TP1 : Les appels système sous Unix

Le but de ce TP est d'implémenter en C un programme qu'on nommera `parexec`. Ce programme prend en argument de ligne de commande un nom de programme `prog`, suivi d'une liste arbitrairement longue d'arguments, et il exécute `prog` en parallèle (dans des processus) sur chacun des arguments. Par exemple, taper la ligne de commande `./parexec gzip fichier1 fichier2 ... fichierN` aura pour effet de lancer en parallèle les commandes `gzip fichier1`, `gzip fichier2` ... `gzip fichierN`.

Le sujet de TP vous guide progressivement vers des versions de plus en plus sophistiquées de ce programme. En plus des consignes (spécifications) nous avons aussi inclus de nombreux pointeurs vers de la documentation. En particulier vous allez devoir utiliser :

- le polycopié d'Éric Guérin sur le langage C, qui est disponible sur le moodle IF-3-SYS.
- le manuel de la GNU C Library (glibc), qui est disponible sur le web. Plutôt que de retaper les adresses URL à la main, n'hésitez pas à ouvrir aussi la version PDF de l'énoncé (sur moodle) dans laquelle les liens sont cliquables.

Mais surtout, n'hésitez pas aussi à poser des questions à votre moteur de recherche préféré et/ou aux enseignants qui sont dans la salle.

1 Préliminaires

Exercice Téléchargez depuis moodle l'archive correspondant à ce TP, puis tapez la commande `tar -zxvf TP1.tgz`. Vous allez travailler dans le répertoire `TP1` ainsi créé. Ce répertoire contient déjà des squelettes de programmes ainsi qu'un Makefile. Lisez ce Makefile et posez des questions sur ce que vous ne comprenez pas.

Exercice Implémentez dans `rebours.c` un programme qui fait un compte à rebours seconde par seconde, à partir d'un nombre passé en argument. Sur chaque ligne, `rebours` affichera son PID et le nombre de secondes restantes, comme illustré ci-dessous. Pour vous aider, lisez aussi les explications données dans la liste à puce encore en-dessous.

```
% ./rebours 5
38997: debut
38997: 5
38997: 4
38997: 3
38997: 2
38997: 1
38997: fin
%
```

Remarques

- Dans cet exercice, vous aurez besoin de :
  - récupérer les arguments de ligne de commande ⟶ cf poly de C §7.1 p63, et documentation de la bibliothèque standard `https://www.gnu.org/software/libc/manual/html_node/Program-Arguments.html`
  - convertir en nombre une chaîne de caractères ⟶ cf poly de C §4.2 p40 et §B.3 p115. Et aussi `https://www.gnu.org/software/libc/manual/html_node/Parsing-of-Integers.html`
  - déterminer le numéro du processus courant ⟶ appel système `int getpid();` dont la doc est accessible par `man getpid` et/ou dans le manuel de la libc `https://www.gnu.org/software/libc/manual/html_node/Process-Identification.html`.
  - suspendre l'exécution pendant un certain temps ⟶ appel système `sleep()`; dont la doc est accessible par `man 3 sleep` et/ou aussi dans le manuel de la libc `https://www.gnu.org/software/libc/manual/html_node/Sleeping.html`

— N'hésitez pas à user et abuser de la fonction `assert()` un peu partout dans votre code. Par exemple pour vérifier que la durée du compte à rebours est correcte, vous pouvez écrire `assert(duree>0)`; Ajouter des assertions permet à la fois de rendre le programme plus lisible en explicitant vos hypothèses, mais également de vérifier ces hypothèses lors de l'exécution !
— Dans le listing ci-dessus, le signe % (pourcent) représente votre «invite de commande» (en VO votre *shell prompt*). Votre programme n'a donc rien à voir avec lui.
— Pourquoi faut-il taper `./rebours` et non pas juste `rebours` ? C'est pour indiquer au shell qu'il doit chercher ce programme dans le répertoire courant, qui se note « . » (point). Si on tape un nom de commande sans chemin, le shell cherche dans une liste de répertoires par défaut contenant par exemple `/bin`, `/usr/bin`, etc.

2 Manipulation des processus avec fork/exec

2.1 Lancement de programmes

Sous Linux, et plus généralement sur tous les systèmes compatibles avec le standard POSIX, le lancement d'un programme se fait en deux étapes distinctes.
Pour demander au noyau de créer un nouveau processus, on invoque l'appel système `fork()` dont l'effet est de *dupliquer* intégralement le processus courant. Au retour de la fonction, les *deux* processus (parent et enfant) exécutent chacun une copie indépendante du même programme. Dans le parent, la valeur de retour de `fork()` est le PID de l'enfant. Dans l'enfant, la valeur de retour de `fork()` est zéro, ce qui permet de différencier les deux processus. En dehors de ça, toutes les variables ont donc initialiement la même valeur, même s'il s'agit bien de varibles distinctes. Autrement dit, changer la valeur de `myvar` dans un processus n'affectera pas la variable `myvar` de l'autre processus.

Pour demander au noyau de changer le programme en cours d'exécution, on invoque l'appel système `exec` (qui est disponible au travers de plusieurs fonctions aux signatures légèrement différentes : execl, execle, execlp, execv, execvp ...) Cet appel de fonction ne retourne jamais : au contraire, le processus oublie tout ce qu'il était en train de faire, et commence à exécuter le nouveau programme depuis le début. Il ne s'agit pas d'un oubli temporaire : lorsque le nouveau programme se termine, le processus se termine aussi, et on ne revient donc jamais au programme précédent.

Cette explication est directement traduite de la documentation de la glibc :
`https://www.gnu.org/software/libc/manual/html_node/Process-Creation-Concepts.html`

Exercice Implémentez `parexec` en vous servant notamment des appels système `fork()` et `execlp()` (ou toute autre fonction de la famille `exec`). On veut que `parexec` ne rende la main au shell que lorsque toutes les exécutions de `prog` se sont terminées, comme illustré ci-dessous. Pour vous aider, lisez aussi les explications données dans la liste à puce encore en-dessous.

```
% ./parexec ./rebours 3 6
41035: debut
41035: 6
41034: debut
41034: 3
41035: 5
41034: 2
41034: 1
41035: 4
41034: fin
41035: 3
41035: 2
41035: 1
41035: fin
%
```

Remarques

— Pour créer un processus vous aurez besoin de l'appel système `fork()`. Commencez par en lire la doc en tapant `man fork` et/ou en cliquant sur lien ci-dessous :
`https://www.gnu.org/software/libc/manual/html_node/Creating-a-Process.html`

— Pour exécuter un programme vous aurez besoin d'un appel système de la famille de `exec`. Faites donc `man 3 exec` et/ou allez lire la page correspontante :
`https://www.gnu.org/software/libc/manual/html_node/Executing-a-File.html`

— Pour attendre que les processus fils se terminent, utilisez l'appel système `wait()` dont vous trouverez la doc en tapant `man 2 wait` et/ou ici : `https://www.gnu.org/software/libc/manual/html_node/Process-Completion.html#index-wait`
Notez que pour l'instant vous n'avez pas besoin des arguments et vous pouvez vous contenter d'écrire `wait(NULL);`

Faites valider votre programme par un enseignant, puis faites une copie de sauvegarde de votre code, par exemple dans un fichier `parexec1.c`, avant de passer à la suite.

2.2 Limitation du nombre de processus simultanés

Exercice Modifiez votre programme `parexec` pour qu'il prenne un argument supplémentaire `N` entre `prog` et `argument1`. Cet argument sera un entier indiquant le nombre maximum d'instances de `prog` à lancer en parallèle. Lorsque ce nombre est atteint, `parexec` doit attendre qu'un de ses fils se termine avant d'en lancer un nouveau. Ce comportement est illustré ci-dessous avec N=2, l'exécution de l'ensemble prenant environ huit secondes.

```
% ./parexec ./rebours 2 3 4 5
13094: debut
13095: debut
13094: 3
13095: 4
13095: 3
13094: 2
13095: 2
13094: 1
13095: 1
13094: fin
13096: debut
13096: 5
13095: fin
13096: 4
13096: 3
13096: 2
13096: 1
13096: fin
%
```

Lorsque vous avez correctement résolu l'exercice, faites une copie de sauvegarde de votre code, par exemple dans un fichier `parexec2.c`, avant de passer à la suite.

3 Détection des arrêts intempestifs : les signaux UNIX

Jusqu'ici, on ne s'est intéressé qu'à des scénarios où tous les processus fils se terminaient normalement, c'est à dire en invoquant l'appel système `exit()`. Rappelez-vous au passage que si vous sortez de votre fonction `main()` par un «`return TRUC`» alors votre programme fera implicitement un appel `exit(TRUC)` (cf poly de C §7.5 p65).
Mais il se peut qu'un programme se termine abruptement (en VO on parle de «*abnormal termination*») par exemple s'il fait un accès mémoire invalide, ou une division par zéro. L'utilisateur peut également interrompre l'exécution de son programme grâce à la combinaison de touches `Ctrl+C`.

Sous Linux, et en général sous POSIX, ces différentes scénarios reposent sur un même mécanisme appelé *signalisation*.

Définition : Un *signal* est une notification envoyée de façon asynchrone à un processus. Il ne s'agit pas d'un message à proprement parler, car un signal n'a pas de «contenu», seulement un numéro.

Les différents numéros disponibles sont standardisés. Par exemple quand je tape `Ctrl+C` dans le terminal mon programme reçoit le signal n° 2. Mais pour des raisons de lisibilité et de portabilité on utilise généralement des noms symboliques. Par exemple le signal associé à `Ctrl+C` s'appelle SIGINT, pour «*Terminal interrupt*». De même une division par zéro provoquera l'envoi du signal SIGFPE, pour «*Erroneous arithmetic operation*».
Sauf cas particulier, la réception d'un signal provoque la terminaison abrupte du programme. On peut d'ailleurs forcer un processus à quitter en lui envoyant manuellement un signal avec la commande `kill`.

Exercice Ouvrez deux fenêtres de terminal. Dans la première exécutez `./rebours 10` et repérez le PID du processus, par exemple 12345. Dans la seconde fenêtre tapez `kill 12345` et constatez que le compte à rebours est alors interrompu instantanément. Tapez ensuite `man 1 kill` et parcourez la page. Faites aussi un `kill -l` qui vous affichera une liste des signaux disponibles et de leurs noms symboliques.

Exercice Modifiez votre programme `parexec` pour que, si une des instances de `prog` se termine anormalement (i.e. sur un signal) alors il ne lance plus de nouvelles instances de `prog`. À la place, il attend que les processus déjà lancés se terminent puis il quitte à son tour.

Remarques
- Pour permettre au parent de connaître la cause de terminaison de chaque fils, vous devrez invoquer `int wait(int *status-ptr)` avec un argument non nul.
- Pour interpréter le *statut* ainsi obtenu, utilisez les différentes fonctions prévues à cet effet : `https://www.gnu.org/software/libc/manual/html_node/Process-Completion-Status.html`
- Pour tester cette nouvelle version de `parexec` vous pouvez avoir recours à la commande `kill`. Ou alors, vous pouvez modifier votre programme `rebours` pour que, lorsqu'on lui passe un certain argument il termine abruptement, par exemple en appelant `abort()` ou en faisant une division par zéro. Tapez donc `man abort` et remarquez au passage que vous êtes maintenant capable de comprendre ce que vous lisez.

Faites valider par un enseignant, puis faites une copie de sauvegarde de votre code, par exemple dans un fichier `parexec3.c`, avant de passer à la suite.

Exercice Modifiez votre programme `parexec` pour que, si une des instances de `prog` se termine anormalement alors il tue immédiatement toutes les autres instances puis il quitte.

Remarques
- Vous allez devoir utiliser la fonction `kill()` pour envoyer des signaux. `man 1 kill` et/ou `https://www.gnu.org/software/libc/manual/html_node/Signaling-Another-Process.html`
- Plusieurs signaux permettent de tuer un programme. Tant qu'à faire, vous utiliserez `SIGTERM` qui est «*the normal way to politely ask a program to terminate*», nous dit la documentation : `https://www.gnu.org/software/libc/manual/html_node/Termination-Signals.html`