Table des matières

Chapitre 1

Introduction

Dans ce chapitre d'introduction, nous allons commencer par tenter de définir ce qu'est un système d'exploitation, puis nous allons décrire à grand traits les premières étapes du démarrage d'un ordinateur, depuis la mise sous tension jusqu'au lancement du premier processus.

Une fois le chapitre assimilé, vous devriez être en mesure de diagnostiquer l'origine du problème dans un ordinateur qui ne démarre pas : problème matériel ou problème logiciel, problème avec le boot loader ou avec le noyau du système.

1.1 Définition d'un système d'exploitation

Je vous demande d'arrêter tout de suite de suivre le cours, et de passer quelques instant à réfléchir à ce qu'est un système d'exploitation avant de lire notre propre définition. Écrivez votre définition pour pouvoir vous y référer par la suite. (De plus, le fait d'écrire la définition encourage à être plus précis que si on se contente de la penser.)

Si vous n'avez pas encore écrit votre définition de ce qu'est un système d'exploitation, vous avez encore une chance. Si vous passez au paragraphe suivant sans l'avoir fait, il sera trop tard.

Maintenant que vous avez écrit une opinion sur ce qu'est un système d'exploitation, je peux vous dire que moi-même, je ne sais pas trop bien ce que c'est. La seule caractéristique d'un système d'exploitation à peu près établie d'entrée de jeu, c'est qu'il s'agit de logiciel ; avec un peu de mauvaise foi, on peut cependant y trouver des contre-exemples. Je vais expliquer pourquoi il me semble que la question est difficile, puis je vais vous donner quelques éléments de définition.

1.1.1 Pourquoi la question est difficile

En premier lieu, il convient de se méfier des questions de la forme *« Qu'est-ce qu'un x ? »*, quel que soit le x, parce qu'elles demandent de délimiter exactement une chose qui n'a en général pas de limite exacte.

Pour reprendre un paradoxe classique, si nous savons faire la différence entre un têtard et une grenouille, en revanche il n'y a pas un instant précis avant lequel l'animal était un têtard et après lequel il est une grenouille : la transformation de l'un en l'autre est graduelle ; donc trouver une définition précise qui nous permette de définir un têtard et qui n'inclut pas une grenouille est impossible.

Toute réponse à une demande de définition précise court le risque d'être trop générale, et d'inclure des objets qui n'appartiennent pas à ce qu'on tente de définir. Exemple : *« Un système d'exploitation, c'est du logiciel »*. Oui, sans doute, mais il y a toute sorte de logiciels qui ne sont pas des systèmes d'exploitation.

Toute réponse court aussi le risque d'être trop limitative, et d'exclure des objets qui appartiennent à ce qu'on veut définir. Exemple : *« Un système d'exploitation, c'est Linux »*. Oui, en partie, mais il y a des systèmes d'exploitation qui ne sont pas Linux ; il y a aussi Solaris ; j'ai même entendu dire que du coté de Seattle une entreprise américaine vendait un système d'exploitation pour les PC.

1.1.2 Une façon d'aborder le problème

Une façon d'aborder le problème consiste à poser la question de savoir ce qui, dans les logiciels d'un ordinateur que nous connaissons, relève du système d'exploitation.

Exercice 1.1 — Pour chacun des composants suivants, estimez-vous qu'il fait partie du système d'exploitation ?

–TeX
–Firefox
–Dia/Xfig
–Emacs/gedit/vi/ed
–Ls
–Cp/mv/rm
–clisp
–gcc
–gdb
–Pwd
–Bash
–Gnome/kde

–X11

Dans la même veine, on peut aussi chercher des exemples types de logiciels qui appartiennent au système d'exploitation et des exemples types de logiciels qui n'appartiennent pas au système d'exploitation.

1.1.3 Exemples de systèmes

Une autre façon d'aborder la question consiste à énumérer tous les systèmes d'exploitation, et de tenter de trouver leurs points communs.

Distributions Linux
Variantes d'Unix
CP-M, MS-DOS et Windows jusqu'à 95
Windows NT et suivants
Systèmes temps réels Qnx, etc.
Systèmes embarqués SymbianOS/Android

1.1.4 Ce qui n'est pas un système d'exploitation

1.1.5 Une définition quand même

Finalement, à la lumière de la discussion qui précède, je vais vous donner une définition à peu près standard de ce qu'est un système d'exploitation, et vous expliquer pourquoi elle ne me semble pas trop satisfaisante.

Définition : *« Un système d'exploitation, c'est le logiciel de base qui permet l'utilisation d'un ordinateur. »*

La définition est à peu près inattaquable, parce qu'elle contient un flou suffisant pour recouvrir celui de la limite entre le système et le reste ; si on tente de définir un *logiciel de base*, ou ce qu'est *l'utilisation d'un ordinateur*, on retombe sur les problèmes de précision de la définition évoqués plus haut.

Peut-être que c'est là que se situe une vraie réponse complète à la question : ce qui apparaîtra à un utilisateur comme un *logiciel de base* peut être une application pour un autre. Celui qui n'utilise sa machine que pour surfer sur le Web aura tendance à considérer son browser comme un logiciel de base ; pour celui qui utilise l'ordinateur pour programmer, le browser et le compilateur sont deux applications qui ne font pas partie du système d'exploitation.

1.2 Le boot d'un ordinateur

Dans cette section, nous allons examiner ce qui se passe quand un ordinateur démarre, dans l'intervalle qui sépare la mise sous tension et le lancement du premier programme.

1.2.1 Mise sous tension, premières instructions

Lors de la mise sous tension de l'ordinateur, ses différents composants sont initialisés ; en général les circuits intégrés complexes ont une patte sur laquelle on peut envoyer un signal pour commander leur initialisation.

Le microprocesseur est l'unité qui exécute les instructions. Son initialisation va placer dans le PC (le *compteur ordinal*) une valeur qui sera l'adresse des premières instructions qu'il commence à exécuter. Ces adresses sont contenues

dans une ROM (une *mémoire morte*) qui conserve ses valeurs même quand elle n'est pas alimentée en électricité.

(Sur les ordinateurs plus anciens, il n'y avait pas de mémoire morte pour le démarrage et le processeur ne commençait pas à exécuter des instructions, mais démarrait dans un état d'attente ; on entrait des valeurs dans la mémoire à l'aide de boutons présents sur la face avant (des clefs) ; cela s'appelait *programmer aux clefs* ; on plaçait de cette manière dans la mémoire les premières instructions à exécuter.)

Sur les PCs, les instructions contenues dans la ROM s'appellent le BIOS (comme *Basic Input Output System*) ; elles vont commencer par tester les différents composants de l'ordinateur, de manière à vérifier leur présence et leur bon fonctionnement. Cette étape s'appelle le POST comme *Power On Self Test* (auto-test de mise sous tension) ; sur les PC, on voit couramment cette étape lors du test de la mémoire présente, sous la forme d'un compteur dont la valeur augmente à mesure que le processeur découvre la mémoire (il vérifie sa présence en y écrivant des valeurs et en s'assurant qu'il peut les relire). Il y a également des paramètres stockés dans une mémoire RAM alimentée par une batterie, pour qu'elle conserve ses valeurs quand le courant est coupé ; on appelle cette mémoire le CMOS (du nom de la technologie couramment utilisée pour cette mémoire). Sur les PCs, il est en général possible d'entrer à ce moment dans une interface d'examen et de modification des valeurs contenues dans cette mémoire, souvent via la touche de fonction *f2*, la touche *suppr* ou la touche *esc*.

A la fin de cette étape, le BIOS affiche, parfois pendant très peu de temps, un résumé des périphériques dont il a détecté la présence, puis il passe à la phase suivante.

1.2.2 Chargement du boot-loader

Une fois le matériel présent dans l'ordinateur découvert par le BIOS, l'ordinateur effectue son *bootstrap*, pour placer en mémoire le logiciel du système d'exploitation.

Le terme *bootstrap* signifie un *tirant de botte* et fait allusion à une des aventures (imaginaires) du baron de Münchausen, qui pour se sortir d'un trou tira avec force sur ses propres bottes, ce qui lui permit de s'élever en l'air. Le mot est passé dans le jargon courant de l'informatique ; on emploie en français le verbe *booter* (prononcer comme *bouter*) plus souvent que *bootstraper* qui s'emploie quand même). Pour dire qu'on va éteindre et redémarrer un ordinateur, on emploie aussi la verbe *rebooter*.

Le boot va choisir un périphérique avec lequel démarrer : il s'agit fréquemment d'un disque dur, parfois d'un CDROM ou d'un DVD ou d'un périphérique USB ou d'une carte réseau. En général, l'interface avec le BIOS permet de paramétrer ce périphérique sous la forme d'une liste ordonnée de périphériques à essayer d'utiliser, conservée dans le CMOS.

Sur les disques durs, le BIOS lit les 512 premiers octets qui forment ce qu'on appelle le MBR (*Master Boot Record* : enregistrement principal de démarrage), les place en mémoire et commence l'exécution des instructions qu'il est censé contenir.

Le code contenu dans le MBR est couramment appelé le *boot loader* (chargeur de boot), ou *boot primaire*. Comme il est petit (moins de 512 octets), il ne peut pas faire grand chose.

Sur les CDROM, les DVD et les périphériques USB, le mécanisme est à peu près le même : le code du BIOS découvre le périphérique, lit les premiers octets qu'il contient et lance leur exécution. Comme le code pour accéder aux périphériques USB est passablement complexe, seuls les BIOS récents (en 2008) permettent de booter dessus.

1.2.3 Boot réseau

Les machines qui n'ont pas de disque peuvent booter sur le réseau, mais les choses sont un peu plus compliquées parce que les cartes réseau ont des interfaces de commande plus variées que les disques, et ce code n'est pas présent dans le BIOS. Certaines cartes réseaux contiennent une ROM avec le code nécessaire interroger le réseau pour découvrir un serveur et télécharger du code depuis celui-ci. Le protocole le plus couramment utilisé pour cela en ce moment s'appelle PXE (comme *Pre-boot eXecution environment*).

Le BIOS ne détecte pas si une carte réseau est bootable ou pas et le réseau n'apparaît en général pas dans la liste des périphériques de boot. Ce qui se passe à la place, c'est que le BIOS exécute du code présent sur la carte (s'il est présent) qui demande en général une confirmation au clavier avant de tenter de booter sur le réseau.

1.2.4 Le MBR, le multi-boot

Dans les disques durs, il est courant de découper le disques en *partitions* : chaque partition se comporte comme un disque indépendant du point de vue de l'utilisateur.

Le Master Boot Record contient une table des partitions, qui décrit au maximum quatre partitions (dites *partitions principales*) et va choisir l'une d'entre elles pour booter ; sous les systèmes d'exploitation Microsoft, la partition choisie est celle qui est marquée comme *active*. On peut modifier la partition active avec le programme de manipulation des partitions `fdisk`.

Les boot-loaders utilisés sous Unix permettent en général de choisir la partition depuis laquelle on souhaite booter, avec une partition par défaut qui est choisie si l'utilisateur n'a rien fait au bout de quelques secondes ; cela permet d'avoir un ordinateur qui peut booter sous plusieurs systèmes. On appelle cela le *multi-boot*.

Une fois la partition choisie, le boot-loader va charger le *boot record* qui se trouve aussi sur le premier secteur de la partition (la partition se comporte comme un disque, elle commence elle aussi par un enregistrement de boot). On appelle cela du *chain loading* ; c'est de cette manière que les boot loaders de Linux permettent de démarrer aussi d'autres systèmes d'exploitation.

1.2.5 Les boot loaders Linux

Sur Linux, les deux boot loaders les plus couramment utilisés sont Lilo et Grub ; leur rôle est de charger le *noyau* du système d'exploitation, qui va ensuite résider dans la mémoire et contrôler l'exécution de tous les autres programmes jusqu'à l'arrêt de l'ordinateur.

Lilo et Grub lisent le noyau sur le disque, dans un fichier stocké d'une manière ordinaire sur une partition ; or le code pour analyser la structure d'un système

de fichier est trop complexe pour être contenue dans le peu de place disponible dans un *boot record*. Ils adoptent deux stratégies distinctes.

Lilo

Pour charger le noyau, Lilo va trouver, à un endroit convenu à l'avance, la liste des endroits sur le disque (les secteurs) où se trouve le fichier qui contient le noyau. Cela signifie qu'il est nécessaire de mettre cette liste des secteurs à jour chaque fois qu'on touche au noyau (soit qu'on en installe un nouveau, soit qu'on le recopie). Cette mise à jour est faite en lançant, sous Linux, la commande `Lilo` qui consulte le fichier de configuration (en général nommé `/etc/lilo.conf` ou `/etc/lilo/lilo.conf`), reconstruit et remet en place la liste des secteurs. *Attention, si on modifie le fichier qui contient le noyau sans relancer la commande* `Lilo`*, il n'y aura plus moyen de rebooter.* Le fichier `lilo.conf` contient en général une entrée de sauvegarde, qui permet de booter sur une copie du noyau du système si on a modifié le fichier qui contient le noyau sans relancer la commande `Lilo` avant de rebooter.

Grub

Grub est le boot loader utilisé normalement sur la distribution Mandriva. Il fonctionne en chargeant en mémoire quelque chose de plus sophistiqué que Lilo, qui est capable de découvrir les fichiers stockés sur le disque. Il n'y a donc pas besoin de reconstruire la liste des blocs après chaque modification du fichier qui contient l'image du noyau.

Les fichiers de grub sont stockés dans le répertoire `/boot/grub`. On y trouve notamment le fichier `stage1` (premier étage) qui est celui qui sera recopié dans le boot record, divers fichiers dont le nom se termine par `stage1_5` qui contiennent des versions de l'étage un et demi différentes pour chaque type de système de fichier qu'on peut rencontrer sur le disque. Le fichier `stage2` contient le second étage du boot de Grub. Dans `menu.lst` on trouve la description du menu à afficher au boot.

1.2.6 Le noyau du système

Le coeur des systèmes Unix est contenu dans un seul fichier, qui contient un seul programme très particulier qu'on appelle le noyau du système d'exploitation (on emploie souvent le terme anglais de *kernel*). Ce programme est chargé en mémoire au démarrage et y réside jusqu'à l'arrêt de l'ordinateur.

Pour souligner l'unité du noyau, on parle de noyau *monolithique*; d'autres systèmes d'exploitation ont un noyau découpé en programmes indépendants qui ensemble jouent le rôle du noyau; on parle alors de systèmes à *micro-noyau* (et non de pépins, même si c'est ce que sont les micro-noyaux; l'expérience laisse penser que cela décrirait bien leur utilisation dans de nombreux cas). Le plus célèbre des systèmes à micro-noyau s'appelle `Mach`; il est notamment à la base du système actuel des ordinateurs Apple.

Les Unix traditionnels sont en général considérés comme des systèmes monolithiques, même si l'existence actuelle de *modules*, qui permet de rajouter des fonctionnalités en cours d'exécution et celle de *processus noyaux* rendent ceci assez faux.

Le noyau du système est le plus souvent stocké dans le fichier nommé `/boot/vmlinuz`. Ce fichier a une taille de l'ordre du Mo dans les installations standards.

Une fois le fichier image du noyau chargé en mémoire par le boot loader, il est exécuté ; pour hâter sa lecture (et lui permettre de tenir sur une disquette dont la capacité normale est de 1,4 Mo) c'est en fait une version compressée du programme dans le noyau qui se trouve dans le fichier, avec le peu de code nécessaire pour le décompresser ; quand le code s'exécute, il décompresse le reste du fichier et lance son exécution.

Le noyau n'a pas les contraintes de taille du BIOS ou d'un boot loader. Il va tester de façon beaucoup plus exhaustive les périphériques présents (à l'aide de code spécialisé pour chaque type de périphérique, qu'on appelle un *pilote* en français, ou *driver* en anglais). Dans le système Linux, la plupart des drivers peuvent se trouver dans des modules que le noyau charge depuis le disque seulement en cas de besoin.

Une fois les initialisations terminées, le noyau lance le premier processus, qui sous Unix s'appelle `init`. Ce processus est l'ancêtre de tous les autres processus ordinaires qui vont tourner sur le système. Les processus n'exécutent que des instructions ordinaires du microprocesseur : le noyau contrôle leur exécution et limite les interactions entre processus.

Toutes les tâches qui ne sont pas des calculs ordinaires, (l'accès aux périphériques pour les entrées-sorties notamment et la communication avec d'autres processus), seul le noyau peut les exécuter ; les processus vont lui demander de les effectuer, à l'aide d'un dispositif nommé *appel système*. Linux compte quelques centaines d'appels système différents ; ils correspondent à toutes les manipulations que le système peut effectuer à la demande d'un processus.

Une vision de l'exécution d'un programme dans ce contexte, c'est que les instructions ordinaires du microprocesseur plus les appels système forment le jeu d'instruction d'une machine virtuelle sur laquelle s'exécutent les processus.

1.3 Exercices

Les exercices lors du boot sont périlleux ; une fausse manoeuvre peut endommager le système jusqu'au point où il sera nécessaire de le réinstaller. Il convient donc de procéder avec prudence, en notant toutes les modifications effectuées dans le BIOS pour pouvoir remettre les anciennes valeurs en cas d'erreur.

Exercice 1.2 — Interrompre le BIOS et examiner les entrées qu'on peut modifier. Le BIOS de votre machine vous permet-il de choisir de booter sur le réseau ?

Exercice 1.3 — Booter sur un DVD pour vérifier qu'il est bootable, puis modifier l'ordre d'examen des périphériques de boot dans le BIOS pour booter sur le disque dur malgré la présence du DVD bootable dans le lecteur. (Vous pouvez conserver cette configuration, qui vous fera économiser à chaque boot les quelques secondes nécessaires pour tester la présence d'un disque bootable dans le lecteur, à condition de vous souvenir de changer l'ordre des périphériques de boot dans le BIOS le jour où vous voudrez booter sur un DVD.)

Exercice 1.4 — Lors du boot, entrer en interaction avec Grub en tapant un caractère pendant les quelques secondes (deux par défaut sous Mandriva) pendant lesquelles il attend de voir s'il peut booter seul. Vérifier que cela fait

apparaître un menu qui permet de choisir plusieurs versions du système Linux et peut-être une autre système d'exploitation. L'entrée `linux-nonfb` indique que le noyau ne doit pas utiliser le *frame buffer*, c'est à dire qu'il ne peut pas afficher de graphiques. L'entrée *failsafe* boote un noyau de sauvegarde. Booter sur l'entrée linux-nonfb et décrire.

Exercice 1.5 — Regarder le fichier `/boot/grub/menu.lst` ; comparer son contenu avec le menu que vous avez vu lors de l'exercice précédent. Vous pouvez modifier ce fichier comme n'importe quel autre, avec `emacs` ou `vi` et la plus grande prudence. La ligne `timeout` indique combien de temps Grub va attendre qu'une touche soit pressée avant de booter automatiquement ; vous pouvez sans doute la modifier sans risque.
On trouve dans la page `http ://www.troubleshooters.com/linux/grub/grub.htm` une description de quelques commandes de Grub. Vous pouvez vous risquer à les mettre en oeuvre.

Exercice 1.6 — Après le boot du système, vous pouvez consulter la liste détaillée des messages des drivers qui testent les périphériques avec la commande `dmesg`. Examinez la ligne qui décrit la mémoire trouvée par le noyau avec `dmesg | grep Memory :`. Celles qui décrivent ce que le système a découvert sur votre premier disque s'obtient sans doute avec `dmesg | grep hda :` sous Mandriva et `dmesg | grep sda :` dans d'autres distributions Linux. Recopiez ces lignes.

1.4 Supplément

La difficulté qu'il y a à définir exactement ce qu'est un système d'exploitation apparaît à plusieurs reprises dans l'informatique. Je raconte brièvement ici l'histoire des polémiques autour du browser web des systèmes d'exploitation Microsoft et celle du débat atour du nom du système d'exploitation Linux ou Gnu/Linux.

1.4.1 Le browser Web de Windows

Lors de la sortie d'une nouvelle version de Windows, l'accès à la plupart des ressources se faisait à travers son browser web Internet Explorer ; une firme qui avait conçu un browser web populaire, nommé Mozilla, intenta un procès à Microsoft, l'accusant d'obliger les utilisateurs de Windows à se servir d'Internet Explorer pour l'évincer du marché.

La question qui a été débattue pendant des années devant les tribunaux américains peut se résumer à : Internet Explorer fait-il partie du système d'exploitation Windows, comme le prétendait Microsoft, ou bien s'agissait-il d'une vente groupée du système Windows et du programme Internet Explorer, comme le prétendait les propriétaires de Mozilla.

1.4.2 Linux ou GNU/Linux ?

Les débuts officiel du mouvement du logiciel libre résident dans une initiative d'un programmeur célèbre, Richard M. Stallman, qui a entrepris de réaliser une copie libre du système d'exploitation Unix, qu'il a appelé GNU (comme *Gnu, Not Unix*). Pour des raisons techniques, le noyau du système d'exploitation du

projet GNU a pris du retard, et il est peu utilisé à l'heure actuelle ; c'est le noyau Linux, qui a été développé indépendamment qui est devenu populaire.

Cependant la plupart des commandes de base qu'on trouve dans une distribution Linux proviennent du projet GNU. Pour cette raison, Richard M. Stallman demande qu'on ne parle pas de Linux (qui ne désigne que le noyau du système) mais de GNU/Linux (qui décrit le noyau et les commandes de bases).

Encore une fois, la question en jeu est celle de savoir ce qu'est un système d'exploitation : les commandes de base en font-elle partie, ou bien sont-elles des logiciels ajoutés par dessus.

Chapitre 2

Un peu de programmation en shell

Nous montrons dans ce chapitre comment utiliser l'interprète de commande, couramment appelé le *shell*, pour lancer des commandes et écrire des programmes simples.

Le but principal est d'acquérir une certaine familiarité avec l'environnement dans lequel se déroulent de nombreux exemples et exercices de la suite du cours.

J'ai placé les choses qui me semblaient intéressantes ou importantes mais qui ne concernent pas directement le cours dans des phrases entre parenthèses.

Une fois le chapitre assimilé, vous devriez être en mesure d'utiliser une partie non négligeable des fonctionnalités du shell `sh`.

2.1 Le shell, l'interface en ligne de commandes

Nous utiliserons dans ce cours de nombreux exemples dans lesquels nous entrerons des commandes dans des fenêtres de texte, pour examiner ou modifier le comportement du système.

Il y a plusieurs bonnes raisons pour utiliser la ligne de commande plutôt que les interfaces graphiques ; nous en exposons quelques unes ici.

2.1.1 Historique

Le système Linux est basé sur le système Unix qui a été développé dans le courant des années 1970. L'interface naturelle pour un système interactif était alors la *télétype*, une sorte de machine à écrire qui communiquait avec l'ordinateur, et qui imprimait sur un rouleau de papier. On pouvait afficher, mais il n'était pas possible d'effacer et l'usage normal consistait à passer à la ligne suivante pour démarrer avec une ligne vide. Dans ce contexte, pour lancer une commande on tapait une ligne, puis on appuyait sur une touche d'envoi pour la faire exécuter.

Ce modèle a été remplacé dans les années 1980 par des *terminaux*, dont le fonctionnement était presque identique, mais dont les caractères au lieu d'être imprimés sur du papier, étaient affichés sur des écrans. En général, les écrans

étaient capables d'afficher 24 lignes de 80 caractères. Comme il était alors possible d'effacer le contenu de l'écran, on a vu apparaître des éditeurs où on voyait plus d'une ligne de texte (comme `vi` ou `emacs`), mais l'interface de lancement des commandes est restée fondée sur le principe : une ligne par commande.

Les interfaces graphiques de lancement des commandes se sont généralisées dans l'univers Unix dans les années 1990 (d'abord principalement avec des interfaces spécifiques des constructeurs, puis avec une interface commune nommée X ou X11 (les béotiens disent « X Window »). Il était maintenant aussi possible de lancer des commandes avec la souris.

Par dessus cette interface graphique se sont développées plusieurs écoles sur la manière d'interagir avec la souris. Les deux plus répandues sont KDE et Gnome. Elles sont incompatibles entre elles.

Le modèle en *ligne de commande* est encore présent dans de nombreux aspects du système, et c'est une raison pour l'utiliser principalement dans le cadre de ce cours. La plupart des interfaces graphiques de lancement de commandes et d'administration du système restent compatibles avec la ligne de commande des origines.

2.1.2 Programmation

Enchaîner des commandes, pour les exécuter les unes après les autres, c'est facile quand on utilise une interface en ligne de commande. Il suffit de mettre les lignes de commandes dans un fichier et de demander au shell d'exécuter le contenu du fichier. De même, pour lancer une commande quand certaines conditions sont réunies (à une heure précise, par exemple), il suffit de demander l'interprétation d'une ligne de commande quand les conditions sont réunies.

Lancer une commande avec un menu ou un clic sur une icône est sans doute plus facile pour le débutant, mais il n'y a pas de modèle simple qui permette de décrire des enchaînements de clics, ou qui prenne en compte un clic à un moment précis.

Il est donc important de maîtriser la ligne de commande pour pouvoir écrire des groupes de commandes.

2.1.3 Universalité

Les interfaces utilisateurs, notamment KDE et Gnome, sont incompatibles entre elles. En revanche, la ligne de commande reste sous-jacente dans les deux systèmes et fonctionne indifféremment sous les deux environnements. Savoir s'en servir permet donc de fonctionner quel que soit l'environnement.

2.1.4 Facilité de documentation

Les commandes entrées dans une ligne sont plus faciles à communiquer que les interfaces graphiques. Dans ce support de cours, cela permet de les montrer en quelques lignes que vous pouvez recopier facilement, au lieu de multiplier les copies d'écran. De même, on peut discuter oralement de commandes avec un collègue, sans avoir besoin d'images qui sont plus difficiles à décrire avec des mots.

2.1.5 Exercices

Exercice 2.1 — Ouvrir une fenêtre d'interaction en ligne de commande (un *terminal* dans la terminologie X11), entrer quelques commandes simples, compter le nombre de lignes et de colonnes affichées par défaut (et non pas pas quand la fenêtre recouvre tout l'écran).

Exercice 2.2 — Quitter l'interface graphique pour une interface *terminal* qui est présente sur votre ordinateur en tapant simultanément sur les touches `Contrôle`, `Alt` et `F1`. Entrez vos nom et mot de passe usuels. Comptez là aussi le nombre de lignes et de colonnes affichées. Vous pouvez basculer d'une interface à l'autre avec `Alt` `F`*n*. Sous Mandriva, l'écran graphique s'obtient avec `Alt` et `F7`.

2.2 Les shells les plus courants

Le *shell* est le nom usuel d'un interprète de commandes sous Unix, parce qu'il s'agit d'une coquille qui entoure et protège l'utilisateur (en anglais, cela signifie la coquille). Il y a plusieurs interprètes de commandes couramment disponibles sous Unix. Nous faisons un rapide tour d'horizon des plus importants.

2.2.1 Le shell Bourne

Le shell d'Unix version 7, distribué en 1976, s'appelle couramment le *Bourne Shell* du nom de son auteur, Steve Bourne. Il était le premier à avoir presque toutes les caractéristiques d'un langage de programmation et c'est l'ancêtre de la plupart des interprètes de commande actuellement utilisés. Ceux qui ont regardé les sources en C du programme en gardent un souvenir impérissable : à grand coup de `#define`, le programme C de l'auteur s'efforçait de ressembler à de l'Algol, un des premiers langages de programmation bien structuré.

La commande portait le nom `sh`, fidèle aux noms brefs utilisés par Unix, et on peut s'attendre à trouver une commande nommée `sh` dans le répertoire `/bin` sur n'importe quel système Unix ou Linux. Dans la distribution Mandriva, `sh` est un autre nom pour le shell `bash` que nous présentons ensuite, mais dans d'autres distributions, il s'agit d'un shell différent (mais compatible).

2.2.2 Le shell Bash

Le projet GNU est à la base de la plupart des commandes du système Linux. Dans le cadre de ce projet, on a réécrit le shell Bourne et on l'a appelé le *Bourne Again Shell*, qui apparaît sous le nom de commande `bash` (regarder le sens de ce mot dans un dictionnaire anglais). Bash à de nombreuses extensions par rapport au shell Bourne, principalement dans son interface utilisateur : il permet de compléter les noms de fichiers (avec la touche de tabulation) et de rechercher, de modifier et de réutiliser les commandes déjà lancées (avec les flèches du clavier). Malgré sa complexité, bash est le standard *de facto* pour les shells à l'heure actuelle sous Linux.

2.2.3 Le shell Csh

Un des centres de développement important du système Unix dans les années 1970 et 1980 se situait à l'université de Berkeley. Dans ce lieu a été développé un shell assez différent de `sh` pour tout ce qui touche aux aspects avancés. Il s'appelle le C shell (nom de commande `csh`). Ce shell a été pendant longtemps un concurrent sérieux du shell Bourne.

2.2.4 Exercices

Exercice 2.3 — Compter le nombre de lignes de documentation dans les pages de manuel de `sh`, `bash`, `csh`. Jeter un oeil sur le contenu de ces pages de manuel. Laquelle est la plus facile à lire pour vous ?

Exercice 2.4 — Installer `csh` s'il n'est pas présent sur le système, le lancer avec la commande `csh`, l'utiliser et trouver une différence de comportement avec `bash`. (voir la suite du cours).

Exercice 2.5 — (trop difficile) Il existe un shell, appelé le *Korn Shell* ou `ksh` qui fut à la mode. Ce shell est-il plus proche de `sh` ou de `csh`.

2.3 Le shell : utilisation

Dans cette section, nous montrons quelques façons d'utiliser le shell. Nous ne cherchons pas à être exhaustifs, mais à montrer ce qui est d'usage courant et ce que nous utiliserons dans la suite du cours. Ce sera aussi l'occasion de revenir sur la programmation en C, que nous utiliserons également dans le cours.

2.3.1 La boucle de commande

En fonctionnement normal, le shell exécute une boucle dans laquelle il affiche un message d'invite (un *prompt*), lit une commande puis l'exécute. Voici un exemple de session, avec les commentaires en italique sur la droite. Le $ en tête de ligne est notre façon de noter le prompt imprimé par le shell. Il ne faut pas le taper au clavier.

```
$ mkdir /tmp/toto    fabrication d'un répertoire
$ cd /tmp/toto       qui devient le répertoire de travail
$ ls                 liste des fichiers du répertoire
$ ls -l              liste détaillée de ces fichiers
total 0              la liste est toujours vide
$ touch foo          fabrication d'un fichier vide
$ ls                 nouvelle liste des fichiers :
foo                  le fichier apparaît
$ sleep 60           attendre 60 secondes
$ echo foo et bar    afficher du texte
foo et bar           le texte s'affiche
$                    nouveau prompt
```

En dehors des quelques commandes de base utilisées ici, qu'il convient de connaître, les points cruciaux sont que le shell attend la fin d'une commande

pour afficher le prompt et lire la commande suivante. Par exemple, après le `sleep 60` il faut attendre 60 secondes avant de voir réapparaître le prompt.

On peut noter aussi que les commandes Unix en général n'affichent rien quand tout s'est bien passé. On s'y habitue vite, et cela permet de mieux remarquer les messages d'erreur quand il y en a.

Exercice 2.6 — Vérifier, à l'aide des commandes `sleep` et `echo`, qu'on peut entrer des commandes alors que le shell n'a pas encore affiché son prompt. Que se passe-t-il quand la première commande est exécutée ? (Décrire la manip et inclure une copie des lignes significatives dans la fenêtre terminal)

On n'est pas obligé de terminer une commande en appuyant sur la touche *Entrée*. On peut aussi la terminer avec le caractère point virgule ';'. Il s'agit simplement d'une facilité d'expression (on dit que c'est du *sucre syntaxique* ou du *glaçage syntaxique* : ça ne change rien au fond, mais ça aide à faire passer).

```
$ echo foo ; echo et ; echo bar   trois commandes :
foo                               affichage de la première
et                                affichage de la deuxième
bar                               affichage de la troisième
```

On peut placer des commentaires au milieu de ses commandes, avec le caractère dièse ('#'). Le shell ne traitera pas ce qui suit dans la ligne à partir de ce caractère,

```
$ echo foo # et bar   commande et commentaire
foo                   tout jusqu'au commentaire
```

Exercice 2.7 — Examiner si le ';' se comporte comme une fin de ligne du point de vue des commentaires. Donner une commande qui permette de confirmer votre conclusion.

2.3.2 Les processus en arrière-plan, les jobs

On peut indiquer au shell qu'il n'est pas nécessaire d'attendre la fin de la commande lancée pour afficher le prompt, en terminant la commande avec une esperluette (le caractère '&'). Le shell va alors lancer la commande normalement, mais il va afficher le prompt immédiatement sans attendre la fin de la commande.

On appelle les commandes lancées ainsi des commandes en *arrière-plan* (en *background* en anglais), alors que les commandes entrées normalement sont des commandes de *premier plan* (en *foreground*).

Quand on lance une commande en arrière-plan, le shell imprime le numéro de la commande pour le shell (son *numéro de job*) entre crochets et son numéro d'identification pour le système, comme dans :

```
$ sleep 3600 &           une commande en arrière-plan
[1] 4990                 job 1, commande numéro 4990
$ sleep 10 & sleep 5 &   deux commandes en arrière-plan
[2] 4991                 job 2, commande 4991
[3] 4992                 job 3, commande 4992
$                        et nouveau prompt
```

Les jobs sont numérotés à partir de 1 dans le shell ; leurs numéros sont donc en général petits ; les commandes elles sont numérotées d'une façon unique sur le système : les numéros sont en général plus gros.

Quand une commande en arrière-plan se termine, le shell affiche un compte rendu quand il pense que ça ne dérangera pas (en général, juste avant d'afficher un prompt). Par exemple :

```
$ sleep 10 & sleep 5 &          deux commandes en arrière-plan
[2] 4991                        job 2, commande 4991
[3] 4992                        job 3, commande 4992
$ sleep 11                      attendre 11 secondes
[2]- Done         sleep 10      compte rendu de la première
[3]+ Done         sleep 5       compte rendu de la seconde
```

Avec `bash`, on peut faire revenir le job au premier plan (en *foreground*) avec la commande interne au shell `fg %`n ou n est le numéro du job. Le shell va alors attendre la fin de ce job comme si on l'avait lancée normalement.

(Il est possible de faire passer un job du premier plan à l'arrière plan en le stoppant avec la touche CTRL Z : le shell affiche alors son numéro de job ; puis en entrant `bg %`n. C'est fort utile quand on oublié de placer l'esperluette à la fin d'une commande : on la stoppe avec CTRL Z et on la relance en background.)

(On peut consulter la liste des commandes lancées en arrière-plan ou stoppées avec la commande `jobs`.)

2.3.3 L'expansion des noms de fichiers, les quotes

Dans une ligne de commande, le shell va supposer que certains mots sont des *modèles* qui désignent une liste de noms de fichiers qui existent déjà. Il va remplacer le modèle par la liste des noms de fichiers avant de lancer la commande.

Pour décrire des modèles, on peut employer l'étoile (qui peut-être remplacée par n'importe quelle suite de caractère), les crochets (entre lesquels on peut placer une liste de caractères possibles) et le point d'interrogation (qui peut remplacer n'importe quel caractère).

Le shell va prendre ce modèle, vérifier si des noms de fichiers y correspondent et si c'est le cas, remplacer le modèle par la liste des noms fichiers. Voici des exemples d'utilisation des modèles.

`*`	*tous les fichiers du répertoire courant*
`abc*`	*tous ceux dont le nom commence par abc*
`a*z`	*tous ceux dont le nom commence par a et finit par z*
`*abc*`	*tous ceux dont le nom contient abc*
`*[abc]*`	*tous ceux dont le nom contient a, b ou c*
`[0-9]*`	*tous ceux dont le nom commence par un chiffre*
`*/foobar`	*tous les fichiers foobar dans un sous-répertoire*

Je souligne que ce remplacement d'un modèle par une liste de noms de fichiers est effectué *par le shell*, avant le lancement de la commande ; celle-ci n'a donc pas besoin de faire ce travail. Par exemple, écrivons un programme élémentaire qui affiche les arguments avec lesquels il a été lancé, un peu à la façon de la commande echo. Une façon de faire est le programme C suivant.

```
/* ba-echo.c
 Pour afficher le contenu de la ligne de commande
```

```
 jm@univ-paris8.fr, fevrier 2008
*/
# include <stdio.h>

int
main(int ac, char * av[]){
  int i;

  printf("Programme lance sous le nom %s avec %d arguments\n",
         av[0], ac - 1);
  for(i = 1; i < ac; i++)
    printf("argument %d : %s\n", i, av[i]);
  return 0;
}
```

On peut le compiler avec

```
$ gcc -g -Wall ba-echo.c -o mon-echo
```

et le résultat sera un fichier exécutable nommé `mon-echo`. Si on lance le programme avec `mon-echo *`, on voit bien qu'il est appelé avec les noms de fichiers du répertoire courant dans la liste des arguments.

Exercice 2.8 — Que fait le shell quand aucun nom de fichier ne correspond pas au modèle ? Que fait csh ? Ces réponses peuvent aussi servir pour un exercice précédent.

(En cas d'urgence, `echo *` est une façon bon marché de connaître les noms des fichiers du répertoire courant.)

Pour empêcher l'interprétation de ces caractères spéciaux, on peut utiliser le simple quote ou le double quote pour délimiter des zones qui n'ont pas à être traduites. Le caractère contre-oblique (ou *antislash*) sert lui à empêcher la traduction du prochain caractère. Par exemple :

```
echo \* ; echo '*' ; echo "*"
```

va afficher trois lignes avec chacune une étoile. Pour afficher les noms des fichiers qui contiennent une étoile, on peut faire

```
echo *'*'* ; echo *'*'* ; echo *\**
```

Les quotes simples et doubles et la contre-oblique, permettent aussi d'éviter que les espaces soient interprétés comme des séparateurs. Comparer ce qu'imprime

```
$ mon-echo ceci et cela
```

avec la sortie de

```
$ mon-echo 'ceci et cela'
```

Les quotes simples et doubles se comportent différemment vis à vis des variables, que nous verrons plus loin.

2.3.4 La redirection d'entrée-sortie vers des fichiers

Sur la ligne de commande, on peut indiquer que les programmes doivent lire et écrire dans un fichier, au lieu du clavier et du terminal, avec les chevrons ouvrant et fermant '>' et '<' Par exemple, pour faire un programme C tout simple, on peut taper simplement

```
$ echo 'main(){}' > rien.c
```

L'argument de la commande `echo` est le texte d'un programme C qui ne fait rien ; il sera écrit dans le fichier `rien.c`.

Pour ajouter à un fichier au lieu de remplacer son contenu, on peut utiliser deux chevrons fermant '>>'.

(Supplément : trouver à quoi servent les doubles chevrons ouvrants.)

Une manière compliquée de recopier un fichier dans un autre est d'utiliser la commande `cat` en lui faisant lire le fichier à recopier et écrire dans la copie comme dans :

```
$ cat < ancien > nouveau
```

Pour ne voir que les messages d'erreurs d'une commande, `make` par exemple, il est courant de rediriger la sortie pour que seuls les messages d'erreur apparaissent à l'écran, comme dans

```
$ make > /dev/null
```

Pour ne conserver que les messages d'erreur, et non les impressions normales, on utilise la formule magique '2>', comme dans

```
$ make > normal 2> erreur
```

Les messages ordinaires seront imprimés dans le fichier `normal` et les messages d'erreur dans le fichier `erreur`.

On insiste ici aussi sur le fait que ce traitement des redirections d'entrée-sortie est effectuée par le shell, sans que la commande puisse y faire quoi que ce soit. Le programme C que nous avons vu plus haut ne contient rien pour traiter la redirection de la sortie.

```
$ mon-echo foo > toto et bar   lancement de la commande
                               on ne voit rien sur le terminal
$ cat toto                     on regarde le fichier toto qui contient
                               la sortie normale de notre programme
Programme lancé sous le ...
argument 1 : foo
argument 2 : et
argument 3 : bar
$
```

Comme on le voit, le programme n'a pas été informé de la redirection ; c'est le shell qui l'a mise en place avant que le programme ne commence et il ne lui passe pas comme argument les caractères `> toto` qui ont servi à l'indiquer sur la ligne de commande.

2.3.5 Les pipes

Il y a moyen de rediriger la sortie d'un programme non vers un fichier mais vers l'entrée d'un autre programme, avec le caractère *barre verticale*. On appelle cette connexion entre les deux programmes un *pipe*.

Un exemple simple :

```
$ mon-echo foo et bar | wc
4 21 102
```

Nous utilisons la commande `wc` (qui compte les lignes, les mots et les caractères sur son entrée et affiche le résultat) pour analyser la sortie de notre commande `mon-echo`. Pour un exemple plus utile :

```
$ grep super < rapport | wc
```

compte le nombre de lignes du fichier `rapport` où apparaît le mot *super*.

Pour un exemple plus compliqué :

```
$ ls *.bak | sed 's/\(.*\).bak$/mv & \1/' | sh
```

permet de retirer le `.bak` du nom des fichiers du répertoire courant, en enchaînant trois commandes : `ls` liste les fichiers à manipuler, `sed` fabrique les commandes et `sh` les exécute.

2.3.6 Regrouper des commandes

Il est possible de regrouper des commandes en utilisant des parenthèses. Les commandes entre les parenthèses seront exécutées comme d'ordinaire par le shell, mais on peut modifier leur comportement en bloc, en les lançant en arrière-plan ou en redirigeant leurs entrées et sorties.

Ainsi :

```
$ ( sleep 3600 ; echo une heure est passée ) &
```

lance en arrière-plan la suite des commandes qui attend une heure puis affiche un message : les commandes entre parenthèses sont en arrière-plan et on pourra donc continuer à travailler. Comme ces commandes sont séparées par un point-virgule, la seconde (l'affichage du message) ne sera effectuée que quand la première (dormir pendant une heure) sera terminée.

De même :

```
$ ( commande1 ; commande2 ) > journal
```

lance successivement la commande 1 et la commande 2, en redirigeant leurs sorties à toutes les deux vers le fichier journal.

Encore :

```
$ mkdir /tmp/x /tmp/x/a /tmp/x/b   fabriquer trois répertoires
$ cd /tmp/x/a                      changer le répertoire courant
$ pwd                              imprimer le répertoire courant
/tmp/x/a
$ (cd /tmp/x/b ; pwd)              changer de répertoire et ré-afficher
/tmp/x/b
$ pwd                              le répertoire courant ?
/tmp/x/a                           on est sorti des parenthèses, donc
                                   de retour au point de départ
```

Le changement de répertoire courant, obtenu avec la commande `cd`, n'affecte que les commandes qui sont entre parenthèses. De ce fait, la meilleure manière de recopier un répertoire (ici `from`) dans un autre (ici `to`) a longtemps été avec :

```
$ (cd from ; tar cf - ) | (cd to ; tar xf -)
```

On peut aussi utiliser la commande `cp` (*copie*) avec l'option `-R` (*récursif*).

2.3.7 Variables et environnement

Le shell permet de définir et d'utiliser des variables qui lui sont propres. Il permet également de manipuler des variables qui seront connues des programmes qu'il lance, dans ce qu'on appelle l'*environnement*.

Les variables

Pour définir une variable, on écrit simplement son nom suivi du signe '=' et de la valeur de la variable. Pour remplacer une variable par sa valeur, on place le caractère dollar ('$') devant son nom.

Pour travailler entre deux répertoires différents il peut être utile d'utiliser deux variables comme dans

```
$ x=/usr/local/tmp/x          x = premier répertoire
$ y=/tmp/toto                 y = second répertoire
$ cd $x                       va dans le premier
$ diff yoyo.txt $y/toto.txt   compare deux fichiers
$ cp yoyo.txt $y              copie le fichier dans le second
$ cd $y                       va dans le second
$ mv yoyo.txt toto.txt        renommes-y le nouveau fichier
```

Certaines variables ont un rôle particulier. Ainsi la variable `PS1` contient ce qui sert de prompt. On peut la modifier comme on le souhaite :

```
$ PS1='toto '     modifier le prompt
toto cd /tmp      le prompt est modifié
toto pwd             mais le shell fonctionne
/tmp                 normalement
toto PS1='$ '     revenons à un prompt plus ordinaire
$
```

Les variables sont spécifiques de chaque invocation du shell. Quand on lance une commande, la variable a perdu sa valeur. Ceci apparaît nettement quand on lance un shell comme une commande :

```
$ x=foo          la variable x de ce shell vaut foo
$ echo $x        vérification
foo              vérification réussie
$ sh             on lance un autre shell
$ PS1='2$ '      on modifie son prompt
2$ echo $x       valeur de x dans le second shell ?
                 x n'a pas de valeur dans ce shell
2$ x=bar         on y met une valeur
2$ echo $x       vérification
bar              vérification réussie
2$ exit          on termine le second shell
$                retour dans le premier shell
$ echo $x        la valeur de x a-t-elle changée ici ?
foo              non
```

La variable x peut avoir une valeur différente dans chaque exécution du shell. La modification de la valeur d'une variable dans une exécution ne modifie pas sa valeur dans les autres.

Certaines variables ont un rôle prédéfini ; dans la suite du cours, nous utiliserons `$$` qui contient le numéro du shell, la variable `$!` qui contient le numéro du dernier processus lancé en arrière-plan et `$?` où se trouve le compte-rendu de la dernière commande lancée en premier plan.

L'environnement

Il existe dans le système Unix quelque chose qui ressemble à des variables, mais qui est partagé entre le shell et les commandes qu'il lance. On appelle ceci l'*environnement*.

On peut regarder les variables présentes dans l'environnement avec la commande `printenv`. (Puisque l'environnement est connu des commandes lancées par le shell, il peut exister une commande pour l'examiner.)

L'environnement est une manière ordinaire de paramétrer de façon semi-permanente le comportement de commandes ; par exemple c'est la variable `LS_COLORS` qui indique à `ls` comment il doit colorier les noms de fichier pour les rendre illisibles. (Si comme moi, vous préférez avoir tous les noms de fichiers de la même couleur, vous pouvez désactiver les couleurs avec `unalias ls`.)

Une variable d'environnement intéressante est la variable `$HOME` qui contient le nom du répertoire de login. Pour la voir, on peut utiliser

```
$ printenv | grep HOME=
```

pour ne garder de la sortie de `printenv` que la ligne qui contient HOME. Puisque les variables de l'environnement apparaissent comme des variables du shell, on peut aussi regarder sa valeur plus simplement avec :

```
$ echo $HOME
```

Une variable d'environnement très importante est la variable `PATH`. Elle contient la liste de tous les répertoires dans lequel le shell va chercher les fichiers qui correspondent aux commandes tapées. Celle liste est stockée sous la forme d'une chaîne de caractère, dans laquelle le caractère *deux points* (' :') sert à séparer les éléments :

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin
```

Couramment, le `PATH` ne contient pas le répertoire courant et pour faire exécuter le résultat d'une compilation, on est obligé de taper `./mon-echo`. Pour pouvoir appeler la commande simplement avec `mon-echo`, il suffit d'ajouter point (`'.'`) au PATH, par exemple avec

```
$ PATH=.:\$PATH
```

Exercice 2.9 — (trop difficile) Comment avoir dans PATH un répertoire dont le nom contient ' :' et pouvoir exécuter les fichiers qu'il contient ?

Subtilités entre variables et environnement

On peut placer une variable dans l'environnement de façon définitive avec la commande `export`. Par exemple

```
$ toto=xxx
$ export toto
```

affecte une valeur à la variable `toto` puis place la variable dans l'environnement avec sa valeur. Certains shells (dont bash) permettent d'abréger ceci en une seule commande :

```
$ export toto=xxx
```

Par la suite, les modifications de la valeur de `toto` n'affecteront que la variable locale ou bien aussi la valeur dans l'environnement suivant le shell qu'on utilise.

Exercice 2.10 — sous `bash`, les modifications de variables sont-elles répercutées dans l'environnement ?

Sous `bash`, on peut retirer une variable de l'environnement avec la commande `unset` : la variable shell et la variable d'environnement sont toutes les deux supprimées.

Quand on préfixe une commande par une affectation de variable, ceci n'affecte que l'environnement de la commande, sans modifier la variable du shell. Exemple en utilisant `PS1` :

```
$ echo prompt $PS1      quelle est la valeur du prompt ?
prompt $                juste un dollar
$ PS1='? ' sh           lancer un autre shell avec un autre prompt
? echo foo              le second shell a le nouveau prompt
foo
? echo prompt $PS1      vérification
prompt ?                réussie
? exit                  sortie du second shell
$                       dans le premier, PS1 n'a pas bougé
```

2.3.8 Fabrication d'un programme shell

On peut placer des commandes dans un fichier et les faire exécuter en redirigeant l'entrée du shell vers ce fichier. Il lira alors les commandes depuis le fichier au lieu de les lire depuis le clavier. Par exemple, si le fichier toto contient

```
echo Début du programme
x=foo
echo La variable x contient $x
echo Fin du programme
```

on peut le faire exécuter avec `sh < toto`.

Une autre façon plus simple consiste à modifier les attributs du fichier pour indiquer au système qu'il est exécutable, avec la commande

```
$ chmod a+x toto
```

Il est maintenant possible d'exécuter le fichier comme n'importe quelle autre commande avec

```
$ toto
```

parce que le système, quand il rencontre un fichier marqué comme exécutable mais dont il ne reconnaît pas le type suppose que ce fichier contient des commandes et le fait interpréter par le shell.

Une meilleure manière consiste à placer sur la première ligne du fichier toto un commentaire qui contient

```
#! /bin/sh
```

qui indique au système que le contenu du fichier est à faire interpréter par le shell.

(On peut placer n'importe quelle commande à la place du shell. Par exemple, on peut parfaitement utiliser le programme C que nous avons écrit plus haut en indiquant sur la première ligne

```
#! mon-echo
```

On verra alors que le programme indiqué sur la première ligne est lancé avec le nom du fichier comme premier argument)

On appelle ces petits programmes sous forme texte des *scripts*. Quand ils contiennent des commandes pour le shell, on les appelle des *scripts shell*.

2.3.9 Les structures de contrôle

Le shell est en réalité un langage de programmation complet, avec des structures de contrôle ordinaires (et d'autres moins ordinaires).

Quand les programmes se terminent, ils rendent un compte rendu qui permet de savoir s'ils ont détecté une erreur ou pas. On peut utiliser ce compte rendu dans des tests. Un exemple simple de test :

```
if gcc fichier.c ; then echo 'compil ok' ; else echo erreur ; fi
```

Une boucle qui attend l'apparition d'un fichier nommé joe :

```
while ! test -f joe ; do echo pas de fichier joe ; sleep 10 ; done
```

Le point d'exclamation sert à renverser le sens du test ; le corps est imbriqué entre les mots clefs `do` et `done` ; la commande `test`, avec l'option `-f` vérifie qu'il existe bien un fichier `joe` et renvoie un compte-rendu d'erreur si non.

Une autre forme de boucle permet de donner successivement à une variable chaque valeur prise dans une liste. Un exemple utile qui fait une version de sauvegarde de tous les fichiers du répertoire courant :

```
for i in * ; do cp "$i" "$i.bak" ; done
```

L'étoile est remplacée par la liste de tous les fichiers du répertoire courant ; la variable i prend successivement la valeur de chacun de ces noms de fichiers et on l'utilise dans le corps de la boucle pour recopier le fichier. Les doubles quotes permettent de traiter correctement les fichiers dont le nom contient des espaces (mais la copie échouera pour les répertoires) ; dans les doubles quotes, les variables sont remplacées par leurs valeurs (pas dans les simples quotes).

On peut imbriquer les boucles. Ainsi une manière de produire tous les entiers entre 0 et 99 dans un fichier ;

```
chiffres='0 1 2 3 4 5 6 7 8 9'
for i in $chiffres ; do
   for j in $chiffres ; do
     echo $i$j
   done
done > liste00-99
```

Notez la façon dont la sortie de la boucle et des commandes qui y sont lancées est redirigée en une seule fois vers un fichier. Une manière de les relire les nombres :

```
while read x ; do echo $x ; done < liste00-99
```

On aurait pu lier les deux boucles avec des pipes.

Exercice 2.11 — (difficile) Faire un script shell qui permette d'imprimer comme dans la boucle précédente tous les nombres de n chiffres où n est passé comme unique argument

2.3.10 Le reste

Nous n'avons fait ici qu'effleurer la programmation en shell. En pratique il est intéressant, pour devenir un utilisateur averti, de survoler la documentation d'un shell (probablement de `bash`), sans entrer dans les détails, afin de savoir tout ce qu'on peut faire avec. Quand on affrontera un problème que le shell permet de résoudre facilement, on le saura grâce au survol rapide et on pourra alors lire avec soin la documentation pour trouver comment procéder.

Une alternative consiste à maîtriser complètement la programmation dans le langage `perl` qui combine les propriétés du shell, celles de quelques commandes couramment utilisées dans la programmation shell comme `test`, `grep` et `sed` et bien d'autres choses encore.

2.4 Supplément : `sh` ou `bash` ?

Le shell `bash` a des extensions par rapport à `sh`, le shell usuel d'unix. Par exemple avec `bash` on peut faire de l'arithmétique directement en entourant une expression avec des parenthèses et un dollar, comme dans :

```
$ x=$((1+1))
$ echo $x
2
```

alors qu'avec `sh` il faut passer par une commande extérieure comme `expr` ou `bc`.

Si on emploie des extensions de `bash`, il faut le noter car une tentative d'exécution des commandes par `sh` produira au mieux une erreur et plus probablement des catastrophes. C'est particulièrement important dans les scripts, où une première ligne `#! /bin/sh` n'aura pas le même effet que `#! /bin/bash`.

La situation est encore compliquée : certaines distributions de Linux ont deux commandes différentes pour `sh` (sans extension) et `bash` (avec extensions), alors que d'autres ont remplacé `sh` par `bash`, si bien que les extensions `bash` fonctionnent toujours. On rencontre donc des scripts shell (qui utilisent les extensions bash) qui fonctionnent sur certaines distributions de Linux et pas sur d'autres.

Personnellement, j'évite d'utiliser les extensions spécifiques de `bash`. Je vous conseille de faire de même.

Chapitre 3

Les processus : création et destruction

Dans le premier chapitre, nous avons vu ce qui se passe au démarrage quand s'installe sur l'ordinateur un super-programme : le *noyau* (*kernel*) du système d'exploitation. Un des rôles du kernel est de coordonner l'action des autres programmes qui tournent sur l'ordinateur, en les isolant les uns des autres tout en leur permettant de communiquer entre eux d'une façon contrôlée.

Une fois le chapitre assimilé, vous aurez une idée plus précise de ce que sont un thread et un processus, et vous serez en mesure d'écrire des programmes C qui en fabriquent autant que de besoin.

3.1 Le processus : définitions

Il y a plusieurs façons de définir ce qu'est un processus dans le contexte d'un système d'exploitation. Les deux définitions les plus importantes sont du point de vue de l'utilisateur et du point du vue du système d'exploitation.

3.1.1 Processus = un programme qui tourne

Du point de vue de l'utilisateur, un processus est un programme qui tourne.

Un programme, c'est un ensemble d'instructions et de données pour effectuer une tâche ; la tâche peut être simple, comme dans les commandes `sleep` ou `echo`, ou plus complexe comme dans les commandes `emacs` ou `gcc`.

Si on prend un programme simple, comme `sleep`, il est évident qu'on peut lancer ce programme plusieurs fois. On peut même le lancer plusieurs fois en même temps comme dans

```
$ sleep 10 & sleep 15 & sleep 20 &
```

Ici le programme `sleep` a été lancé trois fois, avec trois arguments différents : il est donc en train de s'exécuter trois fois dans le système. Ce sont ces exécutions qu'on appelle des processus.

La définition n'est pas complètement satisfaisante parce qu'un programme complexe, comme `gcc` va en fait lancer l'exécution de plusieurs processus qui

sont chargés chacun d'une partie du travail. C'est néanmoins souvent une bonne façon de voir un processus.

3.1.2 Processus = entité à laquelle le système attribue les ressources

Le système d'exploitation est chargé, entre autres, d'arbitrer entre les différents programmes qui tournent sur la machine pour leur attribuer les ressources disponibles. Les ressources peuvent être de la mémoire centrale, de l'espace disque, du temps d'exécution sur le processeur, etc.

Du point de vue du système, l'unité qui utilise des ressources est le *processus*. Le système gardera en mémoire les ressources utilisées par chaque processus, et utilisera cette information pour décider de l'attribution des ressources futures.

Cette définition n'est pas complètement satisfaisante parce qu'il existe une entité plus petite que le processus, qu'on appelle un *thread* ou *processus léger* (*lightweight process*) à laquelle le système attribue également des ressources.

3.1.3 Processus, tâche ou job

Le processus est un concept qu'on rencontre dans (presque) tous les systèmes d'exploitation, mais il porte des noms variés. Dans certains systèmes, ce qu'on appelle un processus sous Unix s'appelle un *job* (attention, sous Unix les jobs sont autre chose, comme on l'a vu dans le chapitre sur le shell) ; dans d'autres cela s'appelle une *tâche* (*task* en anglais). Dans les sources du noyau Linux, la structure principale utilisée pour représenter un processus porte le nom `task_struct`.

3.1.4 Qu'est-ce qui caractérise un processus ?

Cherchons à énumérer ce que nous pouvons deviner de ce qui caractérise un processus. Une autre façon de poser cette question est : *étant donné un processus, de quelles informations ai-je besoin à un moment précis pour être en mesure de le recréer à l'identique s'il venait à disparaître ?*

Parce qu'un processus est un programme en cours d'exécution, il faut bien sur savoir de quel programme il s'agit ; il faut donc connaître les instructions qui le composent ; soulignons que les instructions sont fixées une fois pour toute (lors de la compilation) et qu'elles ne seront jamais modifiées : plusieurs processus qui exécutent le même programme ont les mêmes instructions.

Il faut aussi connaître ses données, qui vont probablement changer de valeurs au cours de l'exécution. A mesure que le programme s'exécute, de nouvelles données peuvent même apparaître, parce qu'on a utilisé l'opérateur `new` en C++ ou la fonction `malloc` en C, ou parce qu'on a appelé des fonctions qui ont utilisé la pile pour sauver des valeurs. Les données d'un processus lui sont propres, plusieurs processus qui exécutent le même programme ne peuvent pas partager leurs données. (En fait, pour économiser la mémoire, on s'efforce de distinguer les données qui ne seront jamais modifiées pour pouvoir les partager entre processus ; c'est pour cette raison que gcc interdit d'écrire à l'intérieur des chaînes de caractères si on n'a pas compilé avec l'option `-fwritable-string`.)

Parce que le programme est en cours d'exécution, il est aussi nécessaire de connaître l'adresse de l'instruction courante, généralement stockée dans un re-

gistre qu'on appelle `PC` comme *Program Counter* (parfois appelé *pointeur ordinal* en français).

Il y a d'autres choses également, comme le répertoire de travail (le *working directory* affiché par la commande `pwd`). On a vu dans le chapitre sur le shell que chaque processus possède un *environnement*. Il a aussi des fichiers ouverts, au moins pour lire, afficher les messages ordinaires et afficher les erreurs : ces descripteurs de fichiers peuvent varier d'un processus à l'autre, comme on l'a vu avec les redirections d'entrée sortie. En pratique, les connexions réseau que le processus a pu ouvrir se comportent, du point de vue du système, à peu près comme des descripteurs de fichiers.

Il y a beaucoup d'autres caractéristiques qu'il est nécessaire de connaître pour tout savoir sur un processus ; nous en verrons un certain nombre dans la suite du cours ; la chose qu'il est important de retenir pour le moment, c'est qu'un processus est une chose assez complexe.

Le PID

Pour nommer les processus, le système (Unix ou Linux) leur colle un numéro d'identité nommé PID comme *Process ID*. Le premier processus au démarrage du système reçoit le PID numéro 1, le second le PID 2 et ainsi de suite. On verra plus loin dans ce chapitre ce qui se passe quand on a atteint le PID maximum.

3.1.5 Les commandes `ps` et `top`

La commande `ps` (comme *process status*) permet d'examiner certaines caractéristiques des processus. Un exemple de sortie :

```
 PID TTY          TIME CMD
4937 pts/1    00 :00 :00 bash
4976 pts/1    00 :00 :09 emacs
5123 pts/1    00 :00 :00 ps
```

La commande `ps` décrit chaque processus par une ligne. La colonne `PID` contient son numéro, la colonne `TTY` la fenêtre depuis laquelle il a été lancé, la colonne `TIME` le temps pendant lequel il a calculé (en minutes, secondes et centièmes de secondes) et la colonne `CMD` la commande qu'il est train d'exécuter.

Pour avoir des informations plus détaillées, on peut utiliser `ps -l`. Nous découvrirons, au long du cours, le rôle de la plupart des colonnes.

Par défaut, `ps` n'informe que sur les processus lancés depuis la même fenêtre. Pour avoir des informations sur *tous* les processus, il faut dire `ps ax`.

La commande `top` fonctionne presque comme la commande `ps`, mais elle met à jour les informations sur les processus à intervalles réguliers (toutes les trois secondes par défaut), et elle place en tête de liste les processus actifs. Quand un ordinateur est lent, une des première chose à faire est de regarder avec `top` s'il y a des processus qui font de nombreux calculs.

Exercice 3.1 — Compiler le programme suivant, qui calcule beaucoup (des calculs sans intérêt), le lancer et regarder l'état des processus avec `top` pendant qu'il tourne. Le lancer plusieurs fois en même temps. Combien de fois faut il lancer le programme en même temps pour commencer à en ressentir les effets ?

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

```
int
main(){
      int x;

      for(;;)
              x += 1;
      return 0;
}
```

3.2 La mort d'un processus

Nous commençons par examiner la fin des processus, qui est plus facile que la création.

Pour se terminer, un processus demande au noyau du système de le supprimer avec un appel système spécifique, qui s'appelle `exit`. (En réalité, la fonction `exit` fait le ménage avant de demander au noyau de terminer le processus. L'appel système lui-même s'appelle `_exit`.)

Il y a un paramètre à `exit`, qui est le compte-rendu de l'exécution du programme. Par convention, un processus pour qui tout s'est bien passé renvoie la valeur 0 ; n'importe quelle autre valeur signale un problème. C'est facile de s'en souvenir parce que c'est le contraire de ce qui se passe en C.

Un programme C qui ne fait que sortir :

```
int main(){ exit(0) ; }
```

Compilez et testez le programme, constatez qu'il n'imprime pas la valeur de sortie. On peut récupérer cette valeur dans la variable shell `$?`.

Ce programme existe déjà, sous le nom de `true`. On l'utilise pour la programmation shell quand on veut faire une boucle infinie. Il y a un programme symétrique nommé `false`, qui ne fait rien, mais s'arrête avec un compte-rendu différent de 0.

```
$ true
$ echo $?
0
$ false
$ echo $?
1
$ gcc correct.c
$ echo $?
0
$ touch incorrect.c
$ gcc incorrect.c
(.text+0x18) : undefined reference to `main'
collect2 : ld returned 1 exit status
$ echo $?
1
```

Exercice 3.2 — Le programme suivant se contente de sortir avec le compte rendu d'exécution qu'on lui a passé sur la ligne de commande :

```
/* ca-sortir.c
 Un programme qui s'arrete avec la valeur fournie en argument
*/
# include <stdio.h>
# include <stdlib.h>

int
main(int ac, char * av[]){
        if (ac != 2){
                fprintf(stderr, "usage: %s N\n", av[0]);
                return 1;
        }
        return atoi(av[1]);
}
```

Le compiler sous le nom `sortir`. L'appeler avec des valeurs diverses et consulter la valeur de la variable $?. En conclure le type de données dans lequel la valeur de sortie est stockée. Exemples d'appel :

```
$ sortir 0 ; echo $?
0
$ sortir 23 ; echo $?
23
$ sortir 1234 ; echo $?
210
```

Regarder notamment les valeurs quand on l'appelle avec `sortir 1024`.

Exercice 3.3 — (facile) Que vaut la valeur de sortie d'un processus interrompu avec CTRL C (par exemple interrompre la commande `sleep 60` et consulter la valeur de $?). Et si on l'interrompt avec Contrôle-\ ?

Exercice 3.4 — (plus difficile) Que vaut la valeur de sortie d'un processus interrompu parce qu'il tente de lire le contenu de l'adresse 0? Attention, ici il faut que vous fassiez un programme C parce que les programmes ordinaires se gardent bien de lire cette adresse.

Exercice 3.5 — (encore plus difficile) Que vaut la valeur de sortie pour un programme qui essaye d'effectuer une division par 0? Attention, si on fait un programme élémentaire comme `int main(){ 23 / 0 ; }` gcc va détecter la division par 0 (et donner un warning) et la retirer du programme.

3.3 La création d'un nouveau processus 1 : `fork`

On pourrait imaginer d'avoir un appel système qui servirait à créer un processus. Un problème c'est que cet appel système devrait avoir autant d'arguments qu'un processus a de caractéristiques (et elles sont numbreuses) : il faudrait spécifier le programme à lancer, le répertoire de travail, les fichiers à ouvrir, la fenêtre à utiliser etc. Il parait que la fonction de création de processus sous Windows a quinze paramètres : c'est beaucoup trop pour qu'on l'utilise facilement.

Sous Unix, la création de processus se fait en deux temps, avec deux appels système plutôt simples : fork et exec. Nous commençons ici par examiner `fork`.

L'appel système `fork` est tellement simple qu'il en est étrange : quand un processus appelle `fork`, le système en fait une copie intégrale, avec le même code, les mêmes données, le même point d'exécution, les mêmes fichiers ouverts etc. Seules deux choses changent dans la copie : le numéro du processus (le PID) est différent et dans la copie, `fork` renvoie 0 alors que dans l'original il renvoie le PID de la copie.

On parle souvent de processus *parent* pour l'original et de processus *enfant* pour la copie.

L'exemple le plus simple de fork :

```
/* cb-fork-simple.c
 Un fork simple
 */
# include <stdio.h>
# include <unistd.h>

int
main(){
  if (fork())
    printf("Vrai\n");
  else
    printf("Faux\n");
  return 0; /* ou return EXIT_SUCCESS; */
}
```

Quand le processus appelle `fork`, le système le copie dans un processus enfant. Dans le parent, `fork` renvoie le PID du processus enfant et il imprime `Vrai`. Dans l'enfant, `fork` renvoie 0 et il imprime `Faux`.

Dans un vrai programme, il faut prendre en compte le fait que l'appel système `fork` peut échouer. Dans ce cas, il renvoie une valeur inférieure à 0. Pour cette raison, la manière ordinaire d'utiliser `fork` est la suivante :

```
int
main(){
  int t;

  ...
  t = fork();
  if (t == -1){              // erreur
     traitement d'erreur
  } else if (fork() == 0){ // enfant
     traitements spécifiques de l'enfant
  } else {                 // parent
     traitement spécifiques du parent
  }
  return 0;
}
```

Exercice 3.6 — Qu'imprime le programme suivant, et pourquoi ?

```
/* cc-fork-etoile.c
 Fork et printf : les noeuds d'un arbre binaire
```

```
*/
# include <stdio.h>
# include <unistd.h>

int
main(){
  int i;

  setbuf(stdout, NULL);
  for(i = 0; i < 10; i++){
    printf("%d\n", i);
    fork();
  }
  return 0;
}
```

3.3.1 La bombe fork

Le programme suivant est simple, mais dangereux :

```
int
main(){
    for(;;)
        fork();
}
```

Quand on le lance, il crée un processus puis le parent et l'enfant créent au tour de boucle suivant un autre processus ; au tour suivant on a huit processus, puis seize, puis trente deux, etc. Très rapidement, l'ordinateur ne va plus faire autre chose qu'essayer de créer des processus, et toutes les autres opérations seront ralenties de façon significative. On peut essayer de supprimer les processus, mais ils sont créés avec fork plus vite qu'on ne les détruit.

En pratique, la solution la plus aisée consiste à redémarrer l'ordinateur. On appelle ce programme la *bombe fork*.

On peut essayer d'éviter le problème avec l'appel système `ulimit` qui permet de placer des limites aux ressources qu'un utilisateur peut obtenir. On peut notamment limiter le nombre de processus créés. En général, la limite n'est pas définie par défaut.

3.4 Attendre la fin d'un processus enfant : `wait`

Il existe un appel système `wait` qui permet à un processus parent d'attendre la fin d'un processus enfant et récupérer alors la valeur avec laquelle celui-ci s'est terminé (l'argument qu'il a fourni à `exit`).

On passe à `wait` l'adresse d'un entier où le compte-rendu sera placé, et il renvoie le PID du processus qui s'est terminé, ce qui est utile pour les processus qui ont plusieurs processus enfants.

Avec `wait`, on peut écrire un programme vraisemblable, même s'il ne fait rien d'utile, avec la création d'un processus.

```
/* cd-fork-wait.c
 lancer un processus, attendre qu'il se termine
 */
# include <stdio.h>
# include <unistd.h>
# include <stdlib.h>

int
main(){
  int t, pid, etat;

  printf("Je suis le processus de depart\n");
  t = fork();
  if (t == -1){
    perror("fork");
    return 1;
  } else if (t == 0){
    printf("enfant: j'attend 5 secondes\n");
    sleep(5);
    printf("enfant: je termine\n");
    return 0;
  } else {
    for(;;){
      pid = wait(&etat);
      if (pid == t)
        break;
      printf("parent: wait a rendu la valeur %d, j'attend encore\n", pid);
      if (pid == -1)
        perror("wait");
    }
    printf("parent: l'enfant s'est termine avec la valeur %d\n", etat);
    printf("parent: je termine\n");
    return 0;
  }
}
```

Notez le traitement en cas d'erreur : on appelle la fonction `perror` qui imprime le message standard qui détaille la raison pour laquelle `fork` n'a pas fonctionné. Sur un système bien configuré, ce message sera dans la langue choisie lors de l'installation du système.

(Pour faire échouer le fork et voir le message d'erreur, on peut utiliser la commande `ulimit` qui permet de limiter les ressources, notamment le nombre de processus qu'un utilisateur peut lancer. Pour ce faire, il faut commencer par compter le nombre de processus qu'on a déjà lancés, par exemple avec `ps -u nom-de-login | wc`. Si cette commande a compté 19 lignes, c'est qu'on a 16 processus actifs (puisqu'il faut retirer la première ligne de la sortie de `ps` qui nomme les colonnes et les deux processus `ps` et `wc`). On va donc limiter le nombre de processus à 18 avec `ulimit -u 18`. Maintenant, quand le shell lance la commande `a.out`, cela fonctionne, mais quand notre programme tente de créer un nouveau processus, cela échoue puisqu'on a atteint le nombre maximum de

processus.)
Exercice 3.7 — (facile) Que se passe-t-il quand un processus fait un `wait` alors qu'il n'a pas de processus enfant ?

Exercice 3.8 — (moyen) wait attend-il la fin (1) d'un processus enfant direct ou bien (2) d'un processus enfant ou d'un descendant d'un processus enfant ? Pour trouver la réponse, le plus simple est de faire une expérience. Écrire un programme P1 qui crée un processus P2 et attend la fin d'un enfant avec wait. L'enfant P2 crée un petit enfant P3 et attend 20 secondes. Le petit enfant attend 10 secondes. Quand on lancera le programme, s'il se termine au bout de 10 secondes, c'est que wait attend la fin d'un enfant *ou* d'un petit enfant. S'il se termine au bout de 20 secondes, c'est qu'il attend la fin de l'enfant. On peut connaître le temps qui s'écoule entre le lancement d'une commande et sa fin avec la commande time.

3.4.1 Le PID maximum

On a vu dans le chapitre sur le shell que le système attribue un PID à chaque processus en incrémentant un compteur. Une question à se poser est ce qui se passe quand ce compteur atteint la valeur maximum.

Plutôt que de trouver (difficilement) cette réponse dans la documentation ou en lisant les sources du noyau, on peut écrire un programme expérimental, qui va créer des processus jusqu'à ce que le PID du nouveau processus soit plus petit que celui du précédent : cela signifiera probablement que le précédent avait atteint le PID maximum.

Le programme suivant est une manière simple (et fausse) de faire :

```
/* ce-pidmax1.c
 Trouver le PID maximum (programme faux)
 */
# include <stdio.h>
# include <unistd.h>

int
main(){
  int oldpid = 0;
  int newpid;

  for(;;){
    newpid = fork();
    if (newpid == 0){
      // enfant : ne rien faire
      return 0;
    }
    // parent
    if (newpid < oldpid){
      printf("Le PID max semble etre %d\n", oldpid);
      return 0;
    }
    oldpid = newpid;
  }
```

}

Quand on l'exécute, il donne des résultats qui ne semblent pas cohérents :

```
$ gcc -g -Wall ce-pidmax1.c
$ a.out
Le PID max semble être 8880
$ a.out
Le PID max semble être 12636
$ a.out
Le PID max semble être 16392
```

Une rapide inspection du code met à jour qu'on a oublié que l'appel système `fork` peut échouer. Ajoutons un test dans le parent pour ce cas juste après l'appel à fork :

```
if (newpid < 0){
  perror("fork") ;
   return 1 ;
}
```

Maintenant, quand on exécute le programme, on est prévenu si la création du nouveau processus échoue. Effectivement :

```
$ a.out
fork : Resource temporarily unavailable
```

On peut considérer ceci comme une piqûre de rappel : *il faut toujours tester les valeurs renvoyées par les appels système pour détecter et traiter les erreurs.*

Ce qui provoque sans doute l'erreur, c'est que le processus parent crée trop de processus enfants avant que ceux-ci aient le temps de se terminer. Pour résoudre le problème, il suffira alors que le parent attende la fin de l'enfant, comme dans le programme (correct) suivant :

```
/* ce-pidmax3.c
 Trouver le pid maximum (programme corrige, avec wait)
 */
# include <stdio.h>
# include <unistd.h>
# include <assert.h>

int
main(){
  int oldpid = 0;
  int newpid;
  int t, etat;

  for(;;){
    newpid = fork();
    if (newpid == 0){
      // enfant : ne rien faire
      return 0;
    }
```

```
    // parent
    if (newpid < 0){
      perror("fork");
      return 1;
    }
    t = wait(&etat);
    assert(t >= 0);
    if (newpid < oldpid){
      printf("Le PID max semble etre %d\n", oldpid);
      return 0;
    }
    oldpid = newpid;
  }
}
```

Ici, j'utilise `assert` qui permet de vérifier simplement que la condition est remplie. Si ce n'est pas le cas, le programme sera interrompu violemment avec un message d'erreur. Cette fonction est pratique pour vérifier que les cas « *qui ne peuvent pas se produire* » ne se produisent effectivement pas.

Maintenant, quand on lance ce programme, on obtient toujours la même réponse :

```
$ a.out
Le PID max semble être 32767
$ a.out
Le PID max semble être 32767
```

Cette valeur de 32767 semble assez particulière pour faire confiance à ce résultat : il s'agit de $2^{15} - 1$, la plus grande valeur positive qu'on peut stocker dans un entier sur deux octets.

Le même programme tournant sur d'autres systèmes Unix (FreeBSD et Solaris) me donne des résultats différents.

3.4.2 Le temps nécessaire pour créer un processus

De quel ordre de grandeur est le temps nécessaire pour créer un processus ? On peut utiliser la programme précédent comme base pour le savoir : pour avoir un temps mesurable, créons par exemple 100 000 processus ; divisons le temps que cela prend par 100 000 et nous aurons une bonne approximation du temps pour créer un processus.

Le programme modifié est le suivant :

```
/* cf-forktime.c
 Combien de temps pour un fork ?
 */
# include <stdio.h>
# include <unistd.h>
# include <assert.h>

enum {
  Nfois = 100 * 1000,
};
```

```
int
main(){
  int i, t, etat, newpid;

  for(i = 0; i < Nfois; i++){
    newpid = fork();
    if (newpid == 0){
      // enfant : ne rien faire
      return 0;
    }
    // parent
    assert(newpid > 0);
    t = wait(&etat);
    assert(t == newpid);
  }
  return 0;
}
```

J'ai ici utilisé `assert` pour vérifier que fork ne produit pas d'erreurs. J'ai aussi défini la constante `Nfois` en utilisant un `enum`, parce que je cela trouve plus lisible qu'un `#define`.

Pour mesurer le temps nécessaire, pas besoin de sortir son chronomètre : il existe une commande `time` qui lance une commande, attend qu'elle se termine puis affiche le temps qu'elle a pris :

```
$ gcc -g -Wall cf-forktime.c
$ time a.out

real 0m16.352s
user 0m1.408s
sys 0m14.789s
$ time a.out

real 0m14.012s
user 0m1.216s
sys 0m12.737s
$ time a.out

real 0m16.056s
user 0m1.216s
sys 0m14.749s
```

La commande `time` affiche trois temps : le temps `real` est le temps qu'aurait mesuré un chronomètre entre le lancement de la commande et sa fin ; il peut varier largement pour une même commande en fonction de la charge de l'ordinateur. Le temps `user` est celui pendant lequel le processus a effectué des calculs. Le temps `sys` est celui pendant lequel le noyau du système a traité les appels système du processus. Notons que les valeurs sont susceptibles de varier d'un appel à l'autre, en fonction de ce qui se passe sur l'ordinateur (ce ne sont que des statistiques) ; en revanche, l'ordre de grandeur reste constant. Il faut donc, sur

mon ordinateur, environ 0,15 millisecondes pour créer et terminer un processus. Une autre façon de voir, c'est qu'on peut y créer environ 6500 processus par seconde.

Exercice 3.9 — (facile) Sur votre ordinateur, quels sont les résultats ?

3.5 La création d'un nouveau processus 2 : `exec`

L'appel système `fork` permet de créer un nouveau processus, mais celui-ci n'est qu'une copie presque à l'identique de l'ancien. Il est donc complété par un second appel système qui change le programme exécuté par un processus en laissant toutes ses autres caractéristiques inchangées : il s'agit de l'appel système `exec`.

On donne à l'appel système `exec` deux arguments (au moins) qui sont d'une part le fichier qui contient le programme, et d'autre part les arguments tels qu'ils apparaîtront dans `argc` et `argv`, les deux paramètres de la fonction `main`. Le système va modifier le processus pour qu'il commence à exécuter le programme contenu dans le fichier depuis le début, mais (presque) tous les autres paramètres du processus comme son PID, son répertoire courant, les fichiers qu'il a ouvert, etc. restent identiques.

Il existe plusieurs fonctions C qui permettent d'utiliser l'appel système `exec` ; les deux plus importantes sont `execl` et `execv`. Voici un programme simple qui utilise `execl` :

```
/* cg-exec-simple.c
 Un execl tout simple
 */
# include <stdio.h>
# include <unistd.h>

int
main(){
  int t;

  t = execl("/bin/echo", "commande", "arg1", "arg2", 0);
  if (t < 0){
    perror("execl");
    return 1;
  } else {
    printf("l'execl a fonctionne, ce message ne sera jamas affiche\n");
    return 0;
  }
}
```

On peut le compiler et l'exécuter :

```
$ gcc -g -Wall cg-exec-simple.c
$ a.out
arg1 arg2
$
```

On voit dans la sortie que la commande `echo` a bien été exécutée ; en revanche,le message annonçant la réussite n'apparaît pas. C'est normal : puisque l'`exec` a

fonctionné, le processus n'est plus en train d'exécuter ce programme, mais celui qui se trouve dans le fichier indiqué par le premier argument de `execl`.

Parce que `exec` remplace le programme en cours d'exécution, c'est (avec `exit`) un des rares appels système pour lequel il n'y a pas besoin de tester la valeur renvoyée. Si le programme qui appelle `execl` continue ensuite son exécution, c'est que l'exec n'a pas fonctionné ; il y a donc eu une erreur. Dans un programme, on peut écrire :

```
execl(fichier, nom, arg1, 0) ;
perror("exec") ;
exit(1) ;
```

3.5.1 La combinaison fork + exec

La combinaison des deux appels système `fork` et `exec` est la façon normale sous Unix de créer un nouveau processus en train d'exécuter un nouveau programme.

Ce qui est particulièrement intéressant dans la combinaison de ces deux appels, c'est la manière dont elle permet d'effectuer les modifications des caractéristiques du processus enfant : en général, c'est le processus parent qui sait de quelle manière l'environnement doit être modifié. Après le `fork`, le processus enfant exécute le programme du processus parent et peut donc modifier les caractéristiques du processus avant de lancer avec `exec` l'exécution du nouveau programme. Le schéma général est donc

```
t = fork() ;
assert(t != -1) ;
if (t == 0){
  // enfant
  modifier les caractéristiques de l'enfant
  execl(...) ;
  traiter l'erreur lors du exec
}
// parent
...
```

On verra un exemple pratique de ce modèle dans les redirections d'entrée-sortie.

Une fonction `spawn1` qui lance un nouveau processus en train d'exécuter un nouveau programme avec un argument peut s'écrire simplement :

```
/* ch-spawn.c
 Une fonction pour lancer un programme dans un nouveau proc.
*/
# include <stdio.h>
# include <unistd.h>
# include <sys/types.h>
# include <sys/wait.h>
# include <stdlib.h>

/* spawn1  --  lance un processus qui execute le fichier */
int
spawn1(char * fichier, char * commande, char * arg){
```

```
  int t;

  t = fork();
  if (t < 0)                    // erreur
    return -1;

  if (t == 0){                  // enfant
    execl(fichier, commande, arg, (void *)0);
    exit(1); // erreur dans l'enfant

  }
                                // parent
  return t;
}
```

Il y a deux étapes : on crée le processus avec `fork` puis on lui fait exécuter le programme avec `exec`.

On peut l'utiliser simplement comme dans l'exemple suivant, où on lance 10 fois la commande `mon-echo` que nous avons étudiée dans le chapitre sur le shell.

```
/* ci-spawn-main.c
 Un exemple d'utilisation de la fonction spawn1
*/
# include <stdio.h>
# include <assert.h>
# include <sys/types.h>
# include <sys/wait.h>

int spawn1(char *, char *, char *);

int
main(){
  int i, t, tt, etat;

  for(i = 0; i < 10; i++){
    t = spawn1("mon-echo", "nom bizarre", "ceci est un argument");
    if (t < 0){
      perror("fork"); // echec lors de la creation
      break;
    }
    tt = wait(&etat);
    assert(tt == t);
    if (etat != 0){
      perror("exec mon-echo"); // erreur lors de l'exec()
                               // ou dans la commande
      break;
    }
  }
  return i != 10;
}
```

Les trois arguments de `spawn1` sont le nom du fichier exécutable (ici `mon-echo`), un nom de commande et un argument. Un exemple d'utilisation :

```
$ gcc -g -Wall ba-echo.c -o mon-echo
$ gcc -g -Wall ch-spawn.c ci-spawn-main.c
$ a.out
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
Programme lancé sous le nom nom bizarre avec 1 arguments
argument 1 : ceci est un argument
$
```

Exercice 3.10 — Modifier le programme précédent pour donner à `spawn1` un nom de fichier qui n'existe pas ou qui n'est pas exécutable. Quel message d'erreur obtient-on ? Expliquer.

3.5.2 La variante `execv`

La fonction execl n'est pas très pratique si on ne connaît pas à l'avance le nombre d'arguments avec lequel il faut lancer la commande. La fonction `spawn1` de la section précédente ne permet de lancer que des commandes avec un argument. Pour les commandes avec deux arguments, il faudrait faire une autre fonction `spawn2` et ainsi de suite pour tous les nombres d'arguments possibles.

Une meilleure solution consiste à utiliser la fonction `execv` (avec 'v' comme *variable*, par opposition au 'l' comme *liste* de `execl`) dans laquelle on passe le nom de la commande et les arguments dans un tableau.

Nous pouvons réécrire notre programme qui utilisait `execl` avec `execv` comme dans :

```
/* cj-execv.c
 Une utilisation simple de execv
 */
# include <stdio.h>
# include <unistd.h>
```

```
char * tab[] = { "commande", "arg1", "arg2", 0, };

int
main(){
  execv("/bin/echo", tab);
  perror("en essayant de faire execv");
  return 1;
}
```

Exercice 3.11 — Écrire et tester une fonction `spawn` qui permet de lancer une commande avec un nombre variable d'arguments.

Réponse — Exercice 3.12 — (assez difficile) Quelle est le nombre ou la taille maximum des arguments qu'on peut spécifier dans un appel à exec.

3.6 Le `PATH`, les fonctions `execlp` et `execvp`

Nous avons vu dans le chapitre sur le shell que la variable d'environnement `PATH` contient la liste des répertoires où le shell va chercher s'il trouve un fichier exécutable. Quand on donne une commande, le shell va donc tenter d'exécuter un fichier qui porte le nom de la commande successivement dans chacun des répertoires.

Cette opération est tellement courante qu'il existe des fonctions de bibliothèque, `execlp` et `execvp`, qui font ce travail automatiquement.

Une réalisation de la fonction execvp pourrait être la suivante

```
/* cn-execvp.c
Une re-implementation de la fonction execvp
 */
# define _GNU_SOURCE // pour asprintf
# include <stdio.h>
# include <unistd.h>
# include <string.h>
# include <assert.h>
# include <stdlib.h>

int
monexecvp(char * file, char * av[]){
  char * p;
  char * path;
  int t;

  p = getenv("PATH");
  if (p == 0){
    execv(file, av);
    return -1;
  } else {
    p = strdup(p);
    assert(p != 0);
    for(p = strtok(p, ":"); p != 0; p = strtok(0, ":")){
```

```
      t = asprintf(&path, "%s/%s", p, file);
      assert(t != 0);
      execv(path, av);
      free(path);
    }
    free(p);
  }
  return -1;
}
```

Exercice 3.13 — (assez difficile) Un exercice d'un chapitre précédent demandait comment faire pour ajouter au `PATH` un nom de répertoire qui contient le caractère deux points (`':'`); réécrire la fonction `execvp` pour répondre correctement à cette question.

3.6.1 La fonction de bibliothèque `system`

Il existe une fonction `system` dans la bibliothèque usuelle qui permet de lancer une commande. Cette fonction lance (avec `fork` et `exec`) un shell et lui passe la commande en argument. Le shell à son tour lance, avec `fork` et `exec` dans la plupart des cas, la commande qu'on a passé comme argument à `system`.

L'intervention du shell signifie que les commandes lancées avec `system` peuvent utiliser toutes les fonctionnalités du shell, notamment le globbing et les redirections d'entrées-sorties ; pour cette raison, elle est la source de nombreux problèmes de sécurité.

Exercice 3.14 — (assez facile) Mesurer la différence entre les temps d'exécution de `system("true")` et `execl("/bin/true", "true", 0)`. (pour obtenir des mesures de temps significatives, il faudra l'exécuter de nombreuses fois).

Exercice 3.15 — Comparer l'effet de `system("echo * > toto")` avec celle de
`execlp("/bin/echo", "echo", "*", ">", "toto", 0)`.

3.6.2 Passage de l'environnement aux processus enfants

(section facultative)

En fait, lors d'un `exec`, on passe un troisième paramètre qui décrit l'environnement, tel que nous l'avons présenté dans le chapitre sur le shell. Il existe pour cela des variantes de `execl` et `execv` qui s'appellent `execle` et `execve` et qui prennent la description de l'environnement comme argument supplémentaire.

L'argument est décrit, dans la mémoire d'un programme, sous la forme d'un tableau de chaînes de caractères, comme `argv`. Chaque chaîne contient la définition d'une variable, comme imprimé par la commande `printenv`.

Quand un programme commence à s'exécuter, il reçoit l'environnent sous la forme d'un troisième paramètre de la fonction main, rarement représenté, qu'on appelle communément `envp`. On peut facilement réécrire la commande `printenv` comme dans le programme suivant :

```
/* cl-monprintenv.c
 Refaire le travail de printenv
*/
# include <stdio.h>
```

```
int
main(int ac, char * av[], char * envp[]){
  int i;

  for(i = 0; envp[i] != 0; i++)
    printf("%s\n", envp[i]);
  return 0;
}
```

Exercice 3.16 — Écrire une commande `unexport` qui permet de lancer une commande en ôtant une variable de l'environnement dans l'environnement du processus enfant sans toucher à celui du processus parent. Exemple d'utilisation :

```
$ printenv | grep LOGNAME
LOGNAME=jm
$ unexport LOGNAME printenv | grep LOGNAME
$
$ printenv | grep LOGNAME
LOGNAME=jm
```

3.7 Récapitulation : un shell simple

Pour donner un exemple complet d'utilisation de ces appels système, nous présentons ici un shell simple, qui permet d'entrer au clavier et de faire exécuter des commandes. Les exercices qui suivent proposent quelques extensions.

3.7.1 Découper une chaîne de caractères

Commençons par définir une fonction `decouper` qui nous permette de découper une chaîne de caractères en sous-chaînes. Elle utilisera la fonction de bibliothèque `strtok`.

```
/* cm-decouper.c
 Un wrapper autour de strtok
*/
# include <stdio.h>
# include <string.h>

/* decouper  --  decouper une chaine en mots */
void
decouper(char * ligne, char * separ, char * mot[], int maxmot){
  int i;

  mot[0] = strtok(ligne, separ);
  for(i = 1; mot[i - 1] != 0; i++){
    if (i == maxmot){
      fprintf(stderr, "Erreur dans la fonction decouper: trop de mots\n");
      mot[i - 1] = 0;
    }
    mot[i] = strtok(NULL, separ);
```

```
  }
}

# ifdef TEST
int
main(int ac, char * av[]){
  char *elem[10];
  int i;

  if (ac < 3){
    fprintf(stderr, "usage: %s phrase separ\n", av[0]);
    return 1;
  }

  printf("On decoupe '%s' suivant les '%s' :\n", av[1], av[2]);
  decouper(av[1], av[2], elem, 10);

  for(i = 0; elem[i] != 0; i++)
    printf("%d : %s\n", i, elem[i]);

  return 0;
}
# endif
```

La fonction reçoit une ligne de texte (en fait une chaîne de caractères) et la découpe en mots qui sont placés dans le tableau qu'on lui passe comme troisième argument. Le second argument est la liste des caractères qui peuvent servir de séparateurs, et le quatrième est le nombre d'emplacements disponibles dans le tableau de mots.

La définition de la fonction est accompagnée dans le fichier, par une fonction `main` qui permet de la tester. Grâce au `#ifdef` ··· `#endif`, la fonction main ne sera compilée que si on appelle le compilateur avec l'option `-DTEST`. Ce genre de *test unitaire* est utile pour s'assurer que chaque partie d'un programme fonctionne bien, indépendamment des autres. De plus, le fait de réfléchir à la manière dont on peut tester un module conduit en général à une meilleure répartition des fonctions entre les modules.

Compilons et faisons tourner le programme de test :

```
$ gcc -g -Wall -DTEST co-decouper.c      compiler
$ a.out                                  un test du message d'erreur
usage : a.out phrase separ
$ a.out 'x et y' ' '                     test simple
On découpe 'x et y' suivant les ' ' :
0 : x
1 : et
2 : y
$ a.out '' ' '                           découper une chaîne vide ?
On découpe '' suivant les ' ' :          ok
$ a.out 'x.....y' '.'                    plusieurs fois le séparateur ?
On découpe 'x.....y' suivant les '.' :
0 : x
1 : y                                    ok
$ a.out '...x.y....' '.'                 séparateur au début et à la fin
On découpe '...x.y....' suivant les '.' :
0 : x
1 : y                                    ok
$ a.out '..ceci.,,et,.cela,' '.,'        plusieurs séparateur ?
On découpe '..ceci.,,et,.cela,' suivant les '.,' :
0 : ceci
1 : et
2 : cela
```

Pour une fonction plus complexe, ça pourrait valoir la peine d'automatiser ces tests, de façon à pouvoir vérifier que ces cas restent correctement traités après une modification éventuelle de la fonction. On appelle ceci des *tests de (non) régression.*

3.7.2 La boucle principale

Le corps de la boucle principale lit une ligne de commande, la découpe (avec la fonction `découper` de la section précédente), fabrique un nouveau processus (avec `fork`), et tente d'y lancer la commande en allant chercher un fichier du même nom dans chacun des répertoires indiqué par `PATH`.

```
# include <stdio.h>
# include <stdlib.h>
# include <unistd.h>
# include <sys/types.h>
# include <sys/wait.h>
# include <assert.h>

enum {
  MaxLigne = 1024,                  // longueur max d'une ligne de commandes
  MaxMot = MaxLigne / 2,            // nbre max de mot dans la ligne
  MaxDirs = 100,                    // nbre max de repertoire dans PATH
  MaxPathLength = 512,              // longueur max d'un nom de fichier
};
```

```
void decouper(char *, char *, char **, int);

# define PROMPT "? "

int
main(int argc, char * argv[]){
  char ligne[MaxLigne];
  char pathname[MaxPathLength];
  char * mot[MaxMot];
  char * dirs[MaxDirs];
  int i, tmp;

  /* Decouper PATH en repertoires */
  decouper(getenv("PATH"), ":", dirs, MaxDirs);

  /* Lire et traiter chaque ligne de commande */
  for(printf(PROMPT); fgets(ligne, sizeof ligne, stdin) != 0; printf(PROMPT)){
    decouper(ligne, " \t\n", mot, MaxMot);
    if (mot[0] == 0)              // ligne vide
      continue;

    tmp = fork();                 // lancer le processus enfant
    if (tmp < 0){
      perror("fork");
      continue;
    }

    if (tmp != 0){               // parent : attendre la fin de l'enfant
      while(wait(0) != tmp)
        ;
      continue;
    }

                                  // enfant : exec du programme
    for(i = 0; dirs[i] != 0; i++){
      snprintf(pathname, sizeof pathname, "%s/%s", dirs[i], mot[0]);
      execv(pathname, mot);
    }
                                  // aucun exec n'a fonctionne
    fprintf(stderr, "%s: not found\n", mot[0]);
    exit(1);
  }

  printf("Bye\n");
  return 0;

}
```

On peut effectuer des tests simples de ce programme :

```
$ ls                              liste des fichiers avec le shell ordinaire
co-decouper.c co-main.c
$ gcc -g -Wall *.c                compilation
$ a.out                           lancement du nouveau shell
?                                 on voit le prompteur
? ls                              lancement d'une commande simple
a.out co-decouper.c co-main.c     ok
? echo *                          y a-t-il quelqu'un pour faire le globbing ?
*                                 non !
? pwd                             une autre commande ?
/tmp/essai                        ok
? exit                            sortir ?
exit : not found                  il n'y a pas de commande exit !
? ^D                              taper à la fois sur les touches CTRL et D.
Bye
$
```

Ce shell élémentaire est une occasion d'identifier finement la répartition du travail entre le shell, le noyau et les commandes. Ainsi `echo *` a simplement affiché une étoile au lieu de la remplacer par la liste des fichiers du répertoire courant, parce que notre shell ne remplace pas les modèles par les noms de fichiers qui correspondent. De même, la commande `exit` est introuvable, parce qu'il s'agit d'une commande *interne* au shell (qui lui indique de terminer le travail), et non pas d'une commande lancée via fork et exec.

Nous utiliserons ce shell comme base pour d'autres exercices quand nous parlerons d'entrées-sorties.

3.7.3 Exercices

Exercice 3.17 — (assez facile) Rajouter la possibilité de lancer des processus en arrière-plan quand la ligne de commande se termine par un &. (indication : il suffit de ne pas faire le `wait`).
La commande `cd` n'existe pas dans notre shell; pourtant il existe un appel système `chdir` qui permet de changer le répertoire courant. On pourrait l'utiliser dans une commande comme la suivante (exécutable à placer dans le fichier `moncd`) :

```
/* cp-moncd.c
 Pour faire un chdir (pas trop utilisable)
*/
# include <stdio.h>
# include <unistd.h>
# include <stdlib.h>

int
main(int ac, char * av[]){
  char * dir;
  int t;
                                  // traiter les arguments
  if (ac < 2){
```

```
    dir = getenv("HOME");
    if (dir == 0)
      dir = "/tmp";
  } else if (ac > 2){
    fprintf(stderr, "usage: %s [dir]\n", av[0]);
    return 1;
  } else
    dir = av[1];
                                  // faire le boulot
  t = chdir(dir);
  if (t < 0){
    perror(dir);
    return 1;
  }
  return 0;
}
```

On se contente de faire un `chdir` dans le répertoire donné en argument ou dans celui indiqué par `HOME` l'environnement comme la commande `cd` usuelle.

Exercice 3.18 — (facile) Pourquoi la commande `moncd` ne fonctionne-t-elle ni sous notre shell, ni sous `bash` ?

Exercice 3.19 — (moyen) Faire une commande interne `moncd` dans notre shell qui fonctionne.

Exercice 3.20 — (facile si l'exercice précédent a été bien fait) Rajouter une commande interne `exit`.

3.8 Les threads

Les processus présentent deux inconvénients majeurs : d'une part ils sont assez coûteux à créer (on a vu qu'on ne pouvait en créer que quelques milliers par secondes), d'autre part la communication entre les processus est assez complexe et délicate (comme nous le verrons dans le chapitre sur la communication entre les processus).

(Pour rendre le `fork` moins coûteux, certains systèmes Unix ont une variante, appelée `vfork` qui est plus rapide. Le système ne recopie que ce qui est indispensable des caractéristiques du processus parent, parce que l'enfant obtenu avec `vfork` va très prochainement effectuer un `exec` ; le parent reste bloqué jusqu'à l'`exec` de l'enfant.)

Pour pallier à ces inconvénients, il existe un autre niveau, entre programme et processus, qu'on appelle *processus légers* ou *threads*. Dans une famille de threads, les membres partagent (presque) toutes les caractéristiques d'un processus sauf le point d'exécution et les données sauvées dans la pile (donc les variables locales aux fonctions). En revanche le code, les variables globales, les fichiers ouverts sont partagés par tous les threads d'une même famille.

Un inconvénient des threads, c'est qu'il existe de nombreuses manières de les mettre en place, qui sont bien évidemment incompatibles entre elles. Une question importante est notamment de savoir jusqu'à quel point le noyau participe

à l'existence des threads (il est possible de les réaliser sans aucune modification du noyau).

Nous présentons ici sommairement la variante la plus répandue, qu'on appelle les `pthreads` (le 'p' vient de ce qu'ils ont été spécifiés par un comité de normalisation d'Unix qui s'appelle *Posix*).

3.8.1 Deux visions des threads

Il existe (au moins) deux manières de voir les threads : soit comme des sous-processus soit comme des fonctions parallèles.

Dans la vision comme des *fonctions parallèles*, chaque thread fonctionne comme une fonction ordinaire appelée par un processus. La fonction à accès, comme d'ordinaire, à ses variables locales et aux variables globales. Les threads ont cependant ceci de particulier qu'il est possible d'avoir, en même temps, plusieurs appels de fonctions ; ils se déroulent en parallèle. Le degré de parallélisme dans l'exécution dépend de la machine et de la façon dont les threads sont implémentés.

Dans la vision comme des sous-processus, on voit les threads d'une même famille comme des processus indépendants, qui ont en plus la particularité de partager de nombreuses caractéristiques entre eux, notamment la mémoire qui contient les variables globales.

3.8.2 Les threads : naissance, vie et mort

Voici un programme simple qui crée quelques threads :

```
/* cq-thread.c
 Simple creation de threads
*/
# include <stdio.h>
# include <assert.h>
# include <pthread.h>

enum {
  Nthread = 3,
  Nfois = 10,
};

static void *
foo(void * arg){
  int i, t;
  char * name = arg;

  for(i = 0; i < Nfois; i++){
    t = rand() % 5;
    printf("%s: je dors %d secondes\n", name, t);
    sleep(t);
  }
  printf("%s: j'ai fini, merci\n", name);
  return arg;
```

```
}

int
main(){
  static char * name[Nthread] = { "thread 0", "thread 1", "thread 2" };
  pthread_t id[Nthread];
  void * junk[Nthread];
  int i;

  printf("main: creation de %d threads\n", Nthread);
  for(i = 0; i < Nthread; i++)
    pthread_create(&id[i], 0, foo, name[i]);

  printf("On attend la fin des trois threads\n");
  for(i = 0; i < Nthread; i++){
    pthread_join(id[i], &junk[i]);
    printf("main: %s est termine\n", junk[i]);
  }
  return 0;
}
```

Il a fallu inclure le fichier `pthread.h` pour avoir les prototypes de fonction et les types de données spécifiques des threads, et il faut explicitement indiquer la bibliothèque qui contient le code de gestion des threads en compilant avec :

```
$ gcc -g -Wall cq-thread.c -lpthread
```

Un thread exécute une fonction principale comme un processus exécute la fonction `main`. Ici c'est la fonction `foo` pour tous les threads. Cette fonction reçoit un argument et renvoie une valeur qui est un pointeur sur n'importe quoi (en C, c'est un `void *`).

Ici la fonction `foo` se contente de dormir quelques fois, pendant des intervalles de temps pris au hasard.

On crée un thread avec la fonction `pthread_create` dont les arguments sont d'une part un emplacement où placer les informations sur le thread créé (du type `pthread_t`, il joue le même rôle que la valeur renvoyée par `fork` dans le processus parent), des paramètres facultatifs sur le thread avec lesquels on peut jouer un peu sur ses caractéristiques, la fonction principale du thread (ici `foo`) et son argument.

On attend la fin du thread avec la fonction `pthread_join` (qui joue un peu le même rôle que `wait` pour un processus) : on lui passe comme arguments l'identité du thread dont on veut attendre la fin et un emplacement où sera placée la valeur renvoyée.

Exercice 3.21 — Écrire un programme pour mesurer le temps de création d'un thread sur votre ordinateur, comme nous l'avons fait pour fork. Comparer les résultats.

Exercice 3.22 — Que se passe-t-il quand un thread appelle la fonction `exit` ? Est-ce que tous les threads du processus s'arrêtent ou bien seulement celui qui a appelé `exit` ? (indication : le plus simple est d'écrire un programme qui permette de répondre à la question).

Implémentation des threads dans les processus et le noyau

On peut implémenter les threads dans le noyau et il seront alors gérés comme des processus, mais le noyau va se compliquer.

On peut les implémenter dans les processus, mais il faut remplacer les appels système bloquant (comme celui utilisé par la commande `sleep`) par quelque chose qui active un autre thread pendant que celui qui effectue l'appel système est bloqué. D'autre part cela ne permet pas de tirer parties des processeurs multi-core.

Les threads : version simple

(section facultative)

On peut définir des threads d'une façon qui n'est pas si compliquée, mais il faut aller manipuler des choses de bas niveau. Nous présentons dans cette section une implémentation simple de threads minimaux.

Le code en C est le suivant :

```
/* cv-thread.c
 Des threads simplifies a l'extreme
*/
# include <stdio.h>
# include <stdlib.h>

typedef struct Ctxt Ctxt;

enum {
  StackSize = 1024,             // taille de la pile
};

/*
  Le contexte d'un thread
 */
struct Ctxt {
  int ebp;                      // frame pointer
  int esp;                      // stack pointer
  int edi, esi, ebx;            // registres a sauver : edi, esi, ebx
                                // eax, ecx edx ont ete sauves par l'appelante
  char * name;                  // purement informatif
  int stack[StackSize];         // de l'espace pour la pile
};

Ctxt ctxt[2];                   // deux threads
Ctxt * curthread;               // le thread courant
Ctxt original;                  // le thread original

void save(Ctxt*);               // en assembleur dans swtch.s
void restore(Ctxt*);

void sched(void);               // dans la suite du code
void createthread(int num, void (*fun)(void), char * name);
```

```
void run(void);
static void endthread(void);

int globi = 0;;

/* principal  --  un thread elementaire */
void
principal(){
  int i;

  for(i = 0; i < 5; i++){
    printf("Je suis %s, i vaut %d, globi %d\n",
           curthread->name, i, globi++);
    sched();
  }
}

int
main(){
  createthread(0, principal, "thread 0");
  createthread(1, principal, "thread 1");
  run();
  return 0;
}


/* sched  --  elementaire : round robin avec deux taches */
void
sched(){
  static int tour;

  save(curthread);
  curthread = &ctxt[tour ^= 1];
  restore(curthread);
}

/* endproc  --  fin d'un processus : on arrete tout */
static void
endthread(){
  printf("endthread: on rentre dans la fonction\n");
  restore(&original);
  printf("endthread: impossible\n");
}

/* run  --  lance le premier thread */
void
run(){
  curthread = &original;
  original.name = "Original";
  save(&original);
```

```
  curthread = &ctxt[0];
  restore(&ctxt[0]);
  printf("run: c'est termine\n");
}

/* createthread  --  fabrique un thread : numero, fonction, nom */
void
createthread(int num, void (*fun)(void), char * name){
  int * p;

  p = &ctxt[num].stack[StackSize - 1];  // p sur la fin de la pile
  *p-- = (int)endthread;
  *p = (int)fun;                  // la fonction comme adresse de retour
  ctxt[num].esp = (int)p;
  ctxt[num].name = name;
}
```

Le fichier définit une structure `Ctxt` qui définit le contexte d'un thread. Elle contient de l'espace pour sauver les registres, le nom du thread et de la mémoire pour contenir la pile.

Dans notre exemple, il y a trois contextes : celui des deux threads que nous utilisons dans notre programme et le contexte original, qui sera utilisé pour sortir.

La fonction `principal` est la fonction avec laquelle les threads sont lancés. Dans notre exemple, elle se contente de faire quelques tours dans une boucle en affichant la valeur de deux variables, l'une locale et l'autre globale, qu'elle incrémente à chaque tour. Notre fonction appelle explicitement la fonction `sched` qui va activer le thread suivant.

Finalement la fonction `main` fabrique les deux threads (avec `createproc`), les lance (avec `run`). Quand les threads on terminé leur travail, elle reprend la main et s'arrête proprement.

Les fonctions suivantes pourraient être placées dans une bibliothèque. La fonction `sched` sauve l'état du thread courant, en choisit un autre et l'active (comme on n'a que deux threads, le choix est facile). La fonction `endthread` n'est pas utilisée, mais elle permettrait de terminer tous les threads. La fonction `run` est appelée pour le lancement des threads : elle sauve le contexte du processus (pour `endproc`) et installe le contexte du premier thread.

La fonction `createthread` sert à initialiser le contexte d'un thread. Elle se livre à des manipulations de pile peu orthodoxes qui ne sont pas détaillées ici.

Il nous manque les deux fonctions `save` et `restore` qui sont utilisées pour sauver et réinstaller le contexte d'un thread ; ces deux fonctions sont écrites en assembleur, comme dans ce qui suit :

```
#  %ebp (pointeur de frame)
#  %esp (pointeur de pile)
#  %edi, %esi, %ebx (qui peuvent avoir des valeurs a conserver)
#  %eax, %ecx, %edx ont ete sauves par l'appelante si necessaire
        .text
        .globl save, restore
```

```
# a appeler avec "save(ctxt)"
save:
        movl    4(%esp),%eax    # eax pointe sur le contexte
        movl    %ebp,0(%eax)    # sauve les registres dans old
        movl    %esp,4(%eax)
        movl    %edi,8(%eax)
        movl    %esi,12(%eax)
        movl    %ebx,16(%eax)
        ret
# a appeler avec "restore(ctxt)"
restore:
        movl    4(%esp),%eax    # eax pointe sur le contexte
        movl    0(%eax),%ebp    # remplit les registres avec new
        movl    4(%eax),%esp
        movl    8(%eax),%edi
        movl    12(%eax),%esi
        movl    16(%eax),%ebx
        ret
```

Le rôle de ces deux fonctions est simplement de copier le contenu des registres du processeur dans la mémoire (pour `save`) ou de remettre la copie qui est dans la mémoire à l'intérieur des registres (pour `restore`)/

On peut compiler le programme qui se trouve dans les deux fichiers avec `gcc -g -Wall cu-swtch.s cv-thread.c`; le compilateur se charge de combiner le C et l'assembleur (comme indiqué par le nom des fichiers). On peut ensuite lancer le programme :

```
$ gcc -g -Wall cu-swtch.s cv-thread.c
$ a.out
Je suis thread 0, i vaut 0, globi 0
Je suis thread 1, i vaut 0, globi 1
Je suis thread 0, i vaut 1, globi 2
Je suis thread 1, i vaut 1, globi 3
Je suis thread 0, i vaut 2, globi 4
Je suis thread 1, i vaut 2, globi 5
Je suis thread 0, i vaut 3, globi 6
Je suis thread 1, i vaut 3, globi 7
Je suis thread 0, i vaut 4, globi 8
Je suis thread 1, i vaut 4, globi 9
endthread: on rentre dans la fonction
run: c'est terminé
$
```

On voit que les deux fonctions `principal` tournent à tour de rôle. Chacune possède sa propre variable locale `i` et elles partagent la variable globale `globi`.

Exercice 3.23 — (difficile) Ce programme ne fonctionne que sur les processeurs Intel 32 bits. Le modifier pour qu'il tourne sur les processeurs Intel 64 bits. (Il n'y a pas de corrigé pour cet exercice car je n'ai pas encore fait ce travail.)

3.8.3 L'appel système `clone` de Linux

Sous Linux, le noyau connaît les threads, qui sont créés avec un appel système appelé `clone`. Cet appel système permet de spécifier quelles caractéristiques du processus sont partagées entre le parent et l'enfant, lesquelles seront recopiées et lesquelles seront réinitialisées.

Chapitre 4

La généalogie des processus

Dans le premier chapitre, nous nous sommes arrêtés au moment où le noyau, ayant terminé ses initialisations, lance le premier processus qui s'appelle `init` et qui est l'ancêtre de tous les autres processus. Nous allons ici examiner sa descendance.

Attention, les opérations de cet aspect varient beaucoup d'un système à l'autre, et même d'une distribution de Linux à une autre. Nous décrivons dans ce chapitre ce qui se passe avec Mandriva, mais l'organisation est assez différente avec Ubuntu et Gentoo.

Une fois le chapitre assimilé, vous comprendrez la répartition du travail entre les processus présents sur une station de travail. Vous saurez où regarder pour examiner les démons lancés par votre ordinateur et votre compréhension des mécanismes sous-jacents vous permettra de maîtriser facilement l'outil d'administration qui permet de les contrôler. Vous comprendrez l'organisation des processus quand X11 est lancé et vous pourrez utiliser toutes les consoles virtuelles de votre ordinateur.

4.1 Le processus `init`

Le processus `init` est le seul processus créé par le noyau. Tous les autres en seront des descendants, créés avec des `fork` et des `exec`.

4.1.1 Les runlevels

Pour init, le système peut se trouver dans plusieurs configurations différentes, qui s'appellent des *runlevels* (niveau d'exécution) et qui correspondent à des services différents. Un niveau correspond à un mode dégradé dans lequel on se place pour les opérations de maintenance ; pour des raisons historiques, il s'appelle le niveau *single user* (un seul utilisateur). Par opposition, les autres niveaux sont dits *multi-user* (multi-utilisateurs) et correspondent à des fonctionnements normaux : on a un niveau sans réseau, un niveau avec réseau et un niveau avec réseau et interface graphique.

Init permet de passer d'un niveau à l'autre, en configurant des périphériques et en lançant des processus.

Chaque niveau est encodé par un caractère, soit un chiffre entre 0 et 6, soit 's' ou 'S'.

Le comportement d'`init` est contrôlé par un fichier de configuration nommé `/etc/inittab` que nous examinons dans la section suivante.

4.1.2 Le fichier `/etc/inittab`

Le fichier `/etc/inittab` est constitué de texte. si bien qu'on peut le consulter avec les outils usuels comme `cat` ou `less` et le modifier avec n'importe quel éditeur de texte (mais seul l'utilisateur `root`, qui joue le rôle de l'administrateur a les autorisations nécessaires pour l'éditer).

En dehors des caractères dièses # qui servent à introduire des commentaires, le fichier est organisé sous forme de lignes indépendantes ; chaque ligne contient différents champs, séparés par des deux points (' :'). Une ligne contient normalement quatre champs. Par exemple dans :

```
1 :2345 :respawn :/sbin/mingetty tty1
```

le premier champs (ici 1) est un nom sans signification particulière ; le dernier champs, du dernier ' :' à la fin de la ligne, est la commande à lancer ; le second champs est une liste des runlevels où la ligne s'applique ; ici la ligne est pertinente pour tous les runlevels de 2 à 5 ; le troisième champs donne des précisions sur les moments où la commande doit être lancée et sur la façon de le faire.

Le processus init lit le contenu de son fichier de configuration et exécute ce qui est indiqué dedans. Il contient de nombreuses références à d'autres fichiers qui se trouvent dans la branche `/etc` du système de fichier.

4.1.3 L'initialisation du système

Au départ, il y a un certain nombre de tâches à effectuer pour initialiser les périphériques et lancer des programmes qui vont tourner jusqu'à l'arrêt de la machine. Elles sont indiquées par des lignes dont le deuxième champs contient `sysinit`, `boot` ou `bootwait`. Mandriva n'utilise que le mot `sysinit`. On peut facilement voir la ligne exécutée pour ce mot avec la commande :

```
$ grep :sysinit /etc/inittab
si : :sysinit :/etc/rc.d/rc.sysinit
$ grep :boot /etc/inittab
$
```

La ligne indique qu'à l'initialisation du système, le processus init lance la commande qui s'appelle `rc.sysinit` dans le répertoire `/etc/rc.d`. Examinons le type de ce fichier de commandes :

```
$ file /etc/rc.d/rc.sysinit
/etc/rc.d/rc.sysinit : Bourne-Again shell script text executable
```

La commande `file` nous dit qu'il s'agit d'un script pour le shell `bash` ; nous pouvons donc l'examiner, par exemple avec `cat`, `less` ou un éditeur de texte quelconque. Attention, le fichier est long et compliqué : il contient les commandes qui servent à initialiser toutes sortes de choses dont le noyau du système ne se charge pas. Il fait plus de 1400 lignes, sans compter les scripts ancillaires

auxquels il fait appel. La plupart de ces commandes affichent un compte rendu qu'on peut voir en pressant sur la touche `Escape` sous Mandriva pendant le boot.

4.1.4 Le choix du runlevel par défaut

Une ligne du fichier `/etc/inittab` contient le runlevel dans lequel doit se placer la commande `init` par défaut. Cette ligne contient `initdefault` dans le troisième champs. On peut l'extraire avec :

```
$ grep initdefault /etc/inittab
...
id :5 :initdefault :
```

Les différents runlevels qu'on peut choisir aisément sont indiqués dans un commentaire au début du fichier `/etc/inittab` de Mandriva :

```
# Default runlevel. The runlevels used by Mandriva Linux are :
# 0 - halt (Do NOT set initdefault to this)
# 1 - Single user mode
# 2 - Multiuser, without NFS (The same as 3, if you do not have networking)
# 3 - Full multiuser mode
# 4 - unused
# 5 - X11
# 6 - reboot (Do NOT set initdefault to this)
```

Les runlevels qui nous intéressent sont le 1, le 2, le 3 et le 5. Le runlevel 1 est utilisé pour remettre le système en état après un incident : il y a un shell lancé sur la console, et c'est tout. Le runlevel 2 correspond à un fonctionnement presque normal, mais sans réseau et sans écran graphique. Dans le runlevel 3, il y a le réseau mais la console graphique n'est pas initialisée. Finalement, le runlevel 5 correspond au fonctionnement habituel, avec l'écran graphique, où la répartition des tâches entre les processus est détaillée plus loin dans le chapitre.

Exercice 4.1 — Faire une copie de sauvegarde du fichier `/etc/inittab`, modifier l'état par défaut pour le runlevel 3, rebooter la machine puis réussir à réinstaller la version originale du fichier avant de rebooter.

Exercice 4.2 — On peut forcer `init` à passer dans un runlevel avec la commande `telinit`. Passer dans le runlevel 1. Revenir dans le runlevel normal.

4.1.5 L'entrée dans un état

Lors de l'entrée dans un runlevel, `init` a un certain nombre de tâches à effectuer, notamment des processus à arrêter et d'autres à lancer. Ceci est indiqué par des lignes du fichier `/etc/inittab` qui suivent celle qui lance le script d'initialisation, dans le fichier `inittab` de Mandriva. Il y a une ligne par runlevel de 0 à 6, toutes de la forme :

```
l0 :0 :wait :/etc/rc.d/rc 0
```

Ce qu'elles indiquent à init, c'est qu'il doit lancer le programme `/etc/rc.d/rc` avec comme argument le numéro du runlevel. Le travail que fait ce script est essentiellement d'arrêter tous les processus inutiles et de lancer tout ceux qui sont nécessaires dans le runlevel. Le `wait` dans le troisième champs de la ligne fait que `init` attendra la fin de la commande pour passer à la suite.

La liste des processus à traiter dans un runlevel n se présente sous la forme d'un répertoire, nommé `/etc/rc.d/rc`n`.d`, qui contient des fichiers dont le nom est soit `K`*nn*`nom` pour les processus à tuer ('k' comme *kill*), soit `S`*nn*`nom` pour les processus à démarrer ('s' comme *start*). Les fichiers `K` sont traités en premier, puis les fichiers `S`, dans l'ordre alphabétique : les deux chiffres permettent de forcer l'ordre d'exécution des opérations.

Chaque fichier est une copie d'un script qui se trouve dans le répertoire `/etc/rc.d/init.d`, qui lance les commandes nécessaires pour installer ou arrêter un service, suivant l'argument avec lequel il est lancé. (En réalité, le répertoire ne contient pas vraiment une copie mais un *lien symbolique* vers le script de `init.d`; nous verrons plus loin ce qu'est un lien symbolique.)

Le script `/etc/rc.d/rc`, lancé par `init`, va principalement lancer chacun des scripts dont le nom commence par `K` avec l'argument `stop`, puis tous ceux dont le nom commence par `S` avec l'argument `start`. Une version simple de `/etc/rc.d/rc` pourrait être :

```
runlevel=$1
cd /etc/rc.d/rc$runlevel.d
for i in K* ; do
     $i stop
done
for i in S* ; do
     $i start
done
```

Le vrai script `rc` de Mandriva est un peu plus compliqué, principalement à cause de la possibilité d'une entrée *interactive* dans le runlevel où une confirmation sera demandée avant le lancement de chaque commande, et de la nécessité d'afficher un compte-rendu de l'exécution de chaque commande.

Pour créer un service `toto`, il faut donc faire un script de lancement et d'arrêt du service dans la répertoire `/etc/rc.d/init.d`. Pour lancer le service dans le runlevel n, on nomme (une copie de) ce script `S99toto` dans le répertoire `rc`n`.d` et `K99toto` dans les autres. Les interfaces graphiques d'administration du système qui proposent de lancer ou d'arrêter un service se contentent essentiellement d'ajouter ces noms de fichiers.

4.1.6 Entrées diverses de `inittab`

Dans cette section, on regarde quelques autres entrées du fichier inittab de Mandriva.

```
ca : :ctrlaltdel :/sbin/shutdown -t3 -r now
```

L'entrée de type `ctrlaltdel` est activée quand les trois touches CTRL Alt et Del sont appuyées. L'entrée indique qu'il faut rebooter l'ordinateur, mais on peut remplacer le `-r` par `-h` pour l'arrêter, ou supprimer la commande pour ne rien faire.

```
pf : :powerfail :/sbin/shutdown -f -h +2 "Power Failure ; System Shutting Down"
pr :12345 :powerokwait :/sbin/shutdown -c "Power Restored ; Shutdown Cancelled"
```

Les deux entrées de type `powerfail` et `powerokwait` sont activées par le système hard qui gère les batteries : un peu avant qu'il n'y ait plus d'énergie, `init` appelle la commande `shutdown` pour arrêter proprement l'ordinateur ; si le courant revient (par exemple parce qu'on a rebranché le portable sur le secteur), alors l'arrêt est annulé. Le terme `UPS` du commentaire faire référence aux systèmes de batterie de sauvegarde utilisés dans les gros centres de calcul pour protéger les serveurs contre les coupures de courant.

```
:S :wait :/bin/sh
```

Cette entrée correspond au mode de maintenance : `init` se contente de lancer un shell sur la console et d'attendre qu'il se termine.

Exercice 4.3 — Quand init est dans le run-level *single user* et que le shell se termine (par exemple parce qu'on tape `exit`, que fait init ? Passe-t-il à un autre *run-level* ou bien relance-t-il un shell ? (Préciser comment vous avez trouvé la réponse.)

4.1.7 Les consoles virtuelles et `getty`

Tous les runlevels ordinaires (de 2 à 5) permettent de travailler sur des *consoles virtuelles* : des écrans dans lesquels on travaille comme autrefois sur les terminaux. Ces terminaux *virtuels* utilisent le clavier et l'écran *réels* de l'ordinateur, et on contrôle le terminal virtuel actif avec la combinaison de touche `CTRL, ALT` et `F`*n*.

Les lignes du fichier `inittab` qui indiquent à init comment il faut se servir de ces terminaux virtuels sont :

```
# Run gettys in standard runlevels
1 :2345 :respawn :/sbin/mingetty tty1
2 :2345 :respawn :/sbin/mingetty tty2
3 :2345 :respawn :/sbin/mingetty tty3
4 :2345 :respawn :/sbin/mingetty tty4
5 :2345 :respawn :/sbin/mingetty tty5
6 :2345 :respawn :/sbin/mingetty tty6
```

Le second champs de la ligne contient `2345` pour indiquer que les lignes sont actives dans n'importe lequel des runlevels de 2 à 5. Le troisième champs vaut `respawn` pour indiquer qu'une fois que la commande est terminée, il faut la relancer ; c'est grâce à cela que quand on termine une session sur un terminal virtuel (par exemple avec `logout`), on peut de nouveau s'identifier avec son nom de login et son mot de passe. La fin de la ligne indique la commande à lancer. Ce point est développé dans la section suivante.

4.2 Du login à la sortie

Dans cette section, on présente la succession des programmes lancés lors d'une session sur un terminal virtuel. Ceci démarre avec une variante de la commande `getty` lancée par le processus `init`, continue avec `login` puis le `shell`.

4.2.1 Lire : `getty`

Le premier programme est lancé par `init` avec un `fork` et un `exec`. Comme on l'a vu dans la section précédente, lorsque le processus dans lequel ce programme est lancé se terminera, `init` le relancera de nouveau parce que la ligne dans `inittab` contient le mot clef `respawn` dans le troisième champs.

Le rôle du programme `getty` est d'initialiser les entrées-sorties ; il ouvre le terminal qu'on lui a donné en argument comme entrée standard, sortie standard et sortie d'erreur, il affiche un message (normalement le contenu du fichier `/etc/issue` puis le message `login :`) et s'efforce de lire une ligne de caractères : une fois que la ligne est lue, son travail se résume à lancer la commande `/bin/login` avec la chaîne lue comme unique argument. Le point important ici est que `/bin/login` est lancé avec un `exec`, mais pas de `fork` : il remplace le programme `getty` *dans le même processus*, et donc le processus parent `init` ne reçoit par de notification de fin du processus `getty`.

La commande `getty` existe en plusieurs versions : la distribution Mandriva utilise `mingetty` par défaut, mais on trouve aussi couramment un `agetty`. Chacune de ces versions fait le même genre de travail, il n'y a que les détails qui changent.

4.2.2 Authentification : `login`

Le rôle du programme `/bin/login` est de vérifier l'identité de l'utilisateur, qui lui a normalement été passée par `getty` comme premier argument et de lancer le shell. La section décrit la version simple de `/bin/login` ; des choses plus complexes (`nsswitch.conf` et `pam`) sont mentionnées plus loin dans le chapitre.

Dans sa version simple, `login` affiche un prompt (`Password :`), supprime l'écho des caractères tapés et lit un mot de passe. L'écho est supprimé pour empêcher qu'on lise le mot de passe par dessus votre épaule, ou simplement qu'on découvre le nombre de caractères qu'il contient.

Le couple (nom de login, mot de passe) est vérifié dans un ou deux fichiers qui contiennent la liste des informations sur les utilisateurs : `/etc/passwd` et `/etc/shadow`.

Le fichier /etc/passwd

Le fichier principal est le fichier `/etc/passwd` : il décrit un utilisateur par ligne, dans des champs séparés par des deux points. J'obtiens la ligne qui me décrit sur ma machine avec :

```
$ grep jm /etc/passwd
jm :x :500 :500 :Jean Mehat :/home/jm :/bin/bash
```

Le premier champs est mon nom de login (`jm`), le second (ici `x`) devrait être mon mot de passe, le troisième et le quatrième sont mon numéro d'utilisateur et mon numéro de groupe, qui sont utilisés en interne par le noyau pour m'identifier ; le cinquième peut contenir n'importe quoi, mais on y place en général le nom complet de l'utilisateur décrit par la ligne ; on l'appelle parfois encore le champs GCOS ; les deux derniers champs contiennent le répertoire de login et le shell à utiliser.

Pour vérifier le mot de passe, la commande `login` cryptait le mot de passe qu'on lui a fourni, en utilisant le mot de passe lui-même comme clef d'encryption, et le comparait avec le champs *mot de passe* du fichier `/etc/passwd`. Cela signifie que le mot de passe en clair n'était stocké nulle part et ne pouvait donc pas être volé ; cela signifiait aussi qu'un mot de passe oublié ne pouvait pas être retrouvé.

Ce schéma d'authentification a été victime d'attaques *à base de dictionnaires*. On prend un gros dictionnaire, on crypte chacun des mots qu'il contient et chacune des variantes de ces mots et on compare avec les mots de passe cryptés : on trouve facilement des programmes qui font ça, par exemple *Crack* ou *Jack the Ripper*.

Le fichier `/etc/shadow`

Pour éviter les attaques à base de dictionnaire, il faut que seul les programmes qui ont besoin d'effectuer l'authentification puissent accéder au mot de passe. Cependant, il n'est pas possible d'interdire la lecture du fichier `/etc/passwd` aux programmes ordinaires, parce que c'est lui qui est utilisé pour faire la liaison entre les numéros d'utilisateur et les noms ; si `/etc/passwd` n'est pas lisible, la commande `ls` par exemple affichera les numéros d'utilisateur des propriétaires des fichiers au lieu d'afficher leurs noms.

La solution qui a été adoptée consiste à ce que le fichier `/etc/passwd` continue à avoir la même structure que celle décrite auparavant, mais ne contienne plus les mots de passe cryptés, seulement un 'x' à la place. Les mots de passe cryptés sont conservés dans un autre fichier, nommé `/etc/shadow`, qui n'est pas lisible par tout le monde.

Pour regarder le fichier `/etc/shadow`, il faut obtenir les autorisations de l'administrateur Linux (on dit aussi l'utilisateur `root` ou le *super-utilisateur*), ce qui se fait avec la commande `su` qui vérifie le mot de passe de l'administrateur (que vous avez choisi lors de l'installation du système) et lance un shell avec les bonnes autorisations. Pour bien montrer qu'on est dans un état spécial, le shell affiche un prompteur spécial `#`

```
$ grep jm /etc/shadow              tentative d'accès au fichier :
/etc/shadow : Permission denied    échec
$ su                               lancer un shell avec les droits de l'administrateur
Password :                         il faut taper le mot de passe de l'administrateur
# grep jm /etc/shadow              tentative d'accès
jm :$1$aXA2KlHg$IhFNOpxwkwWpi5kq6Ivm70 :13933 :0 :99999 :7 : : :
                                   réussi
# exit                             sortie du shell de l'administrateur
$                                  retour au shell de départ
```

Le fichier `/etc/shadow` a le même genre de structure que le fichier `/etc/passwd` : chaque ligne décrit un utilisateur, avec des champs séparés par des deux points. Les deux premiers champs contiennent le nom de l'utilisateur et son mot de passe crypté. Les autres champs contiennent des dates, et sont utilisés pour avoir des comptes inactifs ou pour forcer les utilisateurs à changer leurs mots de passe.

Exercice 4.4 — Créer un utilisateur toto via l'outil d'aministration du système. Quelles modifications aux fichiers `/etc/passwd` et `/etc/shadow` cela a-t-il

provoqué ? Le fichier `/etc/group` a-t-il aussi été modifié ? (Une copie de ces fichiers et l'usage de la commande `diff` aident à trouver simplement la réponse à cette question).

D'autres mécanismes d'authentification

Le mécanisme d'authentification à base de fichiers locaux à une machine est bien adapté pour un ordinateur isolé, mais se révèle complètement inadéquat dans le cas d'un réseau d'ordinateurs : avec les fichiers de configuration, il serait nécessaire par exemple de modifier son mot de passe sur chacun des ordinateurs du réseau. alors qu'évidemment on ne veut le faire qu'une seule fois pour tout le réseau.

Pour résoudre ce problème, diverses solution ont été trouvées. La plus commune s'appelait les *yellow pages* et a été renommée *Network Information Service* (nis) : une machine centrale contient des enregistrements, dans le même format que `/etc/passwd` et un démon fournit à la demande à la commande `/bin/login` les enregistrements de cette base de données.

Une autre solution est fondée sur un protocole compliqué appelé LDAP (comme *Lightweight Directory Application Protocol*, qui a servi de base pour l'*Active Directory* de Microsoft.

La commande `ssh` qui permet de se connecter aux ordinateurs distants utilise également ces mécanismes et d'autres qui lui sont propres, fondés sur des systèmes d'encryption à clef publique.

Une tentative existe pour choisir le mécanisme d'authentification des utilisateurs, ainsi que pour d'autres bases de données système. Le fichier `/etc/nsswitch.conf` contient, pour chaque base de donnée du système, une ligne qui indique une liste de mécanismes à employer.

Notez que pour une raison que j'ignore, certaines installations de Mandriva placent dans ce fichier la ligne erronée : `hosts : files nis dns` qui empêche de résoudre proprement les noms de domaines. Il convient de la remplacer par `hosts : files dns`

Lancement du shell

Une fois que login a vérifié l'identité de l'utilisateur il se place dans le répertoire de login (indiqué dans le fichier `/etc/passwd`) et il fait un exec du shell, sans faire de fork. Du point de vue de la commande `init`, le processus qu'elle a lancé n'est donctoujours pas terminé (quoiqu'elle ait lancé `getty` qui s'est transformé en `login` qui devient maintenant le shell). Puisque ces transformations se sont faites avec des exec mais sans fork, c'est toujours le même processus.

4.2.3 Le shell

Au départ, le shell commence par exécuter les commandes qu'il trouve dans des fichiers de configuration propres au système et à l'utilisateur, qu'il recherche dans `/etc/profile` et dans `.profile` dans le répertoire de login. `Bash` permet de remplacer ce dernier fichier par un fichier nommé `.bash_profile` ou `.bash_login` et exécute de plus les commandes qui se trouvent dans le fichier `.bashrc` chaque fois qu'on le lance, et celles contenues dans le fichier `.bash_logout` quand il se termine. Un conseil personnel quant à ces multiples

fichiers de configuration : n'utilisez que le fichier `.profile` qui est commun à `sh` et `bash`, sinon on arrive à perdre plus de temps à configurer bash que la configuration ne permet d'en gagner. Une fois que les fichiers de configuration sont lus, bash en mode interactif rentre dans la boucle de commande.

Comme on l'a vu dans le chapitre précédent, le fonctionnement le plus ordinaire du shell consiste à lire une commande, à fabriquer un nouveau processus avec `fork` puis à lancer la commande dans ce nouveau processus avec `exec`.

Il existe de nombreuses manières de terminer le shell : l'une d'entre elles consiste à lancer la commande interne `exit` qui demande au shell de se terminer. Une autre manière consiste à taper un CTRL D en tête d'une ligne vide, ce qui lui fait le même effet que s'il lisait les commandes dans un fichier et qu'il était arrivé au bout du fichier. On peut aussi préfixer une commande avec le mot `exec` ; ceci sera interprété par le shell comme une demande de lancer la prochaine commande avec un `exec`, sans faire de `fork` auparavant. Quand la commande se termine, le processus disparaît, `init` reçoit la notification que ce processus enfant s'est terminé et relance la commande `getty`.

4.3 Le lancement de X11

Quand `init` entre dans l'état 5 pour lancer directement une session graphique, les choses se passent d'une façon un peu différente : le travail est réparti entre trois programmes principaux qui sont le *serveur X11*, le *display manager* et le *window nanager*.

4.3.1 Le lancement du serveur

Il y a un programme spécialisé dans la gestion du graphique, le *serveur X11*, qui est lancé au départ : il prend en main ce qui passe sur l'écran, le clavier et la souris (et sur les autres périphériques d'interaction, s'il y en a et qu'il est correctement configuré).

A partir de là, tous les programmes qui souhaitent faire des lectures ou des écritures vont devoir se connecter au serveur X11 et lui envoyer des requêtes dans un format particulier (et compliqué) qu'on appelle le *protocole X11*.

L'identification de l'utilisateur est faite par un autre programme, le *display manager*. Après cette étape, le programme principal est en général le *window manager*.

4.3.2 Le display manager

Le display manager se charge de l'authentification et l'utilisateur comme la commande `/bin/login`. Il procède de la même manière, en vérifiant le mot de passe crypté dans le fichier `/etc/shadow`. Sur un ordinateur personnel, il arrive parfois qu'on configure le display manager pour qu'il saute l'étape d'authentification et se comporte comme si le nom de l'utilisateur (unique) et le mot de passe corrects avaient été entrés.

Il y a trois principaux display managers : `xdm` est le plus ancien. Kde et Gnome offrent aussi chacun un display manager qui leur est propre et qui s'appellent respectivement `kdm` et `gdm`.

Une fois que l'utilisateur s'est authentifié, le display manager lance un programme qui va gérer la *session* de travail de l'utilisateur. Certains display managers permettent de choisir la session lors du login. `Xdm` lance, via fork et exec, un shell qui exécute les commandes présentes dans le fichier `.xsession` dans le répertoire de login de l'utilisateur, s'il est présent.

Quand le programme lancé par le display manager se termine, celui-ci réinitialise le serveur X11 et redemande à un utilisateur de s'identifier.

En général, le programme lancé par le display manager est un *window manager*, dont nous expliquons le rôle dans la section suivante.

4.3.3 Le window manager

Le serveur X11 se contente de gérer les périphériques graphiques. Les manipulations avec la souris pour créer, déplacer, détruire des fenêtres etc. sont gérées par un programme spécifique, le *window manager* (gestionnaire de fenêtres). Ce programme est indépendant du serveur, ce qui permet d'utiliser sous X11 des environnements très différents.

Il existe une variété phénoménale de window managers. Les plus courant sont celui de Gnome et celui de Kde, mais il est intéressant d'en essayer d'autres pour voir du pays.

Le lancement de gnome se fait avec un programme nommé `gnome-session`. Celui de kde avec le programme `startkde`.

4.3.4 X11 en pièces détachées

C'est un choix délibéré de X11 de permettre de paramétrer à peu près tout. C'est un avantage, puisque chacun peut adapter sa configuration à ses choix particuliers. Cela complique aussi sérieusement le passage d'un ordinateur à un autre, puisque à peu près tout peut se comporter d'une façon différente, simplement du fait de changement dans les fichiers de configuration.

Dans cette section, nous allons lancer explicitement quelques programmes qui sont normalement démarrés automatiquement par le système ou par le display manager.

Lancement du serveur X11

Sous Mandriva, le serveur X11 lancé ordinairement tourne sur la première console virtuelle libre, qui porte le numéro 7.

Le matériel disponible et sa configuration est décrit pour X11 par un fichier de configuration qui sous Mandriva porte le nom `/etc/X11/xorg.conf`. Il s'agit d'un fichier texte, qu'on peut examiner et modifier (prudemment).

On peut avoir plusieurs serveurs X11 qui tournent simultanément sur l'ordinateur, utilisant chacun un terminal virtuel différent.

Pour lancer seulement un autre serveur X11, on peut rentrer la commande `X :1`. L'argument `:1` indique qu'on veut un second poste de travail. On a maintenant deux serveurs qui tournent sur deux consoles virtuelles différentes, entres lesquels on peut basculer avec `CTRL Alt F7` et `CTRL Alt F8`. Sur le second écran virtuel, il n'y a que le serveur X11 qui tourne, et il n'y a donc pas moyen d'y faire plus pour le moment que de déplacer la souris.

On peut tuer ce nouveau serveur X11 soit avec la combinaison de touche `CTRL Alt Backspace` sur la console où il tourne, soit avec un `CTRL C` dans la fenêtre depuis laquelle on l'a lancé.

Exercice 4.5 — Que se passe-t-il si on tente de lancer deux serveurs X11 avec même numéro (par exemple avec `X :1 & X :1`) ?

Exercice 4.6 — Peut-on lancer un serveur sur le display numéro 2 (avec `X :2`) sans en avoir lancé auparavant un sur le display numéro 1 ? Si oui, sur quel terminal virtuel apparait-il ? Et avec le display numéro 3 ?

Ouverture d'une fenêtre

Pour ouvrir une fenêtre, on peut revenir sur la console de départ (avec `CTRL Alt F7`) et lancer la commande `xterm -display :1`. L'argument indique à xterm qu'il doit contacter le serveur X11 que nous venons de lancer (le serveur ordinaire porte le numéro :0).

On peut maintenant revenir sur l'écran cet écran et taper des commandes dans la fenêtre. Les commandes ordinaires fonctionnent. On peut aussi demander au serveur X11 de modifier des paramètres, par exemple la couleur du fond de l'écran avec `xsetroot -solid red`.

Lancement d'un window manager

Depuis la fenêtre, on peut lancer un window manager. Essayer par exemple la commande `icewm` ; on peut l'arrêter simplement en tapant un `CTRL C` dans la fenêtre depuis laquelle on l'a lancé. Comparer le temps de lancement avec celui de gnome (qu'on lance avec `gnome-session`).

Les fenêtres de terminal

Il existe plusieurs programmes pour simuler un terminal. Nous utilisons en général `xterm`, mais on peut aussi se servir de `rxvt` qui est plus petit, de `kterminal` qui est développé pour KDE ou de `gnome-terminal` qui appartient à l'environnement de gnome.

Par défaut, ces terminaux fabriquent une fenêtre et lancent (avec fork et exec) un shell auquel on peut taper des commandes ; on peut en fait leur demander de lancer n'importe quel autre programme. Par exemple on peut lancer le calculateur `bc` avec la commande `xterm -e bc`. Une fois la commande lancée (avec fork et exec) le terminal attend (avec wait) qu'elle se termine puis il détruit la fenêtre.

4.3.5 Le chemin d'un caractère

Quand on tape un caractère sur le clavier, il suit un chemin tortueux : au départ, la presse de la touche est détectée par le noyau du système d'exploitation, qui le signale au serveur X11.

Le serveur X11 transmet le caractère au processus qui gère la fenêtre active avec un message du protocole X11 (qui s'appelle un évènement). Supposons que le programme qui le reçoit est l'émulateur de terminal `xterm`,

La commande `xterm` va provoquer l'écho du caractère en demandant au serveur X11 de l'afficher dans la fenêtre, à nouveau avec le protocole X11. (On

peut indiquer à xterm qu'il ne doit pas faire l'écho avec la commande `stty -echo` ; c'est de cette façon que le mot de passe n'apparaît pas en clair quand on tape la commande `su`). Il va aussi transmettre le caractère à l'application qu'il a lancée.

Chapitre 5

Systèmes de fichiers

Dans ce chapitre, nous présentons des détails sur l'implémentation des fichiers sur Linux. Nous commençons par préciser quelques aspects de l'utilisation des fichiers dans les commandes usuelles. Nous présentons ensuite les caractéristiques physiques importantes des disques durs, puis les systèmes de fichiers les plus communs, leurs montages et leurs démontages.

Une fois le chapitre assimilé, vous aurez des éléments pour choisir l'organisation de vos systèmes de fichiers sur les partitions, de réparer avec `fsck` les partitions défectueuses, d'accèder aux disques et clefs USB amovibles même quand ils ne sont pas reconnus par les outils présents dans votre distribution. Vous comprendrez presque toutes les caractéristiques d'un système de fichier Unix ordinaire et notamment la différence entre les liens en dur et les liens symboliques.

5.1 Retour sur des notions de base et des commandes usuelles

Dans cette section, nous précisons quelques notions que nous utilisons régulièrement dans les systèmes de fichiers : *l'arborescence des fichiers*, la différence entre *nom* de fichier et *chemin d'accès* aux fichiers.

5.1.1 Les fichiers et les répertoires, l'arborescence

Il y a plusieurs sortes de fichiers, comme nous le verrons par la suite. Pour le moment, nous nous contenterons de noter que certains fichiers ont un statut spécial : ce sont les *répertoires* qui contiennent d'autres fichiers, qui peuvent eux-mêmes êtres des répertoires. (Sur d'autre systèmes, on les appelle des *dossiers*.)

Ces répertoires forment un arbre (on parle de l'*arborescence* des fichiers), dont la racine porte le nom / (prononcer *root* ou *slash*).

Un répertoire s'appelle en anglais un *directory* ; ce nom apparaît sous forme abrégée dans de nombreuses commandes ; par exemple on fabrique un répertoire avec `mkdir` et on le détruit avec `rmdir`.

5.1.2 Nom de fichier et chemin d'accès

Il faut parfois faire la différence entre un nom de fichier et un nom de chemin d'accès au fichier (*filename* et *pathname*). En pratique, on omet souvent cette distinction et on emploie les deux mots indifféremment, mais dans ce chapitre nous allons prendre garde à ne pas les mélanger.

Nom de fichier

Le nom d'un fichier est une chaîne de caractères qui le désigne dans un répertoire ; il peut contenir des octets de n'importe quelle valeur, sauf l'oblique (le *slash* '/') qui sert de délimiteur entre les noms de fichiers et la valeur 0 qui sert de marqueur de fin de chaîne. Il y a une longueur maximum, qu'on peut trouver dans la documentation ou déterminer expérimentalement, par exemple avec le script shell suivant :

```
#! /bin/sh
#   Cree des fichiers avec des noms de + en + long, jusqu'a ce que ca rate
name="x"
while touch $name ; do
    rm $name
    name=${name}x
done

echo "echec avec le nom $name qui fait " `echo -n $name | wc -c` " caracteres"
```

Exercice 5.1 — (facile) Faire tourner le script shell et déterminer la longueur maximum d'un nom de fichier.

Exercice 5.2 — (plus difficile) Trouver dans la documentation l'endroit qui mentionne la longueur maximum d'un nom de fichier.

Chemin d'accès

Le chemin d'accès (*pathname*) désigne aussi un nom de fichier, mais contient une suite de répertoires qui permet d'y accéder, séparés par le caractère /, comme dans `/home/jm/tmp/foo`.

Le chemin d'accès peut-être *absolu*, s'il commence par le caractère '/' et cela indique que la liste des répertoires est à parcourir depuis la racine de l'arbre que forment les répertoires, ou bien il peut être *relatif* (s'il commence par n'importe quel caractère différent de /) pour indiquer que le chemin est à parcourir à partir du répertoire courant du processus.

Il existe deux noms de fichiers spéciaux : `.` et `..` (à prononcer comme *point* et *point-point*). Le nom `.` désigne le répertoire courant et le nom `..` le répertoire parent du répertoire courant.

Exercice 5.3 — (facile) Les noms de fichiers `/tmp/foo` `/tmp//foo`, `/tmp///foo` etc. sont-ils équivalents ? (c.a.d. : peut-on mettre plusieurs / pour séparer deux composants d'un chemin d'accès.)

Exercice 5.4 — (assez facile) Modifier le script qui sert à mesurer la taille maximum d'un nom de fichier pour trouver la longueur maximum d'un chemin d'accès. (Indication : puisque `.` désigne le répertoire courant, les chemins d'accès `foo`, `./foo`, `./././foo` etc. sont équivalents.)

Le fonction `iname`

Quand le noyau doit faire une référence à un fichier, il utilise une représentation interne du fichier que nous pouvons considérer (pour simplifier) comme composée d'un disque et d'un numéro d'index de ce fichier sur le disque.

Il y a dans le noyau une fonction, nommée `iname`, dont le rôle est de convertir un chemin d'accès au fichier. Voici une version simplifiée de l'algorithme réalisé par cette fonction :

```
si le nom commence par /
  rep ← répertoire racine
sinon
  rep ← répertoire de travail courant
Pour chaque composant du chemin d'accès
  x ← référence du composant dans rep
  rep ← x
x contient la référence vers le fichier désigné par le chemin initial
```

La fonction part de la référence vers le répertoire de travail ou du répertoire racine, suivant que le chemin d'accès est absolu ou relatif ; pour le premier nom qui apparaît dans le chemin d'accès (jusqu'à la fin du chemin, ou jusqu'au /), elle trouve la référence vers le fichier de le répertoire ; cela permet de trouver la référence dans les chemins courts, comme `"foo"` ou `"/usr"`. Si le chemin comporte plusieurs noms, comme `"/usr/local/bin/comix"` ou `"foo/bar/joe/jill`, la fonction utilise le fichier trouvé comme nouveau répertoire de départ et recommence le travail.

Il est à souligner que les entrées `.` et `..`, qui apparaissent dans tous les répertoires, n'ont pas besoin d'un traitement spécial de la fonction.

Nous reverrons cette fonction, en parlant des liens symboliques et du montage des systèmes de fichiers.

5.1.3 Commandes usuelles

Vous savez sans doute qu'on obtient la liste des fichiers d'un répertoire avec la commande `ls`, qu'on copie un fichier avec la commande `cp`, qu'on le détruit avec la commande `rm`, qu'on le change de répertoire avec la commande `mv`.

Commandes aliasées

La plupart des distributions Linux utilisent le mécanisme des *alias* pour invoquer des commandes avec des arguments qui en modifient le comportement. (On peut obtenir la liste des alias du shell courant avec la commande `alias`.)

Les commandes `rm`, `cp`, et `mv` sont invoquées avec l'option `-i` (comme *interactif*), qui leur fait demander une confirmation avant de détruire un fichier. Attention, vous risquez un jour de vous trouver devant un shell où ces alias ne sont pas installés (ou bien où ils ont été retirés) et ces commandes détruisent alors des fichiers sans poser de question. Comme je trouve cela dangereux, j'ai retiré ces alias avec la commande `unalias` dans mon fichier de configuration `.profile`, et les exemples qui suivent utilisent des versions sans alias.

La commande `ls` est appelée avec l'argument `-color`. Comme je trouve la multiplication des couleurs qui en résulte difficile à lire, j'ai retiré l'alias de la même manière et les exemples qui suivent ne l'utilisent pas.

Si vous ne souhaitez pas retirer l'alias de ces commandes de façon définitive, vous pouvez obtenir le même effet de manière temporaire en plaçant une contre-oblique devant le nom de la commande que vous appelez.

Des options aux commande usuelles

La commande `ls` a de nombreuses options, dont nous n'utiliserons qu'un petit nombre.

L'option `-d` sert à lister les caractéristiques d'un répertoire au lieu de lister celles des fichiers qu'il contient. Par exemple,

```
$ mkdir foo                  fabriquer un répertoire
$ touch foo/a foo/b foo/c    y placer quelques fichiers
$ ls foo                     regarder le contenu du répertoire
a b c
$ ls -d foo                  regarder le répertoire lui-même
foo
```

L'option `-l` sert à obtenir plus d'informations sur les caractéristiques des fichiers listés.

```
$ ls -l foo
total 0
-rw-r-r- 1 jm jm 0 2008-03-07 19 :10 a
-rw-r-r- 1 jm jm 0 2008-03-07 19 :10 b
-rw-r-r- 1 jm jm 0 2008-03-07 19 :10 c
$ ls -ld foo
drwxr-xr-x 2 jm jm 4096 2008-03-07 19 :10 foo
```

En première approximation, les sept premiers caractères donnent le type et les permissions d'accès au fichier. Le nombre qui suit compte les noms du fichier (noter que le répertoire `foo` possède deux noms : `foo` et `.`). Suivent le nom et le groupe du propriétaire du fichier, sa taille en octets (ici 0 pour les fichiers ordinaires et 4K octets pour le répertoire), la date de sa dernière modification et finalement le nom du fichier.

Par défaut, la commande `ls` ne liste pas les fichiers dont le nom commence par un point. Avec l'option `-a` (comme *all*), elle affiche ces fichiers aussi.

```
$ touch foo/.cache    fabrique un fichier dont le
                      nom commence par '.'
$ ls foo              liste le contenu de foo :
a b c                 on ne voit pas le fichier
$ ls -a foo           idem avec l'option -a
. .. a b c .cache     on le voit, ainsi que '.' et '..'
```

Exercice 5.5 — (facile) Qu'affiche la commande `ls -d`? Pourquoi?

La commande `rmdir` sert spécialement à détruire un répertoire, mais elle ne traite que des répertoires vides. La façon ordinaire de détruire un répertoire qui n'est pas vide est d'utiliser la commande `rm` avec l'option `-r` (comme *récursif*).

```
$ rm foo                                          supprimer foo ?
rm : cannot remove ‘foo’ : Is a directory         échec parce que
                                                  c'est un répertoire
$ rmdir foo                                       supprimer le répertoire
rmdir : foo : Directory not empty                 échec parce que
                                                  il n'est pas vide
$ rm -r foo                                       supprimer le répertoire
                                                  et son contenu
$                                                 ok
```

(L'existence de deux commandes distinctes `rmdir` et `rm` pour détruire les répertoires et les autres fichiers est une trace de la manière dont fonctionnaient `mkdir` et `rmdir` dans les versions antérieures d'Unix : `mkdir` était une commande qui, après avoir fabriqué un répertoire vide y ajoutait les noms `.` et `..` vers le répertoire parent ; `rmdir` détruisait ces noms avant de détruire le répertoire. Cela pouvait être source de confusion dans le système de fichier : en cas d'arrêt de la machine entre deux opérations de la commande, on avait des répertoires qui ne contenaient pas les entrées `.` et `..` ; par la suite, `mkdir` et `rmdir` sont devenus des appels système, ce qui supprime en pratique ce risque d'incohérence dans le système de fichier.)

5.2 Le disque

Parce que les disques sont des périphériques plutôt lents quand on les compare avec le processeur et la mémoire, leurs caractéristiques physiques présentent une certaine importance ; nous les présentons dans cette section, avec quelques uns des mécanismes qui permettent de diviser et de réunir les disques.

5.2.1 Physique

Physiquement, les disques durs sont constitués de surfaces magnétiques sur lesquelles se déplace un bras qui porte une tête qui peut magnétiser la surface pour écrire un bit d'information ou examiner la façon dont la surface est magnétisée pour relire cette information.

Cylindres, Têtes, Secteurs

Du point de vue mécanique, un disque se compose de *surfaces magnétiques* solidaires sur le même axe, que fait tourner un moteur ; on appelle couramment ces surfaces des *platines*. Les vitesses de rotation tournent autour de 5000 tours par minutes (noté rpm : *rotation per minute*). Un seul *bras* permet de déplacer une tête de lecture au-dessus de chaque platine. Le nombre de têtes de lectures est donc égal au nombre de platines.

Pour une position donnée du bras, c'est toujours la même partie du disque qui passe sous une tête de lecture ; on appelle ceci une *piste*, et l'ensemble des *pistes* de toutes les platines pour une même position du bras s'appelle un *cylindre*.

Sur les disques, on ne peut pas lire ou écrire bit par bit, ou octet par octet comme dans la mémoire : la plus petite unité qu'il est possible de lire ou d'écrire s'appelle un *secteur* ; la taille d'un secteur est le plus souvent de 512 octets.

La taille d'un disque peut donc se mesurer avec ces trois paramètres : le nombre de cylindres (qui correspond au nombre de positions différentes du bras de lecture), le nombre de têtes (qui correspond au nombre de platines), et le nombre de secteurs par piste. En multipliant ces trois nombres en eux, puis par le nombre d'octets par secteur, on obtient la capacité du disque. On trouve ces trois valeurs sur la description des disques sous le nom *C/H/S* (comme *Cylinder*, *Head*, *Sector*).

Attention, cette vision de l'organisation des disques est toujours utilisée pour décrire les caractéristiques des disques dans certains programmes de bas niveau (notamment les programmes pour découper un disque dur en partitions), mais elle ne correspond plus vraiment à la réalité. Par exemple, tous les disques modernes profitent de ce que les cylindres vers l'extérieur du disque sont plus longs que ceux de l'intérieur pour y placer plus de secteurs ; le circuit de contrôle du disque se charge de faire les opérations de conversion nécessaires. On peut aussi souvent accéder au disque en mode LBA (comme *Linear Block Address* : adressage de bloc linéaire), qui ignore la distinction entre cylindres, têtes et secteurs : le disque se présente aux logiciels comme une suite de secteurs numérotés sans référence aux caractéristiques physiques du disque.

Le formatage de bas niveau des disques, bad blocks

Un disque doit être formaté pour être utilisé : cela consiste à inscrire sur la surface magnétique, une fois pour toute, des en-têtes qui permettent de délimiter les secteurs et les pistes. Depuis une dizaine d'années, cette opération est faite en usine lors de la fabrication du disque avec des instruments particuliers qui ne sont pas accessibles à un utilisateur ordinaire.

Une étape supplémentaire consiste à identifier sur le disque les secteurs défectueux ; on les appelle des *bad blokcs* ; ces secteurs sont réaffectés par le contrôleur du disque à des secteurs de rechange (*spare blocks*) qui ont été réservés à cet effet. Cette opération peut prendre plusieurs heures pour un disque dur ordinaire.

Depuis que Microsoft a réemployé le verbe *formater* pour décrire la construction d'un système de fichier vide, on appelle souvent ces opérations le *formatage de bas niveau*.

Temps d'accès au disque

Comparé à la vitesse de calcul du processeur ou à celle de l'accès à la mémoire qui sont de l'ordre de la nanoseconde, les disques sont des périphériques lents. Les principales sources de lenteur viennent des déplacements des platines et du bras.

Quand le bras est positionné sur un cylindre, il faut attendre pour lire ou écrire un secteur que celui-ci passe sous la tête de lecture ; en moyenne, cela demande la moitié du temps d'une rotation complète de la platine (voir exercice).

Quand le bras n'est pas dans la bonne position, il faut alors le déplacer pour l'amener sur la bonne piste. Le temps nécessaire dépend pour partie de la distance à parcourir : il est plus rapide de déplacer le bras jusqu'au cylindre voisin que d'aller d'un cylindre situé vers l'intérieur de la platine à un cylindre situé à l'extérieur (ou le contraire).

Ces délais sont partiellement masqués par le contrôleur du disque qui, sur les disques modernes, utilise une mémoire locale pour gérer une file de secteurs

à lire et écrire. Cela ne masque cependant pas l'ordre de grandeur du temps nécessaire

La mesure la plus fiable est donnée par le *mean seek time* (mst), qui est le temps moyen pour accéder à un secteur donné sur le disque. Sur les disques actuels il est de l'ordre de la dizaine de milliseconde.

Il existe une troisième source de délais pour accéder aux disques : le temps nécessaire pour transférer les données entre le disque et la mémoire. Celui-ci dépend du bus utilisé pour les connecter et de la façon dont ils y accèdent. Les bus les plus courants pour les disques internes de PC sont le bus ATA, en version parallèle (nommée PATA ou EIDE) ou en version série plus moderne, (nommée SATA). Pour les disques externes on rencontre surtout le bus USB. Les vitesses de transmission annoncées dans les spécifications d'un disque ne tiennent compte que des délais dus au bus et pas à ceux dus à la mécanique, évoqués au dessus : il s'agit de débit *de crête* : en pratique, celui que le constructeur garantit que personne n'atteindra.

Exercice 5.6 — Étant donné un disque tournant à 5000 rpm et un processeur à deux gigahertz, en supposant que le microprocesseur exécute une instruction à chaque cycle d'horloge, combien d'instructions le processeur exécute-t-il pendant une demie rotation des platines ? (Une demie rotation correspond au temps moyen pour qu'un secteur donné passe sous la tête de lecture.)

Écritures par le système : ordonnancement des entrées sorties

Toutes les lectures et les écritures sur le disque sont effectuées par le système d'exploitation (sauf quelques rares exceptions) ; pour diminuer le temps d'accès à un secteur donné du disque, celui-ci les réordonne pour minimiser les déplacements du bras et tente d'anticiper sur les lectures.

Quand il faut procéder à une lecture ou à une écriture, le noyau place la demande dans une file d'attente ; l'ordre de la file d'attente dépend d'un algorithme, appelé *ordonnancement*. Nous présentons ici trois ordonnancements standards, avec leurs avantages et leurs inconvénients, puis nous évoquons brièvement l'ordonnancement utilisé dans le noyau Linux.

ordonnancement FIFO Une liste d'attente *FIFO* (comme *« First In, First Out »*, c'est dire *Premier entré, premier sorti*) correspond aux files d'attentes ordinaires auxquelles nous participons : les demandes sont traitées dans l'ordre où elles arrivent. Pour les disques, cela présente l'avantage de la simplicité et garantit que toutes les demandes seront traitées au bout d'un temps qui est proportionnel à la charge du système. En revanche cela ne permet pas de minimiser les mouvements du bras : les performances ne sont pas excellentes.

ordonnancement *Short Seek First* Le principe est de traiter en premier la requête qui nécessite le plus petit déplacement du bras : cela conduit à minimiser ses déplacements et on obtient donc des performances globales qui sont les meilleures possibles. Il y a cependant un inconvénient majeur : les demandes qui nécessitent un long déplacement du bras sont défavorisées et il peut se faire qu'elles en soient jamais exécutées ; pour un exemple élémentaire, imaginons qu'un processus $P1$ renouvelle en permanence des demandes de lecture sur les pistes 0 et 1, alors qu'un autre processus $P2$ a demandé une lecture sur la piste 10 : quand la lecture sur la piste 1 est terminée, c'est la demande de $P1$ sur la

piste 0 qui est choisie puisque c'est la plus proche ; quand c'est la demande sur la piste 1 qui est terminée, c'est la demande de $P1$ sur la piste 0 qui est choisie ; la demande de $P2$ ne sera jamais satisfaite. On appelle ce type de situation ou une demande n'est jamais satisfaite une *famine*.

ascenseur Un compromis qui permette de minimiser les attentes globales tout en évitant les famines consiste à utiliser l'algorithme utilisé dans les ascenseurs pour choisir leur déplacement : le bras est déplacé de la première à la dernière piste en satisfaisant, dans l'ordre, toutes les demandes présentes dans la file d'attente ; une nouvelle demande pour une piste inférieure est simplement placée en attente. Une fois que la dernière piste sur laquelle il y a une requête est atteinte, le sens est inversé et le bras va être ramené de la dernière piste à la première, en satisfaisant de nouveau toutes les requêtes.

Linux L'ordonnancement des requêtes sous Linux se fait avec un algorithme appelé CFQ, comme *Completely Fair Queuing* (*Files d'attentes parfaitement équitables*) qui fait la distinction entre les entrées sorties *synchrones* (qui bloquent les processus) et les *asynchrones* (qui bloqueront probablement le processus plus tard). Celles-ci sont organisées en prenant en compte, comme toujours, la minimisation des mouvements du bras mais aussi la priorité du processus.

5.2.2 Partitions

Les disques durs sur les PCs sont souvent divisés en parties indépendantes qu'on appelle des *partitions*. Le premier secteur du disque, le *Master Boot Record* (ou MBR), contient du code de démarrage de l'ordinateur et une table qui indique comment le système doit considérer que le reste du disque est découpé en au plus quatre partitions indépendantes. Ce sont les *partitions principales*.

L'une d'entre elles peut être marquée comme *partition active*, et c'est sur celle-là que les logiciels Microsoft s'attendent à trouver le système d'exploitation. Les bootloaders *Grub* et *Lilo* permettent eux de choisir la partition qui contient le système.

Chacune de ces partitions se comporte comme un disque indépendant, et comment donc aussi par un *boot record*, ici un *Volume Boot Record* (VBR) pour les différencier du MBR. Ce VBR contient à son tour une table des partitions qui permet de diviser la partition en sous-partitions, des *partitions étendues*. Il n'y a pas de limite au nombre de partitions étendues.

Les partitions sont gérées par le noyau du système d'exploitation, qui a lu la table des partitions au moment du démarrage. Sous Linux, les disques ordinaires portent le nom `sd`X ou X est un caractère pour différencier les disques, et on peut voir les messages qui les concernent dans la sortie de la commande `dmesg` :

```
$ dmesg | grep sd[a-z] :
sda : sda1 sda2 < sda5 sda6 sda7 > sda3
```

La ligne nous indique que l'ordinateur contient un disque dur, divisé entre trois partitions principales (nommées de `sda1` à `sda3`. La partition `sda2` est elle même divisée en trois partitions étendues, numérotées de `sda5` à `sda7`.

Le disque et les partitions se retrouvent, comme des fichiers spéciaux, dans le répertoire `/dev` (comme device) :

```
$ ls /dev/sd*
/dev/sda       /dev/sda2  /dev/sda5  /dev/sda7  /dev/sda9
/dev/sda1      /dev/sda3  /dev/sda6  /dev/sda8
```

Le disque dans son entier apparaît sous le nom `/dev/sda` et les partitions avec un nom qui correspond à leur numéro : entre 1 et 4 pour les partitions principales, à partir de 5 pour les partitions étendues. Nous reviendrons sur les fichiers spéciaux et sur le répertoire `/dev` dans la suite du chapitre.

5.2.3 RAID

A l'inverse des partitions qui servent à diviser un disque physique en disques logiques indépendant, il est possible de prendre un ensemble de disques physiques et de les faire apparaître au système comme un seul disque logique, en rajoutant un contrôleur ou un pilote spécifique. Ce modèle est appelé RAID, comme dans *Redundant Array of Inexpensive Disks* (tableau redondant de disques bon marché).

Il existe plusieurs niveaux de RAID suivant la manière dont les données sont réparties entre les disques. Le plus courant est le RAID 5, dans lequel un disque de parité supplémentaire permet de continuer à fonctionner même quand un des disques tombe en panne. On remplace alors le disque défaillant, et son contenu est reconstruit à partir du contenu des disques restant.

Mon expérience personnelle me conduit à penser que les disques RAID sont une solution intéressante seulement pour les gros centres de calculs avec de gros moyens : le remplacement d'un disque dans un système RAID est une opération délicate et pour espérer qu'elle fonctionnera sans accroc quand une panne se produira, donc lors d'une situation de crise, il faut procéder régulièrement à des répétitions.

Pour un utilisateur personnel ou un petit centre de calcul, les solutions à base de disques amovibles sont beaucoup plus pratiques et à peine moins sures. Elles permettent aussi de suivre plus facilement l'augmentation régulière de la capacité des disques durs pour un investissement moindre.

5.3 La hiérarchie usuelle des fichiers à grand traits

Cette section présente les répertoires les plus importants sur un système Unix ou Linux.

5.3.1 La racine

La racine des systèmes de fichiers se trouve sur une partition que le noyau a déterminée au moment du boot. Il est difficile d'en changer par la suite. Cette partition contient les fichiers essentiels au démarrage, dans des répertoires séparés.

`/boot`

Le répertoire `/boot` contient les fichiers essentiels pour les premières étapes du démarrage de Linux : le fichier qui contient l'image du noyau dans un fichier dont le nom commence sans doute par `vmlinuz`, une pseudo-partition utilisée au

boot dont le nom commence sans doute par `initrd`, les fichiers de configuration de `grub`.

`/bin` et `/sbin`

Les répertoires `/bin` et `/sbin` contiennent les commandes, sous forme de programmes exécutables. Dans `/bin` on trouve les commandes ordinaires ; dans `/sbin` les commandes d'administration qui ne sont normalement exécutées que par root.

`/etc`

Le répertoire `/etc` contient les fichiers de données de l'administration du système.

`/lib`

Dans `/lib`, on trouve les bibliothèques (statiques) utilisées pour la compilation des programmes dont les noms se terminent avec `.a` et les bibliothèques (dynamiques) qui contiennent du code ajouté aux programmes en cours d'exécution ; leurs noms se terminent avec `.so` suivi d'un numéro de version.

`/tmp`

Le répertoire `/tmp` est destiné à contenir les fichiers temporaires.

5.3.2 Les répertoires spéciaux `/dev` et `/proc`

Ces deux répertoires jouent un rôle très spécial. Ils n'existent pas vraiment sur le disque ; le noyau du système les a contruits quand il a démarré.

Dans `/dev`, tous les périphériques apparaissent comme des fichiers ; l'examen de son contenu est une façon simple de savoir ce que le noyau a détecté : par exemple, l'existence d'un fichier `/dev/video0` montre qu'il a trouvé une caméra.

Dans `/proc`, on trouve un répertoire par processus (avec son PID en guise de nom) et de nombreux fichiers qui permettent de consulter et parfois de modifier les caractéristiques du noyau. Par exemple `/proc/meminfo` contient les caractéristiques de la mémoire.

5.3.3 `/usr`

Il y a très très longtemps, le répertoire `/usr` (prononcer *slachyuseur*) contenait les fichiers des utilisateurs, d'où son nom. Quand la partition racine a débordé, il a été utilisé comme partition de secours pour la racine et maintenant on n'y trouve plus de fichiers d'utilisateurs. En revanche, il y a là une copie des répertoires de la racine avec les choses qui ne sont pas indispensable au moment du boot : un `/usr/bin` et un `/usr/sbin` avec des programmes exécutables, un `/usr/lib` avec des bibliothèques, etc. En principe, ce morceau de l'arborescence peut être partagé par plusieurs ordinateurs qui ont la même architecture.

On y trouve aussi un répertoire `share` en dessous duquel sont placés les fichiers de données des commandes qui peuvent être partagés par des machines d'architectures différentes. C'est là qu'on place la documentation, dans `/usr/share/man` pour le manuel et dans `/usr/share/doc` pour le reste.

Il y a également dans `/usr/include` les fichiers à inclure dans les programmes C et dans `/usr/src` les sources des commandes et du noyau, si on les a installés.

Autrefois, `/usr` était une toujours une partition séparée ; à l'heure actuelle, il y a de nombreuses distributions de Linux qui le placent dans la même partition que la racine.

5.3.4 `/usr/local`

Sous `/usr/local`, on trouve encore une partition avec des répertoires `bin`, `sbin`, `etc`, `share`, `src` ; c'est là que se placent les commandes non standard rajoutées par dessus l'installation du système.

5.3.5 `/home`

Le répertoire `/home` est souvent dans une partition à lui tout seul. Il contient les fichiers des utilisateurs. Chaque utilisateur y dispose d'un répertoire qui porte son nom et qui lui appartient, dans lequel il est libre d'organiser ses fichiers de la manière qui lui convient.

5.3.6 `/var`

Le répertoire `/var` contient les fichiers *variables*. Le plus important est le répertoire `/var/log` qui contient les fichiers dans lesquels le système et les démons placent des messages d'information sur leur activité. Pour empêcher ces fichiers de dévorer toute la place disponible sur le disque, `/var` est souvent placé dans une partition séparée.

5.4 Montage et démontage des systèmes de fichiers

Les systèmes de fichiers qui se trouvent dans des partitions différentes sont indépendants les uns des autres. Ils ont chacun leur répertoire racine et une arborescence par dessous. Pour l'utilisateur, ces différents arbres sont unifiés dans une seule arborescence, grâce aux commandes `mount` et `umount`.

5.4.1 Le système de fichier racine

Au démarrage du système, le noyau identifie une des partitions comme celle qui contient le système de fichier racine. (On peut spécifier interactivement cette partition au bootloader ; la partition se trouve par défaut dans son fichier de configuration (`/etc/lilo.conf` pour `lilo` et `/boot/grub/menu.lst` pour `grub`.) Une fois que le système de fichier racine est choisi, il est difficile d'en changer sans rebooter l'ordinateur.

FIG. 5.1 – La racine du système de fichier de la partition 2 est accrochée avec la commande `mount` au répertoire `/home` de la partition 1 qui est la partition racine. Le fichier `/jm/y` de la partition 2 est maintenant accessible sous le nom `/home/jm/y`. Le fichier `/home/x` de la partition 1 n'est plus accessible tant que la partition 2 n'est pas démontée

(Certaines distributions de Linux utilisent un système de fichier temporaire comme une étape supplémentaire du boot, qui porte le nom de `initrd`. Ce système de fichier est contenu dans un fichier ordinaire, qu'on trouve sous Mandriva dans `/boot/initrd`. Après les premières initialisations, ce système de fichier racine temporaire est remplacé par le définitif.)

5.4.2 Mount et umount

Pour accrocher une arborescence dans le système de fichier, on utilise la commande `mount` qui exécute l'appel système du même nom. Elle prend un périphérique qui contient un système de fichier et accroche sa racine au répertoire qu'on lui donne comme second argument. Ce qui en résulte est suggéré par la figure 5.1 : la racine de la partition montée (la partition 2 dans l'exemple) apparaît maintenant avec le nom du répertoire sous lequel elle est montée.

On peut retirer une partition avec la commande symétrique, `umount` qui retire le lien installé avec le `mount` : les fichiers de la partition deviennent inaccessibles. La commande échoue s'il y a un fichier utilisé sur le système de fichier.

Les appels système `mount` et `umount` ont pour effet de modifier une table des systèmes de fichiers montés qui est présente dans le noyau du système. Elle contient des couples qui associent un répertoire avec un système de fichier. `Mount` ajoute une entrée dans la table alors que `umount` en retire une.

Cette table est utilisée par la fonction (`iname`) que nous avons présentée au début du chapitre, qui sert à traduire un chemin d'accès en référence vers un fichier. La fonction modifiée est la suivante :

si le nom commence par /
  rep ← répertoire racine
sinon
  rep ← répertoire de travail courant
Pour chaque composant du chemin d'accès
  x ← référence du composant dans *rep*
  si x apparaît dans la table des systèmes de fichiers montés
    x ← racine du système de fichier associé
  rep ← x
x contient la référence vers le fichier désigné par le chemin initial

La commande `mount` sans argument imprime la liste des systèmes de fichiers montés :

```
$ mount
/dev/sda5 on / type ext3 (rw,noatime)
none on /proc type proc (rw)
/dev/sda7 on /home type ext3 (rw,noatime)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
/dev/sda1 on /mnt/windows type ntfs (rw,umask=0022,nls=utf8)
```

En général, s'il existe une partition pour Windows, celle-ci est inutilisée par les processus Linux et on peut la manipuler librement, comme dans l'exemple suivant

```
# mount | grep windows        information sur la partition windows
/dev/sda1 on /mnt/windows type ntfs (rw,umask=0022,nls=utf8)
# ls /mnt/windows             regarder le contenu de la partition
audio.log Chkmail.log pagefile.sys ricoh.log
...                           quelques lignes omises
# umount /mnt/windows         démonter la partition
# ls /mnt/windows             le répertoire est maintenant vide
# mkdir /tmp/toto             fabriquer un autre répertoire
# mount /dev/sda1 /tmp/toto   y remonter la partition
# ls /tmp/toto                regarder son contenu
audio.log Chkmail.log pagefile.sys ricoh.log
...                           quelques lignes omises
```

5.4.3 Le fichier /etc/fstab

Le fichier de configuration `/etc/fstab` contient la liste des partitions à monter. Il s'agit d'un fichier de texte ; on peut le regarder et l'éditer avec les outils usuels.

Chaque ligne du fichier contient la description d'un système de fichier : le nom du fichier (spécial) qui désigne le périphérique, le nom du répertoire sur lequel le monter, le type de système de fichier, des options qui modifient la façon dont les fichiers sont gérés et deux nombres qui indiquent dans quel ordre la commande `fsck` doit vérifier l'état des partitions.

Parmi les nombreuses options de gestion des fichiers, on trouve `users` pour indiquer que la commande `mount` doit pouvoir être utilisée par un utilisateur ordinaire, pas seulement par l'administrateur du système ; `noauto` qui indique que le système de fichier ne doit pas être monté automatiquement ; `ro` (comme

read only) pour indiquer que le système de fichier ne doit être modifié en aucune façon.

Pour les périphériques amovibles, comme les lecteurs de DVD ou de CDROM et les clefs USB, il est nécessaire de monter le système de fichier qu'elles contiennent quand le périphérique est branché (ou qu'un CDROM est introduit). Certaines distributions de Linux (mais pas Mandriva par défaut) installent un démon *automounter* : un processus qui tourne en permanence et qui examine si un périphérique est introduit ; quand c'est le cas, il lance une commande `mount` pour le rendre accessible dans l'espace des fichiers.

5.5 La commande pwd

La commande `pwd` (comme *Print Working Directory*) imprime le répertoire courant. Sous le shell ordinaire (`bash`), il s'agit d'une commande interne : le shell affiche lui-même le nom du dernier répertoire où on est allé avec `cd`. Pour obtenir une commande, on peut la lancer avec son chemin d'accès complet `/bin/pwd`.

On a vu un exemple de différence de comportement entre la commande interne du shell `pwd` et la commande `/bin/pwd` :

`$ cd`	*aller dans le répertoire de login*
`$ mkdir foo`	*fabriquer un répertoire foo*
`$ cd foo`	*aller dedans*
`$ ln -s . bar`	*y ajouter un lien symbolique vers .*
`$ cd bar/bar/bar`	*suivre ce lien symbolique plusieurs fois*
`$ pwd`	*commande interne ?*
`/home/jm/foo/bar/bar/bar`	*là où on a fait le cd*
`$ /bin/pwd`	*commande externe ?*
`/home/jm/foo`	*le vrai nom du répertoire*

Pour trouver le nom répertoire courant, la commande `pwd` trouve le numéro d'inode du répertoire courant en ouvrant le répertoire '.' et en y trouvant la description de '.', puis dans `..` elle trouve l'entrée qui porte le même numéro d'inode : elle a obtenu le dernier nom qui compose le chemin d'accès ; il faut recommencer l'opération à partir du nouveau répertoire courant. Une fois arrivé à la racine du système de fichier de la partition, le numéro d'inode de '.' et de '..' désignent le même numéro d'inode. La commande `pwd` va alors interroger la table des systèmes de fichiers montés pour connaître le nom du répertoire sur lequel il est monté. (La table est celle imprimée par la commande `mount` quand on la lance sans argument ; on la trouve dans le fichier `/etc/mtab`.)

5.6 Un des systèmes de fichier de Linux : ext2fs

On peut voir l'organisation des fichiers sur un disque comme une unique structure de donnée plutôt complexe, stockée dans l'espace mémoire (permanent) que représente le disque.

Le système de fichier utilisé par défaut dans la plupart des distributions de Linux actuelles est le système ext2fs, largement inspiré du système de fichier d'Unix BSD, ou son successeur nommé ext3fs. Le système utilise principalement un *superblock* pour décrire le disque, un *inode* pour décrire chaque fichier et des blocs de données pour contenir les données de chaque fichier.

Les structures de données utilisées par ext2fs peuvent se trouver dans le fichier `/usr/include/linux/ext2_fs.h`.

5.6.1 Organisation générale

Le contenu du fichier est placé dans des *blocs* d'une taille fixe, généralement de 4 Ko ; c'est la plus petite unité que le système de fichier lira et écrira pour un fichier. Pour éviter de perdre de la place, le dernier bloc d'un fichier peut être stocké dans un *fragment* : un bloc qui est divisé en parties plus petites partagées entre plusieurs fichiers.

La liste des blocs d'un fichier qui contiennent les données et les autres informations sur ce fichier (en dehors de son nom et de son chemin d'accès) sont stockés dans des structures qu'on appelle des *inodes*.

La liste des blocs libres et celles des inodes libres sont stockées dans des blocs, sous forme de bitmaps.

Pour éviter de longs déplacements du bras de lecture, le disque est divisé en *groupes de cylindres* : chaque groupe de cylindre contient des inodes et des blocs de données, et la stratégie d'allocation du disque s'efforce de stocker l'inode et les blocs de données d'un même fichier dans le même groupe de cylindres.

Enfin, on trouve au début du disque un *superbloc*, qui décrit les paramètres généraux du disque et la position des blocs de cylindres. Pour plus de sûreté, le superbloc est recopié au début de chaque groupe de cylindres. L'organisation du superbloc est définie par la struct `ext2_super_block` dans `/usr/include/ext2_fs.h`.

5.6.2 Les répertoires

Les répertoires sont des fichiers presque comme les autres, si ce n'est que les processus ne peuvent pas y écrire : le contenu est modifié par le système lorsqu'on crée ou qu'on détruit un fichier. Une version simplifiée de la structure est la suivante :

```
struct ext2_dir_entry {
    char inode[4];       /* Inode number */
    char rec_len[2];     /* Directory entry length */
    char name_len[1];    /* Name length */
    char file_type[1];   /* File Type */
    char name[255];      /* File name */
};+
```

Le champs `inode` est un entier sur quatre octets qui contient le numéro dans le disque de la structure inode qui décrit le fichier. Nous présentons le contenu de cette structure dans la section suivante. Le champs `name` contient le nom sous lequel le fichier apparait : il est déclaré ici comme un tableau de 255 octets, mais si le nom est plus court, seule la partie utilisée sera allouée à l'entrée dans le répertoire. Le champs `rec_len` permet de trouver le départ de l'enregistrement suivant, malgré la longueur variable du champs `name`. Le champs `name_len` permet de réutiliser une entrée qui a été libérée par la destruction d'un fichier pour un fichier dont le nom est plus court.

Les plus petites entrées dans un répertoire occuperont donc 9 octets : 8 octets pour les champs de longueur fixe, plus 1 octet pour le nom de fichier. Les plus

grandes entrées, pour un fichier dont le nom comporte 255 caractères. occupera 8+255 octets.

Quand un répertoire vide est créé, il a besoin des deux entrées qui décrivent `.` et `..` qu'y place l'appel système `mkdir`. Celles-ci occupent respectivement 9 et 10 octets.

La recherche d'un nom dans un répertoire s'est longtemps faite en parcourant le contenu du répertoire de manière séquentielle. Ceci est satisfaisant tant que le répertoire ne contient pas trop de fichiers (typiquement tant que les données tiennent dans un bloc de 4 Ko), mais quand le répertoire grossit, le temps nécessaire pour trouver un fichier grossit d'une façon insupportable. Pour cette raison les répertoires peuvent être organisés pour accélérer les recherches, sous forme de B-tree comme dans les bases de données.

5.6.3 La structure inode

Un inode décrit toutes les caractéristiques d'un fichier (sauf ses données ; on dit parfois que l'inode contient des *méta-données*). Dans ext2fs, un inode occupe 128 octets.

A chaque fichier correspond un inode sur le disque. Une fois qu'un fichier est représenté par un inode, il le conservera jusqu'à ce que le fichier soit détruit. On peut connaître le numéro d'un inode avec l'option `-i` de la commande `ls`, comme l'exemple suivant :

```
$ mkdir toto                    Fabriquer un répertoire,
$ touch toto/a toto/b toto/c    y mettre quelques fichiers
$ ls -id toto                   Numéro d'inode du répertoire ?
2080831 toto                    Le voila
$ ls -ia toto                   numéros d'inodes des fichiers dans toto
                                Avec l'option -a pour avoir aussi . et ..
2080831 . 1278427 .. 2080837 a 2080840 b 2080841 c
$ rm -fr toto
```

Exercice 5.7 — (facile) Où la commande `ls -il` affiche-t-elle le numéro d'inode ?

Exercice 5.8 — (facile) à quel inode correspond les noms '.' et '..' dans la racine du système de fichier.

L'organisation précise d'un inode est décrite par la struct `ext2_inode` du fichier `/usr/include/linux/ext2_fs.h`. Comme souvent dans les noyaux Unix, elle définit d'une façon compliquée quelque chose de relativement simple. La table 5.1 en contient une version simplifiée.

Nous allons examiner presque tous les champs de cette structure ; cela nous servira de prétexte pour creuser certains des aspects particuliers des systèmes de fichier d'Unix.

Les dates de création, modification, accès et destruction

Les champs `ctime`, `mtime`, `atime` et `dtime` contiennent la date et l'heure des opérations sur un fichier. Cette date est stockée, comme la plupart des dates sous Unix, sous la forme d'un entier qui contient le nombre de secondes qui se sont écoulées depuis le premier janvier 1970.

```
struct ext2_inode {
    char mode[2];           /* File mode */
    char uid[2];            /* Low 16 bits of Owner Uid */
    char size[4];           /* Size in bytes */
    char atime[4];          /* Access time */
    char ctime[4];          /* Creation time */
    char mtime[4];          /* Modification time */
    char dtime[4];          /* Deletion Time */
    char gid[2];            /* Low 16 bits of Group Id */
    char links_count[2];    /* Links count */
    char blocks[4];         /* Blocks count */
    char flags[4];          /* File flags */
    char osd1[4];           /* OS dependent 1 */
    char block[4][15];      /* Pointers to blocks */
    char generation[4];     /* File version (for NFS) */
    char file_acl[4];       /* File ACL */
    char dir_acl[4];        /* Directory ACL */
    char faddr[4];          /* Fragment address */
    char osd2[12];          /* OS dependent 2 : fragment */
};
```

TAB. 5.1 – Une version simplifiée de la structure d'un inode. Les types variés ont été remplacés par des tableaux de char qui indiquent le nombre d'octets utilisés. Les commentaires sont d'origine.

`Ctime` contient la date de création, `mtime` la date de dernière modification, `Atime` la date de dernier accès et `dtime` la date de destruction.

Le champs `atime` est problématique, parce que chaque accès à un fichier impose une écriture pour mettre ce champs à jour. Sur les disques ordinaires, cela ralentit seulement les opérations, mais c'est particulièrement gênant sur les disques à base de mémoire flash, comme on en trouve dans de nombreuses clefs USB, car celle-ci ne permet que quelques dizaines de milliers d'écritures avant de tomber en panne. On peut supprimer la mise à jour de ce champs sur un disque.

La commande `ls` affiche normalement la date de dernière modification des fichiers quand on l'appelle avec l'option `-l`. Avec les options `-lt`, elle trie également les fichiers par date de modification ; avec `-ltu` elle trie par date d'accès ; ainsi `ls -lt | sed 3q` affiche les trois fichiers les plus récemment modifiés ; c'est une façon simple de vérifier si on a travaillé récemment dans un répertoire.

Exercice 5.9 — (facile) Jusqu'à quand sera-t-il possible de stocker des dates sur quatre octets avant de retomber sur le 1er janvier 1970 ?

Exercice 5.10 — (facile) Quelles sont les options de ls pour trier par date de création, avec les dates les plus anciennes en premier ?

L'identité du propriétaire du fichier

Les champs `uid` et `gid` contiennent l'identité du propriétaire du fichier, sous la forme de deux entiers de deux octets qui contiennent son numéro d'utilisateur et de son numéro de groupe.

(Pour permettre à un ordinateur d'avoir plus de 32768 utilisateurs, le noyau Linux a fait passer les numéros d'utilisateur et de groupe à quatre octets ; la partie haute de ces numéros est alors stockée dans le champs `osd2`.)

Le propriétaire du fichier est celui du processus qui l'a créé ; il est possible ensuite de modifier le propriétaire avec la commande `chown` (comme *Change Owner*), mais seul root peut l'utiliser, afin d'éviter qu'on donne des fichiers aux autres utilisateurs pour contourner les dispositifs de *quotas*, qui imposent des limites par utilisateur à l'utilisation du disque.

Exercice 5.11 — (difficile) la commande `chgrp` permet de changer le groupe du propriétaire d'un fichier. Expliquer pourquoi elle est utile alors que la commande `chown` permet elle aussi de le modifier

Les permissions d'accès aux fichiers

Le champs `mode`, sur deux octets, contient à la fois le type du fichier et ses protections.

Il y a trois sortes de permissions : pour lire, écrire et exécuter le fichier. Sur les fichiers ordinaires, le rôle de chacune ces permissions est évident. Pour rendre un script exécutable, il suffit d'ajouter le bit d'exécution à ses protections.

Sur les répertoires le rôle des protections est un peu moins évident : la lecture permet de consulter la liste des fichiers qui y sont présents, l'écriture permet d'en détruire les fichiers (même si on n'a pas le droit d'accéder au fichier lui-même). L'exécution permet d'accéder (si les permissions du fichier l'autorisent) aux fichiers contenus dans le répertoire (avec le bit d'exécution sans le bit de lecture, on ne peut accéder qu'aux fichiers dont on connaît déjà le nom).

Ces trois bits de protection sont souvent exprimés par les trois lettres `rwx` (comme *read*, *write* et *execute*) ou comme un chiffre en octal, comme dans la table suivante qui regroupe les modes les plus courants

`rwx`	7	tout est permis
`r-x`	5	lire et exécuter
`---`	0	rien n'est permis
`rw-`	6	lire et écrire (texte)
`r--`	4	lire seulement (texte)

Les permissions existent en trois blocs : l'une décrit celles qui s'appliquent au propriétaire du fichier, la seconde celles pour les utilisateurs de son groupe, le troisième celles pour les autres utilisateurs : on indique donc fréquemment sous la forme d'un nombre de trois chiffres en octal, notamment pour la commande `chmod` qui permet de modifier les permissions d'un fichier. Les permissions ordinaires sont `755` pour un répertoire (seul le propriétaire peut ajouter ou détruire des fichiers du répertoire) et `644` (seul le propriétaire peut modifier le contenu du fichier texte).

La commande `chmod` accepte aussi une description symbolique des modifications à apporter aux permissions d'un fichier. Un caractère `+` ou `-` indique s'il faut ajouter ou retirer des permissions. A droite du signe, les caractères `rwx` indiquent quelles sont les permissions concernées. A gauche du signe, les caractères `ugo` comme *user*, *group* et *other* ou bien le caractère `a` comme *all* pour les trois groupes, indiquent dans quels blocs les permissions doivent être modifiées. L'usage le plus courant en est `chmod a+x` qui permet de transformer un fichier de commandes en script exécutable.

Les bits exotiques Il reste trois bits spéciaux dans les permissions : le *sticky bit* et les bits *set user id* et *set group id*.

Le sticky bit était autrefois utilisé pour indiquer que le contenu d'un fichier exécutable devait être conservé après son exécution dans une zone où le système pouvait y accéder rapidement ; cet usage est tombé en désuétude (mais le bit a conservé son nom) ; maintenant on utilise ce bit dans les répertoires pour indiquer que seuls les utilisateurs ayant le droit de modifier le contenu d'un fichier ont le droit de détruire ce fichier : c'est utile pour éviter les sabotages dans un répertoire comme `/tmp` dont les permissions doivent permettre la création de fichiers par tous.

Les bits *set user id* (suid) et le *set group id* servent à indiquer qu'un processus qui fait un exec du fichier doit changer de propriétaire : le nouveau propriétaire du processus devient celui du fichier. C'est cela qui permet d'acquérir les permissions de root, l'administrateur du système, pour exécuter des tâches de maintenance.

L'idée est que la commande `su` va faire des vérifications (l'utilisateur qui lance la commande appartient-il au groupe *wheel*? connaît-il le mot de passe de l'administrateur ?) avant de lancer un shell avec l'identité de l'administrateur.

Le bit suid est à la source de la grande majorité des problèmes de sécurité qu'a connu le système Unix.

Les Access Control List Le mécanisme des groupes n'est pas suffisamment fin pour les environnements où la sécurité est importante, parce qu'il ne permet pas de contrôle au niveau des fichiers : soit un utilisateur appartient à un groupe, et il a accès à tous les documents du groupe, soit il n'y appartient pas et il n'a alors accès à aucun des documents. Pour permettre un contrôle plus fin, l'inode contient des champs `file_acl` et `dir_acl` pour permettre la mise en place de listes d'utilisateurs avec des accès autorisés. Je ne crois pas que ce mécanisme soit très utilisé dans Linux.

Le champs `flags` permet d'étendre les permissions du champs `mode` : on peut l'utiliser pour interdire la modification ou pour indiquer un mode de mise à jour particulier sur le fichier (par exemple que l'atime n'a pas besoin d'être mis à jour).

Les types de fichiers, les liens symboliques

Le champs mode contient aussi l'indication du type de fichier. Nous avons déjà rencontré les fichiers ordinaires et les répertoires. Il existe aussi des fichiers *spéciaux* qui sont utilisés pour nommer les périphériques. Les fichiers sont également utilisés pour nommer des points pour la communication entre processus, qu'on appelle des *fifo* et des *sockets*. Finalement, un dernier type est le *lien symbolique* qui permet de donner des *surnoms* à des fichiers. Nous ne développons pas les *sockets* dans ce chapitre parce qu'elles appartiennent plutôt au réseau.

Les fichiers spéciaux C'est une des particularité d'Unix d'utiliser le système de fichier pour nommer les périphériques. Cela permet aux programmes ordinaires d'accéder aux périphériques comme à des fichiers ordinaires, et de laisser le système de fichier s'occuper de la gestion des permissions d'accès aux périphériques.

La plupart du temps, les fichiers spéciaux sont tous regroupés dans le répertoire `/dev`, où ils ont été placés par le système en fonction des périphériques disponibles, mais il est possible d'en faire de nouveaux avec la commande `mknod`.

Il y a deux sortes de fichiers spéciaux, suivant que les entrées-sorties se font directement sur le périphérique (on parle alors de fichiers *caractères*) ou bien qu'elles passent par des buffers (on parle alors de fichier *bloc*).

La commande `mknod` n'est accessible qu'à l'utilisateur root, puisque sinon n'importe qui pourrait, en fabriquant un fichier spécial qui lui appartient, accéder à n'importe quel périphérique.

Un fichier spécial est identifié, dans son inode, par deux numéros, dits de *majeur* et de *mineur*. On peut considérer que le numéro de majeur identifie le pilote à utiliser, qui dépend du type du périphérique, alors que le numéro de mineur identifie le numéro du périphérique.

Pour illustrer l'utilisation des fichiers spéciaux, nous allons fabriquer une copie du fichier `/dev/tty` qui est utilisé pour représenter le terminal auquel est attaché un processus.

```
 $ ls -l /dev/tty        Examiner le fichier /dev/tty
crw-rw-rw- 1 root tty 5, 0 2008-03-08 09 :09 /dev/tty
                         C'est un fichier spécial caractère,
                         avec un majeur de 5 et un mineur de 0
 $ su                    Acquérir les autorisations de root
 Password:               Donner le mot de passe de l'administrateur
 #                       Le prompt change
 # mknod toto c 5 0      Fabriquer le fichier spécial toto
                         avec les caractéristiques de tty
 # chmod 222 toto        Tous peuvent y écrire
 # exit                  C'est tout pour root
 $ ls -l toto            Examiner les caractéristiques de toto
crw-r-r- 1 root root 5, 0 2008-03-09 04 :08 toto
 $ ls -l /dev/tty        Les permissions de tty n'ont pas changées
crw-rw-rw- 1 root tty 5, 0 2008-03-08 09 :09 /dev/tty
 $ echo foobar > toto    Écrire sur toto
 foobar                  Le texte apparait dans la fenêtre !
 $ rm toto               Détruire le fichier spécial
```

Exercice 5.12 — Identifier les numéros de majeurs qui correspondent à votre disque et à ses partitions.

Exercice 5.13 — Si vous connaissez un autre système d'exploitation que Linux : comment y désigne-t-on les périphériques ? Quels avantages et quels inconvénients cela présente-t-il par rapport à la méthode d'Unix ?

Les fichiers fifo Les fichiers fifo permettent aux programmes d'échanger des données à des points de rendez-vous fixés dans un système de fichier. Quand un processus ouvre le fifo pour y lire ou y écrire, il reste bloqué jusqu'à ce qu'un autre processus ouvre à son tour le fifo. Une fois que les deux ont ouvert le fifo, les données écrites par l'un sont lues par l'autre.

Un exemple d'utilisation avec le lecteur lancé en premier :

```
$ mkfifo toto
```

```
$ while read x; do echo "J'ai lu $x"; done < toto &
[2] 20506
$ ( echo a; echo b; echo c ) > toto; sleep 1
J'ai lu a
J'ai lu b
J'ai lu c
[2]+  Done                    while read x; do
    echo "J'ai lu $x";
done < toto
```

Un autre exemple sur la même fifo avec l'écrivain lancé en premier :

```
$ ( echo a; echo b; echo c ) > toto &
[2] 20509
$ while read x; do echo "J'ai lu $x"; done < toto; sleep 1
J'ai lu a
J'ai lu b
J'ai lu c
[2]+  Done                    ( echo a; echo b; echo c ) > toto
$ rm toto
$
```

Les commandes `sleep 1` servent à donner le temps au processus en arrière-plan de se terminer avant l'affichage du prompteur, pour que la sortie soit plus lisible.

Exercice 5.14 — (facile) Comment la commande `ls` liste-t-elle les *fifo* ?

Les liens symboliques Les liens symboliques sont largement utilisés : ils permettent de fabriquer un nom de fichier qui, quand on tentera d'y accéder renvoie vers un autre fichier d'une façon transparente.

On fabrique les liens symboliques avec la commande `ln -s` ; l'option `-s` comme *symbolique* sert à différencier ces liens des liens *hard* (*en dur*) que nous verrons plus loin. Par exemple, avec

```
$ ln -s /tmp/toto joe
```

on crée dans le répertoire courant un lien symbolique nommé `joe` qui désigne en fait le fichier `/tmp/toto`.

Exercice 5.15 — (facile) comment la commande `ls` liste-t-elle les liens symboliques ?

Exercice 5.16 — Comment un lien symbolique est-il modifié quand le fichier vers lequel il pointait est détruit ? Et quand il est recréé ?

Exercice 5.17 — Peut-on créer un lien symbolique vers un fichier qui n'existe pas ?

Exercice 5.18 — On peut créer un lien symbolique qui pointe vers lui même avec `ln -s toto toto` par exemple. Que se passe-t-il quand on tente d'y accéder ?

Exercice 5.19 — (facile) on peut faire un lien symbolique vers le répertoire courant avec `ln -s . toto`. On peut ensuite changer de répertoire sans bouger avec `cd toto`, qui revient à aller dans le répertoire courant. Comment `pwd`

affiche-t-il le répertoire courant après le changement de répertoire ? Et après avoir changé plusieurs fois de répertoire ? `/bin/pwd` affiche-t-elle la même chose ? (difficile) Expliquer.

La taille du fichier et les blocs de données

Dans le champs `size`, il y a le nombre d'octets que contient le fichier. Le contenu du fichier lui même ne se trouve pas dans l'inode, mais dans des blocs de données qui font normalement quatre Ko.

L'inode contient quinze numéros de blocs qui contiennent les octets du fichiers. Les douze premiers sont des blocs directs : les données se trouvent dans le bloc qui porte le numéro indiqué. Cela permet de stocker simplement des fichiers de 48 Ko.

Le treizième numéro (si la taille du fichier indique qu'il est utilisé) est celui d'un bloc *indirect* : le bloc de données contient 1024 numéros de blocs qui contiennent des données. cela permet d'avoir un fichier qui contient $4 \times 1024 + 48$ Ko de données, soit environ 4 Mo.

Le quatorzième numéro est celui d'un bloc *doublement indirect*, qui contient 1024 numéros de blocs indirects. Il permet d'avoir jusqu'à 16 Go dans un fichier.

Finalement le quinzième et dernier numéro est celui d'un bloc *triplement indirect*, qui contient des numéros de blocs doublement indirects.

Pour éviter les pertes d'espace dues à la taille des blocs, le dernier bloc d'un fichier peut être un *fragment* : certains blocs de données sont divisés en sous parties. Le champs `faddr` contient le numéro du fragment, et le champs `osd2` contient la taille et la position du fragment dans le bloc de donnée qu'utilise le fichier.

Le nombre de liens, les liens hard

Le champs `links_count` contient le *nombre de liens* qu'a le fichier. Il s'agit ici des liens *en dur* (*hard*), et c'est quelque chose de très différent des liens symboliques. L'inode ne contient pas le nom du fichier, ni celui du répertoire dans lequel il apparait ; il est possible d'avoir plusieurs noms pour le même fichier, y compris dans des répertoires différents.

Chaque fois qu'on crée un nouveau lien pour un fichier, le champs `link_count` est incrémenté. Quand on « détruit un fichier », en réalité on ne détruit que le lien vers le fichier (et le champs `link_count` est décrémenté) ; le fichier ne sera réellement détruit que quand le nombre de liens descendra à 0.

Le nombre de liens est notamment modifié chaque fois qu'on crée un répertoire. Puisque chaque répertoire contient un lien (nommé `..`) vers son répertoire parent, chaque nouveau répertoire modifie le nombre de liens vers son répertoire parent. Dans l'exemple suivant, on fabrique un répertoire `foo` ; son nombre de liens vaut 2, puisque le répertoire existe sous les noms `foo` et `foo/.` ; quand on y crée trois répertoires `a`, `b` et `c`, le nombre de liens passe à 5, puisque le répertoire porte aussi les noms `foo/a/..`, `foo/b/..` et `foo/c/..`.

```
$ mkdir foo
$ ls -l foo/.
total 0
$ ls -ld foo/.
drwxr-xr-x 2 jm jm 4096 2008-03-09 16:11 foo/.
```

```
$ mkdir foo/a foo/b foo/c
$ ls -ld foo/.
drwxr-xr-x 5 jm jm 4096 2008-03-09 16:11 foo/.
```

La commande pour créer un nouveau lien vers un fichier est la commande `ln`, comme pour les liens symboliques mais sans l'option `-s`.

Exercice 5.20 — Y a-t-il moyen de fabriquer un lien en dur vers un répertoire ?

Les champs qui restent

Il reste le champs `generation` qui est utilisé par le protocole de partage de fichiers par le réseau NFS qui identifie les fichiers par leurs numéros d'inode. Quand le dernier lien vers un fichier est supprimé, l'inode est marqué comme libre et peut être ré alloué à un autre fichier. Le champs génération sera alors incrémenté, ce qui permettra au protocole NFS de savoir qu'il ne s'agit plus du même fichier, quoique le numéro d'inode soit identique.

Les champs `osd1` et `osd2` ont des utilisations qui peuvent varier suivant le système d'exploitation. Comme on l'a vu, le champs `osd1` sous Linux contient la partie haute de l'uid et du gid du propriétaire du fichier ; le champs `osd2` contient, le cas échéant, les informations complémentaires sur la position du fragment dans le bloc qui le contient.

5.6.4 La commande fsck

La structure de données sur un disque est relativement complexe, et pour accélérer les entrées-sorties en minimisant les déplacements du bras, elle est mise à jour dans un ordre qui peut conduire à des incohérences si l'ordinateur est arrêté au milieu d'une modification. La commande `fsck` sert à vérifier qu'il n'y a pas d'incohérence, ou bien à les corriger s'il en existe.

fsck au boot

A chaque boot, le système lance `fsck` (comme *File System Check*), le programme de vérification de l'état des systèmes de fichiers. La commande vérifie si le superbloc contient le marqueur qui indique qu'il a été arrêté proprement ; si le marqueur n'est pas positionné (ou bien au bout d'un certain nombre de boots : 25 par défaut), la commande `fsck` vérifie la cohérence du contenu du système de fichier.

Au boot, la commande est lancée avec l'option `-p` pour lui indiquer de corriger automatiquement les incohérences mineures (par exemple un inode dans lequel `link_count` vaut 0, mais qui n'apparaît pas dans les inodes libres).

Dans le cas (exceptionnel) où l'incohérence est grave, le processus de boot est interrompu et le système bascule en mode *single user*. Il faut alors lancer la commande `fsck` à la main pour réparer le disque avant de rebooter : `fsck` affiche des questions pour proposer des réparations qui risquent d'entraîner des pertes de données.

Les vérifications de fsck

`fsck` passe par 5 étapes de vérification.

Dans la première étape, fsck vérifie les inodes et les numéros de blocs. Les erreurs graves qui peuvent se produire sont des blocs qui paraissent appartenir à plusieurs fichiers (ce sont des DUPs) et les numéros de blocs qui n'existent pas sur le système de fichier (ce sont des BADs). Dans le cas où un fichier contient des BAD blocks, il n'y a pas tellement mieux à faire que de détruire le fichier comme fsck le propose. Dans le cas où il y a des DUPs, fsck propose de détruire tous les fichiers qui en contiennent ; on peut parfois reconnaître le fichier qui pose problème et ne détruire que celui-là.

Dans la seconde et la troisième étape, fsck vérifie le contenu des répertoires : chaque répertoire a-t-il la structure prévue, chaque numéro d'inode correspond-il à un inode du système de fichier, chaque répertoire contient-il les entrées `.` et `..`, y a-t-il moyen d'atteindre chaque fichier en partant de la racine du système de fichier ? S'il y a des fichiers qu'on ne peut pas atteindre, fsck les place dans le répertoire `lost+found` qui se trouve à la racine du système de fichier, avec son numéro d'inode en guise de nom puisqu'il ne peut pas retrouver le nom du fichier.

Dans la quatrième étape, fsck vérifie que le champs `link_count` de chaque inode est correct.

Dans la cinquième étape, il vérifie que les listes des blocs et d'inodes libres dans les groupes de cylindres sont corrects.

La commande fsck peut prendre plusieurs minutes quand le disque est gros, puisqu'elle doit le parcourir plusieurs fois pour en extraire les informations à vérifier.

5.6.5 Les systèmes de fichier journalisés, Ext3fs

Pour éviter que les modifications dans l'ordre d'écriture des blocs ne puissent conduire à des incohérences, une solution est de conserver sur le disque un *journal* : une liste des modifications qui vont être apportées au système de fichier, écrite d'une façon compacte et contiguë.

Si le système est interrompu, le système de fichier trouve la liste de ces modifications à effectuer quand il redémarre : il effectue ces modifications et le système de fichier retrouve sa cohérence.

Le successeur de ext2fs s'appelle ext3fs : il s'agit essentiellement de la même organisation que ext2fs avec la journalisation en plus. Ext3 permet aussi d'organiser les répertoires sous la forme de hash tables. Ext3fs est le système de fichier par défaut sur Mandriva.

5.7 La multitude des systèmes de fichiers

Un ordinateur sous Linux doit pouvoir accéder à plusieurs types de systèmes de fichiers : en plus des fichiers ordinaires, il a à traiter les DVDs et les CDROMs, ceux d'une partition où se trouvent les fichiers d'un système d'exploitation Microsoft, ceux qui se se trouvent sur d'autres ordinateurs connectés par le réseau, etc. Cette capacité à traiter des systèmes de fichiers divers est mise en oeuvre dans les noyaux Unix avec des *vnodes*, que nous présentons dans cette section. Nous présentons ensuite quelques systèmes de fichiers communément rencontrés.

5.7.1 Les vnodes

Les noyaux Unix et Linux utilisent des *vnodes*, avec un *v* comme dans *virtual*, pour accéder à des systèmes de fichiers divers.

En bref, les vnodes forment une couche logicielle qui permet d'accéder à un système de fichier quelconque avec une interface qui ressemble à celle des *inodes*, mais avec en dessous les données d'un système de fichier qui peut être très différent.

Ceci est réalisé dans le noyau du système avec une forme de programmation orientée objet qui repose néanmoins sur le langage C, utilisé dans le noyau, qui n'est pas spécialisé dans les objets. Le vnode est une structure qui contient les données représentant un fichier dans le système de fichier et un ensemble de pointeurs sur des fonctions (qui jouent le rôle des *méthodes*) qui réalisent, quand c'est possible, les fonctions d'accès aux fichiers équivalentes à celles qu'on rencontre sur les inodes du système de fichier historique. Il y a de même des objets pour représenter le superbloc et les répertoires, sous forme de structures et de pointeurs sur des fonctions.

5.7.2 Le système de fichier nfs

Un système de fichier important dès qu'on dispose de plusieurs ordinateurs est celui qui permet de partager des fichiers entre machines. Historiquement, ce rôle est joué sous Unix par NFS (comme *Network File System* : « système de fichier réseau »). Il a été conçu pour le système Unix et offre une interface qui ressemble à celle des systèmes de fichiers usuels sous Unix, comme ext2fs.

NFS est un système ancien et malgré ses évolutions au cours des années, il reste fondé sur quelques supposés sur la vitesse des échanges de données sur le réseau ou l'authentification des utilisateurs qui ne correspondent plus à la réalité (ses détracteurs disent que NFS signifie *No File Security*).

Un inconvénient significatif de NFS est qu'il ne permet d'exporter que les fichiers ordinaires, mais pas les fichiers spéciaux ou les fifo : toutes les ressources qui sous Unix sont représentées comme des fichiers ne peuvent pas être partagées entre ordinateurs avec le protocole NFS.

5.7.3 Les systèmes de fichiers synthétiques

L'interface sous forme de systèmes de fichiers est très pratique : tout le monde comprend sa structure et on peut utiliser pour l'explorer des outils familiers, aussi bien en mode ligne qu'en mode graphique. Pour cette raison, un certain nombre de ressources sont présentées dans l'univers Linux sous forme de systèmes de fichiers *synthétiques* : ils ne correspondent pas à des fichiers présents sur un périphérique de stockage comme un disque ; ils sont utilisés pour présenter des ressources diverses du système.

/proc

Le plus important système de fichiers synthétique est `/proc`, qui sert à représenter les processus. On trouve dans le répertoire `/proc` un répertoire par processus actif, dont le contenu permet de consulter les caractéristiques du processus. Le répertoire a le PID en guise de nom.

Dans le répertoire, on trouve par exemple une description de l'état du processus dans le fichier `status`. Le répertoire courant se trouve sous le nom `cwd` (comme *Current Working Directory*). Ainsi, on peut connaître les répertoires de travail de tous les processus avec la commande shell :

```
for i in /proc/[0-9]*/cwd ; do
   (cd $i ; /bin/pwd)
done | sort -u
```

Les fichiers ouverts par le processus apparaissent dans le sous-répertoire `fd`, avec le numéro du descripteur en guise de nom. Puisque la variable shell `$$` contient le numéro de processus du shell et que le descripteur 2 est la sortie standard pour les erreurs, shell, la commande `echo erreur > /proc/$$/fd/2` permet d'afficher un message d'erreur sur le bon descripteur de fichier. (Si on n'utilise pas cette méthode, on est obligé d'employer une construction spécifique du shell pour afficher un message sur le bon descripteur : `echo erreur 1>&2` ; cette méthode ne fonctionne pas avec le shell `csh` qui emploie une autre syntaxe.) Dans le sous-répertoire `task`, on trouve des sous-sous-répertoires qui contiennent des informations spécifiques sur chacun des threads du processus.

On trouve aussi dans `/proc` des fichiers qui donnent des informations sur l'état de l'ordinateur et du système : `cpuinfo` contient une description sur l'état du processeur, `meminfo` celle de l'état de la mémoire. Le fichier `/proc/filessystems` liste tous les types de systèmes de fichier que le noyau connaît.

Le point important des systèmes de fichiers synthétiques, c'est que les données qu'on lit ne sont pas stockées de manière explicite. Lors d'une lecture d'un de ces fichiers, le noyau du système transmet le travail au système de fichier ; celui-ci consulte les données du noyau, et c'est en fonction de leurs valeurs qu'il va synthétiser, au vol, le contenu du fichier que renverra la fonction de lecture.

Exercice 5.21 — Écrire un programme `psfind` à qui on donne des critères et qui affiche les PIDs des processus qui satisfont ces critères. Il serait utile de pouvoir utiliser comme critère le nom de la commande, l'utilisateur qui l'a lancée, le répertoire de travail, la taille mémoire utilisée, le terminal de contrôle.

Exercice 5.22 — Écrire un programme `psprint` à qui on donne un numéro de processus et qui affiche les informations qui concernent ce processus à la manière de `ps l`.
On pourra combiner les deux commandes précédentes pour avoir une commande équivalente à `ps` qui permet un contrôle fin des processus affichés.

/dev

Le répertoire `/dev` fonctionne également comme un système de fichier synthétique. Quand on branche ou qu'on retire un périphérique amovible, comme un porte-clefs USB, les fichiers spéciaux qui les représentent apparaissent et disparaissent.

5.8 Supplément : le système Plan 9

Il existe un système d'exploitation nommé Plan 9 qui est principalement fondé sur cette notion de fichiers montés : la plupart des ressources y appa-

raissent comme des systèmes de fichiers et les opérations spéciales aux périphériques s'y font avec de simples lectures et écritures dans des fichiers de contrôle.

Avec cette conception et un protocole de partage de fichiers adéquat appelé 9P, il est possible de partager toutes les ressources avec n'importe quelle machine qui pratique le protocole.

La commande `mount` y existe, comme sous Unix, mais au lieu d'un fichier spécial qui désigne un périphérique, on y monte un *descripteur de fichier* où échanger des messages du protocole 9P.

Chapitre 6

Les entrées-sorties : noyau, bibliothèque, processus

Le chapitre précédent à présenté comment chaque fichier est identifié par un numéro de périphérique et un numéro de fichier sur le périphérique. (Le numéro est un numéro de *vnode* qui contient tout ce qui est nécessaire pour accéder au fichier, comme on l'a vu dans le cas particulier des *inodes* du système de fichier *ext2fs*).

Dans ce chapitre, nous exposons comment se fait la liaison entre cette vision des fichiers et celle des programmes qui effectuent des lectures ou des écritures dans les fichiers.

Une fois le chapitre assimilé, vous comprendrez la répartition du travail entre la bibliothèque standard d'entrée-sortie, les appels systèmes émis par le processus, les tables du noyau du système d'exploitation. La maitrise de la différence entre *fichier* et *descripteur de fichier* vous permettra de mettre en oeuvre les redirections d'entrées sorties.

6.1 Préliminaire : simplifier les includes

Les programmes que nous examinerons dans ce chapitre utilisent des fonctions de la bibliothèque C dont les prototypes sont définis dans différents fichiers `.h` à inclure dans les programmes.

Ces fichiers sont organisés sous une forme dispersée, parce qu'il y a une vingtaine d'années le compilateur C était si lent qu'on souhaitait minimiser le nombre de lignes qu'il avait à traiter.

Cette considération n'est plus d'actualité, mais la dispersion des fichiers à inclure persiste. (Pour trouver les fichiers à inclure, la page de manuel est utile.) Pour simplifier notre travail et ne pas encombrer les exemples avec des lignes inutiles, nous allons placer dans un fichier `sys.h` des directives pour inclure *tous* les fichiers `.h` dont nous pourrions avoir besoin. Nos programmes incluront systématiquement ce fichier.

Le contenu du fichier `sys.h` est :

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

FIG. 6.1 – Un programme effectue des écritures avec la stdio : celle-ci accumule les octets dans un buffer puis les écrit d'un coup avec l'appel système `write`. Dans le noyau, le numéro du fichier est utilisé comme index dans la table des fichiers ouverts du processus pour trouver l'adresse d'une entrée dans la table globale des fichiers ouverts du système. Une entrée dans la table globale des fichiers ouverts contient (entre autres) le mode d'ouverture du fichier, le numéro de l'octet courant et l'identification du fichier.

```
# include <stdlib.h>
# include <unistd.h>
# include <sys/stat.h>
# include <sys/types.h>
# include <fcntl.h>
# include <sys/wait.h>
```

Vous pouvez faire pareil dans vos propres programmes : il est souvent plus simple d'avoir un seul fichier à inclure qui contient tous les prototypes de fonctions et les structures de données, plutôt que d'utiliser des mécanismes complexes et peu sûrs comme `configure`.

6.2 L'organisation générale des tables

Cette section présente l'organisation des structures de données entre un descripteur de flux, accessible en C sous un nom comme `stdin` ou `stdout` et les informations nécessaires pour procéder effectivement à la lecture ou à l'écriture des données dans le fichier.

Cette organisation est résumée dans la figure 6.1 et expliquée (de droite à gauche) dans les paragraphes qui suivent.

6.2.1 Dans le noyau : la table des fichiers ouverts

A l'intérieur du noyau, il existe une table qui décrit tous les fichiers actuellement ouverts. Chaque entrée de la table contient trois champs essentiels : d'une part, l'identité du fichier avec le couple (*numéro de périphérique, numéro de*

vnode), d'autre part le type d'opérations d'entrées sorties qu'on peut faire sur le fichier (lecture, écriture ou les deux), finalement le numéro du prochain octet dans le fichier à lire ou à écrire dans le fichier.

Chaque ouverture d'un fichier par un processus correspond à la création d'une nouvelle entrée dans cette table.

6.2.2 Dans le noyau : les fichiers ouverts d'un processus

Le noyau contient une structure par processus qui décrit les informations qui le concernent. On y trouve notamment une table des fichiers ouverts du processus. Chaque entrée de la table ne contient en pratique qu'un pointeur sur une entrée dans la table générale des fichiers ouverts.

6.2.3 Dans le processus : le numéro de fichier ouvert

Pour un processus, un fichier ouvert se décrit par un (petit) nombre entier. Le noyau reçoit ce numéro de fichier ouvert dans les appels système qui effectuent des entrées-sorties, et l'utilise comme index dans la table des fichiers ouverts du processus pour accéder à une entrée dans la table des fichiers ouverts du système.

6.2.4 Dans la stdio : le pointeur sur la structure `FILE`

Les programmes accèdent rarement aux fichiers directement avec les appels système ; ils font plutôt appel à la *bibliothèque d'entrée-sortie standard* (dont on abrège souvent le nom en *stdio*), dont un rôle important est d'agréger les entrées sorties pour diminuer le nombre d'appels système.

Pour la stdio, un fichier ouvert se décrit avec un pointeur sur la structure `FILE` qui contient (entre autres) un buffer dans lequel les octets sont accumulés et le numéro qui correspond au descripteur de fichier au niveau appel système. (Un fichier ouvert s'appelle en général un *stream*, c'est à dire un flux d'octets dans la documentation de la stdio.)

6.3 Les appels système d'entrée-sortie

Cette section décrit les appels système les plus courants pour effectuer des entrées sorties : ouverture, fermeture, positionnement, lecture et écriture.

6.3.1 Descripteurs de fichiers

Dans un processus, les fichiers ouverts sont identifiés au niveau des appels système par des (petits) entiers. C'est une pure convention, mais presque universellement respectée que le descripteur 0 fait référence à l'entrée standard, le descripteur 1 à la sortie standard et le descripteur 2 à l'erreur standard.

6.3.2 Ouvrir et fermer un fichier

Un processus ouvre un fichier avec l'appel système `open`, qui prend deux ou trois arguments. Le premier argument est toujours le chemin d'accès au fichier. Le second est une combinaison de flags qui indiquent de quelle manière le fichier

doit être ouvert ; dans le cas où les flags indiquent que le fichier doit être créé, le troisième argument indique les protections du fichier.

Les flags les plus ordinaires sont `O_RDONLY`, `O_WRONLY` et `O_RDWR` pour lire, écrire ou lire et écrire à la fois. La création se demande avec le flag `O_CREAT`.

La fermeture d'un fichier s'obtient avec l'appel système `close`. La plupart des programmeurs considèrent qu'un `close` ne peut jamais échouer et ne prennent pas la peine de tester la valeur renvoyée.

Exercice 6.1 — A quelles valeurs correspondent les constantes `O_RDONLY`, `O_WRONLY` et `O_RDWR` ?

Exercice 6.2 — (Nombre maximum de fichiers ouverts) Le programme suivant ouvre autant de fichiers que possible. Le compiler, vérifier qu'il n'ouvre pas les fichiers 0, 1 et 2 (ils sont déjà ouverts) et déduire de ce qu'il imprime le nombre maximum de fichiers qu'un processus peut ouvrir.

```
# include "sys.h"

int
main(){
  int t;

  for(;;){
    t = open("toto", O_RDONLY);
    if (t < 0){
      perror("open");
      exit(1);
    }
    printf("%d: ok\n", t);
  }
}
```

Exercice 6.3 — (Création de fichier) Vérifier que le programme suivant ne peut pas ouvrir le fichier s'il n'existe pas avant son lancement. Quel est le message d'erreur ? Trouver ensuite, soit dans la page de manuel, soit dans l'exercice suivant, une façon de créer le fichier s'il n'existe pas.

```
# include "sys.h"

int
main(){
  int t;

  unlink("toto"); // detruire le fichier s'il existe

  t = open("toto", O_WRONLY);
  if (t < 0){
    perror("toto");
    exit(1);
  }
  exit(0);
}
```

Exercice 6.4 — (Redirection de sortie) Le programme suivant ferme le fichier 1 puis rouvre un autre fichier avant de lancer une commande (via `exec`). Vérifier que la commande imprime son résultat dans le fichier ouvert au lieu de l'imprimer sur la sortie.

```
# include "sys.h"

int
main(){
  int t;

  t = close(1); // fermer la sortie
  assert(t >= 0);
                // l'ouvrir de nouveau vers le fichier toto
  t = open("toto", O_WRONLY|O_CREAT, 0666);
  assert(t == 1);
                // lancer la commande
  execl("/bin/ls", "ls", "-l", 0);
  perror("execv");
  exit(1);
}
```

Exercice 6.5 — La redirection d'entrée sortie peut se faire au niveau de la stdio avec la fonction `freopen`. Réécrire le programme précédent en utilisant cette fonction à la place de `open` et `close`.

Exercice 6.6 — (important) Modifier le shell vu dans le chapitre sur la création de processus pour permettre la redirections d'entrée-sortie comme dans un shell ordinaire, avec les chevrons.

6.3.3 Lire et écrire

La lecture et l'écriture se font avec un seul appel système chacun : `read` et `write` ; leurs trois arguments sont le descripteur de fichier, l'adresse des données et le nombre d'octets concernés. La valeur renvoyée est le nombre d'octets qui ont été effectivement lus ou écrits.

Pour écrire une chaine de caractères sur la sortie standard, on peut utiliser l'appel système `write` de plusieurs manières :

```
write(1, "foo bar", 7);
write(1, "foo bar", strlen("foo bar"));
write(1, "foo bar",  sizeof "foo bar" - 1);
```

Dans les trois cas, le 1 désigne le descripteur de fichier (ici la sortie standard), le deuxième argument est l'adresse des caractères à écrire et le troisième est, sous trois formes différentes, le nombre d'octets à écrire.

La valeur renvoyée par `write`

L'appel système `write` renvoie le nombre d'octets qui ont été effectivement écrits, qui peut être différent de celui demandé par le processus. Dans le cas d'une erreur d'écriture, `write` renvoie une valeur inférieure à 0.

Il convient de prendre garde au fait que l'appel système correspond souvent seulement à la recopie des octets dans un buffer du système (ce qui est en général suffisant : les données sont maintenant sous la responsabilité du noyau du système; cependant cela peut poser un problème si l'ordinateur s'arrête brusquement, car le buffer du système pourra ne pas avoir été écrit sur le disque). (Pour être (presque) certain que les données sont écrites sur le disque, il existe un appel système `fsync` qui bloque le processus tant que le noyau n'a pas vidé son buffer.)

La lecture avec `read`

Pour lire des données, on utilise l'appel système `read` qui prend trois arguments qui ressemblent à ceux de `write` : le descripteur de fichier, l'adresse où placer les octets lus et le nombre maximal d'octets à lire (c'est à dire la taille utilisable à l'adresse passée comme deuxième argument). Comme `write`, l'appel système `read` renvoie le nombre d'octets lus ou une valeur inférieure à 0 s'il se produit une erreur de lecture; c'est tout à fait ordinaire pour `read`, à la différence du `write`, de ne lire qu'une partie des octets demandés.

Un cas particulier important est celui d'un `read` qui ne lit aucun octet et renvoie donc la valeur 0 : c'est utilisé par le système pour indiquer qu'on a atteint la fin du fichier. Soulignons qu'atteindre la fin du fichier n'est pas considéré comme une erreur : `read` ne renvoie pas une valeur négative.

Une façon de recopier un fichier avec une boucle équivalente à `while((c = getchar()) != EOF) putchar(c)` serait la boucle de la fonction `copier` du programme suivant :

```
# include "sys.h"

enum {
  Size = 1,
};

char buffer[Size];

/* copier  --   recopie le fichier decrit par in dans out */
void
copier(int in, int out){
  int t;

  while((t = read(in, buffer, sizeof buffer)) > 0)
    write(out, buffer, t);
  if (t < 0)
    perror("read");
}

int
main(){
  copier(0, 1); /* recopie l'entree standard dans la sortie standard */
  return 0;
}
```

Noter que dans la fonction `copier`, on utilise le nombre d'octets lus renvoyé par `read` comme troisième argument pour le `write`. Il ne faut pas écrire plus d'octets qu'on en a lu quand on recopie !

Exercice 6.7 — Quand on est au bout du fichier, l'appel système `read` renvoie 0 ; si on essaye encore de lire des données par la suite, `read` renvoie-t-il de nouveau 0 ou bien une valeur négative pour indiquer une erreur de lecture ? (facile) Écrire un programme C pour trouver la réponse à cette question. (difficile) Trouver dans la documentation l'endroit où ce point est mentionné.

Exercice 6.8 — Quelle est la taille maximale pour le tableau `buffer` dans le programme précédent ? (il suffit de tester le programme en faisant varier la valeur de la constante `Size`).

Exercice 6.9 — à l'aide de `time`, d'un gros fichier et du programme ci-dessus, mesurer le temps d'exécution du programme en fonction de la taille du buffer définie par `Size`. Faire varier `Size` entre 1 et 2^{20} en doublant la taille à chaque test.

Les lectures sur le clavier (facultatif)

Quand un descripteur de fichier fait référence au clavier, les choses se passent d'une façon un peu particulière : l'appel système `read` reste bloqué jusqu'au moment où on a tapé une ligne complète. Au moment où on tape le caractère de fin de ligne (qui porte parfois les étiquettes *Enter* ou *Newline* ou *Envoi* ou plus couramment une flèche suggérant un *retour du chariot* sur une machine à écrire mécanique), la ligne est considérée comme complète et le pilote de terminal qui accumulait les caractères de la ligne les transfère au noyau du système ; ces caractères sont placés dans le buffer fourni lors de l'appel de `read` et l'appel système se termine.

Ce mécanisme d'accumulation des caractères permet d'avoir une façon uniforme d'entrer les corrections : le caractère d'effacement (*BackSpace* ou *Delete* ou *Suppr* ou plus couramment une flèche vers la gauche) retire le dernier caractère placé dans le buffer de la ligne, *et le programme n'a pas besoin de gérer ces corrections.* De la même façon, on peut effacer toute la ligne avec le caractère `CTRL U` et le dernier mot avec `CTRL W`.

(On peut modifier le comportement du pilote de terminal avec la commande `stty` : notamment, `stty cbreak` permet de lui demander de renvoyer les caractères dès qu'ils sont tapés. Constater le comportement de la commande `cat` après cette commande : la copie de chaque caractère apparaît dès qu'il a été tapé ; le `CTRL D` ne joue plus son rôle et il faut interrompre le programme avec `CTRL C`. On peut remettre le pilote dans son état normal avec `stty -cbreak`.)

La figure 6.2 résume l'interaction entre les différents modules : le processus demande une lecture sur le clavier avec l'appel système usuel `read` au noyau du système d'exploitation ; celui-ci détecte qu'il s'agit d'une lecture sur un périphérique et passe la main à la fonction de lecture du pilote de terminaux ; celle-ci attend que le buffer de ligne associé au terminal contienne une ligne complète, puis renvoie les caractères de la ligne au noyau qui recopie ces caractères dans l'espace mémoire du processus, dans le buffer que celui-ci avait passé comme second argument à `read`.

FIG. 6.2 – Une vision de l'imbrication des entrées sorties entre un processus, le clavier et l'écran : les caractères tapés au clavier sont accumulés par le pilote de terminal dans un buffer ligne, avec un écho immédiat à l'écran. Quand un processus lit sur le clavier, le noyau attend que le pilote indique qu'une ligne complète a été lue et transfère cette ligne au processus.

La fin de fichier sur le clavier Le caractère `CTRL D` joue un rôle particulier pour le pilote de terminal : il lui indique qu'il doit envoyer les caractères présents dans le buffer sans attendre la fin de la ligne. Quand on tape un `CTRL D` en début de ligne, le pilote de terminal renvoie tout de suite les caractères présents dans la ligne : comme la ligne est vide, cela se traduit par un `read` qui renvoie 0 (le nombre de caractères lus) ; la bibliothèque d'entrées sorties standard interprète cette valeur comme une fin de fichier ; si le processus avait demandé la lecture avec `fgetc`, la fonction renvoie `EOF` (c'est à dire -1) pour lui signifier cette fin de fichier ; si la lecture avait été demandée avec `fgets`, c'est la valeur `NULL` (c'est à dire 0) qu'elle renvoie.

6.3.4 Choisir la place du prochain octet lu ou écrit

Pour choisir le prochain octet à lire dans un fichier, on utilise l'appel système `lseek`. Les trois arguments en sont le descripteur de fichier, la position souhaitée et une constante qui indique la manière de calculer la position : `SEEK_SET` compte depuis le début, `SEEK_END` depuis la fin et `SEEK_CUR` depuis la position courante. En bonus `lseek` (s'il a réussi) renvoie la nouvelle position courante dans le fichier.

Exercice 6.10 — Expliquer comment le programme suivant mesure la taille d'un fichier.

```
# include "sys.h"

/* mesurer  --  imprime le nom et la taille d'un fichier, renvoie 0 si ok */
int
mesurer(char * path){
  int fd;
  int size;

  fd = open(path, O_RDONLY);
  if (fd < 0)
    return 1;
  size = lseek(fd, 0, SEEK_END);
  if (size >= 0)
    printf("%s: %d octets\n", path, size);
  close(fd);
  return size < 0;
}

int
main(int ac, char * av[]){
  int i, err;

  err = 0;
  for(i = 1; i < ac; i++)
    err += mesurer(av[i]);

  exit(err != 0);
}
```

Exercice 6.11 — (hors sujet) Modifier le programme précédent pour imprimer la taille du fichier en utilisant des unités qui rendent le résultat plus lisible à la façon de l'option `-h` de la commande `df`.

Exercice 6.12 — (hors sujet, pas élémentaire) Modifier le programme précédent pour mesurer la taille du fichier sans l'ouvrir avec l'appel système `stat`. Constater que cela permet de connaître la taille des fichiers dont on n'a pas le droit de lire le contenu.

Exercice 6.13 — Le programme suivant essaye de positionner l'octet courant *avant* le premier octet du fichier `toto`. Regarder ce qui se passe. S'il y a une erreur, quelle est-elle ?

```
# include "sys.h"
int
main(){
  int fd;
  int t;

  fd = open("toto", O_RDONLY);
  if (fd < 0){
    perror("ouverture de toto");
    exit(1);
  }

  t = lseek(fd, -1, SEEK_CUR);
  if (t < 0){
    perror("lseek");
    exit(1);
  }

  printf("L'octet courant est le numero %d\n", t);
  exit(0);
}
```

Exercice 6.14 — Que se passe-t-il si on essaye de faire un `lseek` sur un descripteur de fichier qui ne permet pas de choisir le prochain octet courant, comme le clavier par exemple ? Et si on spécifie un déplacement de 0 octets par rapport à l'octet courant (c'est à dire *pas d'opération*) ?

6.4 Implications, applications

Cette partie présente des conséquences intéressantes de l'organisation des données vue dans le début du chapitre. Cela nous permet de comprendre certains aspects du fonctionnement du système. Dans le cadre du cours, on peut aussi vérifier qu'on a correctement saisi la manière dont les choses s'organisent.

FIG. 6.3 – Quand un processus a ouvert deux fois un fichier, chaque descripteur pointe sur une entrée différente dans la table des fichiers ouverts du système et les opérations sur un descripteur n'affectent pas celles sur l'autre descripteur.

6.4.1 Deux entrées dans la table des fichiers ouverts vers le même fichier

Si un processus ouvre deux fois le même fichier, chaque descripteur conduira vers deux entrées distinctes dans la table des fichiers ouverts avec chacune son numéro d'octet courant, comme suggéré dans la figure 6.3. Donc les octets qui seront lus dans un des descripteurs n'affecteront pas l'autre descripteur.

Pour cette raison, le programme suivant va lire (et afficher) deux fois le contenu du fichier toto.

```
# include "sys.h"

int
main(){
  int fd[2];
  char lu[2][1024];
  int nlu;
  int i, t;

  for(i = 0; i < 2; i++){
    fd[i] = open("toto", O_RDONLY);
    if (fd[i] < 0){
      perror("toto");
      exit(1);
    }
  }

  for(nlu = 0; nlu < sizeof lu[0] - 1; nlu++)
    for(i = 0; i < 2; i++){
      t = read(fd[i], &lu[i][nlu], 1);
```

FIG. 6.4 – L'état des tables après l'appel système `b = dup(a)`. les deux descripteurs de fichier pointent vers la même entrée dans la table des fichiers ouverts, à la différence de la figure 6.3 ; le numéro de l'octet courant est donc partagé par les deux descripteurs de fichiers.

```
      if (t < 1){
        lu[i][nlu] = 0;
        break;
      }
      lu[i][nlu + 1] = 0;
    }
  if (t < 0){
    perror("read");
    exit(1);
  }

  for(i = 0; i < 2; i++)
    printf("On a lu dans le fichier %d : '%s'\n", i, lu[i]);
  exit(0);
}
```

6.4.2 Deux descripteurs différents vers le même fichier ouvert

On peut aussi avoir deux descripteurs différents vers le même fichier ouvert : le cas le plus fréquent est celui du fork, mais il existe un appel système nommé `dup` qui permet d'obtenir le même résultat.

Un processus passe à `dup` un descripteur de fichier et reçoit en retour un *autre* descripteur qui décrit non seulement le même fichier, mais *la même entrée* dans la table générale des fichiers ouverts. Quand on modifie l'octet courant sur un des descripteurs (par exemple en lisant des données), cela modifie aussi l'octet courant de l'autre descripteur. L'organisation des structures de données est montrée sur la figure 6.4, à comparer avec la figure 6.3.

Dans le programme suivant, on ouvre le fichier `toto` et on duplique un descripteur vers le fichier toto, puis on lit alternativement à travers les deux descripteurs comme dans le programme précédent. Comparer avec le programme précédent.

```
# include "sys.h"

int
main(){
  int fd[2];
  char lu[2][1024];
  int nlu;
  int i, t;

  fd[0] = open("toto", O_RDONLY);
  if (fd[0] < 0){
    perror("toto");
    exit(1);
  }
  fd[1] = dup(fd[0]);
  if (fd[1] < 0){
    perror("dup");
    exit(1);
  }

  for(nlu = 0; nlu < sizeof lu[0] - 1; nlu++)
    for(i = 0; i < 2; i++){
      t = read(fd[i], &lu[i][nlu], 1);
      if (t < 1){
        lu[i][nlu] = 0;
        break;
      }
      lu[i][nlu + 1] = 0;
    }
  if (t < 0){
    perror("read");
    exit(1);
  }

  for(i = 0; i < 2; i++)
    printf("On a lu dans le fichier %d : '%s'\n", i, lu[i]);
  exit(0);
}
```

On peut constater que chaque octet du fichier n'est lu qu'une seule fois, à travers l'un ou l'autre des descripteurs ; parce qu'on lit alternativement à travers les deux descripteurs, on a lu les caractères pairs dans un des descripteurs et les caractères impairs dans l'autre.

```
$ gcc -g -Wall dup1.c                    compiler le programme
$ echo foo et bar > toto                 mettre des données dans le fichier toto
On a lu dans le fichier 0 : 'foe a
'
On a lu dans le fichier 1 : 'o tbr'
```

Commandes et dup (facultatif) Les shells `sh` et `bash` permettent d'utiliser l'appel système `dup` depuis la ligne de commande, en spécifiant la redirection d'un des descripteurs vers la duplication d'un autre descripteur, en ajoutant le caractère esperluette après le chevron qui indique la redirection. Pour stocker messages ordinaires et messages d'erreur dans un seul fichier, on fait :

```
$ make > make.out 2>&1
```

Le `2>&1` indique que la sortie 2 (l'erreur) sera une duplication de la sortie 1 (sortie standard) qu'on a déjà redirigée avec `> make.out`. De même, pour imprimer un message d'erreur avec `echo` qui imprime sur la sortie standard, on utilise dans les scripts shells :

```
echo message 1>&2
```

La sortie standard (numéro 1) de la commande `echo` est une copie de la sortie d'erreur du shell (numéro 2)

Descripteurs de fichier après un fork Le partage de descripteurs de fichier est fréquent après un `fork`. Les processus parent et enfant contiennent des pointeurs différents vers la même entrée de la table générale des fichiers ouverts. De ce fait, quand l'un des processus lit des données sur un des descripteurs, le numéro de l'octet courant est modifié *pour les deux processus*. On peut voir un exemple de l'organisation des données dans la figure 6.5.

Le programme suivant ressemble au précédent, mais cette fois ce sont les deux processus qui lisent à tour de rôle. Le `sleep(1)` après la lecture de chaque octet est destiné à synchroniser les processus.

6.4.3 Le `goto` du shell d'Unix V6 (facultatif)

Le shell d'Unix Version 6 ne comprenait pas de primitives pour la programmation. Cependant, il permettait de faire des boucles en utilisant des *commandes*, notamment `if`, *goto* et `:`, comme dans le script suivant qui lance la commande `faire` tant que la commande `tester` renvoie vrai :

```
: debut
faire
if tester "goto debut"
```

La commande `if` lance (avec `fork` et `exec`) une première commande, puis lance la seconde si la première est sortie avec 0. La commande `:` ne fait rien ; elle sert à définir une étiquette. La commande `goto` se positionne au début du fichier (avec un `lseek`), puis le parcourt à la recherche de la ligne qui contient l'étiquette ; après la sortie du `goto`, le shell continue la lecture à partir de cet octet et exécute donc la commande qui suit.

FIG. 6.5 – Après un fork, les processus parent et enfant sont chacun décrits dans le noyau par un structure `task_struct`. Chacune contient une liste des fichiers ouverts du processus qui pointe vers la même entrée dans la table générale des fichiers ouverts.

La commande `:` existe encore dans le shell, et il est courant de l'employer à la place de `true` quand on veut un test qui soit toujours vrai, comme dans le programme suivant qui liste les propriétés du fichier toutes les 60 secondes :

```
$ while : ; do ls -l fichier ; sleep 60 ; done
```

Exercice 6.15 — Ajouter les commandes `if`, et `goto` au mini-shell du chapitre sur la création de processus.

6.4.4 Dup2 (facultatif)

Il existe une variante de l'appel système `dup` qui permet de choisir directement le numéro du nouveau descripteur de fichier ; elle s'appelle `dup2`. C'est en général elle qui est utilisée pour mettre en oeuvre la redirection d'entrées sorties.

6.5 Le buffer de la stdio

La gestion du buffer dans lequel la stdio accumule les caractères pour les passer en blocs aux appels système est un peu confuse. Par défaut, la stdio utilise un buffer de quelques Ko, mais quand le descripteur de fichier désigne un terminal, le buffer est limité à une ligne.

On peut contrôler la bufferisation de la stdio avec les fonctions `setbuffer` et `setlinefuf`. Pour reprendre un exemple légèrement modifié du chapitre sur la création de processus, si on force la sortie à utiliser un buffer, le résultat est très différent :

```
# include "sys.h"

int
main(){
```

```
  int i;
  char buffer[4 * 1024];

  setbuffer(stdout, buffer, sizeof buffer);

  for(i = 0; i < 10; i++){
    printf("%d", i);
    fork();
  }
  printf("\n");
  return 0;
}
```

A cause du `setbuffer`, chaque processus accumule les caractères à écrire dans le buffer qui est recopié dans le processus enfant. C'est au moment où chaque processus sort que le contenu de son buffer est imprimé. Tous les processus écrivent donc la même ligne qui contient tous les chiffres de 0 à 9.

Chapitre 7

Communication entre les processus

Ce chapitre aborde la question de la synchronisation et de la communication entre les processus.Il présente principalement les deux mécanismes les plus utilisés sous Unix : les pipes et les signaux.

7.1 Les fichiers

Les fichiers sont un mécanisme important et souvent pratique de communication entre les processus. Ils peuvent être traités du point de vue des programmes comme des zones de mémoire partagées dans lesquelles on lit et on écrit avec des primitives spéciales (celles qui servent à lire et à écrire dans les fichiers).

Cela permet de faire communiquer simplement, avec des mécanismes bien connus, les processus dont on est certain qu'ils partagent un accès à un système de fichier commun ; c'est bien évidemment le cas pour les processus qui se trouvent sur le même ordinateur, mais aussi pour des processus sur des ordinateurs diférents via l'utilisation des systèmes de partage de fichier en réseau du type NFS.

Un inconvénient de l'utilisation des fichiers est que les multiples niveaux de cache (dans la librarie d'entrées sorties standard, dans le noyau, dans le protocole réseau le cas échéant) ne permettent pas une synchronisation fine.

7.2 Les pipes

Le pipe (prononcer *paille peuh*; certains francisent le terme en *tubes*) est la méthode la plus simple quand on n'a que deux processus dont l'un lit et l'autre écrit des données. Le système alloue à un pipe un buffer auquel les processus accèdent via deux descripteurs de fichiers : l'un pour lire et l'autre pour écrire.

7.2.1 Les pipes dans la ligne de commande

Les pipes sont largement utilisés dans les lignes de commandes sous Unix, car les outils originels étaient le plus souvent conçus comme des filtres, produisant

des données sur la sortie standard, à partir de données lues sur l'entrée standard. On peut combiner ces commandes au niveau du shell avec le symbole |, de façon à connecter avec un pipe, la sortie du processus nommé à gauche de la barre avec l'entrée du processus nommé à droite.

Une manière d'avoir la liste des utilisateurs présent sur un ordinateur, triée par ordre alphabétique :

```
$ who | awk '{print $1}' | sort | uniq
```

La commande `who` produit une liste des utilisateurs actifs avec des informations détaillées sur chacun d'eux, la commande `awk` est utilisée pour ne conserver que les noms, la commande `sort` les trie par ordre alphabétique et la commande `uniq` supprime les doublons.

7.2.2 Le pipe dans les programmes C : l'appel système

L'appel système `pipe` permet de fabriquer un pipe dans un processus ; on lui passe comme argument un tableau d'entiers, et il renvoie dans les deux premières cases du tableau les deux numéros de descripteurs qui permettent d'y lire et d'y écrire.

Le programme suivant fabrique un pipe, puis y lit et y écrit des données (sans intérêt).

```
/* vb-pipe.c

 Fabriquer un pipe, y ecrire et y lire des donnees.
*/
# include "sys.h"

int
main(){
  int fd[2];
  int t;
  char buf[1024];

  t = pipe(fd);
  assert(t >= 0);

  t = write(fd[1], "foo", 3);
  assert(t == 3);

  t = write(fd[1], " et bar\n", 8);
  assert(t == 8);

  t = read(fd[0], buf, sizeof buf);
  assert(t >= 0);

  buf[t] = 0;
  printf("lu %d octets : %s\n", t, buf);

  return 0;
}
```

Les descripteurs de fichiers qui correspondent aux pipes se comportent normalement vis à vis de la plupart des appels systèmes. On peut notamment les fermer avec un `close`.

7.2.3 Les pipes dans la librairie d'entrées sorties standard

Il existe une fonction de la librairie d'entrées sorties standard qui permet d'utiliser les pipes relativement simplement : il s'agit de `popen`, qui fabrique un pipe et lance (via fork et exec) un processus qui lira ou écrira dans le pipe. Consulter la page de manuel de la fonction.

7.2.4 Limitations des pipes

Le principal inconvénient des pipes est que les processus qui l'utilisent pour communiquer doivent avoir un ancêtre commun qui a créé le pipe, de manière qu'ils héritent du descripteur de fichier déja ouvert. Les Fifos présentés dans le chapitre sur les systèmes de fichiers ou les *pipe nommés* sont une façon d'éviter cette limitation : le pipe est créé indépendamment des processus ; il dispose d'un *nom* dans le système de fichier (un nom de fichier) ; cela permet à n'importe quels processus d'y accéder, même s'ils ne sont pas apparentés.

Le pipe entre deux processus est fondamentalement un canal de communication mono-directionnel : l'un y écrit et l'autre y lit ; si les deux processus utilisent un pipe dans les deux sens, pour y lire *et* y écrire des données, ils risquent en fait de relire des données qu'ils avaient écrites à leur interlocuteur. Quand on veut faire dialoguer deux processus, le plus simple est d'ouvrir deux pipes, un pour chaque sens de communication. Cela complique sérieusement les affaires.

Le pipe ne permet de faire communiquer que des processus qui tournent sur le même ordinateur. Pour les communications entre processus qui se trouvent sur des ordinateurs distants, même reliés par le réseau, il est nécessaire d'utiliser un autre mécanisme. (L'interface la plus répandue à cet effet s'appelle les *sockets* et sera traitée dans le cours de réseau.)

7.3 Les signaux

On appelle évènements *asynchrones* les évènements qui se produisent sans qu'il soit possible de prédire à quel moment ou dans quelle tranche de temps ils vont se produire. Un exemple est la frappe des touches CTRL et C (qui interrompt un processus) sur le clavier.

Il existe des mécanismes qui permettent de signifier ces évènements asynchrones aux processus, qu'on désigne habituellement sous le nom de *signaux*.

7.3.1 Présentation générale

Il existe une liste d'une trentaine de types de signaux, qui indiquent chacun une catégorie d'évènements. Un processus peut choisir le traitement par défaut (en général : le processus s'arrête avec un compte-rendu supérieur à 128 et le shell imprime un message d'erreur) ; on peut aussi demander que le signal soit ignoré, ou bien spécifier le nom d'une fonction qui sera appelée quand le signal sera reçu par le processus.

On trouve une description complète de tous les signaux que peut recevoir un processus dans la page de manuel `signal(7)`. Les plus importants sont décrits ici :

- `KILL` est le signal qu'un processus ne peut ignorer ni traiter d'une façon spéciale. C'est la garantie qu'on pourra venir à bout de tous les processus, quoiqu'ils fassent. Elle n'est à utiliser qu'en dernier ressort.
- `HUP` était le signal envoyé à un processus pour lui signifier que la ligne téléphonique utilisée par l'utilisateur pour se connecter avait été coupée ; les processus lancés depuis une fenêtre terminal le reçoivent quand la fenêtre est fermée. Actuellement on l'utilise souvent pour demander à un démon de relire ses fichiers de configurations. C'est aussi une manière énergique de demander à un processus de s'arrêter.
- `TERM` est le signal utilisé pour demander poliment à un processus de terminer. En le recevant, la plupart des processus fermeront proprement les fichiers qu'ils ont ouvert avant de s'arrêter.
- `INT` est envoyé au processus lancé en premier-plan quand on tape un CTRL C. Il existe une variante un peu plus énergique appelée `QUIT` qu'on provoque avec le CTRL \.
- `FPE`, `MEM` et `BUS` et `SEGV` sont le plus souvent provoqués par des erreurs dans les programmes : le premier (*Floating Point Exception*), signale en général une division par 0, les autres une erreur d'adressage.
- `STOP` et `CONT` sont utilisés pour stopper et relancer les processus, comme avec les touches CTRL Z et les commandes `fg` et `bg`.

7.3.2 La commande kill : les signaux depuis le shell

La commande `kill` sert à envoyer explicitement un signal à un (ou plusieurs) processus, identifiés par leur PID ou leur numéro de job. Un exemple simple :

```
$ sleep 10000 &
[3] 8736
$ kill -TERM 8736
$
[3]-  Terminated              sleep 10000
```

La commande `top` permet, avec la lettre 'K' (comme *kill*) d'envoyer un signal à un processus.

On peut attraper (*to catch*) les signaux dans un script shell avec la commande `trap` quand on fait des scripts complexes, par exemple pour détruire des fichiers temporaires avant de sortir.

7.3.3 Les signaux depuis le C

Pour envoyer les signaux depuis un programme C, il existe un appel système `kill`, qui prend en argument un numéro de processus et le numéro d'un signal à émettre. *Attention, les arguments sont dans l'ordre inverses de ceux de la commande, où le numéro du signal vient en premier ; ici le numéro du signal est le second argument.*

Pour se préparer à recevoir un signal, un programme C peut indiquer avec `signal` une fonction à appeler lors de l'arrivée d'un signal. Cette fonction prend le numéro du signal et (un pointeur sur) la fonction.

Voici l'exemple simple d'un programme qui ne fait rien, sinon afficher *Aie !* quand il reçoit le signal `INT` (c'est à dire en général quand on tape sur CTRL C).

```
/* ve-aie.tex

Imprime quelque chose quand un signal arrive
*/
# include "sys.h"

void
aie(int stupide){
  printf("Aie !\n");
}

# define ever (;;)

int
main(){
  signal(SIGQUIT, SIG_IGN);      // ignorer
  signal(SIGINT, aie);           // traiter
  for ever
    sleep(1);
  return 0;
}
```

7.4 La communication entre applications X11

Un cas particulier de communication entre les processus est celle qui peut avoir lieu entre des clients d'un même serveur X11. Cette communication est nécessaire pour permettre le couper-coller entre diverses applications sur le même piste de travail.

La communication entre les clients et le serveur utilise en général les mécanismes du réseau. Ceci permet de faire tourner les applications sur n'importe quel ordinateur mais présente l'inconvénient que deux clients peuvent ne rien partager d'autre que leur connexion au serveur : ils ne tournent pas nécessairement sur le même ordinateur, n'ont pas nécessairement accès aux mêmes fichiers ; on pourrait même imaginer (je n'ai jamais rencontré le cas) qu'ils utilisent des communications réseau incompatibles entre elles.

La communication entre les clients peut se faire à travers le serveur X11, avec un protocole appelé ICCCM (comme *Inter Client Communication Convention Manual*). Le protocole est lourd, et rendu encore plus complexe pour permettre la communication et le traitement des données multimédia.

La principale différence d'architecture entre les interfaces les plus courantes sous X11 : KDE et Gnome, réside précisément dans la manière dont les données sont communiquées entre les applications ; par exemple, on peut parfaitement utiliser une application KDE comme `kterminal` sous Gnome ou n'importe quel autre gestionnaire de fenêtre tant qu'on ne tente pas de lui faire partager des donnée d'une façon sophistoquée avec les autres applications X11.

Exercices

Exercice 7.1 — Quand on lance une commande qui contient un pipe, dans quel ordre les processus sont-ils créés et par quel processus ? Il est facile de répondre à cette question avec un petit programme et la commande adéquate, recopiée ici :

```
$ cat va-comm.c
/* ha-comm.c

 Comment les processus d'un pipe sont-ils créés ?
 */
# include "sys.h"

int
main(int ac, char * av[]){
  assert(ac == 2);
  fprintf(stderr, "proc %s, pid %d, parent %d\n", av[1], getpid(), getppid());
  return 0;
}
$ gcc -g -Wall ha-comm.c
$ a.out A | a.out B | a.out C
```

Notez au passage comme il est pratique de pouvoir utiliser stderr ; des données que nous aurions écrites sur stdout auraient été envoyées au processus suivant, dans le pipe.

Exercice 7.2 — Exécuter le programme de 7.2.2 permet de savoir si les données, qui ont été écrites en deux fois dans le pipe, seront relues en deux fois ou en une seule. Qu'en est-il ?

Exercice 7.3 — Que se passe-t-il si on fait un lseek sur le descripteur de fichier de lecture d'un pipe ? Et sur le descripteur d'écriture ?

Exercice 7.4 — Que se passe-t-il quand on essaye de lire dans un pipe dont le descripteur d'écriture est fermé ? Et quand on essaye d'écrire dans un pipe dont le descripteur de lecture est fermé ? Le programme suivant permet de trouver la réponse :

```
/* vd-pipeclosed.c

 Lire dans un pipe dont une extremite est fermee

 A appeler avec 'a.out 0' et 'a.out 1' suivant l'extremite a fermer
*/
# include "sys.h"

int
main(int ac, char * av[]){
  int fd[2];
  int afermer;
  static char * nom[] = { "lire", "ecrire" };
  int t;
  char dummy;
```

```
  if (ac != 2 ||
      (av[1][0] != '0' && av[1][0] != '1') ||
      av[1][1] != 0){
    fprintf(stderr, "usage: %s [0|1]\n", av[0]);
    return 1;
  }
  afermer = av[1][0] - '0';

  t = pipe(fd);
  assert(t >= 0);

  printf("On ferme le descripteur pour %s\n", nom[afermer]);
  t = close(fd[afermer]);
  assert(t >= 0);

  printf("On essaye de %s\n", nom[afermer^1]);
  if (afermer == 0)
    t = write(fd[1], "X", 1);
  else /* if (afermer == 1) */
    t = read(fd[0], &dummy, 1);

  if (t >= 0)
    printf("Ca marche pour %d octets\n", t);
  else
    perror("Ca a rate avec le message");

  return 0;
}
```

Faire tourner ce programme (avec les deux arguments 0 et 1), recopier et commenter les résultats.

Exercice 7.5 — (Important) Modifier le mini-shell du chapitre sur la création de procesus pour permettre de lancer des commandes reliées par des pipes.

Exercice 7.6 — (très facultatif) Si plusieurs processus sont en concurrence pour la lecture dans un pipe, que se passe-t-il ?

Exercice 7.7 — Indiquer trois méthodes pour arrêter le programme de 7.3.3

Exercice 7.8 — Certains signaux ne peuvent pas être interrompus ou ignorés par un processus. Lesquels ? [Pour une fois, je pense qu'il est plus simple de regarder la documentation que de faire un programme expérimental pour trouver la réponse à cette question.]

Chapitre 8

La mémoire

Dans ce chapitre, nous parlons de l'organisation de la mémoire telle qu'elle est gérée par le système d'exploitation.

Une fois le chapitre assimilé, vous serez en mesure d'évaluer la consommation mémoire des programmes et vous aurez les éléments pour choisir entre l'utilisation de structure de données en mémoire ou sur le disque, avec les coûts associés.

8.1 L'unité de gestion mémoire et la mémoire paginée

Dans un modèle simple d'ordinateur, le processeur lit et écrit des données directement dans la mémoire, comme dans le schéma 8.1.

Dans les ordinateurs réels, entre le processeur et la mémoire, il y a un dispositif : l'unité de gestion mémoire ou MMU (comme *Memory Management Unit*). Son rôle est de faire la traduction entre les adresses *virtuelles* auxquelles accède un processus et les adresses *physiques* où les données sont effectivement stockées. La MMU est donc placée entre le processeur et la mémoire (figure 8.2).

La MMU divise la mémoire en *pages*, et pour cette raison on appelle souvent la mémoire gérée par une MMU de la mémoire *paginée*. Les adresses avant leur traduction par la MMU sont appelées des adresses *virtuelles* ; celles après le traitement par la MMU, telles qu'elles sont envoyées à la mémoire, sont nommées des adresses *physiques*.

La mémoire paginée présente plusieurs avantages : elle permet au noyau du système de limiter la mémoire qu'un processus peut lire ou modifier, de faire tourner les processus dans des adresses qui sont toujours identiques, d'éviter la fragmentation de la mémoire physique, de faire tourner des processus qui seraient trop gros pour tenir dans la mémoire physique, de déporter sur le disque les parties de la mémoire qui sont peu utilisées.

Ces avantages seront décrits plus en détail après la présentation du fonctionnement d'une MMU pour un processeur de 32 bits dans la section suivante.

FIG. 8.1 – Le processeur lit et écrit des données dans la mémoire à des adresses qu'il spécifie..

FIG. 8.2 – Les adresses émises par le processeur sont traduites par la MMU avant d'être transmises à la mémoire.. En pratique, dans les processeurs actuels la MMU est intégrée au microprocesseur.

FIG. 8.3 – La MMU découpe l'adresse virtuelle en numéro de page et adresse dans la page. Le numéro de page permet de trouver l'adresse de la page dans la table des pages ; celle-ci est combinée avec l'adresse dans la page pour donner l'adresse physique.

8.1.1 Fonctionnement de la MMU

La mémoire est divisée en pages qui occupent en général 4 Ko. Quand le processeur émet une adresse (virtuelle), la MMU divise l'adresse en deux parties : les douze bits de poids faibles désignent une adresse dans la page ($4K = 2^{12}$) et les 20 bits restant représentent le numéro de la page.

La MMU dispose d'une *table des pages* qui décrit chaque page qui peut exister ; cette table des pages contient en général :

- un bit qui indique si la page est utilisable par le processus.
- des bits de permission qui indiquent si la page peut être lue et écrite. Certaines MMU permettent de noter qu'une page contient des valeurs qui peuvent être lues mais pas exécutées, d'autres MMU ne font pas la distinction entre lecture d'une donnée et lecture d'une instruction ; c'est notamment le cas de la MMU associée au processeur Intel.
- l'adresse physique où la page est stockée.
- un marqueur qui indique si la page à été utilisée.
- éventuellement un compteur qui indique combien de fois la page a été utilisée.

Chaque adresse émise par le processeur est transmise à la MMU. Celle-ci découpe les 32 bits de l'adresse en vingt bits de numéro de page et douze bits d'adresse dans la page. Elle utilise le numéro de la page pour trouver son adresse physique et y ajoute les douze bits pour obtenir une adresse physique (figure 8.3.

FIG. 8.4 – La table des pages pour un processus qui n'utilise que les 4 premiers et les 4 derniers Ko de son espace d'adressage virtuel quand on a une table des pages à deux niveaux.

Organisation de la table des pages

Chaque processus dispose de sa propre table des pages. Or la table des pages est énorme, puisqu'avec 32 bits d'adresses et des pages de 4 Ko il peut y avoir 1 Mega pages différentes. Avec une table ordinaire, toutes les entrées seraient nécessaires, ne serait-ce que pour y stocker le bit qui indique que la page n'est pas valide.

Pour diminuer la taille de la table des pages, celle-ci est organisée en table à plusieurs niveaux : en général deux niveaux sur un processeur 32 bits. Les dix bits de poids forts servent à choisir une entrée parmi 1024 dans la table des pages de niveau 1. Chaque entrée indique l'adresse d'une table de niveau 2 qui contient à son tour une table pour 1024 entrées, dans laquelle les dix bits de poids faible du numéro de page servent à choisir une entrée.

Le mécanisme est représenté sur la figure 8.4 : la table des pages de niveau 1 ne contient que deux entrées valides : 0 et 1023 ; cela signifie que seules les adresses virtuelles dont les bits de poids forts contiennent ces deux valeurs sont valides. Chaque entrée valide dans la table de niveau 1 contient l'adresse d'une page de niveau 2 qui contient à son tour 1024 entrées : les bits 12 à 21 servent à choisir une entrée qui contient l'adresse physique de la page. Dans l'exemple

de la figure 8.4, les seules adresses virtuelles valides sont celles entre 0 et 4095 (elles sont traduites dans des adresses physiques entre 0x01000000 et 0x01000fff) et celles entre 0xfffff000 et 0xffffffff (elles seront traduites dans des adresses physiques entre 0x01001000 et 0x01001fff).

Le mécanisme de table à deux niveaux permet que la table des pages d'un processus qui occupe peu de mémoire reste compacte, mais la table est néanmoins trop grosse pour être stockée dans la MMU : elle est en fait placée dans la mémoire. Quand la MMU reçoit une adresse, elle doit faire trois accès à la mémoire : l'un pour lire l'entrée de la table de niveau 1, la seconde pour lire l'entrée de la table de niveau 2, puis la troisième pour accéder à l'adresse physique. Pour accélérer les accès à la mémoire, la MMU utilise une mémoire cache appelée le *Translation Lookaside Buffer* (TLB), présenté dans la section suivante.

Le Translation Lookaside Buffer

Pour accélérer les accès à la mémoire en évitant de consulter en permanence la table des pages, la MMU dispose d'une mémoire temporaire dans laquelle elle stocke les résultats des recherches récemment effectuées, qu'on appelle le *Translation Lookaside Buffer* ou *TLB*.

Les numéros de pages récemment utilisés et leurs adresses physiques sont conservés dans la TLB. C'est seulement quand il est fait référence à une nouvelle page que la MMU doit consulter la table des pages dans la mémoire pour traduire le numéro de page en adresse physique ; l'information est ensuite conservée dans le TLB.

Le TLB est stocké dans une mémoire très rapide qui se trouve à l'intérieur de la MMU ; son temps d'accès est négligeable par rapport à celui d'un accès à la mémoire.

Lorsque le noyau lance un nouveau processus à la place de celui qui était en train de s'exécuter, il doit remplacer la table des pages de l'ancien processus par celle du nouveau : l'opération est rapide puisqu'il suffit de faire pointer le registre de table des pages vers la nouvelle table. En revanche il est également nécessaire de vider la TLB ; pour chaque référence à une nouvelle page par le nouveau processus, la MMU devra lire le contenu de la table des pages ; cela ralentira le démarrage du nouveau processus.

8.1.2 L'utilité de la mémoire paginée

Nous revenons dans cette section sur les caractéristiques qui font l'intérêt de la mémoire paginée.

Tous les processus ont le même espace d'adressage

La MMU permet au noyau de faire exécuter chaque processus dans un espace d'adressage (virtuel) qui lui est propre : de ce fait, les programmes sont préparés pour se trouver à des adresses fixes dans la mémoire. Au moment du lancement d'un processus, le noyau lui attribue de la mémoire et prépare sa table des pages pour que le code et les données, qui peuvent être n'importe où dans la mémoire physique, soient à l'adresse virtuelle convenue.

Examinons le programme suivant qui se contente d'afficher 3 fois l'adresse et la valeur d'une variable, avec un intervalle d'une seconde entre deux impressions

pour nous permettre de suivre l'exécution.

```
/*
 Imprimer l'adresse et la valeur d'une variable.
 */
# include <stdio.h>

int toto;

int
main(){
  for(toto = 0; toto < 3; toto ++){
    printf("Processus %d : la variable toto est a l'adresse 0x%x, elle vaut %d\n",
           getpid(), (int)&toto, toto);
    sleep(1);
  }
  return 0;
}
```

Quand on compile le programme et qu'on le lance, le processus résultant imprime :

```
$ gcc -g -Wall wa-adr.c
$ a.out
Processus 6027 : la variable toto est a l'adresse 0x6009dc, elle vaut 0
Processus 6027 : la variable toto est a l'adresse 0x6009dc, elle vaut 1
Processus 6027 : la variable toto est a l'adresse 0x6009dc, elle vaut 2
$
```

La variable se trouve à l'adresse `0x6009dc`, mais il s'agit là d'une adresse *virtuelle*, avant le passage dans la MMU. On peut voir que cette adresse est câblée dans le programme exécutable en examinant la table des symboles qui se trouve dans le fichier `a.out`.

```
$ nm a.out | grep toto
00000000006009dc B toto
```

La commande `nm` nous affiche le contenu de la table des symboles ; le `grep` nous permet de sélectionner la ligne qui nous intéresse. On voit que dans le programme exécutable, l'adresse de la variable est déjà fixée à `0x6009dc`.

(La table des symboles est purement informative ; on peut la retirer du fichier exécutable avec la commande `strip`.)

Lançons le programme plusieurs fois, par exemple avec

```
$ a.out & (sleep 1 ; a.out)
[3] 6033
Processus 6033 : la variable toto est a l'adresse 0x6009dc, elle vaut 0
Processus 6033 : la variable toto est a l'adresse 0x6009dc, elle vaut 1
Processus 6036 : la variable toto est a l'adresse 0x6009dc, elle vaut 0
Processus 6033 : la variable toto est a l'adresse 0x6009dc, elle vaut 2
Processus 6036 : la variable toto est a l'adresse 0x6009dc, elle vaut 1
Processus 6036 : la variable toto est a l'adresse 0x6009dc, elle vaut 2
[3]+  Done                    a.out
```

On voit que les deux processus contiennent leur propre variable `toto`, avec sa propre valeur indépendante, mais que tous les deux voient leur exemplaire de la variable `toto` à la même adresse : c'est qu'il s'agit d'une adresse *virtuelle* que la table des pages du processus traduit dans une adresse physique différente.

Lorsqu'un processus tente d'accéder à un mot mémoire qui n'apparaît pas dans la table des pages, le noyau du système d'exploitation en reçoit notification de la MMU et envoie au processus un *signal Memory Fault* ou *Segmentation Violation* qui, par défaut, provoquent l'arrêt du processus.

Contrôle de la mémoire des processus

Seul le noyau du système d'exploitation peut modifier la table des pages d'un processus. Cela lui permet de contrôler finement les endroits de la mémoire auxquels un processus peut accéder, et d'éviter qu'un processus consulte ou modifie la mémoire d'un autre processus ou du noyau de manière incontrôlée.

Fragmentation (facultatif)

On pourrait avoir les deux avantages précédents avec un mécanisme beaucoup plus simple : le noyau attribuerait à un processus une zone de mémoire contiguë ; un registre serait ajouté à chaque adresse et un compteur permettrait de vérifier que le processus ne déborde pas de la mémoire qui le noyau lui a attribué. On parle alors de mémoire *segmentée*.

Le principal inconvénient de cette mémoire segmentée est qu'à mesure que les processus sont créés et détruit, la mémoire libre est divisée en petites zones qui ne sont pas nécessairement contiguës. Il peut arriver un moment où cela empêche de créer un processus : il y a suffisamment de mémoire libre, mais elle est divisée en zones indépendantes alors qu'un processus doit utiliser une zone contiguë. On appelle ce phénomène la *fragmentation*.

La mémoire paginée permet d'éviter la fragmentation, puisque les pages n'ont pas besoin d'être contiguës.

Changement de la taille de la mémoire d'un processus

En cours d'exécution, la taille mémoire d'un processus peut augmenter, principalement dans la pile et dans les données. Lorsqu'on appelle la fonction `malloc` (ou l'opérateur `new` en C++), la fonction demande l'allocation de plus de mémoire au noyau (avec l'appel système `brk` ou `sbrk`).

La mémoire virtuelle paginée permet de faire grossir la mémoire attribuée au processus sans le déplacer dans la mémoire physique. Du moment qu'il y a de la mémoire libre, le noyau peut l'attribuer au processus ; lui permettre d'y accéder n'implique qu'une mise à jour de la table des pages du processus.

Plus de mémoire physique que de mémoire paginée

Nous n'avons rien dit sur le nombre de bits nécessaires pour représenter l'adresse physique d'une page dans la table des pages (revoir par exemple les figures 8.3 et 8.4). S'il est plus grand que le nombre de bits du numéro de page dans l'adresse virtuelle, cela permet à un ordinateur (mais pas à un processus) de disposer d'une quantité de mémoire supérieure à celle à laquelle le processeur lui-même peut accéder (4 Go sur un processeur 32 bits).

Par exemple, si l'adresse physique des pages dans la table de niveau 2 est stockée sur 30 bits, cela permettra d'équiper un processeur 32 bits avec 4 To de mémoire sans modifier le processeur. Les adresses virtuelles utiliseront toujours 32 bits et un processus donné ne pourra accéder qu'à 4 Go de mémoire, mais les adresses physiques occuperont 42 bits ce qui permettra d'utiliser les 4 To de mémoire pour y conserver plusieurs processus.

8.2 La mémoire virtuelle

Sur les ordinateurs actuels, la mémoire paginée est presque toujours associée à un autre mécanisme qu'on appelle la mémoire *virtuelle*. Pour qu'un processus puisse s'exécuter, toutes ses pages n'ont pas besoin d'être présentes dans la mémoire au même instant ; les pages qui ne sont pas utiles à un moment donné peuvent être recopiées sur le disque.

Chaque entrée dans la table des pages contient un bit supplémentaire qui indique si la page est présente en mémoire. Lorsque le processus tente d'accéder à une page qui n'est pas présente (on parle de *défaut de page* ou *page fault*), la MMU le signale au noyau du système. Celui-ci recopie la page depuis le disque dans la mémoire puis relance l'instruction qui a échoué.

8.2.1 Le swap

Sur les systèmes Unix et Linux, il existe une zone du disque qui est spécialement utilisée pour recopier les pages de la mémoire : on l'appelle le `swap`.

Sous Linux, le swap est une (ou plusieurs) partitions spécifiques qui apparaissent dans le fichier `/etc/fstab`. On peut les voir avec

```
$ grep swap /etc/fstab
/dev/hda5 swap swap defaults 0 0
```

La partition étendue numéro 5 contient le swap. Elle a été créée lors de l'installation du système. Les installations Linux standards lui attribuent par défaut entre 2 et 4 fois la taille de la mémoire physique. (D'autres systèmes Unix permettent d'utiliser un fichier ordinaire ou les zones inutilisées des systèmes de fichiers montés en guise de swap.)

Le swap ne contient pas une arborescence de fichiers comme les autres systèmes de fichiers, aussi la partition n'est pas montée avec la commande `mount`. Il existe une commande d'administration `swapon` dans le répertoire `/sbin` pour indiquer au noyau qu'il peut utiliser la zone pour copier les pages ; la commande est utilisée dans les scripts d'initialisation lancés par `init` au démarrage du système. Il existe une commande symétrique `/sbin/swapoff` pour interdire l'utilisation d'une partition.

8.2.2 Les algorithmes de remplacement de page

Lorsqu'un processus à besoin d'une page qui est sur le swap, le noyau doit la copier dans la mémoire physique : cela nécessite de conserver des pages libres ou de libérer des pages. Pour ce faire, le noyau doit choisir quelles pages présentes dans la mémoire doivent être libérées, ou recopiées sur le disque si elles ont été

modifiées (on dit souvent qu'une page modifiée depuis qu'elle a été lue depuis le swap est *dirty* (sale)).

Nous présentons ici les algorithmes les plus couramment utilisés pour choisir les pages à modifier. Rappelons qu'il y a une différence de six ordres de grandeur entre le temps d'exécution d'une instruction et celui de la lecture d'un secteur sur le disque : il est donc utile de passer du temps en calcul pour tenter de minimiser le nombre de pages lues et écrites sur le swap.

Optimal : l'algorithme optimal consiste à choisir de se débarrasser de la page qui sera utilisée le plus loin possible dans le futur. Cet algorithme n'est bien évidemment pas réalisable, puisqu'il faudrait que le noyau puisse prévoir le comportement des processus pour le mettre en œuvre, mais il est utile pour savoir quel est l'objectif dont nous souhaitons nous rapprocher.

LRU comme *Least Recently Used* : la page la moins récemment utilisée; il faut pour la mettre en oeuvre que la MMU permette d'associer avec chaque page l'instant à laquelle elle a été utilisée, ce qui n'est en général pas possible : la plupart des MMU disposent seulement d'un bit qui indique si la page a été utilisée, sans qu'on puisse préciser à quel instant.

NRU comme *Not Recently Used* : une page pas récemment utilisée; cela consiste à choisir une page qui n'a pas été utilisée récemment. Régulièrement, le noyau met à 0 le bit d'utilisation des pages dans la MMU puis au bout d'un intervalle de temps fixé, le noyau examine l'état de ce bit et marque les pages pour lesquelles il est à 1 comme utilisées récemment. Quand il faut libérer une page, le noyau la choisit parmi celles qui n'ont pas été marquées.

Les autres : il existe de nombreuses variantes des algorithmes LRU ou NRU que nous ne détaillons pas ici.

8.2.3 Mémoire virtuelle, commandes, programmes

Cette section contient des exemples de commandes et de programmes qui permettent d'examiner le fonctionnement de la mémoire virtuelle.

La commande `top`

Dans la page affichée par la commande `top`, la taille de chaque processus est indiquée par deux colonnes intitulée `VIRT` et `RES`. La colonne `VIRT` contient la taille totale de la mémoire qu'utiliserait le processus s'il était complètement présent dans la mémoire. La colonne `RES` indique elle la quantité de mémoire effectivement utilisée (ce sont les pages *résidentes*). Sur l'ordinateur où j'écris ce texte, `top` m'affiche :

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 5797 jm        15   0  566m 108m  24m S    5  5.4   1:43.24 firefox-bin
 6206 jm        15   0 18952 1320  972 R    0  0.1   0:00.12 top
 ...
```

Ceci indique que `firefox` utilise plus de 550 Mo de mémoire, mais qu'à peine plus de 108 Mo résident actuellement dans la mémoire. De même, la commande top utilise près de 20 Mo, mais seuls 1,3 Mo sont présents.

Dans le bloc de lignes qu'il affiche en haut de la fenêtre, `top` indique également le bilan global de l'utilisation de la mémoire vive et du swap. Par exemple, sur ma machine avec deux Go de mémoire et quatre Go de swap, top affiche au moment où j'écris :

```
Mem:   2063296k total,   468576k used,  1594720k free,    17996k buffers
Swap:  4095588k total,        0k used,  4095588k free,   222852k cached
```

La première ligne décrit l'utilisation de la mémoire. Les `buffers` sont la mémoire qui contient des données en attente d'entrée-sortie vers un périphérique.

8.2.4 Un programme pour faire travailler la mémoire virtuelle

On peut facilement fabriquer un programme pour faire travailler la mémoire virtuelle. Par exemple le programme suivant utilise un Go de mémoire sans modèle d'utilisation prévisible :

```
/* ie-pagefault.c
 Un programme pour faire des defaults de pages
*/
# include <stdio.h>

enum {
  Nfois = 1000 * 1000 * 100,
  Size = 1024 * 1024 * 1024,
};

char tab[Size];

int
main(){
  int i;

  for(i = 0; i < Nfois; i++)
    tab[rand() % Size] = 0;
  return 0;
}
```

Le tableau `tab` occupe un Go de mémoire. 100 millions de fois, le programme accède à une case du tableau choisie au (pseudo) hasard, grâce à la fonction `rand()`. On peut lancer le programme une fois, avec `time` pour mesurer son temps d'exécution :

```
$ time a.out

real    0m7.937s
user    0m7.120s
sys     0m0.796s
```

On constate qu'il faut environ 8 secondes pour faire tourner le programme, dont plus de 7 passées à exécuter le code du programme. On peut obtenir des informations complémentaires en utilisant le *programme* `/usr/bin/time` au lieu de la *commande* `time` qui est interne à bash :

```
$ /usr/bin/time a.out
7.12user 0.76system 0:07.89elapsed 99%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+262265minor)pagefaults 0swaps
```

Ce qui nous intéresse ici est le nombre de défauts de pages (affichés comme un nombre de défauts de pages majeurs et mineurs) ; les défauts mineurs correspondent ici au premier accès aux pages de mémoire qui contiennent le tableau, que le système n'a pas attribué au lancement : il a attendu que le processus l'utilise ; il n'y a pas de défauts majeurs qui correspondraient à une référence à une page qui aurait été recopiée sur le swap.

Quand on lance le programme deux fois, on pourrait penser qu'il va falloir deux fois plus de temps. En fait, les processus commencent à déborder de la mémoire vive et le swap doit être utilisé pour sauver les pages. Les résultats sur la même machine que précédemment sont :

```
$ time a.out & time a.out
[1] 6114

real    0m29.438s
user    0m8.237s
sys     0m1.376s
$
real    1m5.583s
user    0m8.313s
sys     0m1.724s

[1]+  Done                    time a.out
$
```

Le temps de calcul n'a pas été beaucoup modifié : pour les deux programmes il tourne toujours autour de 8 secondes. Le temps système augmente significativement : il tourne maintenant autour de une seconde et demie ; c'est le temps nécessaire lors de chaque défaut de page pour que le noyau stoppe le processus en attendant que la page manquante soit rechargée dans la mémoire. La majeure partie du temps (30 secondes pour l'un, plus d'une minute pour l'autre !) est passée à attendre que les pages modifiées soient échangées entre le swap et la mémoire.

Tout le système commence à aller plus lentement. Même l'écho de la souris sur l'écran n'est plus mis à jour de façon régulière. On se retrouve à taper des caractères en aveugle en attendant que leur écho apparaisse à l'écran. Lorsqu'on fait apparaître une zone d'une fenêtre à l'écran, le temps nécessaire pour que le processus qui la contrôle en dessine le contenu devient notable.

Il est instructif de comparer la ligne de résumé qu'affiche `top` suivant le nombre de processus qui exécutent le programme. En l'absence d'exécution du programme, la machine dispose de 2 Go de mémoire vive, dont 750 Mo environ sont utilisés, et de 4 Go de swap tous libres.

```
Mem:   2063296k total,   743340k used,  1319956k free,   149600k buffers
Swap:  4095588k total,        0k used,  4095588k free,   312924k cached
```

Quand on lance le programme une fois , la quantité de mémoire utilisée augmente mais le swap est toujours inutilisé.

```
Mem:   2063296k total,  1794804k used,   268492k free,   149692k buffers
Swap:  4095588k total,        0k used,  4095588k free,   313044k cached
```

Quand on le lance deux fois, la mémoire vive ne suffit plus et il faut commencer à utiliser le swap : les pages des deux tableaux `tab` des deux processus tentent d'occuper toute la mémoire.

```
Mem:   2063296k total,  2046748k used,    16548k free,      208k buffers
Swap:  4095588k total,   483344k used,  3612244k free,    11016k cached
```

Si on lance le programme trois ou quatre fois, l'ordinateur devient à peu près inutilisable. On se retrouve dans une situation où le noyau passe son temps à faire entrer et sortir des pages de la mémoire. Comme le noyau envoie sur le swap les pages de tous les processus, tous sont affectés, et non pas seulement les gros programmes. On appelle ce phénomène du *trashing*.

Pour l'utilisateur, cela se manifeste sous la forme d'une activité intense du disque (souvent visible par une LED qui reste allumée en permanence) et une quasi inactivité du CPU : cela signifie soit qu'un processus utilise trop de mémoire — en général à cause d'une erreur — soit que la mémoire de la machine est insuffisante pour un fonctionnement normal.

Copy On Write et fork

Le mécanisme de mémoire paginée permet d'accélérer les opérations quand des processus partagent de la mémoire : au moment d'un `fork`, les processus parent et enfant doivent disposer chacun d'une copie de la mémoire du processus du parent. Avec la mémoire paginée, il est possible de ne pas effectuer la recopie : il suffit de recopier la table des pages du processus parent dans celle du processus enfant, et tous les deux accèdent alors à la même mémoire physique.

Cependant, quand un des processus souhaite modifier une donnée dans sa mémoire, cette modification ne doit pas affecter la mémoire de l'autre processus. Pour cela, le système utilise une technique appelée *copy on write* (parfois abrégé en *COW*) : les pages partagées entre les deux processus sont marquées comme interdites en écriture ; quand un des processus tente de modifier la page, la MMU signale l'erreur au noyau ; celui-ci recopie alors la page et modifie la table des pages d'un des processus pour utiliser la copie ; maintenant, chacun des processus peut modifier les données contenues dans sa copie de la page sans affecter les données de l'autre.

Le noyau de cette manière n'a besoin que de recopier les pages modifiées. Dans le cas fréquent où le `fork` est rapidement suivi d'un `exec` par l'un des processus, il y a peu de pages modifiées et ceci accélère les opérations.

8.3 L'over-commitment de la mémoire de Linux

A faire ?

8.4 Le format a.out statique

text, data, bss, heap -> <- stack.
Plus tellement utilisé à cause des bibliothèques dynamiques.

8.5 La mémoire partagée

Les bibliothèques dynamiques.
La colonne SHR de top.

Exercices

Exercice 8.1 — Si un processus utilise 1 Go de mémoire contiguë, en supposant que chaque table de niveau 1 ou 2 utilise 4 Ko, quelle est l'ordre de grandeur de la mémoire nécessaire pour représenter sa table des pages.

Exercice 8.2 — Trouver et décrire l'algorithme utilisé par le noyau Linux pour choisir les pages à remplacer. (Ne pas oublier de citer vos sources dans la réponse.)

Exercice 8.3 — (Facile) Quelles options faut il donner à la commande `ps` pour obtenir la taille virtuelle et la taille résidente des processus qu'il affiche.

Exercice 8.4 — Modifier l'ordre dans lequel le programme qui sature la mémoire accède aux cases du tableau en remplaçant la ligne

```
tab[rand() % Size] = 0;
```

par

```
tab[i % Size] = 0;
```

Maintenant, au lieu d'accéder aux cases au hasard, le programme y accède en séquence. Comparer les performances de ce programme avec celles du programme original pour l'exécution d'une, deux ou trois instances du programme. Conclusion ? (Il peut être nécessaire de modifier la taille du tableau `tab` pour l'adapter à la taille de la mémoire de votre ordinateur.)

Chapitre 9

Le scheduler

Une partie importante du noyau du système d'exploitation est celle qui choisit le processus qui va pouvoir s'exécuter. On appelle cette partie le *scheduler* (en français : *l'ordonnanceur*).

9.1 L'état d'un processus

Un processus peut se trouver principalement dans trois états : soit il est prêt à s'exécuter, soit il en train de s'exécuter, soit il est *bloqué*, en attente d'un évènement avant de pouvoir continuer son exécution, comme évoqué sur la figure 9.1.

Au départ, un processus qui vient d'être créé est prêt à s'exécuter : il est dans l'état *Ready* (ou *Runnable*). À un moment donné, le scheduler va choisir ce processus pour lui permettre de s'exécuter (on dit parfois que le processus est *élu*) : le processus passe dans l'état *Running*. Pendant son exécution, le processus effectuer une opération qui va bloquer son exécution (il s'agit en général d'une demande d'entrée sortie) : le processus rentre dans l'état *Blocked*. Un autre cas se présente quand le processus calcule sans arrêt et que le scheduler reprend la main et choisit d'élire un autre processus (on dit que le processus est *préempté*) : le processus revient alors dans l'état *Ready*. Finalement, un processus en cours d'exécution peut se terminer, par exemple avec l'appel système `exit` : le système ne peut pas supprimer complètement le processus parce qu'il faut attendre que

FIG. 9.1 – Le graphe d'état d'un processus.

le processus parent fasse le `wait` qui lui permettra de recevoir la valeur avec laquelle le processus s'est terminer : sous Linux, on appelle cet état particulier l'état *Zombie*.

On peut noter que sous Linux la création ne concerne que `init`, le premier processus créé par le noyau : tous les autres processus sont créés avec l'appel système `fork` : pour effectuer cet appel système, le processus parent doit être *Running*. Le résultat d'un `fork` est un processus enfant qui est une copie du processus parent : l'un des processus est dans l'état *Running* et l'autre dans l'état *Ready*.

La commande `top`, qui affiche en continu l'état des processus, indique leur état dans la colonne intitulée `S`. La commande `ps l` l'indique dans la colonne `STAT`.

9.2 Le rôle du scheduler

Le choix du prochain processus *Running* est du ressort d'une partie du noyau qu'on appelle le *scheduler* : c'est une partie importante parce qu'un scheduler mal conçu peut rendre l'usage d'un système d'exploitation très désagréable. De plus le scheduler peut avoir à tourner fréquemment : il donc important qu'il travaille rapidement.

Les principaux buts d'un scheduler sont :

- en tout premier lieu de minimiser l'impression de lenteur que peut ressentir l'utilisateur ; pour cette raison, un scheduler va tenter de favoriser les processus interactifs (dits *IO-bound* : limités par les entrées-sorties) par rapport aux processus qui ne font que du calcul (dits *CPU-bound* : limités par le temps de calcul disponible).
- d'éviter les famines : il faut que tous processus puissent s'exécuter à un moment donné.
- de favoriser l'équité : tous les processus équivalents doivent pouvoir s'exécuter de façon équivalente.
- pour les systèmes temps réel, le scheduler doit, autant que faire se peut, respecter les contraintes de temps déclarées par les processus.

9.2.1 Préemption et time-slice

En général, les scheduler attribuent aux processus une tranche de temps d'exécution (une *time-slice* ou *time quantum*). Si pendant cette tranche le processus passe dans l'état bloqué, le scheduler active un autre processus. En revanche, si au bout de la tranche de temps le processus est encore en cours d'exécution, le scheduler reprend la main pour éventuellement élire un autre processus. Ce mécanisme où le scheduler reprend la main s'appelle la *préemption* ; un scheduler qui la met en oeuvre est un scheduler *préemptif*.

Le choix de la taille de la tranche de temps obéit à deux impératifs contradictoires : pour avoir un système réactif, il faut que la tranche de temps soit la plus petite possible. D'un autre coté, le changement de processus actif est une opération coûteuse en temps et on cherche donc à la minimiser. En pratique, les tranches de temps attribuées par les schedulers varient entre 1 et 100 millisecondes.

Sur le système Linux, depuis la version 2.6 du noyau, le scheduler ajuste la taille de la tranche de temps en fonction du nombre de processus dans l'état *Ready*.

9.2.2 Ordinateurs parallèles

Sur les ordinateurs avec plusieurs microprocesseurs ou dont les microprocesseurs peuvent exécuter plusieurs programmes en même temps, on peut avoir plusieurs processus en même temps dans l'état *Running*.

Le travail du scheduler est compliqué par deux nouveaux impératifs contradictoires : d'une part, il est souhaitable d'équilibrer la charge sur les microprocesseurs pour qu'ils soient tous actifs ; d'autre part, la *migration* d'un processus d'un processeur vers un autre est une opération coûteuse, qu'il convient donc de minimiser.

9.2.3 Le temps réel

Les systèmes temps réels *durs* sont ceux, rappelons le, dans lesquels une réponse qui arrive trop tard ne vaut pas mieux qu'une réponse fausse. Un exemple classique est celui de la prévision météorologique : s'il faut calculer pendant un mois pour prévoir le temps qu'il fera la semaine prochaine, le calcul est sans intérêt.

Les systèmes temps réels *softs* sont ceux où la qualité du service se dégrade à mesure que les réponses arrivent trop tard. Un exemple est celui du décodage d'une vidéo compressée : la qualité de la vidéo sera inversement proportionnelle au nombre d'images qui n'auront pas pu être traitées à temps pour être affichées.

Ces systèmes temps réels ont des impératifs particuliers en terme de scheduling.

9.2.4 Quelques algorithmes de scheduling

Pour répondre aux contraintes, il existe de nombreux algorithmes de scheduling : en mai 2008, la page de WikiPédia contient 37 liens vers la description de 37 algorithmes différents. Ne sont présentés ici que les algorithmes les plus importants.

FIFO

Une queue *FIFO* (comme *First In, First Out* : premier entré premier sorti) présente l'avantage de la simplicité. Les processus sont élus dans l'ordre où ils sont passés dans l'état *Ready*.

Si un processus calcule sans arrêt, on aura une famine : une fois qu'il sera élu, il en rendra pas le processeur et tous les autres processus seront bloqués.

Round-Robin

Le *Round-Robin* (en français on peut dire le *tourniquet*) consiste à ajouter la préemption au *FIFO* : chaque processus de la file d'attente reçoit l'usage du CPU pour une tranche de temps. A l'expiration de la tranche, si le processus calcule encore il est replacé à la fin de la liste des processus *Ready*.

L'algorithme n'est pas beaucoup plus compliqué que le FIFO et il garantit qu'il n'y aura pas de famine.

Shortest Job First

L'algorithme *Shortest Job First* consiste à élire de préférence le processus qui va utiliser le CPU pendant le moins de temps. On peut prouver que cela permet de minimiser temps moyen de terminaison d'un processus.

En pratique, l'algorithme n'est pas réalisable parce qu'il n'y a pas moyen pour le scheduler de prévoir le temps de CPU dont va avoir besoin un processus (sauf sur certains systèmes temps réels où les processus doivent annoncer leurs besoins). En revanche, on peut s'en approcher en faisant calculer au scheduler une *supposition* du temps CPU dont un processus aura besoin fondé sur son comportement passé.

Round-Robin par niveau de priorité

La combinaison de *Round-Robin* et de *Shortest Job First* conduit à un algorithme qui ressemble à celui couramment utilisé sous Unix : le système affecte à chaque processus un niveau de priorité qui est ajusté en fonction de son comportement passé de manière à favoriser les processus interactif. De temps en temps (à chaque seconde), le système recalcule la priorité de tous les processus.

Pour chaque niveau de priorité, le système conserve une file d'attente. Le scheduler élit les processus en Round Robin dans la file d'attente des processus les plus prioritaires.

Le scheduler de Linux

Le scheduler scheduler de Linux est une variante du scheduler Unix.

Il existe une catégorie de tâches, dites *temps réel*, qui sont toujours élues en premier. Elles peuvent être soit FIFO soit *Round Robin*, suivant que le scheduler utilise ou pas la préemption.

Pour toutes les autres tâches le système utilise le *round-robin* dans le niveau de priorité le plus élevé.

Depuis la version 2.6 du noyau, le scheduler Linux présente deux particularités : d'une part il ajuste le *time-slice* en fonction du nombre de processus dans l'état *Ready*; d'autre part, le temps de calcul utilisé par le scheduler est indépendant du nombre de processus ready (on dit que c'est scheduler en $O(1)$).

Exercice 9.1 — (exercice type corrigé) Quatre processus ont besoin de calculer respectivement pendant 30, 10, 40 et 10 millisecondes. Quel sera le temps moyen de terminaison d'un processus avec les trois premiers algorithmes de scheduling présentés, en utilisant un time-slice de 5 millisecondes.

FIFO la figure 9.2 montre le scheduling des processus quand on utilise l'algorithme FIFO. Chaque processus est représenté par une ligne qui est noircie quand le processus est actif. Les processus se terminent respectivement à $t = 30$, $t = 40$, $t = 80$ et $t = 90$. Le temps moyen de terminaison d'un processus est donc de $(30 + 40 + 80 + 90)/4 = 60$ms.

FIG. 9.2 – Le scheduling des processus avec l'algorithme FIFO.

FIG. 9.3 – Le scheduling des processus avec l'algorithme Shortest Job First.

FIG. 9.4 – Le scheduling des processus avec l'algorithme Round Robin

Shortest Job First Comme indiqué dans la figure 9.3, avec *Shortest Job First*, les processus se terminent à $t = 10$, $t = 20$, $t = 50$ et $t = 90$ donc le temps moyen est de $(10 + 20 + 60 + 90)/4 = 45$ms.

Round Robin Avec Round Robin et un *time-slice* de 5 ms, les processus sont schedulés comme sur la figure 9.4. Les temps de terminaison sont $t = 30$, $t = 40$, $t = 75$ et $t = 90$ donc le temps moyen est de $(30 + 40 + 75 + 90)/4 = 58.5$ms.

9.3 La priorité

Pour décider de la file d'attente dans laquelle seront placés les processus qui ne sont pas étiquetés comme temps réel, les noyaux Unix calculent leur *priorité*. Il s'agit d'un nombre entier. Ce nombre est mis à jour à intervalles réguliers.

Quand un processus passe dans l'état bloqué avant l'expiration du temps CPU qui lui a été attribué, la priorité du processus est augmentée ; en revanche, s'il calcule pendant tout son quantum, sa priorité est diminuée : cela garantit un avantage aux processus interactifs.

La priorité d'un processus apparaît dans la colonne PR affichée par la commande `top`.

9.3.1 La commande `nice`

L'utilisateur peut ajouter une composante statique aux ingrédients utilisés par le scheduler pour calculer la priorité des processus : on l'appelle usuellement le *nice*. Plus la valeur est forte, moins le processus est prioritaire.

Une fois que le processus est lancé, son *nice* peut être modifié avec la commande `renice`.

Un processus ordinaire a un `nice` de 0. Un utilisateur ordinaire peut augmenter cette valeur jusqu'à 19 pour faire baisser la priorité d'un de ses processus. L'administrateur système peut l'abaisser jusqu'à -20, pour augmenter la priorité.

La valeur du *nice* d'un processus apparaît dans la colonne `NI` affichée par `top`. On peut changer la priorité d'un processus sous `top` avec la touche `'r'` ; sinon, il existe aussi des commandes `nice` et `renice`.

9.4 Le load average

Le *load average* est le nombre moyen de processus prêts ou actifs sur un intervalle de temps. En première approximation, un *load average* de 1 indique que le processeur est occupé à temps complet (mais voir le contre exemple plus loin).

Trois valeurs du *load average* sont affichées en haut à droite de la page affichée par `top`. Il existe aussi des interfaces graphiques, comme la commande `xload`, pour voir l'évolution du load average.

Comme le *load average* n'est qu'une moyenne, sa valeur est à prendre avec prudence. Si on imagine deux processus qui tournent chacun pendant une demie seconde, le *load average* peut être de 1 s'ils entrent dans l'état *Ready* à tour de rôle ou de 1.5 s'il y entrent en même temps, comme indiqué sur la figure 9.5

FIG. 9.5 – Deux processus sont actifs la moitié du temps. S'ils entrent dans l'état *Ready* de façon synchronisée, comme (a), le load average sera de 1.5 ; En revanche, s'ils entrent dans l'état *Ready* à tour de rôle comme dans (b), le load average sera de 1.

Exercices

Exercice 9.2 — Après un `fork`, sera-t-il préférable de mettre le parent ou l'enfant dans l'état *Ready* ? Discuter.

Exercice 9.3 — Pour mettre un processus dans l'état *Zombie*, il suffit que le processus parent crée un enfant qui sort immédiatement puis attende sans faire de `wait`. Écrire un programme pour ce faire et indiquer comment `top` et `ps` indiquent un processus dans l'état *Zombie*.

Exercice 9.4 — (facultatif mais instructif) La colonne `STAT` de `ps l` indique avec quelques détails la raison pour laquelle un processus est bloqué. Trouver dans la documentation toutes les valeurs qui peuvent apparaître dans la colonne.

Exercice 9.5 — Sur quels intervalles de temps est calculé le *load average* ?

Exercice 9.6 — Dans le chapitre sur les processus, nous avons fait tourner un programme qui occupait en permanence le CPU :

```
main(){ int x; for(;;) x += 1; }
```

La question vous était posée de savoir combien de fois il fallait lancer le programme pour sentir le ralentissement de l'ordinateur. Refaire la même expérience, mais en lançant maintenant le programme avec la priorité la plus basse.

Chapitre 10

Commandes, fonctions et appels système

Ce petit chapitre a pour ambition de rappeler en un seul endroit une des idées clefs qui sous-tend le contenu du cours : la différence entre appel système, appel de fonction C et commande.

Il utilise comme exemple la commande `sleep` et est l'occasion de présenter brièvement un mécanisme de communication inter-processus courant sous Unix : les *signaux*.

Il présente également quelques outils commun qui permettent d'analyser le contenu des commandes.

10.1 La commande sleep

La commande sleep ne fait rien pendant un temps qu'on lui précise en argument. On l'utilise principalement dans les scripts shell.

10.1.1 Utilisation

L'utilisation de la commande `sleep` est simple : on lui passe le nombre de secondes comme un argument sur la ligne de commande. Pour s'arrêter pendant un nombre de secondes donné, la commande appelle la fonction `sleep()` de la librairie.

Comme (presque) toutes les commandes, `sleep` est documenté par une page de *manuel* qui se trouve dans le premier chapitre de la documentation. Pour voir cette page, le plus simple est d'appeler la commande `man` (comme *manuel*, par exemple avec :

```
$ man sleep
```

On peut forcer la commande `man` à ne regarder que dans le premier chapitre du manuel (qui traite des commandes) en ajoutant son numéro, comme dans :

```
$ man 1 sleep
```

La page du manuel indique que la commande peut avoir deux options (`--help` pour afficher un bref résumé de la commande et `--version` pour connaître son

numéro de version) et que l'intervalle de temps peut se terminer par un caractère qui indique s'il s'agit de secondes (par défaut), de minutes, d'heures ou de jours.

Comme souvent, la documentation est incomplète : elle indique que la commande n'accepte qu'un seul argument, alors que la version qui tourne sur ma machine en accepte plusieurs. Dans ce cas, sleep enchaîne les intervalles de temps :

```
$ date ; sleep 1 1 1 1 ; date
Thu May  8 18:47:42 CEST 2008
Thu May  8 18:47:46 CEST 2008
```

On constate qu'il s'est écoulé quatre secondes entre les deux appels à la commande `date`.

10.1.2 Analyse de la commande

Comme toutes les commandes qui ne sont pas internes au shell, la commande `sleep` est un fichier dans un des répertoires listé dans la variable `PATH`. On peut savoir où se trouve une commande en utilisant une autre commande qui s'appelle `which` :

```
$ which sleep
/bin/sleep
```

Nous savons maintenant que la commande `sleep` correspond au fichier `/bin/sleep`. On peut parfaitement recopier le fichier. Le résultat sera lui aussi exécutable :

```
$ cp /bin/sleep /tmp/toto          copie du fichier qui contient la commande
$ /tmp/toto 3                      lancement de la nouvelle commande
                                   3 secondes passent. Ok
$ /tmp/toto -z                     lancement avec un argument incorrect
/tmp/toto: invalid option -- z     le message d'erreur est modifé
Try '/tmp/toto -help' for more information.
```

Pour savoir de quel type est le contenu d'un fichier, on peut utiliser la commande `file` qui tente de deviner de quel type sont les données qui s'y trouvent :

```
$ file /bin/sleep
/bin/sleep: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), for
GNU/Linux 2.6.8, dynamically linked (uses shared libs), stripped
```

La commande `file` nous indique que le contenu du fichier est le résultat d'une compilation : il contient du code pour le processeur. Il n'est donc pas lisible facilement par nous.

On peut parfois obtenir des informations intéressantes sur une commande en examinant le contenu de la table des symboles du fichier, mais ici la table des symboles a été retirée (sans doute pour gagner un peu de place sur le disque) avec la commande `strip` :

```
$ nm sleep
nm: sleep: no symbols
```

Avec la commande strings, on peut extraire les chaînes de caractère qui apparaissent dans le fichier pour les examiner. On obtient alors toutes les séquences d'au moins quatre caractères imprimables.

```
$ strings /bin/sleep | grep sleep
nanosleep
sleep
xnanosleep.c
Axnanosleep
$
```

On déduit de cet exemple que la commande `sleep` utilise sans doute une fonction `nanosleep`, définie dans un fichier `nanosleep.c`.

On peut également regarder quelles bibliothèques la commande utilisera lors de son exécution à l'aide de la commande `ldd` :

```
$ ldd /bin/sleep
        libc.so.6 => /lib/libc.so.6 (0x00002aafd9a0b000)
        /lib64/ld-linux-x86-64.so.2 (0x00002aafd97ed000)
```

Il s'agit de bibliothèques *dynamiques* qui contiennent des fonctions qui seront ajoutées à l'espace mémoire du programme quand il en aura besoin.

Finalement, parce que nous utilisons un logiciel libre, nous pouvons examiner le source du programme. Si on a installé les sources, le fichier se trouve quelque part dans le répertoire `/usr/src`. Si les sources ne sont pas installés, on le trouve facilement en utilisant un moteur de recherche avec le mot clef `sleep.c`. Le source est un programme C de 151 lignes.

Notre commande `sleep` utilise la fonction `sleep` qui se trouve dans la bibliothèque.

10.2 La fonction `sleep`

10.2.1 Documentation de la fonction

Comme (presque) toutes les fonctions, celle-ci est documentée dans une page du troisième chapitre du manuel. Quand on appelle la commande `man` pour obtenir la page de manuel de `sleep`, celle-ci nous affiche la page concernant la commande et non pas celle de la fonction. Pour voir la page de la fonction, il faut forcer `man` à rechercher dans le chapitre 3 avec

```
$ man 3 sleep
```

Le début de la page contient un résumé en une ligne puis un paragraphe intitulé *synopsis* qui contient l'indication du fichier à inclure pour déclarer le prototype de la fonction, puis une indication de ses arguments et de la valeur qu'elle renvoie.

Pour pouvoir trouver la page de manuel, il faut connaître le nom de la fonction. Pour trouver le nom de la fonction, la commande `man` accepte une option `-k` (comme *keyword*) qui lui fait imprimer la ligne de résumé de toutes les pages de manuel qui contiennent un mot. Par exemple, avec :

```
$ man -k sleep
```

on apprend l'existence de la commande et de la fonction `sleep`, mais aussi de `usleep`, `nanosleep`, `apmsleep`, etc.

10.2.2 Réimplémentation de la commande

Le travail de la commande `sleep` est simple. On peut facilement réécrire le programme en utilisant la fonction `sleep` de la bibliothèque. Nous pouvons choisir de le faire comme dans le programme suivant, sans traiter les nombres avec virgule et sans les options qui sont d'une utilité douteuse. (C'est une bonne surprise que la commande `sleep` ne nous affiche pas un message de Copyright annonçant qu'il s'agit d'un logiciel libre.)

```
/* sleep.c
Une re-implementation de la commande sleep
*/
# include <stdio.h>
# include <stdlib.h>
# include <ctype.h>
# include <unistd.h>

static void
invalid(char * prog, char * s){
  fprintf(stderr, "%s: invalid time interval %s\n", prog, s);
  exit(1);
}

int
main(int ac, char * av[]){
  int i;
  int nsec;

  if (ac != 2){
    fprintf(stderr, "usage: %s N\n", av[0]);
    return 1;
  }

  for(i = 0; isdigit(av[1][i]); i++)
    ;

  if (! isdigit(av[1][0]))
    invalid(av[0], av[1]);
  nsec = atoi(av[1]);
  switch(av[1][i]){
  default:
    invalid(av[0], av[1]);
  case 'd':
    nsec *= 24;
  case 'h':
    nsec *= 60;
  case 'm':
    nsec *= 60;
  case 's':
  case 0:
    ;
```

```
  }

  sleep(nsec);
  return 0;
}
```

La plus grosse partie de la cinquantaine de lignes du programme est consacrée à analyser la validité de l'argument et à le convertir en un nombre entier de secondes, depuis la chaine de caractères formée de chiffre qui le compose..

Ceci ne nous dit pas comment la fonction sleep est réalisée : ce point est traité dans la section suivante.

10.3 Les appels système

Les appels système sont le mécanisme avec lequel un processus demande au noyau d'exécuter une tâche qu'il ne peut pas faire lui-même.

L'utilisation d'un appel système ressemble à un appel de fonction ordinaire, mais le mécanisme sous-jacent est différent : le code demande au noyau du système d'effectuer le travail avec un mécanisme spécial (une interruption logicielle appelée un *trap*).

Alors qu'il est facile de redéfinir une fonction de bibliothèque, les appels système sont fixés par le noyau et un processus ne peut pas le modifier.

Les appels système sont documentés dans le deuxième chapitre du manuel, à la différence des fonctions qui sont documentés dans le troisième chapitre.

10.3.1 Les signaux

La plupart des interactions entre le noyau et les processus ont lieu à l'initiative du processus : celui-ci lance un système qui demande au noyau d'effectuer une tâche (ouvrir un fichier, lire des données, ...).

Il est nécessaire aussi dans certaines situations de permettre au noyau de prendre l'initiative d'une communication avec le processus : par exemple, quand on tape un CTRL-C au clavier, le noyau doit le signaler au processus.

Ceci se fait avec un mécanisme appelé les *signaux* : il existe quelques dizaines de type de message que le noyau peut prendre l'initiative d'envoyer au processus.

Par défaut, quand un processus reçoit un signal, il s'arrête. Il peut cependant choisir d'ignorer le signal, ou demander au noyau d'appeler une fonction du processus quand le signal arrive.

10.3.2 Réimplémentation de la fonction sleep

Voici une implémentation de la fonction `sleep` :

```
/* yc-sleepfn.c
Une re-implementation de la fonction sleep
 */
# include <stdio.h>
# include <unistd.h>
# include <signal.h>
```

```
void
dring(int n){
  /* rien a faire */
}

unsigned int
sleep(unsigned int nsec){
  int avant, apres;

  avant = time(0);
  signal(SIGALRM, dring);
  alarm(nsec);
  pause();
  apres = time(0);
  return (avant + nsec) - apres;
}
```

Avec l'appel système `signal`, la fonction demande au noyau d'appeler la fonction `dring` quand un signal du type `SIGALRM` arrive.

Avec l'appel système `alarm`, elle demande l'envoi d'un signal au bout du nombre de secondes qu'elle a reçu en argument.

Avec l'appel système `pause`, le processus s'interrompt jusqu'à l'arrivée d'un signal.

Au bout du temps écoulé, le noyau va envoyer le signal `ARLM` au processus ; l'arrivée du signal provoque l'interruption de l'appel système `pause en cours` et l'appel de la fonction `dring` comme celui-ci l'a spécifié. La fonction dring ne fait rien, mais le signal a interrompu le `pause` et le processus continue.

Finalement la fonction sleep renvoie, comme indiqué dans la documentation, le nombre de secondes qui restaient à dormir. Normalement ce nombre est égal à 0, mais l'appel système `pause` a pu être interrompu par l'arrivée d'un autre signal et dans ce cas la pause a pu être plus courte que prévue.

10.3.3 Utilisation de la fonction

On peut compiler notre programme initial pour utiliser notre fonction `sleep` au lieu de la fonction de la bibliothèque :

```
$ gcc wa-sleep.c wc-sleepfn.c
$ a.out 12
```

Le compilateur ne va chercher dans la bibliothèque que les fonctions qui ne sont pas définies dans le programme. Nous pouvons réécrire les fonctions dont le comportement ne nous satisfait pas, y compris des fonctions qu'on finit par considérer comme faisant partie du langage, comme `printf`. *Attention*, cette possibilité est à utiliser avec prudence si on ne veut pas se retrouver avec des programmes illisibles.

10.4 Conclusion

Il y a deux choses importantes dans ce chapitre. (1) Il est important de distinguer les *commandes*, les *fonctions* et les *appels système*. Une commande

est un fichier exécutable. Une fonction de bibliothèque est un bout de code qu'on peut appeler depuis nos programmes ; il est facile de la redéfinir. Un appel système se présente aussi comme une fonction, mais interagit directement avec le noyau du système d'exploitation et il est à peu près impossible de le redéfinir. (2) Les programmes et les fonctions n'ont pas besoin d'être compliqués.

Exercices

Exercice 10.1 — Quelles commandes du répertoire `/bin` ne sont pas *dynamically linked*? (Indication : l'option -v de `grep` permet de sélectionner rapidement les lignes intéressantes dans la sortie de la commande `file /bin/*`).

Exercice 10.2 — Trouver le source de la commande sleep.

Exercice 10.3 — Modifier le programme du cours pour que la commande `sleep` accepte plusieurs arguments (Indication : pour le faire, je n'ai ajouté que deux lignes au programme.)

Exercice 10.4 — Ajouter un printf dans notre fonction `sleep` pour être bien certain que c'est *notre* fonction `sleep` et non celle de la bibliothèque qui est appelée.

GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom : to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation : a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals ; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers

or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties : any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts : Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things

in the Modified Version :

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version..

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another ; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History" ; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate,

this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http ://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been

published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.