Chapitre 2 : Noyau et appels systèmes

Systèmes d'exploitation

Auteur: Pierre-Antoine Champin
Adresse: Département Informatique - IUT - Lyon1
Date: 2011

Plan du chapitre

1 Problématique et rappels

- Les trois premières fonctions du sysème d'exploitation (aide, abstraction, augemntation) pourraient être rendues par une **bibliothèque** de fonctions standard, liée aux applications au moment de la compilation.
- Les applications pourraient, le cas échéant, *ne pas passer* par cette bibliothèque.
- Incompatible avec les fonctions d'*arbitrage* et de *autorisation*

Pré-requis

- Il est donc nécessaire d'*empêcher* les applications d'accéder aux ressources sans passer par le SE.
- La même instruction processeur doit donc avoir un comportement différent selon que c'est le système d'exploitation ou une application qui cherche à l'exécuter.

  $\rightarrow$ contrainte forte sur le matériel

Rappel 1 : modes d'exécution

- cf. cours d'Architecture des ordinateurs
- Le processeur plusieurs **modes d'exécution**, au minimum un mode *superviseur* et un mode *utilisateur*.
- D'autres modes peuvent exister, selon les plateformes :
  - différents « niveaux » utilisateur ou superviseur
  - mode hyperviseur pour la virtualisation

Mode superviseur (également appelé mode noyau) :

- Toutes les instructions sont autorisées.

  → seul le SE doit y avoir accès

Mode utilisateur

- Certaines instructions sont interdites ou limitées.

  (par exemple: plage d'adresses mémoire autorisées)

- L'utilisation d'une instruction interdite déclenche une *erreur*.

  → les applications doivent toujours s'exécuter dans ce mode

Rappel 2 : interruptions

Certains événements (**interruption**) entrainent un comportement particulier du processeur.

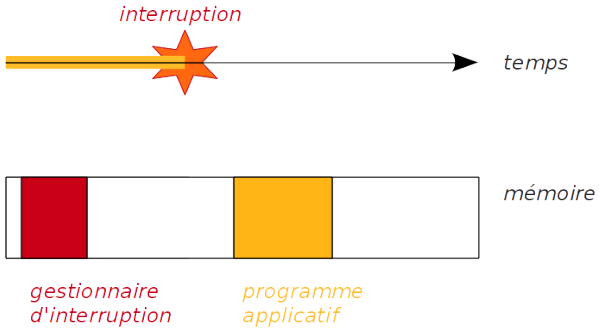
Ces événènemenets peuvent être de différents types:

- erreurs logicielles (*e.g.* division par zéro)
- interruptions matérielles (selon le périphérique)
- appel système (cf. ci-après)

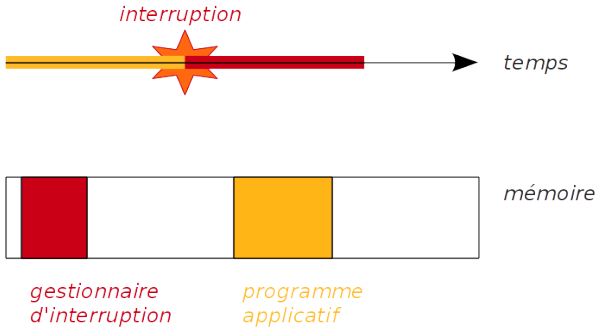
Lorsqu'une interruption se produit :

- selon son type, le processeur choisi le bon **gestionnaire d'interruption**
- *il passe en mode superviseur*
- il sauvegarde le contexte (état des registres) du programme courant
- il exécute le gestionnaire d'interruption
- il restaure le contexte sauvegardé et reprend le programme courant (dans le bon mode d'exécution)

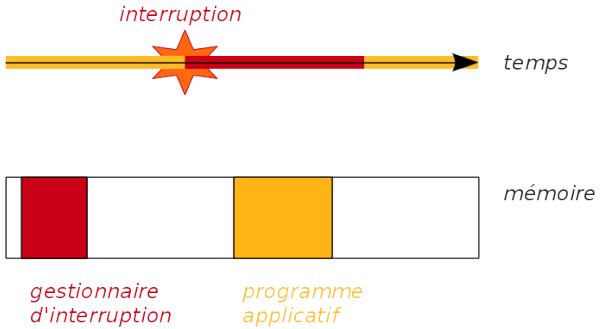
Rappel 2 : interruptions



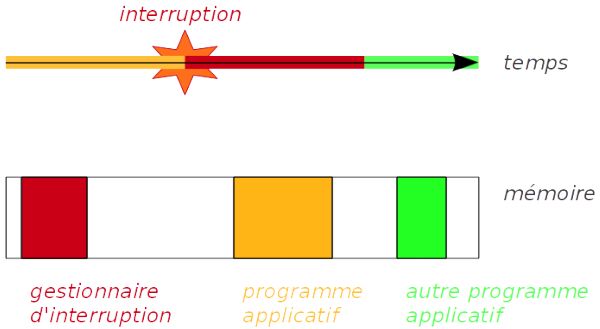
Rappel 2 : interruptions



Rappel 2 : interruptions



Rappel 2 : interruptions et erreurs



2 Fonctionnement général

- Le **noyau** du système d'exploitation est un *ensemble de gestionnaire d'interruptions*, chargés en mémoire au démarrage du système.
- Lorsqu'une application, qui s'exécute en mode utilisateur, souhaite accéder à une ressource, ou plus généralement invoquer une fonction du SE, elle procède à un **appel système**.

Démarrage du sysème

- Au démarrage, les gestionnaires d'interruption qui consistituent le noyeau sont chargés en mémoire aux emplacements définis.
- Le noyau passe ensuite la main à un programme applicatif, en *mode utilisateur* (exemple : shell, environnement graphique).
- Le noyau « attend » ensuite les interruptions pour s'exécuter.

Déroulement d'un appel système

- Le programme applicatif décrit le service souhaité en renseignant des registres dédiés.
- Il déclenche une interruption d'un type particulier, ce qui passe la main au noyau.
- Ce dernier exécute (en mode superviseur) la fonction requise (ou refuse, le cas échéant), et inscrit le résultat dans un registre dédié.
- Il rend la main au programme de l'application, en *mode utilisateur*.

Bibliothèque système

- Le mécanisme d'appel système est généralement encapsulé par une bibliothèque.
- Les appels systèmes se présentent donc comme des fonctions/procédures classiques.
- ⚠ Cependant, elles cachent un coût supérieur à des appels de fonction classiques (interruption logicielle, changement de mode)

  → parfois nécessaire de minimiser le nombre d'appels systèmes pour des raisons d'optimisation

Discussion

Ce mécanisme permet bien au SE de remplir ses fonctions d'arbitrage et d'autorisation :

- L'accès aux ressources (périphériques, fichiers, etc...) passe forcément par les appels système, et est donc soumis au **contrôle** du système d'exploitation.
- Toute tentative d'accéder aux ressources sans passer par les appels systèmes déclenchera une interruption de type « insrtuction interdite », qui renvoie également au noyau (gestionnaire d'interruption).

3 Illustration : Entrées/sorties

```
fd = open("mon_fichier.txt", O_WRONLY);
write(fd, "hello world\n", 12);
close(fd);
```

Références :

```
int open (const char *path, int oflag, [int mode]);
int read (int fildes, void *buf, int nbyte);
int write(int fildes, void *buf, int nbyte);
int close(int fildes)
```

Gestion des erreurs (1)

```
fd = open("mon_fichier.txt", O_WONLY);
if (fd == -1) exit(-1);
status = write(fd, "hello world\n", 12);
if (status == -1) exit(-1);
status = close(fd);
if (status == -1) exit(-1);
return 0
```

Au minimum, interrompre le programme en cas d'erreur inattendue.

Gestion des erreurs (2)

```
fd = open("fichier.txt", O_WRONLY);
if (fd == -1) {
    if      (errno == EACCES) exit(-1); // accès interdit
    else if (errno == ENOENT) exit(-2); // fichier inexistant
    else                      exit(-3); // autre problème
}
status = write(fd, "hello world\n", 12);
if (status == -1) exit(-4); // problème d'écriture
close(fd);
if (status == -1) {
    printf("Attention: problème à la fermeture\n");
} // mais on termine normalement malgré tout
return 0;
```

Discussion (1)

- Ce niveau de granularité est encore assez complexe : les bibliothèques systèmes sont souvent elle-même encapsulées dans des bibliothèques standard.

  Exemples :

  - en C : `fopen`, `fprintf`
  - en C++ : `ofstream`, `<<`
- Les langages offrant un mécanisme d'**exceptions** permettent notamment une gestion plus élégante des erreurs.

Gestion des erreurs (3)

```
try:
    f = open("fichier.txt", "w")
    f.write("hello world\n")
    f.close()
except:
    print "une erreur s'est produite"
    exit(-1)
```

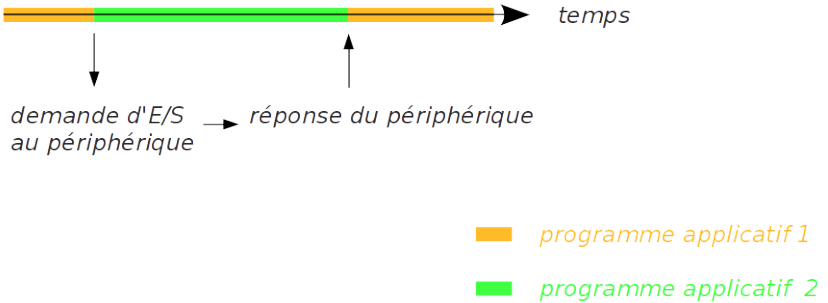
Discussion (2)

- Les entrées/sorties prennent beaucoup plus de temps que les instructions de calcul du processeur.
- Dans le programme précédent, on attend que l'instruction qui suit le `write` s'exécute une fois ce dernier terminé.
- Dans l'intervalle, un autre programme peut en profiter pour s'exécuter.

4 Temps partagé

Le principe du **temps partagé** consiste à mettre à profit les temps de latences liés aux entrées/sorties pour exécuter plusieurs programmes en parallèle.

Principe



- Le programme applicatif 1 n'est pas du tout pénalisé.
- Le programme applicatif 2 peut démarrer plus tôt.

  → gagnant-gagnant

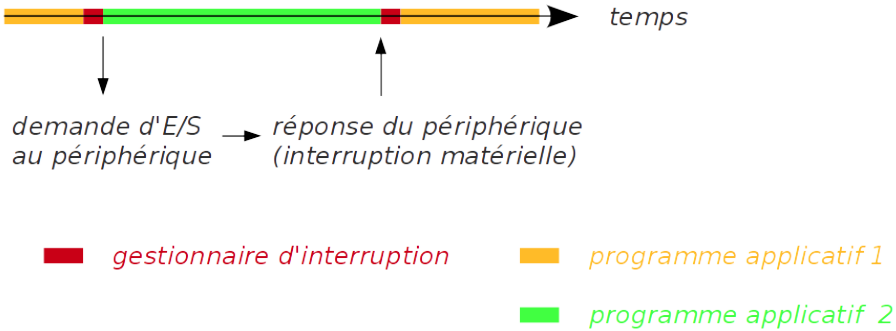
Rappel 3 : interruption matérielle

Les périphériques utilisent des **interuptions matérielles** pour notifier le processeur d'un événement

- directement sollicité (*e.g.* demande de lecture ou d'écriture sur un disque dur)
- ou non (*e.g.* données disponibles depuis le réseau, le clavier, la souris...).

→ Le noyau a donc la main au début et à la fin de chaque opération d'entrées/sorties.

Mise en œuvre du temps partagé



À la fin de chaque gestionnaire d'interruption, le système d'exploitation décide à quel processus il va rendre la main.

Ordonnanceur

- L'**ordonnanceur** (en anglais *scheduler*) est la partie du système d'exploitation qui est chargé de l'arbitrage du processeur.
- Il est invoqué à la fin de chaque gestionnaire d'interruption, afin de déterminer à quel programme il faut ensuite passer la main.
- Son implémentation doit permettre un compromis entre le surcoût et un certain nombre de critères qui dépendent du type de SE et des *souhaits* de l'administrateur système.

Définitions

Équité

propriété d'un ordonnanceur garantissant à tous les processus les même chances d'obtenir le processeur

Famine

situation dans laquelle un processus attend indéfiniment le processeur

NB : les performances d'un ordonnanceur se mesurent différemment selon le type de système (système batch, système interactif)

Remarque

Il existe un cas ou un processus peut « échapper » au contrôle du système d'exploitation : si il ne fait jamais d'appel système.

Multi-tâche coopératif

Appel système dédié `yield` : le programme cède simplement la main.

Avantage

surcoût faible, adapté sur des machines peu puissantes

$\rightarrow$ MacOS < 10, Windows < 95/NT

Inconvénient

dépend de la bonne volonté du programmeur ; inacceptable dans un contexte multi-utilisateur

Multi-tâche préemptif

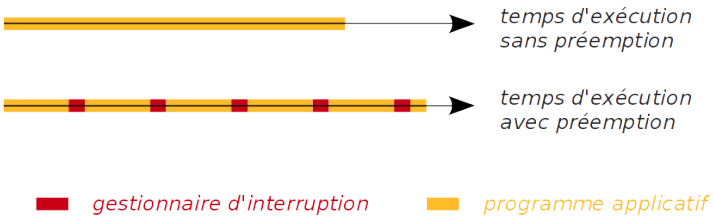
Un périphérique dédié, l'*horloge*, émet une interruption à intervalle régulier.

Avantage

garantit un arbitrage régulier par *préemption*

Inconvénient

surcoût en temps de calcul


temps d'exécution
sans préemption
temps d'exécution
avec préemption
gestionnaire d'interruption
programme applicatif

En résumé

Contrôle sur les ressources

- Obligation matérielle pour les applications de passer par le SE (mode utilisateur)
- Noyau, Appel système

Arbitrage du processeur

- Ordonnanceur
- Interuption d'horloges pour permettre la préemption par le SE

Annexe : micro-noyau

- Objectif : augmenter la stabilité et la sécurité du système en réduisant la quantité de code qui s'exécute en mode superviseur
- Un micro-noyau a donc des fonctionalités minimales, principalement la communication inter-processus, et implémente toutes les autres fonctions sous forme de services s'exécutant en mode utilisateur.