# Programmation Système
# TP no 1 : Processus, tubes, signaux

Guillaume Mercier
email : mercier@enseirb.fr

12 Octobre 2018

## 1 Processus, tubes, mémoire partagée

Le but de cette partie du TP est de mettre en place un programme qui va permettre d'effectuer un calcul de façon distribuée, c'est à dire en répartissant la charge de ce calcul sur plusieurs processus. Afin de fixer les idées, nous allons supposer que le calcul en question sera la détermination du maximum d'un ensemble de valeurs entières (le maximum d'un tableau par exemple, les valeurs pouvant être déterminées aléatoirement si vous le désirez).

### 1.1 Communications par tubes

Dans un premier temps, nous allons supposer qu'un ensemble de $N$ processus vont participer à ce calcul. Au début, un unique processus va posséder le tableau sur lequel tous les autres processus vont travailler. Ce processus-maître va donc être responsable de :

— créer les $N-1$ autres processus participants ;
— initialiser le tableau d'entiers ;
— créer les tubes nécessaires pour communiquer ;
— distribuer aux processus les parties du tableau surs lesquelles ils doivent effectivement travailler (via les tubes).

Dans un schéma de type maître-esclave, les différents processus esclaves vont communiquer uniquement avec le processus-maître. Ils n'ont pas besoin de communiquer entre-eux (*a priori*. Chaque processus-esclave va calculer le maximum de sa partie de tableau et communiquer ce maximum local au processus-maître qui va ensuite procéder au calcul du maximum global en travaillant sur tous les maxima locaux qui lui auront été communiqués.

### 1.2 Communication par mémoire partagée

Dans un deuxième temps, les processus utiliseront de la mémoire partagée pour effectuer ce calcul :

— le tableau d'entiers sera stocké dans une zone de mémoire accessible par tous les processus
— chaque processus va travailler sur un partie du tableau qui lui est propre
— un variable en mémoire partagée sera également utilisée pour stocker le résultat

Afin de synchroniser les accès à cette variable, les $N$ processus utiliseront un tube (anonyme ou nommé), en mode bloquant. Au départ, ce tube contiendra *un* caractère. quand un processus voudra accéder à la variable (en lecture et/ou écriture), il devra effectuer une opération de lecture.

Après avoir accédé (et éventuellement modifié) la variable, le processus écrira *un* caractère dans le tube.

### 1.3 Évaluation de performance

Si vous avez le temps, vous pouvez comparer les performances des différents approches :
— l'approche séquentielle, où un seul processus effectue le calcul : ce sera votre base de comparaison
— l'approche maître-esclave (avec les tubes de communication) : vous pouvez faire varier le nombre de processus-esclaves (1, 2, 3, 4, etc.) et analysez ces performances au regard du nombre de processeurs disponibles sur votre machine. En particulier, que se passe-t-il lorsque le nombre de processus dépasse le nombre de processeurs ?
— l'approche mémoire partagée, là aussi en faisant varier le nombre de processus et en comparant avec les processeurs disponibles.

## 2 Signaux

### 2.1 Communications en Morse

Cet exercice a pour but de réaliser une communication utilisant le code *morse* (ou un quelconque code similaire) entre deux processus — un client et un serveur — au moyen de signaux. Un résumé du codage morse est disponible dans le fichier `morse.txt` du répertoire `/net/ens/mercier/TP/Prog_Sys`. Pour information, il faut y ajouter les particularités suivantes :
— les signes morses sont séparés par un blanc ;
— les lettres sont séparées par trois blancs ;
— les mots sont séparés par sept blancs.

#### Questions

— écrivez un programme `tstsig_1.c` qui boucle infiniment et affiche :
  — `'.'` lorsqu'il reçoit un signal `SIGUSR1` ;
  — `'-'` lorsqu'il reçoit un signal `SIGUSR2` ;
  — `' '` lorsqu'il reçoit un signal `SIGALRM`.
  Vous utiliserez la fonction `sigaction(3)` pour gérer la mise en place des gestionnaires (*handler*) de signaux. Vous pourrez utiliser la commande `kill(1)` pour envoyer des signaux au processus depuis un terminal.

— écrivez un programme `tstsig_2.c` à partir de `tstsig_1.c` qui — en plus des fonctionnalités de `tstsig_1.c` — se termine en affichant le message `"[over]"` lorsqu'il reçoit un signal `SIGTERM`.
— écrivez un programme `serveur.c` et un programme `client.c` avec les fonctionnalités suivantes :

**serveur** Le serveur itère en demandant un message (directement en morse, ou en format texte si vous avez le temps de programmer la conversion en fin de TD) et un numéro de `pid`, puis envoie le message au processus client correspondant en utilisant les 4 signaux ci-dessus avec les mêmes conventions de signification.

**client** Le client est une version étendue de `tstsig_2.c`. Il prend le `pid` du serveur en argument de ligne de commande puis reçoit et affiche un message envoyé par le serveur.

Note : il est indispensable que le message soit transmis de manière fiable.