TP à rendre 3

Description

Il vous est demandé de programmer un système composé de deux processus, le *commandant* et le *lieutenant*, qui est processus fils du commandant. Le commandant parcourt la liste des entrées du répertoire courant (sauf `.` et `..`). Pour chaque entrée il transmet le nom de l'entrée au lieutenant pour qu'il le traite, fait une sieste, et attend une réponse du lieutenant avant de transmettre le nom de l'entrée suivante. Lorsqu'il n'y a plus d'entrée à traiter, le commandant prévient le lieutenant de la terminaison du système.

Le processus lieutenant attend d'avoir un nom à traiter. Il n'est capable de traiter qu'un seul nom à la fois, mais il peut en traiter plusieurs l'un après l'autre. Le traitement d'un nom est fait par une fonction qui vous est fournie. Ce traitement peut réussir ou échouer, et le lieutenant informe le commandant du résultat du traitement. Il attend ensuite soit de recevoir un nouveau nom, soit d'être informé de la terminaison du système.

Le programme `delegation` doit être appelé avec un seul argument :

```
./delegation config
```

L'argument doit être transmis à la fonction de configuration qui vous est fournie : il détermine le temps pris par les différentes opérations (la sieste et le traitement d'un nom), ainsi que le délai dans lequel un lieutenant peut subir une erreur fatale.

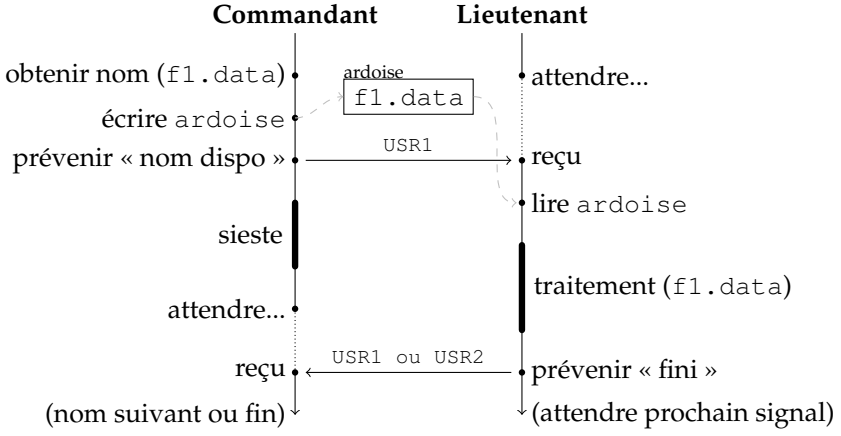
Fonctionnement

Le processus exécutant le programme `delegation` est le processus commandant ; il doit créer un processus fils qui devient le lieutenant. Les deux processus communiquent par l'intermédiaire d'un fichier appelé `ardoise`, qui ne doit contenir qu'un seul nom de fichier : le commandant y écrit un nom de fichier, que le lieutenant lit. Lorsque le commandant n'a plus de fichiers à examiner, il efface le fichier `ardoise`, prévient le lieutenant, et attend que celui-ci se termine, puis se termine lui-même (voir section suivante).

Les deux processus doivent se synchroniser pour le traitement d'un nom, selon les règles suivantes :

- le lieutenant ne peut ouvrir et lire `ardoise` que lorsque le commandant a fini de l'écrire et l'a fermé ;
- le commandant ne peut ouvrir et écrire `ardoise` que lorsque le lieutenant a fini de le lire et a fini le traitement associé ;
- le commandant ne peut demander un nouveau traitement au lieutenant que lorsque le lieutenant a fini le traitement précédent ;
- le processus commandant ne peut se terminer que lorsque le processus lieutenant est terminé.

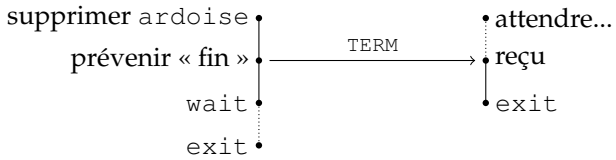
Toutes les synchronisations pour la distribution d'un travail doivent se faire par envoi du signal `SIGUSR1`. Les réponses du lieutenant doivent se faire par envoi d'un des signaux `SIGUSR1` ou `SIGUSR2` selon que le traitement a réussi ou échoué respectivement. Voici un scénario représentant un seul échange :



Le signal envoyé par le lieutenant après traitement (`SIGUSR1` ou `SIGUSR2`) dépend de la valeur de retour de la fonction appelée pour traiter un nom, et indique le succès ou l'échec de ce traitement.

Terminaison

Les échanges se poursuivent tant que le commandant obtient des noms. Lorsqu'il n'en obtient plus, il doit supprimer le fichier de transfert et prévenir le lieutenant en lui envoyant le signal `SIGTERM` :



On veut également avoir la possibilité de terminer prématurément le système, en envoyant au processus commandant le signal `SIGTERM` à partir d'un terminal (avec la *commande* `kill -TERM <pid>`). Dans ce cas, le commandant doit poursuivre normalement l'échange courant mais ne doit plus transmettre de nouveau nom au lieutenant : il doit se terminer comme s'il avait atteint la fin du répertoire.

Dans tous les cas où il se termine « normalement » (en appelant `exit()`), chaque processus affiche avant de se terminer le nombre de noms traités avec succès et le nombre de noms pour lesquels le traitement a échoué (ce qui permet de vérifier l'égalité des statistiques des deux processus). Ces lignes doivent apparaître sur la sortie standard, et être de la forme suivante :

```
COMMANDANT 123 45
LIEUTENANT 123 45
```

Votre programme ne doit pas afficher d'autre ligne sur la sortie standard, mais il peut utiliser le canal d'erreur standard sans restriction.

Gestion des erreurs et codes de retour

Le processus lieutenant peut rencontrer différents types d'erreurs :

- Si il ne parvient pas à lire le fichier `ardoise` (ou à l'ouvrir, ou à le fermer), ou si le traitement a échoué, il prévient le commandant en lui transmettant le signal `SIGUSR2` au lieu de `SIGUSR1`. Il attend ensuite le signal suivant en provenance du commandant. Si une telle erreur survient une ou plusieurs fois, le code du processus lieutenant devra être égal à 1. Si cette erreur ne survient jamais, son code de retour devra être égal à 0.
- Il est possible que le traitement d'un nom provoque un « plantage » du processus lieutenant, par appel de la fonction `abort()`. Dans ce cas, le processus commandant (informé par la réception d'un signal `SIGCHLD`), doit se terminer après avoir effacé le fichier `ardoise` et appelé `wait()`.

Le processus commandant peut rencontrer les erreurs suivantes :

- Si il rencontre une erreur en tentant d'obtenir un nouveau nom, ou en écrivant le fichier `ardoise`, il suit la procédure normale pour prévenir le lieutenant de la terminaison (voir section précédente), et se termine avec un code de retour égal à 2.
- Si le commandant constate que le processus lieutenant s'est terminé anormalement (par appel à `abort()` au cours du traitement, ou suite à la réception d'un signal autre que `SIGUSR1` ou `SIGTERM`), il efface le fichier `ardoise`, appelle `wait()` et se termine avec un code de retour égal à 3.
- Si le commandant reçoit un signal `SIGTERM`, il poursuit normalement l'échange en cours mais ne transmet pas de nouveau nom. Il suit alors la procédure normale de terminaison, et se termine lui-même avec un code de retour égal à 4.

Si le commandant constate plusieurs de ces erreurs simultanément, il doit utiliser le code de retour de la première dans la liste. Si il ne constate aucune de ces erreurs, son code de retour doit être le même que celui du processus lieutenant.

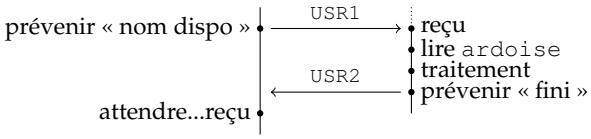
Il reste à traiter toutes les erreurs « habituelles ». Si le programme est appelé avec des arguments incorrects, son code de retour doit être 6. Sinon, le commandant doit tenter d'ouvrir le répertoire courant avant de créer le processus lieutenant ; si cette tentative échoue, il doit s'arrêter avec un code de retour égal à 7. De même, si le commandant n'arrive pas à créer le lieutenant, le code de retour doit être égal à 7.

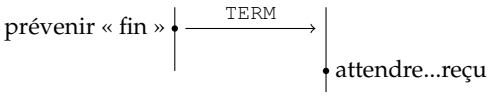
Enfin, si n'importe lequel des processus rencontre une erreur sur les appels système concernant les signaux, il s'arrête immédiatement avec un code de retour égal à 8. Toutefois, si il s'agit du processus commandant, il doit essayer avant cela d'envoyer le signal `SIGKILL` au lieutenant, mais sans tester le résultat de cette tentative.

Les autres erreurs liées aux appels de primitives système doivent, tout au plus, afficher simplement un message d'erreur.

Remarques sur les attentes

Notez que dans tous les exemples on suppose qu'un processus est en attente lorsqu'il reçoit le signal. Ce n'est pas vrai en général, parce que les processus s'exécutent indépendamment en dehors des synchronisations explicites. Un signal peut donc arriver avant que le processus soit prêt à l'attendre. Par exemple, on peut avoir les deux situations suivantes :





Dans ces cas, il n'y a pas de réelle attente dans le processus récepteur. Bien sûr, aucun signal ne doit être perdu.

Réalisation

Programmation : Vous écrirez le programme `delegation` en langage C en n'utilisant que des primitives systèmes. Vous pouvez utiliser des fonctions de bibliothèque pour la gestion des chaînes de caractères (`strcmp`, `strlen`) et pour l'affichage des messages (`fprintf`). Tous les noms de fichiers que vous manipulerez seront de longueur inférieure ou égale à `NAME_MAX`, une constante définie dans `limits.h` (ce fichier a une page de manuel). Votre programme ne devra pas comporter d'allocation dynamique de mémoire.

Pour ce qui est de la gestion des signaux :

- Vous devez utiliser l'API POSIX (les programmes utilisant l'API Unix V7 ne seront pas examinés).
- Un gestionnaire de signal ne doit appeler *aucune* fonction. Aucune.
- Votre programme ne doit pas contenir d'attente active, même ralentie, de la forme :

```
while (... /* une condition quelconque */)
    sleep(1); /* ou n'importe quel code non-bloquant */
```

  Si il doit attendre, votre programme doit le faire par appel d'une primitive système spécifique.
- Votre programme ne doit pas modifier le comportement par défaut des signaux qu'il n'utilise pas.

Vous trouverez sur Moodle une archive contenant deux fichiers contenant le code des fonctions `sieste()` et `traitement()`, ainsi qu'un fonction `configuration()`. L'usage de ces fonctions est décrit dans le fichier `traitement.h`. Le *Makefile* se charge de compiler `traitement.c`. Vous devez dans votre programme appeler la fonction `condiguration()` au début de votre programme, en lui passant l'unique argument de la ligne de commande. Vous ne devez en aucun cas modifier les fichiers `traitement.h` et `traitement.c` : le code que vous rendrez sera compilé avec la version originale de ces fichiers.

Votre programme doit compiler sur `turing.u-strasbg.fr` avec les options `-Wall -Wextra -Werror -Wvla` (utilisez le *Makefile* fourni). Les programmes qui ne compilent pas avec ces options ne seront pas examinés. Les tests seront lancés sur cette même machine.

Pour les tests manuels pendant le développement, votre programme doit être lancé *depuis* un répertoire de test (puisqu'il parcourt les fichier du répertoire courant). Le fichier *Makefile* contient un exemple d'appel dont vous pouvez vous inspirer : c'est la cible appelée `essai`.

À rendre : Vous devrez rendre sur Moodle un *unique* fichier `delegation.c`. Si votre fichier porte un autre nom ou n'est pas un programme C, il ne sera pas examiné.

Tests : Un script de test est mis à votre disposition sur Moodle. Celui-ci exécute votre programme sur des jeux de tests. Il vous affiche le résultat des 4 tests, qui servira de base à l'évaluation de votre rendu. Pour savoir comment l'utiliser, n'hésitez pas à lire ce script et le fichier de log généré : ils peuvent vous aider à mettre votre programme au point. N'hésitez pas non plus à contacter votre enseignant si vous constatez un comportement anormal ou si vous souhaitez ajouter un test.

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