IUP GMI D'AVIGNON

www.centraliup.fr.st

Les Systèmes d'Exploitation des Ordinateurs

> 1/3

Pascal Nocera

Version 1.0

Objectifs

L'objectif est double : connaître les mécanismes internes des **Systèmes d'Exploitation** en général, et appliquer ces connaissances au système **Unix** en particulier, et plus concrètement, étudier l'interpréteur de commandes, la programmation Shell, les filtres, et les processus systèmes. Tout ce qui concerne l'application pratique au système Unix sera étudié dans la partie 2.

Principe des Systèmes d'Exploitation

Programmes systèmes

Un ordinateur, c'est un ensemble de composants (processeur, mémoire centrale, horloges, registres, etc.) qui resterait inerte, si aucun logiciel ne pourrait le faire fonctionner. Un ordinateur associés à des logiciels permet de mémoriser, de traiter et restituer des programmes et des informations, gérer des bases de données, éventuellement jouer, etc.

Ce qui mène à distinguer deux types de programmes : les programmes systèmes qui permettent le fonctionnement de l'ordinateur, et les programmes d'application qui résolvent les problèmes des utilisateurs.

On s'intéressera dans ce cours précisément aux **programmes systèmes**.

Rôle

Il y a des dizaines d'années, les ordinateurs occupaient un immeuble entier, nécessitaient une main d'œuvre spécialisée, et ne pouvaient exécuter qu'un programme à la fois, après quoi il fallait redémarrer et reconfigurer la machine. Le premier système d'exploitation était le premier programme chargé en mémoire, et permettait de charger un programme après l'autre, sans tout réinitialiser.

Aujourd'hui, les systèmes d'exploitation ont plusieurs rôles :

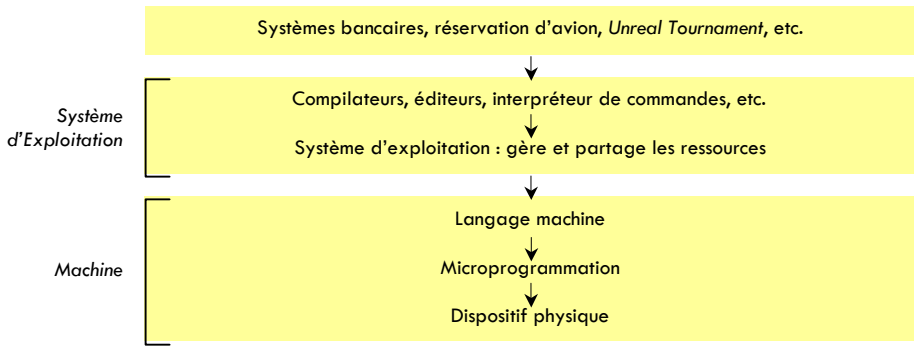
- Ils libèrent le programmeur des contraintes matérielles.
- Ils enrobent le matériel par une **couche logicielle** qui gère l'ensemble du système.
- Ils présentent à l'utilisateur une « machine virtuelle » plus facile à comprendre et à utiliser ; qui donne une image du matériel pas toujours fidèle à la réalité (par exemple, l'arborescence de répertoires dans l'explorateur *Windows* ne correspond pas à la disposition physique des données sur le disque, mais facilite grandement leur traitement et leur compréhension).

Exemple : *Le système Unix peut fonctionner sur des processeurs totalement différents, mais il présente toujours la même machine virtuelle à l'utilisateur, de sorte que celui-ci n'a pas à se préoccuper des contraintes matérielles.*

La machine virtuelle

Une **machine virtuelle** fournit : des fonctions de gestion de l'information, des fonctions d'exécution, mais aussi des services divers tels que l'aide à la mise au point, etc.

On peut réaliser de manière pratique une machine virtuelle en suivant ce modèle théorique :

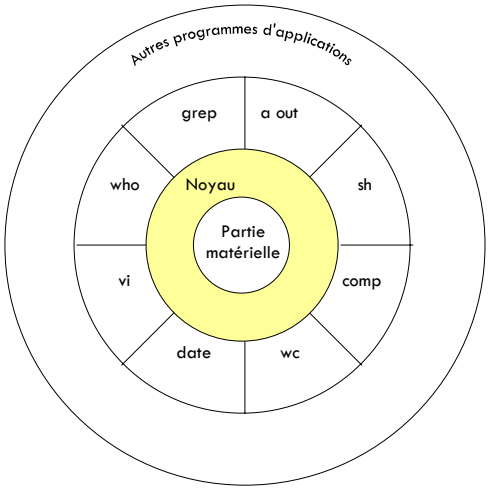


Services

Un système d'exploitation permet : la gestion des ressources physique, le partage et l'échange d'informations entre usagers, la protection mutuelle des usagers, et enfin des services divers tels que des statistiques d'utilisation, des mesures de performances, etc.

Application au Système Unix

Couches

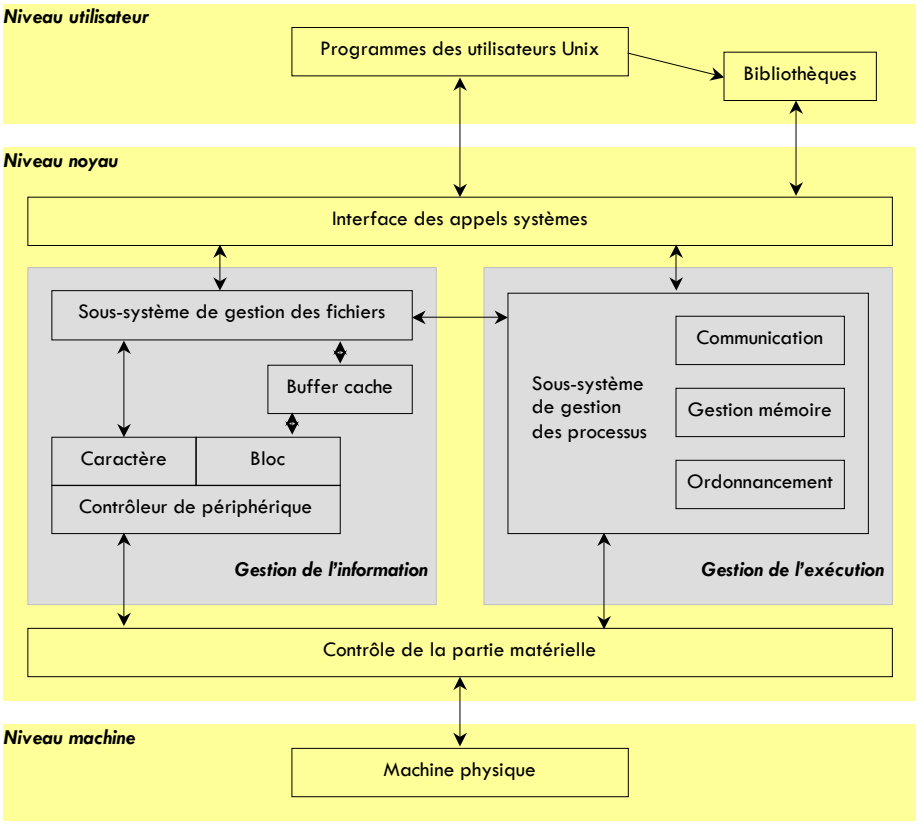


Le système **Unix** est composé de plusieurs couches :

- La partie matérielle ;
- Le noyau : la partie visible du système ;
- Programmes et utilitaires systèmes ;
- Autres programmes d'application.

On remarque qu'il n'y a pas de saut entre les différentes couches. Chaque couche n'est reliée qu'à ses couches adjacentes, sans passe-droit !

Structure du noyau



Le Buffer cache

Un disque dur est constitué de millions de blocs (clusters) de 512 octets ou 1 ko, qui sont écrits ou lus en une seule fois, bloc par bloc, et non octet par octet. Dans un programme utilisateur, on a pas cette contrainte : on lit et écrit ce qu'on veut. C'est le système d'exploitation qui fait la jonction et gère l'écriture et la lecture par blocs à l'aide du **buffer cache**.

En effet, un certain nombre de blocs disques sont gardés en mémoire. Ceci a pour avantage de minimiser les accès au disque et d'accroître des performances d'entrées-sorties. Par contre, ce procédé est sensible aux coupures d'alimentation, car tout ce qui est dans le cache et qui n'a pas été encore écrit sur le disque est perdu lors d'une coupure inopinée, ce qui peut dans certains cas rendre le système instable par la suite. C'est pourquoi des mécanismes d'accès direct au disque,

sans passer par le buffer cache, sont disponibles pour les fonctions sensibles du système.

Gestion des fichiers

Unix est le premier système à gérer un **structure en fichiers hiérarchisés** (en répertoires et sous-répertoires). Il existe trois types de fichiers :

- Les fichiers ordinaires : chaque fichier possède un numéro d'i-node. Ces fichiers sont simplement des suites d'octets.
- Les répertoires : ce sont des fichiers de couples nom de fichier-numéro d'i-node.
- Les fichiers spéciaux : les opérations de lecture et d'écriture dans ces fichiers activent les mécanismes physiques adéquats (L'écran ou le clavier sont ainsi considérés sous Unix comme des fichiers spéciaux).

Gestion de processus

Il est important de différencier **programme** (fichier exécutable) et **processus** (programme en cours d'exécution). Un processus a besoin non seulement d'un programme, mais également des ressources de la machine physique :

- Le processeur sur lequel s'exécute le programme,
- La mémoire dans laquelle il est stocké, ainsi que les variables nécessaires,
- Les périphériques : écran, clavier, souris, imprimante, disques, etc.

Unix peut gérer plusieurs processus simultanément, mais les ressources restent limitées au nombre de processeurs, à la quantité de mémoire disponible, et à certains périphériques communs non partageables (écrans ou imprimante).

D'où la nécessité d'optimiser les ressources. Le processeur est alloué par tranche de temps à chaque processus (multiprogrammation). Une partie de la mémoire est stockée sur disque sous forme de mémoire virtuelle, afin de permettre aux processus de manipuler plus de mémoire que de mémoire réelle.

Interface machine/système

L'interface machine/système permet un développement indépendant de la machine physique. Le coût du portage sur une nouvelle machine est seulement la réécriture de l'interface. Unix est maintenant sur toutes les machines.

Perspectives

Le système d'exploitation est un **gestionnaire de ressources**. Il nécessite de nombreuses qualités : fiabilité, efficacité, sécurité, simplicité.

Les systèmes sont de plus en plus complexes et les systèmes d'exploitation doivent s'adapter à la gestion de nouvelles fonctions telles que les systèmes multiprocesseurs, les traitements parallèles (plusieurs instructions en même temps), et les ressources distribuées.

Un système d'exploitation devra faire face aux difficultés suivantes : gérer de plus en plus de ressources, des ressources qui seront de moins en moins locales…

Mécanismes d'Exécution et de Communication

Définitions

Les **mécanismes d'exécution et de communication** ont le rôle de partager l'activité entre des tâches multiples, qui doivent assurer la prise en main du système ainsi que la multiprogrammation. Ils doivent également interagir avec le monde extérieur (entrées/sorties).

Le contexte d'une activité est l'ensemble des informations accessibles au cours d'une exécution. Elle est composée de deux contextes :

- Le contexte « processeur » :
  Registres programmables et mot d'Etat ;
- Le contexte « mémoire » :
  Segment de données et segment de programme.

Le **mode d'Etat du processeur** donne des informations sur les données accessibles et les droits sur celles-ci (table des segments et protection mémoire) ; des informations sur le déroulement de l'activité en cours (compteur ordinal, et code condition, pour avertir des erreurs et des dépassements) ; et surtout des informations sur l'état du processeur :

- Etat d'exécution : actif ou attente ;
- Mode d'exécution : maître ou esclave. Le système d'exploitation passe dans l'état maître pour l'exécution d'opération d'entrées/sorties demandées par l'utilisateur, puis repasse en mode esclave.
- Masque des interruptions : le système d'exploitation peut empêcher l'utilisateur d'interrompre une opération particulière qui mettrait par exemple en jeu l'intégrité du système. Les interruptions sont dans ce cas là traitées ultérieurement.

Le **contexte** représente en fait l'état d'une activité à un moment précis. Si ce contexte est sauvegardé intégralement, puis restauré ultérieurement, l'activité interrompue reprendra sans dommages. C'est de cette manière que le processeur sera partagé entre plusieurs processus fonctionnant en parallèle.

C'est ainsi que l'on pourra gérer l'**asynchronisme**, c'est-à-dire que les évènements extérieurs pourront influer un processus, car pour les traiter, il faudra bien interrompre le processus en cours. De même, on pourra pratiquer la multiprogrammation, c'est-à-dire que le processeur sera partagé entre les différents processus, par tranches de temps.

Pour gérer le **partage du processeur**, on peut utiliser deux méthodes de mesure du temps. Le processeur peut par exemple contrôler l'heure à chaque instant, ce qui est peu rentable. Une horloge externe peut également agir sur le processeur dans certaines conditions, ce qui laisse le processeur exclusivement à son traitement.

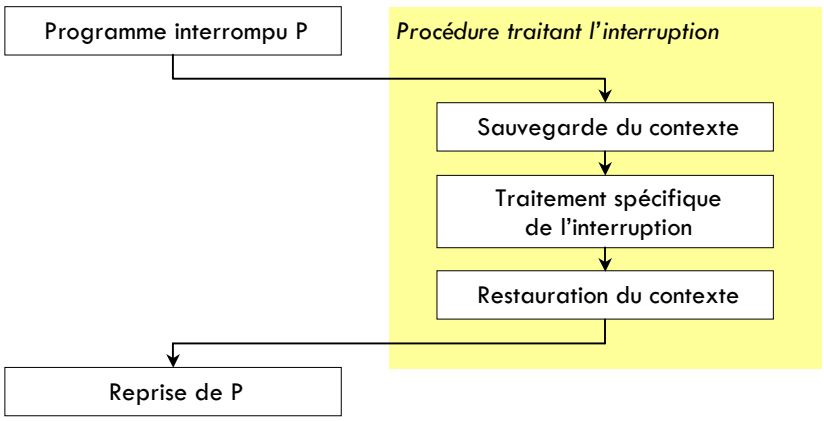
Une intervention extérieure, comme l'utilisateur, peut également interrompre le processeur à n'importe quel moment, par sécurité.

Commutation de contexte

Le **changement de contexte** s'effectue par un mécanisme appelé commutation de contexte. Elle consiste en deux étapes : rangement du mot d'état du processeur, et chargement d'un nouveau mot d'état.

Trois évènements peuvent générer une commutation de contexte :

- **Interruption** : Pendant que le processeur est attelé à la tache, un événement extérieur à l'activité en cours survient. Cet événement est vu comme un signal envoyé au processeur, ce qui force le processeur à réagir de façon asynchrone (pour la réalisation d'entrées/sorties ou la multiprogrammation).



- **Déroutement** : Il est provoqué par l'instruction en cours. C'est un moyen de signaler une anomalie dans le déroulement d'une instruction (débordement, violation d'accès mémoire, division par zéro, etc.).
- **Appel au superviseur** : Il sert à l'utilisateur pour demander au système des services, comme appeler une fonction système. Ceci entraîne le changement de mot d'état du processeur, car le mode d'exécution passe de esclave à maître.

Le processeur est ainsi interrompu à tout moment, et le système est chargé de gérer ces interruptions et doit faire en sorte que les activités s'effectuent parallèlement et sans heurt.

Processus

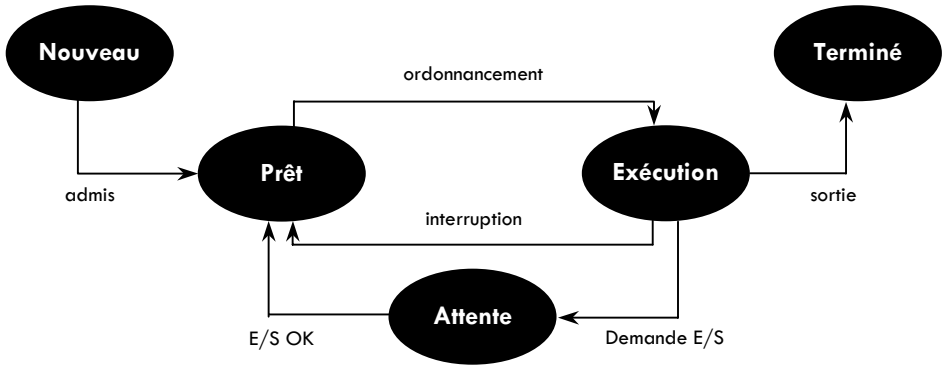
Définition

Un **processus** est fondamentalement différent d'un programme. Un processus a besoin d'un programme, mais également de ressources, allouées au début ou en cours d'exécution (CPU, mémoire, fichiers, périphériques).

Le système d'exploitation doit créer et détruire sans cesse des processus. Pour cela, il doit :

- Ordonnancer les processus : choisir celui qui s'exécute et ceux qui attendent ;
- Synchroniser les processus : assurer leur coopération ;
- Communiquer ;
- Gérer les problèmes d'inter-blocages.

Un processus passe par de nombreux états au cours de sa vie :

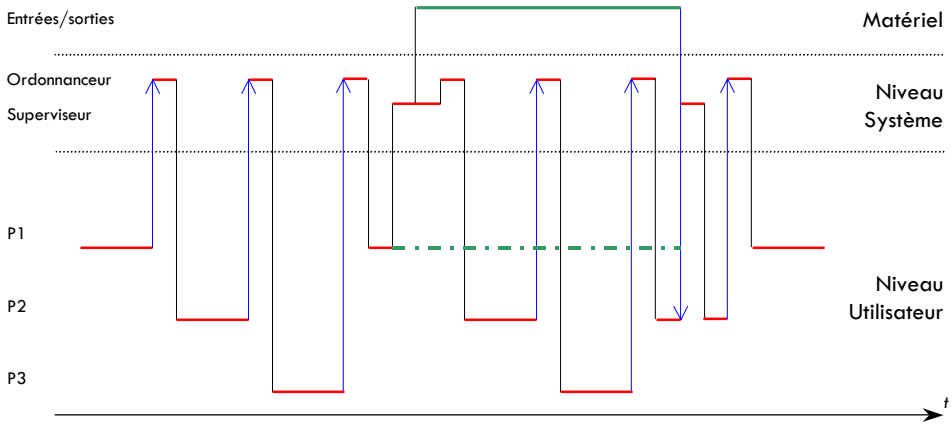


Le **bloc de contrôle** est composé des éléments suivants : l'état du processus, le compteur ordinal, les registres, les informations pour l'ordonnanceur, des informations pour la gestion de la mémoire, divers compteurs (numéro du processus, temps CPU), et l'état des entrées/sorties (listes des entrées/sorties, listes des fichiers ouverts).

Ordonnanceurs

Le système d'exploitation permet à plusieurs processus d'être chargés en même temps, mais la CPU n'est donnée qu'à un seul processus à la fois. Elle est alternativement donnée à chaque processus par l'**ordonnanceur**.

Ce schéma illustre l'ordonnancement de trois processus, avec préemption. Le premier processus fait appel à un moment donné à une entrée/sortie.



Le système gère différentes listes de processus :

- La liste des nouveau processus,
- La liste des processus prêts,
- La liste des processus en attente d'une entrée/sortie,
- La liste des processus en attente d'un événement.

L'**ordonnanceur à long terme** détermine quels processus faire passer de l'état 'nouveau' à l'état 'prêt'. Il mesure le degré de multiprogrammation par le nombre de processus actuellement prêts. Il peut se permettre d'attendre un peu afin d'optimiser l'utilisation du processeur. L'ordonnanceur à long terme est chargé d'équilibrer les processus d'entrées/sorties, qui ne sollicitent pas beaucoup le processeur puisqu'ils sont la majorité du temps en attente ; et les processus de calcul, mobilisant beaucoup plus de ressources CPU. Il est à noté que l'ordonnanceur à long terme est rarement présent sur les systèmes à temps partagé, où l'ajustement est géré par les utilisateurs, ce qui peut conduire à un manque de ressources et à un effondrement des ressources.

L'**ordonnanceur à moyen terme** supprime certains processus de la mémoire centrale pour réduire le degré de multiprogrammation. Ces processus seront repris plus tard. On parle alors de 'swapping'.

Enfin, l'**ordonnanceur à court terme** fait passer les processus de l'état 'prêt' à l'état d'exécution. Il est souvent sollicité et nécessite une décision rapide, contrairement à l'ordonnanceur à long terme qui dispose de plus de temps.

Opérations sur les processus

Un processus est toujours créé par un autre processus. Il y a donc une relation père/fils entre les processus, relations qui forment un véritable arbre de processus. Lorsqu'un processus vient d'être créé, il a deux alternatives :

- Le fils et le père continuent leur exécution en parallèle ;
- Le père attend la fin de son fils.

A la fin d'un processus, celui-ci est détruit. Il retourne des données à son père, et le système d'exploitation récupère toutes les ressources allouées à ce processus pour les libérer et les attribuer à d'autres processus.

L'ordonnancement

Principe

L'**ordonnanceur** est le programme des base des **systèmes multi-programmés.** Historiquement, il y a 20 ans, il y avait un seul utilisateur pour un ordinateur, et un seul processus à la fois. Puis des ingénieurs ont travaillé pendant des mois pour réaliser qu'il serait plus rentable d'avoir plusieurs processus en même temps. Problème : un processeur ne peut exécuter qu'un seul processus à un moment donné. D'où le rôle de l'ordonnancement !

Le principe de l'ordonnancement est que lorsque un processus, au cours de son exécution, est amené à attendre, par exemple une entrée/sortie, la CPU devient inactive pour ce processus. Ce processus est alors suspendu, et un autre processus s'exécute, comme nous l'avons vu au chapitre précédent.

C'est justement l'ordonnanceur qui choisit, dès que la CPU est libre, quel est le processus suivant parmi tous les processus prêts. L'ordonnanceur qui effectue cette partie du travail à un nom particulier : c'est l'**ordonnanceur à court terme**.

L'ordonnanceur intervient en fait dans les circonstances suivantes :

- Passage de l'état d'exécution à l'état d'attente (pour une entrée/sortie) ;
- Passage de l'état d'exécution à l'état prêt ;
- Passage de l'état d'attente à l'état prêt ;
- Fin d'un processus.

Certains ordonnanceurs, assez anciens il est vrai, sont dits **sans préemption**. Ils ne gèrent pas la deuxième circonstance ci-dessus (état exécution à état prêt) car ils attendent que le processus ait terminé sa tache ou demande une entrée/sortie pour donner la CPU à un autre processus. Par contre, les ordonnanceurs **avec préemption** interrompent de façon périodique grâce à une horloge le processus en cours, pour laisser la place à un autre processus.

Pour juger de l'efficacité d'un ordonnanceur, on peut l'évaluer sur les critères suivants :

1. Utilisation de la CPU ;
2. Nombre de processus rapporté à une unité de temps ;
3. Temps d'exécution : c'est le temps réel qu'il faut à un processus pour s'exécuter ;
4. Temps d'attente : c'est le temps qu'un processus passe dans l'état prêt ;
5. Temps de réponse : c'est le temps qui s'écoule entre la soumission et la première réponse.

Bien entendu, tout ordonnanceur censé s'efforcera de maximiser les deux premiers points, et de minimiser les trois derniers critères.

Dans les **systèmes interactifs**, il est considéré comme plus important de minimiser la variance (la variance donne une idée de la distribution autour d'une moyenne, i.e. une variance nulle signifierait que tous les temps sont égaux) du temps de réponse, plutôt que de minimiser le temps lui-même. On obtient ainsi un temps de réponse plus prévisible, que l'on préfère à un temps plus court en moyenne mais aussi plus variable…

Algorithmes d'ordonnancement

Nous allons étudier différents algorithmes permettant de déterminer quel processus exécuter avant l'autre, et évaluer leur efficacité respective.

> Algorithme FIFO sans préemption

Cet algorithme est basé sur une file d'attente FIFO (*First In First Out*). Le premier processus arrivé est exécuté d'abord, puis le suivant, etc., sachant que le suivant ne peut démarrer que si le processus en cours est terminé. Le temps d'attente résultant est malheureusement assez long en moyenne, notamment si les gros processus se présentent en premier, comme dans l'exemple suivant.

	T. Exécution	*T. Attente*
P1	*24*	*0*
P2	*3*	*24*
P3	*3*	*27*

Moyenne : 17

> Algorithme Le Plus Court d'Abord

Pour remédier au problème posé par l'algorithme précédent, il est judicieux de faire passer les processus les plus courts d'abord. Pour cela, le temps CPU du dernier passage est conservé, et le processus suivant est celui de la liste qui a utilisé le moins la CPU la dernière fois.

Pour l'ordonnanceur à long terme, chaque utilisateur doit estimer le temps d'exécution du programme. Bien sûr, en cas de dépassement, le travail est arreté, puis repris plus tard, pour éviter la fraude…

L'ordonnanceur à court terme ne peut pas savoir le prochain temps d'utilisation de la CPU. Il utilise donc une prédiction, basée sur la formule suivante :

$$\Gamma_{n+1} = \alpha \times t_n + (1-\alpha) \times \Gamma_n$$ où Γ_i représente la prédiction à l'instant *i*.

Le paramètre α est compris entre 0 et 1, et donne plus ou moins d'importance à la prédiction ou au temps d'utilisation CPU précédent *t* selon sa valeur.

Cet algorithme est également envisageable dans un système avec préemption. Si un processus arrive par soumission ou par sortie de file d'attente dans la file 'prêt' avec une durée inférieure à celui qui s'exécute, la CPU lui est donné immédiatement.

Ordonnancement avec priorité

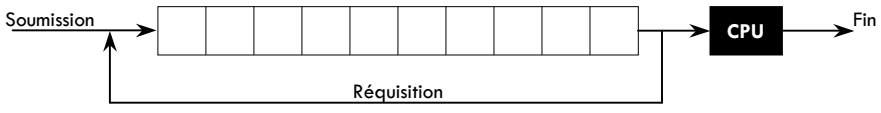
Chaque processus possède une **priorité**, et la CPU est donnée à celui qui possède la plus grande priorité. Dans l'algorithme Le Plus Court d'Abord, avoir une priorité élevée revient à avoir un temps très faible.

Les priorités sont déterminées par le système d'exploitation, et dépendent de plusieurs facteurs, tels que l'utilisateur qui lance le processus (un super-utilisateur aura plus de privilèges qu'un simple étudiant…). Il existe une commande sous Unix, *nice*, qui permet de modifier la priorité d'une processus en mémoire. Malheureusement, cette commande ne peut que baisser la priorité.

Dans un système sans préemption, le nouveau processus est inséré dans la liste 'prêt', qui est ordonnée par priorité ; alors que dans un système avec préemption, la CPU est réquisitionnée si le nouveau processus a une priorité plus grande que celui qui s'exécute actuellement.

Le tourniquet

Le **tourniquet** est une méthode d'ordonnancement à court terme avec priorité, pour les systèmes avec préemption. Le processus est interrompu au bout d'un **quantum**, et un autre prend sa place. La liste 'prêt' est une liste FIFO.



Les performances dépendent de la valeur du quantum. Si cette valeur est trop élevée, on revient dans le cas d'un ordonnancement FIFO simple, sans préemption. Si cette valeur est trop courte, on perdra du temps dans les changements incessants d'activité. Par exemple, si la commutation de contexte occupe 10% d'un quantum, la CPU ne sera utilisée effectivement qu'à 90%.

Le tourniquet multi-niveaux

Le **tourniquet multi-niveaux** est basé sur la constatation que le temps de réponse demandé est différent selon les processus. On classe donc les processus en deux niveaux : arrière-plan (*background*) et avant-plan (*foreground*).

Pour traiter ces deux types de processus, on dispose de deux solutions :

- Première alternative : avoir plusieurs files, une par classe de processus, sachant que la file des prioritaires en avant-plan est prioritaire ;
- Seconde alternative : utiliser des quanta différents selon la classe du processus, par exemple 80% pour les processus en avant-plan, et 20% pour les processus en arrière-plan.

En généralisant ce principe, on peut ainsi élaborer un **tourniquet multi-niveaux deuxième version**, qui propose de multiples files d'attente déterminées par le système, et non par l'utilisateur, sachant que les processus les plus cours resteront dans les couches basses, de niveau *n* ou *n-1*, et que les processus plus longs monteront au fur et à mesure de leur réquisition dans les couches les plus hautes. Comme précédemment, ce sera exclusivement le système qui estimera le type du processus.