Programmation Unix 2 – cours n°7

Edouard THIEL

Faculté des Sciences

Université d'Aix-Marseille (AMU)

Janvier 2017

Les transparents de ce cours sont téléchargeables ici :
http://pageperso.lif.univ-mrs.fr/~edouard.thiel/ens/unix/

Lien court : http://j.mp/progunix

Plan du cours n°7

1. Variables d'environnement
2. Les signaux Unix
3. Envoi de signaux
4. Signaux usuels
5. Capter un signal en bash

1 - Variables d'environnement

Rappel : en `bash`

`export` permet de créer les variables d'environnement :

```
$ export foo="ga bu"
$ declare -p foo
declare -x foo="ga bu"
```

Elles sont ensuite

- recopiées dans les sous-shells ;
- transmises dans les commandes.

Recouvrement

Appel de `main` par le système :

```
_start() {
  int argc; char **argv, **environ;
  // Réception de argc, argv, environ

  int status = main (argc, argv, environ);
  exit (status);
}
```

Réception des variables d'environnement par `main` :

```
int main (int argc, char *argv[], char *envp[])
```

(non POSIX.1) ou encore

```
int main (int argc, char *argv[]) {
  extern char **environ;
```

Format de envp

envp est une liste terminée par NULL de chaînes de la forme
`NOM=valeur`

• Lorsqu'on recouvre un processus, on peut lui changer entièrement ses variables d'environnement :

```
char **envp = {"USER=korben", "SHELL=/bin/bash",
               "HOME=/home/dallas", "PWD=/bin", NULL};
execle ("ls", "ls", "/", NULL, envp);
```

envp est ensuite reçu par `_start` puis transmis à `main`.

• Lorsqu'on duplique un processus, il reçoit une copie de envp.

Accès direct

```
#include <stdlib.h>
char *getenv(const char *name);
```

Renvoie la chaîne correspondant à la variable d'environnement `name` (cherchée dans `environ`), sinon `NULL`.

Exemple : `char *home = getenv("HOME");`

```
#include <stdlib.h>
int putenv(char *string);
```

Modifie ou rajoute la variable d'environnement `NOM=VALEUR` dans `environ`. Renvoie 0 pour succès.

Exemple : `putenv("HOME=/home/dallas");`

Recouvrement en changeant l'environnement

```
int execvpe(const char *file, char *const argv[],
            char * const envp[]);
int execle(const char *path, const char *arg,
           ..., char * const envp[]);
```

Ces fonctions permettent de plus de donner de nouvelles variables d'environnement, dont la liste est dans `envp`.

2 - Les signaux Unix

Forme limitée de communication inter-processus (IPC) :

- notification asynchrone d'un évènement,
- envoyée à un processus ou à un groupe de processus.
- Seule information envoyée : le nom du signal.

Lorsqu'un processus reçoit un signal :

- son flot d'exécution est interrompu par le système ;
- puis un *handler* de signal est appelé ;
- enfin le flot d'exécution est éventuellement repris.

Un signal = une interruption logicielle.

handler de signal = fonction appelée pendant l'interruption.

Vu en TP

- `^C` provoque l'envoi du signal `SIGINT` au processus en avant-plan, ce qui l'INTerrompt.
- `^Z` provoque l'envoi du signal `SIGTSTP` au processus en avant plan, ce qui Temporairement le SToPe.
- `fg` ou `bg` provoque l'envoi du signal `SIGCONT` ce qui réveille (CONTinue) le processus.
- `kill pid` envoie le signal `SIGTERM` au processus `pid`, ce qui le TERMine.
- `kill -9 pid` ou `kill -KILL pid` envoie le signal `SIGKILL` au processus `pid`, ce qui le tue.
- Une violation d'adresse mémoire (sortie de tableau) provoque l'envoi de `SIGSEGV` (SEGment Violation) $\rightarrow$ core dump.
- Une division par zero provoque l'envoi de `SIGFPE` (Floating Point Exception) $\rightarrow$ core dump.

Bilan

Des signaux Unix peuvent être envoyés par

- le terminal (`^C` ou `^Z`)
- le shell (`fg` ou `bg`)
- l'utilisateur avec la commande `kill`
- le système (erreurs matérielles ou logicielles)
- etc

À réception d'un signal, un processus peut

- se terminer
- ignorer le signal
- être tué, endormi ou réveillé
- générer une image mémoire (core dump)
- capter le signal et décider quoi faire.

Portabilité

Différentes implémentations :

- BSD (implémentation originale, années 1970)
- SysV (début années 1980)
- POSIX.1 (normalisations 1990, 2001)

→ Différents signaux et comportements.

Chaque système implémente :

- les signaux standard POSIX
- éventuellement des signaux temps réel
- des signaux particuliers au système

→ portabilité limitée.

Identification des signaux

Chaque signal a un nom et un numéro.

Définis dans `signal.h`, voir `man 7 signal` :

`NSIG` nombre de signaux admis sur le système, numérotés `1 .. NSIG-1`

`SIG`*name* constante désignant le signal *name* :

```
SIGHUP  SIGINT  SIGQUIT  ...
  1       2       3
```

Connaître le numéro d'un signal :

```
<> kill -l TERM        # ou SIGTERM
15
<> kill -l 15
TERM
```

Afficher les signaux

```
<> kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS
34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7
42) SIGRTMIN+8  43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11
46) SIGRTMIN+12 47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15
50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11
54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3
62) SIGRTMAX-2  63) SIGRTMAX-1  64) SIGRTMAX
```

3 - Envoi de signaux

Permission :

un processus A peut envoyer un signal à un processus B si

- ils ont le même user
- ou si A est privilégié.

Envoi de signaux en bash

- `kill -`*num* *pid*
  `kill -`*name* *pid*
  `kill -SIG`*name* *pid*

  Envoie le signal au processus *pid*.

  ```
  <> kill -INT 1234
  ```

- Si on ne précise pas le signal, `SIGTERM` est envoyé par défaut.

  ```
  <> kill 1234
  ```

- Si le pid est `-1`, le signal est envoyé à tous les processus permis (sauf `init` et lui-même).

  ```
  <> kill -9 -1
  ```

Envoi de signaux avec `killall`

- Pour connaître les instances d'un programme :

```
<> pidof bash
3779 3759 3705 3639 3630 3575 2182
```

On peut donc envoyer un signal à toutes les instances :

```
<> kill -QUIT $(pidof bash)
```

- La commande `killall` le fait directement :

```
<> killall -QUIT bash
```

`SIGTERM` est envoyé par défaut :

```
<> killall bash
```

- On peut utiliser des regexp :

```
<> killall -QUIT -r 'gnome-(calc|sys).*or'
```

Envoi de signaux en C

- ```
  #include <signal.h>
  int kill (pid_t pid, int sig);
  ```

Envoie le signal `sig` au processus `pid`
(si `-1` : à tous les processus permis sauf `init` et lui-même).

Renvoie 0 succès, -1 erreur.

Exemple :

```
if (kill (1234, SIGHUP) < 0) perror ("kill");
```

- ```
  #include <signal.h>
  int raise (int sig);
  ```

Envoie le signal `sig` au processus appelant ;
équivalent à `kill (getpid(), sig)`

Tester l'existence d'un processus

`kill -0 pid` (en bash)
ou `kill (pid, 0)` (en C)

n'envoit pas de signal,

mais réussit / renvoie 0 si le processus `pid` existe
sinon échoue / renvoie -1.

4 - Signaux usuels

- `SIGHUP` : Hang UP

adressé à tous les processus d'une session lorsque le processus leader d'une session se termine.

Effet : terminaison

Protéger une commande : `nohup commande`

- `SIGINT` :

adressé à tous les processus d'un groupe en premier plan sous le contrôle d'un terminal lorsque frappe `^C`.

Effet : terminaison

Signaux avec image mémoire

- `SIGSEGV` : SEGment Violation

Violation de page mémoire

Effet : terminaison + core dump

- `SIGFPE` : Floating Point Exception

Division par zéro

Effet : terminaison + core dump

Signaux courants

- `SIGUSR1`, `SIGUSR2`

Sans signification absolue

Effet : terminaison

- `SIGCHLD` : fin d'un CHiLD

Envoyé au père lorsqu'un fils se termine

Effet : aucun (signale un zombie à lever)

Signal retardé

- `SIGALRM` : ALaRMe

Signal émis à un processus à sa propre demande, au bout d'un délai fixé par :

```
#include <unistd.h>
unsigned int alarm (unsigned int seconds);
```

Chaque nouvel appel annule et remplace le précédent.

Si `seconds` est 0, le décompte est annulé.

Renvoie le nombre de secondes restant avant la fin prévue du précédent appel, sinon 0.

Effet de `SIGALRM` : termine le processus.

Ne pas utiliser en même temps `alarm` et `sleep`

5 - Capter un signal en bash

```
trap com liste_signaux
```

Demande à bash de capter ces signaux ;

pour chaque signal capté, bash exécutera `com`.

Exemple :

```
<> trap 'echo "signal reçu"' TERM QUIT
<> kill -QUIT $$
signal reçu
```

De plus :

`trap - liste_signaux`	rétablit défaut
`trap '' liste_signaux`	pour ignorer

Exemple

```
$ cat alive.sh
#! /bin/bash

capter_INT ()
{
    echo "  Ouch !"
}

trap capter_INT SIGINT

while true ; do
    sleep 1
    echo "Alive !"
done
```

```
$ ./alive.sh
Alive !
Alive !
^C  Ouch !
Alive !
Alive !
^C  Ouch !
Alive !
Alive !
^Z
[1]+  Arrêté  ./alive.sh
$ kill %%
$
[1]+  Complété  ./alive.sh
$
```

Avant-terminaison avec `trap`

```
trap com 0
```

La commande `com` sera exécutée avant la terminaison du script.

```
<> cat essai.sh
#! /bin/bash
trap 'echo "atexit ..."' 0
echo "Running ..."
exit 0

<> ./essai.sh
Running ...
atexit ...
```

Déboguage avec `trap`

```
trap com DEBUG
```

la commande `com` est exécutée avant chaque commande.

Désactiver : `trap '' DEBUG`

```
<> trap 'echo "# $BASH_COMMAND"' DEBUG
<> true && echo "ok"
# true
# echo "ok"
ok
```

Autres possibilités : `help trap`