Recouvrement de processus - Signaux

Thèmes traités

- Recouvrement de code.
- Notions de base et mise en œuvre des signaux UNIX.
- Opérations essentiellles de l'API signaux Unix, protocole d'usage.

1 Questions

1. Qu'apporte le mécanisme d'interruptions à la gestion et la supervision des E/S par le système d'exploitation ? Est ce que l'utilisation de ce mécanisme est pertinente dans tous les cas ? Pourquoi ?
2. Du point de vue de l'application et du point de vue des mécanismes mis en jeu, quelles sont les différences entre la réception d'un signal et l'appel d'une procédure ?

2 Chargement d'un code à exécuter : recouvrement de processus

```
int execl(char *chemin, char *arg0, char *arg1, ..., char *argn, 0);
int execlp(char *chemin, char *arg0, char *arg1, ..., char *argn, 0);
int execle(char *chemin, char *arg0, char *arg1, ..., char *argn, 0,
           char *env[]);
int execv(char *chemin, char *argv[]);
int execvp(char *chemin, char *argv[]);
int execve(char *chemin, char *argv[], char *env[]);
```

Les primitives `exec` sont une famille de primitives permettant le lancement de l'exécution d'un nouveau programme par un processus appelant. Cela n'entraîne pas la création d'un nouveau processus mais seulement le chargement et le lancement d'un nouveau programme exécutable, en lieu et place du programme contenant l'appel à la primitive `exec` : le nouveau programme recouvre (remplace directement) l'ancien.

Les lettres suivant `exec` signifient :

- l/v : liste/tableau(vecteur)
- p : utilisation de la variable `PATH` pour la recherche de la commande à exécuter
- e : passage de l'environnement

Après recouvrement :

- une nouvelle image mémoire est allouée ;
- les signaux non ignorés sont associés à leur traitant par défaut ;
- les descripteurs de fichiers restent ouverts, sauf ceux indiqués par `fcntl` (`FD_CLOEXEC`) ;
- les autres attributs sont conservés.

Question. *Écrire un programme qui :*

1. *crée un processus fils ;*
2. *dans le processus fils :*
   - *exécute la commande `ls -l` en utilisant `execl` ;*
   - *termine avec le code `EXIT_FAILURE` en cas d'échec.*
3. *dans le processus père :*

— *attend la terminaison du fils ;*
— *affiche le code de retour de la terminaison du fils.*

Question. *Modifier le code précédent en utilisant* `execv` *à la place de* `execl`*.*

Question. *Quel est le code de retour du fils si l'exécution de la commande* `exec` *réussie.*

3 Manipulation des signaux et gestion du temps

3.1 Notion de signal

Un signal traduit l'occurrence d'un événement « observé » par l'environnement du processus qui reçoit le signal :
— erreur liée à l'exécution du processus récepteur (accès erroné...)
— certains événements matériels (frappe de caractères particuliers...) transmis par le système
— événements applicatifs transmis par d'autres processus utilisateurs
Le comportement du processus à la réception du signal ressemble à celui des mécanismes d'interruption : arrêt de l'exécution en cours et appel d'une fonction de traitement. Il existe différents types de signaux, identifiés par un numéro, qui ont un traitement par défaut. Par exemple, la réception du signal `SIGINT` entraîne la terminaison du processus. La liste des différents signaux est définie dans le fichier `signal.h`. Il y en a `NSIG`-1.

Quelques signaux :

Nom	Événement correspondant	Traitement par défaut
`SIGHUP`	terminaison du leader	terminaison
`SIGINT`	control-C au clavier	terminaison
`SIGQUIT`	control-\au clavier	terminaison+core
`SIGSTP`	control-Z au clavier	suspension
`SIGCONT`	continuation d'un processus stoppé	reprise
`SIGKILL`	terminaison	terminaison
`SIGPIPE`	écriture dans un tube sans lecteur	terminaison
`SIGFPE`	erreur arithmétique (overflow...)	terminaison
`SIGCHLD`	l'état du fils a changé	ignoré
`SIGALRM`	interruption horloge	terminaison
`SIGTERM`	terminaison normale	terminaison
`SIGUSR1`	laissé à l'utilisateur	terminaison
`SIGUSR2`	laissé à l'utilisateur	terminaison

Remarque : Le nom des signaux est le même pour tous les UNIX. En revanche, leur valeur peut être différente. Par exemple, `SIGCHLD` vaut 17 sous Linux et 20 sous MacOS. En conséquence, toujours utiliser leur nom.

3.2 Association d'un traitement à un signal

```
int sigaction(int signal, const struct sigaction *nv_action,
              struct sigaction *anc_action);
```

La primitive `sigaction` permet de définir le comportement du processus à la réception d'un `signal`. Ce comportement est défini via `struct sigaction` :

```
struct sigaction {
    void (*sa_handler)(int); // le traitement associé
    sigset_t sa_mask;        // ensemble des signaux à masquer
    int sa_flags;            // options
}
```

Cette structure s'initialise de la façon suivante :

```
struct sigaction action;
action.sa_handler= traitement;
sigemptyset(&action.sa_mask);
action.sa_flags= 0;
```

> **Remarque :** Attention : il n'est pas possible de modifier le traitement associé aux signaux `SIGKILL` et `SIGSTOP`. `sigaction` retournera -1 en cas de tentative.

Traitement associé au signal. Le traitement est une procédure de la forme :

```
void traitement(int signal) { ... }
```

Deux traitements par défaut existent :
- `SIG_DFL` : traitement par défaut associé au signal ;
- `SIG_IGN` : traitement vide pour ignorer le signal.

Ensemble des signaux à masquer. L'ensemble des signaux à masquer est défini par le type abstrait de données `sigset_t`. Ce type dispose des fonctions :
- `int sigemptyset(sigset_t *set)` : `set` $\leftarrow \emptyset$
- `int sigfillset(sigset_t *set)` : `set` $\leftarrow \{1, ..., \texttt{NSIG}\}$
- `int sigaddset(sigset_t *set, int signal)` : `set` $\leftarrow$ `set` $\cup \{$ `signal` $\}$
- `int sigdelset(sigset_t *set, int signal)` : `set` $\leftarrow$ `set` $\setminus \{$ `signal` $\}$
- `int sigismember(sigset_t *set, int signal)` : retourne 1 si `signal` $\in$ `set`, 0 sinon.

Ces fonctions retournent -1 si erreur et 0 si ok (sauf `sigismember`).

Les options. Les options permettent des définir le comportement à adopter à l'exécution du traitement. Par exemple, `SA_RESETHAND` permet de reprendre le traitement par défaut après l'exécution du traitement courant associé au signal.

> **Remarque :** Il est possible d'utiliser la primitive `int signal(int sig, void (* traitement)(int))` pour associer un traitement au signal `sig`. Le comportement associé au signal après l'exécution du traitement n'est pas normalisé, parfois le traitement est repositionné à `SIG_DFL`, parfois non. A utiliser si on sait ce qu'on fait.

Héritage des signaux. Le processus fils hérite des traitements des signaux du processus père après exécution de `fork`. Après `exec` :
- le masque est conservé ;
- les signaux ignorés (associés à `SIG_IGN`) le restent ;
- les autres signaux reprennent leur traitement par défaut (`SIG_DFL`).

3.3 Attendre un signal

```
int pause();
```

La primitive `pause` permet d'attendre un signal quelconque.

3.4 Envoyer un signal

```
int kill(pid_t pid, int signal);
```

Cette primitive envoie un `signal` au processus identifié par son `pid`.

```
int alarm(int sec);
```

Cette primitive initialise l'envoi du signal `SIGALRM` après `sec` secondes.

3.5 Exercice

Le but de l'exercice est d'écrire un programme C qui affiche le numéro de tout signal reçu (signal émis depuis le terminal ou depuis un autre processus). Le programme devra :

1. indiquer toutes les 3 secondes qu'il est toujours actif et en attente de signaux ;
2. s'arrêter au bout de 27 secondes ou lorsque 5 signaux seront reçus.

Le gestion du temps sera gérée par la programmation de l'envoi du signal `SIGALRM`.

Question. *Écrire le programme qui réalise ce comportement.*

Question. *La primitive `alarm` est-elle la seule à envoyer le signal `SIGALRM` ? Quelle conséquence cela peut-il avoir si ce n'est pas le cas ?*