ARCHITECTURE TECHNIQUE ET SYSTÈME

Denis MOREAUX

Table des Matières

Introduction....4
I Démarrage et premiers pas sous Linux....5
I.1 VirtualBox....5
I.1.a Présentation générale de VirtualBox....5
I.1.b La fenêtre de la VM....6
I.1.c Les SnapShots....8
I.1.d Configuration d'une VM....8
I.2 Installation et lancement de la VM....9
I.2.a Changement du répertoire pour la VM....9
I.2.b Importation de la VM....9
I.2.c Création d'un répertoire partagé....9
I.2.d Démarrage et arrêt de la VM, entrée en session....10
I.2.e Changements avancés de VirtualBox....11
I.3 OS, Shell, X11, Window Manager, Desktop Environment....11
I.4 Commandes de base....12
I.4.a Répertoires....12
I.4.b Les fichiers....14
I.4.c Les redirections et les filtres....16
I.5 Editer un fichier....17
I.6 Midnight Commander....18
I.7 Les permissions....19
I.7.a Les mécanismes disponibles....19
I.7.b Comment fonctionnent les permissions de base ?....19
I.7.c SetUID/SetGID/Sticky....20
I.7.d Utilisation en pratique....20
I.7.e Quelques cas d'utilisation....21
II Programmation sous Linux....23
II.1 GCC....23
II.2 Makefile....24
II.2.a Structure de base....24
II.2.b Règles implicites et macros....24
II.2.c Un Makefile plus complet....25
II.3 Pages de manuel....26
III Accès fichiers de bas niveau....27
III.1 File descriptor vs FILE *....27
III.2 Erreurs....27
III.3 open/close....27
III.4 read/write....28
III.5 lseek....28
III.6 dup/dup2/fcntl/ioctl....29
III.7 select/FD_CLR/FD_SET/FD_ISSET/FD_ZERO....29
IV Process....31
IV.1 Process et shell....31
IV.2 Créer un nouveau process en C....31
IV.3 Signaux entre process....32
IV.3.a Signal(2)....33
IV.3.b sigaction(2)....33
IV.4 Reap Childs....33
IV.5 alarm(2)....34

V Les pipes....35
V.1 Pipes anonymes....35
V.2 pipes nommés....35
V.3 Utilisation des pipes....35
VI Mémoire partagée....37
VI.1 Commandes shell....37
VI.2 Programmation en C....37
VII Les sémaphores....39
VII.1 Définition....39
VII.2 Les fonctions C pour gérer les sémaphores....39
VII.2.a Créer un bloc de sémaphores....39
VII.2.b Actions sur les sémaphores (P() et V())....39
VII.2.c Modifications d'un sémaphores....40
VII.3 Zone critique (MutEx)....41
VII.4 Producteur-Consommateur....41
VII.5 Le problème du rendez-vous....42
VII.6 Problème des lecteurs/rédacteurs....42
VII.7 Problème du Deadlock....42
VII.8 Le problème du dîner des philosophes....43

Introduction

Les processeurs actuels sont de plus en plus puissants. Cette augmentation de puissance se fait principalement par la multiplication du nombre de cœurs dans les processeurs actuels. Mais, pour bien gérer cette puissance, il est nécessaire que les programmes effectuent plusieurs tâches simultanément, chaque tâche n'occupant qu'un seul cœur de processeur.

Ces diverses tâches doivent pouvoir communiquer entre elles afin de former un tout. Les différentes manières d'effectuer cette communications seront présentées dans ce cours. Ces explications seront envisagées pour un environnement de type Unix (Solaris, OpenBSD, FreeBSD, Linux, AIX, MacOS/X...) mais les concepts sous-jacents sont également valables dans d'autres environnements.

Afin de pouvoir travailler dans ces environnements, des explications de base[1] sur l'environnement Unix seront données, ainsi que sur l'utilisation de l'environnement de virtualisation VirtualBox. Un Linux/Debian sera utilisé pour les manipulations.

Au terme du cours, vous serez capables d'éditer et de compiler un programme en C sous Linux, de créer un programme travaillant en multitâches et de faire communiquer les tâches à l'aide de fichiers ou des IPC System V.

1 Un cours plus poussé sur l'administration Unix est prévu en troisième année

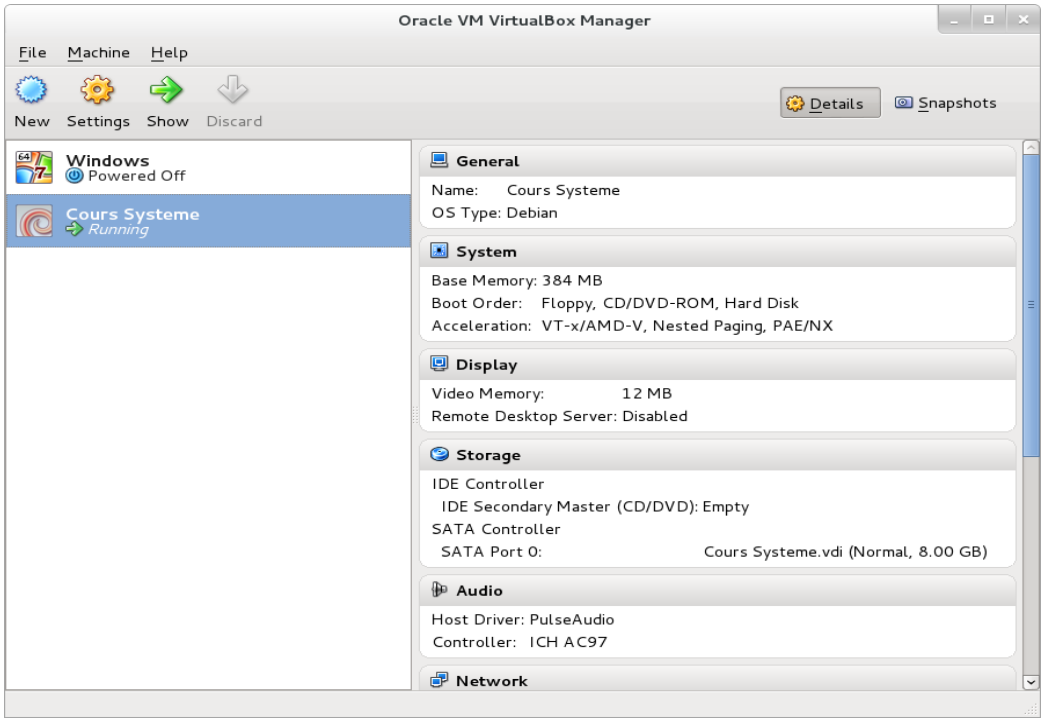
I Démarrage et premiers pas sous Linux

I.1 VirtualBox

I.1.a Présentation générale de VirtualBox

VirtualBox permet de faire tourner des machines virtuelles à l'intérieur d'un OS hôte. Développé longtemps chez Sun, il a été acheté par Oracle et est disponible aussi bien sous Linux que sous MS windows.

Quand on le lance, on obtient un écran semblable à celui repris sur la capture d'écran I.1



Capture I.1: Gestionnaire de machines virtuelles de VirtualBox

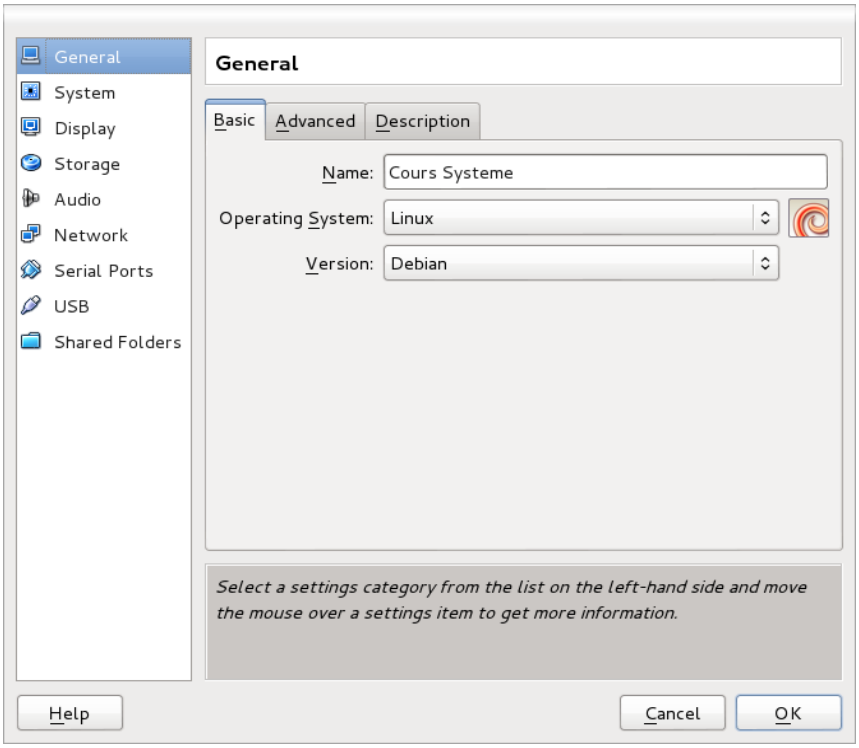
La liste des machines virtuelles (VM) disponibles apparaît sur la gauche de la fenêtre, avec leur status (Running pour une machine allumée, Powered Off pour les machines éteintes, ...). Les caractéristiques principales de la machine sélectionnée apparaissent sur le côté droit de la fenêtre.

Pour créer une nouvelle VM, on utilise le bouton New qui lance un Wizard qui permettra d'entrer les principales caractéristiques de la VM (des valeurs par défaut sont proposées selon le type de VM). Il sera cependant nécessaire de retoucher ces caractéristiques à l'aide de l'écran de paramètres avancés accessible à l'aide du bouton Settings. Le bouton Start permet de démarrer la VM sélectionnée et devient alors un bouton Show qui permet d'afficher l'écran de la VM sélectionnée si ce dernier n'est pas visible.

Un clic droit sur une VM fait apparaître un menu qui permet d'effacer une VM que l'on ne désire pas conserver, de cloner une VM (en créer une copie), d'accéder aux fichier de la VM sur le disque dur,...

L'option Settings, tout comme le bouton Settings en haut de l'écran, permet d'éditer les paramètres

de la VM. On arrive alors à l'écran repris dans la capture I.2 qui permettra de modifier de nombreux paramètres.



Capture I.2: Paramètres d'une VM

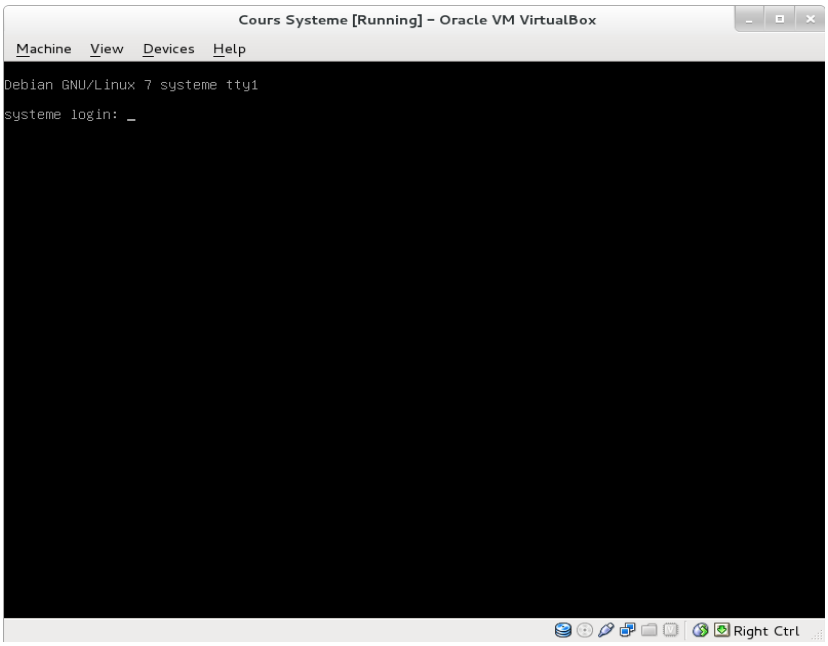
Sur la gauche, on dispose des catégories de paramètres, sur la droite, on peut modifier les paramètres en question, parfois répartis sur plusieurs onglets. Les principaux paramètres sont décrits à la section I.1.d

Dans le menu File, on trouve les options Export et Import qui permettent d'exporter une VM au format OVF compatible avec d'autres systèmes de machines virtuelles ou d'importer une VM au format OVF. Lorsque l'on importe une VM, on peut décider de générer de nouvelles adresses MAC afin d'éviter des conflits entre les différentes VM.

Le menu préférences permet d'installer le package d'extensions (support USB2, RDP et PXE pour carte réseau Intel 1000) officiel ou d'autres venant d'autres sources, de gérer les interface réseau hôte ou de choisir la touche utilisée comme *HostKey.*

I.1.b La fenêtre de la VM

Par défaut, quand une VM est lancée, elle apparaît dans une fenêtre telle que celle de la capture I.4.



Capture I.3: Ecran de la VM

En l'absence des *Guest Additions*, une fois que l'on clique dans cette fenêtre, la souris et le clavier sont capturés par la VM. Pour récupérer le contrôle au niveau du système hôte, il suffit d'utiliser la *HostKey* (par défaut, contrôle-droit).

La *HostKey* permet également d'autres fonctions.

HostKey+F	Permet de passer en mode plein écran ou de revenir dans le mode fenêtré
HostKey+Del	Permet d'envoyer un Ctrl+Alt+Del à la VM (tant Windows que Linux l'interceptent au niveau du système)
HostKey+F1 ... HostKey+F12	Permet d'envoyer un Ctrl+Alt+F1... Ctrl+Alt+F12 (utilisé sous Linux/X11 pour changer de console virtuelle)
HostKey+H	Permet de simuler un appui sur la touche Power (appui bref, qui laisse à l'OS la possibilité de s'arrêter proprement)
HostKey+R	Permet de simuler un appui sur le bouton Reset (par exemple, si la VM est plantée)

Lorsque l'on est en mode plein écran, une petite barre d'outil apparaît lorsque l'on amène le pointeur de souris en bas de l'écran, au milieu. Cette barre d'outils reprend les options du menu présent dans la fenêtre de la VM, ainsi qu'une icône qui permet de verrouiller cette barre d'outil (afin qu'elle reste toujours présente).

Dans le menu Devices, on peut choisir les images de CD ou DVD que l'on veut activer dans le lecteur de CD virtuel, modifier les répertoires partagés entre ordinateur hôte et VM, gérer les périphériques USB accessibles à la VM ou activer l'image de CD avec les *Guest Additions* pour les installer dans la VM (à noter que la Debian détecte d'elle-même qu'elle tourne dans une VM et dispose des packages nécessaires pour installer l'intégration VirtualBox sans devoir passer par cette

étape).

Lorsque l'on ferme une machine virtuelle, on peut soit sauver son état (un peu l'équivalent de mettre la machine en hibernation), envoyer un signal d'extinction (comme *HostKey*+H) ou éteindre *brutalement* la machine (l'équivalent de retirer la prise).

Les *Guest Additions* permettent la communication entre la machine hôte et la VM. Copier/coller, intégration de la souris (il n'est plus nécessaire de la capturer), dossiers partagés, ...

I.1.c Les SnapShots

VirtualBox permet de prendre des snapshots de la VM (qu'elle soit à l'arrêt ou en train de tourner). Ces snapshots sont des états complets de la machine et de ses disques au moment où le snapshot est pris.

On peut prendre un tel snapshot à l'aide de HostKey+T dans la VM ou à partir de l'onglet snapshot de la fenêtre principale de VirtualBox (l'icône en forme d'appareil photo). Une boite de dialogue demande alors le nom que l'on désire donner au snapshot ainsi qu'une description.

Les snapshots apparaissent dans la liste des snapshots dans la fenêtre principale. Ils sont agencés sous une forme d'arbre et un snapshot particulier apparaît sous le nom *Current State* indiquant la situation actuelle de la VM.

Par la suite, on peut revenir à un snapshot. L'état actuel de la VM est perdu (il peut être intéressant de créer un snapshot pour le conserver avant de revenir à un état antérieur) et on revient à la situation sélectionnée. Il est également possible d'effacer un snapshot dont on n'a plus besoin.

A noter que revenir à un ancien snapshot a également pour effet de ramener les disques associés à la VM à l'état dans lequel ils se trouvaient. Il s'agit réellement d'un retour dans le passé. Cela implique également que les snapshots peuvent utiliser un espace disque non négligeable, même si seulement les modifications du disque sont enregistrées.

Les snapshots permettent de tester différentes configurations (par exemple, une mise à jour dont on craint qu'elle ne rende certain services non-fonctionnels) et de revenir en arrière si un problème est constaté. Ce mécanisme est également mis en œuvre par VirtualBox lorsque l'on clone une VM en liant le disque au lieu de le copier. On se retrouve alors avec un snapshot commun à la VM originale et à la copie.

I.1.d Configuration d'une VM

Lorsque l'on accède à la configuration d'une VM, on obtient l'écran repris à la capture I.2 (page 6). Cette configuration permet de définir le hardware utilisé par la VM ainsi que quelques options de fonctionnement.

Seuls les options utiles pour ce cours sont expliquées ici. Pour plus d'informations, se référer au manuel de VirtualBox

I.1.d.i General

Les options générales permettent de changer le nom de la VM.

I.1.d.ii System

L'onglet motherboard permet de changer la taille de mémoire disponible pour la VM

L'onglet processeur permet de choisir le nombre de processeurs disponibles et d'imposer une limite

à la consommation de processeur par la VM (attention, cela peut amener à des problèmes de timing).

I.1.d.iii Shared Folders

Si les *Guest Additions* ont été installées, il est possible de rendre accessible des répertoires de la machine hôte à partir de la VM. Le répertoire de la machine hôte doit être physiquement présent sur cette machine (il ne peut pas se trouver sur un disque réseau).

Un tel partage doit avoir un nom (utilisé pour y accéder à partir de la VM) et un répertoire (sur la machine hôte). On peut également le forcer en lecture seule ou demander qu'elle soit automatiquement attachée à la VM lorsque cette dernière démarre. Sous Linux, il apparaîtra dans le répertoire /media/sf_nom

I.2 Installation et lancement de la VM

I.2.a Changement du répertoire pour la VM

Par défaut, les VM sont sauvées dans le répertoire de l'utilisateur. Comme ce répertoire peut faire partie d'un profil errant ou être accédé au travers du réseau, il est nécessaire de le déplacer sur le disque dur local.

Pour ce faire, suivre la procédure suivante (elle ne doit être effectuée qu'une seule fois par machine, sauf si la machine est réinstallée)

- Créer un répertoire sur le disque C: (par exemple : C:\VirtualBoxVM)
- Menu File -> Preference
- Dans l'onglet General, modifier Default Machine Folder pour qu'il pointe sur le répertoire créé ci-dessus

I.2.b Importation de la VM

La VM est fournie sous format OVA. Il est nécessaire de l'importer dans VirtualBox avant de l'utiliser.

La procédure est assez simple :

- Menu File -> Import Appliance
- Dans la première page du Wizard, choisir le fichier OVA CoursSysteme.ova
- Dans la seconde page, choisir le nom que l'on désire donner à la VM (ligne Name) et cocher la case Reinitialize the MAC address of all network cards
- Valider en cliquant sur Import

Une fois l'importation terminée, la VM apparaîtra dans VirtualBox.

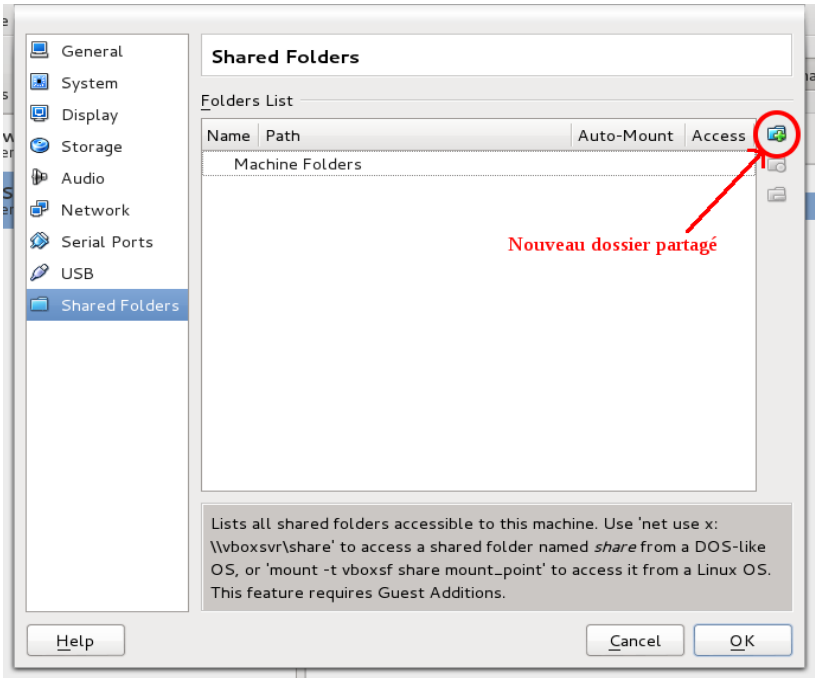
I.2.c Création d'un répertoire partagé

Afin de pouvoir transférer des fichiers entre la VM et l'ordinateur hôte, il faudra prévoir un répertoire partagé entre ces derniers.

- Créer un répertoire sur le disque C: (par exemple C:\CoursSystemeShare)

- Sélectionner la VM et cliquer sur Settings (ou clic-droit->Settings)
- Sélectionner les options Shared Folders dans la liste de gauche (voir capture I.4)
- cliquer sur le dossier avec un +, en haut à droite par rapport à la liste des folders
- Dans Folder Path, entrer le répertoire créé ci-dessus
- Dans Folder Name, entrer host
- cocher la case AutoMount et valider
- Valider les options

Le répertoire partagé apparaîtra dans la VM sous le nom /media/sf_host



Capture I.4: Dossiers partagés

I.2.d Démarrage et arrêt de la VM, entrée en session

Pour lancer la VM, il suffit de la sélectionner et de cliquer sur l'icône Start (ou clic-droit -> start)

Lors du démarrage de linux, le premier écran est celui du boot-manager (GRUB). Cet écran va proposer pour chaque version de noyau installée deux options : le démarrage normal ou le mode recovery. Si on attend quelques secondes, la première ligne sera exécutée (on peut passer ce délai en appuyant sur Enter).

Une fois GRUB passé, la fenêtre va se redimensionner et le chargement de linux va commencer. Une fois l'écran de connexion affiché, la VM est prête à l'emploi.

Pour l'arrêter, dans la fenêtre de la VM, utiliser les touches Contrôle-droit et H. Attendre que la fenêtre se ferme.

Si on désire éviter la procédure de démarrage, on peut mettre la machine en pause au lieu de l'éteindre. Pour ce faire, il suffit de fermer la fenêtre et de choisir l'option Save the machine state. L'extinction est plus rapide mais la VM est bloquée et il n'est pas possible de changer ses paramètres ou de l'exporter. Il est donc conseillé de toujours éteindre la VM complètement.

La mise en pause d'une machine ne libère pas les IPC qui n'auraient pas été nettoyés ni ne tue les tâches qui tourneraient en boucle.

La machine virtuelle proposée possède un utilisateur dont le login et le mot de passe sont *student*.

I.2.e Changements avancés de VirtualBox

La VM a été configurée pour utiliser peu de mémoire et un seul processeur. Ces valeurs ont été choisies afin que la VM puisse tourner sur presque toutes les machines (qu'il s'agisse des ordinateurs de l'école ou de vos machines personnelles).

Il est cependant possible d'augmenter la mémoire si cela s'avère nécessaire. Attention que cette augmentation se fait au détriment du système hôte, toute la mémoire allouée à la VM lui étant réservée lorsqu'elle est lancée.

Cette augmentation de mémoire doit se faire avec la VM complètement arrêtée (il n'est pas possible de le faire si la machine est en pause). Au démarrage suivant, Linux s'adaptera sans aucun problème à la nouvelle quantité de mémoire.

Tout au long du cours, différents exercices seront proposés. Si l'on désire repartir avec une machine propre à chaque exercice, il existe plusieurs solutions :

- Effectuer un snapshot de la VM après l'avoir importée et un second une fois l'exercice terminé, avant de revenir au snapshot de départ
- Effectuer un snapshot de la VM et copier les fichiers source sur le disque partagé avant de revenir au snapshot de départ
- Cloner la VM en mode Linked Clone

Il est fortement déconseillé de faire un clone complet de la machine ou d'importer la machine plusieurs fois.

Quelle que soit la méthode choisie, il est préférable de transférer ses sources de la VM vers un support externe (clé USB, disque personnel, ...) afin de ne pas perdre le travail effectué si la machine hôte venait à être réinstallée.

I.3 OS, Shell, X11, Window Manager, Desktop Environment

Contrairement à ce que les utilisateurs ont tendance à penser, l'Operating System reste caché en arrière plan. Ce que les utilisateurs voient sont en fait des applications qui tournent au dessus de l'OS.

L'OS est responsable de la gestion de la mémoire, des tâches, des entrées-sorties et fourni des services qui peuvent être utilisés par les applications. Dans le cas des systèmes Linux, il s'agit du *kernel* qui ne se manifeste normalement qu'en début de démarrage de la machine, avant le démarrage des services.

Une fois l'OS démarré, des applications seront lancées. Il s'agit par exemple des divers serveurs présents sur la machine (serveur web, FTP, DNS, ...), de *daemons* (applications tournant en arrière plan et fournissant divers services comme la détection des périphériques USB à chaud), de programmes effectuant des tâches de configuration (clavier, réseau, ...).

Dans le cas d'un système multi-utilisateurs, cette phase de démarrage se termine par le lancement des mécanismes de login qui permettent à l'utilisateur d'entrer ses identifiants.

Une fois l'utilisateur connecté, la main est passée à un programme appelé shell. Le principal rôle du shell est de permettre à l'utilisateur de lancer des applications. Ce shell peut être en mode texte

(comme bash sous Unix) ou en mode graphique (comme l'explorateur sous MS Windows). Un OS peut proposer plusieurs shells différents (cmd.exe est un shell en mode texte sous MS Windows, csh, ksh sont des shells en mode texte sous Unix)

Un système Unix est basé sur le mode texte. Le système graphique est une application séparée appelée serveur X11 (créée par Xerox, bien avant les GUI modernes). Les programmes en mode graphique communiquaient avec le serveur X11 au travers de la connexion réseau. Maintenant, ils utilisent également la mémoire partagée afin de permettre un transfert plus rapide des informations graphiques.

Pour le serveur X11, une fenêtre est un simple rectangle avec un contenu. Les décorations (bordures permettant de redimensionner, barre de titre, ...) sont gérées par un programme séparé appelé window manager.

Au dessus du window manager, on peut placer un shell graphique qui permet de disposer d'icones sur le fond d'écran, de naviguer dans les dossiers, ... On parle de Desktop Environment. Dans le système MS Windows, le Desktop Environment est l'explorateur (explorer.exe)

Si les distributions Linux proposent un (plusieurs en fait) environnement graphique complet, ce cours se basera sur le mode texte. Le mode texte, même si un peu rébarbatif au début est souvent bien plus puissant que le mode graphique.

I.4 Commandes de base

Sous Unix, un bon nombre des commandes de base ne sont pas inclues dans le shell mais sont des petits programmes. Cela permet de mettre une de ces commandes à jour sans toucher au reste du système. Cela permet également de rendre la programmation de ces commandes plus simples dans la mesure où il s'agit de programmes de petite taille très spécialisés.

I.4.a Répertoires

Sous Unix, la quasi totalité des éléments accessibles à l'utilisateur sont des fichiers. Certains de ces fichiers ont des types spéciaux (pipes nommés, socket unix, devices), d'autres seront des conteneurs qui contiendront d'autres fichiers (on parle de répertoires).

Les informations relatives aux fichiers sont séparées des noms de ces dits fichiers. Elles sont situées dans des éléments appelés i-nodes. Cela permet de disposer de plusieurs noms pour le même fichier, on parle de liens (physiques, par opposition aux liens symboliques qui sont des fichiers d'un type particulier).

Chaque répertoire contient deux noms de fichiers particuliers . et .. qui pointent respectivement sur le répertoire lui-même et sur le répertoire parent. Un répertoire donné pourra donc avoir un grand nombre de liens qui pointent vers lui : son nom de répertoire, le nom . dans le répertoire et les noms .. de chacun des répertoires qui sont contenus dedans.

Lorsqu'un fichier (ou un répertoire qui n'est qu'une sorte de fichier spécial) n'a plus aucun lien qui pointe dessus ni aucun programme qui l'aie ouvert qui tourne, le fichier sera effectivement effacé du disque dur. Ce n'est donc pas parce que l'on a effacé un fichier du répertoire qui le contient que l'espace disque sera réellement effacé.

On peut former le nom d'un fichier (ou répertoire) de plusieurs façons.

La première est le chemin absolu. On part du répertoire racine (celui qui contient tout le système) et on indique tous les noms des répertoires à traverser pour atteindre le fichier. Ces noms seront

séparés par le caractère /[2]. On aura donc un chemin du style /usr/include/stdio.h qui indique le fichier stdio.h contenu dans le répertoire include, lui même contenu dans le répertoire usr qui est contenu dans le répertoire racine.

On peut également fournir un chemin relatif. Ce chemin part du répertoire courant et suit les noms de répertoires indiqués. Par exemple, si on se trouve dans le répertoire /usr/lib, on pourra référencer le fichier précédent sous la forme ../include/stdio.h (on prend le répertoire parent de /usr/lib, à savoir /usr, on va dans le répertoire include et on prend le fichier stdio.h). Un chemin relatif contient au moins un caractère /.

La troisième façon de désigner un fichier est de parler d'un fichier dans le répertoire courant. Par exemple, si on se trouve dans /usr/include, on pourra utiliser simplement stdio.h.

A noter que parfois, il est nécessaire de fournir un chemin relatif ou absolu (mais pas un fichier dans le répertoire en cours) et on peut utiliser ./nomfichier pour pointer sur un fichier du répertoire en cours.

<table>
<tr><td>cd</td><td>Change le répertoire courant
<table>
<tr><td>cd nom</td><td>Entre dans le répertoire nom présent dans le répertoire courant</td></tr>
<tr><td>cd ~</td><td>Retourne dans le répertoire personnel de l'utilisateur courant</td></tr>
<tr><td>cd ..</td><td>Remonte dans le répertoire père du répertoire courant</td></tr>
<tr><td>cd /usr/include</td><td>Se déplace dans le répertoire /usr/include</td></tr>
<tr><td>cd ~user</td><td>Se déplace dans le répertoire de l'utilisateur user</td></tr>
</table></td></tr>
<tr><td>pwd</td><td>Affiche le répertoire courant</td></tr>
<tr><td>ls</td><td>Affiche le contenu d'un répertoire
<table>
<tr><td>ls</td><td>Affiche la liste des fichiers contenus dans le répertoire courant. Les fichiers dont le nom commencent par un point sont cachés</td></tr>
<tr><td>ls repertoire</td><td>Affiche les fichiers contenus dans le répertoire indiqué. Ex : ls /bin, ls src/projet1, ...</td></tr>
<tr><td>ls -a [répertoire]</td><td>Affiche tous les fichiers, y compris ceux dont le nom commencent par un point. On peut éventuellement fournir un nom de répertoire.</td></tr>
<tr><td>ls -l [répertoire]</td><td>Affiche les fichiers sous le format long, qui inclus les droits, le nombre de liens sur le fichier, le propriétaire et le groupe du fichier, sa taille et sa date de dernière modification.</td></tr>
<tr><td>ls -F [répertoire]</td><td>Affiche les fichiers en ajoutant un caractère indiquant le type de fichier après le nom. * indique un fichier exécutable, / un répertoire, = un socket unix, @ un lien symbolique, | un pipe nommé.</td></tr>
<tr><td>ls -alF [répertoire]</td><td>Affiche tous les fichiers (y compris les fichiers commençant par un .), sous le format long et avec un caractère indiquant le type de fichier. Pour les liens symboliques, le fichier pointé est indiqué et le caractère de type de fichier indique le type du fichier pointé.</td></tr>
<tr><td>ls *.jpg</td><td>Affiche tous les fichiers se terminant par .jpg, le caractère * remplace zéro ou plusieurs caractères</td></tr>
</table></td></tr>
</table>

2 Pour ceux qui se demandent pourquoi un / et non un \ comme sous MS Windows, posez vous la question inverse : pourquoi MS windows utilise-t'il un \... Unix était là avant que MS DOS ne commence à supporter les répertoires

<table>
<tr><td>mkdir</td><td>Crée un sous répertoire par rapport au répertoire courant. L'option -p permet de créer une série de répertoires imbriqués
<table>
<tr><td>mkdir abc</td><td>Créer un sous-répertoire abc</td></tr>
<tr><td>mkdir -p abc/def/ghi</td><td>Crée un sous-répertoire abc. Dans ce dernier, crée un sous-répertoire def. Dans le sous-répertoire def, crée le sous-répertoire ghi</td></tr>
<tr><td>mkdir abc def</td><td>Crée deux sous-répertoires dans le répertoire courant : abc et def</td></tr>
</table></td></tr>
<tr><td>rmdir</td><td>Efface un ou plusieurs sous-répertoire vide ex : rmdir abc, rmdir abc def</td></tr>
<tr><td>du</td><td>Liste le répertoire fourni et ses sous-répertoires ainsi que la taille des données dans chacun d'entre eux
<table>
<tr><td>du répertoire</td><td>Donne la taille des données dans le répertoire fourni et les taille des données dans chacun de ses sous-répertoires, en blocs. Il s'agit de la taille occupée sur le disque et pas de la taille utilisée</td></tr>
<tr><td>du -s répertoire</td><td>Donne la taille des données dans le répertoire fourni (summary)</td></tr>
<tr><td>du -m répertoire</td><td>Donne la taille en Mb au lieu de blocs</td></tr>
<tr><td>du -k répertoire</td><td>Donne la taille en Kb au lieu de blocs</td></tr>
<tr><td>du -h répertoire</td><td>Donne la taille dans un format lisible par un humain (1K, 2M, 3G, ...) l'unité de mesure est choisie séparément pour chaque valeur.</td></tr>
</table></td></tr>
</table>

I.4.b Les fichiers

Sous Unix, les fichiers sont définis par un élément appelé i-node. Cet élément contient toutes les informations relatives au fichier à l'exception de son nom. En d'autres termes, il contient la taille du fichier, ses permissions, son propriétaire, la date de création, celle de dernière modification, le nombre de liens sur le fichier, son type, les blocs utilisés sur le disque dur, ...

Le type de fichier ne concernent pas le type de données contenues dans un fichier (image, son, programme, ...) mais indiquent s'il s'agit d'un fichier de données, d'un répertoire, d'un lien symbolique, d'un pipe nommé, d'un device, ...

Sous Unix, le contenu d'un fichier de donnée ne dépend pas de son extension. L'extension est présente à titre purement informatif. C'est ainsi qu'on utilisera souvent un fichier sans extension pour un programme, qu'il s'agisse d'un exécutable compilé, d'un script shell, d'un programme en Perl, en Tk/TCL, en python, ...

Unix utilise des *magic names* pour identifier le type de contenu. C'est ainsi que les scripts qui doivent être interprétés par un programme commencent par #! suivi du chemin absolu de l'interpréteur qui les concernent : #!/usr/bin/perl pour un programme Perl, #!/bin/sh pour un script shell, ... Les exécutables ELF (le principal format) commencent par un octet 0x7f suivi des quatre lettres ELF.

On trouve des magic names dans à peu près tous les types de fichiers (une exception est les fichier MS Office docx, pptx, ... qui ne possèdent que le magic name d'une archive ZIP). Les images GIF commencent par gif87 ou gif89, les fichiers ZIP par les deux lettres PK, ...

Les principales commandes qui peuvent être effectuées sur les fichiers sont reprises dans le tableau qui suit :

<table>
<tr><td>cp</td><td>Copie un fichier d'un endroit vers l'autre<table>
<tr><td>cp ancien nouveau</td><td>Copie le fichier ancien vers nouveau</td></tr>
<tr><td>cp -r anciendir nouveaudir</td><td>Copie un répertoire et tout son contenu de façon récursive</td></tr>
<tr><td>cp fich1 fich2 fich3 répertoire</td><td>Copie plusieurs fichiers vers un répertoire</td></tr>
</table></td></tr>
<tr><td>mv</td><td>Renomme ou déplace un fichier. La syntaxe est mv ancien nouveau. Pour déplacer un fichier vers un autre répertoire, il suffit de préciser le nouveau répertoire au lieu d'un nouveau nom de fichier</td></tr>
<tr><td>rm</td><td>Efface un ou des fichiers. Attention, il n'y a pas de corbeille pour récupérer un fichier effacé.<table>
<tr><td>rm fichier</td><td>Efface le fichier</td></tr>
<tr><td>rm -r répertoire</td><td>Efface le répertoire et son contenu de façon récursive. A utiliser avec précaution</td></tr>
</table>La commande rm est souvent aliasée en rm -i qui demande une confirmation avant effacement. Pour éviter la confirmation, on peut soit appeler directement /bin/rm (ce qui permet d'éviter l'alias) soit utiliser l'option -f (force).</td></tr>
<tr><td>ln</td><td>Crée un lien vers un fichier.<table>
<tr><td>ln ancien nouveau</td><td>Crée un lien physique. Le nouveau nom pointe vers le même i-node que l'ancien nom. Les deux noms sont donc des alias vers le même fichier et les deux liens doivent être effacés pour que le fichier disparaisse du disque dur</td></tr>
<tr><td>ln -s ancien nouveau</td><td>Crée un lien symbolique. Le lien symbolique est un fichier spécial qui pointe vers un autre fichier ou un autre répertoire. Si le fichier pointé est effacé, le lien symbolique pointe vers un fichier inexistant. La plupart des commandes utiliseront le fichier ou répertoire pointé lorsque l'on tentera d'accéder au lien symbolique.</td></tr>
</table>A noter qu'on ne peut pas créer de liens physiques vers un répertoire et qu'un lien physique exige que les deux noms se trouvent sur le même système de fichier (partition, disque, ...)</td></tr>
<tr><td>touch</td><td>Met à jour l'heure d'un fichier. Si le fichier n'existe pas, crée un fichier vide. Il suffit d'utiliser touch nomdefichier.</td></tr>
<tr><td>cat</td><td>Affiche le contenu du ou des fichier(s) fourni(s) en argument(s). cat test.txt affichera le contenu du fichier test.txt. A noter que si le fichier prend plus d'une page, il défilera à l'écran</td></tr>
<tr><td>more</td><td>Affiche le contenu d'un fichier page par page. On utilise Enter pour passer à la ligne suivante et Espace pour passer à la page suivante, q permet d'arrêter l'affichage. Si aucun fichier n'est fourni, affiche page par page l'entrée standard.</td></tr>
<tr><td>tail</td><td>Affiche la fin d'un fichier. l'option -f permet de suivre le fichier (afficher les lignes au fur et à mesure qu'elles sont ajoutées)</td></tr>
</table>

I.4.c Les redirections et les filtres

Il est possible de rediriger les entrées et sorties d'une commande vers un fichier ou une autre commande.

commande > fichier	Ecrit la sortie de la commande dans le fichier. Si le fichier existait, il sera vidé avant l'opération
commande >> fichier	Ajoute la sortie de la commande à la fin du fichier. Si le fichier n'existe pas, il sera créé
commande < fichier	Utilise le contenu du fichier comme entrée pour la commande. Un programme en C lisant stdin recevra le contenu du fichier
commande1 | commande 2	Envoie la sortie de la commande1 comme entrée pour la commande 2. Cela permet de chaîner plusieurs commandes
commande 2> fichier	Envoie les messages d'erreur (stderr) de la commande vers le fichier.
commande > fichier 2>&1	Envoie la sortie de la commande vers le fichier. Les messages d'erreurs seront envoyées vers le même fichier

Le symbole | (pipe) permet de chaîner plusieurs programmes. Cette fonctionnalité est très souvent utilisée sous Unix. en effet, plutôt que de faire des outils complexes, Unix se base sur des outils qui font des tâches simples mais le font de façon efficace et sans erreurs. On peut chaîner plusieurs commandes et envoyer le résultat final dans un fichier par une redirection. Quelques filtres courants sont repris dans la table ci-dessous :

more	Permet d'afficher l'entrée page par page
sort	Permet de trier les lignes par ordre. On peut ajouter l'option -u pour supprimer les lignes en double, -f pour ignorer les différences majuscules/minuscules, -n pour comparer en ordre numérique, -r pour l'ordre inverse
cut	Permet de ne garder qu'un morceau de la ligne. -c permet de sélectionner des intervalles de caractères à garder, -f des intervalles de champs. Dans le cas de -f, -d permet de préciser le caractère qui sépare les différents champs. Les intervalles de champs sont de la forme *début-fin*, plusieurs intervalles sont séparés par des virgules (,). `cut -d: -f1,3-5,7` permet de garder les champs 1, 3,4,5 et 7 d'une ligne où les champs sont séparés par des : (on compte à partir de 1) *-fin* permet de prendre les champs ou caractères jusqu'à *fin* et *debut-* du caractère ou champ *debut* à la fin de la ligne
grep	Permet de ne garder que les lignes contenant le texte passé en paramètre. l'option -i permet d'ignorer les différences majuscules/minuscules. -v permet de ne garder que les lignes ne contenant PAS le texte passé en paramètre. `grep -v OK` permet de ne pas garder les lignes contenant *OK*
wc	Compte le nombre de caractères, mots et lignes. l'option -l permet de n'afficher que le nombre de lignes.

On peut par exemple utiliser la commande `cat /etc/passwd | cut -d: -f 1 | sort`

pour afficher la liste des utilisateurs du systèmes triés par ordre alphabétique.

Si on ajoute `| wc -l` en fin de commande, on affichera le nombre de comptes créés au lieu de la liste des comptes.

I.5 Editer un fichier

Sous Linux, il existe un très grand nombre d'éditeurs de fichiers en mode texte. Les principaux sont vi, nano, joe et emacs.

Si vi est sensé être le point commun à tous les Unix, il n'est pas particulièrement intuitif. C'est la raison pour laquelle joe a été préféré pour ce cours. Il est disponible dans la plupart des distributions Linux.

Emacs est un éditeur à tout faire. Outre le fait de permettre d'éditer des fichiers, il peut servir à lire ses E-Mails, à afficher un calendrier, servir de calculatrice, de lecteur de news, d'interpréteur LISP, ... Par contre, son utilisation requiert l'usage de nombreuses séquences de touches telles que Controle+x Controle+s, Meta+x, ... qui lui ont valu le surnom de Escape Meta Alt Control Shift.

Pour éditer un fichier avec joe, on appelle la commande `joe nomfichier`. Si le fichier n'existe pas, un nouveau fichier sera créé lorsque l'on sauvera le contenu de l'éditeur.

La première commande à retenir est Control+K H qui affiche la page d'aide. Escape . (point) permet de passer à la page d'aide suivante et Escape , (virgule) de revenir à la page précédente. Pour fermer l'aide, il suffit de refaire Control+K H (Par la suite, Controle+X sera abbrégé en ^X)

Pour éditer le contenu, il suffit d'utiliser les touches de curseur, de taper le texte que l'on veut insérer, d'utilisier Suppr et Backspace pour effacer les caractères de trop, ...

Pour quitter joe, on utilisera ^C (Control+C) pour quitter sans sauver ou ^KX (eXit) pour sauver et quitter. Si joe a été lancé sans nom de fichier, ^KX demandera le nom de fichier sous lequel on veut sauver le contenu de l'éditeur.

Si on désire sauver le fichier en cours d'édition, il suffit d'utiliser ^KD. Joe demandera le nom de fichier en proposant le dernier nom utilisé (il suffit de taper Enter si on ne désire pas changer le nom). Pour insérer le contenu d'un fichier à la position du curseur, il faudra utilser ^KR, joe demandera le nom du fichier à insérer.

Pour faire les couper/coller, il faut d'abord marquer le **B**loc**K** que l'on veut manipuler. On se place au début du bloc et on fait ^K**B**, à la fin et on fait ^K**K**. Il reste alors à se placer à la destination et faire ^KC pour Copier, ^KM pour déplacer (Move) ou ^KY pour effacer (Yank). ^KW permet de sauver le bloc dans un fichier (Write)

Joe possède plusieurs options de formattage qu'il peut être nécessaire d'activer ou de désactiver. Pour y accéder, il faut faire ^T. Les options apparaissent dans le bas de l'écran. Il suffit alors de taper sur la lettre correspondant à l'option que l'on désire changer. Les options sélectionnées apparaissent en haut de l'écran, avant le nom du fichier (I pour auto-indent, W pour Word Wrap, ...) Le plus souvent, on voudra désactiver le mode Word Wrap s'il est activé.

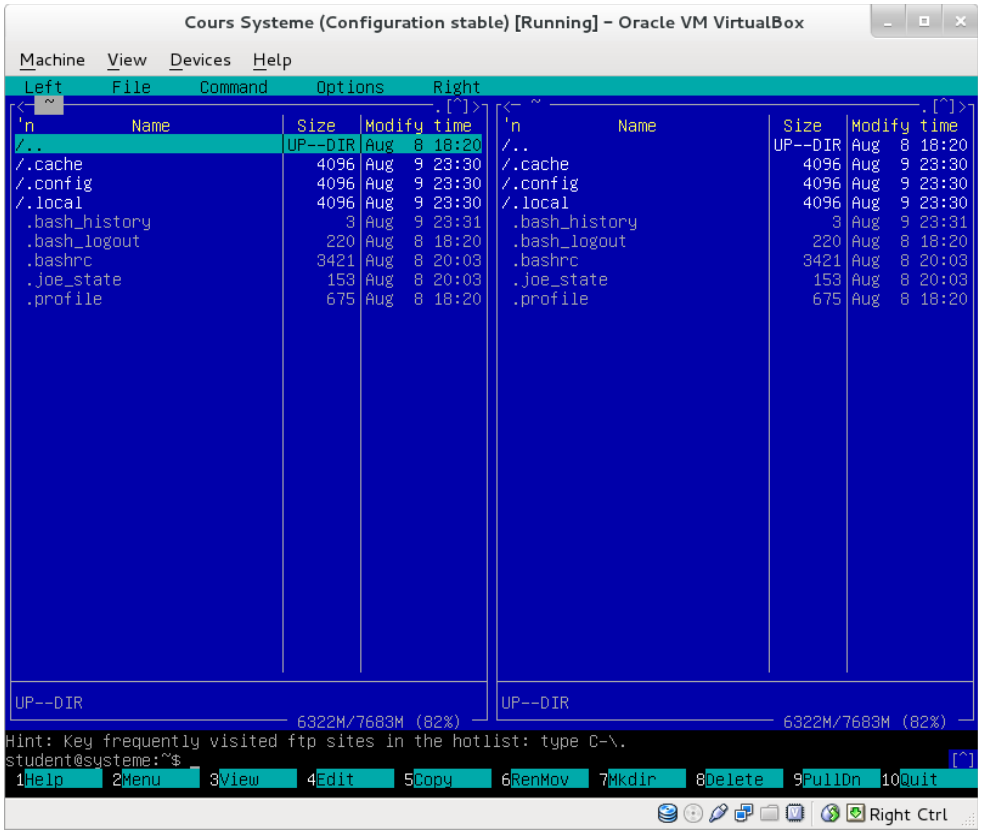
A noter que les options par défauts dépendent de l'extension du fichier créé. C'est ainsi que si on édite un fichier C, l'option W sera désactivée par défaut. Si on se rend compte qu'une ligne qui devient trop longue est scindée en deux, cela signifie que le Word Wrap est activé.

Si on désire faire un gros copier/coller dans joe venant de l'extérieur, il faut d'abord désactiver le AutoIndent (option I) si on ne désire pas passer son temps à effacer des espaces en début de lignes.

I.6 Midnight Commander

Midnight Commander est un outil semblable à Norton Commander sous MS DOS. Cet outil facilite la navigation dans les répertoire et la manipulation des fichiers et répertoires en mode texte. Il est d'un usage nettement plus pratique que la plupart des interfaces graphiques de gestion de fichiers, d'autant plus qu'il laisse l'accès direct à la ligne de commande.

On le lance simplement par la commande `mc`. Il se présente alors sous la forme reprise à la capture I.5.



Capture I.5: Midnight Commander

On dispose de deux panneaux. En haut de chaque panneau, on a le répertoire courant (ici ~, le répertoire de l'utilisateur). Le panneau actif pourra être identifié par le nom du répertoire courant et une des lignes dans le panneau apparaissant en vidéo inverse.

Pour changer le panneau actif, il suffit d'utiliser la touche TAB.

On peut déplacer le curseur du fichier (ou répertoire) dans le panneau actif par les flèches haut et bas.

Pour rentrer dans un sous-répertoire, il suffit d'amener le curseur sur son nom et d'utiliser la touche Enter. Le répertoire .. permet de remonter d'un niveau.

Pour copier/déplacer/effacer un fichier ou répertoire, il suffit d'amener le curseur sur son nom et d'utiliser F8 pour effacer ou F5 pour copier et F6 pour déplacer vers le répertoire sélectionné par l'autre panneau.

Pour renommer un fichier ou répertoire, on utilise F6 comme pour le déplacer mais on remplace la destination par le nouveau nom.

F3 permet de visionner le contenu d'un fichier et F4 permet de l'éditer. Par défaut, l'éditeur système

est utilisé mais une option permet d'utiliser l'éditeur de Midnight Commander[3].

La touche F7 permet de créer un nouveau répertoire. Il suffit d'entrer le nom du nouveau répertoire et de valider.

Si on désire copier, déplacer ou effacer plusieurs fichiers, il suffit de les sélectionner à l'aide de la touche Insert.

F10 permet de quitter Midnight Commander et Control+O permet de masquer temporairement les panneaux (pour pouvoir voir l'écran situé derrière) et de les réafficher.

On peut taper directement des commandes. Tant que la ligne de commande n'est pas vide, la touche Enter cesse de permettre la navigation et permet de valider la commande tapée.

Si on presse Enter en ayant un fichier exécutable sélectionné, cela a pour effet de le lancer sans paramètres.

Midnight Commander permet également de rentrer dans une archive ZIP, TGZ, ... comme s'il s'agissait d'un simple répertoire.

La touche F2 appelle un menu qui dépend de l'item sélectionné. Une des fonctions les plus pratiques est quand une archive (tar.gz, tar.bz2, tgz,...) est sélectionnée et permet de décompresser directement l'archive.

Il y a bien d'autres fonctionnalités disponibles, client FTP, recherche de fichier, changement des permissions, ... mais cela dépasse de cadre de cette introduction.

I.7 Les permissions

I.7.a Les mécanismes disponibles

Plusieurs systèmes de gestions des permissions existent sous Linux et sont utilisables simultanément :

- permissions standard Unix
- Les attributs des fichiers (immutable, append-only, ...)
- ACL (les permissions sont gérées par utilisateur/groupe, comme sous MS Windows)
- SELinux (la gestion des droits se fait non seulement au niveau utilisateur mais également au niveau application, à l'aide de règles définies au niveau du système) pour les systèmes sécurisés
- Les capacités, permettant de contrôler ce que peut faire une application (ou, plus précisément, de laisser tomber certaines permissions).

Dans le cadre de ce cours, seules les permissions standard Unix seront examinées.

I.7.b Comment fonctionnent les permissions de base ?

Chaque élément du système de fichier (fichier, répertoire, ...) possède trois attributs : un numéro d'utilisateur, un numéro de groupe et des drapeaux d'accès.

Lorsqu'un programme s'exécute, il possède un ensemble de 5 valeurs : effective user, effective group, real user, real group et la liste des groupes de l'utilisateur.

3 Cette option a été activée sur la VM qui vous est fournie.

Lorsqu'un utilisateur lance un programme, les effective et real user/group correspondront aux identifiants de l'utilisateur.

Lorsque le programme veut accéder à un fichier, on vérifie d'abord si l'effective user correspond au numéro d'utilisateur du fichier. Si c'est le cas, les permission de l'utilisateur seront utilisée. Sinon, on vérifie si le groupe du fichier est le effective groupe ou dans la liste des groupes de l'utilisateur. Si c'est le cas, les permissions du groupe seront utilisées. Si aucun des deux cas n'est rencontré, on utilise les permissions autres.

Les drapeaux de droits sont une valeur de 3 ou 4 chiffres en octal (base 8, des valeurs de 0 à 7). Les trois derniers chiffres sont dans l'ordre les droits de l'utilisateur, du groupe et des autres. Ainsi, une valeur de 2751 ou de 751 correspond à 7 pour l'utilisateur, 5 pour le groupe et 1 pour les autres.

Les droits utilisateurs/groupes/autres sont la somme de 4 pour les droits en lecture, 2 pour les droits en écriture et 1 pour les droits en exécution. Ainsi, la valeur 751 donne les droits en lecture, écriture, exécution pour l'utilisateur, lecture et exécution pour le groupe et juste exécution pour les autres.

Pour un fichier, le droit en exécution indique s'il s'agit d'un programme exécutable ou non, les autres indiquent si l'on peut lire le contenu du fichier ou le modifier.

Pour un répertoire, le droit en exécution correspond à la permission d'y accéder (et d'accéder aux fichiers qui sont contenus à l'intérieur). Le droit en lecture correspond à la permission de lister les fichiers présents dans le répertoire et le droit en écriture correspond à la permission de créer ou d'effacer des fichiers dans le répertoire (la modification des fichiers est contrôlée par les droits sur le fichier).

I.7.c SetUID/SetGID/Sticky

Le 4ème chiffre (au début) reprend le SetUID bit, le SetGID bit et le Sticky Bit.

Les trois bits spéciaux ont des significations différentes selons qu'il sont utilisés sur un répertoire ou un fichier.

Pour un répertoire, le SetGID bit indique qu'un fichier créé dans le répertoire aura le même groupe que le répertoire (si l'utilisateur a accès au groupe en question) et le Sticky-Bit indique que les fichiers contenus dans le répertoire ne peuvent être effacés que par l'utilisateur propriétaire ou par root (utilisé par exemple pour le répertoire /tmp pour éviter qu'un utilisateur n'efface les ficheirs temporaires d'un autre utilisateur).

Lorsqu'un fichier possède le SetUID bit (ou le SetGID bit), si un utilisateur exécute ce fichier, l'effective user (ou group) du programme auront pour valeur le propriétaire du fichier (ou son groupe). Les real user et real group seront eux toujours égaux à l'utilisateur qui a lancé le programme.

Des appels systèmes permettent d'échanger effective et real user (ou group) (et donc d'alterner entre les droits de l'utilisateur et ceux octroyés par le fichier) ou de fixer une des deux valeurs à l'autre (rendre le real user égal à l'effective user ou le contraire). Lorsque l'on change l'effective user vers le real user, l'ancien effective user est sauvé et il est possible de revenir à cette valeur sauvée.

I.7.d Utilisation en pratique

Dans le cadre de ce cours, nous n'utiliserons pas les SetUID/SetGID/Sticky bits et nous nous contenteront donc de valeurs de trois chiffres octaux. Dans un programme en C, il faudra les précéder du chiffre 0 pour que le compilateur sache qu'il s'agit d'une valeur octale (on utilisera ainsi 0751 et non 751).

Sur la ligne de commande, la commande chmod permet de changer les droits et accepte la valeur sans le 0 initial (chmod 751 monfichier). Seul le propriétaire du fichier (et root) peut changer les permissions d'un fichier.

L'utilisateur root (0) peut accéder à tous les fichiers sans tenir compte des droits qui sont fixés. La seule exception est le droit d'exécution sur les fichiers qui sert à indiquer s'il s'agit d'un programme exécutable ou d'un fichier de données.

Dans le cours, nous utiliserons généralement la valeur 666 pour un fichier de données en lecture/écriture (student est le seul utilisateur, dans un système réel, on utilisera plus souvent 664 ou 644 pour restreindre le droit en écriture ou 640 ou 600 pour également limiter les droits en lecture).

Pour les programmes et répertoires, nous utiliserons 777 (dans un système réel, on utilise plus souvent 775 ou 755)

Pour les fichiers de données en lecture seule, on utilisera 444 (dans un système réel, on utilise aussi 440 ou 400)

Lorsqu'un programme en C crée un fichier, la valeur est modifiée par le umask. Généralement, il vaut 002 ou 022, ce qui a pour effet de retirer les droits en écriture aux autres voire même au groupe.

I.7.e Quelques cas d'utilisation

Dans les Unix récent, chaque utilisateur possède un groupe personnel qui est son groupe principal et est membre d'une série de groupes dépendant des droits qu'il possède. Par exemple, dans la VM, l'utilisateur est membre du groupe vboxsf qui lui donne le droit d'accéder aux dossiers partagés entre la VM et la machine hôte.

I.7.e.i Répertoire de l'utilisateur

Le propriétaire et le groupe du répertoire personnel sont égaux à l'utilisateur et son groupe personnel. On peut donc donner des droits 770 (seul l'utilisateur peut y accéder) ou 775 (tout le monde peut y accéder en lecture).

I.7.e.ii Ressource à accès limité

Un groupe est créé pour la ressource (comme vboxsf). Les utilisateurs qui ont le droit d'utiliser cette ressource sont ajoutés au groupe en question. Les fichiers sont associés à ce groupe et les droits sont donnés au groupe mais pas aux autres (640,660, 664, 750, 770 ou 775 selon le cas).

I.7.e.iii Projet commun

Un groupe est créé pour le projet et les utilisateurs associés au projet y sont ajouté. Les répertoires liés à ce projet (et leurs sous-répertoires) ainsi que les fichiers qui s'y trouvent sont associés à ce groupe.

Les permissions sur les répertoires sont de 2770 ou 2775 (selon qu'on ne désire pas un accès public ou qu'on désire laisser un accès en lecture). Le SetGID bit permet de s'assurer que les nouveaux fichiers créés dans le projet seront également associés au groupe en question.

Les fichiers auront des droits 660 ou 664, les programmes 770 ou 775.

Afin que les nouveaux fichiers créés aient les droits corrects, le umask doit valoir 002 (à noter que de nombreuses distributions Linux utilisent une valeur de 022 par défaut, définie dans le fichier .profile)

I.7.e.iv Répertoire tunnel

Si un utilisateur désire rendre un répertoire accessible à certains utilisateurs, il peut créer un répertoire tunnel. ce répertoire aura des permissions 771 ou 711 qui permettront à tous de rentrer à l'intérieur mais empêcheront de voir son contenu.

Dans ce répertoire, il créera un nouveau répertoire dont le nom est le *mot de passe d'accès* et les droits seront 777 ou 775.

Les utilisateurs qui connaîtront le nom de ce répertoire pourront faire cd ~user/tunnel/secret pour se rendre dans le répertoire en question.

II Programmation sous Linux

II.1 GCC

Le compilateur C sous Linux s'appelle GCC. Ce compilateur est disponible pour la plupart des systèmes Unix sur différentes plateformes, permet la cross-compilation (compiler du code pour ARM à partir d'une machine équipée d'un processeur Intel par exemple) et fait partie d'un ensemble qui comprend compilateur C, C++, Objective C, ADA, Java, ...

La syntaxe de base pour compiler un fichier est

gcc -o *executable source.c*

Il est possible de compiler plusieurs fichiers en même temps. Il suffit de les préciser tous sur la ligne de commande. Cela permet de scinder le programme en plusieurs fichiers (par exemple, pour le rendre plus lisible).

Il supporte un bon nombre d'options qui permettent d'optimiser le code. Généralement, on peut se contenter de l'option -O2 ou -O3 mais il est possible d'activer manuellement des optimisations (ou des limitations) propres à un processeur spécifique ou considérées comme pouvant poser problème dans certains cas de figure.

Si on ne précise pas le fichier exécutable (-o fichier), gcc génèrera un fichier a.out. On peut par la suite renommer le fichier en question vers un nom plus correct

Si on désire, on peut activer les messages de Warning (qui n'empêchent pas la compilation mais permettent parfois de repérer des erreurs) en ajoutant les options -Wall -ansi

La compilation d'un fichier C passe par plusieurs étapes : préprocesseur (pour traiter les #include, #define, #ifdef, ...), transformation en assembleur, transformation en fichier objet et pour finir, liaison avec les librairies.

Dans le cas d'un gros projet composé d'un grand nombre de fichiers .c, on compilera les fichiers .c en objets .o avant de les lier. Comme ça, si un des fichiers du projet change, on ne devra recompiler que les fichiers qui en dépendent, ce qui sera plus rapide.

Si on appelle gcc avec l'option -MM, il retourne la liste des fichiers dont dépendent le fichier source passé en paramètre. La forme retournée est

gcc -MM source.c

source.o: source.c include1.h include2.h ...

Si on appelle avec un ensemble de fichier, la liste des dépendances est retournées. Il suffit de la rediriger dans un fichier par `gcc -MM *.c > .depend`

Il est possible de fournir des #define à partir de la commande gcc. Il suffit d'inclure un (ou des) paramètres -DNOM pour obtenir l'effet de #define NOM ou un paramètre -DNOM=valeur pour l'effet d'un #define NOM valeur. Attention cependant au fait que la commande gcc est interprétée par le shell et donc, si on désire placer des guillements, il faut les précéder d'un \

```
gcc -o test test.c -DMSG=\''message de test\''
```

II.2 Makefile

II.2.a Structure de base

Lorsque l'on désire compiler un ensemble de fichier, il devient vite pénible de compiler chaque élément à la main. On utilise alors l'outil make qui utilisera les règles présentes dans un fichier Makefile pour compiler le projet.

Ce fichier Makefile a une structure très simple :

```
cible: dépendance1 dépendance2 ...
      commande1
      commande2
      ...

cible2: dépendance1 ...
      commande1
      ...
```

On peut appeler la commande make en précisant le nom d'une cible ou seule. Dans ce dernier cas, la première cible sera utilisée.

Chaque règle défini une cible, ses dépendances et les commandes à exécuter pour générer la cible. Si une des dépendances est plus récente que la cible, les commandes seront exécutées.

On pourra ainsi utiliser une règle du genre

```
programme: source1.c source2.c
      gcc -c source1.c
      gcc -c source2.c
      gcc -o programme source1.o source2.o
```

II.2.b Règles implicites et macros

Le logiciel make permet cependant de simplifier la situation. en effet, il permet de définir des règles automatiques qui s'exécutent en fonction des extensions des fichiers et défini un ensemble de règles prêtes à l'emploi.

C'est ainsi qu'il existe une règle qui permet de compiler un fichier .c en .o

Le programme pourra donc être simplifié en

```
programme: source1.o source2.o
      gcc -o programme source1.o source2.o
```

Cela indique que pour compiler programme, on a besoin de source1.o et de source2.o. Comme aucune règle n'existe pour ces derniers, les règles implicites vont être utilisées et exécuter les commandes cc -c source1.c et cc -c source2.c

Mais le compilateur que l'on veut utiliser s'appelle gcc et non cc (même si un alias est défini au niveau du système). Il faut donc préciser le compilateur. En fait, make permet également l'emploi de macros et possède un ensemble de macros prédéfinies.

Le compilateur C est défini dans une macro CC, il suffit donc de la redéfinir dans notre Makefile pour corriger le compilateur utilise : CC=gcc

De même, une série d'autres macros permettent de fournir les options de compilation. La macro CFLAGS contient les options de compilation. On peut donc demander d'optimiser le code en utilisant CFLAGS=-O3

Le fichier devient donc

```
CC=gcc
CFLAGS=-O3
programme: source1.o source2.o
	gcc -o programme source1.o source2.o
```

Il est encore possible de simplifier le Makefile. En effet, lorsque l'on traite une règle, la valeur $+ contient la liste des dépendances, $? la liste des dépendances qui sont plus à jour que la cible, $< la première dépendance (généralement le fichier .c) et la valeur $@ contient la cible (il y a d'autres valeurs prédéfinies). On peut donc modifier la commande gcc en `gcc -o programme $+` ce qui permet de ne devoir reprendre la liste des fichiers qu'une seule fois.

Comme un Makefile peut contenir un grand nombre de règles, il peut être intéressant de définir ses propres macros et de les utiliser dans le code. On peut par exemple définir une macro qui contient la liste des fichiers nécessaires au programme.

```
CC=gcc
CFLAGS=-O3
OBJS=source1.o source2.o
EXE=programme

$(EXE): $(OBJS)
	gcc -o $@ $+
```

Maintenant, make sait comment transformer une liste de fichiers objets en un exécutable et on peut donc laisser tomber la dernière ligne.

Quelques définitions par défaut intéressantes :

Fichier objets vers exécutable	$(CC) $(LDFLAGS) $(TARGET_ARCH)
Fichier C++ vers objet	$(CXX) $(CXXFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
Fichier C vers objet	$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
Répertoires Includes	/usr/include /usr/local/include
Fichiers supplémentaires pour le lien vers un exécutable	$(LOADLIBES) $(LDLIBS)

II.2.c Un Makefile plus complet

Dans le cas d'un projet contenant plusieurs fichiers, il faut préciser les dépendances des différents fichiers sources par rapport aux includes du projet. Le format retourné par gcc -MM correspond déjà au format attendu par make. Il suffit donc de l'include dans le Makefile ce qui peut être réalisé très facilement.

De même, une règle qui ne possède aucune dépendance ou dont la cible ne correspond à aucun fichier créé s'exécute toujours. On peut donc prévoir des règles pour des tâches telles que le nettoyage après compilation, la création d'une archive avec les fichiers du projet, ...

Il est possible de masquer une erreur en précédant la commande par un - (par exemple, dans make clean, on peut précéder le rm d'un tiret pour éviter les erreurs si les fichiers à effacer n'existent pas)

On aura donc un Makefile amélioré comme le suivant :

```
CC=gcc
CFLAGS=-O3
OBJS=source1.o source2.o
EXE=programme
INSTDIR=/home/student/bin

$(EXE): $(OBJS)

depend:
	gcc -MM *.c > .depend

clean:
	-rm *.o


install: $(EXE)
	cp -f $< $(INSTDIR)

-include .depend
```

Pour compiler le programme, il suffira de faire make depend ; make

Ce fichier peut évidemment être modifié selon les besoins, par exemple pour compiler plusieurs exécutables, pour ajouter des options de compilation ou de liaison (par exemple, des répertoires include ou des librairies supplémentaires).

Un dernier changement permet d'éviter que, si l'on crée un fichier appelé clean, install ou depend (sans le point au début), make clean, make install ou make depend arrête de fonctionner. On utilise la directive `.PHONY : clean` pour indiquer que clean ne correspond pas à un fichier. Cette déclaration doit se trouver avant la cible concernée. On ajoutera donc, après les définitions de variables, la ligne suivante :

```
.PHONY : depend clean install
```

II.3 Pages de manuel

En cas de doute sur une fonction du langage C, on peut appeller la page de manuel qui s'y rapporte. Cette page indiquera les #include à ajouter et rappellera la syntaxe de la fonction correspondant.

Pour appeler une page de manuel, on utilise l'instruction

`man` *`[section] page`*

La section indiquera la catégorie de pages de manuels. Par exemple, la page open existe dans la section 1 (commande open qui permet d'ouvrir un terminal virtuel) et dans la section 2 (fonction C open()). Les pages qui nous intéressent pour la programmation se trouvent dans les catégories 2 (fonctions système) et 3 (fonctions C). La catégorie 1 reprend les commandes du shell et la catégorie 8 les commandes systèmes. On représente souvent la section entre parenthèses

open(2) pour `man 2 open`

III Accès fichiers de bas niveau

Lorsque l'on désire travailler avec des fichiers spéciaux (pipes, devices, fichiers virtuels, ...), il est important que les accès aux fichiers soient effectués de façon directe.

Hors, les fonctions classiques du C, fopen, fprintf, fscanf, ... ne font ces accès que de façon indirecte, en implémentant des buffers qui permettent de diminuer le nombre d'accès au noyau et donc, de rendre le programme plus rapide[4].

On devra donc utiliser un autre ensemble de fonction, qui est également particulièrement adapté à la manipulation de fichiers au format binaire. Il est important de ne pas mélanger les deux types de fonctions sur les mêmes fichiers. On peut cependant utiliser printf/scanf pour gérer la console (clavier/écran) et utiliser les appels de bas niveau pour les fichiers.

*III.1 File descriptor vs FILE **

Lorsque l'on utilise les fonctions fopen/..., le fichier est représenté par une structure FILE. Cette structure reprend les informations sur le buffer et sur le fichier en cours d'utilisation.

Pour les appels de bas niveau, on utilisera un entier que l'on appellera *file descriptor* et qui est l'index dans la table des fichiers ouverts du programme en cours d'exécution. Ce nombre doit donc toujours être supérieur ou égal à zéro et une valeur négative représente une erreur.

La structure FILE reprend un file descriptor pour le fichier concerné (en interne, les fonction fprintf/fscanf,... utilisent les fonctions de bas niveau). Même s'il est possible d'obtenir le file descriptor en question, il ne faut jamais l'utiliser pour lire, écrire ou repositionner le pointeur dans le fichier.

Les valeurs de 0 à 2 ont des utilités particulières : 0 correspond à stdin, 1 à stdout et 2 à stderr. Si on utilise les instructions printf/scanf/gets ou les périphériques stdin/stdout/stderr avec les fonctions fprintf, fscanf, ... il ne faut en aucun cas utiliser ces descripteurs.

III.2 Erreurs

Lorsqu'une erreur est déclenchée, de nombreux appels systèmes retournent une valeur spéciale (souvent négative) et le code d'erreur est sauvé dans une variable globale nommée `errno`.

L'include errno.h déclare la variable pour que l'on puisse l'utiliser à partir du C et crée une série de #define qui définissent le sens de chaque erreur (/usr/include/asm-generic/errno.h et errno-base.h pour la liste)

III.3 open/close

Pour ouvrir un fichier, on utilisera la commande **open(2)** .

```
#include <fcntl.h>   // pour open()
#include <sys/stat.h> // pour les #define des droits d'accès
#include <sys/types.h> // pour le type mode_t (équivalent à int)

int fd=open("fichier", O_RDWR | O_CREAT,0644) ;
```

4 Le passage du mode « programme utilisateur » au mode « noyau » est souvent bien plus lent qu'un simple appel de sous-programme

Le premier paramètre est le nom du fichier, le second est le OU (|) entre les différentes options et le troisième, les droits d'accès au fichier (quand il est créé).

Les drapeaux DOIVENT contenir une et une seule des valeurs suivantes : O_WRONLY (écriture seule), O_RDONLY (lecture seule) ou O_RDWR (lecture/écriture).

Ils peuvent en plus contenir les options suivantes : O_APPEND (se positionne à la fin du fichier pour ajouter), O_CREAT (si le fichier n'existe pas, le crée), O_EXCL (avec O_CREAT, échec si le fichier existait déjà), O_TRUNC (le fichier est ramené à une taille de 0), O_NONBLOCK (les accès ne sont pas bloquants)

Les droits d'accès ne sont utilisés que lorsque le fichier doit être créé et sont masqués par la valeur de umask[5]

Lorsque l'on n'a plus besoin d'un fichier, on peut le fermer à l'aide de la fonction

```
close(fd) ;
```

A noter que les fichiers qui n'ont pas été fermés le sont automatiquement quand le programme s'arrête.

Lorsque l'instruction **open** ouvre un fichier, il utilise la première valeur disponible pour le file descriptor.

III.4 read/write

Pour lire ou écrire dans un fichier ouvert à l'aide d'un file descriptor, on utilise les commandes **read(2)** et **write(2)**. Elles demandent l'include

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

Ces commandes prennent en paramètres le file descriptor, le pointeur vers un buffer et la taille du bloc à lire ou écrire. Elles retournent le nombre de caractères lus ou écrits ou une valeur négative en cas d'erreur.

```
int fd,nr ;
char buf[1024] ;
fd=open("fichier.data",O_RDWR) ;
nr=read(fd,buf,1024) ;
write(fd,buf,nr) ;
close(fdà ;
```

Le code ci-dessus ouvre un fichier en lecture, lit au maximum 1024 octets dans un buffer et écrit à la suite les octets lus (soit en réécrivant les octets à partir du 1025ème, soit en rajoutant à la fin du fichier). Si le fichier fait moins de 1024 caractères, le nombre de caractères lus se trouve dans la variable nr.

Si on est arrivé à la fin du fichier (ou si le pipe est vide, ...), la commande **read** retournera une valeur de 0.

III.5 lseek

Pour positionner le pointeur de lecture/écriture dans le fichier, on utilise la commande **lseek(2)**. Cette commande demande les includes suivants :

```
#include <sys/types.h> // pour le type off_t
```

5 Cette valeur, définie par la commande shell umask, indique quels sont les bits de droit à forcer à zéro. Une valeur de 002 forcera le bit "écriture pour les autres" à zéro (mais il peut être changé par la suite à l'aide d'un chmod)

```
#include <unistd.h>    // pour la commande et les constantes SEEK_SET, ...
```

La commande demande trois paramètres : le file descriptor, l'offset voulu et la position à partir de laquelle prendre l'offset en compte. Elle retourne la position à laquelle on se trouve dans le fichier en octets depuis le début du fichier ou une valeur négative en cas d'erreur. A noter que cette commande utilise un entier 32 bits pour se positionner et est donc limitée aux fichiers de moins de 2Gb[6]

La position peut être SEEK_SET (offset par rapport au début du fichier), SEEK_CUR (offset par rapport à la position actuelle) ou SEEK_END (offset par rapport à la fin du fichier).

On peut également utiliser la commande **lseek** pour déterminer la position actuelle du pointeur dans le fichier, en utilisant SEEK_CUR et un offset de 0.

III.6 dup/dup2/fcntl/ioctl

Les commandes **dup(2)** et **dup2(2)** permettent de créer une copie d'un file descriptor. Les deux file descriptors partageront le même fichier et la même position dans le fichier. Par contre, ils ne partageront pas les options de file descriptor

Ces commandes peuvent être utilisées pour transférer un file descriptor vers une des valeurs spéciales (0, 1 et 2) ou disposer d'options différentes. L'instruction **dup** demande juste le file descriptor à dupliquer, **dup2** prend en second paramètre la nouvelle valeur désirée. Ces fonctions retournent la nouvelle valeur (ou une valeur négative en cas d'erreur).

La commande **fcntl(2)** prend en premier paramètre le file descriptor, en second un code de commande et peut accepter des paramètres supplémentaires pour les paramètres de la commande. Elle permet entre autre de lire ou écrire les drapeaux de file descriptor (close on exec et autres), certains drapeaux du fichiers (fournis lors du open), de gérer des verrous, de gérer un envoi de signal quand il redevient possible de lire ou écrire, ...

La commande **ioctl(2)** permet de configurer un device (un device s'ouvre comme un fichier). Le premier paramètre est le file descriptor, les paramètres suivants dépendent du type de device. Cette commande permet par exemple de changer la fréquence d'échantillonnage d'un périphérique /dev/audio, d'ouvrir le tiroir d'un lecteur de CD-ROM, ...

III.7 select/FD_CLR/FD_SET/FD_ISSET/FD_ZERO

La commande **select(2)** permet de surveiller plusieurs file descriptors et d'indiquer ceux qui deviennent "prets" (données disponibles en lecture, buffer permettant une écriture,...). A noter qu'un socket réseau est également un file descriptor et peut donc également être surveillé par **select** (tout comme les pipes, les devices, ...)

Select demande les includes suivants :

```
#include <sys/times.h> // pour la structure timeval
#include <sys/types.h> // pour les types de données tels que les fd_set
#include <unistd.h> // pour la commande select en elle même

#include <sys/select.h> // POSIX ne demande que cet include au lieu des trois
                        ci-dessus
```

Le premier paramètre de select est la valeur du plus haut file descriptor utilisé augmenté de 1.

Les trois paramètres qui suivent sont des **fd_set** qui contiennent respectivement les file descriptor à

6 Il existe une commande lseek64 pour les fichiers plus grands.

vérifier pour la lecture, l'écriture et les exceptions. Ces **fd_set** seront modifiés pour ne plus reprendre que les file descriptors qui permettent une lecture/écriture ou ont une exception.

Le quatrième paramètre fourni le timeout. Un timeout de 0 permet de juste vérifier quels file descriptors peuvent être utilisés, un pointeur NULL indique de bloquer indéfiniment. En sortie, la structure est modifiée pour contenir le temps restant[7].

Les **fd_set** sont gérés au travers des instructions FD_ZERO, FD_SET, FD_CLR et FD_ISSET.

La fonction retourne zéro si le timeout est expiré, une valeur négative en cas d'erreur et le nombre de file descriptor activés dans les **fd_set** si au minimum un file descriptor est devenu disponible.

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

fd_set rfds,wfds,efds ; // fd_set en lecture, écriture, exception
struct  timeval tv ;
int ret ;
FD_ZERO(&rfds) ; // On vide les fd_set
FD_ZERO(&wfds) ;
FD_ZERO(&efds) ;
FD_SET(42,&rfds) ; // on écoute sur le file descriptor 42 (normalement, la
                     variable retournée par open) en lecture
FD_SET(56,&wfds) ; // on écoute sur le fd 56 en écriture
FD_SET(42,&efds) ; // on écoute les exception sur les FD 42 et 56
FD_SET(56,&efds) ;

tv.tv_sec=10 ;    // Timeout de 10 secondes et 0 µs
tv.tv_usec=0 :

ret=select(57,&rfds,&wfds,&efds,&tv) ; // on attend qu'un FD soit prêt
if (ret<0) { // erreur
} else {
      if (ret) { // ret>0 indique qu'un FD est pret
            if (FD_ISSET(42,&rfds)) { // le FD est prêt en lecture
            }
            // on teste les autres fd_set
      } else { // Le timeout est expiré
      }
}
```

A noter que si un des fd_set n'est pas nécessaire, il peut être remplacé par NULL. On utilise parfois `select(0,NULL,NULL,NULL,&tv);` pour attendre une durée inférieure à la seconde.

A noter également que dans le cas où les buffers se remplissent ou se vident lentement, les lectures/écritures suite au select peuvent ne contenir que quelques octets. Il peut donc être nécessaire de récupérer ou envoyer les données "par morceaux" et en un seul **read** ou **write**.

7 Attention, la norme POSIX PERMET ce fonctionnement mais ne l'impose pas. Certains Unix ne modifient pas le timeout en sortie de select

IV Process

Sous Linux, chaque tâche lancée s'appelle un *process*. Un process est désigné par un numéro unique sur la machine[8].

A priori, les différentes tâches sont complètement isolées les unes des autres, ce qui permet d'éviter qu'un programme avec des erreurs ou problèmes de sécurité ne perturbe les autres. Il existe cependant plusieurs mécanismes (qui seront vu plus loin dans le cours) qui permet de faire communiquer plusieurs process.

IV.1 Process et shell

Il existe quelques commandes qui permettent de manipuler les process

Commande **&**	Le & en fin de commande permet de lancer la commande en arrière plan. Le shell rend immédiatement la main. Une tentative de lecture sur le terminal va cependant bloquer le process
ps auxwww	Affiche la liste des process. La première colonne donne le nom du propriétaire du process, la seconde donne nle numéro de process, la troisième, la charge machine, les trois suivantes concernent l'utilisation de la mémoire, la 7eme indique le terminal concerné, la 8eme le status (Running, Sleeping, Zombie), suivent l'heure de lancement du process et le temps processeur qu'il a utilisé et, enfin, la ligne de commande[9]
kill *signal* n°process	Envoie un signal (par défaut, -TERM = "terminate", mais on peut préciser -KILL (ou -9) pour tuer de façon inconditionnelle ou autres) à un process. Permet généralement d'arrêter un process
killall *signal* nomprocess	Permet d'envoyer un signal à tous les process dont le nom d'exécutable est celui passé en argument

A noter que, à part root, un utilisateur ne peut envoyer un signal qu'aux process qui lui appartiennent.

Quand un programme est lancé en avant plan, il est possible de l'interrompre à l'aide de Ctrl+Z (^Z) et de lui dire de continuer en arrière plan à l'aide de la commande `bg`. Quand un process en arrière plan s'arrête parce qu'il a besoin d'une entrée, on peut le relancer au premier plan à l'aide de la commande `fg`.

IV.2 Créer un nouveau process en C

Pour créer un nouveau process, on utilise la fonction **fork(2)**. Cette instruction demande l'include <unistd.h>

Cette fonction retourne -1 en cas de problème, 0 pour le process enfant et le numéro du process enfant au process parent. Le process enfant est une copie conforme du process parent, la seule chose qui permette de les distinguer est donc la valeur retournée par fork.

8 Certains systèmes permettent de s'assurer que le numéro de process soit unique sur un ensemble de machines montées en cluster

9 Une application peut cependant changer la ligne de commande qui apparaît

La façon habituelle de procéder est la suivante :

- le process parent utilise **getpid(2)** pour obtenir son numéro de process (pid) et le sauver dans une variable *parentid*
- le process parent utilise **fork(2)** et sauve le résultat dans une variable *forkresult*
- les deux process testent la valeur de la variable forkresult.
    - Si elle vaut 0, on est dans le process fils, getpid() permet d'obtenir le pid du fils et parentid contient le pid du père.
    - Si elle est >0, on est dans le process père. Parentid contient le pid du père et forkresult celui du fils.
    - Si elle est <0, on a une erreur

```
#include <sys/types.h>   // pour pid_t, on utilise souvent int à la place
#include <unistd.h>   // pour fork et getpid
int parent,res;
parent=getpid();
res=fork();
if (res>0) {
      /*on est dans le  père,  ID père = parent, ID fils = res */
} else if (res == 0) {
      /* on est dans le fils, ID  père = parent, ID fils = getpid() */
} else {
      /* on a une erreur */
}
```

Mais on trouve souvent aussi (quand on a pas besoin des pid)

```
if (fork()) {
      /* process père */
} else {
      /* process fils */
}
```

Lorsqu'un nouveau process est créé, il est une copie conforme du process parent : même code, mêmes variables, les files descriptors correspondent aux mêmes fichiers (si on effectue un lseek d'un côté, il affecte les deux process par exemple).

IV.3 Signaux entre process

Pour envoyer un signal d'un process à un autre, on utilisera la fonction **kill(2)** qui prend en paramètres le numéro de process et le numéro de signal. Les noms des signaux et la fonction sont définis dans l'include <signal.h> (il faut également <sys/types.h> pour le type pid_t)

Les signaux souvent utilisés sont

SIGHUP	HangUP (raccroché), souvent utilisé pour forcer le rechargement de la configuration (par défaut, termine le programme)
SIGKILL	Tue le process (ne peut pas être intercepté, ignoré ou masqué)
SIGUSR1, SIGUSR2	Disponible pour l'utilisateur
SIGTERM	Fin (normale) de programme

Un programme peut intercepter un signal et lui associer une fonction. Cela permet par exemple de sauver les données avant d'arrêter le programme.

Pour ce faire, il existe deux interfaces : l'interface **signal(2)**, plus ancienne, moins complète mais plus simple et l'interface **sigaction(2)** qui est conseillée.

IV.3.a Signal(2)

Cette fonction demande l'include <signal.h>. Elle prend deux paramètres : le numéro du signal et le nouveau gestionnaire de signal. Elle retourne l'ancien gestionnaire de signal.

Le gestionnaire de signal peut-être SIG_IGN pour ignorer le signal, SIG_DFL pour placer le gestionnaire par défaut (voir signal(7)) ou un pointeur vers une fonction qui prend un entier en entrée et ne retourne rien, qui sera le nouveau gestionnaire de signaux.

Un des problèmes avec l'instruction signal(2) est que son comportement varie d'un système à l'autre. Quand un signal est reçu, soit le gestionnaire est remis automatiquement à SIG_DFL, soit les signaux de ce type sont bloqués jusqu'à la fin du gestionnaire. Dans les deux cas, il est plus prudent de remettre le gestionnaire en place en fin de fonction.

```
#include <signal.h>
void gestsig(int a) {
      /* Gestionnaire de signal */
      signal(SIGTERM,gestsig) ;
}
signal(SIGTERM,gestsig); /* Permet de mettre un gestionnaire sur SIGTERM */
```

IV.3.b sigaction(2)

Cette fonction prend trois paramètres : un numéro de signal et deux pointeurs vers des structures sigaction. La première ce ces structures contient la nouvelle valeur et la seconde sera remplie avec l'ancienne valeur. Une valeur de retour négative indiquera une erreur.

La structure sigaction possède les champs suivants :

sa_handler : pointeur vers le gestionnaire de signal (même type de fonction que pour signal) ou SIG_IGN/SIG_DFL comme pour signal

sa_mask : un masque des signaux qui doivent être bloqués pendant le traitement du gestionnaire de signal. (le signal qui est détourné est automatiquement ajouté à la liste)

sa_flags : quelques drapeaux qui changent le comportement et peuvent être ignorés ici.

Pour gérer le sa_mask, on utilisera **sigemptyset(3)** pour mettre le masque à 0, **sigaddset(3)** pour ajouter un signal à la liste et **sigdelset(3)** pour en enlever un. Le premier paramètre est le sa_mask. Les deux dernières fonction prennent un second paramètre qui est le numéro de signal.

IV.4 Reap Childs

Lorsqu'un process fils se termine, le process père reçoit un signal SIGCHLD. Le process père peut alors lire le status de retour (valeur retournée par exit(3). Tant que cela n'a pas été fait, le process reste à l'état de Zombie.

Il est donc nécessaire d'installer un gestionnaire pour le signal SIGCHLD qui lise le status en question afin que le process Zombie puisse être libéré. Pour cela, il faut utiliser une des fonctions **wait(2)** ou **waitpid(2)**.

wait prend en paramètre un pointeur vers un entier, attend qu'un process enfant se soit terminé et sauve le status dans la variable fournie.

waitpid prend un pid, un pointeur vers un entier et des drapeaux.

Si le pid vaut -1, il attend qu'un des fils (n'importe lequel) se termine, une valeur >0 attendra qu'un process fils avec le pid fourni se termine.

waitpid retournera le pid du process dont on aura récupéré le status.

L'option peut valoir WNOHANG, ce qui forcera waitpid à retourner directement avec une valeut non nulle si un process fils a retourné son status.

On utilise souvent la boucle suivante dans un gestionnaire de SIGCHLD pour nettoyer les zombies :

```
void handle_sigchld(int sig) {
  while (waitpid((pid_t)(-1), 0, WNOHANG) > 0) {}
}
[...]
struct sigaction sa;
sa.sa_handler = &handle_sigchld;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART | SA_NOCLDSTOP;
sigaction(SIGCHLD, &sa, 0);
```

La boucle permet de nettoyer plusieurs Zombies si le cas se présente. Les drapeaux de sigaction permettent de s'assurer que le gestionnaire de signal sera bien conservé et permet de ne pas envoyer SIGCHLD quand il s'agit d'un événement de mise en pause/redémarrage du process enfant (^Z, fg, bg)

A noter que si SIGCHLD est mis explicitement à SIG_IGN, les Zombies ne seront pas créés.

IV.5 alarm(2)

La fonction **alarm(2)** permet de déclencher un signal SIGALRM un certain nombre de secondes (passé en argument) plus tard. Une seule alarme peut être active à la fois, un nombre de secondes de 0 permet de stopper l'alarme.

La valeur retournée est le nombre de secondes qui restaient avant l'alarme programmée (ou 0 s'il n'y en avait pas).

Certains unix peuvent utiliser la fonction alarm pour implémenter la fonction sleep... il est donc conseillé de ne pas mélanger les deux.

V Les pipes

Le pipe est un mécanisme qui permet de transférer des données entre deux process. Le pipe est une file de type FIFO dans laquelle un process écrira et l'autre lira.

Il existe deux types de pipes : les pipes nommés, créés sur le système de fichier et les pipes anonymes. Les accès aux pipes se feront à l'aide des fonctions **read(2)** et **write(2)** vues précédemment.

V.1 Pipes anonymes

La fonction **pipe(2)** permet de créer un pipe. Elle prend en paramètre un tableau de deux entiers et écrira dans le premier le file descriptor en lecture et dans le second celui en écriture.

Il suffira ensuite d'effectuer un fork()[10] et de fermer dans un process la moitié en lecture et dans l'autre, la moitié en écriture.

La commande **pipe2(2)** prend un paramètre de plus, des drapeaux de configurations pour le pipe. Cette commande est une extension GNU à la norme POSIX et demande un #define pour être activée.

Les fonctions pipe et pipe2 retourneront 0 si tout se passe bien et -1 en cas d'erreur. On peut trouver un exemple de code dans la page de manuel de **pipe(2)**.

La fonction pipe2(pipefd,O_NONBLOCK) ; correspond à utiliser flg=fcntl(pipefd[0],F_GETFD); pour lire les drapeaux du file descriptor suivi de fcntl(pipefd[0],F_SETFD,flg | O_NONBLOCK); pour activer l'option NONBLOCK (également sur pipefd[1]). On peut donc s'en passer si on désire rendre le code plus portable vers d'autres systèmes UNIX. Voir la manpage de **open(2)** pour les détails sur O_NONBLOCK.

Une tentative de lire sur un pipe alors que l'autre extrémité (écriture) est totalement fermée (dans tous les process qui y ont accès) retournera une valeur de 0.

V.2 pipes nommés

Ces pipes sont créés soit à partir du shell à l'aide de la commande **mkfifo(1)**, soit à l'aide de la fonction C **mkfifo(3)**.

Une fois créé, il pourra être ouvert comme un fichier. L'ouverture en lecture bloquera jusqu'à ce qu'un autre process aura ouvert le pipe en écriture et vice versa.

Si le pipe est ouvert en écriture seule avec l'option O_NONBLOCK et qu'il n'a pas encore été ouvert en lecture, une erreur ENXIO sera déclenchée (open retournera -1 et errno contiendra ENXIO). S'il est ouvert en lecture avec l'option O_NONBLOCK, la commande **read(2)** retournera 0 s'il n'y a aucune information disponible en lecture.

Comme le pipe nommé apparaît comme un fichier, il est possible de l'ouvrir comme si c'était un fichier normal. Par exemple, on peut faire un `cat /home/student/mon_pipe` pour ouvrir le pipe en lecture et afficher son contenu.

V.3 Utilisation des pipes

Un pipe est un mécanisme unidirectionnel. Si on désire une communication bidirectionnelle, il faudra donc en ouvrir deux (un pour chaque sens de conversation).

10 La commande fork() duplique le process, y compris les file descriptors ouverts, dont celui du pipe.

Les pipes peuvent être utilisés pour un système client/serveur. Si on désire que plusieurs clients puisses se connecter, il faut prévoir la création d'autant de pipes de sortie du serveur que nécessaire. A cet effet, le client enverra dans le paquet qu'il écrira dans le pipe une information qui indiquera le nom du pipe de retour (par exemple, le client peut envoyer son process ID et on peut utiliser un pipe client_*pid* pour le retour). Le client se chargera de créer le pipe de retour et de l'effacer (**unlink(2)** qui prend le chemin du fichier à effacer) avant de s'arrêter.

Généralement, le côté écriture sera ouvert en mode bloquant. Le côté lecture pourra par contre être crée en mode O_NONBLOCK mais cela ne permettra plus de détecter la fermeture de l'autre extrémité (la valeur de 0 indiquant aussi bien la fermeture que l'absence de données à lire).

On pourra utiliser la fonction select(2) pour attendre que des informations soient disponibles en lecture sur un pipe ou qu'il soit possible d'écrire sur un pipe. A noter que cette commande permet également de surveiller stdin (dont le file descriptor vaut 0).

Si toutes les extrémités ouvertes en lecture sont fermées, une tentative d'écriture déclenchera un signal SIGPIPE. Si ce signal est ignoré, l'écriture échouera avec un code d'erreur de EPIPE.

La page de manuel **pipe(7)** donne des informations complémentaires sur les pipes.

VI Mémoire partagée

Le mécanisme décrit ci-dessous est celui défini dans les IPC System V[11]. Il permet de demander une zone de mémoire partagée et de l'accrocher dans l'espace du processus en cours.

Chaque segment de mémoire partagée possède un id (généré par l'OS lors de sa création) et une clé. Une clé spéciale indiquera que le segment est privé (seuls les process qui disposent de son id peuvent y accéder).

Des commandes shell permettent de lister et d'effacer les segments de mémoire partagée (shm : Shared Memory) mais aussi les sémaphores (sem : SEMaphores) et les files de messages (msg : MeSsaGe queue).

VI.1 Commandes shell

La commande **ipcs(1)** permet de lister les ressources ipc system V : shm, sem et msg. Par défaut, les trois types sont affichés.

Pour chaque bloc IPC, on disposera de la clé, de l'ID, du propriétaire et des permissions. Pour les segments de mémoire partagée, on disposera également de la taille en octets, du nombre d'attachements en mémoire du segment (tous process confondus) et du status. Pour les sémaphores, on disposera du nombre de sémaphores dans le bloc et pour les files de messages, du nombre d'octets utilisés et du nombre de messages.

Lorsque l'on voudra effacer des segments qui n'auraient pas été nettoyés par une application, on utilisera ipcs pour obtenir l'ID du segment de mémoire partagée concerné et on pourra ensuite l'effacer à l'aide de la commande **ipcrm(1)**.

La commande ipcrm supporte deux syntaxes. L'ancienne utilisait une commande de la forme

```
ipcrm shm id
```

et la nouvelle forme utilise l'option -m id pour effacer un segment à partir de son id ou -M clé pour l'effacer à partir de sa clé (si elle n'est pas IPC_PRIVATE). Cette nouvelle forme est l'utilisation recommandée de ipcrm

```
ipcrm -m id
```

VI.2 Programmation en C

A commande shmget(2) permet de récupérer l'id d'un segment de mémoire partagée à partir de sa clé ou d'en créer un.

Pour récupérer un segment existant, on utilisera (la taille est ignorée)

```
shmid=shmget(clé,taille,0) ;
```

Pour récupérer un segment s'il existe ou le créer s'il n'existe pas, on utilisera (les droits sont définis comme pour les fichiers)

```
shmid=shmget(clé,taille,IPC_CREAT | droits) ;
```

11 A un moment, il existait deux branches principales de UNIX : la branche BSD (d'où émanent FreeBSD, OpenBSD, NetBSD mais aussi MacOS/X) et la branche System V (System 5) principalement présente chez SUN (SunOS, Solaris). Depuis, la norme POSIX a fait converger les UNIX vers un tronc standard, même s'il existe encore quelques différences.

Pour créer un segment privé, on utilisera la forme suivante (on peut ajouter IPC_CREAT aux droits comme pour un segment non privé) :

```
shmid=shmget(IPC_PRIVATE,taille,droits) ;
```

Un segment privé devra être créé avant le fork() afin que les différents process puissent disposer de sa clé.

Lorsque le segment est créé, une structure shmid_ds sera initialisée avec le uid et le gid du process qui l'aura créé, les permissions transmises (juste les 9 bits de permission), la taille du segment et l'heure actuelle (heure de création). Les autres champs sont initialisés à 0.

En cas d'erreur, la valeur de retour vaudra -1 et errno indiquera la raison de l'erreur (EACCESS pour un problème de droits, EINVAL pour une taille trop petite (SHMMIX), trop grande (SHMMAX) ou plus grande que la taille du segment déjà existant, ENOMEM pour indiquer qu'il ne reste pas assez de mémoire libre, ...)

Une fois l'identifiant du segment de mémoire partagée obtenu, il est nécessaire de l'attacher en mémoire pour pouvoir y accéder. Cela se fait à l'aide de l'instruction

ptr=shmat(shmid,NULL,0) ;

Après cet appel, le pointeur ptr pointe vers le début de la mémoire en question, le contenu peut être lu et écrit (à noter qu'il est possible de préciser qu'un segment doit être attaché en lecture seule).

Lorsque l'on n'a plus besoin du segment au niveau du process, on peut le détacher par l'instruction suivante :

shmdt(ptr);

Lors du fork, le nouveau process récupère les segments attachés par son process père.

Si le shmat échoue, le process retourne (void *)-1. shmdt retourne la valeur -1 en cas d'erreur.

Chaque fois que le segment est attaché, il peut obtenir une adresse différente. Il faut donc éviter de sauver des pointeurs dans ce segment (ils n'auraient de sens que pour le process qui les aura sauvés) mais on peut par contre sauver un offset à ajouter à la valeur de ptr.

La commande shmctl(2) permet de récupérer ou modifier les informations de shmid_ds ou d'indiquer que le bloc de mémoire partagée doit être effacé lorsque le dernier process le détachera.

Pour indiquer que le segment doit être effacé au dernier shmdt, on utilisera

shmctl(shmid,SHM_RMID,NULL) ;

Attention que certains UNIX ne permettent plus d'attach une fois le segment marqué pour effacement. Il est donc préférable de ne le faire que avant que le dernier shmdt ne soit exécuté. Si on ne marque pas un segment comme devant être effacé, il sera nécessaire de l'effacer à l'aide de ipcrm.

VII Les sémaphores

VII.1 Définition

Le sémaphore est un mécanisme qui accepte deux opérations P() et V()[12]

- P(SEM) qui sert à tester le sémaphore. Si la valeur du sémaphore est supérieure à 0, elle est diminuée de un. Sinon, le process sera mis en attente jusqu'à ce que la valeur aie été augmentée

- V(SEM) qui sert à incrémenter le sémaphore. Cette opération peut toujours être effectuée.

Ces deux opérations sont dites "atomiques" : pendant qu'elles s'exécutent, aucune autre opération ne peut être faite sur les sémaphores concernés.

La valeur d'un sémaphore sera donc toujours supérieure ou égale à 0 (un P() qui amènerait le sémaphore à une valeur inférieure à 0 sera bloquée).

P() est parfois appelée "Down" ou "Take" et V() est parfois appelée "Up" ou "Release" dans la documentation. Lorsque l'on crée le sémaphore, on lui donne une valeur initiale. Par la suite, le sémaphore n'est plus sensé être manipulé que par les fonctions P() et V().

Les sémaphores permettent de gérer les principaux problèmes de synchronisation de process : accès à une zone critique (un seul process peut accéder à une ressource à la fois), synchronisation (problème du rendez-vous, tous les processus doivent arriver au point de rendez-vous avant d'aller plus loin), problème des producteurs/consommateurs (un ou des process produit des ressources, un ou plusieurs autres les consomment), problème des lecteurs/rédacteurs (un ou plusieurs process peuvent accéder en lecture, un peut accéder en écriture et empêche les accès en lecture).

VII.2 Les fonctions C pour gérer les sémaphores

VII.2.a Créer un bloc de sémaphores

Pour créer un bloc de sémaphores, on utilise la fonction **semget(2)**. Cette fonction travaille d'une façon semblable à shmget(2) et prend comme paramètres la clé, le nombre de sémaphores du bloc et les drapeaux et droits. On utilisera généralement

```
id=semget(clé ou IPC_PRIVATE, nbre semaphores, IPC_CREAT | 0666) ;
```

Dans le cas où on a utilisé IPC_PRIVATE, un bloc privé (qui ne peut être obtenu par une clé) est créé et il faudra faire le fork() après le semget(), afin que les deux process disposent de l'id du bloc de sémaphores.

Les sémaphores créés ainsi ont une valeur indéterminée

VII.2.b Actions sur les sémaphores (P() et V())

La fonction **semop(2)** permet d'effectuer un ensemble d'opérations sur les sémaphores. Ces actions seront effectuées dans l'ordre si et seulement si elles peuvent toutes être effectuées. Généralement, on se contentera d'une seule opération (P() ou V()) sur un seul sémaphore.

La fonction prend trois paramètres : l'id du bloc de sémaphores, un pointeur vers une ou des structures indiquant les opérations à effectuer et le nombre de structures présentes.

La structure reprend trois valeurs : sem_num qui est le numéro du sémaphore, sem_op qui

12 Proberen (tester, essayer) et Verhogen (incrémenter).

indiquera l'opération à faire et sem_flg qui pourra contenir les valeurs SEM_UNDO (si le process se termine, défaire toutes les opérations sur sémaphore effectuées par ce process) et/ou IPC_NOWAIT (si l'opération est sensée bloquer, retourne -1 et errno vaudra EAGAIN).

sem_op	Fonction	Remarques
>0	V()	La valeur du sémaphore est augmentée de sem_op. Pour la commande V() normale, sem_op vaudra 1
<0	P()	La valeur du sémaphore est diminuée de la valeur absolue de sem_op. Pour la commande P() normale, sem_op vaudra -1
0	WAIT-ZERO	Attend que le sémaphore arrive à une valeur nulle

```
struct sembuf sb;
sb.sem_num=0; /* Premier sémaphore */
sb.sem_op=1; /* V() */
sb_sem_flg=0;
res=semop(semid,&sb,1);

struct sembuf sba[2];
sba[0].sem_num=1;
sba[0].sem_op=-1; /* P() */
sba[0].sem_flg=0;
sba[1].sem_num=2;
sba[1].sem_op=-2; /* P() qui diminue le sémaphore de 2 au lieu de 1 */
sba[1].sem_flg=0;
res=semop(semid,sba,2);
```

Si un semop est interrompu par un signal, il retournera -1 et errno vaudra EINTR (interrupted). Les opérations ne seront pas effectuées.

VII.2.c Modifications d'un sémaphores

La fonction **semctl(2)** permet différentes actions sur un sémaphore. Le premier paramètre sera l'id du bloc de sémaphores, le second, le numéro de sémaphore dans le bloc (parfois ignoré), le troisième sera la commande à effectuer et le quatrième dépendra de l'opération (il peut être omis pour certaines opérations).

Cmd	Commande	Paramètre	Valeur de retour
IPC_RMID	Effacer le bloc de sémaphores		
GETALL	Retourne les valeurs des sémaphores	Adresse d'un tableau d'unsigned short	
GETVAL	Retourne la valeur du nième sémaphore		Valeur du sémaphore
SETALL	Change les valeurs de tous les sémaphores	Adresse d'un tableau d'unsigned short contenant les valeurs des sémaphores	
SETVAL	Change la valeur d'un sémaphore	Entier : nouvelle valeur	
GETNCNT	Nombre de process en attente sur un P() pour le nième sémaphore		Nombre de process bloqués sur des

			semop<0 sur le sémaphore
GETZCNT	Nombre de process en attente de sémaphore =0 sur le nième sémaphore		Nombre de process bloqués sur des semop=0 sur le sémaphore

Lorsque l'on crée des sémaphores, leur valeur est indéterminée. Il sera donc nécessaire d'effectuer des SETVAL ou un SETALL pour initialiser les sémaphores avant de les utiliser.

VII.3 Zone critique (MutEx)

Le problème de la zone critique peut être décrit comme suit : on désire qu'un et un seul process puisse accéder à la fois à une ressource.

Cette ressource peut être une variable partagée, afin d'éviter que entre la lecture et la réécriture de cette variable par un process, cette variable n'aie été modifiée par un autre process. Cela peut également être un ensemble de valeurs sur lesquelles plusieurs opérations doivent être faites avant de réutiliser la zone de mémoire, ...

Le terme de MutEx fait référence à l'aspect exclusif de l'accès : Mutually Exclusive (l'un ou l'autre mais pas les deux). Ce terme est souvent utilisé pour un objet qui n'est capable que de gérer cet accès précis d'un sémaphore.

Pour gérer un MutEx avec un sémaphore, on initialise le sémaphore à 1 (le nombre de process qui peuvent le prendre simultanément). Lorsqu'un process désire entrer dans la zone critique, il prend le sémaphore (P()). En prenant le sémaphore, la valeur de ce dernier est amenée à 0, ce qui empêchera qu'un second process puisse prendre le sémaphore pour accéder à la zone critique. Une fois que le process quitte la zone critique, il libère le sémaphore (V()) ce qui le ramènera à 1 et permettra donc à un autre process de le prendre.

VII.4 Producteur-Consommateur

Un ensemble de process peuvent "produire" des blocs de données que d'autres process peuvent "consommer". On peut par exemple imaginer une situation style "travail à la chaîne" où une série de process s'enchaînent pour décomposer le travail. Un premier process peut lire les données du fichier, les découper en petits morceaux qu'un (ou des) process traitera avant de les transmettre à un dernier process qui assemblera le fichier final.

On peut exploiter l'aspect bloquant des sémaphores afin de ne pas devoir tester en boucle si de nouvelles données sont disponibles. On initialise le sémaphore à 0, chaque fois qu'une donnée sera produite, on l'augmentera (V()) et, avant d'en utiliser une, on le testera (P()). La valeur du sémaphore sera égale au nombre de données disponibles et s'il n'y en a aucune, le P() bloquera jusqu'à ce qu'une donnée exploitable soit déposée.

Une variante est le cas où la place dans la file est limitée, on initialise le sémaphore au nombre d'entrées libres, avant d'utiliser un emplacement de la file, on prendra le sémaphore (P()) et quand on libérera un emplacement, on l'incrémentera (V()). On peut imaginer que le sémaphore contient le nombre de cellules libres et que l'on commence par en créer un certain nombre (lors de l'initialisation) et, par la suite, elles seront consommées (on en rempli une) ou produite (on en libère une).

Souvent, on utilisera les deux facettes en même temps, avec un sémaphore indiquant le nombre de

données présentes dans la file et un second indiquant le nombre d'emplacements libres.

VII.5 Le problème du rendez-vous

Plusieurs processus doivent attendre d'avoir tous atteint le point de rendez-vous avant de pouvoir aller plus loin. Si on désire synchroniser N processus, une solution est d'utiliser N sémaphores initialisés à 0. Lorsqu'un processus arrive au point de rendez-vous, il libérera N-1 fois le sémaphore qui lui correspond et puis attendra sur chacun des N-1 autres sémaphores.

Une autre solution utilise deux sémaphores et une variable :

```
Initialisation : count = 0, S1=1, S2=0
Rendez-vous :
  P(S1)  // mutex
    count++
    if (count==N)  V(S2) ;
  V(S1)  // fin mutex
  P(S2)
  V(S2)
```

Le mutex protège l'accès à la variable commune. Chaque process qui arrive au point de rendez-vous augmentera le compteur de 1 et se mettra en attente sur S2. Lorsque le dernier arrivera au point de rendez-vous, count sera égal à N et S2 sera libéré. Un des process sera donc débloqué et libérera le suivant par l'instruction V(S2) qui suit le P(S2), et ainsi de suite jusqu'à ce qu'ils aient tous été libérés.

VII.6 Problème des lecteurs/rédacteurs

Les process doivent accéder à une ressource partagée. D'une part en lecture, d'autre part en écriture.

On peut avoir un nombre quelconque de process qui accèdent à la ressource en lecture en même temps. Par contre, si un process y accède en écriture, il doit être le seul à y accéder.

```
Initialisation : count=0, S1=1, S2=1
Rédacteur :
  P(S1)
    utilisation
  V(S1)
Lecteur :
  P(S2)
    count++
    if (count==1) P(S1)
  V(S2)
    lecture
  P(S2)
    count--
    if (count==0) V(S1)
  V(S2)
```

S1 sert de "mutex" à l'utilisation lecture/écriture. Il n'est changé qu'une fois au niveau lecture (lorsque le premier lecteur accède à la ressource ou le dernier cesse de lire)) mais à chaque rédacteur (pour s'assurer qu'il n'y aie qu'un seul rédacteur).

VII.7 Problème du Deadlock

Lorsque l'on tente de prendre un sémaphore (P()), le process peut être mis en attente qu'un autre process libère le sémaphore en question.

Si on a deux process A et B et deux sémaphores S1 et S2, la situation suivante peut arriver :

A	B	S1=1, S2=1
P(S1)		A prend S1
	P(S2)	B prend S2
P(S2)	P(S1)	A essaye de prendre S2 mais est bloqué, B essaye de prendre S1 mais est bloqué
V(S2)	V(S1)	On n'arrive pas à ces lignes qui permettraient de débloquer les process
V(S1)	V(S2)	

Les deux process A et B se bloquent mutuellement et définitivement. On parle de deadlock (verrou mortel)

Si par contre les process A et B prennent les sémaphores dans le même ordre (d'abord S1 puis S2), la situation ne peut pas se produire. En effet, une fois que l'un des process aura pris S1, l'autre ne pourra pas le prendre et donc, à plus forte raison, prendre S2.

En général, si on dispose d'un ensemble de n sémaphores S1, S2, ... , Sn, on peut éviter les situations de deadlock si tous les process prennent les sémaphore dans l'ordre numérique.

Si un process Pa est bloqué sur un sémaphore Si, ce sémaphore est pris par un autre process Pb qui ne doit plus prendre que des sémaphores dont le numéro est supérieur à i.

Si ce process Pb est lui même bloqué sur un sémaphore Sj, ce sera pas un process Pc qui ne doit plus prendre que des sémaphores de numéro supérieur à j (et donc, pas Pa), ...

Comme le nombre de sémaphore est limité à n, on arrivera a un moment à un process qui ne pourra pas être bloqué (il devrait être bloqué par un sémaphore dont le numéro serait supérieur à n). Ce process pourra donc libérer les sémaphores qu'il avait pris et ainsi de suite jusqu'à ce que Pa soit débloqué.

Dans le cas où on a un problème de producteur/consommateur, il faut s'assurer qu'un process en attente sur un sémaphore pour "consommer" ne bloque pas de sémaphores mutex qui seraient nécessaires pour que le producteur puisse libérer le sémaphore de producteur/consommateur. Le P() d'un producteur-consommateur doit donc se trouver hors de toute zone critique. De même, le producteur ne peut libérer le sémaphore qu'une fois la donnée effectivement disponible et donc, après les zones critiques.

VII.8 Le problème du dîner des philosophes

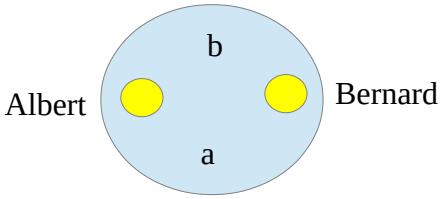
Autour d'une table, on dispose *n* philosophes. Comme ces derniers parlent beaucoup, on décide de faire des économies au niveau des couverts et de ne mettre qu'un couvert entre deux philosophes. Chaque philosophe pour manger devra prendre le couvert qui est à sa gauche et celui qui est à sa droite[13].

Comment faire pour être sur que tous les philosophes puissent manger (un philosophe qui a fini son assiette ne prend plus aucun couvert).

Si on considère deux philosophes, Albert et Bernard, on a deux couverts que nous appellerons a et

13 On peut donc imaginer qu'ils sont chinois et mangent avec des baguettes puisque les couverts sont "interchangeables".

b. Si albert prend le couvert à sa droite (a) et Bernard le couvert à sa droite (b), ils se retrouvent tous deux avec un seul couvert en main et ne savent manger ni l'un, ni l'autre, attendant chacun que l'autre dépose le couvert qu'il a pris.



Par contre, si Albert commence par prendre le couvert à sa droite mais Bernard le couvert à sa gauche, le plus rapide prendra le couvert a et l'autre attendra. Si le plus rapide est Albert (A), il pourra prendre le couvert b tout à son aise et une fois qu'il les déposera, Bernard pourra manger.

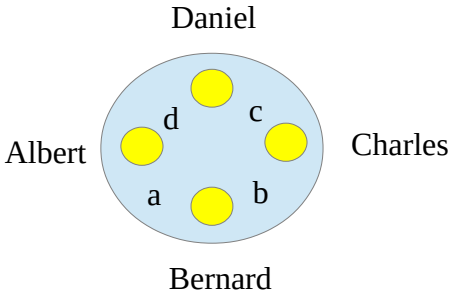
De même, si B prend le couvert a en premier, il pourra prendre le couvert b (A attend le couvert a) et manger.

Si celui qui a pris les couverts les dépose avant d'avoir fini, on revient à la situation de départ. S'il a fini, les deux couverts sont disponibles pour l'autre.

Si on prend quatre philosophes (Albert, Bernard, Charles et Daniel), avec les couverts a,b,c et d, le même raisonnement sera possible.

A et B tenteront de prendre d'abord le couvert a puis respectivement le couvert d ou b. C et D tenteront de prendre d'abord le couvert c puis respectivement le couvert b ou d.

Une fois que deux des philosophes auront leur premier couvert, deux cas de figure peuvent se présenter. Soit ils sont diamétralement opposés (A et C ou B et D) et leur deuxième couvert sera libre (A prend les couverts a et d, C prend les couverts c et b). Soit ils sont l'un à côté de l'autre et ce



sera au plus rapide pour prendre le second couvert (A et D voudront le couvert d, B et C voudront le couvert b)

Dans le second cas, lorsque celui qui aura pris le second couvert le déposera, l'autre pourra le prendre, manger et déposer les deux couverts. On est donc sur qu'ils pourront tous les quatre manger.

Le même raisonnement pourra être fait pour n'importe quel nombre pair de philosophes. Si le nombre est impair, le raisonnement fonctionnera aussi (on aura un philosophe qui ne sera pas en compétition avec un autre pour prendre son second couvert)