Nom : KANY DAPHNÉ Sujet : 17 Jury : Page : 1 / 3

Leçon 17 : Algorithmes d'ordonnancement de tâches et de gestion de ressources

Niveau : Tle / MPII / MPI
Prérequis : Processus ; Threads ; graphes
Source :

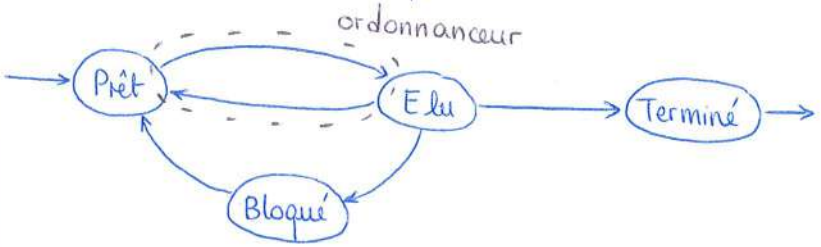
Métaphore filée : Ordonnancement des tâches dans une cuisine.

I - Motivation : le Système d'exploitation

Remarque 1 : Lorsque vous utilisez votre PC, vous exécutez des dizaines de programmes "en même temps" (onglets de navigation, écoute de musique, taper au clavier...) Pourtant votre PC n'a qu'un petit nombre de processeurs.

Définition 2 (Execution concurrente et ordonnanceur)
Le système d'exploitation peut interrompre un processus en cours d'exécution, puis décider à quel processus il rend la main. Le rôle de l'ordonnanceur est de choisir le prochain processus à exécuter parmi une liste de processus candidats.

Schéma 3 : Etats d'un processus (Cycle de vie)



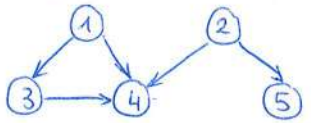
II - Problèmes d'ordonnancement théoriques

II.1 - Définition

Déf 4 (Problème d'ordonnancement)
Une instance d'un pb d'ordonnancement est un graphe orienté $G=(S,A)$, une fonction de poids $\omega : S \to \mathbb{N}^*$ et un nombre $p \in \mathbb{N} \cup \{\infty\}$
Une solution est un ordonnancement $\sigma : S \to \mathbb{N}$ tq $\forall (u,v) \in A$ $\sigma(u) + \omega(u) \leq \sigma(v)$, et une allocation $alloc : S \to [\![1 ; p]\!]$ tq $alloc(s1) = alloc(s2) \Rightarrow [\sigma(s1), \sigma(s1)+\omega 1] \cap [\sigma(s2), \sigma(s2)+\omega(s2)] = \emptyset$

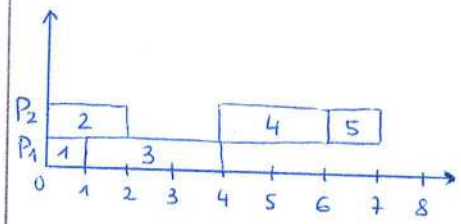
On cherche alors à minimiser $\max_{i \in S} (\sigma(i) + \omega(i))$. On note ce pb $Ordo(p)$.

Exemple 5 :



$p = 2$
$\omega(1) = 1$; $\omega(2) = 2$
$\omega(3) = 3$; $\omega(4) = 2$; $\omega(5) = 1$

Un ordonnancement valide est



Cet ordonnancement est-il optimal ?

Analogie 6 : S sont les tâches, A les dépendances, ω le temps de chaque tâche, p le nombre de processeurs, σ les dates de début de chaque tâche et alloc le processeur sur lequel la tâche s'exécute.

Ne rien écrire hors du cadre.

Remarque 7 : Dans cette définition, les tâches ne sont pas interruptibles, et il n'y a pas de coût de communication entre processeurs.

Théorème 8 : Soit I une instance de Ordo(p). I admet un ordonnancement ssi G est acyclique.

Quand G admet un cycle, on dit qu'il y a interblocage

II.2 - Dans le cas général

Def 9 (tri topologique) : Un tri topologique sur un graphe orienté $G = (S, A)$ acyclique est un ordre total $\leqslant$ sur les sommets tq $(u,v) \in A \Rightarrow u \leqslant v$

Théorème 10 : Il existe un algorithme linéaire pour déterminer un tri topologique.

Pour trouver un ordonnancement, il suffit de faire un tri topologique des tâches et de les faire dans cet ordre. On n'obtiendra pas toujours l'opt (voir exemple 5)

Théorème 11 : Si $p = 1$ ou $p = \infty$, on peut trouver une solution optimale de Ordo(p) en temps linéaire.

II.3 - Dans le cas de tâches indépendantes

Def 12 : Dans le cas où $A = \emptyset$, le problème Ordo(p) est entièrement défini par $\omega : S \to \mathbb{N}^*$. On note ce problème Indep(p).

Théorème 13 : Indep(p) pour $p \in [\![2, +\infty[\![$ est NP-dur.

Développement 1 : 3/2 - approximation gloutonne pour Indep(2).

II.4 - Dans un ordinateur

Dans un ordinateur, les tâches en cours arrivent au fur et à mesure et sont toutes indépendantes. Le système peut de plus les interrompre puis reprendre leur exécution au bon endroit. Il s'agit d'une stratégie préemptive

Algo 14 (Tourniquet) :

L'ordonnanceur utilise souvent l'algorithme du tourniquet : Chaque processus prêt est dans une file (FIFO)

Tant qu'il y a des tâches dans la file :
- On prend la tâche en tête.
- On l'exécute pendant un temps prédéfini
- Si la tâche n'est pas terminée, on l'interromp et on l'insert en queue de file

Remarque 15 : Les nouveaux processus sont ajoutés en queue de file.

Propriété 16 : L'algorithme du tourniquet garantit l'absence de famine.

III - Accès à une ressource partagée

III.1 - Définitions

Def 17 (Concurrence) : Si deux threads ou procesus executent une zone du code d'écriture ou de lecture sur une mémoire partagée et dont le résultat dépend de l'ordre d'éxecution des instructions, on dit qu'il y a concurrence. Une telle zone de code est une section critique.

TP 18 : Ecrire un programme où plusieurs threads se comptent en incrémentant un compteur partagé. Identifier les sections critiques.

III.2 - Les mutex (verrous)

Def 19 (Mutex) : Un mutex est une structure de donnée permettant deux opérations :

- lock() : Appel bloquant, demande l'accès au verrou
- unlock() : Appel qui libère le verrou

Une implémentation efficace des mutex doit avoir les propriétés suivantes :

- Exclusion mutuelle : Un thread à la fois a accès au verrou
- Absence de famine : Si un thread demande le verrou, il finira par l'obtenir
- Absence d'interblocage : Si 2 threads veulent le verrou, l'un l'obtient.

Remarque 20 : Une mauvaise utilisation des mutex peut créer des situations d'interblocage (deadlock)

Exemple 21 :

thread 1 :	thread 2 :
Mutex.lock m1	Mutex.lock m2
wait()	wait()
Mutex.lock m2	Mutex.lock m1
Mutex.unlock m2	Mutex.unlock m1
Mutex.unlock m1	Mutex.unlock m2

Algorithme 22 : Algorithme de Peterson

```
veut_entrer = [ false, false ]
tour = -1

lock (thread_id):
    veut_entrer [thread_id] = true
    other = 1 - thread_id
    tour = other
    while (veut_entrer [other] && tour == other) { }

unlock (thread_id):
    veut_entrer [thread_id] = false
```

Developpement 2 :
Présentation de l'algorithme de Peterson et preuve de fonctionnement.

Ne rien écrire hors du cadre.