Systèmes d'exploitation, introduction

Michel Billaud (michel.billaud@u-bordeaux.fr, michel.billaud@laposte.net)

31 juillet 2020

Table des matières

1 Introduction aux systèmes d'exploitation

Un système d'exploitation est un logiciel qui agit comme intermédiaire entre

- le matériel d'un ordinateur (processeur, mémoire, disque, réseau…),
- les programmes d'application que fait tourner l'utilisateur.

1.1 Fonctions d'un système d'exploitation

Le système d'exploitation fournit un "environnement" dans lesquels les programmes d'application peuvent s'exécuter de façon

- sûre : les programmes sont protégés les uns des autres,
- commode : le système s'occupe des opérations de bas niveau,
- et efficace : l'utilisation des ressources est optimisée.

Le système d'exploitation fournit une API (*Application Programming Interface*), une bibliothèque d'appels de fonctions par lesquels les programmes d'application lui demandent d'exécuter une action.

Illustration, pendant l'**exécution du programme** C suivant

```
#include <stdio.h>

int main() {
   printf("Hello, world\n");
```

```
    return 0;
}
```

l'appel de `printf` demande l'écriture d'une chaîne sur la "sortie standard" associée au processus, ce qui conduira (peut-être) le système d'exploitation à demander au "gestionnaire d'interface graphique" d'afficher des caractères dans une fenêtre, ce qui passera par des demandes d'accès au pilote de la carte graphique.

Ensuite, en retournant la valeur 0, la fonction `main` signale au système la fin du programme, en fournissant le code de retour qui par convention signifie une fin normale.

Précision : `printf()` est définie dans la *bibliothèque standard* du langage C*. Elle fait appel à la fonction `write` qui fait partie de l'API du noyau du système d'exploitation. De même, le `return` du `main` entraîne l'exécution de l'appel système `_exit()`.

Mais avant cette exécution, le système d'exploitation aura **chargé le programme**, c'est-à-dire :

- trouvé l'exécutable du programme,
- réservé de la place suffisante en mémoire,
- copié le fichier exécutable à cet endroit,
- créé un nouveau processus,
- démarré ce processus.

1.2 Évolution conjointes des matériels et des systèmes

Le matériel, les besoins des utilisateurs, et les systèmes d'exploitation ont évolué conjointement.

Dans les premiers ordinateurs, on ne pouvait charger qu'un programme à la fois dans une mémoire de quelques milliers d'octets.

On constate alors que, lorsqu'un programme exécute une opération d'entrée/sortie (sur bande magnétique, sur un terminal, …), le processeur doit attendre le résultat pour continuer : des milliers de "cycles" sont gaspillés pendant ces temps morts.

Avec une mémoire plus grande, on peut envisager d'avoir **plusieurs** tâches (programmes). Quand une tâche est **bloquée** en attente d'une réponse, le processeur peut faire avancer une autre tâche.

Le système d'exploitation est alors chargé de réveiller/endormir les tâches en fonction des requêtes d'entrée-sortie soumises par les tâches, et de l'arrivée des résultats, et de choisir une des tâches pour l'activer.

Ce "multitraitement" est facilité par l'introduction de mécanismes matériels spécifiques dans la conception des processeurs.

2 Fonctionnement d'un processeur

2.1 Base : le cycle "fetch, decode and execute"

Dans une approche très simplifiée, le processeur d'un ordinateur "classique" possède un **compteur de programme** (PC = *program counter*), registre qui contient l'adresse de la prochaine instruction à exécuter.

Les circuits du processeur exécutent un cycle

- aller chercher en mémoire (*fetch*), à l'adresse indiquée par le PC, l'instruction à exécuter,
- décoder l'instruction,
- l'exécuter,
- modifier le PC pour passer à la suivante.

"Passer à la suivante" consiste

- généralement à incrémenter le PC, qui désignera alors l'instruction qui est *physiquement* la suivante,
- dans le cas des **instructions de branchement** (sauts, appels de sous-programme,…), à mettre dans le PC l'adresse de destination.

2.2 Les interruptions

Les interruptions ont été introduites initialement pour traiter les "**exceptions arithmétiques**" (débordements et autres).

2.2.1 Les exceptions arithmétiques de l'UNIVAC 1

Lorsque l'unité arithmétique détecte un débordement, un signal électrique est "**levé**".

En présence de ce signal, la valeur 0 est chargée dans le PC. A cette adresse se trouve le code à exécuter en cas de débordement arithmétique.

L'exécution normale est donc interrompue, "détournée" vers une "**routine de traitement** de l'exception".

A la fin de cette routine se trouve une instruction "retour d'interruption" qui permettra de revenir au code qui a été interrompu.

Il faut pour cela que le PC ait été sauvegardé lors de l'interruption, soit dans un registre spécial, soit dans une pile.

2.2.2 Interruptions matérielles

Une première idée, pour un système de multitraitement, serait d'interroger périodiquement (*polling*) les périphériques pour savoir si ils ont terminé le travail qu'on leur a demandé.

Mais il est beaucoup plus efficace que le périphérique lui-même indique qu'il requiert l'intervention du système d'exploitation (frappe sur un clavier, arrivée d'une trame sur une interface réseau, réponse d'un contrôleur de disques à qui on a demandé de lire un bloc, signal envoyé par un **timer** après un délai programmé, …).

Pour cela le périphérique "lève" un signal électrique qui représente une **interruption matérielle**, traitée comme précédemment.

En pratique, ces signaux sont reçus par un circuit "contrôleur d'interruptions", qui transmettra au processeur un *numéro* d'interruption. Ce numéro permettra au processeur de trouver l'adresse du code à exécuter dans une "table de vecteurs d'interruption"[1].

2.2.3 Programmer avec des interruptions : illustration Arduino

Un exemple sur la plateforme Arduino pour micro-contrôleurs. Sur cette plateforme

- la fonction `setup()` est appelée au démarrage du programme,
- le fonction `loop()` est ensuite exécutée en boucle.
- la fonction `attachInterrupt()` permet d'"armer" une interruption : elle indique quelle fonction sera exécutée quand l'évènement indiqué se produira.

```
/*
 * Utilisation des interruptions sous Arduino.
 *
 * Objectif :  quand on appuie sur le bouton, l'état de la LED Change.
 *
 * Fonctionnement :
 * - dans setup(), la fonction changeState est associée à la montée du signal
 * provenant du bouton
 * - la fonction changeState() inverse (true <-> false) l'indicateur "on".
 * - la fonction loop() reflète l'état de cet indicateur sur la LED.
 */

const int ledPin = 13;
const int buttonPin = 2;

boolean on = true;

void changeState() {
  on =! on;
}

void setup() {
  pinMode(ledPin, OUTPUT);
```

1. Un **vecteur d'interruption** est une structure de données qui contient en particulier l'adresse d'une routine de traitement d'interruption.

```
  pinMode(buttonPin, INPUT);
  attachInterrupt(digitalPinToInterrupt(buttonPin), changeState, RISING);
}

void loop() {
  if (on) {
    digitalWrite(ledPin, HIGH);
  }
  else {
    digitalWrite(ledPin, LOW);
  }
}
```

2.2.4 Masquage et priorités

En pratique, on veut souvent éviter d'interrompre le déroulement d'une routine de traitement. Pour cela on introduit un "masque d'interruptions". Quand une interruption est masquée, elle est mise en attente, et sera prise en compte quand l'exécution de l'instruction `RTI` "démasquera" les interruptions.

Plus généralement, il peut exister des **niveaux** d'interruptions : au niveau N, le processeur ne peut être interrompu que par une interruption de niveau plus élevé.

2.2.5 Interruptions logicielles

Enfin, il existe des instructions machine qui, quand elles s'exécutent, déclenchent une interruption. (`SYSCALL`, `SYSENTER`, `TRAP`, `int xxx`, …)

Ces instructions sont utilisées en particulier pour faire des "appels systèmes".

2.2.6 Exemple d'appels systèmes

Pour faire un appel système, le processus demandeur, qui s'exécute en mode normal (non privilégié) :

- met dans des registres les paramètres de l'appel, et le le numéro du service voulu.
- exécute une instruction d'appel (interruption logicielle).

Exemple, en assembleur x86 sous MS-DOS 2.0 16 bits

```
    org  0x100          ; adresse de chargement

    mov  dx,msg          ; adresse chaine
    mov  cx,13           ; longueur chaine
    mov  bx,1            ; sortie standard
    mov  ah,0x40         ; service = "write device"
    int  0x21

    mov  ah,0x4C         ; service = "terminer le programme"
    int  0x21

msg db "Hello, world!"
```

L'exécution de l'instruction `int 0x21` provoque une interruption logicielle qui

- sauve quelques registres sur une pile (PC, registre d'état, …),
- **passe en mode privilégié** ("anneau de protection" 0, sur x86, qui a 4 modes),
- détourne l'exécution vers l'adresse définie pour l'interruption numéro 33 (`0x21` en hexadécimal) dans la table de vecteurs d'interruption. Cette interruption regroupe les services d'entrée-sortie sous DOS.

À partir de là, le système d'exploitation exécute les actions nécessaires (écriture sur l'écran) et relance l'exécution de la tâche interrompue en revenant au mode normal (ou pas, si il s'agissait d'arrêter le programme).

Note : UNIX comportait au départ 80 appels système. les systèmes actuels en ont plusieurs centaines.

2.3 Modes de fonctionnement

2.3.1 Motivation : protection système / processus

Un système d'exploitation fournit des **services** aux programmes qui tournent dessus. Pour le programmeur d'application, ces services simplifient l'accès aux ressources.

Par exemple, quand un programmeur veut faire afficher quelque chose à l'écran, il écrit `System.out.println("Coucou")` sans avoir besoin de connaître le type de carte écran, ou de carte réseau. Le système d'exploitation fournit une **abstraction** du matériel à travers une API.

Mais pour que ça se passe bien, il ne ne fait pas que les programmes utilisateurs puissent agir eux-mêmes directement sur le matériel en contournant l'API.

Les programmes d'application devront passer par des **appels système**, qui donneront la main au système d'exploitation, qui agira sur le matériel après avoir vérifié que ce qu'on lui demande est acceptable.

2.3.2 Modes normal et privilégié

Pour cela, les processeurs[2] ont (au moins) deux modes de fonctionnement[3]

- un **mode normal** pour l'exécution des programmes d'application,
- un **mode privilégié** pour l'exécution du système.

Certaines instructions (accès aux périphériques par exemple) ne peuvent être exécutées qu'en mode privilégié. En mode normal, toute tentative d'exécuter de telles instructions provoquent une exception "instruction illégale".

En mode normal, certaines instructions (`int`, `sysenter`, …) déclenchent une interruption logicielle qui fait repasser en mode privilégié. Elles servent à réaliser des "appels système".

2.4 Protection mémoire (teaser)

Nous verrons plus loin les **mécanismes de protection mémoire** qui font qu'une tâche qui s'exécute en mode normal ne pourra accéder qu'à la partie de la mémoire qui lui est affectée.

Les instructions (et les données) du système d'exploitation sont dans une zone mémoire inaccessible aux programmes "normaux".

C'est l'exécution d'une interruption logicielle qui fait basculer le processeur en mode privilégié, et le dirige vers un "guichet d'entrée", où sont vérifiés les paramètres qui indiquent le service voulu.

3 Résumé

Un système d'exploitation est un ensemble de logiciels destiné à faciliter l'utilisation (l'exploitation) d'une machine par les programmes d'application.

Le système d'exploitation prend en charge l'accès aux ressources matérielles (mémoire, périphériques etc.).

Un système multi-tâches fait tourner plusieurs programmes présents en mémoire simultanément.

Pour tenir ce rôle avec efficacité, des mécanismes matériels ont été introduits dans la conception des processeurs dans les années 50-60 :

- les interruptions,
- les modes de fonctionnement,
- et enfin, des mécanismes de protection mémoire que nous verrons le moment venu.

Ces mécanismes permettent de faire respecter des principes :

1. Le programmeur d'application souhaite écrire des programmes qui fonctionnent sur des machines diverses, avec des périphériques de types différents, etc. Le système d'exploitation fournit une **couche d'abstraction** qui libère le programmeur d'application des détails spécifiques d'une machine. C'est le même code pour lire un fichier sur disque, sur clé USB ou situé sur un serveur.

2. quand ils sont destinés aux systèmes mutitâches. Ce n'est pas le cas des micro-contrôleurs des cartes Arduino "de base".
3. Sur l'architecture x86 des PC, il y a 4 modes, appelés "anneaux de protection" (*rings*). La plupart des systèmes d'exploitation n'en utilisent que 2.

2. Les programmes d'application n'ont accès qu'aux ressources (mémoire, fichiers, …) dont ils ont strictement besoin. Le **cloisonnement** restreint les conséquences d'un "comportement inattendu".
3. En raison du **principe de moindre privilège**, les programmes d'application s'exécutent dans un mode non-privilégié. Ils invoquent des **appels système** qui donnent la main au noyau du système d'exploitation, qui tourne en mode privilégié, et - après vérification de la légitimité des demandes - exécuté pour eux l'action demandée.

Lecture : https://www.ssi.gouv.fr/guide/recommandations-pour-la-mise-en-place-de-cloisonnement-systeme/

4 Glossaire

- **Appel système** : séquence d'instructions exécutée par un programme d'application pour demander au système d'accomplir une certaine fonctionnalité. Charge les paramètres et le numéro de fonctionnalité dans des registres, et exécute une interruption logicielle spécifique.
- **Exception** (*trap*, *fault*) : interruption synchrone déclenchée par une condition exceptionnelle. Exemple *exception arithmétique* en cas de division par zéro, passage sur un point d'arrêt (breakpoint), …
- **Interruption**, au sens large, évènement qui provoque une suspension du cycle normal d'exécution des instructions dans un processeur. L'état interne du processeur est partiellement sauvé dans une pile système, et l'exécution est détournée vers une *routine de traitement* de l'interruption.
- **Interruption logicielle** (IRQ = interrupt request), est déclenchée par l'exécution d'une instruction spéciale (`INT`, `SYSENTER`, …)
- **Interruption matérielle**, asynchrone, est émise par un périphérique.
- **Masquage** : possibilité, en mode privilégié, d'ignorer provisoirement l'arrivée de certaines interruptions, qui seront prises en compte quand elles sertont *démasquées**.
- **Mode normal** : dans ce mode (destiné aux programmes "normaux") le processeur ne peut utiliser que certains instructions.
- **Mode privilégié** : dans ce mode (en principe réservé pour le système) le processeur peut exécuter toutes les instructions existantes.
- **Privilège** : permet à un composant logiciel (système, programme d'application) qui en dispose de mener légitimement à bien une action sur une ressource.
- **Routine de traitement d'interruption** : séquence d'instructions à exécuter lorsqu'une interruption spécifique se produit.
- **Vecteur d'interruption** : structure qui décrit l'action à effectuer lorsqu'une interruption est traitée : adresse du code, etc. Dans sa forme la plus simple, contient une simple instruction : retour d'interruption ou branchement vers une routine de traitement.