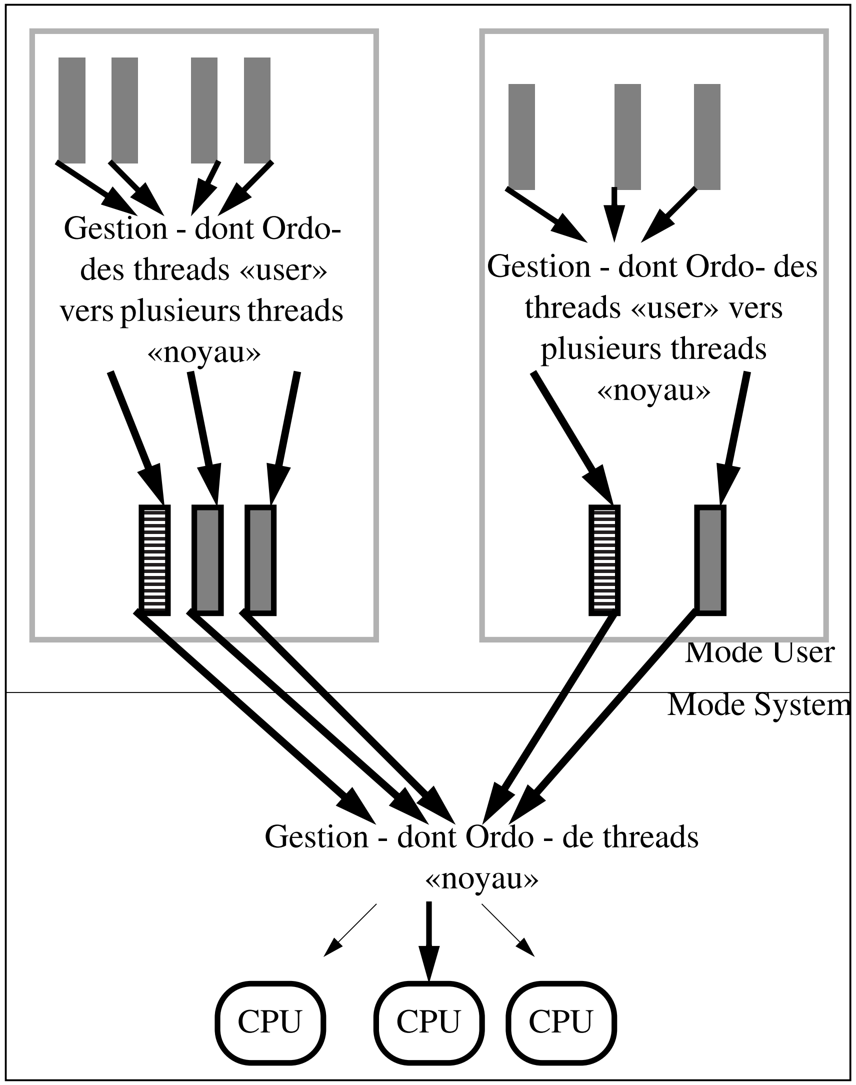
b3) Threads Utilisateur + Threads «Noyau»



•> **solution MIXTE combinant avantages des 2 précédantes :**

- si une thread utilisateur se bloque, ce n'est que la thread noyau qui se bloque (d'autres threads noyau du processus restent éligibles)
- Certaines fonctions de gestion de threads peuvent être prises en charge par la librairie de threads (ex: blocage d'une thread pour synchro) sans avoir besoin de faire de SVC.
- C'est la librairie de threads utilisateur - et non le programmeur - qui peut se charger d'obtenir des nouvelles threads noyau

•> **Ex : Threads de niveau utilisateur du langage Java lorsque la JVM implantée sur Solaris**

Threads de la librairie Posix pthread implantée sous Solaris (voir TPs)

- **•> Le processus en tant que tel n'est pas une entité d'exécution (il sert juste à fournir un espace d'adressage)**

  Chaque processus a au moins 1 thread «noyau»

- **•> Le scheduler gère les diverses threads sans chercher forcément à savoir à quel processus elles sont dédiées.**

- **•> L'ordonnancement des threads noyau est basé sur une politique semblable à l'ordonnancement de processus dans des S.E. classiques :**
  - Préemptif et à temps partagé, avec modif. dynamique des priorités.

- **•> Les threads de cette nature sont souvent appelées des Light Weight Processes («light weight» car contexte d'un LWP plus léger que celui d'un processus)**

  - Ex : sous SunOS 5.x (Solaris), on parle de LWP, et fonctions de gestion (_lwp_**) font des SVCs (_lwp_create, _lwp_exit, _lwp_mutex_lock, ...)
  - Sous NT, les threads noyau s'appellent des threads, alors que les threads utilisateur s'appellent des «fibers»

- **•> Avantages) Une thread se bloque, =>d'autres threads du processus peuvent être élues (si plusieurs CPUs, véritable parallélisme au sein du processus)**

- **•> Inconvénients) coûteux que chaque primitive fasse un SVC**

Lorsque le processus est ELU, c'est la thread qu'il a élue qui s'exécute

La thread BLOQUE ==> le processus BLOQUE

- **•> Les threads sont créées par le biais d'une librairie dédiée (ex, POSIX 1.c pthread), voire «intégrées» au langage (ex. Java)**

fonctions pthread_create, pthread_cancel, pthread_yield, pthread_mutex_lock, ...

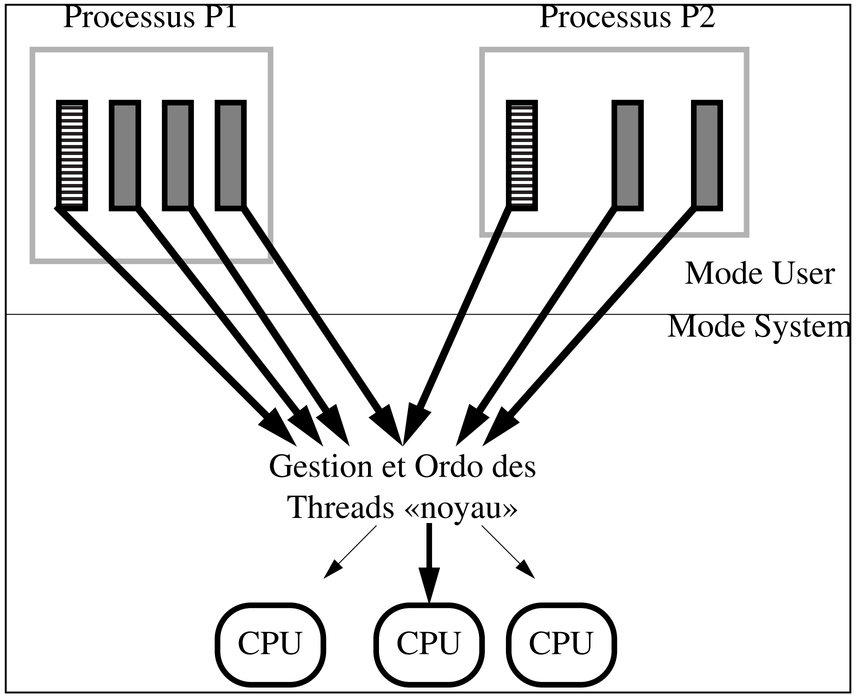
- **•> L'ordonnancement de threads DANS le processus est souvent simple**
  - FIFO + priorité,
  - sans réquisition ou bien préemption aux points d'appels de fonctions de l'API «thread»
  - pas de modification dynamique des priorités

b2) Threads de niveau «Noyau»

- **•> La gestion et le support des threads sont faits (codés) entièrement par l'exécutif (dont noyau).**

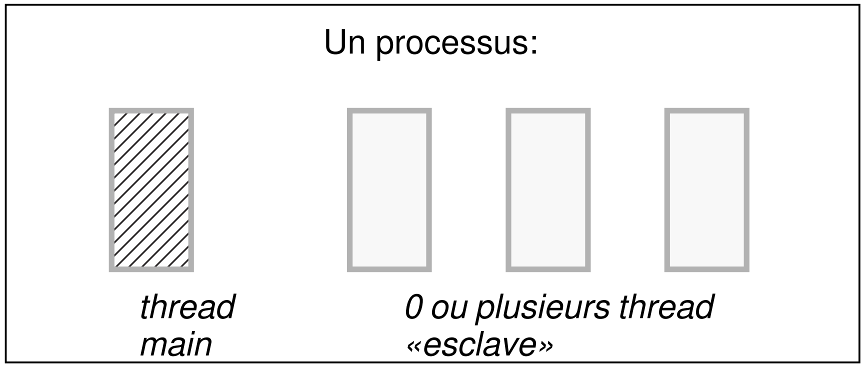
==> ceci est codé dans l'**espace d'adressage système** du processus et nécessite donc que le CPU soit en mode d'exécution System

==> Du point de vue du processus, primitives de gestion de threads ne sont que des encapsulations de SVCs



b) Différentes Mises en Oeuvre du «multi-thread»

•> **Un processus a au moins 1 thread (principale) et peut en créer de nouvelles**



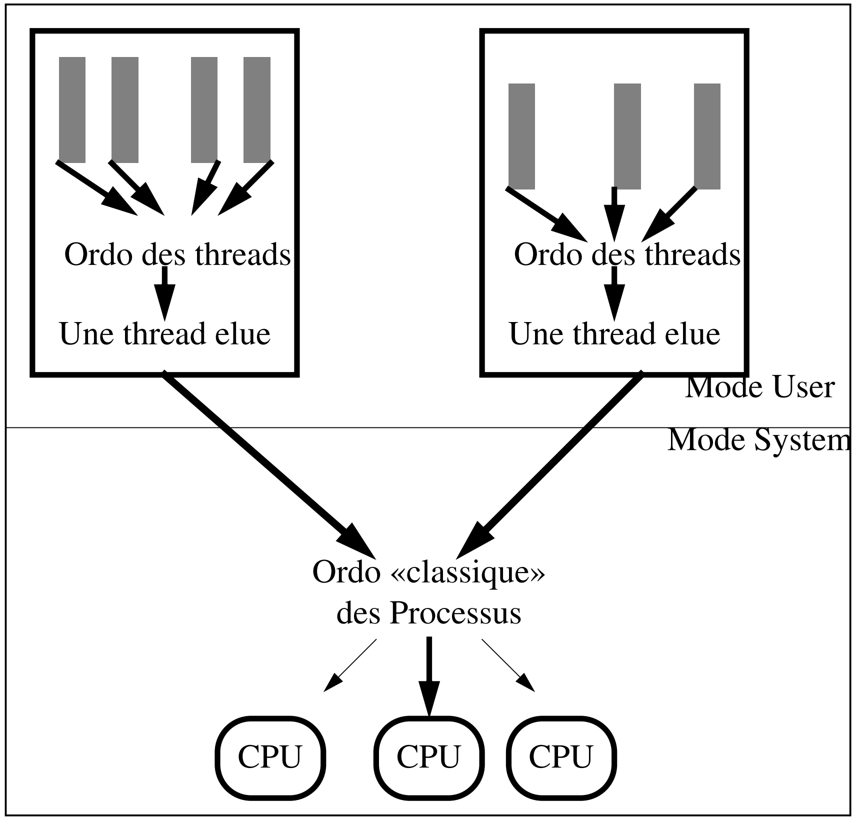
•> **Les threads du processus ont accès à toutes les zones de son espace d'adressage (mais elles ont une pile privée)**

- **Plusieurs natures de threads existent**
  - inconnues du SE et connues du processus seul qui les gére (les crée, les ordonnance, les synchronise, les termine)
  - connues, gérées et ordonnancées par l'exécutif
  - connues du processus mais supportées par des threads de l'exécutif : solution mixte

b1) Threads de niveau «Utilisateur»

•> **La gestion et le support des threads sont faits entièrement par le processus.**

==> tout est codé dans l'**espace d'adressage utilisateur** du processus et ne nécessite pas que le CPU soit en mode système (privilégié)



4) Ordonnancement dans les systèmes Multi-Thread

a) Justification du «multi-thread»

•> Multi-thread == Activité == Fil / Flot / Chemin d'exécution

•> Multi-thread ==> avoir Plusieurs fils d'exécution dans un même Programme ==> Plus facile de coder des applications ayant plusieurs activités concurrentes (c.a.d. pouvant se dérouler dans un ordre quelconque)

Exemples :

- Serveur (httpd, nfsd, X, ...), qui à chaque requête d'un client lui associe un thread pour la servir
- L'explorateur sous Windows
- nous sommes nous-mêmes multithreadés ... bien que peu parallèles

Exemple d'un Serveur

- **Programmé SANS threads :**
  - Attendre une requête
  - Analyser la requête
  - Servir la requête
  - Renvoyer réponse au client

  Si «servir requête» se bloque (ex. pour E/S), TOUT le serveur est bloqué

- **Programmé AVEC threads :**

  Thread Principale **:**
  - Attendre une requête
  - Analyser la requête
  - ALLOUER une thread au Service

  Une Thread esclave (Requête R1)**:**
  - Servir Requête R1 /* *peut être bloquant* */
  - Renvoyer réponse au client

  Une Thread esclave (Requête R2)**:**
  - Servir Requête R2 /* *peut être bloquant* */
  - Renvoyer réponse au client

Exemple : Ressource = CPU

- **Lors de tics principaux, ajustement des priorités dynamiques de TOUTES les entités Prêtes, selon formules +/- élaborées fonction de**
  - la priorité initiale (statique)
  - du nombre tics horloge d'utilisation du CPU depuis le dernier ajustement : moins il y en a eu, plus la priorité augmente
  - du nombre total de tics horloge d'utilisation du CPU
  - charge du système (nbre de processus prêts, ...)
- **OU bien, Ajustement de la priorité de l'entité élue**
  - Diminution de la priorité dynamique si quantum a été utilisé en entier
  - Augmentation de la priorité dynamique si l'entité se bloque volontairement (ex, E/S) (quantum non utilisé en entier)

c) Préemptif ET à temps partagé

•> **Avant tout, PREEMPTIF (files FIFO par priorité)**

•> **TOURNIQUET à l'intérieur d'une FIFO**

•> **Passage à File FIFO de priorité inférieure quand TOUTES celles de niveau supérieur sont VIDES**

- Si l'entité élue est la plus prioritaire, elle peut garder la ressource plus longtemps que la durée du quantum

Politique mise en oeuvre dans les SE dits multiprocessus à temps partagé tels Unix, NT

Possible d'avoir une valeur de quantum PAR niveau de priorité

b) Préemptif (basé sur des priorités)

<u>Préemption</u> : faculté que détient une personne ou une administration en vertu de la loi ou d'un contrat d'acquérir un bien de préférence à tout autre

•> A tout instant (en théorie) la ressource est détenue par l'entité de plus forte priorité

1. ==> Retirer la ressource à l'entité la possédant <u>dès que</u> celle-ci n'est pas la plus prioritaire

<u>Exemple : Ressource = CPU</u>

Mise en oeuvre réaliste du «dès que» ...

- **L'exécutif peut tester la priorité d'une entité qui naît ou qui se réveille, puisque c'est lui qui s'en charge et décider alors d'une commutation.**
- **Si commutation retardée, on parle d'INVERSION de PRIORITE (la ressource n'est pas détenue par l'entité la plus prioritaire ...)**

2. FAMINE des entités à faible priorité (encore plus importante que dans stratégie sans réqusition)

==> Ajustement DYNAMIQUE des priorités.

- Provoque de nombreux changements dans les files de priorité (gestion de files doit être efficace)

Analyse/Remarques

•> **COMPROMIS :**

- Actions dans traitant de tic horloge sont non négligeables, =>ne pas les effectuer trop souvent
- Si quantum trop élevé, mauvaise réactivité du système

•> **Durée exécution des traitements de tics horloge imputée au processus élu**

•> **Lorsque quantum épuisé, le processus était peut-être en cours d'exécution d'une fonction de l'exécutif NON REENTRANTE :**
Si le nouveau/futur processsus élu demande exécution de la même fonction système, problèmes possibles

- réutilisation d'une même variable globale de l'exécutif
- insertion d'un élément dans une liste chainée, nécessitant plusieurs opérations
- écriture dans un fichier

- RETARDER (grâce à des primitives de **synchronisation, cf. chapitres 3-4)** la commutation vers la nouvelle entité élue jusqu'à ce que celle en cours atteigne un POINT de COMMUTATION

Points de commutation :

- Un seul en sortie du mode noyau après un SVC
- ou bien, Plusieurs aménagés à l'intérieur de l'exécutif (cas de SE «plus temps-réel»)

- => **Un tic horloge -- clock tic -- == une «interruption horloge niveau système»**

- **Le quantum de temps pour l'utilisation de la ressource est fixé par X tics horloge, couramment X=10**

  `Suite de l'ex : Quantum = X * 10 msec = 10 * 10 msec = 100 msec = 1/10ème de sec`

Mise en oeuvre du partage du temps CPU

- **Fonctions du traitant des tics horloge («Interruptions Horloge»)**

  1. Décompte temps occupation CPU initialisé à la valeur du quantum
  2. Si temps occupation écoulé : routine ordonnancement est invoquée (**ICI**, élection simple de la prochaine entité : celle en tête de FIFO),

     `ICI : tourniquet simple`

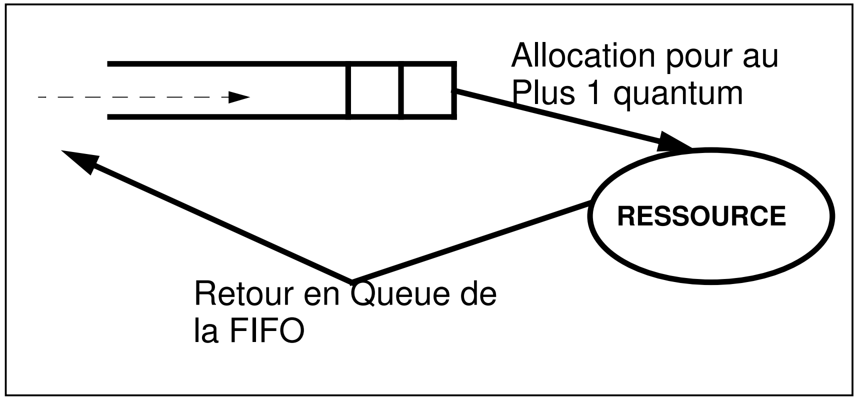
     `Paragraphes suivants : montrer d'autres politiques d'élection`

  3. Autres actions non liées à l'ordonnancement
  4. Actions liées à l'ordonnancement effectuées seulement à des **tics principaux** (ex, tous les 4 tics)
  5. Commutation de contexte vers l'entité élue

a) Tourniquet (Round Robin -- RR)

•> **Fournir à l'une quelconque des n entités en attente, 1/n ème du temps d'utilisation de la ressource.** (ex, si 7 processus, à chacun 1/7 ème du temps utilisation ressource)

•> **n variant dynamiquement, on décide à l'avance d'un QUANTUM (tranche) de TEMPS -- time slice.**



•> **La ressource est allouée à l'entité en tête de file pour au maximum la durée du quantum.**

•> **Quand elle lui est retirée, l'entité rejoint la file (par la queue, si quantum épuisé)**

Exemple : Ressource = CPU

Définition du quantum

•> **Horloge du CPU génère une interruption horloge périodiquement**

•> **Le S.E. peut aussi «fixer» la fréquence de la prise en compte de ce phénomène**

```
Ex du fichier param.h d'un système Unix

var HZ = 100 /* 100 int. horl. par sec =>
1 int. horl toutes les 1/100 ème de sec ==
1 int. horl. toutes les 10 msec */
```

3) ORDONNANCEMENT AVEC REQUISITION

Motivation/Description

Stratégies sans réquisition incompatibles avec activités interactives, et encore plus, avec activités temps réel.

- ==> Partager le temps d'utilisation de la ressource d'exécution, modulo certaines contraintes de priorité.

- ==> Ceci se traduira par une amélioration du critère *temps de réponse maximum*, mais attention à la déterioration possible du critère *temps de traitement moyen.*

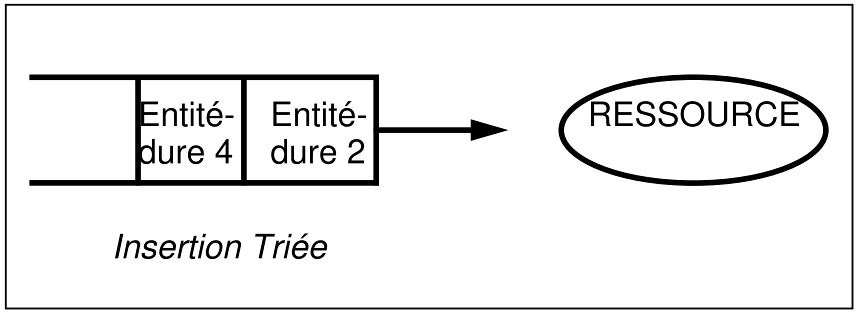
Mise en oeuvre/Exemple

•> A des moments autres que celui de la libération de la ressource, la main est donnée à la routine qui code l'algorithme d'ordonnancement.

•> Toutes les politiques d'allocation du CPU dans des Syst. d'exploitation *généralistes* dits *à temps partagé* (plus précisément à temps CPU(s) partagé(s)) rentrent dans cette catégorie.

b) Politique Plus Court Temps d'Utilisation d'abord

Allocation selon l'ordre croissant de durée d'utilisation prévue de la ressource



•> Inconvénient :

- Pas REALISTE, car difficile de prévoir la durée d'utilisation, surtout dans un système non dédié.
- Famine (Privation) pour entités de longue durée d'itilisation

`=> le temps de réponse maximum est élevé`

•> Avantage :

- Temps attente faible pour entités à courte durée d'utilisation de la ressource

`=> temps moyen bon`

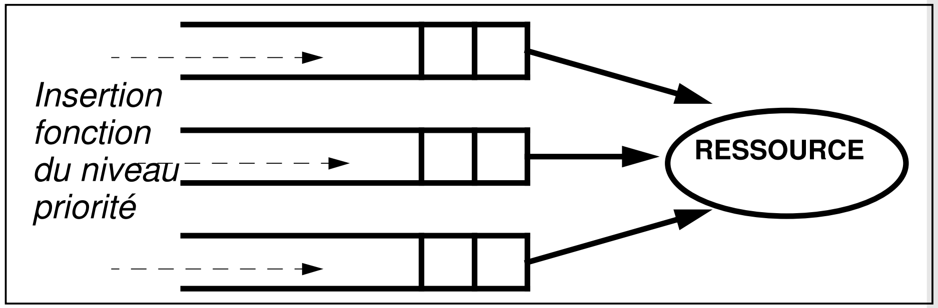
c) Politique FIFO avec priorités

1. Chaque entité a une priorité

2. Une FIFO Par niveau de priorité

3. Ressource allouée à une entité ssi

- FIFOs de priorité supérieure sont vides
- ET l'entité est en tête de sa FIFO



- **Inconvénient : Famine pour les entités de faible priorité.**

`=> en pratique, utilisation parcimonieuse des priorités élevées`

`=> Politique couplée avec MODIFICATION DYNAMIQUE des niveaux de priorité.`

2) ORDONNANCEMENT SANS REQUISITION

Description

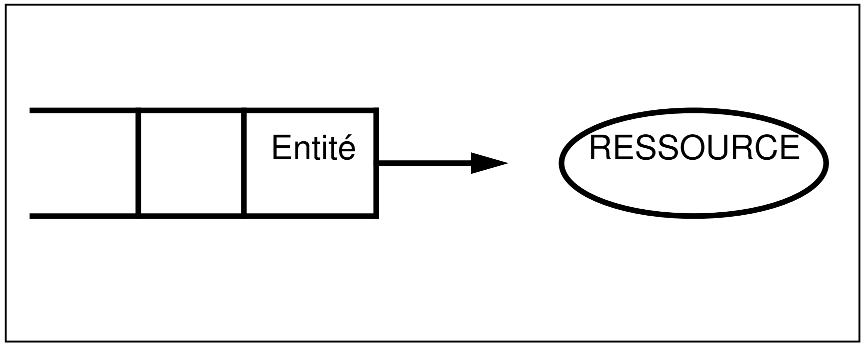
Ressource allouée à une entité jusqu'à ce qu'elle n'en ait plus besoin. Ne peut convenir aux activités interactives ou temps-réel.

Mise en oeuvre

- Au moment de la libération de la ressource, la main est donnée à la routine qui code l'algorithme d'ordonnancement.
- La routine d'ordonnancement choisit l'entité et provoque une commutation vers celle-ci pour lui donner la main.

a) Politique FIFO

Allocation selon l'ordre d'arrivée des entités dans la file d'attente pour l'obtention de la ressource



•> **Inconvénient : Défavorise les entités ayant besoin d'utiliser la ressource pendant un temps court (ex. petits programmes)**

`=> le temps moyen de traitement d'une entité est mauvais`

Def : Rôle d'un ALGORITHME d'ORDONNANCEMENT

DECIDER de l'Allocation de la ressource aux entités qui l'attendent, de sorte à atteindre certains objectifs.

Classification d'un Algorithme d'Ordonnancement (scheduling)

- **•> SANS REQUISITION**

- **•> AVEC REQUISITION**

==> La ressource dont l'algorithme gère l'allocation peut être réquisitionnée au profit d'une autre entité.

Remarque : «Entité d'exécution «

- Processus qui a besoin de la ressource d'exécution CPU
- Fil d'activité à l'intérieur d'un processus qui a besoin de la ressource d'exécution PROCESSUS

Mécanismes de Base nécessaires

- **•> Commutation de Contexte, pour permettre à une autre entité d'utiliser la ressource**

==> Contexte en partie matériel si la ressource est matérielle

Pour Algorithmes AVEC Réquisition

- **•> HORLOGE : pour compter le temps , si durée d'allocation de la ressource est importante**

- **•> Interruption PERIODIQUE AUTOMATIQUE fournie gratuitement par Interruption Horloge**

==> Moyen pour exécuter l'algorithme d'ordonnancement et ainsi, réquisitionner la ressource

Chapitre II

ALLOCATION d'une RESSOURCE (en particulier, le CPU)

1) EXPOSE du PROBLEME

•> Justification SE. multiprocessus :

Recouvrir

- phases attente des processus (dus aux E/S par ex)

par

- phases de calcul d'autres processus

Pour Maximiser

$$TauxUtilisationCPU = \frac{DureeCPUactif}{DureeTotale}$$

et par là même,

$$Debit = NombreProcessusTraites/(UniteTemps)$$

Solution Simple : le SE commute vers un processus prêt dès que le processus élu se bloque

•> Ne convient pas aux travaux interactifs (!= jobs en batch)

Optimiser d'autres critères comme

- Minimiser le TEMPS de TRAITEMENT MOYEN (moyenne durées entre soumission des travaux et leur accomplissement)
- Minimiser/borner le TEMPS de REPONSE MAXIMUM (maximum durées entre soumission des travaux et leur accomplissement)

Ces critères peuvent s'avérer être ANTAGONISTES ...

Il faut trouver des politiques de COMPROMIS

•> Clair besoin d'allouer JUDICIEUSEMENT la ressource (ici, par ex, CPU) aux entités qui en ont besoin pour s'exécuter ... ORDRE d'ALLOCATION est important.