Notes du cours de système d'exploitation (SYS2)

Nathan Furnal

12 mars 2021

Table des matières

1 Système d'exploitation

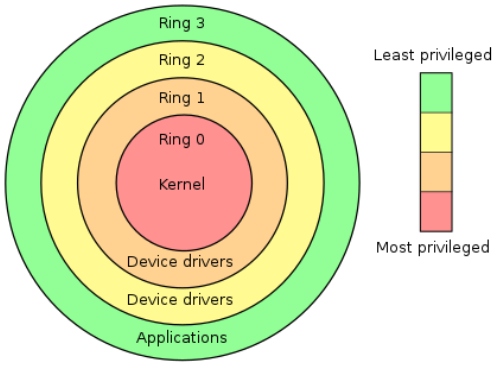
Un système d'exploitation joue le rôle à la fois de **gestionnaire de ressources** et de **machine étendue**.

Dans le cas de la machine étendue, il joue le rôle d'interface applicative, c'est-à-dire qu'il gère les périphériques et leurs interactions.

Dans le cas du gestionnaire de ressources, le système d'exploitation gère le CPU et la RAM. En pratique, quand plusieurs programmes demandent des ressources à l'ordinateur, l'OS va arbitrer quel programme reçoit quelle ressource à quel moment. La gestion passe par un **ordonnanceur** (scheduler) qui gère les processus par exemple, ce rôle est joué par le CPU. Tandis que la RAM charge des données en mémoire.

L'OS est du **logiciel**, le code de l'OS s'exécute en **mode privilégié**. Ce sont les logiciels, les applications qui peuvent basculer en mode privilégié, pas l'utilisateur. Ce mode dénote qu'un code à accès à la totalité des instructions et à accès à la totalité de la RAM. Alors que le code d'un utilisateur s'exécute en mode **non privilégié**.

Voici une représentation du niveau de privilège que les programmes peuvent avoir ou demander.



De ce fait, les applications et les programmes **doivent** passer par les services de l'OS pour accéder à des périphériques, elles n'y ont pas accès directement. Les limitations de l'OS se répercutent donc sur les programmes.

L'OS est matérialisé par du code : des appels système, des traitements d'interruption, des ordonnanceurs, des démons, etc. Et aussi par des données en RAM et sur le disque.

L'ordonnanceur désigne le composant du noyau du système d'exploitation choisissant l'ordre d'exécution des processus sur les processeurs d'un ordinateur. Alors que le chargeur est un composant du système d'exploitation dont le rôle est de charger des programmes en mémoire, afin de créer un processus. Ses principales responsabilités sont la lecture et l'analyse du fichier exécutable, la création des ressources nécessaires à l'exécution de celui-ci, puis enfin le lancement effectif de son exécution.

Finalement, le système d'exploitation gère le démarrage, les processus, l'accès au CPU, l'allocation de mémoire, les traitements d'interruption, les erreurs.

1.1 Rappel système de numération binaire

1. Puissance de 2 :

   KiB 2^{10} bytes et un byte = 8bits

   MiB 2^{20} bytes

   GiB 2^{30} bytes

   TiB 2^{40} bytes

2. Combien de blocs de 4KiB dans un espace de 32GiB ?
   32GiB/4KiB = $(32*2^{30})/(4*2^{10})$ = $(2^5 * 2^{30})$ bytes / $(2^2 * 2^{10})$ bytes = 2^{23} blocs.

3. Quelle est la représentation binaire de 43 ? et de `0x2B` ?
   — 43 codé sur 8 bits → 00101011
   — `0x2B` en binaire → 0010 1011 (donc 43 en décimal)

4. Quel est l'intérêt de la base hexadécimale ?
   La base hexadécimale permet de représenter l'information d'une manière plus compacte que le binaire.

5. Quelle est la différence de représentation de 11 et "11".
   La représentation en mémoire est différente, le premier est une valeur et le second, une chaîne de caractères. Un éditeur de texte pourra lire la chaîne de caractères mais essayera de traduire les valeurs binaires en ASCII, ce qui n'affichera pas correctement le contenu du fichier.

6. Comment visualiser du binaire ? On ne peut pas utiliser `vi` ou `ed` pour voir ces données, on doit utiliser des programmes comme `od`.
7. On peut trouver le modulo d'une division avec une puissance n de 2 en regardant les n bits de poids faibles. Les bits de poids forts restants sont le résultat de la division entière.

On utilise des puissances de 2 pour identifier les tailles et les blocs en mémoire. Cette représentation nous permet de calculer une les restes d'une division sans avoir recours à l'ALU (unité informatique et logique), dans certains cas favorables.

2 Appels système

2.1 Définition

Avant tout,un appel système est du code l'OS. Il réalise un service pour l'OS et s'exécute en mode **privilégié**.

2.2 Privilèges

Quand on parle de privilège, c'est-à-dire qu'on a accès à la **totalité** de la RAM et qu'on a accès aux **instructions** d'accès au disque.

Il n'y a pas d'instruction spécifique pour passer en mode privilégié, le basculement de mode se fait au niveau du processeur (CPU).

— Les appels systèmes provoquent un basculement en mode privilégié.
— Les traitements d'interruption provoquent un basculement en mode privilégié.

2.3 Utilisation

Les services du système correspondent à un numéro (read = 0,...). Les numéros correspondant aux services du système tout le mécanisme de basculement changent selon l'architecture.

Pour utiliser un service un programme doit :

1. Fournir un numéro de service dans un registre qui dépend de l'architecture (`EAX` en 32 bits et `RAX` en 64 bits).
2. Fournir les paramètres nécessaires à l'appel système dans les registres `EBX`, `ECX` en 32 bits et `RDI`, `RSI` en 64 bits.
3. Provoquer le basculement dans le noyau et le saut de privilèges (via l'interruption logicielle `0x80` en 32 bits et via l'instruction `SYSCALL` en 64 bits).
4. Vérifier le statut d'erreur dans le registre `EAX` ou `RAX`.

Dans les exemples aux sections 32 bits et 64 bits, les trois premières étapes sont reprises.

On suit donc bien exactement les étapes décrites.

Les numéros des appels systèmes sont documentés dans des fichiers en langage `C`. Pour connaître les paramètres utilisables, on doit utiliser les manuels via la commande `man`, de niveau 2.

```
man 2 read
man 2 open
```

2.4 Intel 32bits

En 32 bits, `exit(0)` est traduit par :

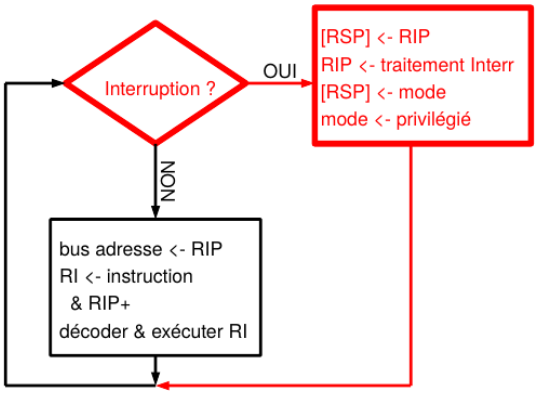
— `mov eax, 1`, le numéro du service exit en 32 bits.
— `mov ebx, 0`, le paramètre de l'appel système `exit`.
— `int 0x80`, l'interruption logicielle.

Pour rappel, `0x80` est le numéro de l'**interruption logicielle** qui assure le basculement et branchement sur le code de l'appel système en Linux.

2.4.1 Mécanisme d'interruptions

Le mécanisme d'interruptions provoque un basculement de mode du CPU.

D'abord le bus d'adresse reçoit `RIP` et `RI` reçoit l'instruction. Aussi, on incrémente `RIP` pour qu'il passe à l'instruction suivante. Puis, il faut **décoder** et **exécuter** l'instruction qui se trouve dans `RI`. Sans interruption, on ne fait que répéter les instructions précédentes. En pratique, à la fin de chaque instruction, le CPU vérifie s'il y a eu une interruption. Si c'est le cas, il met la valeur de `RIP` sur le sommet de la pile, pour pouvoir le récupérer plus tard (après l'interruption). On sauvegarde aussi le mode (privilégié ou pas) et puis `RIP` pointe vers l'interruption dans le code du noyau. Voici une représentation. Une instruction de `RI` se termine toujours à part si elle provoque une **erreur**, on parlera alors d'**exception**.



2.5 Intel 64bits

En 64 bits, `exit(0)` est traduit par :

```
1 mov rax, 60 ; n° du service exit en 64 bits
2 mov rdi, 0  ; paramètre de l'appel système exit
3 syscall
```

L'instruction `SYSCALL` assure le basculement et branchement sur le code de l'appel système.

2.6 Exemples d'appels systèmes

open Crée un descripteur en RAM pour un fichier qu'on souhaite lire ou écrire et y mémorise l'avancement dans le fichier.

read Transfère en RAM *n* bytes depuis un fichier.

write écrit *n* bytes depuis la RAM vers un fichier.

fork Clone un programme qui tourne en mémoire.

exit termine un programme qui tourne.

2.7 Questions

— Un appel système sur Linux en 32 bits provoque toujours la même interruption ?
Pour une architecture et un système donné, ici Linux 32 bits, on utilise bien toujours la même interruption. Pour Linux 32bits `0x80`, Pour Windows 32bits on utilise `0x21`. Donc la réponse est vraie.

— Donnez une brève définition d'un appel système.
C'est du code de l'OS qui réalise un service pour les applications et qui s'exécute en mode privilégié.

— Pourquoi un appel système est-il un passage obligé pour les programmes ?
Parce ça permet de protéger le système et empêcher les programmes de détruire des espaces importants. C'est un moyen centralisé de gérer les ressources, de manière sécurisée.

— `SYSCALL` est une instruction du processeur ?
Oui, c'est une instruction du processeur, celle d'interruption.

— `SYSCALL` est une instruction privilégiée du processeur ?
Non, `SYSCALL` n'est pas une instruction privilégiée, c'est l'instruction qui permet à du code non privilégié d'appeler l'OS.

3 Multiprogrammation et timeslicing

Cette section va concerner la **multiprogrammation** et le **timeslicing**. La multiprogrammation apparaît tôt dans l'histoire de l'informatique car la monoprogrammation était une énorme contrainte de temps et de ressources au niveau des CPU vu que le CPU était bloqué pendant les entrées et sorties lentes.

Multiprogrammation Action de charger plusieurs programmes en mémoire et qui s'exécutent de manière entrelacée alors qu'en monoprogrammation on a un seul programme en mémoire à tout moment.

Timeslicing Ajout de contrainte de temps à la multiprogrammation.

Quand on parle de **processus** ici-bas, ce sont les programmes chargés en mémoire. Ils s'exécutent en mode normal de manière habituelle et en mode **privilégié** quand on exécute du code système pour le compte du processus.

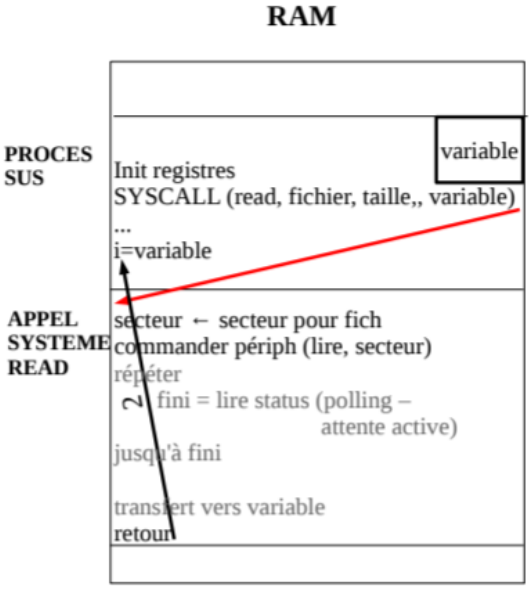
La **multiprogrammation** permet de ré-attribuer le CPU au moment d'une entrée/sortie (*input/output*).

Le **timeslicing** quant à lui, définit un temps maximum pendant lequel un processus peut utiliser le CPU sans interruption. Une fois ce délai passé, le processus n'a plus accès au CPU même s'il ne fait pas de demande d'entrée/sortie.

Nous allons d'abord voir ce qu'il se passe dans le cas de la monoprogrammation et puis les différentes stratégies pour mitiger ce problème.

3.1 Monoprogrammation

Voici un exemple de programme chargé en mémoire et exécuté en monoprogrammation.



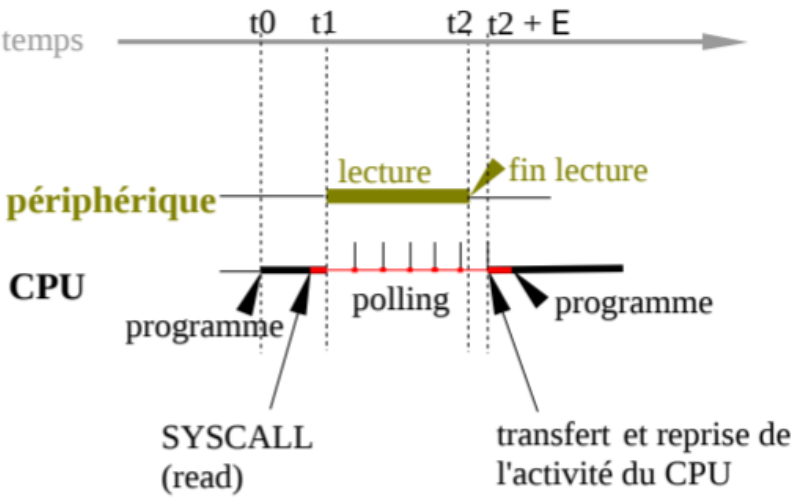
En ce qui concerne le code du **programme**, il est exécuté en mode normal, accède aux registres avec les paramètres nécessaires. Ensuite, l'opération `SYSCALL` provoque le basculement dans le noyau et donc le passage en mode privilégié.

Maintenant, le programme a perdu la main et donc on attend. On demande constamment au CPU si le basculement est terminé ou pas, cette opération s'appelle le **polling**. Une fois qu'on revient au mode normal (quand le **polling** indique que l'appel système est fini), on peut exploiter les données lues en RAM.

Entre temps, le code de l'appel système **read** qui a provoqué le basculement (via `SYSCALL`) va identifier le numéro de secteur nécessaire, commander au périphérique de lire le numéro de secteur et la donnée stockée à cet endroit. Puis, on rapatrie les données lues vers la variable et on rend la main au code utilisateur. Le **polling** se fait constamment durant cette période savoir si on peut reprendre le code utilisateur.

Dans ce cas de monoprogrammation, le CPU **attend** que le périphérique ait mis à disposition les données pour les transférer en RAM et reprendre son activité au profit du programme.

Ici, on voit la ligne du temps du processus et des appels systèmes. Le rouge indique le passage au mode privilégié.



3.2 Processeur canal ou DMA

Pour remédier au problème d'attente et d'usage inefficace du CPU ; on va associer au CPU un processeur externe relié par le bus (CANAL) et lui confier la gestion du périphérique et le transfert depuis la RAM.

On parle alors de processeur canal ou DMA (*Direct Memory Access*).

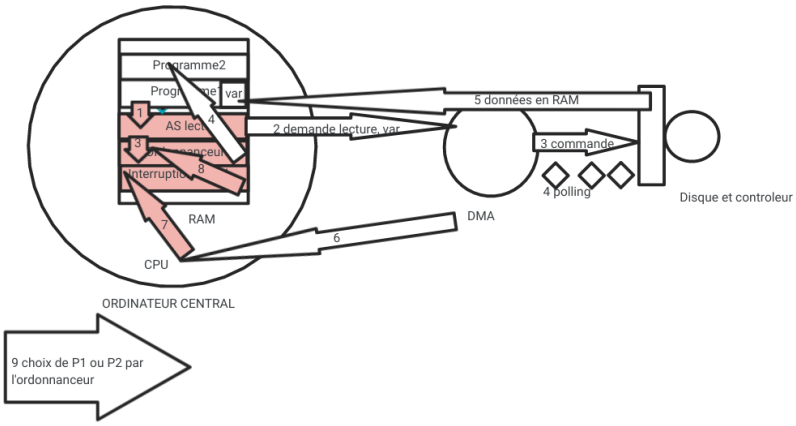
Ce processeur commande le périphérique et **accède à la RAM** pour transférer les données lues. Il génère une interruption pour prévenir de la fin de la lecture. Une fois que l'interruption est établie, le CPU est libéré au profit d'un deuxième processus chargé en RAM. Donc, ce processeur canal est relié au bus et accède à la RAM, il **interfère** avec le fonctionnement du CPU. De ce fait, il faut synchroniser le CPU et le canal (régler les accès à la RAM via le bus).

Grâce à cela, le CPU ne gère plus le polling et le transfert des données, ces tâches sont entièrement déléguées au DMA et permettent de rentabiliser l'usage du CPU. On peut maintenant exécuter plusieurs programmes de manière entrelacée et un programme qui demande une lecture est **bloqué** au profit d'un autre qui récupère le CPU.

Bien que cette stratégie permet d'améliorer l'usage du CPU, on a introduit de la complexité puisqu'il faut garder le compte d'où en est chaque processus. Pour ce faire on va :

- Mémoriser l'état des registres (`RIP`, `RAX`,...) du processeur via une **table des processus** dont le contenu est expliqué plus bas. On dira qu'on mémorise le **contexte** du CPU.
- Gérer qui est exécuté quand grâce à l'**ordonnanceur**.
- Protéger l'accès à la mémoire par la **segmentation** (sur Intel).
- Gérer l'attribution des ressources non partageables et les conflits dûs à l'**interblocage**.

Voici un exemple d'usage de multiprogrammation avec deux programmes, géré par le DMA :



3.3 Gestion des processus

Pour ce qui est du **contexte** et de la **table des process**, elle mémorise les attributions et cessions des processus et elle contient :

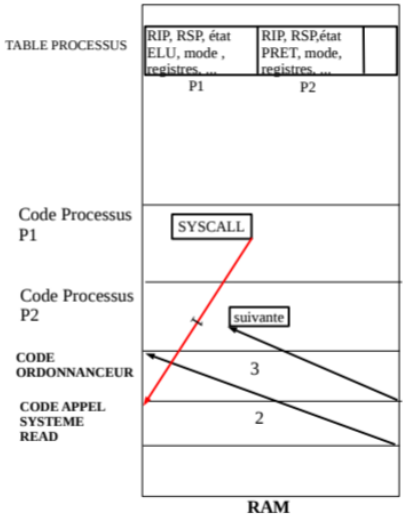
- Une valeur de `RIP` pour suivre où l'instruction se trouve dans le registre des instructions.

— Une valeur de RSP, le registre des pointeurs sur la **pile** (*stack*).
— Une état de **registres** du CPU (RAX, RBX,...).
— Un **mode** de fonctionnement du CPU (privilégié ou pas).
— Un **état** (élu, prêt, bloqué). Un processus est bloqué s'il est dans l'attente de fin d'une lecture/écriture. Prêt s'il est en attente et qu'il peut s'exécuter. Enfin, on dira le processus est **élu** s'il s'exécute et lui **seul**. Plusieurs demandes de lecture/écriture et donc plusieurs processus bloqués peuvent coexister.

Enfin, l'ordonnanceur choisit des processus **prêts** pour leur attribuer le CPU.

3.4 Déroulement d'une lecture

Maintenant, voyons le déroulement d'une lecture en multiprogrammation, avec les programmes P1 et P2. On utilise le mot "processus" pour un programme qui s'exécute en mémoire.



Dans le cas d'une demande de lecture on a :

1. P1 demande une lecture, ce qui provoque un SYSCALL.
2. On a un appel système.
   — état de P1 devient **bloqué**.
   — On sauvegarde le **contexte** de P1 dans la table des process.
   — On commande le périphérique et le processeur canal.
   — RIP prend l'adresse de l'ordonnanceur.
3. Ordonnanceur.
   — L'état de P2 devient **élu**, car c'est le seul prêt (l'autre est bloqué).
   — On utilise la table des process pour charger les registres et le mode de P2.
   — RIP prend la valeur de RIP de P2 qui était stockée dans la table des process.
4. Comme P2 à la main, il continue à s'exécuter.

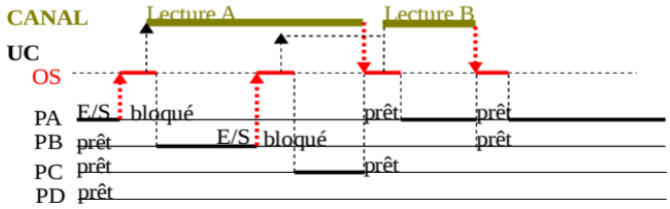
Maintenant, quand on arrive à la fin de la lecture :

1. Il y a une interruption due au processeur canal.
   — On met la valeur de RIP dans la pile (*stack*).
   — Le mode devient **privilégié**.
   — RIP prend l'adresse de la gestion d'interruption.
2. Traitement de l'interruption CANAL.
   — L'état de P2 passe à prêt.
   — On sauvegarde le **contexte** de P2 dans la table des process.
   — L'état de P1 passe à **prêt** car il a fini sa lecture.
   — RIP prend l'adresse de l'ordonnanceur.
3. Ordonnanceur
   — L'état de P1 ou de P2 dans la table passe à **élu** suivant le programme qu'on choisit.
   — On charge les registres et le mode de l'élu depuis la table des process.
   — RIP prend la valeur de RIP de l'élu, qu'on connaît aussi grâce à la table des process.
4. L'élu à la main et s'exécute.

La section suivante donne le découpage du temps pour les actions décrites au-dessus.

3.5 Multiprogrammation et occupation du CPU

Voici à quoi ressemble les tranches de temps de la multiprogrammation décrite au-dessus. Comme avant, le rouge indique le passage à du code de l'OS via un appel système.



Ce qu'on constate c'est qu'un processus qui fait de la lecture/écriture (entrée/sortie) rend la main et permettent donc d'optimiser l'usage du CPU. Par contre, les processus de calcul ne rendent jamais la main.

Donc, on voit que le canal est utile dans les cas d'entrées et de sorties mais qu'il n'est pas vraiment rentabilisé dans des cas de calculs. Une manière de mitiger ce problème est abordé dans la section suivante.

3.6 Time slicing

On a vu que dans les cas de calculs, le programme ne rend pas la main et bloque l'accès aux ressources. Si le calcul prend énormément de temps, on va limiter le temps qui est disponible pour ce process, pour rentabiliser le processus canal.

On va se servir de l'**interruption horloge** pour retirer le CPU au process qui l'utilise. C'est un mécanisme qui retire et assigne et des tranches de temps aux processus.

En pratique, un processus garde le CPU pendant une tranche de temps de durée maximum t et après on lui retire au profit d'un autre. Dans ce cas, l'**ordonnanceur** est appelé par l'interruption horloge. On optimise l'utilisation du canal mais on attention, on doit maintenant gérer les **changements de contexte** chaque fois qu'on passe de tranches en tranches.

Cette gestion de CPU permet l'exploitation interactive, comme partager du temps CPU entre utilisateurs, car elle minimise les temps de réponse. On parle alors de **time sharing**, qui est une notion différente du **time slicing** mais liée.

Les slides suivants décrivent les différents changements d'états pour les processus.

Élu → prêt Via une interruption, que ce soit une interruption horloge ou une interruption canal.

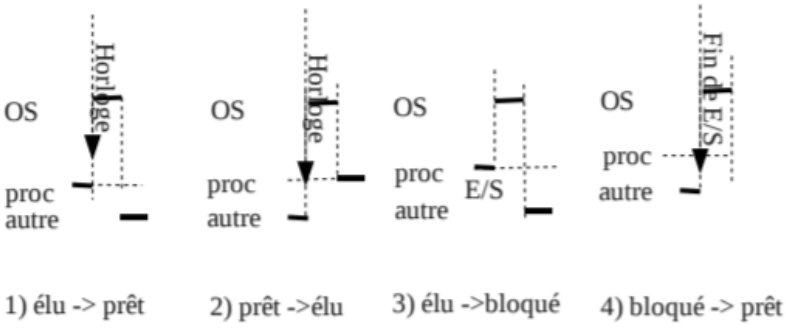
Prêt → élu Quand l'ordonnanceur choisi un processus.

Élu → bloqué Quand on demande une ressource indisponible (lecture, écriture, demande de ressource non partageable).

Bloqué → prêt Lors de l'interruption canal ou interruption liée à des ressources qui deviennent partageables.

On ne peut pas passer de prêt à bloqué car il faut demander des ressources ou être exécuté pour être bloqué, ce qui n'est pas le cas d'un processus prêt. Aussi, on ne peut pas passer de bloqué à élu car c'est l'ordonnanceur qui élit les processus et donc on doit d'abord être prêt avant de pouvoir être élu.

Voici un exemple de basculement d'états :



1. L'interruption horloge interrompt le processus et passe la main à l'autre. Donc l'élu devient prêt car son temps est fini.
2. Interruption du premier processus permet au second processus d'avoir la main, il passe de prêt à élu.

3. La demande d'E/S crée un appel système qui bloque le programme en cours, c'est donc un appel système qui fait basculer d'élu à bloqué. Dans ce cas il n'y a pas d'interruption à proprement parler, c'est l'appel système qui interrompt le processus.
4. C'est le canal qui va gérer le passage de bloqué à prêt, car les ressources utilisées lors de l'E/S sont libérées.

3.6.1 Préemption

Plus tôt, on a parlé de l'action d'allouer ou retirer le CPU au processus via le *time slicing*, c'est la préemption (le fait de prendre).

On parlera de :

système non préemptif Un processus qui se bloque (entrée/sortie) ou rend la main spontanément. Il n'y a pas de notion de *time slicing*.

système préemptif Un processus qui se bloque (E/S) ou rend la main spontanément ou bien lorsque son temps est écoulé (Windows NT, XP, Linux depuis le noyau 2.6).

système coopératif Un processus rend la main spontanément.

Il faut aussi savoir qu'un ordonnanceur préemptif ne peut pas exister sans interruption horloge et qu'il peut être non préemptif même en présence d'interruption horloge.

3.6.2 Précisions sur le time slicing

Du coup, on peut se demander quand à lieu l'ordonnancement exactement ?

1. Demande d'entrée ou sortie
2. Fin d'entrée ou sortie
3. En interruption d'horloge (préemption)
4. Nouveau processus
5. Fin d'un processus
6. ...

De manière générale, à chaque fin de traitement d'interruption et en cas d'appel système, on fait appel à l'ordonnanceur.

3.7 Questions

1. Vrai.
2. Autant de processus qu'on veut tant qu'il y a de la place en mémoire.
3. Le CPU ne s'arrête jamais, donc tous peuvent être prêts sauf un qui doit être élu.
4. Tous peuvent être bloqués sauf un qui doit être élu.
5. Faux.
6. Faux.
7. Le *time slicing* est utile quand on a des processus longs qui ne rendent pas la main. En limitant leur temps attribué, on peut mieux rentabiliser l'usage du canal. Mais l'usage de tranches oblige à conserver les contextes et donc les changements de contexte prennent du temps, ce qui représente un inconvénient. Dans le cas de la multiprogrammation, les entrées et sorties sont très rapide car il n'y a pas changement de contexte mais s'il y a des processus qui bloquent le CPU pendant longtemps, alors on ne rentabilise pas le canal.
8. Un process passe de élu à prêt par une interruption (horloge ou canal) et un processus passe de bloqué à prêt par une interruption canal.

4 Ordonnancement

L'ordonnanceur est du code de l'OS qui choisit le prochain processus élu parmi les processus prêts.

Ce choix est complexe et il faut prendre en compte plusieurs contraintes pour un "bon" algorithme d'ordonnancement. Ces contraintes sont souvent mutuellement exclusives donc optimiser une contrainte peut créer des ralentissements dans d'autres contraintes.

— Équité : Les mêmes processus prennent un temps similaire.
— Temps de réponse : Un processus ne bloque pas les réponses d'autres processus pendant trop longtemps. Important pour les systèmes interactifs.

— Temps d'exécution : Un processus s'exécute rapidement alors que les entrées et sorties pénalisent le temps d'exécution.
— Rendement/efficacité : Utilisation correcte du temps et des ressources du CPU.
— Équilibre : Utilisation correcte et équilibrée des différentes parties du système.

On voit donc qu'il faut trouver un compromis entre ces contraintes. Il n'existe malheureusement pas d'algorithme magique qui optimise l'utilisation du CPU dans toutes les situations. On peut donc adopter plusieurs stratégies décrites dans les points suivants, qui supposent généralement l'utilisation du *time slicing*.

4.1 Tourniquet

Avec cette stratégie, chaque processus **prêt** reçoit, à tour de rôle, un quantum du temps processeur.

Un processus perd le CPU :

— parce qu'il a terminé
— parce qu'il a besoin d'une entrée/sortie (élu $\rightarrow$ bloqué)
— parce qu'il a épuisé son quantum de temps.

Puis, l'ordonnanceur élit le suivant de la liste.

On doit être prudent dans le choix du quantum de temps. Voici un exemple avec 10 processus prêts en même temps et un changement de contexte qui prend **5ms**. Le dénominateur de l'efficacité est la somme du quantum et du changement de contexte.

quantum	temps max attente	efficacité CPU
5 ms	95 ms	50 % = 5/10
100 ms	+- 1 seconde	95 % = 100/105
1 sec	+- 10 secondes	99.5 % = 1000/1005

4.1.1 Priorité

Un soucis du tourniquet et qu'il place tous les processus sur un même niveau de priorité alors que certains processus peuvent être importants et devraient être traités au plus vite.

Pour remédier à ça, on associe des priorités aux processus et l'ordonnanceur élit les processus dans la liste, par priorité décroissante. Cette solution est avantageuse mais risque de laisser les process de priorité basse sur le côté, on parle alors de **famine**.

Une solution est décrite au point suivant, en utilisant des priorités dynamiques.

4.2 Priorités dynamiques

4.2.1 Solution par décrémentation

Une solution au problème de famine est de décrémenter la priorité d'un processus chaque fois qu'il a accès au processeur. Grâce à cette solution, on peut permettre à des processus basse priorité d'avoir quand même accès au processeur.

Par contre, on oublie la priorité initiale du processus et s'il est important et tourne souvent, on le pousse constamment vers le bas des priorités, ce qui n'est pas optimal non plus, une autre solution doit être choisie.

4.2.2 Solution par quantum utilisé

Du coup, une autre idée sera de favoriser les processus qui demandent beaucoup d'entrées et sorties en leur attribuant tout de suite le processeur, en ajustant leur priorité en fonction du temps déjà utilisé.

Un processus qui demande beaucoup d'entrées et sorties doit être favorisé car ils sont exécutés en parallèle avec le processeur.

$$\text{priorité} = \frac{\text{quantum}}{\text{temps du quantum utilisé}}$$

Par exemple, un processus qui a un quantum de 100ms et est bloqué par une demande d'entrée / sortie après 5ms. Alors, il aura une priorité 100/5 = 20. On voit que moins le temps a été utilisé, plus la priorité est haute.

4.3 Files multiples ou classes

On a vu que la question des priorités était problématique et on remarque que pour les processus qui utilisent beaucoup le CPU, la performance augmente quand le quantum est grand. Tandis que pour les processus qui font beaucoup d'entrées et sortie, on a un meilleur temps de réponse quand le quantum est petit.

De là, on tire la conclusion qu'on peut attribuer un grand quantum à un processus qui demande beaucoup de calcul et une **priorité** plus grande ainsi qu'un plus petit quantum pour les processus qui demandent beaucoup d'E/S.

Voilà un exemple avec 5 tourniquets, un par classe. Tous les processus reçoivent un quantum en fonction de leur classe et un processus de classe inférieure est élu quand aucun processus de classe supérieure n'est en état prêt.

Classe	Quantum	priorité	remarque
1	40 ms	5	pour un processus d'E/S
2	80 ms	4	...
3	160 ms	3	intermédiaire
4	320 ms	2	...
5	640 ms	1	pour un processus de calcul

Mais attention, comment connaître la classe d'un processus *a priori* ? On ne la connaît pas et on choisira de la fixer au départ.

Par exemple, on mettra tous les processus à la classe 1. Ce qui représente un désavantage car on voulait les classer et donc on choisira de les faire évoluer en fonction de leur usage du CPU.

Chaque fois qu'un processus consomme tout son quantum, il descend d'une classe. De ce fait, les processus qui font des E/S restent en classes élevées et les processus de calcul baissent de classe et ont un plus grand quantum.

Mais qu'en est-il des processus hybrides ? Par exemple un processus qui demande des E/S et qui fait des calculs. Pour remédier à ce problème, on augmente la classe de 1 chaque fois qu'un processus demande une E/S. C'est une bonne stratégie dont on peut cependant abuser car si un programmeur connaît cette stratégie alors il peut créer un processus qui demande des E/S après chaque calcul pour s'assurer de rester en classe haute.

4.4 Questions

— L'ordonnancement du tourniquet utilise le plus souvent un quantum fixe. Donnez un argument en faveur d'un petit quantum et un argument en faveur d'un grand quantum.
Le petit quantum permet de minimiser le temps d'attente mais a une faible efficacité d'utilisation du CPU, car le temps qu'on passe pour les changements de contexte est proportionnellement plus grand.
Le grand quantum permet de maximiser l'usage du CPU et donc son efficacité car le changement de contexte est proportionnellement plus faible. Par contre, on a un temps de réponse qui augmente et peut être problématique.
  — Soit un quantum de 92ms et un temps d'ordonnancement de 8ms, calculez l'efficacité du processeur.
  On mesure le temps utile sur le temps total donc :

$$\frac{92}{(92+8)} = 92\%$$

— Donnez une haute priorité à un processus favoris sont temps de réponse.
Vrai.
— Avec un ordonnancement par tourniquet un processus s'interrompt pour une seule raison.
Faux, il y a aussi fin de tâche, demande de ressource (E/S) ou interruption matérielle.
— Un ordonnanceur à priorités statiques améliore le temps de réponse de tous les processus.
Faux, car il peut y avoir famine pour certains processus de basse priorité.

5 Mémoire

Cette section traite de l'usage de la mémoire.

5.1 Introduction

Dans une vue idéale de la mémoire, chaque processus doit disposer d'une mémoire :

— privée
— infiniment grande
— rapide
— non volatile
— réalisée dans une technologie "bon marché"

TYPE	TAILLE	RAPIDE	NON VOLATILE	BON MARCHÉ
cache	MiB	++	NON	-
mémoire vive	GiB	+	NON	+-
disques	TiB	-	OUI	+

Le meilleur compromis se trouve au niveau de la RAM (*Random Access Memory*) qui est utilisée pour les instructions et variables d'un programme.

On y accède aux "mots" via des adresse. Le "mot" est l'**unité adressable**. Chaque "mot" en RAM correspond à une adresse. En architecture Intel, le "mot" est le **byte**. En déposant l'adresse sur le bus d'adresse, on peut lire ou modifier le "mot" en RAM.

5.2 Sans abstraction

Sans abstraction de la mémoire, le programme manipule des adresses **physiques**.

```
1 MOV [8192], reg
```

Dans ce cas, 8192 correspondant à une vraie adresse en RAM. Le programme est maître des adresses qu'il utilise en RAM. Donc, deux programmes pourraient utiliser la même adresse physique, ce qui cause énormément de problèmes. Donc, ce mode d'adressage ne peut être utilisé qu'en monoprogrammation (on exécute un seul programme à la fois).

Si on utilise plusieurs programmes à des moments différents, on doit s'assurer qu'ils écrivent à des endroits différents en mémoire. Ce mode d'adressage ne permet pas la multiprogrammation.

5.3 Abstraction de la mémoire

Pour pallier les problèmes de la monoprogrammation, on va utiliser la **relocation statique**. Ce qui signifie que le programme utilise des adresses relatives ajustées au **chargement**.

5.3.1 Relocation statique

En relocation statique, la traduction des adresses relatives en adresses physiques est réalisée une seule fois au **chargement** du programme.

Toute référence à la mémoire est corrigée en y ajoutant l'adresse de chargement du programme. Le programme qui s'exécute utilise les **adresses physiques** que le chargeur (*loader*) aura adaptées. Par exemple, l'adresse 8192 deviendra 108192 pour le programme chargé en 10000 et 308192 pour le programme chargé en 30000.

Après le chargement du programme, toutes les adresses auront été converties en adresses **physiques**. En relocation statique, l'utilisation des adresses est absolue, on y fait directement référence.

5.3.2 Relocation dynamique

En relocation **dynamique**, les adresses du programme qui s'exécutent sont exprimées par rapport à un espace d'adressage propre au programme et chaque programme a son ou ses propres espaces d'adressage.

Aussi, les adresses ne sont pas modifiées au chargement du programme. Elles sont exprimées relativement à l'espace d'adressage. Aussi, en Intel 32bit, l'espace d'adressage est limité à 4GiB.

Ce qui est dynamique, c'est la traduction de chaque instruction à la volée au **moment de l'exécution**. Ce processus est géré par un dispositif **hardware**. Sur Intel c'est le **MMU** (*Memory Management Unit*) qui est responsable de la traduction des adresses à l'exécution.

En relocation dynamique, les adresses sont relatives, on les exprime par rapport à une base jusqu'à une certaine limite de taille. En pratique, deux registres **spéciaux** mémorisent l'adresse de chargement de la **base** et de la **limite**. Lors de chaque accès, l'adresse relative est convertie et on vérifie qu'on est bien dans les limites permises. Cette vérification se fait par un dispositif **hardware**. Si on dépasse la limite, le **MMU** génère une exception (*Segmentation fault*).

Dans le cas dynamique, exécuter une instruction nécessite **deux accès** à la RAM et donc deux traductions. Il faut lire l'instruction depuis la RAM et écrire en RAM le contenu du registre.

On a donc une plus grande flexibilité mais les corrections d'adresses ralentissent l'exécution du programme.

En conclusion, la relocation dynamique :

— Permet la **coexistence** des programmes en mémoire et la protection de leurs espaces respectifs. (+)

— Le **swapping** et le déplacement sont facilement envisageables car il n'y a pas de corrections à apporter au programme étant donné qu'on modifie le registre de base. (+)
— Par contre, chaque accès mémoire nécessaire une **addition** et une **comparaison**, ce qui ralentit l'exécution. (-)

5.4 Taille et *swap*

Parfois, il arrive que la taille de la mémoire physique soit insuffisante pour accueillir l'ensemble des programmes qui tournent en même temps. Deux approches permettent d'éviter le problème.

1. Le *swapping* des programmes.
2. Le *swapping* des parties de programme.

Dans le premier cas, un programme sort de mémoire au profit d'un autre. Aussi, tous les programmes n'utilisent pas le même espace en RAM. Cette stratégie crée donc de la **fragmentation** de la RAM qui est coûteuse à défragmenter. On parle de fragmentation quand la mémoire présente une multitude d'espaces libres mais de taille insuffisante à accueillir un nouveau processus. On défragmente pour libérer des espaces contigus.

Dans le second cas, on utilise le fait qu'un programme ne doit pas entièrement résider en RAM à tout moment de son exécution. On va donc charger des morceaux de programme, appelés **pages**. Seulement une partie des pages des programmes résideront en RAM à un instant donné. C'est ce qu'on appelle la **pagination** ou mémoire virtuelle.

5.5 Segmentation (mode réel et protégé)

L'espace d'adressage physique d'un processus peut être séparé en segments (code, données,...).

La segmentation sur l'architecture Intel est une gestion de la mémoire à deux niveaux : le mode **réel** et le mode **protégé**.

5.5.1 Mode réel

Dans les architectures Intel, le mode réel n'est accédé qu'au moment du démarrage de l'ordinateur, pendant un court instant. Tout ce qui concerne les **rings** ne se déroule qu'en mode protégé. Ce dernier est utilisé sur la majorité des machines actuelles.

Le mode réel est le seul mode d'adressage des premiers processeurs 8086. Comme dit au-dessus, ce mode d'adressage réel subsiste dans les processeurs actuels, uniquement au démarrage. Le processeur accède à 1 MiB de RAM dans ce mode.

5.5.2 Mode protégé

La manière de traduire les adresses différencie les deux modes ; en mode protégé la traduction d'adresse se fait en plusieurs étapes :

$$\text{adresse logique} \rightarrow \text{adresse linéaire} \rightarrow \text{adresse physique}$$

La dernière étape est nécessaire si on utilise la pagination. Si on est segmentation pure (pas de pagination), l'adresse linéaire est égale à l'adresse physique.

Pour bien comprendre ces traductions, on doit comprendre ce qu'est une adresse **logique**.

Une adresse **logique** est une adresse dans un segment, elle est composée de deux parties :

— Le **registre sélecteur** et l'**offset**.
— Les **registres sélecteurs** :

`CS` associé au segment de code

`DS` associé au segment de données

`SS` associé au segment de pile

...

Un programme utilise plusieurs segments et leur utilisation est parfois implicite :

— `jmp boucle` → `jmp CS:boucle`

— `mov eax, [ebx]` → `mov eax, [DS :ebx]`

— `push eax` → utilise `SS` :`ESP`

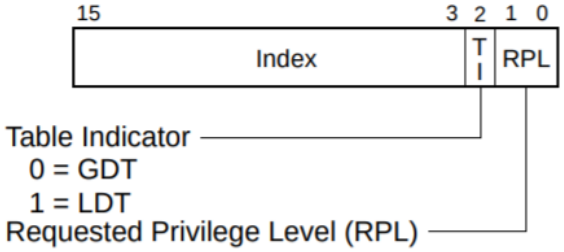
Plus précisément, le **sélecteur** de segment est codé sur 16 bits et on retrouve les registres mentionnés juste avant (`CS`, `DS`, `SS`, ...). On a aussi l'**offset** dans le segment, sur 32 bits.

Chaque segment est un espace d'adressage limité par une base et une limite, comme expliqué dans la section sur la relocation dynamique.

L'architecture Intel prévoit 6 registres sélecteurs de segment, ce sont : `CS`, `SS`, `DS`, `ES`, `FS`, `GS` qui sont tous des registres de 16 bits. Un programmeur peut utiliser plusieurs segments mais seulement 6 seront disponibles en même temps.

5.5.3 Sélecteur de segment

Voici une représentation du sélecteur de segment, il est composé d'un **index** sur 13 bits et donc au maximum 2^{13} descripteurs de segments sont possibles par table. Il y a aussi un indicateur de table sur 1 bit qui précise son on a affaire à la *global description table* (0) ou la *local description table* (1). Enfin, il y a le niveau de privilège demandé (*requested privilege level*) codé sur 2 bits et qui prend valeur entre 0 et 3, faisant référence aux **rings**. On a donc bien 16 bits au total.



Pour trouver l'adresse linéaire, on va simplement additionner la **base** et l'**offset**, on trouve la base grâce au sélecteur de segment.

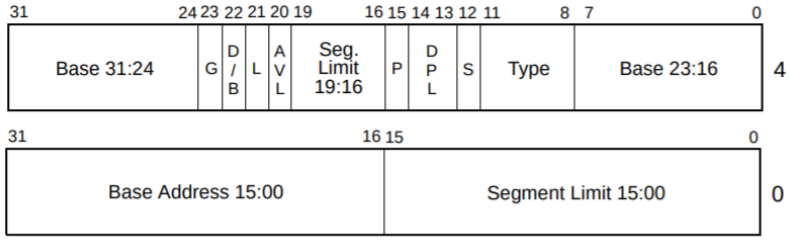
Dans le sélecteur, on a mentionné avoir 1 bit pour savoir si on utilise la table de description globale ou locale. Ce sont deux tables de 8 *bytes* en RAM qui rassemblent les **descripteurs** de segments. La table globale est partagée par tous les process tandis que la table locale appartient à une process. Ce sont les registres `gdtr` et `ldtr` qui contiennent les adresses des tables.

On a aussi mentionné les 2 bits de privilège, ils représentent le *requested privilege level* (*RPL*) ou le *current privilege level* (*CPL*) dans le cas du *code segment*. C'est donc le niveau de privilège demandé ou accordé.

Finalement, l'index de 13 bits sélectionne un des 8192 descripteurs possibles dans la table globale ou locale des descripteurs. Le processeur multiplie la valeur de l'index par 8 (le nombre de *bytes* dans un descripteur de segment) et ajoute le résultat à la **base** de l'adresse de *GDT* ou *LDT*, provenant des registres **GDTR** ou **LDTR**. On a donc $GDTR + \text{index} * 8$ ou $LDTR + \text{index} * 8$; pour trouver l'adresse du descripteur.

5.5.4 Descripteur de segment

Voici une représentation plus complexe, celle du descripteur. On décrira seulement quelques informations pertinentes. Toutefois, on doit comprendre ce que sont la **base**, la **limite** et la **granularité**. La base fait 32 bits, c'est là qu'on trouve l'adresse 0 d'un segment, elle peut être allouée dans un espace de 4 *gigabytes* (GiB). La limite fait 20 bits et spécifie la **taille** du segment. Le processeur interprète la limite de deux manières suivant que le drapeau de granularité soit levé ou pas. Si la granularité est à 0, le segment peut être de taille 1 *byte* à 1 MiB, par incréments en *byte*. Si la granularité est à 1, le segment peut être de taille KiB à 4 GiB, par incréments de 4KiB. Dans le premier cas on utilise le byte comme unité et dans le second on utilise la **page** (donc la pagination). Sans pagination, la base est une adresse physique.



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

En présence de pagination, la **base** est une adresse dans l'espace d'adressage du programme.

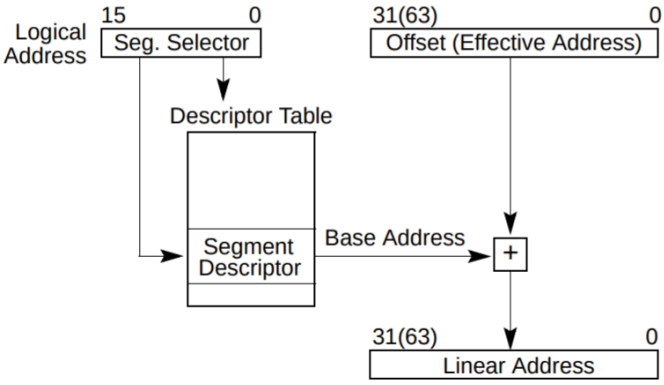
La taille maximum d'un est exprimée par la **limite** et la **granularité**, elle est de 4GiB en pagination.

On a aussi le *descriptor privilege level* (*DPL*) qui donne le niveau de privilège du segment, 0 étant le plus privilégié. Le *DPL* sert à contrôler l'accès au segment. On voit aussi dans la table, le **type** qui donne les droits qu'on a sur le segment : lecture, écriture, exécutable.

Enfin, il y a un drapeau (*P*) qui permet d'indiquer si un segment est présent ou non et qui est utilisé lors du *swap*.

5.5.5 Traduction d'adresse logique en adresse linéaire

Voici une image qui donne le processus de traduction d'adresse logique en adresse linéaire.



5.5.6 Performance de segmentation et protection

Quand on exécute du code et qu'on déplace des données, le **MMU** traduit l'adresse et chaque référence à la RAM requiert une lecture supplémentaire pour le descripteur de la table. Par exemple dans le cas suivant :

```
1 mov ebx, [0x1000]
```

Il y a 3 accès nécessaires, c'est un acte coûteux qu'on va essayer de réaliser le moins de fois possible.

Pour ça, on considère le principe de **localité**. On mémorise les derniers descripteurs de segments utilisés (64 bits sont utilisés). On divise un sélecteur de segment en une partie visible sur 16 bits et une partie cachée sur 8 *bytes* qui contiennent des informations en *cache*, ceci accélère la traduction. De plus, le descripteur est lu uniquement quand on modifie le sélecteur.

En ce qui concerne la protection, on a plusieurs parties de segments qui en sont responsable. Le niveau de privilège (**rings**), le type d'accès (lecture, écriture, exécutable) et la limite de taille. Les privilèges d'un code qui s'exécute sont marqués dans le sélecteur `CS` via le *CPL* allant de 0 à 3. Aussi, les privilèges nécessaires pour accéder à un segment de données se trouvent dans le descripteur de segment (*DPL*) allant lui aussi de 0 à 3.

Ces conditions de protections sont vérifiées et génèrent des erreurs lorsqu'elles sont enfreintes. Un privilège insuffisant générera une exception de type **general protection fault**, en général quand le *CPL* > *DPL*. Le dépassement d'une limite entraînera une exception de type **segmentation fault**.

6 Interblocage