Signaux

Thèmes

- Définition d'un traitant de signal, attente et envoi d'un signal
- Mécanismes de masquage et démasquage des signaux.
- Utilisations d'alarmes
- Sauvegarde et restauration de points de reprise

Ressources : pour ce TP, comme pour les suivants, vous pourrez vous appuyer sur

- Le polycopié intitulé « Systèmes d'exploitation : Unix », qui fournit une référence généralement suffisante sur la sémantique et la syntaxe d'appel des différentes primitives de l'API Unix. Chaque section du sujet de TP indique la (ou les) section(s) du polycopié correspondant au contenu présenté.
- Les pages du manuel en ligne (commande `man`), et plus particulièrement les sections 2 et 3.

Déroulement et objectifs pour la séance : La durée de la séance de TP devrait (normalement) (à peu près) permettre à tous de traiter (en comptant une dizaine de minutes par exercice) :

- les exercices 1 à 6 de la section 1.1 ;
- les questions 1 à 3 de la section 1.2 ;
- l'exercice de la section 2.

Rendu (9/4) : réponses aux questions de la section 1.2, et code source correspondant.

0 Avant de commencer...

La réalisation du TP demande une connaissance de base de la syntaxe d'appel et de la sémantique des primitives de l'API signaux Unix. Ces notions sont présentées de manière progressive dans le tutoriel proposé en **préparation** du TP (*Attention* : ce tutoriel ne remplace en aucun cas le TP ; il devrait être suivi en dehors de la séance de TP, dans le cas où vous estimeriez utile d'avoir une présentation « en douceur » de l'API).

Le QCM accompagnant ce TP va vous permettre de vous situer par rapport à cette connaissance de base. Comptez une petite dizaine de minutes. Si votre score est inférieur à 75/100, vous auriez sans doute (eu) intérêt à jeter un coup d'œil au tutoriel...

1 Les signaux (polycopié API Unix, section 2.3)

1.1 Opérations essentielles

Un signal est une notification asynchrone envoyée à un processus afin de signifier l'occurrence d'un événement externe à ce processus. Lorsqu'un signal est envoyé à un processus, le système d'exploitation interrompt l'exécution du processus. Si le processus a déjà enregistré un *traitant de signal*, ce traitant est exécuté. Sinon, le traitant par défaut est exécuté (un traitant par défaut est défini pour chacun des signaux). Les opérations suivantes permettent de gérer les signaux :

- `signal(...)` (paragraphe 2.3.3) et `sigaction(...)` (`man sigaction(...)`) permettent d'associer un traitant à un signal. Ce traitant peut être une fonction définie par l'utilisateur, ou être l'un des deux traitants prédéfinis : ignorer le signal (`SIG_IGN`) ou utiliser le traitant par défaut (`SIG_DFL`) Deux signaux conservent toujours leur traitant par défaut, et ne peuvent donc pas être interceptés : `SIGKILL` et `SIGSTOP`.
  A l'issue d'un appel à `fork()`, le processus fils hérite des gestionnaires des signaux définis par son père.
- `kill(...)` (paragraphe 2.3.2) permet d'envoyer un signal à un processus [1].
- `pause()` (paragraphe 2.3.5) permet d'attendre un signal.
- `alarm(...)` (paragraphe 2.3.6) permet de programmer l'envoi d'un signal `SIGALRM` à une échéance définie en secondes.

1. ou au groupe de processus |pid| dans le cas où le pid est négatif.

Traitants et cycle de vie des processus

- A l'issue d'un appel à `fork()`, le processus fils hérite des associations entre traitants et signaux définies par son père.
- Le recouvrement (primitives `exec`) ne conserve que les associations au traitant`SIG_IGN`. Les autres traitants associés par le processus avant l'appel à exec `exec` sont remplacés par le traitant par défaut spécifique à chaque signal (`SIG_DFL`).

Exercices

*Pour chacune des questions qui suivent, prédire l'effet attendu **avant** d'effectuer l'action.*

1. On considère le programme suivant (disponible avec le sujet sous le nom `sgn.c`) :

```
#include <stdio.h>    /* entrées/sorties */
#include <unistd.h>   /* primitives de base : fork, ...*/
#include <stdlib.h>   /* exit */

#define MAX_PAUSES 10     /* nombre d'attentes maximum */

int main(int argc, char *argv[]) {
        int nbPauses;

        nbPauses = 0;
        printf("Processus de pid %d\n", getpid());
        for (nbPauses = 0 ; nbPauses < MAX_PAUSES ; nbPauses++) {
                pause();                // Attente d'un signal
                printf("pid = %d - NbPauses = %d\n", getpid(), nbPauses);
    } ;
    return EXIT_SUCCESS;
}
```

   Il s'agit d'observer le comportement d'un processus exécutant ce programme en réponse à différentes actions.
   (Compiler puis) lancer ce programme. Pour observer l'état du du processus exécutant le programme, vous pouvez lancer la commande `"ps l"` dans un autre terminal. Quel sera l'état de ce processus

   - juste après son lancement ?
   - après avoir entré `Ctrl-Z` depuis le terminal de lancement ?
   - après avoir lancé `kill -CONT 0000` (où `0000` doit être remplacé par le pid du processus observé) depuis un autre terminal ?
   - après avoir entré `"fg 1"` dans le terminal de lancement ?
   - après `kill -INT 0000` depuis un autre terminal (ou `Ctrl-C` dans le terminal de lancement) ?

   Relancer un nouveau processus. Quel sera l'état de ce processus

   - après avoir lancé `kill -STOP 0000` depuis un autre terminal (ou `Ctrl-Z` dans le terminal de lancement) ?
   - après avoir lancé `kill -INT 0000` (ou `kill -QUIT 0000`) depuis un autre terminal ?
   - après avoir lancé `kill -CONT 0000` depuis un autre terminal ?

   Relancer un nouveau processus. Quel sera l'état de ce processus

   - après avoir lancé `kill -STOP 0000` depuis un autre terminal (ou `Ctrl-Z` dans le terminal de lancement) ?
   - après avoir lancé `kill -KILL 0000` depuis un autre terminal ?

2. Dupliquer le programme `sgn.c` et compléter la copie en associant à tous les signaux (de `1` à `SIGRTMAX`) un traitant qui affiche le pid du processus et le numéro du signal reçu.

   Tester avec `Ctrl-Z`, `Ctrl-C`, `kill -USR1`, `kill -STOP`, `kill -CONT`, `kill -KILL`.

   Quel est le comportement du processus vis-à-vis des signaux SIGSTOP, SIGKILL, et SIGCONT ?
3. Compléter (dans une copie) le code de l'exercice 2 afin que le programme crée un processus fils juste après l'association des signaux avec le traitant. Le fils exécute le même code que le père.

Effectuer les mêmes tests que pour l'exercice 2 [2]. Quelles conclusions peut-on en tirer ?

4. Compléter (dans une copie) le code de l'exercice 3 afin que le code du fils (seulement) se réduise à un appel à `sleep(100)`.

   Tester avec `Ctrl-Z`, puis `"ps l"` dans un autre terminal. Quels signaux le père a-t-il reçus ? Que s'est-il passé pour le fils ? [3]

   Modifier en remplaçant l'appel à `sleep(100)` par une une boucle répétant 100 fois un appel à `sleep(1)`, puis tester à nouveau.
5. Compléter (dans une copie) le code de l'exercice 4 en remplaçant la boucle `sleep` du processus fils par le chargement de la commande `sleep`, par recouvrement
   (par exemple : `execl("/bin/sleep", "/bin/sleep", "100", NULL)`).

   - Tester avec `Ctrl-C`. Que s'est-il passé pour le fils ? Quel est son état ?
   - Relancer et tester avec `Ctrl-Z`. Quel est l'état du fils ? Quels signaux le père a-t-il reçus ?
   - Dans un autre terminal, lancer `kill -CONT`, avec le pid du fils. Quel est l'état du fils ? Quels signaux le père a-t-il reçus ?
6. Dupliquer le programme `sgn.c` et compléter la copie en ajouter ce qu'il faut, sans modifier la boucle du père et sans utiliser `sleep(—)`, afin qu'un message indiquant le pid du père soit affiché toutes les 5 secondes.

Compléments (***optionnels***, à voir à votre gré, ***après*** la séance de TP)

- exercices de révision/récapitulation (avec réponses) autour de la primitive `sleep(--)`, de `SIGINT` et de l'**héritage des signaux**, sous la forme d'un QCM de 4 questions.
- **utilisation d'horloges**
  Quelques exercices sont disponibles en annexe, pour illustrer l'utilisation d'`alarm(—)`, puis d'horloges programmables. Le document présente pour finir une API d'horloges programmables à haute résolution (de l'ordre de la nanoseconde).

1.2 Gestion des masques de signaux (polycopié API Unix, paragraphe 2.3.4)

Exercice L'objectif de cet exercice est d'utiliser les fonctions de l'interface POSIX pour la manipulation des signaux et des masques de signaux.. Ecrire un programme

1. qui associe à $SIGUSR1$ et $SIGUSR2$ un traitant affichant le numéro du signal reçu.
   Le message affiché par le traitant devra **exactement** être la chaîne constituée par le mot Reception, suivi d'un espace, suivi du numéro du signal recu.
   Exemple : `"Reception 24"` suite à la réception du signal 24.
2. puis **masque** les signaux $SIGINT$ et $SIGUSR1$.
3. puis attend durant 10s au plus la réception d'un signal non masqué et non ignoré, laps de temps pendant lequel $SIGINT$ est envoyé via le clavier ;
4. puis s'envoie 2 $SIGUSR1$, attend durant 5 s au plus la réception d'un signal non masqué et non ignoré, et s'envoie 2 $SIGUSR2$
5. démasque $SIGUSR1$;
6. attend durant 10s au plus la réception d'un signal non masqué ou ignoré, puis démasque $SIGINT$.
7. enfin, affiche un message de terminaison commençant (**exactement**) par le mot Salut

Indication : l'attente d'un délai fixé `d` pourra être simplement implanté par appel à la primitive `sleep(d)`.

2. Il existe un comportement particulier associé au terminal : la frappe de certains caractères, (comme `ctrl-C` ou `ctrl-Z`) est traduite par l'envoi de signaux (comme `SIGINT` ou `SIGTSTP`) au processus en avant-plan. Dans ce cas (et seulement dans ce cas), le signal est également transmis aux descendants du processus en avant-plan.

3. Il faut savoir que la réception d'un signal fait sortir de la primitive sleep (et la valeur retournée est le temps qu'il restait avant que sleep ne se termine ≪ naturellement ≫)

Questions

On suppose que $SIGINT$ est envoyé à temps lors de l'étape (3).

1. combien de $SIGUSR1$ et de $SIGUSR2$ seront ils affichés ?
2. quel sera l'ordre d'affichage ?
3. au bout de combien de temps (à 2 secondes près) s'affichera le message de terminaison ?

2 Gestion de points de reprise (polycopié API Unix, section 2.4)

Exercice Le but de cet exercice est d'illustrer le fait que `setjmp` ne sauvegarde qu'une partie du contexte processeur, et *non le contenu* de la pile d'exécution. Autrement dit, par exemple, la référence vers le bloc d'activation courant est sauvegardée, mais non le contenu du bloc d'activation.
Ecrire un programme qui initialise une variable locale à 0, puis sauvegarde un point de reprise, affiche la valeur de la variable, l'incrémente et effectue un `longjmp` vers le point de reprise sauvegardé. Avant de terminer, le programme affiche la valeur finale de la variable.

- Si vous avez scrupuleusement réalisé le comportement demandé, qu'allez-vous observer ?
- Modifiez ce programme pour assurer que le `longjmp` n'est effectué qu'une fois.
- Quelles seront les valeurs successives affichées pour la variable ? Vérifiez votre réponse.

Complément : utilisation des points de reprise pour réinitialiser des tâches périodiques
Une annexe propose la réalisation d'une petite application qui fait appel aux signaux, aux alarmes et aux points de reprise.

3 Indications pour le projet

Étape 6 (*remise le 17/4)*

Le traitement de cette étape s'appuie sur quelques points techniques précis. En particulier, sur l'utilisation de la primitive `waitpid`, qui permet de suivre les changements d'état des fils d'un processus :

```
pid_t waitpid (pid_t pidf, int *status, int options)
```

retourne le pid du fils ayant changé d'état (ou -1 en cas d'erreur), où

- `pidf` est le pid du fils à attendre. (-1 : n'importe quel fils)
- `options` : est un ou logique entre zéro ou plusieurs parmi les constantes
  - `WNOHANG` : retour immédiat même si aucun changement d'état (dans ce cas le retour vaut 0) ;
  - `WUNTRACED` : retour si un fils a été suspendu ;
  - `WCONTINUED` : retour si un fils suspendu a été relancé par le signal `SIGCONT`.

  `waitpid (-1, &status, 0)` est équivalent à `wait(&status)`
- `status` pointe un entier contenant les informations d'état du fils ayant débloqué waitpid(–). Comme pour wait(–), cet entier peut être examiné avec les macros :
  - `WIFSTOPPED(status)` renvoie vrai si le fils a été suspendu
  - `WSTOPSIG(status)` : numéro du signal qui a causé la suspension du fils
  - `WIFCONTINUED(status)` renvoie vrai si le processus fils a été relancé par le signal `SIGCONT`.
  - et `WIFEXITED, WEXITSATUS, WIFSIGNALED, WTERMSIG`.

Ce point est présenté, avec quelques autres, dans une courte note, disponible sur Moodle, qui récapitule les principales difficultés posées par le traitement de l'étape 6.

Étapes 7 et 8 (*remises avec le rendu intermédiaire)*

Une note, disponible sur Moodle, récapitule les principales difficultés posées par le traitement de ces étapes, et fournit les éléments techniques nécessaires à leur résolution.

En termes de connaissances, avec ce TP et la lecture de la note mentionnée plus haut, vous disposerez alors des éléments permettant de réaliser les étapes 7 et 8 du projet. Cependant, ces étapes demandent un temps de réflexion, et de mise au point. En effet, les mécanismes mis en jeu doivent être bien maîtrisés, et le test de programmes parallèles est nettement moins simple et rapide que le test de programmes séquentiel.

Par conséquent, il n'est pas attendu pour le rendu intermédiaire que ces étapes soient pleinement et correctement traitées. Par contre, il est demandé qu'une proposition d'implémentation soit en place, que les difficultés aient été bien identifiées, et que des pistes de solutions soient clairement ébauchées.