Systèmes d'exploitation

Cours no. 12

Jean-Sébastien Coron

Université du Luxembourg

November 22, 2009

Programmation en C sous UNIX

- Manipulation de processus.
  - Les commandes `fork()`,`wait()`.
- Communication inter-processus.
  - Les signaux

Les processus

- Création d'un processus.
  - Lorsqu'on entre une commande, le shell lance un processus pour l'exécuter.
  - Le shell attend la fin du processus, puis attend la commande suivante.
- Arborescence de processus.
  - Chaque processus a un *père*, celui qui l'a lancé.
  - Le numéro du processus du père est le PPID.
  - Un processus père peut avoir plusieurs processus *fils*
  - Sous UNIX, un premier processus `init` est créé avec un PID de 1. Ancètre de tous les processus.

Création de processus

- Duplication d'un processus.
  - Seule façon de créer un processus à bas niveau.
  - Appel système `fork()`
  - Crée une copie complête du processus, fils de ce processus.
  - Le fils change de programme en appelant `exec()`.
  - La fonction `exec()` remplace le code et ses données par celui d'une nouvelle commande.

Exécution d'une commande

- Exécution d'une commande: `tar -zcvf file *`
  - Le shell se duplique avec `fork()`. On a alors deux processus shell identiques.
  - Le shell père attend la fin du processus fils avec `wait()`.
  - Le shell fils remplace son exécutable par celui de la commande `tar` avec `exec()`
  - La commande `tar` s'exécute. Lorsqu'elle se termine, le processus fils disparait.
  - Le shell père est réactivé et attend la commande suivante.

La primitive `int fork()`

- L'appel a `fork()` duplique le processus.
  - L'exécution continue dans les deux processus.
- Valeur retournée par `fork()`:
  - Dans le processus père, retourne le PID du fils.
  - Dans le processus fils, retourne 0.
  - Si le fork échoue, renvoie -1.
- `getpid()` et `getppid()`
  - `getpid()` retourne le PID du processus.
  - `getppid()` retourne le PID du père du processus.

Exemple avec fork()

```
#include <stdio.h>
#include <unistd.h>
int main()
{
  int pidfils=fork();
  if(pidfils!=0)
  {
    printf("Je suis le père.");
    printf("PID de mon fils:%d\n",pidfils);
  } else
  {
    printf("Je suis le fils.");
    printf("Mon pid est:%d\n",getpid());
  }
}
```

Exemple avec `fork()`

- Résultat:
  - ```
    $ forksimple
    Je suis le père.PID de mon fils:2864
    Je suis le fils.Mon PID est:2864
    $
    ```
- Peut aussi apparaître dans l'ordre inverse.
    ```

wait et exit

- La primitive `wait` permet à un processus d'attendre la fin d'un de ses fils.
  - Si pas de fils ou erreur, retourne -1.
  - Sinon, retourne le PID du fils qui s'est terminé.
  - Doit être appelé avec `wait(0)`
- La primitive `exit` permet de terminer le processus qui l'appelle.
  - `void exit(int status)`
  - `status` permet d'indiquer au processus père qu'une erreur s'est produite.
  - `status`=0 si pas d'erreur.

La primitive `sleep`

- Similaire à la commande shell `sleep`
  - `int sleep(int seconds)`
  - Le processus qui appelle `sleep` est bloqué pendant le nombre de secondes spécifié.
- Différence avec `wait()`
  - `wait()` bloque jusqu'à la fin du fils, alors que `sleep()` bloque un temps spécifié.

Exemple

```
int pidfils=fork();

if(pidfils!=0) {
  printf("Je suis le père.\n");
  wait(0);
  printf("mon fils a terminé.\n");
} else {
  printf("je suis le fils.\n");
  sleep(2);
  printf("arrêt du fils.\n");
  exit(0);
}
```

Résultat

- Résultat
  - ```
    $ fork
    Je suis le père
    je suis le fils
    arrêt du fils
    mon fils a terminé
    $
    ```
    ```

Les signaux asynchrones

- Les signaux permettent d'indiquer à un processus qu'un certain évenement s'est produit.
  - Il existe 32 types de signaux prédéfinis.
  - Un signal peut-être envoyé par un processus ou par le système (lorsqu'une erreur se produit).
- Traitement des signaux.
  - Les signaux reçus peuvent être traités par le processus: une fonction traitante peut être automatiquement appelée dès réception d'un signal.
  - Sauf pour le signal numéro 9 qui entraine toujours la fin du processus.

Envoyer des signaux

- Depuis le shell: commande kill
  - `kill -`*sig* *pid*
  - *sig* peut être soit le nom d'un signal, soit son numéro.
  - `kill -9 myprog` ou `kill -KILL myprog`
- Liste des signaux:
  - 9: SIGKILL, fin du processus.
  - 10: SIGUSR1, définissable par l'utilisateur.
  - 18: SIGCONT, reprise d'un processus stoppé.
  - 19: SIGSTOP, stopper un processus.

Enovyer des signaux

- Depuis le clavier:
  - Ctrl-C.
    - Envoie un signal SIGINT au processus qui s'exécute.
    - Cela entraine par défaut la fin du processus.
  - Ctrl-Z:
    - Envoie un signal SIGTSTP au processus qui s'exécute.
    - Par défaut, l'exécution du processus est suspendu.

Envoyer des signaux en C

- `int kill(int pid,int signum);`
  - Envoie le signal de numéro `signum` au processus de pid `pid`.
- Traitement d'un signal en C:
  - La fonction `signal` permet de faire appeler automatiquement une fonction particulière lors de la réception d'un signal.
    `signal(signum, nomfonction);`
  - On peut aussi demander d'ignorer un signal en utilisant la constant `SIG_IGN`.
    `signal(signum, SIG_IGN);`

Exemple

- Affiche CTRL-C quand l'utilisateur tape CTRL-C au clavier.

```
#include <signal.h>
#include <stdio.h>
void traite_int(int signum)
{
  signal(signum,traite_int);
  printf("Ctrl-C\n");
}
int main()
{
  signal(SIGINT,traite_int);
  while(1) pause();
}
```