Université Côte d'Azur
Faculté des Sciences - Département d'Informatique

Licence d'Informatique L2
Introduction aux Systèmes et Réseaux

TD n ° 4 : Réalisation des processus – Signaux

1 Signaux

Un signal est un message asynchrone destiné à un processus pour l'informer d'une situation particulière. Un processus qui reçoit un signal réagit à ce signal en exécutant une action spécifiée, qui dépend de la nature du signal.

Un signal est analogue à une interruption ; mais alors qu'une interruption est destinée à un processeur physique, un signal est destiné à un processus. Certains signaux traduisent d'ailleurs directement des interruptions.

Un signal peut être envoyé par le système d'exploitation (par exemple pour indiquer une erreur), ou bien par un processus. Lorsqu'il reçoit un signal, un processus exécute une séquence de code qui a auparavant été spécifiée pour le signal en question. Les signaux les plus courants sont `signal.SIGINT` (frappe du caractère *control-C*), `signal.SIGSTOP` (signal de suspension d'un processus), `signal.SIGCONT` (continuation d'un processus suspendu), `signal.SIGKILL` (signal de terminaison), `signal.SIGCHLD` (fin ou suspension d'un processus fils signalée à son père). Voir `man 7 signal` pour une liste complète des signaux sous Unix.

Un signal est *pendant* tant qu'il n'a pas été pris en compte par le processus destinataire. Il ne peut exister qu'un signal pendant d'un type donné. Donc il est possible que des signaux soient perdus, par exemple si plusieurs signaux d'un certain type arrivent pendant le traitement d'un signal de ce même type.

La primitive [1] `os.kill(pid, sig)` permet d'envoyer le signal `sig` à des processus (si `pid` > 0 : au processus `pid` ; si `pid` $= 0$: aux processus du même groupe que `pid` ; si `pid` < -1 aux processus du groupe $|$`pid`$|$).

Tout signal a une action par défaut sur le processus destinataire (par exemple tuer le processus pour `signal.SIGKILL`, suspendre le processus pour `signal.SIGSTOP`, ne rien faire pour `signal.SIGCHLD`, etc.). Il est néanmoins possible de spécifier un comportement particulier lors de la réception d'un signal. Cela peut se faire grâce à une primitive appelée `signal` :

```
signal.signal(signum, handler)
```

qui associe la fonction `handler` au signal `signum`. Notons que la fonction `handler` a une forme bien spécifiée qui doit impérativement comporter deux paramètres : le premier est le numéro du signal et le second n'a pas d'intérêt pour nous pour le moment (mais doit absolument être présent) :

```
def handler(signum, ignore):
```

1. cette primitive est nommée `kill` pour des raisons historiques, mais son effet n'est pas nécessairement de tuer le processus destinataire.

Tous les comportements par défaut associés aux signaux peuvent ainsi être changés, sauf ceux des signaux `signal.SIGKILL` et `signal.SIGSTOP`. Par exemple, on peut spécifier un programme qui traite la frappe du caractère *control-C* (signal `signal.SIGINT`) :

```
import signal, sys

def handler(sig, ignore):
    print("Caught SIGINT")
    sys.exit(0)

signal.signal(signal.SIGINT, handler) # Met en place le nouveau traitant
signal.pause()  # attend la reception d'un signal
sys.exit(0)
```

1. La primitive `time.sleep(t)` suspend le processus appelant pendant `t` secondes ou jusqu'à l'arrivée d'un signal.
   Écrire un programme qui suspend le processus pendant `t` secondes passé en paramètre, et affiche le message "interruption" en cas d'interruption par *control-C*.
2. La primitive `signal.alarm(t)` provoque l'envoi d'un signal `signal.SIGALRM` au processus appelant au bout de `t` secondes.
   Écrire un programme qui ne fait rien (exécute une boucle vide) mais qui reçoit un signal `signal.SIGALRM` toutes les secondes, et affiche alors un message "bip". à la réception du sixième signal, le programme affiche "bye" et se termine.
3. On considère le programme suivant.

   ```
   import os, signal, sys, time

   counter = 0

   def handler(sig, ignore):
       global counter
       counter += 1
       time.sleep(0.1)  # do some work in the handler

   def parent():
       signal.signal(signal.SIGUSR2, handler)
       try:
           os.wait() # wait for the child to end
       except:
           pass # ignore l'exception si le fils est déjà mort
       print("counter= {}".format(counter))
       sys.exit(0)

   def child():
       for i in range(5):
           os.kill(os.getppid(), signal.SIGUSR2)
   ```

```
        print("sent SIGUSR2 to parent")
    sys.exit(0)

# do some work in the handler
if __name__ == "__main__":
    pid = os.fork()
    if pid == 0:
        child()
    else:
        parent()
```

Que fait ce programme ?

Quelle est à votre avis la valeur imprimée pour la variable `counter` ?

Voyez-vous un problème potentiel ?

4. Lorsqu'un processus crée des processus fils, il peut attendre leur fin avec la primitive `os.wait` ou `os.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`.

   Écrire le programme d'un processus créant plusieurs fils et traitant leur fin comme il vient d'être indiqué.

   Attention : tenir compte du fait que les signaux ne sont pas mémorisés (un seul signal d'un type donné peut être dans l'état pendant, cf question précédente).

2 Recouvrement en sursis

Écrire un script `atkill.py` (usage : `atkill.py nbsec sig command [arg ...]`) qui :

1. lance `command [arg ...]` et signale une éventuelle erreur lors du lancement,
2. provoque l'envoi du signal `sig` à `command` au bout de `nbsec` secondes puis attend sa terminaison.

3 Alarme

1. Écrire un programme Python qui recopie l'entrée standard sur la sortie standard, octet par octet. Toutes les 2 secondes, le programme incrémente un compteur et l'affiche ; il se termine au bout de 5 incrémentations.
2. Écrire un programme Python qui crée un fils. Le père recopie l'entrée standard sur la sortie standard, caractère par caractère. Le fils vérifie chaque seconde l'existence du père, et affiche un message et se termine lorsqu'il a détecté la fin du père.

   Note : la fonction `kill(pid, sig)` du module `os` appelée avec `sig=0` n'envoie pas de signal mais lève une `OSError` si le processus `pid` n'existe pas.