/test.rb:9: undefined method 'd' for #<A:0x1088160>␣
↪(NoMethodError)
$ git bisect bad
a7c9a71684fccf8121f16803f8e3d758f0dea001 is the first bad commit
commit a7c9a71684fccf8121f16803f8e3d758f0dea001
Author: Yukihiro "Matz" Matsumoto <matz@ruby-lang.org>
Date:   Tue Oct 15 12:49:41 2013 +0900

    better error position display

:040000 040000 67b00e2d4f6acadc0474e00fc0f5e6e13673c64a␣
↪036eb9c3b9960613bde3882b7a88ac6cabc56253 M      include
:040000 040000 5040dd346fea4d8f476d26ad2ede0dc49ca368cd␣
↪903f2d954d8686e7bfa7bcf5d83b80b5beb4899f M      src
```

Maintenant que nous connaissons la source du problème, il ne faut pas oublier de confirmer à git(1) que la recherche est bien terminée, et que nous désirons remettre le dépôt dans son état normal.

```
$ git bisect reset
Previous HEAD position was a7c9a71... better error position display
HEAD is now at 7ca2763... write_debug_record should dump info
recursively; close #1581
```

13.6.1 Automatisation de la procédure

Exécuter ce test à la main est cependant répétitif, prône aux erreurs d'inattention, et surtout très facile à automatiser. Écrivons donc un script qui vérifie que la ligne mentionnant `A.a` est bien présente à chaque fois, appelons le par exemple `~/code/sh/Iznogoud.sh`. Il s'agit de renvoyer 0 si tout se passe bien et une autre valeur s'il y a un problème.

```
#!/usr/bin/env bash
make clean && make && ./bin/mruby ~/code/rb/test.rb 2>&1 | grep A\.a
```

Puisque `grep` renvoie 1 quand il ne trouve pas de ligne contenant le motif qu'on lui passe en argument et 0 sinon, notre script renvoie bien 1 si la sortie de mruby ne contient pas la ligne mentionnant `A.a` et 0 sinon.

N'oubliez pas de changer les permissions du script pour en permettre l'exécution :

```
$ chmod +x ~/code/sh/Iznogoud.sh
```

Ce test n'est en bien sûr pas infaillible, mais sera suffisant ici. Il faut d'abord redonner à git(1) l'intervalle dans lequel se trouve la révision fautive.

```
$ git bisect start
$ git bisect bad
$ git checkout 3a27e9189aba3336a563f1d29d95ab53a034a6f5
Previous HEAD position was 7ca2763... write_debug_record should dump info␣
↪recursively; close #1581
HEAD is now at 3a27e91... move (void) cast after declarations
$ git bisect good
Bisecting: 116 revisions left to test after this (roughly 7 steps)
[fe1f121640fbe94ad2e7fabf0b9cb8fdd4ae0e02] Merge pull request #1512 from␣
↪wasabiz/eliminate-mrb-intern
```

Il suffit maintenant d'utiliser `git bisect run` avec le nom du script pour l'utiliser. Il est possible de rajouter d'autres arguments après le nom du script, qui seront passés au script lors de chaque exécution. Par exemple, si vous avez dans votre `Makefile` une tâche test qui renvoie 0 si tous les tests passent et 1 si certains échouent, alors `git bisect run make test` permettrait de trouver à partir de quand les tests ont cessé de fonctionner.

Si vous exécutez la ligne suivante, vous devriez bien trouver, après quelques compilations, le même résultat qu'avant :

```
$ git bisect run ~/code/sh/Iznogoud.sh
(...)
a7c9a71684fccf8121f16803f8e3d758f0dea001 is the first bad commit
commit a7c9a71684fccf8121f16803f8e3d758f0dea001
Author: Yukihiro "Matz" Matsumoto <matz@ruby-lang.org>
Date:   Tue Oct 15 12:49:41 2013 +0900

    better error position display

:040000 040000 67b00e2d4f6acadc0474e00fc0f5e6e13673c64a␣
↪036eb9c3b9960613bde3882b7a88ac6cabc56253 M      include
:040000 040000 5040dd346fea4d8f476d26ad2ede0dc49ca368cd␣
↪903f2d954d8686e7bfa7bcf5d83b80b5beb4899f M      src
bisect run success
```

À nouveau, n'oubliez pas d'utiliser `git bisect reset` avant de continuer à travailler sur le dépôt.

CHAPITRE 14

SSH

ssh(1) (Secure Shell) est un outil qui permet de se connecter depuis l'Internet à la console d'une autre machine et donc d'y exécuter des commandes. Dans l'infrastructure INGI vous pouvez vous connecter via ssh aux différents machines des salles en utilisant votre login et mot de passe INGI. Pour savoir les noms de machines, visitez le student-wiki.

Depuis l'extérieur vous devez passer via `studssh.info.ucl.ac.be` pour ensuite pouvoir vous connecter sur les machines des salles.

Quelques exemples d'utilisation de ssh(1) qui vous seront utiles :

- `ssh [username]@[hostname]` : Avec ceci vous pouvez vous connecter à la machine `hostname`. Exemple : `ssh myUserName@yunaska.info.ucl.ac.be` pour vous connecter à la machine `yunaska` de la salle intel. Il faut d'abord se connecter à sirius avant de se connecter aux machines des salles.
- `ssh -X [username]@[hostname]` : L'option `-X` vous permet d'exécuter des programmes sur la machine distante mais en voyant l'interface graphique en local sur votre machine (pour autant qu'elle supporte X11). Exemple : `ssh -X myUserName@yunaska.info.ucl.ac.be` et ensuite dans le terminal `gedit test.c` pour ouvrir l'éditeur de texte.
- `ssh [username]@[hostname] [commande]` : Pour exécuter la commande sur la machine distante. Exemple : `ssh myUserName@sirius.info.ucl.ac.be cc test.c -o test` pour exécuter `cc test.c -o test` sur sirius.
- `scp [local_file] [username]@[hostname]:[path]` : scp(1) permet de copier des fichiers locaux vers un répertoire distant (et l'inverse). Exemple : `scp test.c myUserName@sirius.info.ucl.ac.be:INFO1252/projet_S2/` copie test.c vers le dossier `INFO1252/projet_S2/` de la machine sirius.

Le site Getting started with SSH contient une bonne description de l'utilisation de ssh. Notamment l'utilisation de ssh sur des machines UNIX/Linux. Si vous utilisez Windows, il existe des clients ssh(1) comme putty

14.1 Authentification par clé

Face à la faiblesse au niveau sécurité de l'authentification par mot de passe, l'authentification par clé se révèle être un moyen nettement plus efficace.

L'authentification par clé consiste en un premier temps à générer une paire de clés et son mot de passe :

- La *clé publique* que l'on exporte vers chaque hôte sur lequel on veut se connecter.
- La *clé privée* que l'on garde précieusement sur notre ordinateur, et qui sert à prouver à l'hôte notre identité. **Attention : ne jamais dévoiler sa clé privée !**
- Le *mot de passe* permet de sécuriser sa clé privée.

Le mot de passe ne servant à rien sans les clé et vice versa, on devine aisément que le niveau de sécurité d'une telle connexion est largement accru par rapport à une simple authentification par mot de passe.

Pour générer ces clés et choisir votre mot de passe, il suffit d'entrer la commande

```
$ ssh-keygen -t rsa -C "login"
# remplacer "login" par votre nom d'utilisateur

Generating public/private rsa key pair.
Enter file in which to save the key (~/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in ~/.ssh/id_rsa.
Your public key has been saved in ~/.ssh/id_rsa.pub.
The key fingerprint is:
17:bc:98:ab:39:f6:a2:db:1d:07:9a:63:d7:c7:9b:e0 "login"
```

Maintenant que les clés ont été crées, il faut communiquer votre clé publique à l'hôte sur lequel vous voulez vous connecter. Pour cela, le plus simple est d'utiliser la commande suivante

```
$ cat ~/.ssh/id_rsa.pub | ssh [username]@[hostname] "cat - >> ~/.ssh/
↪authorized_keys"
```

Cette commande va récupérer votre clé publique dans votre dossier `~/.ssh`, se connecter en ssh à l'hôte et va placer votre clé dans son répertoire de clés autorisées. Maintenant nous pouvons utilisez en toute sérénité votre connexion ssh sécurisée !

Petit mot sur les permissions du dossier `~/.ssh` où sont stockées les clés :

```
.ssh user$ ls -ld
drwx------  6 user  staff  204 22 aoû 10:29 .
```

Les bits de permissions sont définis comme `drwx------` ce qui fait du propriétaire de ce dossier la seul personne capable de lire, d'écrire et d'exécuter le contenu de se dossier. La clé privée est donc belle et bien privée !

14.2 Utiliser Git avec ssh

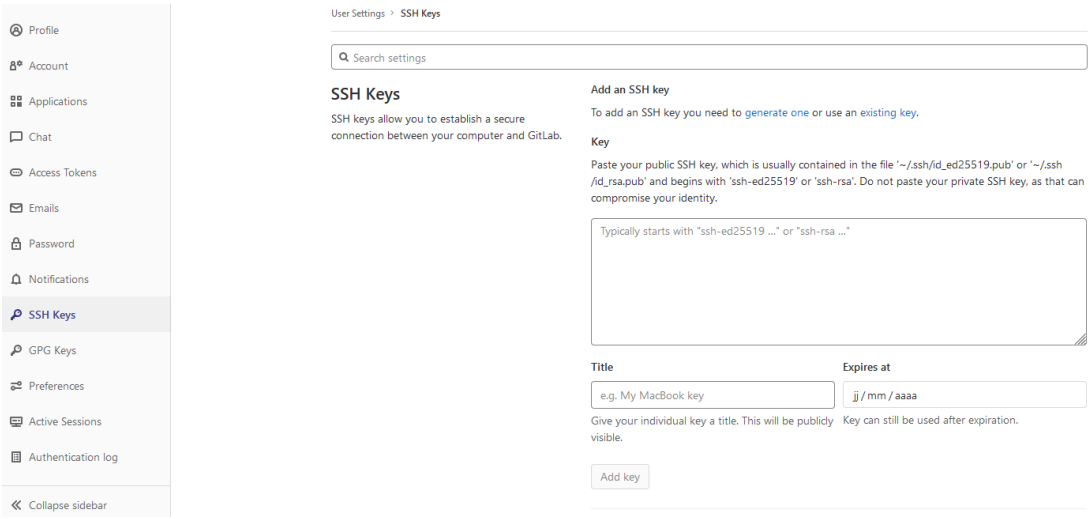
Il est également possible de s'authentifier auprès de Git en utilisant une clé ssh à la place de la traditionnelle combinaison nom d'utilisateur et mot de passe. L'avantage de ceci est qu'il n'est plus nécessaire de fournir ses identifiants à chaque push ou pull. L'identité est directement vérifiée depuis les clés ssh présentes sur la machine.

Pour activer l'identification avec ssh, il faut fournir sa clé publique sur la plateforme en ligne utilisée avec Git, par exemple GitHub ou GitLab. Pour ce faire, il suffit de suivre les étapes suivantes :

- Ouvrir les paramètres du profil.
- **Ouvrir la page de gestion des clés SSH.** Sur GitLab, il s'agit de `SSH Keys`. Sur GitHub, il s'agit de `SSH and GPG keys`.
- **Copier la clé publique à l'endroit indiqué. Attention à ne pas copier la clé privée !** La clé publique se trouve généralement dans le fichier `~/.ssh/id_rsa.pub`. Il faut copier tout le contenu de ce fichier pour que la clé soit correcte.
- Donner un titre à la clé, qui permet d'identifier la clé parmi plusieurs sur la plateforme.
- Finaliser en ajoutant la clé.

A titre d'exemple, voici la page permettant d'ajouter une clé SSH sur GitLab :

Une fois que la clé publique est ajoutée sur son profil, il est possible d'utiliser ssh pour s'identifier auprès de Git. Pour ce faire, lorsqu'un projet est cloné depuis la plateforme (GitHub ou GitLab), il faut choisir le lien `Clone with SSH`. De cette manière, c'est la clé ssh qui sera utilisée pour s'identifier, et il ne faudra plus indiquer le nom d'utilisateur et le mot de passe à chaque opération.



14.3 Synchronisation de fichiers entre ordinateurs

Quand nous avons besoin de synchroniser des fichiers entre 2 ordinateurs différents, Unix nous vient en aide avec l'utilitaire rsync.

L'utilisation la plus basique de rsync est :

```
rsync *.c [hostname]:src/
```

rsync va copier tout les fichiers qui correspondent au pattern `*.c` du répertoire courant vers le dossier `src/` sur la machine hôte. De plus, si certains ou tout les fichiers sont déjà présents chez l'hôte, rsync va procéder à une mise à jour différentielle de ces fichiers (seuls les changements sont transférés).

L'ajout du drapeau `-avz` permet de synchroniser les fichiers en mode archive. Cela veut dire que tous les liens, permissions, propriétaires, etc de ces fichiers seront préservés durant le transfert.

Nous pouvons aussi utiliser rsync dans l'autre sens :

```
rsync -avz [hostname]:src/bar /data/tmp
```

Maintenant tous les fichiers de la machine hôte, dans le dossier `src/bar` vont être copiés vers le répertoire local `/data/tmp`.

CHAPITRE 15

Jenkins

Jenkins est un outil d'intégration continue pour un projet informatique. L'**intégration continue** est le fait de, lors de l'évolution d'un projet informatique, vérifier que les nouvelles modifications s'intègrent bien au programme existant, et n'introduisent pas de bug ou ne supprime pas de fonctionnalité.

Pour ce faire, Jenkins permet d'enregistrer des commandes (généralement des commandes terminal) qui seront automatiquement exécutées lorsqu'une modification est apportée au projet. En pratique, Jenkins peut être lié à un projet GitLab, et donc exécuter les commandes à chaque commit. Les commandes en question peuvent être de différents types :

- Lancer une suite de tests unitaires CUnit.
- Utiliser valgrind(1) pour détecter les fuites de mémoire (voir la section dédiée pour plus d'informations).
- Utiliser cppcheck pour analyser le code et détecter de potentiels bugs.

La suite de ce document détaille comment inclure Jenkins à un projet GitLab, pour profiter de l'automatisation des tests à chaque commit, et éviter les bugs et la perte de fonctionnalité.

15.1 Création du projet sur Jenkins

- Se connecter sur https://jenkins.student.info.ucl.ac.be et cliquer sur le bouton *S'identifier*.
- S'identifier à l'aide du compte UCLouvain.
- Une fois connecté, dans le menu à gauche, cliquer sur *Nouveau Item*.
- Saisir le nom du projet, choisir *Construire un projet free-style*, et cliquer sur *OK*.
- Le menu de configuration s'ouvre. Cocher la case *Activer la sécurité basée projet*.
- Ajouter les membres du projet a l'aide de leur Jenkins User ID, visible sur leur profil Jenkins.
- Donner tous les droits aux membres du groupe (cocher toutes les cases). Attention à ne pas donner tous les droits aux anonymes !
- Dans le menu déroulant *GitLab Connection*, choisir *Forge UCLouvain*.
- Dans *Gestion de code source*, choisir l'option *Git*.
- Récupérer l'URL du dépôt git sur GitLab, avec l'option *Clone with SSH*.
- Indiquer l'URL du dépôt git récupérée à l'étape précédente, et choisir *git (SSH key - jenkins_ingi - forge.uclouvain.be)* comme identifiant.
- Dans *Ce qui déclenche le build*, cocher *Build when a change is pushed to GitLab*. **Noter quelque part le web hook URL précisé juste à côté**.
- Un cadre s'ouvre, cliquer sur le bouton *Avancé*. En dessous du champ *Secret token* se trouve un bouton *Generate*. **Générer un token et le noter aussi**.
- Dans *Build*, ajouter une étape *Exécuter un script shell*. Y introduire :

```
#!/bin/bash

exit 0
```

— Dans *Actions à la suite du build*, ajouter une action *Publish build status to GitLab*.
— Cliquer sur *Sauver* pour construire le projet.

15.2 Déclenchement automatique des builds

— Sur GitLab, dans la vue du projet, aller dans le menu *Settings* => *Webhooks*
— Copier le *web hook URL* ainsi que le *Secret token* mis de côté au point précédent. Cocher *Push events* et finalement cliquer sur *Add webhook*.

Au prochain commit, un build sera déclenché automatiquement. Comme le script shell retourne 0, le build passera tout le temps. Si le lien est fonctionnel, une pastille verte s'affichera à côté de la description du commit dans GitLab.

15.3 Modifier le script sur Jenkins

— Sur Jenkins, dans la vue du projet cliquer sur le menu *Configurer*
— Aller jusqu'à la section *Build* et modifier le script shell
— Rajouter les commandes nécessaires à l'exécution des tests de votre projet (`make tests` par exemple)

Si le code ne passe pas tous les tests, la valeur de retour de la commande sera différente de 0 et le build sera marqué comme *Failed*, une pastille rouge apparaîtra dans GitLab.

15.4 Projet d'exemple pour Jenkins

Nous vous fournissons un projet donnant un exemple plus complet de configuration d'un projet GitLab avec Jenkins. Il est disponible à l'adresse suivante : https://forge.uclouvain.be/alegay/jenkinslepl1503/. Le projet contient les fichiers suivants :

— `ex-lepl1503.c` : un fichier C qui contient :
  — `int maxi(int, int)` : une fonction qui calcule le maximum entre deux entiers
  — `void test_maxi(void)` : un test correct (oui le maximum entre 0 et 2 est bien 2)
  — `void test_maxifailed(void)` : un test incorrect (non le maximum entre 0 et 2 n'est pas 0 !)
  — `void erreurmalloc(void)` : une fonction qui fait une assignation à une case mémoire non allouée
  — Une procédure main qui :
    — définit une suite de tests CUnit basée sur `test_maxi` et `test_maxifailed`. Pour plus de détails sur CUnit, reportez-vous à la section dédiée.
    — appelle la fonction `erreurmalloc`.
— Un fichier `Makefile` qui comprend plusieurs règles. Pour plus de détails sur la conception du makefile, reportez-vous à la section dédiée.

Voici quelques exemples d'utilisation du projet :

— `make` : compile le programme avec l'option -lcunit et appelle les outils *valgrind* et *cppcheck*. Les résultats sont sauvés dans des fichiers xml.
— `make clean` : efface l'exécutable et les .xml.

Ne pas hésiter à faire un clone de ce projet sur votre machine, et jouer avec la commande `make` pour bien comprendre comment le programme et ses différents outils fonctionnent. Comme vous êtes un·e utilisateur·rice *guest*, vous ne pourrez pas faire de commit dans ce projet. **Attention** : si vous n'avez pas installé CUnit, valgrind ou cppcheck sur votre machine, vous ne pourrez pas compiler le programme ou effectuer les tests.

Vous pouvez voir sur la page principale du projet sur GitLab une croix rouge à côté du dernier commit effectué. Cela vient du projet Jenkins lié à ce GitLab, et montre que ce commit n'a pas réussi tous les tests. Ce projet Jenkins est disponible à l'adresse suivante : https://jenkins.student.info.ucl.ac.be/job/ex-lepl1503/. **Attention** : si vous n'êtes pas connecté·e sur Jenkins, vous arriverez sur une page avec une erreur 404. Dans ce cas, retournez sur la page d'accueil de Jenkins (https://jenkins.student.info.ucl.ac.be), connectez-vous et réessayez.

Lors de votre connexion, vous verrez qu'un certain nombre de commits ont été faits. Certains avec succès (les bleus) d'autres pas (les rouges). Vous pourrez aussi observer les résultats de CUnit, valgrind et cppcheck. Rendez-vous dans le menu *Configurer*. Vous y trouverez les actions faites avant et après le build : *Ce qui déclenche le build*, *Build* et *Actions à la suite du build*. Vous verrez par exemple dans *Actions à la suite du build*, l'utilisation de plugins pour afficher les rapports XML générés par `make`.

Notez que vous n'avez pas le droit de modifier cette configuration. Cela se vérifie dans l'onglet *Activer la sécurité basée projet* du menu *configure* où seul l'utilisateur Axel Legay a tous les droits. Lorsque vous lierez votre Jenkins à votre projet GitLab, pensez à donner tous les droits à tous les utilisateur·rices mainteneur·ses de votre projet en utilisant *Add user or group*.

CHAPITRE 16

Profiling de code

Lorsque l'on cherche à optimiser les performances de programmes écrits en C, il est utile de les exécuter à travers un profiler tel que gprof(1). Ce profiler permet de collecter des statistiques sur les fonctions les plus utilisées à l'intérieur du programme. Ce sont les fonctions qu'il faudra optimiser pour améliorer les performances. De nombreux articles décrivent les principes de base de l'utilisation de gprof(1). Un tutoriel est disponible à l'adresse https://www.thegeekstuff.com/2012/08/gprof-tutorial/. oprofile est un profiler plus puissant que gprof(1) mais malheureusement plus difficile à utiliser.

CHAPITRE 17

Bibliographie

Bibliographie

[DeveloppezMake] Introduction à Makefile, https://gl.developpez.com/tutoriel/outil/makefile/
[GNUMake] The GNU Make Manual, https://www.gnu.org/software/make/manual/make.html