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 arguments 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 moodle.
- 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.

Toujours sur moodle, vous trouverez aussi des exemples de programmes C illustrant l'usage des API Linux. Toutes ces ressources sont là pour vous aider, mais 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

Cette première partie ne porte pas encore sur l'implémentation de `parexec`, mais sur la prise en main des différents appels système qui vous seront utiles dans la suite.

Exercice Téléchargez depuis moodle l'archive correspondant à ce TP, puis tapez la commande `tar -zxvf SYS-TP1.tgz`. Vous allez travailler dans le répertoire `SYS-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.

1.1 Appels système sleep() et getpid()

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.

```
user@machine$ ./rebours 5
38997: debut
38997: 5
38997: 4
38997: 3
38997: 2
38997: 1
38997: fin
user@machine$
```

Remarques

- Dans cet exercice, vous aurez besoin de :
  - récupérer les arguments de ligne de commande via `argc` et `argv` $\longrightarrow$ 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 avec `atoi()` ⟶ cf poly de C §4.2 p40 et §B.3 p115, et `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, `user@machine$` représente l'«invite de commande» du shell, et non pas un affichage de notre programme.
- 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.

1.2 Création de processus : fork()

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. Dans un premier temps, il faut créer un nouveau processus et dans un second temps, ce processus doit démarrer le programme voulu.

Pour demander au noyau de créer un nouveau processus, on invoque l'appel système `fork()` dont l'effet est de *dupliquer* intégralement le processus appelant. Au retour de la fonction, les deux processus, qu'on appelle respectivement le «parent» et «l'enfant», exécuteront 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. C'est cette valeur de retour qui permet de différencier les deux processus. En dehors de ça, toutes les variables ont donc initialement la même valeur, même s'il s'agit bien de variables distinctes. Autrement dit, changer la valeur de `myvar` dans un processus n'affectera pas la variable `myvar` de l'autre processus.

Exercice Implémentez dans `fourche.c` un programme qui affiche son PID, puis se dédouble, puis affiche son PID. À l'exécution, le résultat doit ressembler par exemple au listing ci-dessous.
Faites valider votre programme par un enseignant.

```
user@machine$ ./fourche
54333: hello world
54333: je suis le parent
54334: je suis l'enfant
user@machine$
```

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`

Exercice Lisez le programme ci-dessous et répondez aux questions suivantes :

- combien de fois la lettre A sera-t-elle affichée en tout ?
- dessinez un diagramme de séquence indiquant quelles lignes sont exécutées par chaque processus.

Pour vous aider, vous pouvez retaper ce programme et rajouter des `printf()` partout, pour afficher notamment `getpid()`. N'oubliez pas de terminer chaque appel à `printf()` avec un retour à la ligne `\n`. Dans tous les cas, faites valider votre réponse par un enseignant.

```
1 main()
2 {
3   fork();
4   if ( fork() )
5   {
6      fork();
7   }
8   printf("A\n");
9 }
```

1.3 Changement de programme : exec()

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, etc. Dans ce TP, on utilisera `execl()`
Un appel à `exec` ne «retourne» jamais : au contraire, le processus oublie tout ce qu'il était en train de faire, et se met à exécuter le nouveau programme depuis le début. Attention il s'agit d'un oubli définitif : lorsque notre nouveau programme se terminera, notre processus se terminera aussi, et on ne reviendra donc jamais au programme «précédent».

Exercice Lisez le programme ci-dessous (et les remarques encore en dessous) puis répondez aux questions suivantes :

- combien de fois la lettre A sera-t-elle affichée en tout ?
- combien de processus sont impliqués dans l'exécution ?

```
int main(void)
{
    printf("A\n");
    execl("./rebours", "./rebours", "5", NULL);
    printf("A\n");

    return 0;
}
```

Remarques

- Le code ci-dessus utilise l'appel système `exec`. Faites donc `man 3 exec` et/ou allez lire la documentation : `https://www.gnu.org/software/libc/manual/html_node/Executing-a-File.html`
- Notre exemple utilise `execl()`, dont les paramètres doivent être :
  - le chemin vers le fichier contenant l'exécutable, donné en syntaxe absolue p. ex. `"/dir/prog"` ou relative p. ex. `"./prog"` ou `"../dir/prog"`
  - les arguments de ligne de commande (i.e. les chaînes `argv[...]`) y compris l'argument n° 0, qui par convention est le *nom* du programme. En pratique vous pouvez toujours répéter la même valeur pour les deux arguments *chemin* et `argv[0]`, comme dans l'exemple.
  - et un pointeur nul pour marquer la fin de la liste des arguments.

Exercice Implémentez dans `doublerebours.c` un programme qui se dédouble, puis qui exécute, dans chaque processus obtenu, le programme `rebours` avec des arguments différents, que vous fixerez arbitrairement. À l'exécution, le résultat doit ressembler à l'un ou l'autre listing ci-dessous. Essayez différents paramètres pour obtenir au choix les deux comportements.

```
user@machine$ ./doublerebours
55338: debut
55338: 2
55337: debut
55337: 4
55338: 1
55337: 3
55338: fin
55337: 2
55337: 1
55337: fin
user@machine$
```

```
user@machine$ ./doublerebours
80599: debut
80598: debut
80599: 5
80598: 1
80598: fin
80599: 4
user@machine$ 80599: 3
80599: 2
80599: 1
80599: fin
```

Remarques

— Si le processus parent se termine *avant* le processus enfant, vous obtiendrez un affichage qui ressemble au listing ci-dessus à droite : le shell affiche son prompt alors que notre commande est toujours en train de s'exécuter.

— Pour nettoyer votre terminal, vous pouvez alors envoyer au shell quelques lignes vides (en appuyant sur Entrée) pour le forcer à afficher de nouveau son prompt proprement.

— Ceci n'est pas un bug : on dit que le processus *s'exécute en arrière-plan* vis-à-vis du shell. Certaines commandes sont même conçues pour être utilisées ainsi.

2 Manipulation de processus avec fork/exec/wait

Dans cette seconde partie, vous allez implémentez le programme `parexec` décrit au début du TP, qui permet d'exécuter une même commande plusieurs fois en parallèle.
Plus précisément, `parexec` prend en arguments 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. Autrement dit, `./parexec prog arg1 arg2 ... argN` exécutera simultanément toutes les commandes `prog arg1`, `prog arg2` ... `prog argN` chacune dans un processus distinct.

Dans ce TP, `prog` sera typiquement le programme `rebours`, comme illustré dans les encadrés page suivante.

2.1 Exécution de programmes en avant-plan

Exercice Implémentez `parexec` en vous servant notamment des appels système vus jusqu'ici. Le nom du programme `prog` à lancer ainsi que chacun des arguments sont passés à `parexec` sur la ligne de commande. Attention, on veut que `parexec` ne rende la main au shell que lorsque toutes les exécutions de `prog` se seront terminées, comme illustré ci-dessous avec `rebours`. Pour vous aider, lisez également les remarques sous les encadrés.

```
user@machine$ ./parexec ./rebours 4
28393: debut
28393: 4
28393: 3
28393: 2
28393: 1
28393: fin
user@machine$
```

```
user@machine$ ./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
user@machine$
```

```
user@machine$ ./parexec ./rebours 1 2 3
57371: debut
57372: debut
57371: 2
57372: 3
57370: debut
57370: 1
57371: 1
57370: fin
57372: 2
57371: fin
57372: 1
57372: fin
user@machine$
```

Remarques

- Pour attendre que les processus enfants se terminent, vous utiliserez l'appel système `wait()` dont la documentation est disponible en tapant `man 2 wait` et/ou ici : `https://www.gnu.org/software/libc/manual/html_node/Process-Completion.html#index-wait`
- Rappel : `wait()` ne fonctionne *que* sur les enfants directs, et pas sur les descendants plus éloignés.
- Attention : Si le processus enfant est *déjà* terminé au moment où le parent appelle `wait()`, alors celui-ci n'est *pas* suspendu et l'appel système rend la main immédiatement.
- Notez que pour l'instant vous n'avez pas besoin des arguments et vous pouvez vous contenter d'écrire `wait(NULL);`
- Attention, comme nous dit la documentation, cette primitive «*is used to wait until any one child process terminates*». Vous voulez attendre la terminaison de plusieurs enfants, il vous faudra donc appeler `wait(NULL)` le bon nombre de fois.

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 enfants 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.

```
user@machine$ ./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
user@machine$
```

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 enfants 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 enfant, 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 2 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*» comme nous dit la documentation : `https://www.gnu.org/software/libc/manual/html_node/Termination-Signals.html`