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

Licence d'Informatique L2
Introduction aux Systèmes et Réseaux

TP n ° 2 : Processus: Signaux et terminaison

1 Traitement des signaux d'interruption

1. On rappelle que 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 un nombre de secondes passé en paramètre.
2. Écrire un programme qui lorsqu'il reçoit le signal `signal.SIGINT` affiche le numéro de ce signal, et ne fait rien le reste du temps (on rappelle que la primitive `signal.pause()` permet à un processus de se bloquer en attendant l'arrivée d'un signal). Faire de même avec `signal.SIGKILL` (primitive `os.kill(pid, sig)`) et `signal.SIGSTOP` (*control-Z*). Que constatez-vous ?
3. Exécuter le programme de la question 1.3 du TD4. En ajoutant un `time.sleep()` dans le programme des fils (après l'instruction `os.kill()`), voir que l'on peut obtenir différentes valeurs (entre 1 et 5) pour la valeur finale de `counter`.

2 Traitement de la terminaison d'un fils

1. Tester le programme `waitpid1.py` présenté au début du TD2 qui attend la mort de ses fils et affiche leur résultat. Le voici reproduit ici :

```
import errno, os, sys

nbChildren = 20
for i in range(nbChildren):
    pid = os.fork()
    if pid == 0:  # child
        sys.exit(100 + i)
try: # parent waits for all of its children to terminate
    while True:
        pid, status = os.waitpid(-1, 0)
        if os.WIFEXITED(status):
            print("child {} terminated normally with exit status={}".\
            format(pid, os.WEXITSTATUS(status)))
        else:
            print("child {} terminated abnormally".format(pid))
```

```
except OSError as e:
    print("waitpid error: {}, {}".format(\
            errno.errorcode[e.errno], os.strerror(e.errno)), file=sys.stderr)
    if e.errno == errno.ECHILD:
        print("No more children left. Bye", file=sys.stderr)
sys.exit(0)
```

2. Reprendre le programme `waitpid1.py` et modifier ce programme pour qu'il ait l'effet suivant :
   — chaque processus fils s'endort 20 secondes avant de se terminer normalement par `sys.exit`.
   — après la création de chaque fils, le père affiche le numéro de pid du fils créé
   — le père indique la fin anormale des fils en imprimant le numéro de chaque fils et la cause de sa terminaison (numéro de signal).
   — tester ce qui se passe quand un processus fils est tué par la commande `kill` depuis une nouvelle fenêtre shell, avant qu'il ne se réveille.

   On rappelle que `os.waitpid` renvoie un tuple `(pid,statut)` et que l'on peut tester le `statut` pour connaître l'état du processus qui s'est terminé. En particulier :
   — `os.WIFSIGNALED(statut)` renvoie vrai si la fin du processus est due à un signal non traité.
   — si `os.WIFSIGNALED(statut)` renvoie vrai, alors `os.WTERMSIG(statut)` renvoie le numéro du signal qui a causé la terminaison du processus.

3. 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 `signal.SIGCHLD`.
   Dans le programme `signal1.py`, le père crée dix fils et traite le signal `signal.SIGCHLD` envoyé lors de leur terminaison. Puis le père lit quelque chose au clavier et se met en boucle. Le voici reproduit ici :

```
import os, signal, sys

def CHLD_handler(signal, ignore):
    """Traitant de l'interruption SIGCHLD"""
    pid, status = os.waitpid(-1, 0)
    if os.WIFEXITED(status):
        print("**père** child %d terminated normally with exit status=%d" %\
            (pid, os.WEXITSTATUS(status)))
    else:
        print("**père** child %d terminated abnormally" % pid)

# Programme principal
if __name__ == '__main__':
    # Installe le traitant
    signal.signal(signal.SIGCHLD, CHLD_handler)
    # Création des fils
```

```
for i in range(10):
    pid = os.fork()
    if pid == 0:
        print("------> fils (pid = %d). Je me termine." % os.getpid())
        sys.exit(1 + i)
    else:
        print("**père** fils créé, pid = %d" % pid)

# À tout moment l'exécution peut être interrompue par la réception
# d'un signal SIGCHLD. Si le programme est en train de faire un appel
# système (par exemple pour attendre une lecture clavier), cette
# interruption provoque la levée d'une exception.
# Problème: si on est interrompu pendant qu'on attend une saisie
# clavier, cette saisie est interrompue.
# => Il faut donc recommencer la saisie en cas d'exception.
# Astuce: Pour savoir si la saisie a été interrompue, et donc savoir
# s'il faut recommencer (avec une boucle while) il suffit de regarder
# si la variable de saisie existe.
while not 'saisie' in globals():
    try:
        saisie = input("**père** tapez quelque chose... ")
    except:
        # La réception d'un signal provoque l'interruption de la saisie
        pass
print("**père** Terminé !")
sys.exit(0)
```

Exécuter ce programme, attendre que le père se mette en attente d'une saisie, le suspendre par *control-Z* et vérifier, par `ps -x`, que tous les fils n'ont pas été collectés (il reste des zombies). Cela résulte du fait que les signaux ne sont pas mémorisés (un seul signal d'un type donné peut être dans l'état pendant). Modifier le programme pour corriger cette erreur.

3 Comptage de signaux

Reprendre l'exercice 1 du TD5. Tester avec $n = 10,\ 1\,000,\ 100\,000$. Que peut-on conclure ?