

<p style="text-align: center;">info524 : Systèmes d'exploitation TD 2 : ordonnancement et processus, suite</p>
--

Pierre Hyvernats et Florian Hatat
Pierre.Hyvernats@univ-savoie.fr
Florian.Hatat@univ-savoie.fr

Exercice 1 : Processus et threads

Question 1. On essaie d'ordonner les processus suivants sur un système temps réel :

- 2 communications sonores, qui utilisent chacune 1 ms de processeur toutes les 5 ms,
- un flux vidéo à 25 trames par seconde, chaque trame ayant besoin de 20ms de processeur.

Peut-on ordonner ce système ?

Question 2. On considère deux processus qui nécessitent chacun 10 minutes de temps processeur pour faire leurs calculs. Chacun des processus est bloqué en moyenne pendant 50% du temps sur des entrées sorties.

- combien de temps dure l'exécution des deux processus, s'ils sont exécutés séquentiellement ?
- combien de temps dure l'exécution si les processus sont entrelacés ?

Donnez dans chaque cas le taux d'utilisation du processeur, si on ne compte que ces deux processus.

Question 3. Pour des threads utilisateurs, est-ce que chaque thread possède sa propre pile d'exécution ? Qu'en est-il pour les threads noyaux ?

Question 4. À votre avis, quand un processus multi-thread fait un `fork`, est-ce que tous les threads sont dupliqués pour le fils ?

Question 5. L'interface POSIX pour les processus légers (threads) contient l'instruction `sched_yield` qui permet à un thread d'abandonner le processeur au profit des autres threads. Un processus normal n'a aucun intérêt à faire ceci. Pourquoi est-ce que cette instruction existe pour les processus légers ? Donnez des exemples d'utilisation possible.

Exercice 2 : Un ordonnanceur multifeiles avec feedback

Question 1. En supposant qu'il y a 4 niveaux de priorités : de 0 (priorité élevée) à 3 (priorité basse), écrivez une petite fonction en C pour faire un ordonnanceur :

- on a une file par priorité,
- les nouveaux processus sont placés dans la file 0,
- à la fin du quantum de temps, si le processus en exécution était dans la file n :
 - . s'il était en exécution, il est remplacé en fin de la file de priorité inférieure ($n + 1$)
 - . s'il était bloqué pour des entrées sorties, il est remplacé en fin de la file de priorité supérieure ($n - 1$)

Si le processus ne peut pas changer de file, on le replace en fin de la même file.

Vous pouvez utiliser un type de données `file` avec les opérations

- `void enfile(file f, donnee d)` qui rajoute la donnée `d` dans la file `f`
- `donnee defile(f)` qui enlève et renvoie la première donnée de la file `f`
- `file nouvelle_file(void), int file_vide(file f), donnee premier(file), ...`

Remarques :

- on suppose que les nouveaux processus sont automatiquement mis dans la file 0,
- votre fonction `ordonnanceur` ne fait que gérer les files de processus ; ce n'est pas elle qui lance ou arrête l'exécution des processus.

Que pensez-vous de cet algorithme ?

Les 4 priorités sont des priorités *internes* (tous les processus peuvent passer par toutes les priorités). Comment peut-on rajouter des priorités *externes*, pour favoriser par exemple les processus `root` et défavoriser les processus lancés avec `nice` ?