Systèmes d'exploitation

Timothy Bourke, Marc Pouzet
Notes par Antoine Groudiev

Version du 3 février 2024

Table des matières

Introduction

Ce document est l'ensemble non officiel des notes du cours *Systèmes d'exploitation* du Département Informatique de l'ENS Ulm. Elles sont librement inspirées des notes de cours sous forme de présentation rédigées par Timothy Bourke et Marc Pouzet.

1 Micro-noyau

Commençons ce cours par la programmation d'un micro-noyau fortement simplifié, implémenté en OCaml. Notre objectif sera de retenir l'essentiel d'un noyau de système classique, en exécutant le moins de fonctions possibles en mode super-utilisateur [1].

1.1 Description générale d'un micro-noyau

Un micro-noyau contient une ou plusieurs *applications*, comme un système de fichier ou un driver de disque.

Les principales fonctionnalités d'un micro-noyau sont de gérer les processus, la communication entre eux, et la mémoire virtuelle. Il doit être capable de créer, arrêter, ordonner les processus en fonction de leur priorité.

On se donne les caractéristiques suivantes pour l'architecture machine :

— elle est capable d'exécuter un seul processus à la fois

— elle possède cinq registres, de `r0` à `r4`

1.2 Appels système

Le micro-noyau doit être capable de réagir à deux types d'évènements :

— l'interruption d'un compteur de temps (`timer`)

— des interruptions logicielles (`system trap` ou `software interrupt`)

Les processus de l'utilisateur peuvent changer le contenu des registres et générer des appels système arbitraires. Quand un appel système est déclenché, le micro-noyau lit le contenu des registres pour déterminer l'appel effectué et les arguments de cet appel. Il réagit en effectuant l'appel (par exemple, la mise à jour de l'état du système) et en plaçant les valeurs de retour dans les registres.

On définit les codes d'appels systèmes suivants : En cas d'appel système invalide (pour une

Registre `r0`	Appel système
0	`new_channel`
1	`send`
2	`receive`
3	`fork`
4	`exit`
5	`wait`

valeur de `r0` non renseignée dans le tableau), le noyau n'exécute aucun code et place la valeur -1 dans `r0`.

1. Les fonctions exécutées en mode "super-utilisateur" ont un accès non protégé aux ressources.

1.3 Constantes et types OCaml

On définit les constantes et type OCaml suivant pour représenter notre micro-noyau :

```
let max_time_slices = 5 (* 0 <= t < max_time_slices *)
let max_priority = 15 (* 0 <= p <= max_priority *)
let num_processes = 32
let num_channels = 128
let num_registers = 5

type pid = int (* process id *)
type chanid = int (* channel id *)
type value = int (* values transmitted on channels *)
type interrupt = int (* software interrupt *)
type priority = int (* priority of a process *)

type registers = {
    r0 : int;
    r1 : int;
    r2 : int;
    r3 : int;
    r4 : int;
}

let get_registers { registers } = {
  r0 = registers.(0); r1 = registers.(1);
  r2 = registers.(2); r3 = registers.(3);
  r4 = registers.(4); }
(* the set of processes ordered by priority *)

let set_registers { registers } { r0; r1; r2; r3; r4 } =
  registers.(0) <- r0;
  registers.(1) <- r1;
  registers.(2) <- r2;
  registers.(3) <- r3;
  registers.(4) <- r4
```

On définit ensuite un processus à l'aide du type suivant :

```
type process_state =
    | Free (* non allocated process *)
    | BlockedWriting of chanid
    | BlockedReading of chanid list
    | Waiting
    | Runnable
    | Zombie

type process = {
    mutable parent_id : pid;
    mutable state : process_state;
    mutable slices_left : int;
    saved_context : int array;
}
```

Les états des processus sont décrits par le diagramme ci-dessous :
On définit par ailleurs un état du noyau à l'aide du type `state` suivant :

```
type channel_state =
    | Unused (* non allocated channel *)
    | Sender of pid * priority * value
    | Receivers of (pid * priority) list

type state = {              (* kernel state *)
    mutable curr_pid   : pid; (* process id of the running process *)
    mutable curr_prio  : priority; (* its priority *)
    registers  : int array;        (* its registers *)
    processes  : process array;       (* the set of processes *)
    channels   : channel_state array; (* the set of channels *)
    runqueues  : pid list array;
}

let get_current { curr_pid = c } = c
```

Finalement, on définit un évènement, qui peut être soit un `timer`, soit un appel système :

```
type event = | Timer | SysCall

type syscall =
    | Send of chanid * value
    | Recv of chanid list
    | Fork of priority * value * value * value
    | Wait
    | Exit
    | NewChannel
    | Invalid
```

1.4 Détermination de l'appel système

Ajoutons une fonction `decode: state -> syscall` qui décode la valeur des registres et détermine l'appel système.

```
let decode { registers } =
match registers.(0) with
| 0 -> NewChannel
| 1 -> Send (registers.(1), registers.(2))
| 2 -> Recv [registers.(1);
             registers.(2);
             registers.(3);
             registers.(4)]
| 3 -> Fork (registers.(1), registers.(2),
             registers.(3), registers.(4))
| 4 -> Exit
| 5 -> Wait
| _ -> Invalid
```

1.5 Appel système `fork`

Description L'appel système `fork` crée un nouveau processus fils. Chaque processus est associé à une priorité comprise entre 0 (la plus basse) et 15 (la plus haute). Le registre `r1` spécifie la priorité du processus créé.

Si la priorité donnée est strictement plus grande que la priorité du processus qui crée le processus fils, l'appel système se termine sans créer de processus et en plaçant 0 dans `r0`. Concrètement, un processus ne peut pas engendrer un processus de priorité plus élevée que la sienne.

Si la priorité est valide et qu'un nouveau processus peut être créé, `r0` reçoit la valeur 1 et `r1` reçoit le numéro du processus créé. Si un nouveau processus ne peut pas être créé, `r0` reçoit la valeur 0. Ceci arrive en particulier lorsque le nombre maximum de processus pouvant être créés est égal à `num_processes`.

Dans le processus fils créé, `r0` est initialisé à 1, `r1` est initialisé au numéro de processus du père (qui a fait l'appel à `fork`), et les autres registres (`r2`, `r3` et `r4`) sont copiés du processus parent.

Implémentation On implémente `fork` sous la forme d'une fonction de type

```
fork: state -> int -> int -> int -> int -> unit
```

qui sera appelée sous la forme

```
fork state nprio d0 d1 d2
```

où `state` est l'état du système, `nprio` est la priorité à donner au processus fils, `d0`, `d1` et `d2` sont les valeurs à passer au fils pour initialiser ses trois derniers registres.

```
let fork { curr_pid; curr_prio; registers; processes; runqueues } nprio d0 d1 d2 =
    let rec new_proc i =
        if i >= num_processes then None
        else if processes.(i).state = Free then
            let np = processes.(i) in
            np.parent_id <- curr_pid;
            np.state <- Runnable;
            np.slices_left <- max_time_slices;
            np.saved_context.(0) <- 2;
            np.saved_context.(1) <- curr_pid;
            np.saved_context.(2) <- d0;
            np.saved_context.(3) <- d1;
            np.saved_context.(4) <- d2;
            Some i
        else new_proc (i + 1)
    in
    match new_proc 0 with
    | None -> registers.(0) <- 0
    | Some npid -> begin
            registers.(0) <- 1;
            registers.(1) <- npid;
            runqueues.(nprio) <- runqueues.(nprio) @ [npid]
        end
```

1.6 Appel système `exit`

Description L'appel système `exit` termine l'exécution du processus l'exécutant. Son argument, la valeur de retour de l'appel, est placé dans le registre `r1`.

Après appel à `exit`, le processus entre dans l'état `Zombie`, et ce jusqu'à l'exécution de l'appel système `wait` qui récupèrera la valeur de retour.

Si le processus terminé avait des fils, ils deviennent *orphelins*. L'identifiant de leur père devient alors le processus 1, appelé `init`.

Implémentation On implémente `exit` sous la forme d'une fonction de type

```
exit: state -> unit
```

```
let exit { curr_pid; curr_prio; registers; processes; runqueues } =
    let { parent_id } as p = processes.(curr_pid) in

    (* tous les fils ont maintenant comme père le processus n°1 *)
    let f p = if p.parent_id = curr_pid then p.parent_id <- 1 in
    Array.iter f processes;

    runqueues.(curr_prio) <-
        List.filter (fun pid -> pid <> curr_pid) runqueues.(curr_prio);

    if processes.(parent_id).state = Waiting
    then begin
        processes.(parent_id).state <- Runnable;
        processes.(curr_pid).state <- Free;
        let saved_registers = processes.(parent_id).saved_context in
        saved_registers.(0) <- 1;
        saved_registers.(1) <- curr_pid;
        saved_registers.(2) <- registers.(0)
    end
    else processes.(curr_pid).state <- Zombie
```

1.7 Appel système `wait`

Description Un processus est *en attente* (mode `Waiting`) jusqu'à ce qu'un de ses fils meurt. S'il ne lui reste plus aucun fils, l'appel système rend la main immédiatement en plaçant 0 dans `r0`. S'il reste un processus fils dans le mode `Zombie` ou lorsqu'un fils termine, l'appel à `wait` termine en plaçant 1 dans `r0`, l'identifiant du fils dans `r1` et la valeur de retour de ce fils dans `r2`. S'il y a plusieurs fils dans le mode `Zombie`, l'un d'eux est choisi arbitrairement.

Implémentation On implémente `wait` sous la forme d'une fonction de type

```
wait: state -> bool
```

où résultat de `wait state` est vrai s'il est nécessaire de réordonnancer le processus courant (c'est-à-dire le replacer dans l'état du système et choisir un nouveau processus à ordonnancer).

```
let wait {curr_pid; registers; processes} =
    let rec already_dead has_child i =
        if i = num_processes then has_child, None
```

```
        else begin
            let { state; parent_id; saved_context} = processes.(i) in
            if state = Zombie && parent_id = curr_pid
            then true, Some (i, saved_context.(0))
            else already_dead (has_child || parent_id = curr_pid) (i + 1)
        end
    in
    match already_dead false 0 with
    | has_child, None ->
        if has_child
        then (processes.(curr_pid).state <- Waiting; true)
        else (registers.(0) <- 0; false)
    | _, Some (pid, v) ->
        processes.(pid).state <- Free;
        registers.(0) <- 1;
        registers.(1) <- pid;
        registers.(2) <- v;
        false
```

1.8 Appel système `new_channel`

Description La communication entre processus s'effectue par envoi et écriture dans un canal numéroté, suivant un protocole de *rendez-vous* ("handshake"). L'appel système `new_channel` crée donc un nouveau canal. La valeur de retour `r0` de cet appel système est soit le numéro du canal créé, soit une valeur négative si un nouveau canal n'a pas pu être créé. `num_channels` est le nombre maximal de canaux pouvant être créés.

Implémentation On implémente `new_channel` sous la forme d'une fonction de type

```
new_channel: state -> unit
```

```
let new_channel {registers; channels} =
let rec new_channel i =
    if i >= num_channels then -1
    else if channels.(i) = Unused
        then (channels.(i) <- Receivers []; i)
        else new_channel (i + 1)
in
registers.(0) <- new_channel 0
```

1.9 Appel système `send`

Description Cet appel prend deux arguments : un canal `r1`, et une valeur à envoyer `r2`. Le numéro du canal doit être valide, c'est-à-dire avoir été créé par un appel à `new_channel`.

Si un autre processus est déjà en train d'envoyer une valeur sur le canal (le canal est bloqué en attente d'un récepteur), ou si le canal est invalide, la valeur de retour de `send` placée dans `r0` vaut 0.

Si un autre processus est déjà en attente sur le canal, l'appel `send` réussit immédiatement, et le processus en attente sur le canal passe alors du mode `Blocked` au mode `Runnable`. Lorsque plusieurs processus récepteurs sont en attente, le processus de plus fort priorité est choisi arbitrairement et les autres restent bloqués.

Sinon, le processus émetteur se bloque jusqu'à l'arrivée d'un récepteur. La valeur de retour de l'appel système est 1, dans `r0`.

Implémentation On implémente `send` sous la forme d'une fonction de type

```
send: state -> chanid -> value -> bool
```

```
(* il y a deja quelqu'un qui attend sur le canal *)
let release_receiver {channels; processes} rid =
    let clear ch =
        channels.(ch) <-
            match channels.(ch) with
            | Receivers rs -> Receivers (List.filter (fun (pid, _) -> pid <> rid) rs)
            | v -> v
    in
    (match processes.(rid).state with
    | BlockedReading rchs -> List.iter clear rchs
    | _ -> assert false);
    processes.(rid).state <- Runnable


let send
    ({curr_pid; curr_prio; registers; processes; channels} as s) ch v
    match channels.(ch) with
    | Sender _ | Unused -> (registers.(0) <- 0; false)

    | Receivers [] ->
        (* personne n'attend sur le canal *)
        channels.(ch) <- Sender (curr_pid, curr_prio, v);
        processes.(curr_pid).state <- BlockedWriting ch;
        true

    | Receivers ((rid, rprio)::rs) ->
        release_receiver s rid;
        let saved_registers = processes.(rid).saved_context in
        saved_registers.(0) <- 1;
        saved_registers.(1) <- ch;
        saved_registers.(2) <- v;
        registers.(0) <- 1;
        rprio >= curr_prio
```

1.10 Appel système `receive`

Description L'appel système `receive` permet de se synchroniser avec au plus 4 canaux, spécifiés dans les registres `r0` à `r1`. Cet appel permet donc d'écouter sur plusieurs canaux à la fois. Il réussit lorsqu'un rendez-vous a lieu avec un des émetteurs.

Les canaux invalides sont ignorés. Si aucun canal valide n'est spécifiés, l'appel système rend la main immédiatement en plaçant 0 dans `r0`.

Si un ou plusieurs émetteurs sont en attente sur un des canaux valides, l'un est choisi arbitrairement et l'appel `receive` rend la main immédiatement en plaçant 1 dans `r0` et en

donnant à `r1` la valeur du canal choisi pour la réception, et en plaçant dans `r2` la valeur envoyée sur le canal.

Sinon, le récepteur est bloqué jusqu'à ce qu'une émission ait lieu sur un des canaux spécifiés.

Implémentation On implémente `receive` sous la forme d'une fonction de type

```
receive: state -> chanid list -> unit
```

```
let recv {curr_pid; curr_prio; registers; processes; channels} chs =
    let rec sender_ready chans =
        match chans with
        | [] -> None
        | ch::chs ->
            match channels.(ch) with
            | Sender (sid, sprio, sv) -> Some (ch, sid, sprio, sv)
            | _ -> sender_ready chs in
    let rec add_to_receivers blocked chans =
        match chans with
        | [] -> blocked
        | ch::chs ->
            match channels.(ch) with
                | Unused | Sender _ -> add_to_receivers blocked chs
                | Receivers rs ->
                channels.(ch) <- Receivers (insert_receiver (curr_pid, curr_prio) rs);
                add_to_receivers (ch::blocked) chs
    in
    match sender_ready chs with
    | Some (ch, sid, sprio, sv) ->
        channels.(ch) <- Receivers [];
        processes.(sid).state <- Runnable;
        registers.(0) <- 1;
        registers.(1) <- ch;
        registers.(2) <- sv;
        sprio >= curr_prio
    | None ->
        match add_to_receivers [] chs with
        | [] -> (registers.(0) <- 0; false)
        | bchs -> (processes.(curr_pid).state <- BlockedReading bchs; true)
```

1.11 État initial du système et ordonnancement des processus

1.11.1 État initial du système d'exploitation

Le système démarre en créant deux processus :

— le processus `idle` de numéro 0, de priorité 0 et de père égal à lui-même ;

— le processus `init` de numéro 1, de priorité 15 et de père égal à lui-même ;

Tous les registres sont initialisés à 0, et aucun canal n'est créé. On peut supposer que le processus `idel` est toujours exécutable et que ni le processus `idle` ni le processus `init` ne terminent. L'*état observable* du système est l'identifiant du processus en cours d'exécution et le contenu des cinq registres. L'*état interne* du système d'exploitation ne peut être observé ni modifié de l'extérieur.

1.11.2 Ordonnancement des processus

Le rôle du noyau est alors d'élire un processus à exécuter parmi la liste des processus, en lui allouant un quantum de temps maximum, et de traiter les appels système considérés précédemment.

Un processus est exécutable ou prêt (`Runnable`) s'il n'est pas bloqué sur un canal ni n'attend l'un de ses fils, et n'est pas un zombie. Le noyau choisit les processus prêts de priorité la plus forte avec un protocole *"round robin"* : lorsqu'un processus en cours d'exécution est interrompu, il retourne en fin de queue parmi les processus de même priorité.

1.11.3 Interruptions et tranches de temps

Le système reçoit des interruptions périodiques venant d'une horloge externe (`timer`). Une interruption indique la fin d'une *tranche de temps* (*time slice*) ; un processus ne peut pas s'exécuter pendant une durée égale à au plus de cinq tranches de temps (*max_time_slices*). Ce temps n'est décompté que pour le processus en cours d'exécution. Le noyau doit donc mettre à jour le compteur de temps du processus en cours d'exécution, et seulement celui-ci.

1.11.4 Changement de contexte

Un *changement de contexte* (changement du processus en cours d'exécution) se produit dans deux cas :

— lorsque le processus se bloque, par exemple en exécutant un `send` et qu'aucun processus n'écoute sur le canal correspondant ;

— lorsqu'il est préempté parce qu'il a atteint sa durée maximale d'exécution.

Les valeurs des registres doivent alors être sauvegardées et restaurées au travers du changement de contexte.

1.12 Implémentation de l'ordonnancement

La dernière partie de l'implémentation du noyau consiste en une fonction

```
transition: even -> state -> unit
```

qui, en fonction de l'évènement reçu, exécute le code de l'appel système, fait avancer le pas de temps du processus en cours d'exécution, ou élit un processus à exécuter.

```
let save_context { registers; processes } pid =
    Array.blit
        registers 0 processes.(pid).saved_context 0 num_registers

let restore_context { registers; processes } pid =
    Array.blit
        processes.(pid).saved_context 0 registers 0 num_registers

let choose_process { runqueues; processes } =
    let rec find_within rq =
        match rq with
        | [] -> None
        | rid::rq' ->
            if processes.(rid).state = Runnable then Some rid
```

```
            else find_within rq'
    in
    let rec find prio =
        match find_within runqueues.(prio) with
        | None -> find (prio - 1)
        | Some pid -> prio, pid
    in
    find max_priority

let schedule ({curr_pid=prev_pid} as s) =
    let next_prio, next_pid = choose_process s in
    save_context s prev_pid;
    restore_context s next_pid;
    s.curr_pid <- next_pid;
    s.curr_prio <- next_prio

let timer_tick { curr_pid; curr_prio; processes; runqueues } =
    let ({ slices_left = remaining } as p) = processes.(curr_pid) in
    p.slices_left <- remaining - 1;
    if remaining = 0
    then
        (p.slices_left <- max_time_slices;
                          runqueues.(curr_prio) <-
    List.filter
        (fun pid -> pid <> curr_pid) runqueues.(curr_prio)
        @ [curr_pid]; true)
    else false


let transition ev ({curr_pid; registers; curr_prio} as s) =
    let reschedule =
        match ev with
        | Timer -> timer_tick s
        | SysCall ->
            match decode s with
            | Send (ch, v) -> send s ch v
            | Recv chs -> recv s chs
            | Fork (prio, d0, d1, d2) ->
                ((if prio <= curr_prio
                  then fork s prio d0 d1 d2
                  else registers.(0) <- 0); false)
            | Wait -> wait s
            | Exit -> (exit s; true)
            | NewChannel -> (new_channel s; false)
            | Invalid -> (registers.(0) <- -1; false)
    in
    if reschedule then schedule s
```

On peut enfin définir une fonction d'initialisation du système d'exploitation :

```
let init =
    let new_procs i =
        if i = 0 then {
        parent_id = 0;
        state = Runnable;
        saved_context = Array.make num_registers 0;
        slices_left = max_time_slices }
    else if i = 1 then {
        parent_id = 1;
        state = Runnable;
        saved_context = Array.make num_registers 0;
        slices_left = max_time_slices }
    else {
        parent_id = 0;
        state = Free;
        saved_context = Array.make num_registers 0;
        slices_left = 0 }
    in
    let new_queues i =
        if i = max_priority then [1]
        else if i = 0 then [0]
        else []
```

2 Introduction à Unix et au *shell*

Le chapitre précédent a présenté les éléments fondateurs d'un micro-noyau. En pratique, de nombreux systèmes d'exploitation sont fondés sur la norme POSIX, qui descend elle-même du système Unix.

2.1 Introduction

Le système Unix a été developpé dans les années 1970 par Dennis M. Ritchie et Ken Thompson. C'est un système d'exploitation puissant créé en moins de deux personnes/années, court développement comparée à ses contemporais Multics et OS/360. Il implémente plusieurs idées importantes encore utilisées de nos jours, et trouve un point d'équilibre entre les limitations matérielles de l'époque, la difficulté d'implémentation, et l'abstraction pour les utilisateurs.

L'héritage d'Unix est vaste et complexe. Sa création a été suivie de décennies d'implémentations diverses, de problèmes juridiques et de questions de portabilité. Parmi les decendants d'Unix, on retiendra notamment :

— System V, d'AT&T, ancêtre de nombreuses versions commerciales.

— Berkeley Software Distribution (BSD), version open-source ancêtre de FreeBSD, OpenBSD, et en partie macOS

— Linux et les utilitaires GNU. On notera en particulier la très large compatibilité de Linux, qui sert notamment de fondation à Android de Google.

— Le standard IEEE POSIX (Portable Operating System Interface.)

2.2 Le shell Unix

Le shell Unix est un interpréteur de commandes accessible depuis la console ou un terminal. Malgrès la généralisation des interfaces graphiques, il est toujours utile d'apprendre à utiliser le shell.

En effet, le shell reste très efficace pour programmer, administrer des machines à distance. La capacité des commandes à être combinées permet d'automatiser une partie des tâches faites manuellement. Par exemple, `ssh` est particulièrement adapaté à la connection à distance aux services de cloud, comme Amazon Web Services. Par ailleurs, le terminal est un bon moyen de comprendre le fonctionnemnet de Unix.

Les *scripts shell* sont souvent utilisés pour améliorer la productivité des actions nécessitant le terminal. Ils permettent d'automatiser facilement une séquence d'opérations, comme la création de nouvelles commandes, ou encore d'installer ou mettre à jour automatiquement ses propres applications via des *scripts de configuration*. Leur maîtrise est un gain de temps, et utile pour débugger les scripts d'autres personnes.

2.3 Éléments d'un système d'exploitation

Un système d'exploitation est constitué d'un *noyau*, de *librairies*, et d'*applications*. Il régit le partage de la mémoire, de l'utilisation des processeurs, et des périphériques (claviers, disques, cartes graphiques, ...). Le système d'exploitation ajoute un niveau d'abstraction permettant l'indépendance matérielle d'une application, de services communs (systèmes de fichiers, contrôle d'accès), et la concurrence et communication des applications, tout en maintenant protection et contrôle d'accès.

Un noyau contient de nombreuses structures de données à des fins de comptabilité, des modules abstraits (interfaces, strucutres de données, fonctions) pour les différents services. Il contrôle les éléments matériels à très bas niveau grâce à des *drivers*.

2.4 Hiérarchie du système de fichier

Introduction La hiérarchie du système de fichier sous Unix est constituée d'un unique arbre, l'*espace de noms hierarchique*, pour nommer et accéder au stockage local ou en réseau, aux drivers, ou au structures de données du noyau.

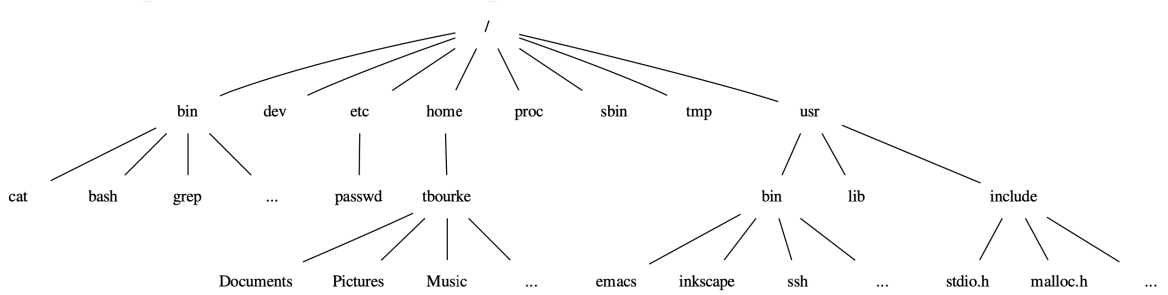

FIGURE 1 – Arborescence Unix par défaut

La commande `ls` utilisée avec un *chemin relatif* permet d'afficher le contenu d'un dossier :

```
ls /etc
```

La commande `cat` permet d'afficher le contenu d'un fichier :

```
cat /etc/passwd
```

La commande `df` permet d'analyser comment le stockage physique est associé à la hiérarchie.

Standards Cette hiérarchie suit des standards et conventions, qui sont détaillés dans le manuel `man hier`. On peut notamment citer :

— `/bin` : les programmes utilisés pour les réparations, le démarrage, le mode à utilisateur unique

— `/sbin` : identique à `/bin`, mais n'est normalement pas exécuté par les utilisateurs normaux

— `/etc` : fichiers de configuration du système

— `/home` : dossiers des utilisateurs

— `/tmp` : fichiers temporaires, pouvant être nettoyé au redémarrage

— `/var/tmp` : fichiers temporaires conservés au redémarrage

— `/usr` : contient la majorité des programmes, bibliothèques

— `/dev` : fichiers spéciaux pour accéder aux appareils d'entrée et de sortie

Les commandes suivantes sont utiles pour manipuler des fichiers et dossiers :

— `mkdir` créé un nouveau dossier

— `rmdir` supprime un dossier vide

— `rm` supprime un fichier

— `find` ***`path`*** affiche une sous-arborescence ou des fichiers qui vérifient le critère spécifié

En plus de la liste de répertoires précédente, `/proc` est un dossier virtuel permettant de réaliser des requêtes sur les structures de données du noyau. On pourra essayer par exemple :

— `cat /proc/version`

— `cat /proc/cpuinfo`

— `cat /proc/limits`

2.5 *Everything is a file*

Pour Unix, un fichier est un flux d'octets. Par exemple, un fichier est une séquence de lignes délimitées par un charactère de retour à la ligne `\n`. Un fichier binaire peut être interprété comme un exécutable, ou ouvert par une application. En particulier, même les dossiers sont des fichiers, constitués d'une liste d'entrée correspondant aux fichiers et sous-dossiers, représentés par un nom et un identifiant.

De nombreux appareils et périphériques sont également représentés par des fichiers. Les imprimantes, ports, et d'autres abstractions de protocoles de communication tels que les *pipes* nommés, ou les *sockets*, peuvent être manipulés comme n'importe quel fichier. Ainsi, un ensemble réduit d'appels systèmes (`open` `read`, `write`, `close`, ...) remplit de nombreuses fonctions.

2.6 Gestion des permissions

La commande `ls -l` permet d'afficher plus d'informations à propos des fichiers dans un répertoire, par exemple :

```
-rwxr-xx   2 user grp 4096 Feb 10 07:05 file
```

Chaque répertoire et fichier a un utilisateur et groupe défini. Il existe trois *groupes de permissions* : `user`, `group`, et `other` (`ugo`). Chaque groupe peut avoir les permissions de lecture, d'écriture, et d'exécution (`read`, `write`, et `execute`).

Ainsi, le code affiché au début du résultat de `ls -l` correspond à la convention suivante :

```
d rwx rwx rwx
```

`-`	fichier classique
`d`	dossier
`l`	lien symbolique
`b`	appareil par bloc
`c`	appareil par charactère
`f`	fifo
`s`	*socket* local

où l'emplacement du `d` correspond au type de fichier.

La commande `chmod` (*change mode*) pour changer les permissions d'un fichier. Par exemple :

— `chmod g+x /path/to/file` donne l'autorisation d'exécution à tous les membres du groupe

— `chmod o-rw /path/to/file` retire l'accès en lecture et écriture aux membres du groupe "autres"

— `chmod u=rw,g=r /path/to/file` permet d'assigner les permissions directement