

MASTER MIAGE / Système

Enseignant : Hendry FERREIRA CHAME

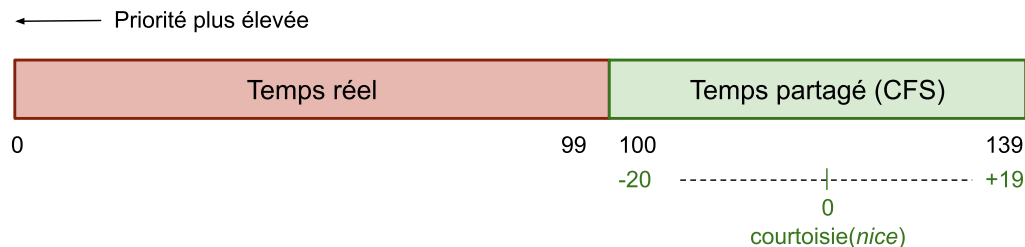
TD5 : APPLICATIONS TEMPS PARTAGÉ ET TEMPS RÉEL (LINUX, C)

INTRODUCTION

Dans un SE Unix moderne, deux grandes classes d'algorithmes d'ordonnancement sont disponibles, en fonction du comportement souhaité du système :

- L'ordonnancement temps partagé (TP) : présent par défaut sous Linux (en tant que descendant d'Unix), c'est un mode d'ordonnancement essayant de répartir le plus équitablement possible le temps d'UC disponible (algorithme par défaut CFS).
- L'ordonnancement temps réel (TR) : assure qu'une tâche d'une priorité donnée ne pourra jamais être préemptée ou laissée en attente tandis qu'une tâche de priorité moindre dispose du processeur.

La figure en bas montre l'échelle de priorité disponible pour les ordonnanceurs TP et TR.



Ce TD vous propose d'étudier ces deux types d'ordonnancement disponibles dans le noyau Linux. Les objectifs de cette formation sont :

- Comprendre la différence entre les modes d'ordonnancement temps partagé et temps réel disponibles dans le noyau Linux.
- Comprendre le système de priorité d'exécution de processus en Linux.

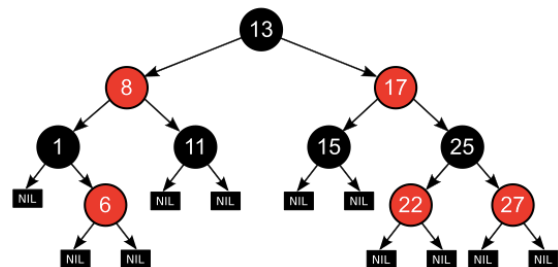
Le travail est organisé en deux parties. La Partie I vous propose d'étudier l'algorithme d'ordonnancement par défaut en linux (CFS) permettant la mise en place des applications à temps partagé. Dans la deuxième partie, vous allez étudier la mise en place des applications temps réel via l'ordonnanceur FIFO avec des priorités.

PARTIE I : ETUDE DE L'ORDONNANCEMENT TEMPS PARTAGÉ

L'ordonnanceur complètement équitable(CFS)

Un point de départ pour votre étude est l'article d'IBM¹ (en anglais) qui explique le contexte dont l'ordonnanceur *completely fair scheduler* (CFS) a été proposé par Ingo Molnar. Cet ordonnanceur fait son apparition avec la version 2.6.23, sortie le 9 octobre 2007. Le principal objectif de l'ordonnanceur CFS est de proportionner un temps d'accès équitable à l'UC. Ainsi, CFS inclut le principe d'équité vis-à-vis des processus dormants (*sleeper fairness*), afin de garantir que les tâches qui ne sont pas exécutables (p. ex. en attente d'E/S) reçoivent une part comparable du processeur lorsqu'elles en ont éventuellement besoin.

Côté réalisation, CFS n'est pas basé sur des files de processus, mais utilise un arbre bicolore (*red-black tree*) pour le triage des processus selon le temps virtuel d'exécution sur l'UC, de sorte que la feuille plus à gauche correspond au processus qui a eu moins de temps d'UC (p. ex. le noeud ayant la valeur de 1 dans la figure à droite²). Un arbre bicolore possède quelques propriétés intéressantes comme : le fait d'être auto-équilibré et d'assurer l'accomplissement des opérations d'insertion et suppression en complexité $O(\log n)$.



Exercices

Télécharger le dossier contenant des fichiers sources en langage C. Le fichier *tp_taux_uc.c* contient un programme très simple qui boucle autour de l'appel système *time()* pour lire l'heure en permanence et compte le nombre d'itération qu'il réalise pendant environ 10 secondes. Une fois le temps expiré, le programme affiche le nombre d'itérations accomplies.

- 1) Compiler le programme à l'aide du fichier *Makefile*. Ouvrir un interpréteur de commande via le raccourci Ctrl+Alt+T. Dans le dossier de téléchargement, exécutez la commande suivante pour compiler le programme :

```
make
```

- 2) Exécuter le programme *tp_taux_uc* cinq fois

```
./tp_taux_uc
```

¹ Cet article pourrait être un point de départ pour votre recherche : <https://developer.ibm.com/tutorials/l-completely-fair-scheduler/>

² Source : https://fr.wikipedia.org/wiki/Arbre_bicolore

Quelle est la valeur la plus élevée de la variable *nb_loops* affichée dans la sortie de l'interpréteur de commandes ? Prenez cette valeur comme référence du nombre d'itérations *N* que le programme peut faire en 10 secondes sur votre système.

- 3) Afin d'étudier la répartition du temps d'utilisation de l'UC, résultante de la politique d'ordonnancement CFS, exécutez en pseudo-parallélisme deux instances du programme *tp_taux_uc*. Cette étude suppose que votre machine est mono-processeur. Sinon, l'affinité du shell et celle de ses descendants doit être attribuée à une UC spécifique, en exécutant :

```
taskset -pc 0 $$
```

Suite à cette commande, les programmes que vous exécutez depuis cette fenêtre du terminal le feront toujours sur l'UC numéro 0.

Ensuite, exécutez deux instances du programme *tp_taux_uc* :

```
./tp_taux_uc N & ./tp_taux_uc N
```

Où *N* correspond au nombre d'itérations obtenues dans la question 2. Notez que le caractère « & » sert à exécuter une commande en arrière-plan, le shell reprend donc la main immédiatement pour lancer l'autre commande. Comparez pour chaque processus le pourcentage d'utilisation de l'UC relatif à la valeur référence *N*. Voici un exemple de la commande (dont *N* = 3391779189 iterations):

```
./tp_taux_uc 3391779189 & ./tp_taux_uc 3391779189
```

- 4) Sur Linux il est possible de fixer dès son démarrage la priorité d'une nouvelle tâche. Pour gérer une priorité différente de la priorité de base. Pour une application temps partagé la fourchette correspond à 0 (priorité maximale pour un utilisateur simple) à +19 (priorité minimale), il suffit de placer sa commande en argument de la commande *nice*³ et en précisant l'option *-n*. La plage de -20 à +19 est disponible seulement en exécution administrateur.

Vous allez analyser à présent le rôle de la commande *nice*. Exécutez une à une les commandes suivantes (n'oubliez pas de substituer l'argument *N* par la réponse à la question 2) :

```
sudo nice -n 0 ./tp_taux_uc N & sudo nice -n 1 ./tp_taux_uc N
```

³ Consultez les informations de courtoisie à l'aide de la commande : *man nice*

```
sudo nice -n 0 ./tp_taux_uc N & sudo nice -n 2 ./tp_taux_uc N  
sudo nice -n 0 ./tp_taux_uc N & sudo nice -n 3 ./tp_taux_uc N  
sudo nice -n 0 ./tp_taux_uc N & sudo nice -n 4 ./tp_taux_uc N  
sudo nice -n 0 ./tp_taux_uc N & sudo nice -n 5 ./tp_taux_uc N
```

- 5) Paramétrez la commande *nice* cherchant à que l'un des processus n'arrive pas à s'exécuter sur l'UC. C'est-à-dire que sa sortie correspond à : "Nombre d'itérations exécutées : 0". Est-il possible d'obtenir un tel résultat avec l'ordonnanceur CFS ? Comment expliquerez vous l'interaction entre la fourchette de priorités définies en Linux (commande *nice*) et le principe de fonctionnement de l'algorithme CFS ?
- 6) Compilez le programme *tp_nice_threads.c*. Il s'agit d'un programme proche de *tp_taux_uc.c*, implémenté en tant que processus multi-threading. Ainsi, le programme instancie cinq threads avec des priorités d'exécution décroissantes. Voyez-vous des différences dans leur temps d'exécution ? Sont-elles proportionnelles à leurs priorités établies ?

PARTIE II : ETUDE DE L'ORDONNANCEMENT TEMPS RÉEL

Sous Linux, les priorités temps réel se trouvent dans l'intervalle [1,99]. Une tâche évoluant avec la priorité 99 est donc la plus prioritaire. Il existe deux types d'ordonnancement temps réel normalisés par la norme POSIX : a) FIFO et b) Tourniquet (round robin). Concernant ce dernier, étant donné la hiérarchie de priorités établies, même avec l'ordonnancement *round robin* une tâche ne sera jamais préemptée par une autre ayant moins de priorité.

Exercices

- 1) Vous allez étudier un processus père qui crée autant de processus fils qu'il y a de CPU disponibles sur le système. Chacun d'entre eux fixera son affinité afin d'occuper tous les processeurs. Ensuite, ils passeront en temps réel FIFO de priorité 99 (la plus élevée en temps réel), avant d'exécuter une boucle infinie. Le comportement souhaité est de planter la machine pendant quelques instants en utilisant tous les CPUs disponibles. Ainsi, le processus père programme une alarme qui arrêtera les processus fils au bout de 15 secondes pour reprendre la main sur le système.

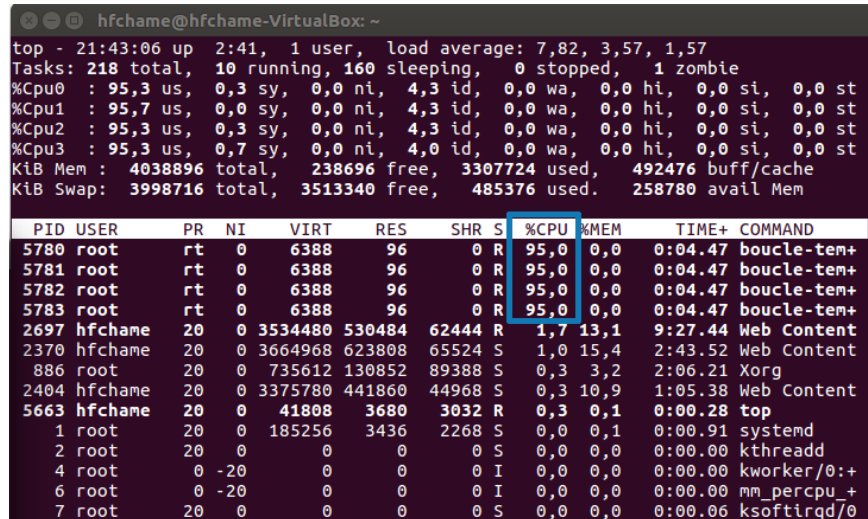
Commencez par exécuter la commande *top* :

```
top
```

Dans une autre fenêtre, compiler le programme *tr_boucle.c*. Exécuter le programme avec les droits d'administration :

```
sudo ./tr_boucle
```

Vous devriez afficher dans la fenêtre de la commande *top*, une sortie similaire à l'image suivante :



```
top - 21:43:06 up 2:41, 1 user, load average: 7,82, 3,57, 1,57
Tasks: 218 total, 10 running, 160 sleeping, 0 stopped, 1 zombie
%Cpu0 : 95,3 us, 0,3 sy, 0,0 ni, 4,3 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu1 : 95,7 us, 0,0 sy, 0,0 ni, 4,3 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu2 : 95,3 us, 0,3 sy, 0,0 ni, 4,3 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
%Cpu3 : 95,3 us, 0,7 sy, 0,0 ni, 4,0 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
KiB Mem : 4038896 total, 238696 free, 3307724 used, 492476 buff/cache
KiB Swap: 3998716 total, 3513340 free, 485376 used, 258780 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
5780	root	rt	0	6388	96	0	R	95,0	0,0	0:04.47	boucle-tem+
5781	root	rt	0	6388	96	0	R	95,0	0,0	0:04.47	boucle-tem+
5782	root	rt	0	6388	96	0	R	95,0	0,0	0:04.47	boucle-tem+
5783	root	rt	0	6388	96	0	R	95,0	0,0	0:04.47	boucle-tem+
2697	hfchame	20	0	3534480	530484	62444	R	1,7	13,1	9:27.44	Web Content
2370	hfchame	20	0	3664968	623808	65524	S	1,0	15,4	2:43.52	Web Content
886	root	20	0	735612	130852	89388	S	0,3	3,2	2:06.21	Xorg
2404	hfchame	20	0	3375780	441860	44968	S	0,3	10,9	1:05.38	Web Content
5663	hfchame	20	0	41808	3680	3032	R	0,3	0,1	0:00.28	top
1	root	20	0	185256	3436	2268	S	0,0	0,1	0:00.91	systemd
2	root	20	0	0	0	0	S	0,0	0,0	0:00.00	kthreadd
4	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	kworker/0:+
6	root	0	-20	0	0	0	I	0,0	0,0	0:00.00	mm_percpu_+
7	root	20	0	0	0	0	S	0,0	0,0	0:00.06	ksoftirqd/0

Malgré un ralentissement probable dans votre machine, celle-ci n'a pas été plantée (la souris continue à se bouger, et on a même l'occasion de faire une capture d'écran comme dans l'image précédente, où les quatre cœurs disponibles dans le système ont été chargés à 95%).

La raison de ce comportement est que le système est paramétré par défaut pour réserver 5% du temps des processeurs aux tâches temps partagés, afin de permettre l'éventuel arrêt d'un programme temps réel qui pourrait présenter un bogue. A l'aide de la commande suivante, 100% du temps de calcul des processeurs est assigné à des application temps réel :

```
sudo sysctl kernel.sched_rt_runtime_us=-1
```

Essayez à nouveau d'exécuter votre le programme :

```
sudo ./tr_boucle
```

Cette fois-ci, la machine devrait se planter pendant 15 secondes ... Afin de reprendre le comportement par défaut :

```
sudo sysctl kernel.sched_rt_runtime_us=950000
```

Notez que dans la commande la spécification de la fraction de temps UC dédiée à l'ordonnancement temps réel est exprimée en microsecondes, relative à la durée d'une seconde.

- 2) Dans cet exercice vous allez étudier l'ordonnancement temps réel FIFO multi-threads avec des priorités.

Compiler et exécuter le programme suivant :

```
taskset -pc 0 $$  
sudo ./tr_threads
```

Dans quel ordre les threads sont-ils instanciés et terminés ? Proposez une explication de ce comportement en examinant le code source du programme *tr_threads*.

- 3) L'ordonnancement par l'algorithme du tourniquet peut être problématique pour les applications en temps réel confrontées à des conditions de concurrence, car un thread peut être ôté du processeur à tout moment pendant son exécution. Une alternative consiste à utiliser l'algorithme FIFO et l'instruction *yield* où le thread appelant cède volontairement le processeur au bon moment pendant son exécution. Ainsi, le thread est déplacé à la fin de la liste des threads prêts selon sa priorité et un autre thread sera exécuté.

Compiler et exécuter le programme suivant pour comprendre l'instruction *yield* :

```
taskset -pc 0 $$  
sudo ./tr_threads_yield
```

Expliquer le fonctionnement du programme *tr_threads_yield*