Travaux dirigés Système n°4 : signaux

Éric Gaudefroy

École Hexagone
Cursus Cyberdéfense - M1

Objectif

La bibliothèque Python *signal* permet d'interagir avec les evenements et de modifier les traitements de niveau OS. Par exemple, si nous pressons CTRL + C durant l'execution d'un processus, cela terminera son execution. C'est ce que l'on appelle un **événement**.
À partir de cet événement, l'OS génère un « *signal* » afin de nous indiquer ce qui s'est passé. Le module Python signal nous permet de définir nos propres réactions vis-à-vis des événements.

1. Introduction aux signaux sous Python

Voici les signaux utilisés sous Windows et Python 3.10. Sous Linux, comme pour macOS, il en existe encore plus, car Windows ne les supporte pas tous.
Nous pouvons utiliser le code suivant pour obtenir une liste des signaux disponibles sous notre OS. Chaque signal dispose d'un numéro d'identification (ID), qui peut être utilisé pour appeler le signal.

```
import signal

valid_signals = signal.valid_signals()
print(valid_signals)
```

Nous obtenons la sortie suivante :

PS D :\Hexagone\tests> py.exe .\signal1.py <Signals.SIGINT : 2>, <Signals.SIGILL : 4>, <Signals.SIGFPE : 8>, <Signals.SIGSEGV : 11>, <Signals.SIGTERM : 15>, <Signals.SIGBREAK : 21>, <Signals.SIGABRT : 22>

2. Utilisation des *handlers* de signal

Comme mentionné plus tôt, vous pouvez « attraper » un signal et en modifier le comportement par défaut. Nous allons donc modifier la réaction du système par rapport à l'événement CTRL+C pendant l'exécution de notre programme.
Afin d'« attraper » un signal comme SIGINT, nous devons utilsier la fonction Pyhton signal() du module du même nom.

```
signal.signal(signal.SIGINT, signal_handler)
```

Le premier paramètre est le signal que nous souhaitons modifier, le second paramètre est le nom de la fonction qui réalisera le nouveau comportement lié à la réalisation de l'événement associé a signal. Cette fonction renvoie l'ID du signal et les détails de la fenêtre mémoire associée.
Si nous activons SIGINT dans son état par défaut, le programme se termine. Changeons ce comportement en lui faisant faire juste un affichage.

```
import signal
import time

# Our signal handler
def signal_handler(signum, frame):
    print("Signal ID:", signum, " Frame: ", frame)

# Handling SIGINT
signal.signal(signal.SIGINT, signal_handler)

time.sleep(5)
```

Notre programme va mettre le processus en pause durant 5 secondes. Durant ce temps, vous pourrez renvoyer d'autres signaux SIGINT (CTRL+C) car le processus est asynchrone. L'affichage renvoie alors :

PS D :\Hexagone\tests> py.exe .\signal2.py
Signal ID : 2 Frame : <frame at 0x000002C348CBB160, file 'D :\\Hexagone\\tests\\signal2.py', line 11, code <module»
Il est important de noter que CTRL+C ne va pas terminer le programme car nous en avons modifié le comportement.

3. Traitements avancés des signaux

Il existe bien plus de fonctions disponibles dans la bibliothèque *signal*. Ils permettent de profondemment améliorer et modifier les signaux.
Nous pouvons par exemple préserver le comportement original de la fonction SIGINT. Peu importe si vous avez défini un nouveau comportement, il existe en effet une fonction retournée qui permet de rappeler le comportement original.
Nous pouvons utiliser cette capacité pour restaurer le comportement initial de SIGINT ou choisir d'appeler notre fonction spécifique en complément du comportement original.

```
import signal
import time

# Our signal handler
def signal_handler(signum, frame):
    global original_handler
    print("Signal ID:", signum, " Frame: ", frame)
    original_handler(signum, frame)

original_handler = signal.signal(signal.SIGINT, signal_handler)
```

```
print("Waiting for 5 seconds...")
time.sleep(5)
```

En exécutant ce code et en tapant CTRL+C, nous obtenons 2 effets : celui de la fonction spécifique que nous avons définie, puis le comportement normal de SIGINT, soit la terminaison du processus.

Une autre fonction intéressante est l'utilisation de *alarm*(), qui génère le signal SIGALRM. Nous pouvons l'associer à un timer qui, après un certain intervalle, ici 5 secondes, va activer l'événement. Cela offre de nombreuses possibilités pour organiser les événements et fonctions associées.

```
import signal
import time

def signal_handler(signum, stack):
    print('Alarm: ', time.ctime())

signal.signal(signal.SIGALRM, signal_handler)
signal.alarm(5)
time.sleep(10)
```

L'appel à time.sleep() est nécessaire, sinon le programme pourrait s'arréter avant que l'action de SIGALRM soit terminée.

Nota : hélas, SIGALRM n'existe pas sous Windows...

Hexagone 2023