Ordonnancement

Olivier Dalle
Université de Nice - Sophia Antipolis
http://deptinfo.unice.fr/

Systèmes d'Exploitation : principes et algorithmes

II. Ordonnancement

1. **Introduction**
2. Ordonnancement sans réquisition
3. Ordonnancement avec réquisition
4. Ordonnancement dans les systèmes multi-thread

II.1 Introduction

Exposé du problème

- Dans un système multi-tâches :
  - De nombreux processus attendent qu'un événement se produise
    - Ils n'ont pas besoin du processeur…
    - … mais doivent pouvoir l'obtenir en priorité dès que l'événement attendu se produit
  - Certains processus souhaitent utiliser le CPU longtemps et de façon intensive
    - Ils souhaitent garder le processeur le plus longtemps possible
  - Conflit d'intérêt !
    - Ordonnanceur = arbitre + chef d'orchestre …

II.1 Introduction

Rôle, objectifs

- Rôle d 'un algorithme d'ordonnancement :
  *Décider de l'allocation de la **ressource** aux **entités** qui l'attendent, de sorte à atteindre certains **objectifs***
- « entités » :
  - Un processus
  - Un fil d'exécution à l'intérieur d'un processus (*thread*)
- Exemple : le CPU
  - Objectif : Aboutir à un partage **efficace** du temps d'utilisation du processeur
  - Problème : que veut dire **efficace** ? Efficace pour qui ?

II.1 Introduction

Exemple : Critères d'efficacité pour l'Ordonnancement du CPU

- Respect de la priorité
  - La plupart des systèmes permettent d'accorder des priorités différentes aux processus…
  - Priorité peut être statique ou dynamique (change au cours du temps) …
- Respect de l'équité
  - Deux processus qui ont le même niveau de priorité doivent pouvoir utiliser le CPU aussi souvent l'un que l'autre

O. Dalle 5

II.1 Introduction

Exemple : Métriques pour mesurer l'efficacité de l'Ordonnancement du CPU

- Utilisation maximale du processeur
  - maximiser

    TauxUtilisation(CPU) = Durée_Actif(CPU)/Durée_Totale
- Débit processus
  - maximiser

    Débit = NombreProcessusTerminés/UnitéTemps
- Temps de traitement moyen : minimal (traitement *batch*)
- Temps de réponse maximum : minimal (traitement interactif)

O. Dalle 6

Remarque

- On peut trouver plusieurs algorithmes d'ordonnancement indépendants :

  chacun gère une ou plusieurs ressources (pas les mêmes !) :
  - CPU (au minimum)
  - Disques, Réseau (en particulier à Qualité de Service), Mémoire (swap, pages), …
  - + outils externes de type « spooler » (imprimante, …)

Classification des algorithmes d'Ordonnancement

- Dans un monde « idéal » (statistiquement) :
  - le hasard fait bien les choses : les processus endormis ne se réveillent pas tous en même temps
- Dans la réalité :
  - les activités des processus sont « corrélées » : les processus ne se réveillent pas au hasard …

⇒ **2 familles d'algorithmes** :
- **Sans réquisition** : c'est aux processus de relâcher volontairement la ressource
- **Avec réquisition** : l'algorithme d'ordo. est capable de récupérer la ressource détenue par un processus au profit d'un autre

Mécanismes de base nécessaires

- Ressources de type CPU :
  - Commutation de contexte
    - Pour permettre à une autre *entité* d'utiliser la ressource
    - Contexte peut être en partie matériel (registres, état)
- Algorithmes avec réquisition :
  - Horloge
    - Pour mesurer la durée d'utilisation
  - Interruption **périodique automatique**
    - fournie « gratuitement » par interruption d'horloge
    - moyen pour exécuter l'algorithme à dates fixe et ainsi, réquisitionner la ressource

II. Ordonnancement

1. Introduction
2. **Ordonnancement sans réquisition**
3. Ordonnancement avec réquisition
4. Ordonnancement dans les systèmes multi-thread

II.2 Ordonnancement sans réquisition

Description

- Ressource allouée à une entité jusqu'à ce qu'elle n'en ait plus besoin
  - Par nécessité (ex: imprimante)
  - Par choix de conception (ex: windows 3.X, 95)

II.2 Ordonnancement sans réquisition

Défauts et qualités

- Inconvénients :
  - Ne peut convenir aux activités temps-réel
  - Convient difficilement aux activités interactives :
    - Obligation de programmer des applications « sociables »
    - Tolérable dans un système faiblement multi-tâches (Windows 3.x, 95, …)
- Avantages :
  - Facile à mettre en œuvre
  - Pas besoin de mécanismes matériels spécifiques…

II.2 Ordonnancement sans réquisition

Mise en oeuvre

- Au moment de la libération de la ressource :
  - Le (ex-)détenteur de la ressource invoque l'algorithme d'ordonnancement
    - Cette action peut être réalisée à l'insu du programmeur (exemple : win3x, win9x)
  - L'algorithme **choisit** l'utilisateur suivant
  - L'algorithme déclenche la **commutation de contexte**

O. Dalle

13

II.2 Ordonnancement sans réquisition

Politiques de choix

- Politique « FIFO » (*First In First Out*)
  - Allocation dans l'ordre d'arrivée (premier arrivé = premier servi)
  - Inconvénient : défavorise les entités ayant besoin d'utiliser la ressource un court laps de temps
    - Le temps d'attente n'est pas proportionnel au temps d'utilisation

      ⇒ pas équitable,

      ⇒ temps moyen de traitement élevé

O. Dalle

14

II.2 Ordonnancement sans réquisition

Politiques de choix (2)

- Politique PCTU (Plus Court Temps d'Utilisation d'abord)
  - Allocation selon ordre croissant de durée d'utilisation *prévue*
  - Inconvénients
    - **Pas réaliste** : exige la **connaissance a priori** des durées d'utilisation
    - **Famine** (privation) : les tâches dont la durée d'exécution estimée est longue peuvent attendre leur tour indéfiniment …
  - Avantages
    - Temps d'attente faible pour entités à courte durée d'utilisation
    - ⇒ temps moyen d'attente minimal

II.2 Ordonnancement sans réquisition

Politiques de choix (3)

- Politique FIFO avec priorités
  - Chaque entité a une *priorité*
  - Une FIFO par niveau de priorité
  - Ressource allouée à une entité Ssi
    - FIFOs de priorités supérieures vides
    - En tête de sa FIFO
  - Inconvénients
    - Famine pour entités de faible priorité
  - En pratique
    - utilisation parcimonieuse des priorités élevées
    - modification dynamique des niveaux de priorité

II. Ordonnancement

1. Introduction
2. Ordonnancement sans réquisition
3. **Ordonnancement avec réquisition**
4. Ordonnancement dans les systèmes multi-thread

Description

- Motivations
  - Politiques sans réquisition mal adaptées voire inadaptées à certaines activités
    - Temps réel
    - Interactivité
- Réquisition :
  - Forcer le partage du temps d'utilisation (modulo contraintes de priorités)
  - amélioration *temps de traitement maximum*
  - détérioration *temps de traitement moyen*

II.3 Ordonnancement avec réquisition

Mise en oeuvre

- Le détenteur de la ressource peut être interrompu avant d'avoir terminé
  - Lorsqu'un délai maximal expire
  - Lorsqu'une entité de priorité plus élevée demande la ressource
- La politique d'ordonnancement choisit la nouvelle entité
- L'entité interrompue est mise en sommeil
- Remarque : c'est la politique utilisée dans les systèmes « à temps partagé » (*time-sharing*)
  - Unix, NT, …

II.3 Ordonnancement avec réquisition

Mise en œuvre : Difficultés liées à l'interruption

- Un processus peut être interrompu alors qu'il exécute une fonction de l'exécutif
  - Problème si fonction non ré-entrante :
    - Le nouveau/futur élu peut demander à son tour l'exécution de la même fonction :
      - Réutilisation d'une même variable globale
      - Insertion non terminée dans une liste chaînée
      - …
  - Solution : retarder la commutation jusqu'à ce que l'exécution atteigne **un point de commutation**
- Points de commutation
  - cas « simple » (linux <= 2.1) : un seul, en sortie du mode noyau
  - cas « difficile » (linux >= 2.1) : plusieurs aménagés à l'intérieur de l'exécutif

II.3 Ordonnancement avec réquisition

Un grand classique : La politique du tourniquet

- Politique du Tourniquet (*Round Robin / RR*)
  - idée : Fournir à l'une quelconque des **n** entités en attente, 1/**n** ème du temps d'utilisation de la ressource
  - pb : **n** varie au cours du temps
  - Solution : le temps est découpé en tranches de taille (durée) identique, appelées **quantum de temps**
    - Les entités sont placées dans une file
    - La ressource est allouée à l'entité en tête de la file, pour une durée **d'au maximum un quantum**
    - Lorsque le quantum est épuisé, l'entité est interrompue et replacée **à la fin** de la file d'attente

II.3 Ordonnancement avec réquisition

Politique du tourniquet : Problème de choix du quantum

- Durée ni trop courte …
  - Lorsque la durée du quantum est écoulée, il faut déclencher un changement de contexte
  - Un changement de contexte prend du temps
  - ⇒ Plus le quantum de temps est petit, plus on perd souvent du temps à changer de contexte !
- … ni trop longue :
  - L'illusion d'exécution concurrente (parallèle) s'estompe
  - Lorsque la durée est trop longue, l'interactivité diminue
    - Exemple :
      Quantum = 1s, 3 tâches de longues durée sont présentes et attendent le processeur. Elles comptent utiliser systématiquement tout leur quantum.
      Une tâche interactive ne peut obtenir le processeur au mieux que toutes les 3 secondes (délai entre frappe clavier et affichage d'au moins 3 secondes …)

II.3 Ordonnancement avec réquisition

Politique du Tourniquet: Mise en œuvre du partage du temps CPU

- Horloge programmée
  - déclenche une interruption à intervalles de temps réguliers
    - interruption appelée « tic d'horloge » : tic, tac, …
  - Le traitant de l'interruption :
    1. Décompte du temps d'occupation CPU pour l'entité courante (initialisé à la valeur de quantum)
    2. Si temps restant = 0 : lancer algorithme d'ordonnancement pour choisir un nouveau processus
    3. Autres actions non liées à l'ordonnancement
    4. actions liées à l'ordonnancement effectuées à des **ticks principaux** (versions évoluées du tourniquet)
    5. Commutation de contexte vers entité **élue**

O. Dalle

23

II.3 Ordonnancement avec réquisition

Politique du tourniquet : Exemple d'application sous Linux

- Tourniquet évolué : Politique tourniquet + priorité dynamique
- Mode de fonctionnement du tourniquet :
  - Matériel (PC) : jusqu'à 3 horloges (selon génération du processeur) !
  - Linux utilise le **PIT** (*Programmable Interval Timer*)
    - L'interruption du PIT (un *tick*) déclenche un changement de contexte
    - Linux programme le PIT pour être interrompu 100 fois par seconde (donc un *tick* = 10 ms écoulées)
    - Le quantum est fixé à environ 20 ticks = 200 ms

O. Dalle

24

II.3 Ordonnancement avec réquisition

Politique d'ordonnancement préemptif (l'autre grand classique)

- Préemption :
  « 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 »
- En théorie : **A tout instant** la ressource est détenue par l'entité **de plus haute priorité**
  - Il faut donc retirer la ressource à l'entité qui la possède lorsqu'elle n'est plus la plus prioritaire
- En pratique :
  - Il suffit que l'exécutif regarde la priorité d'une entité qui naît ou se réveille
  - Si commutation retardée (cf. pb ré-entrance), on parle d'inversion de priorité

II.3 Ordonnancement avec réquisition

Ordonnancement préemptif : Traitement du problème de famine

- Problème : Famine des entités de faible priorité
  - Solution : ajustement dynamique des priorités
    - Plus une entité attend longtemps, plus sa priorité augmente
    - Lorsqu'une entité obtient (enfin) la ressource, sa priorité redescend au niveau initial
  - Conséquence : provoque de nombreux changements dans les files de priorités
    - Car les processus de même niveau de priorité sont placés sur une même file (généralement FIFO)
    - ⇒ La gestion des files doit être efficace !

II.3 Ordonnancement avec réquisition

Ordonnancement préemptif : Mise en œuvre (pour la ressource CPU)

- Lors des **ticks principaux**
  - SOIT : recalcul des priorités *dynamiques* de toutes les entités prêtes, selon formule +/- compliquée, en fonction de
    - priorité initiale (statique)
    - nombre de tics d'horloges écoulés depuis dernier ajustement
    - nombre total de ticks d'horloge utilisés sur une période donnée
    - charge du système (nb de processus prêts)
  - SOIT : 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 bloque volontairement (E/S) : quantum non utilisé en entier

II.3 Ordonnancement avec réquisition

Politique d'ordonnancement mixte : Préemptif + temps partagé

- Avant tout préemptif (FIFO avec priorité)
  - Tourniquet à l'intérieur d'une FIFO
  - Passage a file FIFO de priorité inférieur quand toutes celles des niveaux supérieurs sont vides
- Remarques :
  - Si l'entité élue est la plus prioritaire, elle peut garder la ressource plus longtemps que la durée du quantum
  - Politiques des SE dits « mutiprocessus à temps partagé » (Unix, NT, …)
  - Possibilité d'avoir une valeur de quantum différente par niveau de priorité

Systèmes d'Exploitation : principes et algorithmes

II. Ordonnancement

1. Introduction
2. Ordonnancement sans réquisition
3. Ordonnancement avec réquisition
4. **Ordonnancement dans les systèmes multi-thread**

O. Dalle

29

II.4 Ordonnancement dans les systèmes *multi-threads*

Justification du multi-thread

- Thread = activité = fil/flot/chemin d'exécution
- Multi-thread : avoir plusieurs fils d'exécution dans un même programme
  - Programmation plus complexe
    - Activités concurrentes, pouvant se dérouler dans un ordre quelconque
  - Exemples :
    - serveurs (httpd, nfsd, …) : un thread pour chaque client qui se connecte
    - explorateur de fichiers sous windows

O. Dalle

30

Exemple : Un serveur programmé sans threads

- Algorithme :
  - Attendre une requête
  - Analyser la requête
  - Servir la requête
  - Renvoyer réponse au client
- Problème :
  - si « servir la requête » se bloque, tout le serveur est bloqué

Exemple : Un serveur programmé avec des threads

- Algorithme
  - Thread principale
    - Attendre une requête
    - Analyser la requête
    - Allouer une thread fille au service
  - Une thread fille (requête R1)
    - Servir requête R1 (peut bloquer ici)
    - Renvoyer réponse au client
  - Une thread fille (requête R2)
    - Servir requête R2 (peut bloquer ici)
    - Renvoyer réponse au client

II.4 Ordonnancement dans les systèmes *multi-threads*

Relation thread/processus

- Un processus héberge au moins une thread (appelée *main*)
  - Dans un programme C, c'est elle qui est chargée d'exécuter la fonction main() du programme
- Une thread existante peut créer de nouvelles threads …
- Les threads d'un processus :
  - Ont accès à toutes les zones mémoire de l'espace d'adressage du processus (en particulier variables globales, statiques et constantes)
  - Ont leur propre pile d'exécution (donc en particulier leur propres variables locales lors d'appels de fonctions)

II.4 Ordonnancement dans les systèmes *multi-threads*

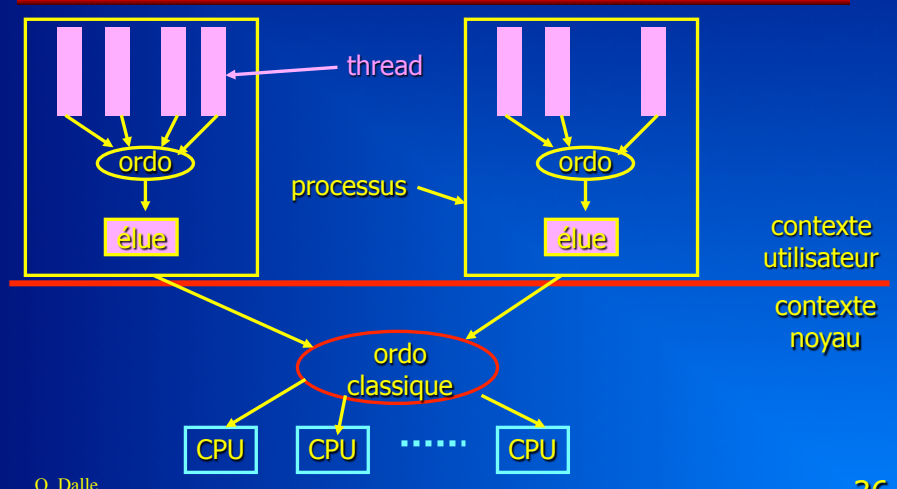
Différentes mises en œuvre du multi-thread

- Différents types de threads :
  - Inconnues du Système d'Exploitation
    - Le processus qui héberge les thread se charge totalement de leur gestion (création, ordonnancement, synchronisation, …)
  - Totalement prise en charge par le S.E.
    - L'exécutif se charge de la création, de la gestion et de l'ordonnancement des threads
  - Prise en charge mixte
    - Threads hébergées par des threads de l'exécutif système
    - Gestion/ordonnancement possible au niveau du processus

Threads de niveau « utilisateur »

- Toute la gestion des threads est assurée par le processus :
  - Le noyau ne voit qu'un unique processus
  - Toute la gestion est programmée dans l'espace d'adressage utilisateur
    - Pas besoin que le CPU soit en mode privilégié
    - Changement de contexte = changer pile d'exécution + changer pointeur instruction

Threads de niveau « utilisateur » (2)



Threads de niveau utilisateur (3)

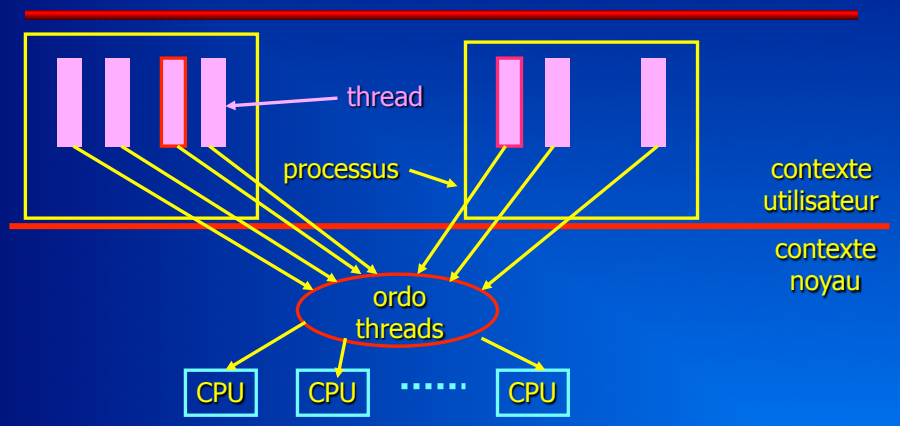
- Lorsque le processus est élu (par SE), c'est la thread élue (par processus) qui s'exécute
- Création/gestion des threads à l'aide d'une librairie (exemple POSIX 1.c *pthreads*), ou intégrées au langage (java)
- L'ordonnancement est souvent simple
  - FIFO + priorité
  - sans réquisition ou bien préemption aux points d'appels de la librairie des threads
  - pas de modification dynamique de priorité

Threads de niveau « noyau »

- La gestion et le support des threads sont programmés dans le noyau
  - exécution des tâches de gestion en contexte noyau
  - nécessite un changement de contexte
    - Appels systèmes spécifiques

II.4 Ordonnancement dans les systèmes *multi-threads*

Threads de niveau « noyau »



II.4 Ordonnancement dans les systèmes *multi-threads*

Threads de niveau « noyau »

- Le processus en tant que tel n'est pas une entité d'exécution (il fournit juste une espace d'adressage)
  - Le système affecte une *thread noyau* à chaque processus
- Ordonnancement « traditionnel » :
  - préemptif à temps partagé, avec modification dynamique des priorités
- Ce type de thread est souvent appelée « processus léger » (LWP : Light Weight Process)

II.4 Ordonnancement dans les systèmes *multi-threads*

Threads de niveau « noyau »

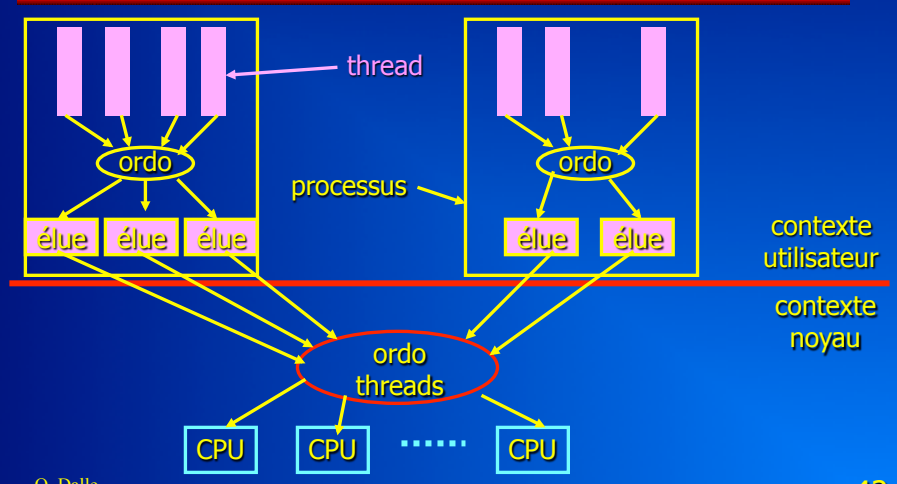
- Avantages
  - Une thread se bloque : les autres threads du processus peuvent être élues
  - machine multi-processeur : plusieurs threads peuvent s 'exécuter simultanément (vrai paraléllisme)
- Inconvénients
  - Changements de contexte/appels système coûteux (durée de réalisation plus longue)

O. Dalle

41

II.4 Ordonnancement dans les systèmes *multi-threads*

Threads mixtes « utilisateur » + « noyau »



O. Dalle

42

Threads mixtes

- Combine avantages des 2 précédentes
  - Si une thread se bloque, les autres threads du processus peuvent être élues
  - Certaines fonctions de gestion des threads peuvent être gérées au niveau utilisateur
    - Evite le coût d'un appel système/changement de contexte
- Exemple
  - Threads du langage Java sous Solaris
  - Threads POSIX sous Solaris