Université Joseph Fourier, Grenoble
Licence d'Informatique L3

Introduction aux Systèmes et Réseaux

TP n°2 : Signaux et mesure du temps

1 Découvrir les signaux

Exercice 1 Faire `man signal`. Comprendre la différence entre `SIGINT, SIGQUIT, SIGTERM, SIGKILL`

Exercice 2 Ctrl-C : lancer un programme, qui ne se termine pas immédiatement (p.ex une boucle infinie), en premier plan et faire Ctrl-C. Que se passe-t-il ? (vérifier avec `ps`)

Exercice 3 Ctrl-Z : lancer un programme, qui ne se termine pas immédiatement (p.ex une boucle infinie), en premier plan et faire Ctrl-Z. Que se passe-t-il ? (vérifier avec `ps`)

Exercice 4 `fg, bg` : lancer un programme en arrière plan (en utilisant &), visualiser le processus avec `ps` et passer le processus en premier plan avec `fg`. Suspendre le processus et le relancer en arrière plan.

Exercice 5 Lancer un programme qui ne se termine pas et tester la primitive `kill -s ...` en ligne de commande.

2 Traitement des signaux d'interruption

Exercice 6 On rappelle que la primitive `unsigned int sleep(unsigned int t)` suspend le processus appelant pendant `t` secondes ou jusqu'à l'arrivée d'un signal. Dans ce dernier cas, elle renvoie le nombre de secondes qui restaient à attendre.

Écrire un programme qui suspend le processus pendant un nombre de secondes passé en paramètre, et imprime le nombre de secondes effectivement passées à attendre, le programme pouvant être interrompu par *control-C* (programme vu en TD).

Exercice 7 Écrire un programme qui, lorsqu'il reçoit le signal `SIGINT`, affiche le numéro de ce signal, et ne fait rien le reste du temps (on rappelle que la primitive `pause()` permet à un processus de se bloquer en attendant l'arrivée d'un signal). Faire de même avec `SIGKILL` et `SIGSTOP`. Que constatez-vous ?

Exercice 8 Exécuter le programme `counterprob.c`. En ajoutant un `sleep()` dans le programme des fils (après l'instruction `Kill`), vérifier que l'on peut obtenir différentes valeurs (entre 1 et 5) pour la valeur finale de `counter`.

3 Traitement de la terminaison d'un fils

Exercice 9 Reprendre le programme `waitpid1.c` vu en TP1. Modifier ce programme pour qu'il ait l'effet suivant :

- chaque processus fils essaie d'écrire dans un emplacement protégé contre l'écriture [1]
- le père indique la fin anormale des fils en imprimant le numéro de chaque fils et la cause de sa terminaison (numéro de signal).

Elements de solution : rappels et compléments On rappelle (cf. TD1) que `waitpid` renvoie un entier (via le paramètre `statut`), que l'on peut tester pour connaître l'état du processus qui s'est terminé. Nous avons déjà vu que `WIFEXITED(ptrStatut)` retourne vrai si le fils s'est terminé normalement et que `WEXITSTATUS(ptrStatut)` donne le code de retour.

Ici, nous allons utiliser :

- `WIFSIGNALED(statut)` renvoie vrai si la fin du processus est due à un signal ;
- si `WIFSIGNALED(statut)` renvoie vrai, alors `WTERMSIG(statut)` renvoie le numéro du signal qui a causé la terminaison du processus ;
- la fonction `psignal` permet d'afficher un message explicitant la cause associée à une numéro de signal donné (voir `man psignal`) [2].

Exercice 10

Lorsqu'un processus crée des processus fils, il peut attendre leur fin avec la primitive `wait` ou `waitpid`. Néanmoins, il ne peut pas faire de travail utile pendant cette attente. C'est pourquoi on souhaite que le processus père traite la fin de ses fils uniquement au moment où cette fin est signalée par le signal `SIGCHLD`.

Nous vous fournissons un programme appelé `signal1.c`, dans lequel le père enregistre un traitant pour le signal `SIGCHLD` et crée plusieurs fils. Puis le père se met à exécuter une boucle infinie.

Exécuter ce programme, attendre que le père se mette en boucle, le suspendre par *control-Z* et vérifier, avec la commande `ps`, que tous les fils n'ont pas été collectés (il reste au moins un zombi). Cela résulte du fait que les signaux ne sont pas tous mémorisés (un seul signal d'un type donné peut être dans l'état *pendant*).

Modifier le programme pour corriger cette erreur. Il y a deux aspects à prendre en compte. D'une part, il faut compenser la perte potentielle de signaux décrite ci-dessus (c'est-à-dire ramasser tous les zombis, même si leur nombre est supérieur au nombre de signaux `SIGCHLD` reçus). D'autre part, en réglant ce problème, il faut veiller à ne pas réintroduire le problème initial : le père ne doit pas se bloquer en attente de ses fils (ni dans le programme principal, ni dans le traitant de signaux). Pour le second aspect, l'option `WNOHANG` de `waitpid` est utile (pour sa signification précise, voir `man waitpid`).

1. Pour cela, on peut par exemple tenter d'écrire un octet à l'adresse `0` ou encore à l'adresse associée à l'étiquette du début du code du programme (`main`). Dans tous les cas, afin d'éviter les erreurs de compilation, il faut utiliser une conversion de type (*cast*), pour manipuler l'adresse comme celle d'un tableau d'octets.

2. Plus précisément, `psignal` prend deux arguments : un numéro de signal et un message personnalisé à afficher avant la description du signal. L'affichage est effectué sur la sortie d'erreur (`stderr`).

4 Pour les curieux (*bonus*) : Mesure du temps

Les systèmes Unix fournissent des outils de mesure du temps, pour obtenir des statistiques telles que celles vues au TD n°2. Ces mesures utilisent une horloge interne au système. La fréquence de cette horloge (nombre d'impulsions, ou "tics" par seconde) est un paramètre du système, que l'on peut consulter en appelant `sysconf(_SC_CLK_TCK)`. La primitive `sysconf` permet par ailleurs d'accéder à de nombreux paramètres du système, voir `man sysconf`.

On trouvera dans le placard sous le nom `cpufraction.c` un programme qui permet de mesurer le temps d'exécution d'un programme donné et le taux d'utilisation du processeur par ce programme. Pour cela, on utilise une structure interne `tms` définie ci-après (dans `<sys/times.h>`). Pour chaque champs de cette structure, la valeur est exprimée en nombre de "tics" d'horloge.

```
struct tms {
    clock_t tms_utime;  /* user time */
    clock_t tms_stime;  /* system time */
    clock_t tms_cutime; /* user time of terminated children */
    clock_t tms_cstime; /* system time of terminated children */
};
```

La fonction `times` permet de faire des mesures en utilisant des variables conformes à cette structure (voir `man 2 times` sous Linux).

(a) Examiner le programme `cpufraction.c` pour comprendre son fonctionnement. Utiliser ce programme pour mesurer le temps d'exécution d'une fonction que vous construirez (par exemple exécuter une boucle vide 10^7 fois). Cette fonction devra avoir pour nom `function_to_time()` car elle est ainsi désignée dans `cpufraction.c` ; ce nom pourra être modifié à condition de modifier `cpufraction.c`. Faire les mesures plusieurs fois ; que constatez-vous ?

(b) Modifier le code de `function_to_time()` pour ajouter des appels système (par exemple, en insérant un appel à `getppid()` toutes les 500 itérations de la boucle) [3]. Relancer le programme et comparer les résultats aux précédents.

(c) Reprendre la première version du programme (sans appels système). Exécuter le programme en parallèle avec lui-même en 2 ou 3 exemplaires ; que constatez-vous ? Dans quel(s) cas est-il possible que la fraction de temps CPU utilisé par un processus ne diminue pas malgré le lancement d'autres processus en parallèle ?

(d) Écrire un nouveau programme qui crée un processus fils, attend la terminaison du fils puis (en utilisant la structure `tms`) affiche le temps passé par le fils à s'exécuter sur le processeur.

3. DANGER Éviter d'utiliser l'appel système `Fork()` dans la boucle pour ne pas saturer le système.