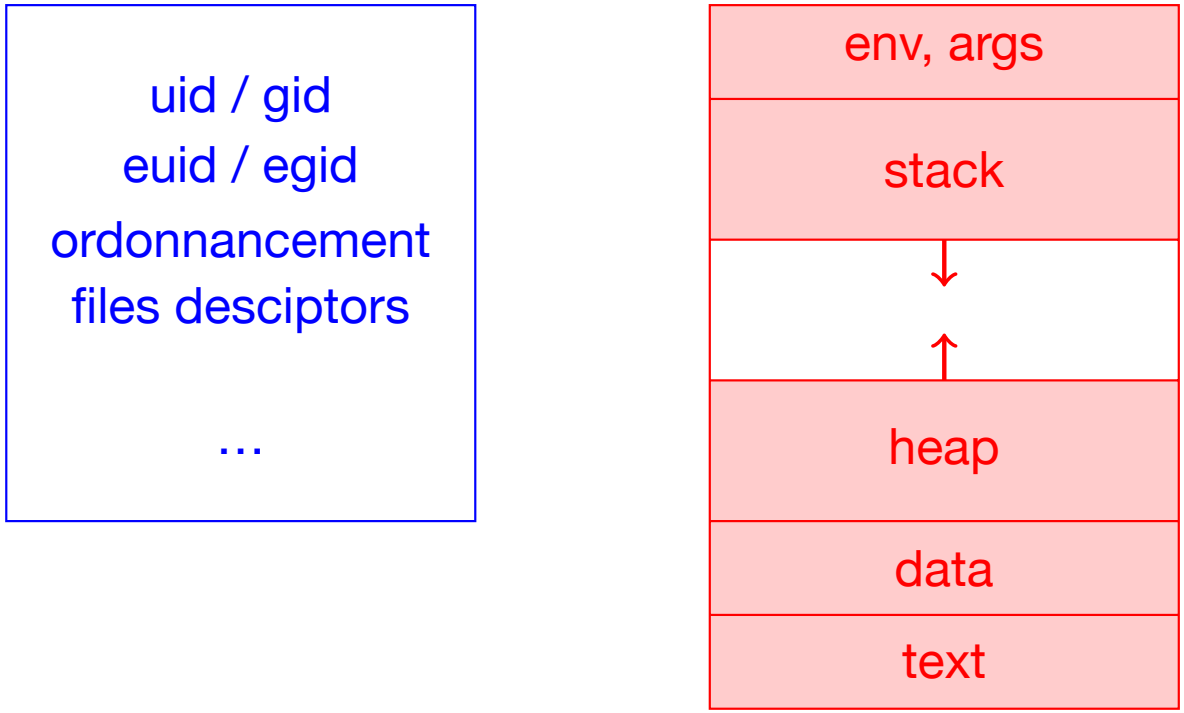
06 - Signaux

Gaétan Richard
gaetan.richard@info.unicaen.fr

L2 – S4 2011/2012

1. Présentation

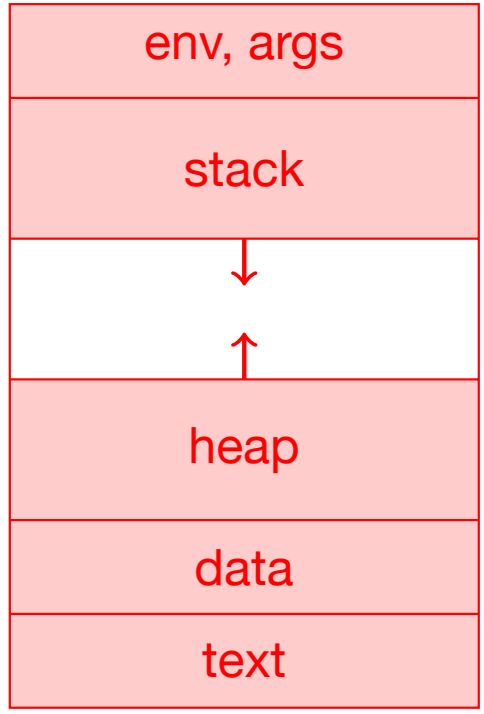
Processus



Question : Comment interagir avec un processus ?

Processus

uid / gid
euid / egid
ordonnancement
files desciptors
signaux
...



Question : Comment interagir avec un processus ?

Avec des **signaux**

Signaux : objectif

Les signaux sont des mécanismes permettant de signaler des évènements à un processus.

Ces évènements peuvent être :

- des erreurs ;
- le déclenchement d'une minuterie ;
- des interactions d'un autre processus ou de l'utilisateur.

Ces notifications se font de façon **asynchrone**.

Signaux : mise en œuvre

Principe de base :

- Les signaux sont représentés par des entiers ;
- Il existe une liste prédéfinie de signification pour les signaux ;
- la réception d'un signal interrompt le déroulement normal du processus.

Un signal peut :

- être ignoré ;
- terminer le processus (avec ou sans "core dump") ;
- arrêter le processus ;
- déclencher une fonction de traitement.

Le comportement peut (sauf indication contraire) être modifié.

Signaux POSIX

n°	nom	action	descr
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
3	SIGQUIT	create core image	quit program
4	SIGILL	create core image	illegal instruction
5	SIGTRAP	create core image	trace trap
6	SIGABRT	create core image	abort program (formerly SIGIOT)
7	SIGEMT	create core image	emulate instruction executed
8	SIGFPE	create core image	floating-point exception
9	SIGKILL	terminate process	kill program
10	SIGBUS	create core image	bus error
11	SIGSEGV	create core image	segmentation violation
12	SIGSYS	create core image	non-existent system call invoked
13	SIGPIPE	terminate process	write on a pipe with no reader
14	SIGALRM	terminate process	real-time timer expired
15	SIGTERM	terminate process	software termination signal
16	SIGURG	discard signal	urgent condition present on socket
17	SIGSTOP	stop process	stop (cannot be caught or ignored)
18	SIGTSTP	stop process	stop signal generated from keyboard
19	SIGCONT	discard signal	continue after stop
20	SIGCHLD	discard signal	child status has changed
21	SIGTTIN	stop process	background read attempted from control terminal
22	SIGTTOU	stop process	background write attempted to control terminal
23	SIGIO	discard signal	I/O is possible on a descriptor (see fcntl(2))
24	SIGXCPU	terminate process	cpu time limit exceeded (see setrlimit(2))
25	SIGXFSZ	terminate process	file size limit exceeded (see setrlimit(2))
26	SIGVTALRM	terminate process	virtual time alarm (see setitimer(2))
27	SIGPROF	terminate process	profiling timer alarm (see setitimer(2))
28	SIGWINCH	discard signal	Window size change
29	SIGINFO	discard signal	status request from keyboard
30	SIGUSR1	terminate process	User defined signal 1
31	SIGUSR2	terminate process	User defined signal 2

Quelques remarques sur les signaux

- Le “core dump” peut être empêché par l’existence de la limite correspondante.
- Dans certains systèmes d’exploitation, un signal peut interrompre un appel système. Dans ce cas, il est possible que l’appel interrompu échoue.
- L’utilisation de la combinaison threads + signaux est à utiliser avec précaution.

2. Gestion des signaux

Principe de gestion

Chaque processus dispose d'une table contenant un gestionnaire (handler). On modifie ces éléments par l'intermédiaire d'appels systèmes dédiés.

L'appel d'un gestionnaire de signal interrompt l'exécution normale d'un processus, il faut être extrêmement prudent dans le code exécuté pour éviter les "effets de bord".

Un gestionnaire de signal peut être interrompu par un signal !

Trap

Commande :

```
trap [COMMANDS] [SIGNALS]
```

Exemple :

```
#!/bin/sh

trap "echo Signal!" SIGINT SIGTERM

while true
do
  sleep 10
done
```

Utilisation courante

En shell, on utilise couramment les gestionnaires de signaux afin de s'assurer au maximum que le script effectuera bien certaines opérations avant de se terminer (en général du "nettoyage").

Exemple :

```
trap "{ rm -f $LOCKFILE; exit 255; }" EXIT
```

signal

Toutes les fonctions pour manipuler des signaux sont regroupés dans le module **signal**.

signal.signal(*signalnum*, *handler*) définit le handler pour un signal donné.

signal.getsignal(*signalnum*) permet de récupérer le handler courant et **signal.SIG_DFL** et **signal.SIG_IGN** désignent respectivement le handler python par défaut et celui ignorant un signal.

Exemple d'utilisation

```
import signal, os

def handler(signum, frame) :
    print 'Signal handler called with signal', signum
    raise IOError("Couldn't open device!")

# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)          # Disable the alarm
```

En C

En C, il existe également un moyen de masquer les signaux durant une phase d'exécution critique. Ceci se fait par l'intermédiaire de l'appel système :
int **sigprocmask**(int how, const sigset_t *restrict set, sigset_t *restrict oset) ;

Ceci n'est pas encore disponible en python car certains systèmes d'exploitation ne le supportent pas.

3. Présentation des signaux

En cas d'erreur

Exemples :

SIGBUS	create core image	bus error
SIGFPE	create core image	floating-point exception
SIGILL	create core image	illegal instruction
SIGSEGV	create core image	segmentation violation
SIGSYS	create core image	non-existent system call invoked

Comportement par défaut : Terminaison du programme avec "core dump". Quasi impossible à récupérer.

Minuteries

Exemples :

SIGALRM	terminate process	real-time timer expired
SIGVTALRM	terminate process	virtual time alarm
SIGPROF	terminate process	profiling timer alarm

Utilisation

Il est possible de déclencher l'alarme à l'aide de l'appel système :
signal.alarm(*time*)

Note : si *time* vaut zéro, l'alarme est annulée.

signal.pause() sert à attendre l'arrivée d'un signal.

Interactions

Exemples :

SIGABRT	create core image	abort program
SIGCONT	discard signal	continue after stop
SIGHUP	terminate process	terminal line hangup
SIGINT	terminate process	interrupt program
SIGKILL	terminate process	kill program (9)
SIGQUIT	create core image	quit program
SIGSTOP	stop process	stop
SIGTERM	terminate process	software termination signal
SIGUSR1	terminate process	User defined signal 1
SIGUSR2	terminate process	User defined signal 2

Raccourcis

Il est possible d'envoyer certains de ces signaux à l'aide des combinaison de touches suivantes :

- **ctrl+c** (noté ^c) pour envoyer SIGINT
- **ctrl+z** pour envoyer SIGSTOP

SIGSTOP permet d'arrêter un processus, il est ensuite possible de le redémarrer (envoyer SIGCONT) à l'aide de la commande **bg** ("background") ou de le ramener au premier plan à l'aide de la commande **fg** ("foreground").

SIGHUP

Pour les processus exécutés depuis un terminal, le signal **SIGHUP** est envoyé lorsque le terminal est fermé. C'est également ce signal qui est envoyé aux processus lors de la fermeture de la session.

Il est possible de lancer un programme ignorant ce signal à l'aide de la commande **nohup**. (Note : cette commande redirige également la sortie dans un fichier)

Job control

Chaque commande du shell en cours d'exécution possède un identifiant de travail (**job id**).

Il est possible d'utiliser ce numéro à l'aide de la syntaxe **%*n***. La commande courante est désignée par **%%**.

Il est également possible d'utiliser ces job ids dans les commandes **fg**, **bg**, …

Autres interactions

Il est possible d'envoyer n'importe quel signal à un processus désiré à l'aide de la commande **kill *-signal pid***.

Il est possible de préciser le signal de façon numérique (**-9**) ou nominative (**-KILL**).

La valeur **-1** comme pid indique tous les processus dont l'utilisateur est propriétaire.

En python

Il existe l'équivalent en python :

os.kill(*pid*, *sig*)

4. Communication entre processus

Rappel

Les processus disposent de leur propre mémoire.

Il est possible de communiquer à l'aide de signaux.

Dans la plupart des cas, on souhaite communiquer entre un père et son fils.

Tubes

Pour communiquer des données entre processus, il est possible d'utiliser des **tubes**.

Il existe des :

- tubes nommés (**fifo**)
- tubes anonymes (**pipe**)

Utilisation

On peut créer des tubes à l'aide des commandes :

- **mkfifo** en shell et **os.mkfifo(*path[, mode]*)** en python ;
- **|** en shell et **os.pipe()** en python.

Un tube est soit en écriture seule, soit en lecture seule. Il a une capacité limité. La lecture d'un tube vide et l'écriture dans un tube plein peuvent provoquer soit un blocage, soit une erreur.

Les primitives d'utilisation du tube sont les mêmes que celle d'un fichier.

Mémoire partagée

Pour utiliser la mémoire partagée, POSIX dispose de **shm**. Il n'existe pas de bibliothèque python standard pour ces fonctionnalités.

Il existe également **nmap** qui permet de mapper des fichiers en mémoire. En utilisant les bonnes options, il est possible de partager cette mémoire entre les différents processus. Ces fonctionnalités sont disponibles dans le modules python **nmap**.

À haut niveau, il est également possible de partager des objets python.