Système et réseaux (SR1)

Gestion des utilisateurs

Il existe un utilisateur privilégié (de nom `root` et de groupe `root`) qui dispose de droits étendus sur le système de fichier et sur le contrôle de l'activité de la machine. Cet utilisateur existe dès l'installation du système et doit créer les autres utilisateurs et éventuellement de nouveaux groupes[1] (commandes *adduser* et *addgroup* dont la documentation est disponible dans le `man` de niveau 8, ainsi que *chfn*, *chage*, *chsh*). Le répertoire `/etc/skel` peut être utilisé comme base de construction des répertoires utilisateurs.
La gestion standard des utilisateurs repose essentiellement sur quatre fichiers. Les deux premiers décrivent les groupes et utilisateurs déclarés sur le système (`/etc/group` et `/etc/passwd`). Les deux autres fichiers contiennent la version codée des mots de passe (`/etc/shadow` et `/etc/gshadow`). Les droits sur ces fichiers sont :

```
-rw-r--r-- 1 root root 1317  8 févr.  2013 /etc/passwd
-rw-r----- 1 root shadow 990 11 févr.  2013 /etc/shadow
-rw-r--r-- 1 root root 846  8 févr.  2013 /etc/group
-rw-r----- 1 root shadow 722  8 févr.  2013 /etc/gshadow
```

Les fichiers `passwd` et `group` définissent les identifiants d'utilisateur et groupe, appelé `uid` et `gid`. Le mot de passe `x` dans `/etc/group` et `/etc/passwd` renvoie aux fichiers « shadow » correspondants. Le mot de passe * ou ! indique qu'il est impossible de se connecter au sens habituel sur le compte (c'est le cas de certains comptes utilisés par des protocoles). Le paramétrage des logins est accessible dans `/etc/login.defs`. Les structures de tous ces fichiers sont décrites dans le `man` (niveau 5 entrées `group`, `passwd`, `shadow`, `gshadow`, `login.defs`). Des extraits des quatre premiers fichiers sont présentés ci-dessous :

```
root:x:0:
users:x:100:mnt,mc,pc

root:x:0:0:root:/root:/bin/bash
mnt:x:1001:100:Marie-Noelle,,,,:/home/mnt:/bin/bash
mc:x:1000:100:Mickael,,,,:/home/mc:/bin/sh

root:$6$XG5GWkYV$UIjJi$XG5G6QYG/NybjOHfH105VAU3kUWkYSI:15744:0:99999:7:::
mnt:FGyXQZ68j7L3y91UwB$ybE.0vBO8VmR/4Zg/HFGQNQsKAMaCI3:15747:0:99999:7:::
mc:FGjJi$XG5Gy91UwB$ybE.V$UIjJi$XNQkUvBOH1058VmR/4WkYS:15745:0:99999:7:::

root:*::
users:*::
```

Un utilisateur peut effectuer un login sur une machine distante (*ssh* ou *sftp*), changer son mot de passe (*passwd*), changer de groupe pour une ou plusieurs commandes (*sg* ou *newgrp*), devenir un autre utilisateur (*su*).

Pour changer de mot de passe dans les salles de tp :
smbpasswd -r ionesco3

De nombreuses fonctions et structures C permettent de récupérer les informations sur les utilisateurs et les groupes *(struct group*, *struct passwd*, *getpwnam()* et *getpwuid()* sur /etc/passwd, *getgrnam()* et *getgrgid()* sur /etc/group, *getspnam()* et *getspent()* sur /etc/shadow, etc.)
Les fichiers `/var/run/utmp` et `/var/log/wtmp` conservent un historique des logins (avec en C *struct utmpx* et *getutxent ()*). Le fichier `/var/log/lastlog` indique la date de dernier login (commande *lastlog* en shell, *struct lastlog* en C).

La fonction *getlogin()* permet de récupérer le nom de login associé au processus en cours. La fonction *crypt()* permet de crypter un mot de passe, la fonction *getpass()* permet de lire au clavier sans affichage pour la saisie des mots de passe.

Les processus

Introduction

Chaque exécution de commande dans un terminal met en œuvre les éléments suivants :

- une commande est susceptible de recevoir des paramètres et des données, d'afficher des résultats,
- une commande est par défaut associée à trois voies d'entrée-sortie : une entrée appelée `stdin` (par défaut, le clavier) ainsi qu'une sortie appelée `stdout` (par défaut, l'écran) et une sortie erreur appelée `stderr` (par défaut, l'écran). Ces trois voies sont associées aux trois premiers descripteurs disponibles (0, 1 et 2). Il est possible de modifier les voies d'entré-sortie :
  - `commande > fichier` envoie les résultats dans le fichier indiqué et non sur `stdout`
  - `commande 2> fichier` envoie les messages d'erreur dans le fichier indiqué et non sur `stderr`
  - `commande < fichier` effectue toutes les lectures dans le fichier indiqué et non sur `stdin` ;
- certaines commandes, appelées **filtres**, prennent leurs données au clavier et affichent leurs résultats sur écran (par exemple *tr*, *more*, *grep*, *head*, *tail*, *cut*, *cpio*, etc.)
- il est possible d'enchaîner plusieurs commandes dont la première affiche sur écran ses résultats et la suivante lit ses données au clavier (symbole `pipe` | dans un terminal, fonction *popen()* en C).

1 *Il est à noter que dans les salles de TP, la gestion des utilisateurs est différente (utilisation d'un descriptif global des utilisateurs).*

Par exemple, le filtre `cpio` option -o prend sur *stdin* une liste de noms ou chemins d'accès à des fichiers et construit sur *stdout* une archive contenant tous les fichiers cités. Avec option -i l'archive est lue sur *stdin* et les fichiers sont extraits sur *stdout*. Parmi les autres options possibles :

- pour conserver les noms de fichiers : option -v
- pour créer si nécessaire des répertoires : option -d

Pour créer une archive contenant les fichiers au point courant : `ls | cpio -ov > archive.cpio`
Pour créer une archive de l'arborescence au point courant : `find . | cpio -ov > archive.cpio`
Pour extraire au point courant le contenu d'une archive : `cpio -idv < archive.cpio`

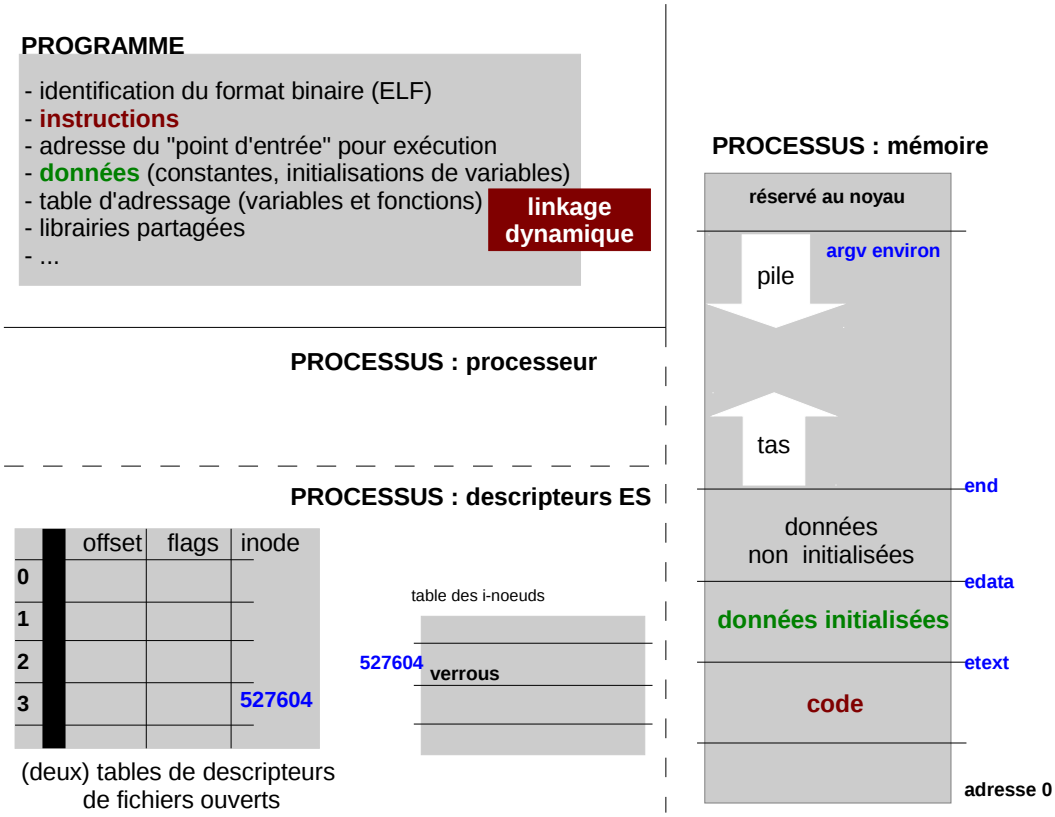
Vocabulaire :

- descripteur de fichier, verrouillage de fichier
- code exécutable, bibliothèque (statique ou dynamique), interprétation, compilation, édition de liens (dynamique)
- mécanismes de virtualisation : pagination, swap, temps partagé (round robin, quota)
- programme, processeur, processus
- mécanisme de gestion des processus : allocation de ressources, préemption, priorité, modes noyau et utilisateur

D'après Kerrisk page 114 : « *a process is an abstract entity, defined by the kernel, to which system resources are allocated in order to execute a program* ».
L'exécution d'un programme est contrôlée par un autre processus (par exemple, le shell dans lequel on en effectue le lancement), elle se fait pas à pas (instruction par instruction) avec des appels aux primitives d'autres ressources que le processeur et la zone mémoire qui lui est allouée (par exemple *open* et *close* pour les entrées-sorties). Les ressources externes doivent être disponibles/accessibles pour l'usage qui va en être fait. D'où l'importance :

- de la hiérarchie des processus,
- de l'environnement d'un processus,
- des propriétaires d'un processus,
- du cycle de vie d'un processus.

Les relations entre programme et processus sont illustrées sur le schéma ci-dessous.



Hiérarchie des processus
Chaque processus a un père dont il reçoit des informations (variables d'environnement, descripteurs de fichiers ouverts, etc.) et auquel il renvoie, en fin d'exécution, un **code retour** (entier). Les processus sont identifiés par un numéro (**pid**) et disposent du numéro de leur processus père (`ppid`). Le processus qui contrôle toute l'activité d'une machine Unix (du boot à l'arrêt de la machine) est le processus `init` dont le pid est 1. Le processus *init* va en particulier lancer des processus de gestion de

terminaux (*getty*) qui lancent des procédures de connexion (*login*) lesquelles lancent des shells (ceux indiqués dans le fichier `/etc/passwd`). Les shells servent de pères aux processus que va lancer l'utilisateur connecté. Voir la figure, très simplifiée, ci-dessous :

Vous pouvez voir tous les processus en cours sur une machine et leur organisation hiérarchique avec les commandes *top*, *pstree* ou *ps -el*.

Chaque processus est supposé attendre la fin de ses processus fils avant de s'arrêter afin de récupérer leurs codes retour (primitive *wait()* en C). Un processus dont le père se termine sans avoir récupéré son code retour devient fils du processus *init* (son ppid devient 1). Un fils qui se termine attend la récupération de son code retour par son père sous la forme de processus **zombie**.

Description des processus

L'**environnement** d'un processus est constitué de variables conservées sous la forme de chaîne de caractères au format `nom=valeur`. Les variables d'environnement sont initialisées à partir de celles du processus père. Vous pouvez ajouter des variables d'environnement à un processus. Elles sont, par défaut, locales au processus mais vous pouvez indiquer, par la commande *export* en shell, qu'une variable doit être connue des fils de ce processus.
En C, les primitives *getenv()*, *putenv()*, *setenv()*, *unsetenv()* permettent de travailler sur les variables d'environnement. La variable *envp* du *main()* en C contient les variables d'environnement du processus qui exécute le programme. Le format est le même que celui de *args* (tableau de chaînes de caractères).

Chaque processus est décrit par un répertoire dans `/proc`. Ce répertoire a pour nom le pid du processus. Il contient de nombreuses entrées que vous ne pouvez pas ouvrir mais, par exemple : le fichier `cmdline` contient la ligne de commande qui a permis de lancer l'exécution, le fichier `environ` contient les variables d'environnement, le répertoire `fd` contient un fichier pour chaque descripteur de fichier ouvert :

```
cat /proc/9324/cmdline
       /usr/bin/gnome-keyring-daemon--daemonize--login
cat /proc/9324/environ
       LANG=fr_FR.UTF-8
       HOME=/home/mnt
       DISPLAY=:0
ls -l /proc/9324/fd
       lr-x------ 1 mnt mnt 64 15 sept. 15:55 0 -> /dev/null
       lr-x------ 1 mnt mnt 64 15 sept. 15:55 1 -> /dev/null
       lrwx------ 1 mnt mnt 64 15 sept. 15:55 11 -> socket:[19864192]
       lr-x------ 1 mnt mnt 64 15 sept. 15:55 2 -> /dev/null
       lr-x------ 1 mnt mnt 64 15 sept. 15:55 3 -> pipe:[19863402]
```

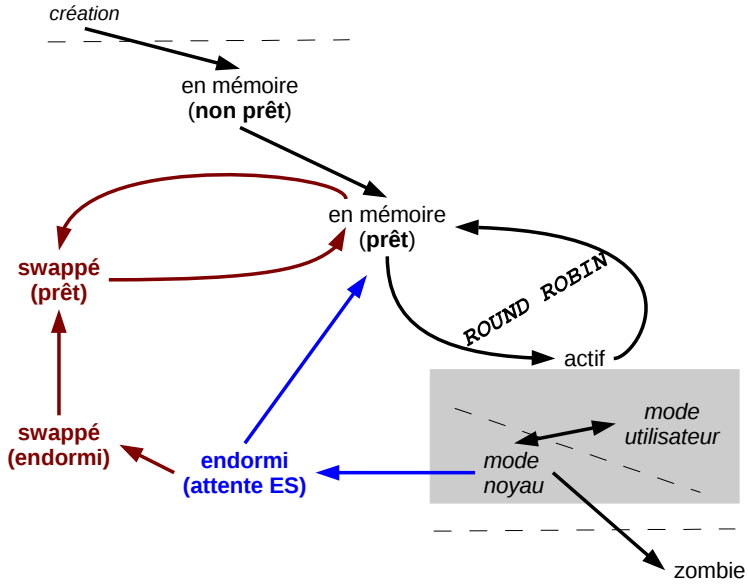
A chaque processus sont associés plusieurs **propriétaires** (et groupes propriétaires) dont :

- le **propriétaire réel** : celui qui possède le processus. Pour tout programme lancé depuis un shell, il s'agit de celui qui a lancé la commande d'exécution, donc l'utilisateur dont le login a été utilisé pour ouvrir le shell ;
- le **propriétaire effectif** (sous Unix) ou fileSystem (sous Linux) : celui dont les droits sont utilisés pendant l'exécution du programme (ils dépendent des bits uid et gid positionnés).

Les accès effectués par un processus sont contrôlés en fonction de ses propriétaire et groupe effectifs. De nombreuses fonctions C permettent de récupérer les propriétaires (*getuid()*, *getgid()*, *getreuid()*, *getregid()*, etc.) et éventuellement de les modifier.

Cycle de vie des processus

Lorsqu'un processus s'exécute, il entre -comme candidat/demandeur- dans le système d'allocation des ressources de la machine.
Les ressources qu'il demande lui sont allouées de la moins coûteuse à la plus coûteuse (de celles qui sont faciles à partager à celles qui ne sont pas partageables). Parmi les ressources demandées, le processeur est toujours alloué en dernier. Le processeur peut être alloué en mode noyau (pour tous les appels sytèmes, dont les entrées-sorties) ou en mode utilisateur. Un processus actif passe de l'un à l'autre de ces modes d'exécution.
La mémoire centrale peut rester allouée à un processus en attente mais -si nécessaire- ce processus peut être **swappé** (sa zone mémoire est temporairement recopié sur disque). Les états d'un processus (non prêt, prêt, actif, zombie) et les



transitions entre ces états en fonction des mécanismes de round robin et de swap sont schématisés ci-dessus.

La création d'un processus est faite par son processus père. Deux primitives sont disponibles à cet effet :

- *fork()* permet de créer un nouveau processus qui est -à son lancement- un clone du processus père (dans l'état où celui-ci se trouvait lorsque le fork a été exécuté). En particulier le processus fils hérite de tous les descripteurs de fichiers ouverts par le processus père avant le fork ;
- *exec()* permet de remplacer un processus par un autre en gardant toutes les propriétés du processus initial (dont son pid). Il existe en fait toute une famille de primitives exec utilisables en C (*execve(), execle(), execlp(), execvp(), execv(), execl()*). Ces primitives exec diffère sur trois points :
  - l'appel doit être fait en indiquant le chemin d'accès complet au code exécutable ou la variable PATH est utilisée,
  - le nombre d'arguments est connu a priori ou pas,
  - l'environnement est obligatoirement celui du processus père ou bien il peut être complété.

Pour créer un nouveau processus, il est donc fréquent de faire un appel à *fork()* pour créer un processus fils puis d'un appel à *exec()* pour donner au processus fils ainsi créé un code exécutable qui lui est propre.

Ces deux primitives sont en général utilisées en C mais vous pouvez aussi utiliser en shell :

- le lancement en arrière-plan d'un processus avec la syntaxe `commande &`. Le processus créé devient un concurrent du shell qui l'a lancé. Il n'est plus associé à l'écran-clavier du processus shell. Le shell peut récupérer son pid dans la variable `$!`
  Il existe des processus qui tournent en arrière-plan pour effectuer certaines tâches système. Ils sont appelés des démons ;
- la commande *exec* qui permet de remplacer le processus en cours. Les deux principales utilisations de cette commande consistent à remplacer un shell par un autre (par exemple `exec ksh`) ou à rediriger une voie d'entré-sortie, souvent `stdin`, pendant l'exécution du programme (par exemple `exec < fichier`) ;
- la commande *nohup* qui permet d'exécuter un programme en le rendant « insensible » à la déconnexion de l'utilisateur. Elle sera souvent combinée avec le lancement en arrière-plan (commande `nohup programme &`). Les sorties `stdout` et `stderr` sont redirigées dans un fichier `nohup.out` qui est créé au point de lancement de nohup. L'entrée `stdin` est redirigée sur `/dev/null ;`
- la commande *sleep* permet d'endormir un processus pendant un temps donné (en seconde, minute, heure, jour, …) ;
- la commande *exit* permet de terminer immédiatement un processus. Cette commande peut avoir un unique paramètre qui est la valeur entière à renvoyer comme code retour du processus. Par défaut l'exécution d'un programme shell se terminera lorsque la fin du fichier sera atteinte.

Les processus et leurs entrées-sorties

Lorsque plusieurs processus interagissent en utilisant des fichiers communs, plusieurs problèmes doivent être traités dont le passage en mode non bloquant des entrées-sorties et l'interdiction temporaire d'accéder à un fichier.

Passage des entrées-sorties en mode non bloquant Un processus qui demande une entrée-sortie va attendre que cette entrée-sortie ait eu lieu. On parle d'**entrée bloquante**. Il y a un risque de voir l'attente d'une information à lire se prolonger; le processus restant bloqué sans pouvoir rien faire. Lorsque plusieurs processus interagissent, un même processus peut attendre des informations sur plusieurs voies d'entrée sans savoir dans quel ordre ces informations vont arriver.

> Par exemple, dans le projet de l'an dernier, une agence était en boucle sur un menu permettant au gestionnaire de traiter les demandes des clients et devait en même temps surveiller l'arrivée éventuelle d'un message de son agence nationale indiquant une mise à jour à prendre en compte (de façon immédiate).
> Le processus de cette agence devait donc surveiller à la fois l'entrée standard, `stdin`, (pour les demandes clients) et la socket ouverte entre l'agence nationale et elle (pour les mise à jour) sans avoir aucune possibilité de déterminer un ordre quelconque entre les deux types d'informations.

Plusieurs solutions existent :

- il est possible d'effectuer des lectures en **mode non-bloquant** (tenter une lecture et si rien n'est disponible, faire « autre chose » avant de tenter un peu plus tard une nouvelle lecture). En C, la fonction *fcntl()* permet de passer en mode non-bloquant ;
- il est aussi possible de **surveiller en boucle plusieurs voies d'entrée** (en lisant sur la première des voies qui devient active). En C, la fonction s*elect()* permet de passer surveiller en boucle plusieurs voies d'ES, éventuellement avec une temporisation (temps à l'issue duquel le select va s'arrêter même si aucune voie n'a été activée).

Interdiction temporaire d'accès à un fichier Lorsque plusieurs processus accèdent au même fichier, il peut y avoir des problèmes en raison des alternances de lecture/écriture par différents processus.

> Par exemple, dans le projet de l'an dernier, un fichier global, partagé par toutes les agences d'un pays contenait les descriptifs des sommes d'argent mises à disposition pour les clients de ce pays. Lors d'un retrait, l'agence devait

> vérifier en lisant ce fichier que la somme demandée était disponible puis effectuer le retrait et mettre à jour le fichier. Il n'était pas possible de laisser une autre agence accéder à ce fichier pendant cette opération, afin que deux retraits ne puissent pas être faits « en même temps » depuis deux agences différentes.

Protéger un fichier commun à plusieurs processus pendant les opérations « à risque » peut être fait en mettant en place un contrôle (souvent appelé **verrou externe** et placé dans le répertoire /var/lock). Ce verrou permettra ou non l'exécution d'une commande ou d'une série de commande. Attention, ce type de verrou sous Unix est limité à des groupes de processus utilisant le même **protocole d'accès**. On parle de verrou « *advisory* » et non « *mandatory* ». C'est la commande *flock* qui permet en shell de gérer ce type de blocage.

Deux syntaxes d'utilisation de la commande *flock* sont possibles. Des exemples sont présentés ci-dessous :

terminal 1 – la commande est lancée en premier ici

terminal 2 –

```
flock /var/lock/toto -c cat
```

```
flock /var/lock/toto -c ls
```

```
flock /var/lock/toto -c cat
```
ne pas débloquer avant la fin du processus dans terminal 2

```
flock -w 1 /var/lock/toto -c ls
```

```
flock /var/lock/toto -c cat
```
débloquer avant la fin du processus dans terminal 2

```
flock -w 1 /var/lock/toto -c ls
```

```
(
  flock -n 98 || exit 1
  sleep 5
  echo "terminal 1"
  sleep 5
          ) 98> /var/lock/toto
```

```
(
  flock -n 98 || exit 1
  echo "terminal 2"
          ) 98> /var/lock/toto
```

```
(
  flock -n 98 || exit 1
  sleep 5
  echo "terminal 1"
  sleep 5
          ) 98> /var/lock/toto
```

```
(
  flock 98 || exit 1
  echo "terminal 2"
          ) 98> /var/lock/toto
```

Les signaux

Les signaux font partie des mécanismes de communication entre processus et sont souvent utilisés en C. Vous allez les tester en shell avec les commandes `kill` et `trap`. La plupart des signaux sont réservés au système mais vous pouvez cependant utiliser :

- les signaux d'arrêt par le clavier : SIGINT (signal numéro 2, envoyé du clavier par `CTR-C`) et SIGQUIT (signal numéro 3, envoyé du clavier par `CTR-D`),
- les signaux à disposition des programmes utilisateur : SIGUSR1 (signal numéro 10) et SIGUSR2 (signal numéro 12),
- le signal SIGKILL (signal numéro 9) qui permet, depuis un terminal, de forcer l'arrêt d'un processus en cours.

Les signaux sont envoyés par un processus à un autre (identifiés par son pid). Le processus recevant le signal n'a aucune information complémentaire, en particulier, il ne peut connaître le processus qui a envoyé le signal.

Information projet

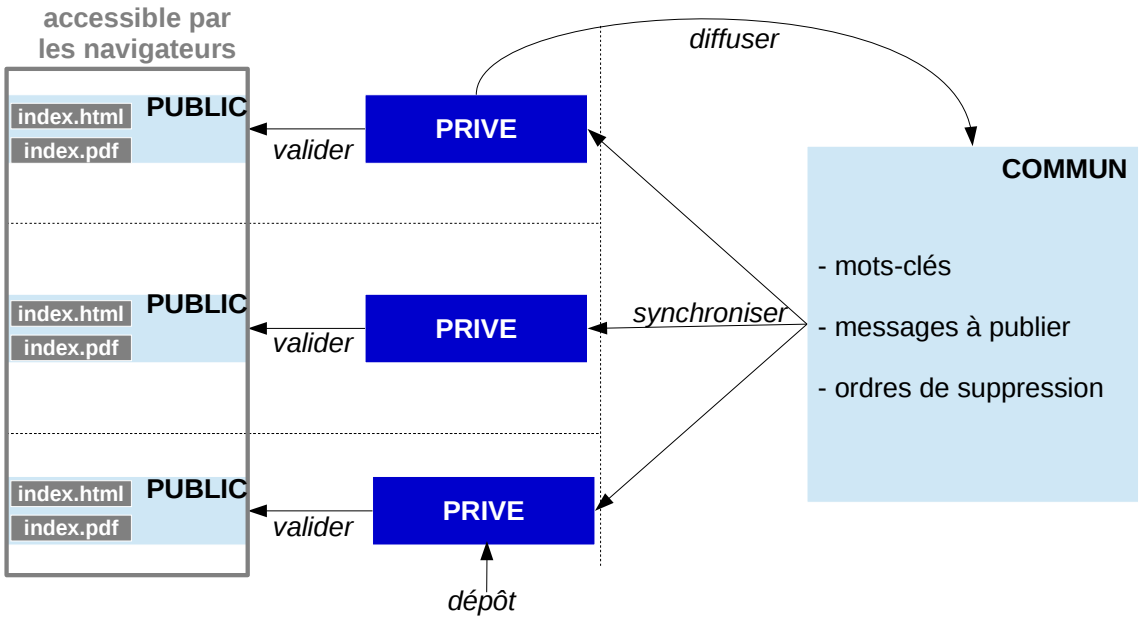
Contexte

Vous devez proposer un schéma de gestion pour un système de publication de messages. Ce système doit être constitué de points d'accès individuels (chacun sur un compte étudiant) et doit disposer d'un point d'accés commun.

Chaque point d'accès individuel est organisé autour de deux répertoires PUBLIC et PRIVE. Le répertoire PUBLIC doit être

accessible via un navigateur. Ce répertoire contient un fichier `index.html` et un fichier `index.pdf` qui seront mis à jour de façon automatique. Le répertoire PRIVE ne doit pas être accessible.

Le point d'accès commun devra mettre à disposition un répertoire COMMUN, accessible à tous les utilisateurs du système de publication. Ce répertoire COMMUN contiendra un descriptif des mots-clés à utiliser pour indexer les messages ainsi que des messages devant être diffusés sur l'ensemble des points d'accès individuels … et devant figurer dans leurs fichiers `index.html` et `index.pdf`. Une opération de synchronisation des points d'accès individuels est régulièrement lancée par le point d'accès commun.



Fonctionnement des points d'accès individuels

Sur chaque point d'accès individuel, vous devez proposer par menu à l'utilisateur :

- de **déposer** un nouveau message. Ce message sera systématiquement placé dans le répertoire PRIVE ;
- de **valider** un message du répertoire PRIVE. Ce message sera alors déplacé dans le répertoire PUBLIC dont les deux fichiers `index.html` et `index.pdf` seront mis à jour automatiquement ;
- de **diffuser** un message du répertoire PRIVE. Ce message sera alors envoyé au point d'accès commun qui le diffusera à tous les points d'accés lors de la prochaine synchronisation ;
- de **supprimer localement** un message du répertoire PUBLIC avec mise à jour automatique des deux fichiers `index.html` et `index.pdf` ;
- de **supprimer un message global** avec mise à jour des fichiers `index.html` et `index.pdf` sur tous les points d'accès individuels à l'occasion de la prochaine synchronisation ;
- de **supprimer de façon urgente un message global** avec mise à jour immédiate des fichiers `index.html` et `index.pdf` sur tous les points d'accès individuels.

Rôle du point d'accès commun

Le point d'accès commun a pour rôle :

- de gérer et diffuser régulièrement (par une opération de synchronisation) la liste des mots-clés à utiliser pour l'indexation des messages ;
- de mettre certains messages à disposition de tous les points d'accès. Ces messages sont appelés messages généraux ;
- de transférer les demandes de suppression des messages généraux.

Indexation

Le point d'accès commun dispose d'une hiérarchie de mots-clés organisée de la façon suivante :

- le fichier des mots-clés est en format texte, chaque mot-clé est un seul mot (avec éventuelllement des soulignés) ;
- il y a une ligne par mot-clé, le mot-clé est en fin de ligne précédé par les catégories auxquelles il appartient. La catégorie la plus générale est en début de ligne, la plus particulière est juste avant le mot-clé ;
- il peut y avoir des lignes vides et des lignes de commentaire (commençant par #).

L'utilisateur qui soumet un message donne une liste de mot-clé. Vous devez définir et mettre en œuvre une stratégie pour le traitement des mots-clés n'existant pas encore dans le fichier commun des mots-clés.
Avant de placer un message dans son répertoire PUBLIC (ou avant de le diffuser sur le point d'accès commun), le point d'accès doit construire automatiquement la liste des mots-clés contenus dans ce message. Cette liste devra être remise à jour à chaque

modification du fichier des mots-clés.

Contraintes diverses

Vous devez travailler sous les contraintes suivantes :

- les noms de fichiers pour les messages seront générés automatiquement (avec `mktemp`) par le point d'accés auquel le message est soumis. Un message sera constitué de trois fichiers : un fichier `.txt` qui contient le texte du message, deux fichiers d'extension `.indAut` et `.indMan` qui contiennent les mots-clés des indexations automatique et manuelle, respectivement ;
- les mots-clés associés automatiquement aux messages devront être recalculés à chaque mise à jour de la liste des mots-clés ;
- ppur chaque message, les mots-clés relatifs aux deux mécanismes d'indexation (automatique et manuelle) devront être présentés dans les fichiers `index.html` et `index.pdf` en deux paragraphes, chacun de ces paragraphes étant trié par ordre alphabétique.

Remarques importantes

Il s'agit du projet de SR1, les aspects interface utilisateur ne seront donc pas pris en compte lors de la correction ni de la démonstration. Il est par contre important de travailler dès maintenant dans l'optique suivante :

- vous devrez travailler au moins avec les deux comptes de votre binôme ;
- chaque point d'accès doit être un processus indépendant des autres, lancé dans son propre terminal ;
- il est essentiel de gérer avec précision :
  - la synchronisation de tous les processus en jeu,
  - les accès et la protection des fichiers et du répertoire COMMUN.

Tous ces points devront être mis en valeur dans votre présentation et votre rapport.

Vous trouverez sur le serveur pédagogique ufrsciencestech l'énoncé complet et les ressources nécessaires.
Le schéma de ce qui est attendu pour le rapport et la présentation
vous sera communiqués après le premier bilan.

Planning prévisionnel

Tous les mails doivent être envoyés à marie-noelle.terrasse@u-bourgogne.fr
Distribution du sujet – 16 septembre
Constitution des binômes – 25 septembre (date limite d'envoi)
par mail, en indiquant les noms et mails des étudiants constituants le binôme
Premier bilan – 23 octobre
par mail avec un pré-rapport en pdf indiquant de façon synthétique (une page) : ce qui est fait, ce qui reste à faire, les éventuels problèmes
Démonstrations - semaine du 7 au 11 décembre au plus tard (inscriptions obligatoires par mail)
Dépôt des rapports définitifs – 16 décembre (date limite d'envoi)
par mail, en pdf pour le rapport et avec une archive contenant les données et programmes

Exemple

Soit le jeu de mots-clés du système de publication :

```
IMAGE FORMAT EPS
IMAGE FORMAT JPEG
IMAGE FORMAT PNG
IMAGE FORMAT SVG
LANGAGE LaTeX
LANGAGE PROGRAMMATION OO C++
LANGAGE PROGRAMMATION OO Java
LANGAGE PROGRAMMATION OO Smalltalk
LANGAGE PROGRAMMATION Perl
LANGAGE PROGRAMMATION Python
LANGAGE PROGRAMMATION Ruby
LICENCE LIBRE General_Public_License
```

Soit le message suivant (fichier `mess-k98u.txt`) :

```
gnuplot ... from Wikipedia, the free encyclopedia

gnuplot is a command-line program that can generate two- and three-dimensional plots of functions, data, and data fits.

It is frequently used for publication-quality graphics as well as education.

The program runs on all major computers and operating systems (GNU/Linux, Unix, Microsoft Windows, Mac OS X, and others).

It is a program with a fairly long history, dating back to 1986.

Despite its name, this software is not distributed under the GNU General Public License (GPL), but its own more restrictive open source license.

gnuplot can produce output directly on screen, or in many formats of graphics files, including Portable Network Graphics (PNG), Encapsulated PostScript (EPS), Scalable Vector Graphics (SVG), JPEG
```

```
and many others.
It is also capable of producing LaTeX code that can be included directly in LaTeX documents, making
use of LaTeX's fonts and powerful formula notation abilities.
The program can be used both interactively and in batch mode using scripts.
The program is well supported and documented. Extensive help can also be found on the Internet.
The gnuplot core code is programmed in C. Modular subsystems for output via Qt, wxWidgets, and
LaTeX/TikZ/ConTeXt are written in C++ and lua.
Despite gnuplot's name, it is not part of or related to the GNU Project, nor does it use the GNU
General Public License, hence the decision to use a lowercase 'g'.
Official source code to gnuplot is freely redistributable, but modified versions thereof are not.
The gnuplot license instead recommends distribution of patches against official releases, optionally
accompanied by officially released source code.
Binaries may be distributed along with the unmodified source code and any patches applied thereto.
Contact information must be supplied with derived works for technical support for the modified
software.
gnuplot can be used from various programming languages to graph data, including Perl (via CPAN),
Python (via Gnuplot-py and SAGE), Java (via jgnuplot), Ruby (via Ruby Gnuplot), Ch (via Ch Gnuplot),
Haskell (via Haskell gnuplot) and Smalltalk (Squeak and GNU Smalltalk).
gnuplot also supports piping, which is typical of scripts. For script-driven graphics, gnuplot is by
far the most popular program.
```

Les mots-clés proposés avec le message, correspondant à une indexation manuelle par l'utilisateur, sont conservés dans un fichier `mess-k98u.indMan`) :	Les mots-clés extraits par le système de publication à partir de sa liste de mots-clés, correspondant à une indexation automatique, sont conservés dans un fichier `mess-k98u.indAut`) :
2D_graphics 3D_graphics command_line gnuplot script_driven_graphics	C++ EPS General_Public_License Java JPEG LaTeX Perl PNG Python Ruby Smalltalk SVG

Un dernier fichier (`mess-k98u.meta`) contiendra les informations relative à ce message telle que l'identité de l'utilisateur, le point d'accès auquel il a été soumis, la date de soumission et de publication …

Principales commandes

`clear`
Cette commande permet de nettoyer le terminal en effaçant tout ce qu'il contient.

`cut -d delimiteur -f champsAgarder`
`cut -c debut-fin`
Ce filtre permet de couper chaque ligne qui lui est passée. Avec l'option -d, le délimiteur détermine les champs parmi lesquels le choix sera fait. Avec l'option -c, seuls les caractères indiqués seront conservés.

`date`
Cette commande permet d'afficher sur écran la date et l'heure. Il est possible de choisir la forme sous laquelle la date est affichée (par exemple, `date +%D` pour avoir un affichage de la forme MM/JJ/AA).

`dc -e 'expressionAevaluer p'`
Il s'agit d'une calculette travaillant en ligne de commande et en notation postfixe (par exemple l'expression `(a+2)*3` sera notée `a 2 + 3 *`). Les nombres négatifs sont notés avec un souligné à la place du - (par exemple `-3` sera notée `_3`). Le `p` suivant l'expression indique qu'il faut afficher sur écran le résultat du calcul.

`exec fichier`
`exec redirection`
Cette commande permet de remplacer le programme en cours d'exécution par un autre (ayant un autre code à exécuter ou avec d'autres voies d'entrée-sortie).

`exit codeRetour`
Cette commande permet de terminer l'exécution d'un programme shell en renvoyant le code retour (entier) indiqué. Le code retour est facultatif.

`export listeVariables`

Cette commande indique que les variables indiquées doivent être communes au processus courant et à ses processus fils.

`flock options verrou –c commande`
`flock options descripteur`
Cette commande permet de bloquer l'exécution d'une commande si un verrou existe. Les verrous sont dans le répertoire /var/lock. La commande introduite par -c doit être unique et sans arguments. `flock` supporte, entre autres, les options suivantes :

- `–n` pour échouer immédiatement si le verrou est posé
- -w delai pour échouer si le verrou n'est pas levé avant l'échéance indiquée

La seconde forme de la commande permet par un jeu de redirection d'exécuter plusieurs instructions avec arguments (voir l'exemple présenté en cours). Attention, le descripteur utilisé doit être libre.

`grep options motif listeFichiers`
Cette commande permet d'extraire des fichiers les lignes qui correspondent au motif pour les afficher sur écran. Le motif est le plus souvent une expression régulière. De nombreuses options sont disponibles dont :

- `–P` pour écrire le motif comme une expression régulière en Perl (`grep –P motif fichier`)
- `–n` pour demander l'affichage des numéros de lignes devant les lignes (`grep –n motif fichier`)
- `–v` pour affichage des lignes qui ne correspondent pas au motif (`grep –v motif fichier`)
- `–o` pour affichage de l'expression reconnue et non de la ligne entière (`grep –o motif fichier`)

`id`
affiche les noms et identifiants d'utilisateur et de groupes effectif ou réel.

`kill –numeroSignal pid`
`kill –l`
Cette commande permet d'envoyer un signal à un processus. Elle est souvent utilisée pour forcer l'arrêt d'un processus qui boucle (envoi du signal SIGKILL, numéro 9).
L'appel `kill –l` permet d'avoir la numération des signaux sur la machine sur laquelle il est exécuté.

`lastlog`
Cette commande indique la date et heure de dernier login des utilisateurs déclarés d'une machine.

`mktemp –p repertoire schema`
Cette commande crée dans le répertoire indiqué un fichier vide, dont le nom n'existe pas encore, et affiche ce nom sur écran. Le schéma du nom est généralement constitué d'un préfixe et d'autant de `X` que l'on souhaite avoir de caractères complémentaires dans le nom du fichier.

`newgrp – groupe`
Cette commande permet de devenir membre du groupe indiqué. Si l'option – est utilisée, un nouvel environnement est lancé (comme à la connexion). Si le groupe n'est pas spécifié, on revient au groupe par défaut (indiqué dans /etc/passwd).

`passwd`
`passwd –[elu] login`
Cette commande permet de changer de mot de passe. Avec la seconde forme d'appel, l'utilisateur `root` peut forcer l'expiration du mot de passe (expire), bloquer ou débloquer (lock unlock) le compte d'un utilisateur.

`ps option`
Cette commande permet de lister les processus en cours sur une machine. L'option `–e` demande un listing de tous les processus. L'option `–u login` demande un listing des processus dont l'utilisateur indiqué est propriétaire.

`pstree`
Cette commande permet un affichage de la hiérarchie des processus en cours sur une machine.

`sg groupe commande`
Cette commande permet d'exécuter la commande en faisant partie du groupe indiqué. Il peut être nécessaire de placer la commande entre guillemets.

`sleep durée unité`
Cette commande met en sommeil le processus qui l'exécute pour le temps indiqué. L'unité de temps peut être `s, m, h,` etc.

`ssh login@adresseMachine`
Cette commande permet de se connecter à distance. Retour au shell initial par *exit*.

`su login`
Cette commande permet de prendre le rôle de l'utilistaeur indiqué (par défaut `root`). Retour au shell initial par *exit*.

`top -n nombreIterations`
Cette commande permet un affichage dynamique des processus en cours. Les premiers affichés sont ceux les plus actifs. La liste est modifiée plusieurs fois au fur et à mesure pendant le nombre d'itérations demandées.

`tr ancien nouveau`
`tr option caractere`
Ce filtre permet de remplacer un caractère par un autre, de supprimer les occurrences multiples d'un caractère (option `-s` pour squeeze), de supprimer un caractère (option `-d` pour delete).

`trap 'instructions' numéroSignal`
Cette commande permet de modifier/définir le comportement du processus qui l'exécute à la réception du signal indiqué.

`xterm -hold -e commande arguments`
Cette commande permet de lancer un nouveau terminal dans lequel la commande est exécutée. L'option `-hold` indique que le terminal doit rester ouvert après exécution de la commande.

