# TP 2: Processus et signaux: Evaluations parallèles

Encadrant : R. Aparicio-Pardo
Author TP : Etienne Lozes

28 novembre 2017

## 1 Présentation

Le but de ce TP est de découvrir la création de processus et la communication par signaux en UNIX. Ces possibilités offertes par le système se révèlent utiles lorsqu'on cherche à implanter des algorithmes parallèles. Aujourd'hui, on se propose d'implanter plusieurs algorithmes d'évaluation parallèle d'une formule logique. Le programme `evallog0`, qui est fourni, prend en argument une formule logique, évalue sa valeur de vérité, et affiche le résultat (vrai ou faux). Ceci implique de se donner un langage pratique pour écrire des formules logiques. On utilise une notation fonctionnelle, ce qui a pour avantage d'une part de rendre inutile les parenthèses, et d'autre part facilite l'analyse descendante. Par exemple, $\top \wedge (\bot \vee \top)$ s'écrit `&1|01`, et on a :

```
> evallog  "|&1|01"
> evalue |&101 en commencant par dormir 1 secondes
> ....
> evaluation de &1|01 terminee : 1
```

Afin de manipuler des formules logiques, on utilise le module `formula.c/h`, qui est aussi fourni. Ce module contient des fonctions pour identifier les sous-expressions droite et gauche d'une formule logique. On se base sur le principe que chaque operation logique a une sous-expression droite et une sous-expression gauche, où une sous-expression consiste en soit une constante booléenne soit une expression logique correcte.

## 2 Évaluation récursive monoprocessus

La première version de notre évaluateur (fichiers C `formula.c/h` et `evallog0.c`) est basée sur une analyse descendante de la formule par un seul processus, par un appel de fonction récursif.

**Question 1.** *Analysez le code fourni pour bien comprendre et testez-le avec la chaine* `|&101`. Combien de fois appelle-t-on au processus `eval` ?

# 3 Évaluation parallèle 1

Dans les appels récursifs précédents, les deux appels sur les sous formules droite et gauche sont moralement indépendants. Ils peuvent donc être faits par deux processus distincts, ce qui fournit un schéma de parallélisation de notre évaluateur : pour évaluer une expression booléenne non triviale, le père crée deux processus et distribue à chacun une sous-expression, attend leur terminaison, et conclue.

Pour implémenter cet évaluateur, vous aurez besoin des fonctions `fork`, `wait`, `waitpid`, et `exit`. Par exemple, vous utiliserez `exit` avec le bon code de sortie (le résultat) pour fermer le processus.

**Question 2.** *Implémentez cet évaluateur parallèle avec le nom* `evallog1.c`.

# 4 Évaluation parallèle 2

Le programme precedent realise toutes les evaluations dans la chaine, même s'il y a des evaluations qui ne sont pas vraiment nécessaires, p. ex. une evaluation `&` lorsque une sous-expression est 0 : le résultat est 0, indépendamment de la valuer de l'autre sous-expression. Si on ne prend pas en compte ces évaluations, (on fait une évaluation paresseuse), on devrait gagner du temps et commencer a exploiter l'intérêt du parallélisme. Ça implique créer un processus par sous-formule à évaluer, attendre la terminaison d'un d'ux, et si ça suffit pour conclure (résultat indépendant de la valuer de l'autre expression), faire `exit` avec le bon code de sortie sans attendre le deuxième.

**Question 3.** *Implémentez un évaluateur paresseux du nom* `evallog2.c`.

# 5 Évaluation parallèle 3

Dans la version antérieure, on est plus rapide pour répondre du fait de l'aspect "concurrent" de l'évaluation, mais le calcul est toujours effectué complètement. Du temps processeur reste donc perdu par des évaluations inutiles. Pour y remédier, on se propose d'élaguer dynamiquement l'arbre des processus dans les branches d'évaluations inutiles : si le premier fils a fourni une valeur qui permet de conclure, on envoie un signal au second pour lui demander d'interrompre son évaluation.

**Question 4.** *En utilisant la fonction* `kill`, *implémentez une première solution simple et radicale du nom* `evallog3.c`.

# 6 Évaluation parallèle 4

L'approche précédente a un problème. La sous-expression redondante (inutile) aurait pu lancé lui-même toute une généalogie de processus qui ne sont

pas arrêtés automatiquement lors de sa mort. Leurs résultats ne sont pas pris en compte, la fonction principal donne le résultat final plus rapidement, mais, ils continuent à s'executer jusqu'à sa fin. Une solution pourrait être de demander au fils tué (la sous-expression redondante) de propager une signal "je suis mort, donc tu dois aussi mourir" à tous ses fils avant de mourrir.

**Question 5.** *En utilisant un signal non réservé (par exemple* `SIGUSR1`*), et en y associant un handler approprié (à l'aide de la fonction* `sigaction`), *mettez en place ce mécanisme de "ramasse-miettes" sur les processus et nommez-le* `evallog4.c`. *NOTE : Pour vous aider, vous pouvez jetter un coup d'oeil ici :* `http://www.thegeekstuff.com/2012/03/catch-signals-sample-c-code/`