TP à rendre 3

Dans cet exercice, il vous est demandé d'écrire un programme appelé `chronos` qui lit un fichier de configuration et lance des processus périodiquement. La particularité de ce programme est qu'il doit être entièrement piloté par des signaux.

Le fichier de configuration (que l'on appelle une *table*) contient une liste de lignes de la forme :

délai:id:commande

où délai est un nombre entier positif de secondes, id est un identificateur quelconque (lettres et chiffres uniquement, maximum 15 caractères), et commande est une suite de mots constituant une commande exécutable. La signification d'une telle ligne est : exécuter commande après délai secondes. L'identificateur id ne sert que pour les affichages.

sujet.table
```
1:job1:sleep 3
2:job2:sleep 1
1:job3:sleep 1
```

Vous trouvez un exemple de table pour `chronos` à la droite de ce paragraphe. Cette table comporte 3 *jobs*, nommés `job1`, etc. Elle prescrit le comportement suivant : si le programme est lancé à t_0, alors la commmande de `job1` doit être lancée (dans un processus fils) à $t_1 = t_0 + 1$, la commande de `job2` doit être lancée à $t_2 = t_1 + 2$ (c'est-à-dire $t_0 + 3$), et la commmande de `job3` doit être lancée à $t_3 = t_2 + 1$ (c'està-dire $t_0 + 4$). Et ensuite on recommence : la commande de `job1` doit être lancée à nouveau à $t_3 + 1$ (c'est-à-dire $t_0 + 5$), celle de `job2` à $t_3 + 3$, etc. Le délai de chaque ligne précise le temps à attendre avant de lancer la commande du job, avec retour au début de la table lorsqu'on a atteint la fin. Notez que le délai est défini à partir du lancement de la commande, et non pas à la fin de l'exécution du processus : on voit dans l'exemple que la commande de `job2` doit être lancée *avant* que la commande de `job1` soit terminée (environ un seconde avant).

./chronos sujet.table
```
[5142 @ 00:37] HELLO
[5142 @ 00:38] INIT job1 5143
[5142 @ 00:40] INIT job2 5144
[5142 @ 00:41] INIT job3 5145
[5142 @ 00:41] FINI job1 5143 ok
[5142 @ 00:41] FINI job2 5144 ok
[5142 @ 00:42] INIT job1 5146
[5142 @ 00:42] FINI job3 5145 ok
[5142 @ 00:44] INIT job2 5147
[5142 @ 00:45] INIT job3 5148
[5142 @ 00:45] FINI job1 5146 ok
[5142 @ 00:45] FINI job2 5147 ok
[5142 @ 00:46] INIT job1 5149
[5142 @ 00:46] FINI job3 5148 ok
[5142 @ 00:48] INIT job2 5150
[5142 @ 00:49] INIT job3 5155
...
[5142 @ 01:12] BYE
```

Le processus de pilotage affiche une ligne pour chaque événement important. Vous trouverez sur la droite de ce paragraphe un exemple de sortie de `chronos` exécuté avec la table donnée en exemple plus haut. Chaque ligne indique entre crochets le numéro de processus de `chronos` ainsi que l'heure courante (minute et seconde uniquement). Le numéro de processus est utile pour envoyer des signaux de contrôle, et l'heure est utile pour vérifier la cadence des jobs (même si elle peut être ponctuellement erronée à une seconde près). Chaque ligne décrit le lancement d'une commande (`INIT`) avec le numéro de processus associé, ou la fin d'un processus (`FINI`), auquel cas on indique également si le processus s'est terminé normalement (`ok` ou `ko`). En démarrant à `00:37`, `job1` est lancé à `00:38` (après 1 seconde), `job2` à `00:40`, et `job3` à `00:41` ; puis `job1` à `00:42`, etc. Les lancements sont entrelacés avec les terminaisons : la première instance de `job1` s'arrête à `00:41`, de même que celle de `job2`, etc. Il n'y a aucune raison de présumer de l'ordre des terminaisons, et avec d'autres commandes, les temps d'exécution seraient beaucoup moins prévisibles.

Quand on analyse une exécution de `chronos`, on peut faire les remarques suivantes :

- Le processus de pilotage ne fait rien la plupart du temps : il lance au maximum un processus par seconde, et souvent moins. On veut bien sûr qu'il lance les processus à la seconde prévue.
- On veut pourtant qu'il réagisse rapidement lorsqu'un job se termine : il va potentiellement lancer beaucoup de processus, et doit éviter que le noyau accumule des processus zombies.
- On veut aussi qu'il puisse, à la demande, afficher des informations sur les jobs en cours d'exécution, ou qu'il arrête/redémarre l'affichage, ou autre chose encore.
- Si on veut arrêter l'exécution du processus de pilotage, il faut au préalable que celui-ci attende la fin des jobs en cours.

Notez que les deux premières remarques sont contradictoires : on ne veut pas appeler `sleep()` pour attendre de lancer un processus (parce qu'un job peut se terminer pendant l'attente), et on ne peut pas appeler `wait()` pour attendre un fils (parce qu'on peut devoir lancer un autre job avant qu'un fils se termine). La troisième remarque implique aussi qu'il peut y avoir des choses à faire à n'importe quel moment : la réaction doit être aussi rapide que possible. La dernière remarque indique que le processus de pilotage ne peut pas s'arrêter n'importe quand : il a la responsabilité des processus qu'il a créés.

But du devoir : Vous devez écrire un programme `chronos.c`, appelé de la façon suivante :

```
./chronos fichier.table
```

Cet appel doit exécuter les jobs du fichier passé en argument en respectant les délais, détecter la fin des jobs aussi rapidement que possible, et répondre aux opérations de contrôle. Le programme `chronos` doit être entièrement piloté par des signaux. Techniquement, sa structure est la suivante :

tantque il faut continuer
(1) attendre un signal (ou plusieurs signaux)
(2) traiter les signaux reçus

Vous prêterez une attention particulière aux points suivants :
- Dans l'étape (1) le programme est effectivement en attente quand il n'a rien à faire : il ne doit pas exécuter d'itération de la boucle « pour rien » (ce qu'on appellerait une *attente active*).
- Le programme ne doit pas « perdre des signaux » : tout signal qui arrive doit être traité tôt ou tard.
- Un gestionnaire de signal ne doit effectuer *aucun* appel de primitive (ou d'une autre fonction) : tous les traitements significatifs doivent être effectués dans l'étape (2) du corps de la boucle.

Les signaux à considérer sont les suivants :

`SIGALRM` : vous utiliserez la fonction `alarm()` pour programmer le lancement de jobs ; il faut donc détecter la remise de ce signal pour savoir quand lancer un nouveau job ;

`SIGCHLD` : lorsqu'un processus se termine, le noyau du système d'exploitation envoie le signal `SIGCHLD` au père de ce processus ; ce signal est normalement ignoré, mais il est tout à fait possible de le gérer ;

`SIGTERM` : il y a une seule façon d'arrêter `chronos` : lui envoyer le signal `SIGTERM`, par exemple depuis un terminal par la commande `kill` ;

`SIGUSR1` : envoyer ce signal au processus de pilotage permet d'activer ou désactiver l'affichage ; au départ l'affichage est activé, le premier `SIGUSR1` le désactive, le suivant le réactive, etc. ;

`SIGUSR2` : sur réception de ce signal, le processus de pilotage affiche la liste de tous les jobs en cours (le code pour l'affichage d'une table vous est fourni) ;

Tous les autres signaux gardent leur disposition initiale (même s'ils sont fatals).

Programmation : Vous écrirez le programme `chronos` en langage C en n'utilisant que des primitives systèmes, et en utilisant exclusivement l'API Posix pour les signaux décrite dans le chapitre 5 du cours (sauf pour `alarm()` – mais dans tous les cas, `signal()` et `pause()` sont proscrits). Les appels à `sleep()` et `wait()` sont également impossibles (cette dernière doit être remplacée par `waitpid()` – lisez son manuel, en particulier l'usage de l'option `WNOHANG`). Il n'y a, hors du code qui vous est fourni, aucune allocation dynamique de mémoire à faire. Pour la mise en forme des lignes, vous pouvez utiliser les fonctions `time()` et `localtime()`. Les lignes de l'exemple de la page précédente sont produites avec :

```
time_t t = time (NULL);
struct tm * tm = localtime (&t);
fprintf (stderr, "[%d @ %02d:%02d] ...",
         getpid(), tm->tm_min, tm->tm_sec, ...);
```

mais vous pouvez utiliser tout autre moyen à votre convenance.

Vous trouverez sur Moodle une archive contenant un *Makefile*, quelques exemples de tables, et deux fichiers `table.h` et `table.c` qui contiennent du code de lecture d'une table dans un fichier : lisez attentivement le fichier `table.h` (qui vous autorise à utiliser `fopen()` et `fclose()`). Ne modifiez sous aucun prétexte les fichiers qui vous sont fournis.

Votre programme doit compiler avec `cc -Wall -Wextra -Werror -Wvla` (utilisez le *Makefile* fourni). Les programmes qui ne compilent pas avec cette commande ne seront pas examinés. Utilisez également `make memcheck` pour vérifier votre programme.

À rendre : Vous devrez rendre sur Moodle un *unique* fichier `chronos.c` (Moodle sait qui vous êtes, il est inutile d'appeler votre programme `Jean-Claude_Dusse_L2S4_Printemps_2019-2020_chronos.c`, et il est inutile aussi de rendre un fichier d'un autre nom ou une archive au format-du-jour – il y aura de sévères pénalités sinon).

Ce TP à rendre est individuel. On rappelle que la copie ou le plagiat sont sévèrement sanctionnés.