TP6.1 – Les signaux

Pour faire les exercices, vous avez besoin de connaître le langage `bash`. Vous pouvez vous référer à l'annexe shell. Vous pouvez aussi trouver une liste d'astuces ici. Tous les exercices sont obligatoires, sauf les exercices notés « défi » ou « optionnel » qui sont optionnels. En particulier, les exercices notés « hors présentiel » sont supposés fait d'une séance sur la suivante.

Objectifs :

- *Manipuler les signaux (**`trap`**, **`kill`**)*
- *Réviser les redirections*

Exercice 1 : *Ave César* (~1h30)

Le but de cet exercice est de vous faire manipuler les signaux. Pour cela, nous concevons un (tout petit) Colisée de Rome à partir de la chaîne de processus que vous avez mise en œuvre à la séance précédente (voir ici). Le Colisée est constitué de gladiateurs qui saluent César, et de César qui les trucide dans la joie.

Partie 1 : Première partie : les gladiateurs

Dans cette première partie, nous créons les gladiateurs. Un gladiateur est un processus qui affiche toutes les 5 secondes « **`X: Ave César`** », où **`X`** est le **`PID`** du gladiateur, et qui meurt en criant « **`X: Morituri te salutant`** » lorsqu'il reçoit un signal **`USR1`**.

Question 1.a. Copiez le script **`chaine.sh`** de l'exercice sur les chaînes de processus en **`gladiateur.sh`**. Vous pouvez soit partir de votre solution, soit télécharger le fichier se trouvant **`ici`**.

Dans votre script, au lieu d'attendre la mort d'un enfant, d'afficher « Processus X termine » et de retourner un code d'erreur vrai, **`gladiateur.sh`** doit maintenant exécuter une boucle infinie (**`while true; do ... done`**) qui affiche toutes les 5 secondes « **`X: Ave César`** », où X est le **`PID`** du gladiateur.

Remarque : *Dans cette question, comme dans toute cette partie, vous devez systématiquement tester votre script en ne lançant qu'un unique gladiateur (**`gladiateur.sh 1`**). Rappelez-vous que vous pouvez interrompre votre script en saisissant la combinaison de touches **`control+c`**.*

Question 1.b. Complétez votre script pour qu'il affiche « **`X: Morituri te salutant`** » lorsqu'il reçoit un signal **`USR1`**. Testez votre script en lançant un gladiateur dans un terminal (**`./gladiateur.sh 1`**), et en envoyant un signal **`USR1`** au gladiateur avec la commande **`kill`** dans un autre terminal.

Question 1.c. Complétez votre script pour qu'il termine le processus en renvoyant un code de retour vrai (0) juste après avoir affiché « **`X: Morituri te salutant`** ».

Question 1.d. Vous avez dû remarquer que lorsque vous envoyez un signal **`USR1`**, votre processus ne reçoit le signal qu'après avoir terminé son attente de 5 secondes. Ce comportement est dû au fait que la commande interne **`sleep`** masque les signaux pendant qu'elle s'exécute.

De façon à éviter cette attente, nous utilisons le fait que la commande **`wait`**, elle, peut être interrompue par un signal. Au lieu de lancer la commande **`sleep`** en avant plan, lancez-la en arrière plan, et utilisez **`wait $!`** pour attendre la fin de la commande **`sleep`** (le **`$!`** permet de n'attendre que la fin de la dernière commande, c.-à-d., **`sleep`**, et non celle de tous les enfants). Testez votre programme avec un unique gladiateur et vérifiez que le processus n'attend plus la fin de la commande **`sleep`**.

Partie 2 : Deuxième partie : le problème de la terminaison

Cette seconde partie de l'exercice a pour but de vous montrer qu'il est difficile de terminer plusieurs gladiateurs.

Question 1.e. Lancez deux gladiateurs avec la commande **`./gladiateur.sh 2`**. Envoyez un signal **`USR1`** a un des gladiateurs. Que constatez vous ?

Question 1.f. Débarrassez-vous du second gladiateur.

Question 1.g. Lancez deux gladiateurs avec la commande **`./gladiateur.sh 2`**. Saisissez la combinaison de touches **`control-c`**. Que constatez-vous ?

Question 1.h. Le cas échéant, débarrassez-vous des gladiateurs qui seraient encore en train de s'exécuter.

Partie 3 : Troisième partie : retrouver les PIDs des gladiateurs

Comme vous avez pu le constater, il est difficile de terminer plusieurs gladiateurs car lorsque vous envoyez un signal, il n'est envoyé qu'à un unique gladiateur. Dans cette partie, nous résolvons le problème avec le processus César. César, qui s'ennuie souvent au Colisée, apprécie envoyer des **`USR1`** à tous les gladiateurs pour les tuer. Pour connaître les **`PIDs`** des gladiateurs, César lit un parchemin nommé **`arene.txt`**, dans lequel se trouve un PID de gladiateur par ligne.

Question 1.i. Un gladiateur doit maintenant enregistrer son **`PID`** dans le fichier **`arene.txt`** de façon à ce que César puisse le trouver. Modifiez le script **`./gladiateur.sh`** pour qu'il ajoute son **`PID`** au fichier **`arene.txt`** après avoir vérifié que les paramètres sont corrects. Testez votre script en lançant deux fois un unique gladiateur que vous interromprez avec un signal **`INT`**, et en vérifiant que **`arene.txt`** contient bien les **`PID`** des gladiateurs :

```
$ ./gladiateur.sh 1
Processus 2460 démarre avec le processus initial 2460
  Fin de chaîne
2460: Ave César
2460: Ave César
^C$ cat arene.txt
2460
$ ./gladiateur.sh 1
Processus 2473 démarre avec le processus initial 2473
  Fin de chaîne
2473: Ave César
^C$ cat arene.txt
2460
2473
```

Question 1.j. Avant d'envoyer des signaux, écrivez un script **`cesar.sh`** qui :

- affiche un message d'erreur et renvoie faux si **`arene.txt`** n'existe pas,
- lit ligne à ligne **`arene.txt`**, et affiche chaque ligne sur la sortie standard,
- supprime le fichier **`arene.txt`** après avoir lu chaque ligne,
- renvoie une code retour vrai.

Testez votre script avec le fichier **`arene.txt`** que vous avez généré à la question précédente.

Question 1.k. Au lien d'afficher les PIDs des gladiateurs, **`cesar.sh`** doit maintenant envoyer un **`USR1`** à chaque gladiateur enregistré dans **`arene.txt`**. Modifiez le script **`cesar.sh`** en conséquence.

Question 1.l. Nous pouvons maintenant utiliser le script **`cesar.sh`** pour terminer proprement tous les gladiateurs lorsque l'utilisateur saisie **`control-c`** ou utilise la commande **`kill`**. Modifiez **`gladiateur.sh`** de façon à lancer **`cesar.sh`** à la réception des signaux **`INT`** et **`TERM`**.

Félicitation ! Vous venez d'écrire votre premier protocole de terminaison !

Techniquement, le protocole que vous venez de mettre en œuvre permet de terminer proprement un ensemble de processus qui collaborent. Les gladiateurs sont des processus qui offrent un service à l'utilisateur (par exemple, chaque gladiateur pourrait être associé à un utilisateur connecté à un serveur Web), et César est le processus permettant de terminer proprement l'application.

TP6.2 – Les tubes

Pour faire les exercices, vous avez besoin de connaître le langage `bash`. Vous pouvez vous référer à l'annexe shell. Vous pouvez aussi trouver une liste d'astuces ici. Tous les exercices sont obligatoires, sauf les exercices notés « défi » ou « optionnel » qui sont optionnels. En particulier, les exercices notés « hors présentiel » sont supposés fait d'une séance sur la suivante.

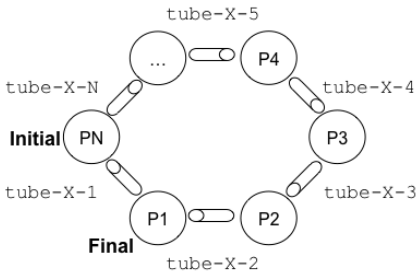
Objectifs :

- *Manipuler les tubes (**`mkfifo`**)*
- *Manipuler les signaux (**`kill`**, **`trap`**)*
- *Réviser les redirections*

Exercice 1 : La boîte à Meuh (~2h)

Le but de cet exercice est de transformer le Colisée que vous avez créé précédemment (voir ici) en boîte à Meuh, c'est-à-dire que les processus vont afficher **`Meuh`** les uns après les autres.

Même si afficher **`Meuh`** sur le terminal peut sembler quelque peu inutile, l'algorithme que vous allez mettre en œuvre est souvent utilisé dans les systèmes répartis et réseaux pour synchroniser un ensemble de processus. Par exemple, l'algorithme qu'on vous demande de mettre en œuvre est utilisé pour construire le protocole **`Token-Ring`** qui permet un accès équitable à une infrastructure réseau.



Techniquement, les processus sont organisés en anneau, comme présenté dans la figure ci-dessus. Chaque processus de la chaîne crée un tube nommé **`tube-X-N`**, où **`X`** est le **`PID`** du processus initial (stocké dans la variable **`pidInitial`**) et **`N`** l'argument donné au script, c.-à-d., le nombre de processus restant à créer dans la chaîne. Pendant l'exécution, les processus communiquent soit avec leur prédécesseur en accédant au tube du prédécesseur, soit avec leur successeur, en accédant à leur propre tube.

À partir de cette topologie, les processus échangent un jeton, symbolisé par le mot **`Meuh`**. Un processus reçoit le jeton lorsqu'il reçoit le mot **`Meuh`** via le tube de son prédécesseur. Le processus conserve ensuite le jeton pendant une seconde avant de le passer à son successeur en écrivant **`Meuh`** dans son tube.

Question 1.a. Pour commencer, il faut que chaque processus trouve (i) le nom du tube pour communiquer avec son prédécesseur et (ii) le nom du tube pour communiquer avec son successeur.

- Le nom du tube successeur est toujours **`tube-$pidInitial-N`**, où **`pidInitial`** est le **`PID`** du processus initial et **`N`** l'argument du script (i.e., **`$1`**).
- Pour le nom du tube prédécesseur :
  - si le processus est le processus initial (c.-à-d., le seul processus démarrant avec une variable **`pidInitial`** non initialisée), ce nom est **`tube-$$-1`**, où **`$$`** est le **`PID`** du processus courant, puisque le processus courant est le processus initial,
  - sinon, le nom du tube est **`tube-$pidInitial-M`**, où **`M`** vaut **`$1+1`**,

Après avoir copié **`gladiateur.sh`** en **`meuh.sh`**, modifiez votre script pour :

- stocker dans les variables **`pred`** et **`succ`** les noms des tubes du prédécesseur et du successeur,
- modifier l'affichage **`Processus $$ démarre avec le processus initial $pidInitial`** de façon à afficher les noms de ces tubes.

Remarque : *On ne vous demande pas de créer les tubes à cette question.*

Question 1.b. Modifiez votre script de façon à ce que chaque processus de la chaîne crée son tube successeur après avoir identifié les noms des tubes successeurs et prédécesseurs.

Question 1.c. Avant d'aller plus loin, il faut être capable de supprimer les tubes créés par notre application. Modifiez **`cesar.sh`** de façon à supprimer chacun des tubes créés. Vous pouvez remarquer que, comme **`meuh.sh`** exporte **`pidInitial`**, cette variable est positionnée dans **`cesar.sh`**. Pour cette raison, il suffit de supprimer tous les fichiers dont le nom commence par **`tube-$pidInitial-`** dans **`cesar.sh`**. Pensez à utiliser l'option **`-f`** de la commande **`rm`**, qui évite de demander une confirmation à l'utilisateur.

Question 1.d. Pour accéder aux tubes, nous utilisons des redirections avancées. En effet, des redirections simples ouvrent et ferment continuellement les tubes, ce qui entraîne des erreurs ou des blocages lorsque les tubes sont fermés pendant qu'il existe des interlocuteurs. Avant la boucle principale qui affiche **`Ave César`**, ouvrez en lecture/écriture les tubes prédécesseurs et successeurs (2 ouvertures différentes). Vous devriez remarquer que vous avez un message d'erreur du type : « mkfifo: tube-3665-1: File exists ». Nous nous occuperons de ce message à la question suivante.

Question 1.e. Le message d'erreur que vous avez est dû à un problème de synchronisation. Le processus initial arrive à atteindre l'ouverture du tube du prédécesseur avant que le processus final ait eu le temps créer son tube. Comme l'ouverture par le processus initial crée un fichier normal, la création du tube dans le processus final échoue avec un message d'erreur. Pour éviter ce problème, il ne faut donc ouvrir le tube du prédécesseur que si le tube existe. Avant d'ouvrir le tube du prédécesseur, ajoutez donc une boucle qui attend tant que le tube n'existe pas (donné par **`[ ! -e $pred ]`**) et qui dort une seconde à chaque pas de boucle. Vérifiez que le message d'erreur a bien disparu.

Question 1.f. De façon à mieux voir comment fonctionne le protocole, nous lançons chaque processus dans un terminal différent. Remplacez la création récursive d'un processus avec **`meuh.sh K`**, où **`K`** est égal au nombre de processus restant à créer dans la chaîne, par **`xterm -reverse -e meuh.sh K`** (l'option **`reverse`** crée un terminal blanc sur fond noir et est optionnelle, alors que l'option **`-e`** indique à **`xterm`** le nom du programme à exécuter dans le terminal). Vous pouvez admirer votre protocole de terminaison qui permet de fermer tous les terminaux en saisissant **`control-c`** dans n'importe quel terminal.

Question 1.g. Nous mettons maintenant en place notre boîte à Meuh. Dans la boucle principale, remplacez l'affichage de **`Ave César`** et l'attente de cinq secondes par :

- une lecture d'une ligne à partir du tube prédécesseur,
- l'affichage de la ligne lue,
- une attente d'une seconde,
- l'écriture de la ligne lue dans le tube successeur.

Pour amorcer votre boîte à Meuh, écrivez « **`Meuh`** » dans un des tubes à partir d'un autre terminal.

Question 1.h. Amorcer la boîte à Meuh à partir d'un autre terminal est relativement fastidieux. Pour cette raison, c'est le processus final qui va générer le premier jeton. Modifiez votre script pour que le processus final (celui qui a pour paramètre 1) écrive **`Meuh`** dans le tube de son successeur juste avant d'exécuter la boucle principale du programme.

Félicitation, vous venez d'écrire votre premier protocole multi-processus complexe !