Signaux

Thèmes

- Définition d'un traitant de signal, attente et envoi d'un signal
- Mécanismes de masquage et démasquage des signaux.
- Programmation d'horloges
- 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.

Cheminement et objectifs pour la séance :

- La durée de la séance de TP devrait (normalement) (à peu près) permettre à tous de traiter :
  - les exercices 1 à 5 de la section 1.1 ;
  - les questions 1 à 3 de la section 1.2 ;
  - l'exercice 1 de la section 2.

  Les exercices suivants des sections 1.1 et 2 demandent sensiblement plus de temps et sont proposés à ceux qui souhaitent davantage approfondir ou pratiquer. Il n'est donc pas attendu que ces exercices, marqués [*], [**], [***], selon le temps de réalisation estimé et distingués par un fond gris clair soient traités durant le TP.
- La section 3 est fournie à titre de simple documentation. Elle présente une API d'horloges à haute résolution.

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.

Exercices

1. Écrire un programme qui imprime le numéro de chaque signal qu'il reçoit. Tester en envoyant des signaux (avec `kill`) depuis un autre terminal.
2. Étendre l'exercice 1 afin que le programme montre qu'il est toujours actif en imprimant un message toutes les 5 secondes, puis s'arrête au bout d'une minute.
3. Étendre l'exercice 2 afin que le programme crée un processus fils. Le programme doit permettre d'observer que le fils hérite des traitants de signaux de son père. [2]. Tester en envoyant des signaux depuis un autre terminal.
4. Modifier l'exercice 3 afin que le processus fils charge par recouvrement un programme (par exemple : `execl("/bin/sleep", "/bin/sleep", "100", NULL)`).
   Observer que les traitants de signaux par défaut sont alors rétablis.
5. Modifier l'exercice 4 afin que le père attende la fin du processus fils et affiche le code de retour du processus fils (ou le signal ayant provoqué la terminaison du fils) . Pour cela, utilisez la function `wait` et les macros `WIFEXITED`/`WEXITSTATUS`, `WIFSIGNALED`/`WTERMSIG`...
6. [*] **Utilisation d'alarmes**. Ecrire un programme qui crée un processus fils. Le processus fils affiche son pid, celui de son père, puis entre dans une boucle sans fin. Chaque fois que le processus père reçoit un SIGALRM, il imprime un message. Après 5 signaux reçus, le père tue le processus fils et se termine.
7. [**] Ecrire un programme qui crée un processus fils. Le père affichera les entiers pairs compris entre 1 et 100 ; le fils affichera les entiers impairs compris dans le même intervalle. Synchroniser les processus à l'aide de signaux pour que l'affichage soit 1 2 3 ... 100.
8. [*] Modifier l'exercice 6 en remplaçant l'utilisation d'`alarm()`/`SIGALRM` par des horloges programmables (paragraphe 2.3.7, <`sys/time.h`>).

1.2 Gestion des masques de signaux (paragraphe 2.3.4)

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

- qui associe à $SIGUSR1$ et $SIGUSR2$ un traitant affichant le numéro du signal reçu.
- puis masque les signaux $SIGINT$ et $SIGUSR1$.
- puis attend 10s, durant lesquelles $SIGINT$ est envoyé via le clavier ;
- puis s'envoie 2 $SIGUSR1$, attend 5 secondes, et s'envoie 2 $SIGUSR2$
- démasque $SIGUSR1$;
- attend 10s, puis démasque $SIGINT$.
- enfin, affiche un message de terminaison.

Répondez aux questions suivantes **avant** de lancer le programme, puis vérifiez que vos réponses sont correctes. Le cas échéant, donnez une explication pour les comportements inattendus.

Questions

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. 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.

2 Gestion de points de reprise (section 2.4)

Exercices

1. 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.

2. [***] Pour cet exercice, il est nécessaire d'utiliser `siglongjmp/sigsetjmp`, qui assurent la sauvegarde du masques de signaux.
   Testez vos réflexes. Ce programme effectue une série de 10 tests. Un test consiste à
   - attendre pour une durée variable (en secondes, dans un premier temps),
   - afficher un message donnant un chiffre (variant pour chaque test) que l'utilisateur doit saisir.
   - à partir de ce moment, l'utilisateur dispose de deux secondes pour saisir ce chiffre au clavier.
     - si l'utilisateur a réagi dans les temps, cela est comptabilisé comme un succès, un message d'acquittement est affiché, et l'exécution enchaîne directement sur le test suivant ;
     - sinon, en l'absence de réaction dans les temps, le test se poursuit avec l'affichage d'un message d'échec.

   Lorsque l'ensemble des tests est fini, un message est affiché donnant le score global.
3. [**] Réaliser une seconde version du programme précédent, prenant en paramètre
   - le délai maximal de réaction (en millisecondes). Pour cela, utiliser des timers.
   - ainsi que le nombre de tests à effectuer.

   En outre, cette version proposera un chiffre aléatoire dans le message d'invite, et réalisera une attente aléatoire avant l'affichage du message d'invite. Pour cela, il est possible d'utiliser les fonctions `srandom` et `random` de `stdlib.h`

Complément : les horloges à haute résolution

Le module <time.h> propose des horloges encore plus fines que celles présentées dans le paragraphe 2.3.7 du polycopié. Ces horloges, définies par la norme *POSIX*, et disponibles sous *Linux* :

- ont une précision supérieure (de l'ordre de la nanoseconde) ;
- permettent de déclencher n'importe quel signal ;
- permettent de lancer un thread à leur échéance.

La mise en œuvre de ces horloges se fait en deux étapes :

1. Initialisation de la structure `sigevent` suivie de la création de l'horloge.

```
timer_t timerid;
struct sigevent sev;
sev.sigev_notify = SIGEV_SIGNAL;
sev.sigev_signo = SIGUSR1;
//pour lancer un thread qui execute la fonction my_handler
//sev.sigev_notify = SIGEV_THREAD;
//sev.sigev_notify_function=(void*)my_handler;
sev.sigev_value.sival_ptr = &timerid;
if (timer_create(CLOCKID, &sev, &timerid) == -1)
        printf("error timer_create");
```

   Une horloge peut être associée à un signal donné via le champ `sigev_signo`.

2. L'initialisation de la structure `itimerspec` suivie du démarrage de l'horloge.

```
struct itimerspec its;
its.it_value.tv_sec = seconds;
its.it_value.tv_nsec = nanoseconds;
its.it_interval.tv_sec = its.it_value.tv_sec;
its.it_interval.tv_nsec = its.it_value.tv_nsec;
if (timer_settime(timerid, 0, &its, NULL) == -1)
        printf("error timer_settime");
```

Les éléments `tv_sec` et `tv_nsec` spécifient le nombre de secondes/nanosecondes de l'horloge. Pour plus d'informations, voir `https://perkamon.alioth.debian.org/online/man2/timer_create.2.php.fr`.