Fonctionnement des systèmes d'exploitation

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

22 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 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

```c
#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()`.

1.2 Tâches et processus

On appelle **processus**, ou **tâche** un programme en cours d'exécution sous le contrôle du système d'exploitation.

Ce cours s'intéresse aux systèmes **multi-tâches** en temps partagé qui font tourner plusieurs processus présents en mémoire en même temps. De nos jours, ces systèmes multi-tâches sont présents dans tous les ordinateurs grands et petits (smartphones).[1]

Le système d'exploitation gère les ressources matérielles

- temps accordé aux processus,
- l'espace mémoire accordé aux processus,
- périphériques d'entrée-sortie,
- périphériques de stockage,
- etc.

1.3 Problématique des systèmes multi-tâches

Un système multi-tâches "fait tourner" plusieurs processus à la fois, même sur une machine qui n'a qu'un seul processeur, grâce la technique de "temps partagé" (*time slicing*)

Pour cela, le système "fait travailler" une tâche pendant un petit laps de temps[2], puis en fait travailler une autre, etc. Le temps étant ainsi partagé, le travail des tâches avance régulièrement en donnant (à notre échelle) une *illusion* d'avancement simultané.

(Plus de détails plus loin)

Il se pose alors plusieurs problèmes :

- empêcher les processus d'utiliser directement les périphériques
- partager équitablement le temps
- partager la mémoire disponible entre les processus, pour éviter que l'un accède à l'espace mémoire de l'autre

2 Mécanismes matériels

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

1. Les systèmes mono-tâches existent dans les petits dispositifs d'informatique embarquée qui ne font qu'une chose à la fois. Par exemple une centrale météo qui relève la température, la pression, etc. et transmet les données périodiquement à un serveur.
2. Valeurs typiques : 20-100 ms.

Par exemple, dans une mémoire de quelques milliers d'octets on ne peut charger et faire exécuter qu'un programme à la fois. Mais avec une mémoire plus grande, on peut envisager d'avoir plusieurs tâches présentes en mémoire. L'utilisation de périphériques (lents) par une tâche laisse des temps morts pendant lesquels la tâche est bloqué en attente du résultat, mais ce temps mort peut être mis à profit pour faire avancer une autre tâche. Ainsi, on rentabilise mieux le temps d'utilisation de la machine.

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

Mais il faut aussi modifier l'ordinateur pour ajouter des mécanismes de protection qui empêchent que les tâches accèdent pas directement (par accident ou malveillance) aux périphériques, à la mémoire du système et des autres tâches etc.

2.1 Fonctionnement d'un processeur, idée de base

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, à l'adresse indiquée par le PC, l'instruction à exécuter,
- exécuter cette instruction,
- 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 Interruptions

2.2.1 Traitement des exceptions

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

Dans l'UNIVAC 1 (1951), une telle exception provoque la levée d'un signal électrique. En présence de ce signal, la valeur 0 est chargée dans le PC, et le programme se continue donc à l'adresse 0 où se trouve le code à exécuter en cas d'exception arithmétique.

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

2.2.2 Reprise

Pour reprendre l'exécution à l'endroit où le travail a été interrompu, la routine de traitement exécute une instruction "retour d'interruption". Il faut pour cela que que le PC ait été sauvegardé lors de l'interruption, soit dans un registre spécial, soit dans une pile.

2.2.3 Interruptions matérielles

Les **interruptions matérielles** sont déclenchées par des signaux extérieurs (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, …), et sont prises en compte de la même façon.

Ces signaux sont centralisés par un circuit "contrôleur d'interruptions", qui transmettra un numéro d'interruption. Ce numéro permettra au processeur de trouver l'adresse du code à exécuter dans un "vecteur d'interruption".

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 ne sera prise en compte que quand la RTI "démasquera" les interruptions.

Plus généralement, il peut exister des niveaux d'interruption : lorsque le processeur est au niveau N, il 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".

2.3 Modes de fonctionnement

2.3.1 Protection système / processus

Dans un ordinateur multi-tâches, on doit **empêcher** les programmes d'application d'accéder directement aux périphériques et à d'autres emplacements mémoire.

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 ont (au moins) deux modes de fonctionnement[3]

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.

- 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".

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.

3 Séparation système / programmes d'application

L'existence de deux modes permet donc de définir une séparation claire entre les programmes d'application, et le système d'exploitation.

Le système d'exploitation fournit des **services** aux programmes d'application, et est le point de passage obligé pour accéder à ces services.

3.1 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

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 le vecteur 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).

3.2 Programmation dans un langage de haut niveau

Pour pouvoir faire tourner des programmes écrits dans des langages de haut niveau, il faut que les compilateurs convertissent les appels aux "fonctions système" en appels à une bibliothèque "runtime" d'exécution, qui fera les appels systèmes en se conformant à l'**ABI** (*Application Binary Interface*) du système, c'est-à-dire aux conventions fixées pour les numéros de service, les registres utilisés etc.

4 Le fonctionnement multi-tâches : partage du temps

La notion d'interruption permet de voir l'ordinateur comme un "système réactif", dont l'état évolue par exécution des instructions, mais qui répond aussi à l'arrivée d'évènements extérieurs.

4.1 Table des processus

La table des processus est une structure de données centrale d'un système d'exploitation.

Elle contient la description des divers processus présents en mémoire, avec pour chacun :

- son état (en cours d'exécution ? bloqué ?…)
- un bloc de contexte qui sauvegarde le contenu des registres,
- une description de l'espace mémoire qu'il utilise,
- son propriétaire, les droits dont il dispose,
- etc.

4.2 États d'un processus

Situation banale : vous avez plusieurs fenêtre ouvertes à l'écran. Dans un navigateur web, vous cliquez sur un lien, qui tarde à répondre. Vous en profitez pour continuer à faire autre chose dans une autre fenêtre.

Que s'est-il passé ?

- En cliquant sur le lien, le navigateur a envoyé (via la couche réseau) une requête HTTP, et il s'est mis en attente d'une réponse.
- Mais il ne passe pas son temps à surveiller activement l'arrivée de cette réponse.

- Au contraire, le navigateur (ou du moins le processus qui s'occupe de l'onglet) est **bloqué** jusqu'à l'arrivée de la réponse (ou l'expiration d'un délai jugé trop long).

Maintenant, le système peut activer une des autres tâches présentes.

Que se passe-t-il ensuite ?

- Lorsque la carte réseau reçoit un paquet, elle signale au système d'exploitation qu'une trame est arrivée est arrivé.
- le système d'exploitation (la pile TCP/IP) examine cette trame, en extrait un paquet IP, constate que c'est un paquet TCP, et qu'il fait partie d'une connexion ouverte par le navigateur.
- le contenu du paquet est donc transmis au navigateur web, qui est débloqué, et pourra alors traiter la réponse.

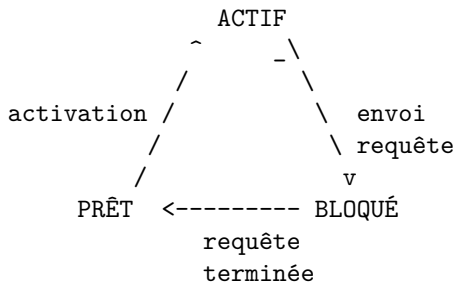
4.3 États et transitions

On voit donc 3 états possibles pour les processus

- état **actif** : le processeur est en train d'exécuter une des instructions de la tâche,
- état **bloqué** : la tâche ne peut pas être activée, elle attend un événement pour pouvoir continuer,
- état **prêt** : la tâche n'est pas bloquée, elle pourrait être activée.

Des changements d'état ont lieu, sous le contrôle du système d'exploitation :

1. Un processus **actif** appelle un service qui prend du temps (par exemple, aller chercher des données sur un disque). Le processus est alors mis à l'état **bloqué**.
2. Lorsque la réponse arrive (interruption), le processus demandeur qui est **bloqué** est mis à l'état **prêt** (il se peut aussi que le processus qui est **actif** à ce moment là soit mis à l'état **prêt**).
3. Quand un processeur/coeur est libre, le système choisit un des processus **prêts** pour le rendre **actif**.



Ceci entraîne des **commutations de contexte**

- quand on processus **sort** de l'état actif, dans lequel un processeur exécutait ses instructions, le contenu des registres du processeur est sauvé dans le "bloc de contexte" du processus

Quand un processus est **activé**, les informations du bloc de contexte sont placées dans les registres, et l'exécution reprend là où elle en était arrêtée.

4.4 Exercices

TODO.

Reprendre quelques classiques, pour montrer

- qu'on peut profiter des temps morts d'une tâche pour en faire avancer une autre
- qu'on y gagne en rendement (par rapport à l'exécution des tâches en séquence)

par exemple tâches qui font quelques cycles calcul/entrée sortie.

Voir aussi cohabitation de tâches "IO-bound" et "CPU-bound".

4.5 Temps partagé

Avec le système expliqué ci-dessus, si un processus ne fait que du calcul et aucun appel système (par exemple parce que le programmeur a fait une boucle sans fin), il reste actif et donc monopolise la machine indéfiniment.

On souhaite évidemment éviter cette situation, que l'on retrouvait dans les systèmes d'exploitation à "**ordonnancement coopératif**" (MacOs jusqu'à la version 9, Windows jusqu'à 3.11)„ dans lesquels on compte sur la bonne volonté et la compétence des programmeurs d'application pour que tout se passe bien.

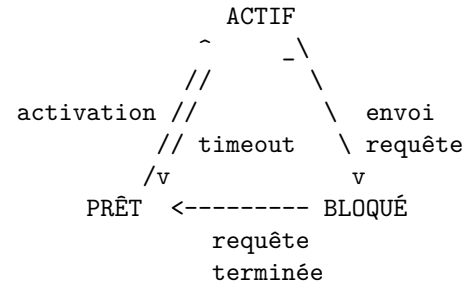
4.5.1 Utilisation d'un timer

La solution est d'utiliser un **timer** (circuit d'horloge) qui émet une interruption au bout d'un certain temps (typiquement 20-100 ms).

Ce timer

- est armé (programmé pour un certain délai) quand un processus est activé
- interrompt le processus actif à échéance du délai (*timeout*).

Le processus actif est alors mis à l'état **prêt**, et un autre processus prêt est choisi pour être activé.



Ainsi, on évite que le "temps de processeur" soit monopolisé par un programme qui boucle (ou qui calcule très longtemps).

4.5.2 Choix du quantum de temps

On appelle "quantum" la durée accordée à un processus actif avant qu'il soit interrompu par le timer. Compromis à trouver :

- si il est long, les processus qui “mangent du temps de calcul” vont “charger” la machine et laisser aux autres processus peu d’occasions d’être activés.
- si il est court, le système d’exploitation sera sollicité plus souvent, et consommera une part plus importante du temps. Les commutations de contexte ont un coût….

Dans la mesure où c’est très dépend de ce qu’on donne à faire à la machine, solution pragmatique : essayer, mesurer, ajuster.

4.6 Politiques d’ordonnancement

Il reste un petit détail à régler concernant l’**ordonnancement**, c’est-à-dire la manière de choisir un processus prêt pour l’activer, sachant qu’il peut y en avoir plusieurs.

On peut imaginer plusieurs façons de faire. On les compare sur des critères

- de sûreté (le fait que tous les processus avancent),
- d’équitabilité,
- d’efficacité (profiter au maximum des ressources),
- …

qui dans la réalité sont évidemment contradictoires.

4.6.1 Tourniquet (*first-in, first-out*)

- Les processus prêts forment une file d’attente.
- Choix du processus le plus ancien de la file.

Sûr et équitable, mais ne permet pas d’avoir des processus plus prioritaires que d’autres.

4.6.2 Niveaux de priorité fixes

- On affecte à chaque processus un niveau de priorité. À chaque niveau correspond une file d’attente.
- On active le processus le plus ancien de la file la plus prioritaire.

Efficace, mais risque : si les processus du niveau le plus élevé bouclent, situation de **monopole** ou de **coalition** qui empêche les autres processus de s’exécuter.

4.6.3 Niveaux de priorité variable

- la priorité d’un processus varie au cours du temps. La priorité d’un processus qui arrive au bout de son quantum baisse. Elle remonte quand il demande des entrées-sorties.

Les tâches courtes sont favorisées, ce qui est agréable pour l’utilisation interactive. Quand un calcul se met à durer, il devient moins prioritaires.

Problème : deux processus qui bouclent en communiquant par un pipe conservent une priorité élevée.

4.6.4 Classes de processus

(Multi-level feedback queues)

- On définit plusieurs files d'attente.
- On choisit "aléatoirement" une file dont on active le processus le plus ancien.

En pondérant (par exemple files choisies à 70%, 20% et 10%) on favorise certains processus, sans risque de monopole.

5 Le partage de l'espace

Dans la mémoire de l'ordinateur vont se retrouver le code et les données des processus, ainsi que du système d'exploitation.

Comment se répartir cet espace, et surtout le protéger contre les erreurs des programmes ?

5.1 Linéaire

Dans un système mono-tâche, le système d'exploitation est chargé au démarrage (typiquement au fond de la mémoire), et y reste jusqu'à l'arrêt de la machine. Et les programmes d'applications sont ensuite chargés au début de l'espace restant.

En passant à un système multi-tâches, on peut imaginer placer les programmes d'applications les uns après les autres. Chaque programme occupera une plage d'adresses (adresse de début / adresse de fin)

5.1.1 Allocation/libération

Comme les programmes apparaissent et disparaissent au gré des chargements et fin d'exécution, le système tient une comptabilité des espaces disponibles.

Le chargement d'un programme sera possible uniquement si on peut lui allouer un espace suffisamment grand pour le loger. Mais il ne suffit pas que le total des espaces libre excède la taille du programme, il faut aussi que cet espace soit contigu (en un seul morceau),

Exemple : on a chargé successivement des programmes A, B, C, D de tailles respectives 20, 10, 20, 30 KiB dans une mémoire de 64 KiB. Les programmes A et C se terminent, ce qui laisse 40 KiB octets disponibles. Mais on ne pourra pas lancer un programme de 25KiB, parce que le plus grand bloc libre est de 20 KiB seulement. La mémoire est trop **fragmentée**.

Le **mécanisme** d'**allocation** consiste à chercher un bloc assez grand pour y loger le programme. Diverses stratégies sont possibles :

- la plus simple **first fit** consiste à regarder les blocs libres dans l'ordre des adresses, et de prendre le premier qui soit assez grand. Ce bloc libre est découpé, entre la partie qui est allouée, et le reste qui est remis dans la liste de blocs libres.

- la stratégie **best fit** est légèrement meilleure. Elle consiste à prendre le plus petit bloc qui soit assez grand, ce qui évite d'entamer de grands blocs quand on peut faire avec les petits.

Dans tous les cas, lors de la **libération**, le bloc qui se libère est fusionné si possible avec les blocs libres voisins pour en former un plus gros.

Exercice : La stratégie *best-fit* est une heuristique[4] qui ne résout pas complètement le problème. Trouvez des scénarios (suite d'allocations et de libérations) pour lesquels

1. *first-fit* réussit, alors que *best-fit* échoue,
2. l'inverse.

5.1.2 Une idée pour la protection : registres limite

Lorsqu'un programme d'application s'exécute, le matériel doit l'empêcher d'accéder à autre chose que son propre espace mémoire.

Une solution simple est d'équiper le processeur de "registres limite" dans lesquels le système mettra les adresses de début et de fin de cet espace. Des comparateurs[5] connectés aux registres et au bus d'adresse détecteront toute tentative d'accéder en dehors de cette plage, et déclencheront une interruption "accès mémoire illégal".

Ces registres ne sont manipulables qu'en mode privilégié.

En pratique, cette idée a été peu utilisée. En effet, a priori les programmes d'applications peuvent être chargés n'importe où, en fonction de ce qui a été chargé avant. Or les programmes contiennent des instructions de branchements, avec des adresses calculées à la compilation. Il devient compliqué de charger des programmes qui contiennent des adresses absolues. De même si on veut les déplacer pour "dé-fragmenter" la mémoire.

5.1.3 Autre idée : espace logique

Une idée plus intéressante est de considérer que chaque processus dispose d'un espace mémoire, qu'il voit à travers des "**adresses logiques**" qui - pour lui- commencent à 0.

Cet espace logique correspond à une plage d'**adresses physiques**. En additionnant une adresse logique et l'adresse de début de la plage, on obtient l'adresse physique correspondante.

Dans le processeur on intègre quelques circuits pour effectuer la **génération d'adresses** physiques :

- un registre "base" qui contient l'adresse physique de début de la plage,
- un additionneur qui calcule une adresse physique en ajoutant le contenu de ce registre les adresses logiques produites par le programme en cours d'exécution.

et pour la **protection** :

4. technique susceptible de fournir une solution approchée à un problème (mais pas toujours).
5. Un comparateur est un soustracteur, dont on utilise le signe du résultat.

- un registre qui contient la taille de la plage,
- un comparateur entre ce registre et l'adresse logique, qui lève une interruption en cas de débordement.

Ceci introduit une distinction conceptuelle entre deux espaces d'adressage :

- "espace logique", avec des adresses logiques qui vont de 0 à taille-1. Ces adresses logiques sont internes au processeur.
- "espace physique", avec des adresses physiques qui vont de base à base + taille - 1. Ces adresses sont utilisées pour les accès à la mémoire.

5.1.4 Déplacement des programmes en mémoire

Cette indépendance permet au système d'exploitation de déplacer un programme, en copiant ailleurs son "image mémoire" et en faisant pointer le registre de base vers le nouvel emplacement. C'est une stratégie curative pour le problème de **fragmentation**.

Malheureusement, c'est une opération qui prend du temps.

Et on doit aussi le faire quand un programme demande à disposer de plus d'espace mémoire pendant son exécution[6], alors que son espace mémoire est suivi par un autre espace occupé.

5.2 Espace paginé

Une approche radicalement différente apporte une solution à ces problèmes, et conduira à la notion de "mémoire virtuelle" (voir plus loin).

5.2.1 Pages et cadres

Découpage de l'espace logique : L'espace logique d'un processus est maintenant considéré comme une succession de "pages" de même taille (une puissance de 2, dépendant de l'architecture de la machine).

Par exemple, sur une machines à pages de 4Kib ($2^12 = 4094$), un processus de 10354 octets occupe 3 pages. La page 0 correspond aux adresses logiques 0 à 4095, la page 1 aux adresses de 4096 à 8191, et la page 2, qui n'est pas complètement occupée, de 8192 à 12287.

Calculer le calcul du numéro de page correspondant à une adresse logique n'est pas compliqué : c'est le quotient de l'adresse logique (par exemple 9876) par la taille de page (4096), soit 2. Et le reste donne l'**offset** (posittion dans la page).

Aucun circuit de calcul n'est nécessaire : l'offset est dans les 12 bits de droite de l'adresse, le numéro de page dans les bits de gauche.

```
                          binaire            décimal
--------------------  ------------------    ----------
              adresse  0010 0110 1001 0100   = 9876
       numéro de page  0010                  = 2
position dans la page        0110 1001 0100   = 1688
```

6. par exemple parce qu'ils font de l'**allocation dynamique** (instruction `new` en C++, Java, etc).

L'espace d'adressage physique est, de la même façon, découpé en "cadres" (de page) la même taille[7].

Bien évidemment, les pages logiques correspondront à des cadres.

5.2.2 Table de pages, génération d'adresses

La correspondance est assurée par un groupe de registres appelé **table des pages**, qui fait partie de la MMU (**memory management unit**), un composant du processeur.

Ces registres établissent une correspondance entre

- le numéro de page extrait d'une adresse logique,
- le numéro du cadre qui contient cette page.

Exemple, avec une table des pages qui contient

page	cadre
0	4
1	10
2	3

l'adresse logique 5100, qui est à la position 1006 de la page 1, (parce que $5100 = 1 \times 4096 + 1006$) se trouve en mémoire à la position 1006 du cadre 10 (qui correspond à la page 1), soit l'adresse physique $10 \times 4096 + 1006 = 41966$.

La table est consultée à chaque accès à la mémoire pour générer l'adresse physique. Pour un accès rapide (en temps élémentaire), elle utilise un indexage matériel (multiplexage, registres associatifs…).

La table est chargée par le système d'exploitation quand un processus est activé. Dans la table des processus, on trouve donc une copie de la table des pages du processus.

5.2.3 Possibilité de partage

Les tables de page permettent d'avoir des espaces mémoires communs à plusieurs processus. Exemple, avec les tables de pages ci-dessous, les processus P1 et P2 ont tous deux accès aux octets du cadre n°6,

```
P1      P2
---     ---
0 4     0 1
1 2     1 0
2 6     2 3
        3 6
```

mais ils ne le voient pas avec les mêmes adresses logiques.

7. pour éviter les confusions, on parle de "page" pour les adresses logiques, et de "cadre" pour les adresses physiques.

Note : les droits d'accès à des pages communes peuvent être différents. Pour faire respecter ces droits d'accès, la MMU contiendra aussi des indicateurs de permissions (lire, modifier, faire exécuter).

6 Mémoire virtuelle

Si on observe ce qui se passe dans la machine, on constate qu'en fait la plupart des pages qui ont été chargées ne sont pas "actives". Par exemple, le début d'un programme a été chargé et exécuté, et on ne revient pas dessus ensuite.

Si on pouvait s'en débarasser, ça permettrait de charger d'autres programmes, et donc de mieux rentabiliser l'ordinateur en faisant plus de choses.

Mais a priori, on ne sait pas si une page qui est inactive à un moment donné va resservir dans le futur ou pas. Une idée naturelle est donc de les sauvegarder sur disque[8], pour pouvoir les récupérer au besoin.

Ceci conduit à la notion de **mémoire virtuelle**, qui contient à la fois les pages présentes en mémoire centrale (**mémoire réelle**), et celles qui ont été sauvegardées sur disque.

Nous allons voir maintenant comment la conjonction du matériel et du système d'exploitation permet de donner aux programmes l'illusion qu'ils fonctionnent comme en mémoire, mais dans un espace qui est beaucoup plus grand.

6.1 Principe

Matériel

- la MMU contient une table des pages **présentes en mémoire réelle**,
- quand le programme utilise une page dont le numéro n'y figure pas, la MMU produit une interruption "défaut de page"[9], ce qui active une routine du système d'exploitation.

Le système d'exploitation

- bloque alors le processus qui était en cours d'exécution,
- trouve un cadre de page disponible,
- va chercher la page manquante sur le disque

Quand cette page finit par arriver (le disque est un périphérique relativement lent) :

- le système copie la page dans ce cadre,
- ajuste la table des pages du processus,
- et le remet à l'état prêt.

Le principe est donc relativement simple. La difficulté est de trouver un cadre de page disponible. Quand toute la mémoire est occupée, il faut "sortir" une page présente (paging out) pour pouvoir ramener (paging in) celle dont le processus a besoin.

8. Une ressource plus abondante, moins chère, mais aussi d'accès plus lent que la mémoire centrale (on ne peut pas tout avoir).
9. dans le sens de "faire défaut", c-a-d manquer.

6.2 Éviction : critères

Les performances de la machine dépendront de la qualité de l'**algorithme d'éviction** utilisé pour choisir la page présente que l'on va remplacer.

Les facteurs qui rentrent en compte

- Il faut éviter d'évincer une page dont on va avoir bientôt besoin,
- On ne peut pas prédire l'avenir, mais il est probable qu'une page récemment utilisée le sera encore prochainement (principe de localité).
- Sauvegarder une page sur disque prend du temps.
- Si une page avait été chargée depuis le disque, et n'a pas été modifiée depuis en mémoire réelle, il est inutile de la sauvegarder.

Et donc : les pages qui n'ont pas servi depuis longtemps et/ou qui n'ont pas été modifiées depuis longtemps sont de "bons candidats" pour l'éviction.

6.3 Déterminer l'âge des pages, stratégie LRU

Une petite modification de la MMU permet de connaître les pages qui n'ont pas été utilisées depuis longtemps.

Un indicateur A est ajouté pour chaque page, il indique si la page a été accédée.

- le bit A est mis à 0 au départ,
- il passe à 1 quand il y a un accès en lecture ou écriture.

Et dans les tables du système, un entier est aussi ajouté pour coder l'"âge" de chaque page :

- l'âge est initialisé à 0 quand la page est chargé en mémoire réelle,
- Périodiquement, le système interroge la MMU. Si le bit A d'une page est à 0, son âge est incrémenté, sinon il revient à 0. Les bits A sont tous remis à 0 ensuite.

Ceci permet de mettre en place la stratégie dite "LRU" (least recently used) où on choisit d'évincer une des pages les plus âgées, qui n'a pas servi depuis longtemps.

6.4 Trouver les pages non-modifiées

Dans la MMU, on ajoute un bit M pour chaque page.

- ce bit est mis à 0 lors du chargement,
- il passe à 1 chaque fois que le processus modifie quelque chose dans la page.

Le système interrogera la MMU pour récupérer la liste des pages modifiées.

En prenant en compte cette information, on obtient la stratégie du type

> choisir une des pages les plus âgées, non modifiée de préférence

Exercice : Dans la littérature, on trouve une version de cette stratégie sous le nom "NRU" (not r&cently used). Elle n'utilise que les bits A et M, et pas l'âge. Ces bits définissent 4 classes de pages. Dans quelle ordre sont-elles choisies ?