Travaux Pratiques de Système n°6

Exercice 19 : Sauvegarde du PID

Certains programmes (les daemons en particulier) sauvegarde leur PID dans un fichier, ce qui permet à des commandes extérieures de leur envoyer un signal plus facilement. Ces fichiers sont généralement sauvegardés dans le répertoire `/var/run` ou `/run`.

Question 19.1 Écrire un programme `myself` qui écrit son pid dans un **fichier texte** nommé `myself.pid`. Ce fichier sera placé dans le répertoire de travail courant du processus.

Exercice 20 : Signaux

Rappel : vous devez compiler tous vos programmes avec les options `-std=c99` et `-Wall`. Dans les exercices de programmation système, vous devrez parfois ajouter l'option `-D_DEFAULT_SOURCE`[4] "to get definitions that would normally be provided by default".
Pour avoir plus d'informations à ce sujet : RTFM `feature_test_macros(7)`.

Remarque : pour installer un handler de signal, vous n'avez pas le droit d'utiliser l'appel système `signal(2)`.
Extrait de la man page de `signal(2)` : "The behavior of signal() varies across UNIX versions, and has also varied historically across different versions of Linux. Avoid its use : use sigaction(2) instead."

Dans cet exercice, il ne faut pas créer un nouveau processus avec `fork(2)`.

Question 20.1 Écrire un programme `invincible`, qui écrit «boom !» quand il reçoit `SIGINT`, et qui continue de s'exécuter. Indices : `sigaction(2)`, `pause(2)`, `signal(7)`.

Question 20.2 Faire en sorte qu'il abandonne au bout de n tentatives (valeur donnée en argument de la commande).

```
$ ./invincible 5
^Cboom!
^Cboom!
^Cboom!
^Cboom!
^CKABOOM!
$
```

4. L'ajout de l'option `-D_DEFAULT_SOURCE` (transmise à `gcc`) est équivalente à l'ajout d'un `#define _DEFAULT_SOURCE` au tout début du fichier source, c'est à dire **avant** les inclusions des fichiers d'en-têtes système. Les fichiers d'en-têtes système contiennent des directives de compilation condionnelle qui permettent de sélectionner les déclarations et les définitions finalement "incluses" dans le fichier source après le passage du préprocesseur.

Question 20.3 Modifier le pour qu'il compte les moutons (1 par seconde) lorsqu'il n'est pas embêté. Mais faites en sorte qu'il reste inerte pendant les cinq secondes qui suivent un «boom !», s'il ne reçoit pas pendant ces 5 secondes un nouveau signal `SIGINT`. Les «boom !» doivent pouvoir se succéder "instantanément". Indice : `alarm(2)`.

```
$ ./invincible 2
1
2
^Cboom!
3
4
5
^CKABOOM!
$ ./invincible 3
1
2
3
^CBoom!
^CBoom!
^CKABOOM!
$
```

Exercice 21 : La commande `kill(1)`

La commande `kill(1)` permet d'envoyer un signal à un processus, `SIGTERM` par défaut. Le signal peut être précisé avec l'option `-s` suivi du numéro du signal voulu. Les processus destinataires du signal sont désignés par leur PID.

Synopsis du programme à écrire :

```
kill [-s SIG] PID...
```

Remarques :
- Utiliser la commande `kill -l` pour visualiser les signaux existants et leur numéro.
- Dans le programme à écrire, le signal à envoyer (i.e. l'argument `SIG`) ne pourra être transmis que sous la forme d'un entier ; les noms symboliques des signaux, comme `SIGTERM`, `SIGKILL`, etc., ne seront pas acceptés.

Question 21.1 Déterminer le signal à envoyer. On supposera que l'option `-s` ne peut apparaître qu'en première position.

Question 21.2 Envoyer le signal à chacun des processus donné en argument. Indice : `kill(2)`

Exemples d'exécution :

```
$ sleep 100 &
[1] 9356
$ sleep 100 &
```

```
[2] 9357
$ sleep 100 &
[3] 9358
$ ./kill -s SIGTERM 9356
SIGTERM : undefined signal
Usage: kill [-s signal] pid ...
$ ./kill -s 0 9356 azerty 9357 10000 9358
azerty : invalid pid
kill (10000): No such process
$ ./kill -s 3 9356 azerty 9357 10000 9358
azerty : invalid pid
kill (10000): No such process
$
[1]   Quitter                 (core dumped) sleep 100
[2]-  Quitter                 (core dumped) sleep 100
[3]+  Quitter                 (core dumped) sleep 100
$
$ ./kill 9356 azerty 9357 9358 10000
kill (9356): No such process
azerty : invalid pid
kill (9357): No such process
kill (9358): No such process
kill (10000): No such process
$
$ ./kill -s 15
Usage: kill [-s signal] pid ...
$ echo $?
1
```

Exercice 22 : Lanceur de commande

Nous allons réaliser un lanceur de commande `launch` chargé de lancer la commande passée en argument et de calculer le temps réel que la commande a mis pour s'exécuter.

Question 22.1 Récupérer l'heure courante à l'aide de `clock_gettime(2)`. Cette fonction permet de récupérer l'heure courante à l'aide d'une structure de type `timespec` :

```
struct timespec {
    time_t   tv_sec;        /* seconds */
    long     tv_nsec;       /* nanoseconds */
};
int clock_gettime(clockid_t clockid, struct timespec *tp);
```

Pour le paramètre `clockid`, vous pouvez utiliser la constante macrodéfinie `CLOCK_MONOTONIC_RAW`.

Question 22.2 Créer un nouveau processus et exécuter la commande passée en argument dans le fils, le père attendant la terminaison du fils. On utilisera la variante suivante de la famille `exec(3)` :

```
int execvp(const char *file, char *const argv[]);
```

Question 22.3 Le paramètre transmis à `wait(2)` permet de récupérer des informations sur le status de terminaison du fils. Utiliser les macros adéquates WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG (cf. manpage de `wait(2)`) pour afficher comment le processus fils s'est terminé.

Question 22.4 Récupérer l'heure courante puis calculer la durée (très approximative) du processus fils.

Exemples d'exécution :

```
$ ./launch sleep 2
8071 exited, status=0
Duration of child process : 2002239977ns

$ ./launch sleep 100 &
[1] 8131
$ ps -H
  PID TTY          TIME CMD
 7951 pts/0    00:00:00 bash
 8131 pts/0    00:00:00   launch
 8132 pts/0    00:00:00     sleep
 8133 pts/0    00:00:00   ps
$ kill -s SIGQUIT 8132
$ 8132 killed by signal 3
Duration of child process : 16700055431ns

[1]+  Fini                    ./launch sleep 100
$
```

Exercice 23 : La fonction `system(3)`

Le but de cet exercice est d'implémenter la fonction `system(3)` dont le prototype est :

```
int system(const char *command);
```

La fonction `system()` exécute la commande indiquée dans `command` en appelant `/bin/sh -c command`, et revient après l'exécution complète de la commande. Durant cette exécution, le signal `SIGCHLD` est masqué (uniquement dans le père), et les signaux `SIGINT` et `SIGQUIT` sont ignorés dans le père (ces 2 signaux seront traités conformément à leurs valeurs par défaut dans le processus fils).

Valeur renvoyée par la fonction `system()` :
- Si `command` est `NULL`, la fonction `system()` renvoie 1 (il n'y a pas de création de nouveau processus dans ce cas)
- Si un nouveau processus ne peut pas être créé, ou si son statut ne peut pas être récupéré par `waitpid(2)`, la fonction `system()` renvoie -1
- Si le recouvrement échoue, le processus fils se termine normalement avec un statut de terminaison égal à 127
- Sinon, elle renvoie le status de terminaison du processus fils récupéré dans le père par un appel de `waitpid(2)`

Question 23.1 Implémenter la fonction `system()`. On justifiera précisément la version de la famille `exec` utilisée à l'aide d'un commentaire dans le code.

Question 23.2 Dans la fonction `main()` (qui appelle votre fonction `system()`), utiliser les macros adéquates (WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG) pour afficher comment le processus fils s'est terminé (cf. manpage de `wait(2)`).

Exemples d'exécution à tester :

```
$ ./system "sleep 10"            --> avec ou sans ^C
$ ./system "ps -lH"
$ ./system "ps -lH | tee /dev/tty | wc -l"
```

Conclusion : la fonction `system()` fournit de la facilité et de la commodité : elle s'occupe de tous les détails d'appel de `fork(2)`, `execl(3)` et `waitpid(2)`, ainsi que des manipulations nécessaires des signaux ; de plus, l'interpréteur réalise les substitutions habituelles et les redirections d'entrées et sorties pour `command`. Le coût principal de `system()` est l'**inefficacité** : des appels système supplémentaires sont nécessaires pour créer le processus qui exécute l'interpréteur shell et pour exécuter ce processus.