Programmation Systeme en C
(retour sur les exemples vus avec Python)

Olivier Dalle
Université de Nice - Sophia Antipolis
http://deptinfo.unice.fr/
D'après le cours original de
Sacha Krakowiak
Université Joseph Fourier
Projet Sardes (INRIA et IMAG-LSR)

http://sardes.inrialpes.fr/~krakowia

Exemple d'usage des interfaces (Unix)

Interface programmatique (en C)

le morceau de programme ci-contre utilise les fonctions `read()` et `write()` pour recopier un fichier dans un autre

```
...
while (bytesread = read(from_fd, buf, BLKSIZE )) {
    if ((bytesread == -1) && (errno != EINTR))
      break;

    else if (bytesread > 0) {
      bp = buf;
      while(byteswritten = write(to_fd, bp, bytesread )) {
        if ((byteswritten == -1) && (errno != EINTR))
          break;
        else if (byteswritten == bytesread)
          break;
        else if (byteswritten > 0) {
          bp += byteswritten;
          bytesread -= byteswritten;
        }
      }
      if (byteswritten == -1)
        break;
    }
  }
...
```

Interface de commande

```
cp fich1 fich2
```

recopie `fich1` dans `fich2`

Documentation

Documentation en ligne par `man`

`man 1` <nom de la commande> : documentation des commandes (option par défaut)
`man 2` <nom de la commande> : documentation des appels système
`man 3` <nom de la commande> : documentation de la bibliothèque C

Retour sur les différences entre C et Python

Combinaison test+affectation
N'existe pas en python

Langage C:

```
...
while (bytesread = read(from_fd, buf, BLKSIZE )) {
    if ((bytesread == -1) && (errno != EINTR))
      break;
    else if (bytesread > 0) {
      bp = buf;
      while(byteswritten = write(to_fd, bp, bytesread )) {
        if ((byteswritten == -1) && (errno != EINTR))
          break;
        else if (byteswritten == bytesread)
          break;
        else if (byteswritten > 0) {
          bp += byteswritten;
          bytesread -= byteswritten;
        }
      }
      if (byteswritten == -1)
        break;
    }
  }
...
```

Test+affectation en C:

```
while (y = f(x)) {
   <calcul sur y ...>
   }
```

Equivalent Python:

```
y=f(x)
while (y):
   <calcul sur y ...>
   y=f(x)
```

Retour sur les différences entre C et Python

Traitement des erreurs au cas par cas

Langage C:

```
...
while (bytesread = read(from_fd, buf, BLKSIZE )) {
    if ((bytesread == -1) && (errno != EINTR))
        break;
    else if (bytesread > 0) {
        bp = buf;
        while(byteswritten = write(to_fd, bp, bytesread )) {
            if ((byteswritten == -1) && (errno != EINTR))
                break;
            else if (byteswritten == bytesread)
                break;
            else if (byteswritten > 0) {
                bp += byteswritten;
                bytesread -= byteswritten;
            }
        }
        if (byteswritten == -1)
            break;
    }
}
...
```

Traitement des erreurs en C:

```
y = f1(x);
if (y == -1) {
    fprintf(stderr, "I/O error(%d): %s », errno, strerror(errno));
}
<traitement si f1() ok...>
y = f2(x);
if (y == -1) {
    fprintf(stderr, "I/O error(%d): %s », errno, strerror(errno));
}
<traitement si f2() ok...>
...
```

Equivalent Python:

```
try:
    y = f1(x)
    <traitement si f1() ok...>
    y = f2(x)
    <traitement si f2() ok...>
except OSError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
```

Retour sur les différences entre C et Python

La mémoire tampon doit etre créée à l'avance

Langage C:

```
...
while (bytesread = read(from_fd, buf, BLKSIZE )) {
    if ((bytesread == -1) && (errno != EINTR))
      break;
    else if (bytesread > 0) {
      bp = buf;
      while(byteswritten = write(to_fd, bp, bytesread )) {
        if ((byteswritten == -1) && (errno != EINTR))
          break;
        else if (byteswritten == bytesread)
          break;
        else if (byteswritten > 0) {
          bp += byteswritten;
          bytesread -= byteswritten;
        }
      }
      if (byteswritten == -1)
        break;
    }
  }
...
```

Gestion de la mémoire en C

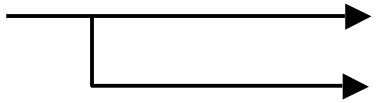
```
char *buf = malloc(n);
bytesread = read(fd,buf,n);
<travail sur buf ...>
free(buf);
```

Equivalent Python:

```
buf = read(fd,n)
<travail sur buf...>
```

Création des processus dans Unix (1)

- L'appel système `pid_t fork()` permet de créer un processus
- Le processus créé (fils) est un clone (copie conforme) du processus créateur (père)
- Le père et le fils ne se distinguent que par le résultat rendu par `fork()`
  - pour le père : le numéro du fils (ou –1 si création impossible)
  - pour le fils : 0



1 programme, 2 processus

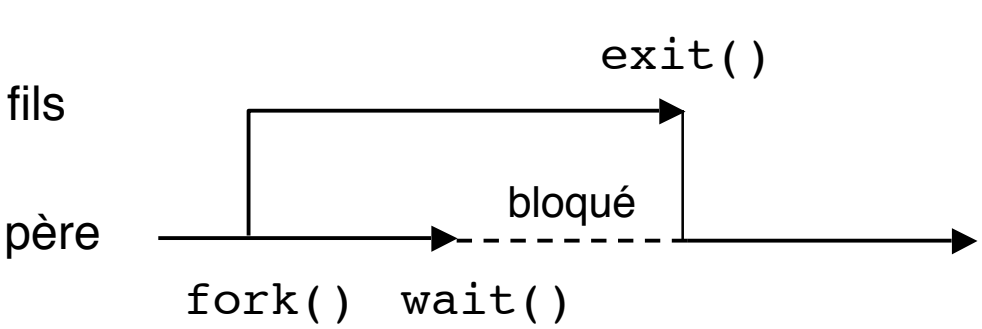
```
if (fork() != 0) {
    printf("je suis le père, mon PID est %d\n", getpid());
  } else {
    printf("je suis le fils, mon PID est %d\n", getpid());
    /* en général exec (exécution d'un nouveau programme) */
  }
   ...
}
```

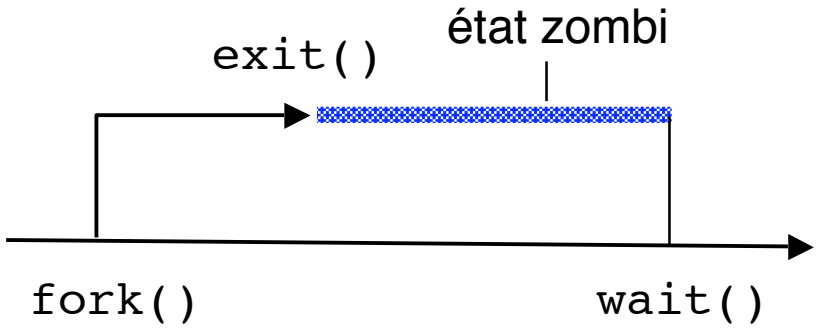
```
je suis le fils, mon PID est 10271
je suis le père, mon PID est 10270
```

Synchronisation entre père et fils

Quand un processus se termine, il délivre un code de retour (paramètre de la primitive `exit()`). Par exemple `exit(1)` renvoie le code de retour 1.

Un processus père peut attendre la fin d'un ou plusieurs fils en utilisant `wait()` ou `waitpid()`. Tant que son père n'a pas pris connaissance de sa terminaison par l'une de ces primitives, un processus terminé reste dans un état dit **zombi**. Un processus zombi ne peut plus s'exécuter, mais consomme encore des ressources (tables). Il faut éviter de conserver des processus dans cet état.





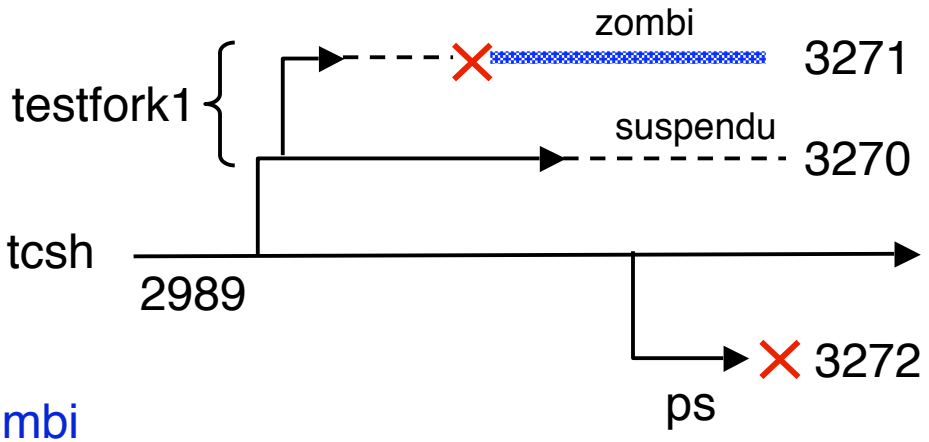
programme
testfork1.c

```
int main() {
  if (fork() != 0) {
    printf("je suis le père, mon PID est %d\n", getpid());
    while (1) ; /* boucle sans fin sans attendre le fils */
  else {
    printf("je suis le fils, mon PID est %d\n", getpid());
    sleep(2)  /* blocage pendant 2 secondes */
    printf("fin du fils\n");
    exit(0);
  }
}
```

```
<unix> gcc -o testfork1 testfork.c
<unix> ./testfork1
je suis le fils, mon PID est 3271
je suis le père, mon PID est 3270
fin du fils
==>frappe de <control-Z> (suspendre)
Suspended
<unix> ps
 PID TTY          TIME CMD
2989 pts/0    00:00:00 tcsh
3270 pts/0    00:00:03 testfork1
3271 pts/0    00:00:00 testfork1 <defunct>
3272 pts/0    00:00:00 ps
<unix>
```

zombi

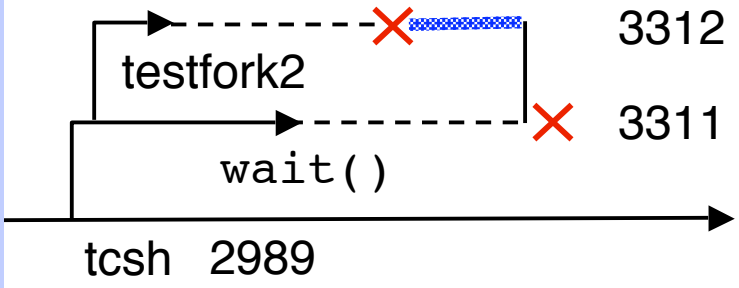
Exemple simple d'exécution



programme
testfork2.c

```
#include <sys/types.h>
#include <sys.wait.h>
int main() {
  if (fork() != 0) {
    int statut; pid_t fils);
    printf("je suis le père %d, j'attends mon fils\n", getpid());
    fils = wait(&statut);
    if (WIFEXITED(statut)) {
        printf("%d : mon fils %d s'est terminé avec le code %d\n",
            getpid(), fils, WEXITSTATUS(statut); };
    exit(0);
  } else {
    printf("je suis le fils, mon PID est %d\n", getpid());
    sleep(2)  /* blocage pendant 2 secondes */
    printf("fin du fils\n");
    exit(1);
```

```
<unix> ./testfork2
je suis le fils, mon PID est 3312
je suis le père 3311, j'attends mon fils
fin du fils
3311: mon fils 3312 s'est terminé avec le code 1
<unix> ps
 PID TTY          TIME CMD
2989 pts/0    00:00:00 tcsh
3313 pts/0    00:00:00 ps
<unix>
```



Exemple simple d'exécution

Exécution d'un programme spécifié

La primitive `exec` sert à faire exécuter un nouveau programme par un processus. Elle est souvent utilisée immédiatement après la création d'un processus. Son effet est de "recouvrir" la mémoire virtuelle du processus par le nouveau programme et de lancer celui-ci en lui passant des paramètres spécifiés dans la commande.

Diverses variantes d'`exec` existent selon le mode de passage des paramètres (tableau, liste, passage de variables d'environnement). Exemple :

```
main() {
  if (fork() == 0) {
     execl("bin/ls", "ls", "-a", 0);
  }
  else {
     wait(NULL);}
  exit(0)
}
```

le fils exécute :

`/bin/ls  -a  ..`

le père attend la fin du fils

Envoi d'un signal

Un processus peut envoyer un signal à un autre processus. Pour cela, il utilise la primitive `kill` (appelée ainsi pour des raisons historiques ; un signal ne tue pas forcément son destinataire).

Utilisation : `kill(pid_t pid, int sig)`

Effet : Soit `p` le numéro du processus émetteur du signal
Le signal de numéro `sig` est envoyé au(x) processus désigné(s) par `pid` :

- si `pid > 0` le signal est envoyé au processus de numéro `pid`
- si `pid = 0` le signal est envoyé à tous les processus du même groupe que `p`
- si `pid = -1` le signal est envoyé à tous les processus (sauf celui de numéro `-1`)
- si `pid < 0` le signal est envoyé à tous les processus du groupe `-pid`

Restrictions : un processus (sauf s'il a les droits de `root`) n'est autorisé à envoyer un signal qu'aux processus ayant le même `uid` (identité d'utilisateur) que lui.

Le processus de numéro 1 ne peut pas recevoir de signal (pour des raisons de sécurité) ; en effet il est l'ancêtre de tous les processus, et sa mort entraînerait celle de tous...). Il est donc protégé.

Redéfinir le traitant associé à un signal

On utilise la structure suivante :

```
struct sigaction {
        void (*sa_handler)() ;      pointeur sur traitant
        sigset_t sa_mask;           autres signaux à bloquer
        int sa_flags;               options
}
```

et la primitive :

```
int sigaction(int sig, const struct sigaction *newaction,
        struct sigaction *oldaction);
```

Autre variante plus simple issue du système BSD:

```
#include <signal.h>
typdef void handler_t (int)

handler_t *signal(int signum, handler_t *handler)
```

Associe le traitant `handler` au signal de numéro `signum`

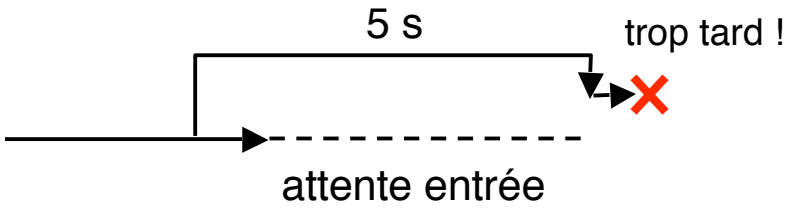
Exemple 2 : utilisation de la temporisation

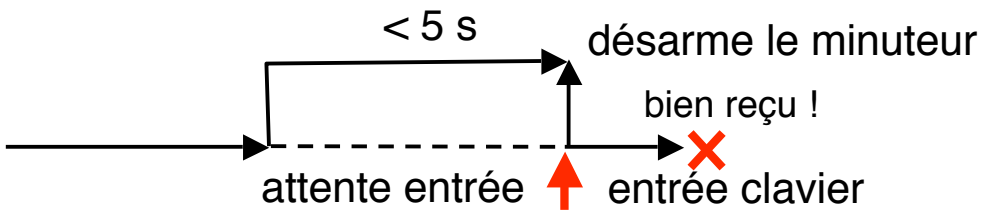
La primitive

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

provoque l'envoi du signal SIGALRM après environ `nbSec` secondes ; annulation avec `nbSec` =0

```
#include <unistd.h>
#include <signal.h>
void handler(int sig) {    /* nouveau traitant    */
  printf("trop tard !\n");
  exit(0);
}
int main() {
  int reponse;
  signal(SIGALRM, handler); /* installe le traitant */
  printf("entrer un nombre avant 5 sec. : ") ; alarm(5);
  scanf("%d", &reponse); alarm(0); printf("bien reçu !\n");
  exit (0);
}
```





Exemple 3 : synchronisation père-fils

Lorsqu'un processus se termine ou est suspendu, le système envoie automatiquement un signal SIGCHLD à son père. Le traitement par défaut consiste à ignorer ce signal.
On peut prévoir un traitement spécifique en associant un nouveau traitant à SIGCHLD

Application : lorsqu'un processus crée un grand nombre de fils (par exemple un *shell* crée un processus pour traiter chaque commande), il doit prendre en compte leur fin dès que possible pour éviter une accumulation de processus zombis (qui consomment de la place dans les tables du système).

```
#include <unistd.h>
#include <sys/wait.h>
void handler(int sig) {          /* nouveau traitant          */
  pid_t pid; int statut;
  pid = waitpid(-1, &statut, 0); /* attend un fils quelconque */
  return;
}
int main() {
  signal(SIGCHLD, handler);      /* installe le traitant      */
  ... <création d'un certain nombre de fils, sur demande> ...
  exit (0),
}
```

Verrouillage de fichiers dans Unix : exemple

testlock.c

```
include <unistd.h>
#include <fcntl.h>
#define TRUE 1
int main(void) {
  int fd;
  fd = Open("toto", O_RDWR); /* doit exister */
  while(TRUE){
    if (lockf(fd, F_TLOCK, 0) == 0){
      printf("%d a verrouillé le fichier\n",
          getpid());
      sleep(5);
      if (lockf(fd, F_ULOCK, 0) == 0)
         printf("%d a déverrouillé le
             fichier\n", getpid());
      return;
    } else {
      printf("%d a trouvé le fichier déjà
         verrouillé, réessaie...\n", getpid());
      sleep (2);
    }
  }
}
```

Question : que se passe-t-il si on remplace `F_TLOCK` par `F_LOCK` ?

```
<unix> testlock & testlock &
15545 a verrouillé le fichier
[4] 15545
15546 a trouvé le fichier déjà
verrouillé, réessaie
[5] 15546
<unix> 15546 a trouvé le fichier
déjà verrouillé, réessaie
15546 a trouvé le fichier déjà
verrouillé, réessaie
15545 a déverrouillé le fichier
15546 a verrouillé le fichier
15546 a déverrouillé le fichier
<unix>
```