Programmation C et Système
Contrôle de processus

Régis Barbanchon

L2 Informatique

Identificateur de processus : `getpid()` (1/3)

Un processus est un programme en cours d'exécution.
Tout processus a un identificateur unique : le pid, un entier positif.

(Le pid d'un processus terminé peut être réattribué par le système, mais en général cette possibilité est utilisée le plus tard possible).

La fonction `getpid()` renvoie le pid du processus :

```
$ man 2 getpid
```

```
SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>
       pid_t getpid(void);
DESCRIPTION
       getpid()  returns  the  process  ID  of the calling process.  (This is
       often used by routines that generate unique temporary filenames.)
```

Identificateur de processus : `getpid()` (2/3)

Exemple : un processus affiche son pid.

```c
// File: getpid1.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main (void) {
  pid_t pid= getpid();
  printf ("I am the process %ld.\n", (long) pid);
  return 0;
}
```

```
$ clang -std=c99 -W -Wall -pedantic getpid1.c -o getpid1
$ ./getpid1
I am the process 26272.
$ ./getpid1
I am the process 26273.
$ ./getpid1
I am the process 26274.
```

On voit que le shell incrémente les pids,
bien qu'il pourrait réutiliser ceux des processsus terminés.

Identificateur de processus : `getpid()` (3/3)

`getpid()` permet de créer des noms de fichier uniques :

```c
// File: getpid2.c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main (void) {
  pid_t pid= getpid();
  char filename [256];
  sprintf (filename, "./process-%ld.log", (long) pid);
  FILE * file= fopen (filename, "w");
  if (file == NULL) { perror ("fopen()"); exit(1); }
  fprintf (file, "I am the process %ld.\n", (long) pid);
  fclose (file);
  return 0;
}
```

Ici, chaque processus écrit dans le fichier de log associé à son pid :

```
$ clang -std=c99 -W -Wall -pedantic getpid2.c -o getpid2
$ ./getpid2
$ ./getpid2
$ ls process-*
process-26275.log
process-26276.log
$ cat process-26275.log
I am the process 26275.
$ cat process-26276.log
I am the process 26276.
```

Création de processus : `fork()` (1/2)

Pour créer un nouveau processus, il faut qu'un processus se clone via la fonction `fork()`.

```
$ man 2 fork
```

```
SYNOPSIS
       #include <unistd.h>
       pid_t fork(void);
DESCRIPTION
       fork()  creates a new process by duplicating the calling process.  The
       new process is referred to as the child process.  The calling  process
       is referred to as the parent process.

       The  child  process and the parent process run in separate memory spa-
       ces.  At the time of fork() both memory spaces have the same  content.
RETURN VALUE
       On  success,  the  PID of the child process is returned in the parent,
       and 0 is returned in the child.  On failure, -1  is  returned  in  the
       parent, no child process is created, and errno is set appropriately.
```

Si `fork()` réussit, on a donc deux processus à son retour, exécutant le même code, dans deux espaces mémoire séparés.
Le processus enfant est créé à l' identique de son parent, mais :

- la valeur de retour de `fork()` diffère,
- l'enfant reçoit un nouveau pid.

Création de processus : `fork()` (2/2)

Exemple : Le parent et l'enfant affichent leur pid après `fork()`.

```c
// File: fork1.c
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <sys/types.h>
#include <unistd.h>

int main (void) {
  pid_t parent= getpid();
  pid_t leg= fork();
  if (leg == -1) { perror ("fork()"); exit (1); }
  if (leg ==  0) {
    pid_t child= getpid();
    printf ("I am the child  %ld of parent %ld.\n", (long)child, (long)parent);
    exit (0);
  }
  assert (getpid() == parent);
  pid_t child= leg;
  printf ("I am the parent %ld of child  %ld.\n", (long)parent, (long)child);
  return 0;
}
```

Une sortie possible :

```
$ clang -std=c99 -W -Wall -pedantic fork1.c -o fork1
$ ./fork1
I am the parent 27269 of child  27270.
I am the child  27270 of parent 27269.
```

Création de processus : `fork()` et mémoire non partagée

On peut vérifier qu'il y a bien deux espaces mémoire, initialement identiques mais non partagés, comme suit :

```
// File: fork2.c (#include omitted)

int main(void) {
  int evil= 666;
  pid_t parent= getpid();
  pid_t leg= fork();
  if (leg == -1) { perror ("fork()"); exit (1); }
  if (leg ==  0) {
    evil /= 2;
    printf ("In child, evil is now %d.\n", evil);
    exit (0);
  }
  sleep (1); // let a chance to child to execute before parent
  printf ("In parent, evil is still %d.\n", evil);
  return 0;
}
```

Sauf accident, l'enfant s'exécute pendant que le parent dort et :

```
$ ./fork2
In child, evil is now 333.          [1 second pause here before next line shows]
In parent, evil is still 666.
```

Création de processus : `fork()` et buffered output (1/3)

Le buffering de `<stdio.h>` peut créer des surprises :

```
// File: fork3.c (#include omitted)

int main(void) {
  fprintf (stdout, "Hello!\n");
  pid_t leg= fork();
  if (leg == -1) { perror ("fork()"); exit (1); }
  if (leg ==  0) { fprintf (stdout, "I am the child.\n"); exit (0); }
  fprintf (stdout, "I am the parent.\n");
  return 0;
}
```

- Si `stdout` va vers la console, alors elle est line-bufferée, et `Hello!\n` a déjà été flushé par `\n` avant l'appel du `fork()`.
- Si `stdout` est redirigée, alors elle est fully-bufferée, et `Hello!\n` est toujours dans le buffer lors du `fork()` :

```
$ ./fork3     # line-buffered
Hello!
I am the parent.
I am the child.
```

```
$ ./fork3 | nl     # fully-buffered
     1  Hello!
     2  I am the parent.
     3  Hello!
     4  I am the child.
```

Dans le dernier cas, on voit donc apparaître `Hello!\n` dans les sorties des deux processus au lieu de celle du parent seulement.

Création de processus : `fork()` et buffered output (2/3)

On peut corriger le problème avec un `fflush()` avant le `fork()` :

```
SYNOPSIS
       #include <stdio.h>
       int fflush(FILE *stream);
DESCRIPTION
       For  output streams, fflush() forces a write of all user-space buffered
       data for the given output or update stream via the stream's  underlying
       write function. If the stream argument is  NULL, fflush()  flushes  all
       open  output streams.
RETURN VALUE
       Upon  successful  completion 0 is returned.  Otherwise, EOF is returned
       and errno is set to indicate the error.
```

```c
// File: fork4.c (#include omitted)

int main(void) {
  fprintf (stdout, "Hello!\n");
  fflush (NULL);
  pid_t leg= fork();
  if (leg == -1) { perror ("fork()"); exit (1); }
  if (leg ==  0) { fprintf (stdout, "I am the child.\n"); exit (0); }
  fprintf (stdout, "I am the parent.\n");
  return 0;
}
```

```
$ ./fork4     # useless fflush()
Hello!
I am the parent.
I am the child.
```

```
$ ./fork4 | nl     # useful fflush()
     1  Hello!
     2  I am the parent.
     3  I am the child.
```

Création de processus : `fork()` et buffered output (3/3)

On peut demander que `stdout` soit line-bufferé via `setvbuf()` :

```
$ man 3 setvbuf
```

```
SYNOPSIS
       #include <stdio.h>
       int setvbuf(FILE *stream, char *buf, int mode, size_t size);
DESCRIPTION
       The setvbuf() function may be used on any open  stream  to  change  its
       buffer.  The mode argument must be one of the following three macros:
       _IONBF (unbuffered), _IOLBF (line buffered), _IOFBF (fully buffered).
RETURN VALUE
       The  function  setvbuf()  returns  0 on success.  It returns nonzero on
       failure (mode is invalid or the request cannot be honored).  It may set
       errno on failure.
```

```c
// File: fork5.c (#include omitted)

int main(void) {
  setvbuf (stdout, NULL, _IOLBF, 0);
  fprintf (stdout, "Hello !\n");
  pid_t leg= fork();
  if (leg == -1) { perror ("fork()"); exit (1); }
  if (leg ==  0) { fprintf (stdout, "I am the child.\n"); exit (0); }
  fprintf (stdout, "I am the parent.\n");
  return 0;
}
```

```
$ ./fork5     # line-buffered
Hello!
I am the parent.
I am the child.
```

```
$ ./fork5 | nl     # line-buffered too
     1  Hello!
     2  I am the parent.
     3  I am the child.
```

Création de processus : `fork()` et unbuffered output

Et si on avait utilisé `write()` pour afficher `Hello\n` ?

Rappel : le file descriptor de `stdout` est `1` ou `STDOUT_FILENO`.

```
// File: fork6.c (most #include omitted)
#include <string.h>

int main(void) {
  char hello[]= "Hello!\n";
  write (STDOUT_FILENO, hello, strlen(hello));
  pid_t leg= fork();
  if (leg == -1) { perror ("fork()"); exit (1); }
  if (leg ==  0) { fprintf (stdout, "I am the child.\n"); exit (0); }
  fprintf (stdout, "I am the parent.\n");
  return 0;
}
```

L'écriture avec `write()` n'est pas bufferée, donc il n'y a pas de problèmes liés aux buffers non-flushés avant leur duplication :

```
$ ./fork6     # 1st line unbuffered
Hello!
I am the parent.
I am the child.
```

```
$ ./fork6 | nl     # 1st line unbuffered
     1  Hello!
     2  I am the parent.
     3  I am the child.
```

Processus parent : `getppid()` et adoption d'orphelin (1/2)

On peut connaître le pid du parent via la fonction `getppid()`. Cependant, si le parent est terminé, l'enfant orphelin est adopté par un processus d'initialisation, et `getppid()` retourne son pid :

```
// File: getppid.c (#include omitted)

int main (int argc, char * argv[]) {
  if (argc != 1+2) { fprintf (stderr, "Usage: ...\n") ; exit(1); }
  int creatorSleepTime= atoi (argv[1]), childSleepTime= atoi (argv[2]);
  pid_t creator= getpid();
  pid_t leg= fork();
  if (leg == -1) { perror ("fork()"); exit (1); }
  if (leg ==  0) {
    sleep (childSleepTime);
    pid_t child= getpid(), parent= getppid();
    printf ("I am the child %ld of parent %ld.\n", (long)child, (long)parent);
    exit (0);
  }
  sleep (creatorSleepTime);
  printf ("I am the creator %ld.\n", (long)creator);
  return 0;
}
```

```
$ ./getppid 1 0
I am the child 30787 of parent 30786.
I am the creator 30786.
```

```
$ ./getppid 0 1
I am the creator 30782.
$ I am the child 30783 of parent 1234.
```

À gauche, le parent survit à son enfant, qui n'est donc pas adopté.
À droite, le parent termine avant l'enfant, qui est alors est adopté.

Processus parent : `getppid()` et adoption d'orphelin (2/2)

La sortie produite rappelée ci-dessous appelle quelques remarques.

```
$ ./getppid 1 0
I am the child 30787 of parent 30786.
I am the creator 30786.
```

```
$ ./getppid 0 1
I am the creator 30782.
$ I am the child 30783 of parent 1234.
```

Remarque 1 : comme le parent termine avant l'enfant à droite, le prompt du shell (ici `$`) se ré-affiche avant l'affichage de l'enfant.

En effet, le shell attend la terminaison du processus en avant-plan (donc le parent `./getpid`) pour redonner la main à l'utilisateur, ce qu'il signale par l'affichage du prompt dès sa terminaison.

Remarque 2 : traditionnellement, c'est le processus `/sbin/init` dont le pid est toujours `1` qui adopte les processus orphelins.

Sur Ubuntu dont est tirée la sortie ci-dessus, ce n'est plus vrai, et c'est le processus `/sbin/upstart --user` qui s'en charge. Il n'a pas de pid fixe reservé (ici c'était `1234` lors de cette session).

Attente par `wait()` de la terminaison d'un enfant (1/2)

Lorsqu'un processus fils termine et que son parent est en vie,
le fils reste dans la table des processus dans un état dit **zombie**,
afin que le père puisse en être informé, jusqu'à ce que :

- le père enterre le fils via `wait()`, `waitpid()`, ou similaire.
- le père termine. Le fils est adopté et enterré par `upstart`.

Enterrer le fils permet de le sortir de la table des processus.
L'appel `int info; pid_t pid= wait (& info);` ...

- retourne `-1` si aucun fils n'est vivant ou zombie lors de l'appel,
- bloque jusqu'à ce qu'il y ait un fils **zombie** sinon.
  Celui-ci sort de la table des processus et son pid est retourné.
  Les infos sur sa terminaion vont dans `info`.

`WIFEXITED (info)` indique si le fils s'est terminé normalement,
et si oui, `WEXITSTATUS (info)` donne son exit status.
`WIFSIGNALED (info)` indique si le fils a été interrompu,
et si oui, `WTERMSIG (info)` donne le numéro du signal reçu.

Attente par `wait()` de la terminaison d'un enfant (2/2)

Le programme ci-dessous crée un fils et attend sa terminaison.
Le fils termine normalement ou non selon la présence d'un arg :

```c
// File: wait.c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <sys/types.h>
#include <sys/wait.h>

int main (int argc, char * argv[]) {
  bool abortChild= argc == 1+1 && strcmp (argv[1], "-abort") == 0;
  pid_t leg= fork();
  if (leg == -1) { perror ("fork()"); exit (1); }
  if (leg == 0 &&   abortChild) abort();
  if (leg == 0 && ! abortChild) exit (13);

  int info; pid_t child= wait (& info);
  if (child != -1 && WIFEXITED (info)) {
    int status= WEXITSTATUS (info);
    printf ("child %ld exited with status %d\n", (long) child, status);
  } else if (child != -1 && WIFSIGNALED (info)) {
    int signalId= WTERMSIG (info);
    printf ("child %ld interrupted by signal %d\n", (long) child, signalId);
  }
  return 0;
}
```

```
$ ./wait
child 2084 exited with status 13
```

```
$ ./wait -abort
child 2086 interrupted by signal 6
```

La variante `waitpid()` de `wait()`

La fonction `waitpid (pid, & info, OPTION|...)`
est une variante configurable de `wait(& info)`.

On peut, par exemple :

- spécifier l'enfant à attendre en précisant son `pid`,
  ou ne pas le spécifier avec la valeur `pid= -1`.
- option `WNOHANG` : vérifier s'il y a un enfant à attendre,
  la fonction retournant `0` sans bloquer s'il n'y en a pas.
- options `WUNTRACED` et `WCONTINUED` : être respectivement
  débloqué lorsqu'un enfant a été stoppé ou réveillé.

On peut voir `wait()` comme un wrapper sur `waitpid()`,
les appels suivants étant équivalents :

- `int info; pid_t child= wait (& info);`
- `int info; pid_t child= waitpid (-1, & info, 0);`

Recouvrement de l'image du processus via `execve()`

`execve()` recouvre l'image du programme appelant par un autre :

```
#include <unistd.h>
int execve (char const path[], char * const argv[], char const envp[]);
```

- `path[]` doit être le chemin d'un binaire ou d'un script, il peut être absolu (débutant par un slash `/`) ou relatif.
- `argv[]` et `envp[]` doivent se terminer par `NULL`, ce sont les args et les vars d'env transmis au programme.
- `execve()` retourne `-1` si elle échoue, et ne retourne pas sinon, le programme appelé se substituant au programme appelant.

Remarques :
En général , on reflète dans `argv[0]` la valeur de `path`, mais ce n'est pas obligatoire, et certains progs exploitent cette liberté.

La fonction `execve()` est typiquement invoquée après un `fork()` pour recouvrir l'image du programme du processus fils.

Exemple de recouvrement via `execve()`

Voici un exemple de recouvrement avec `/bin/date` :

```c
// File: execve.c
#include <stdio.h>
#include <unistd.h>

int main (int argc, char * argv[]) {
  char * path  = (argc >= 1+1) ? argv[1] : "/bin/date";
  char * envVar= (argc >= 1+2) ? argv[2] : "LC_ALL=en_US.UTF-8";
  execve (path,
          (char*[]) { path, "-d", "12/25/2018", NULL },
          (char*[]) { envVar, NULL } );
  perror ("execve()");
  return 13;
}
```

```
$ ./execve /bin/date LC_ALL=fr_FR.UTF-8 ; echo "exit status $?"
mardi 25 décembre 2018, 00:00:00 (UTC+0100)
exit status 0
```

```
$ ./execve /bin/date LC_ALL=en_US.UTF-8 ; echo "exit status $?"
Tue Dec 25 00:00:00 CET 2018
exit status 0
```

```
$ ./execve date ; echo "exit status $?"
execve(): No such file or directory
exit status 13
```

```
$ ./execve /bin ; echo "exit status $?"
execve(): Permission denied
exit status 13
```

Wrappers sur la fonction système `execve()`

La fonction `execv()` est un wrapper sur `execve()` transmettant la variable globale `environ` pour les variables d'environnement :

```
#include <unistd.h>
int execv (char const path[], char * const argv[]);
```

La fonction `execl()` est un wrapper variadique sur `execv()` construisant `argv[]` partir d'une liste variadique d'arguments. Cette liste doit se terminer par `(char *) NULL` :

```
#include <unistd.h>
int execl (char const path[], char const * argv0, ...);
```

Enfin, `execvp`/`execlp()` sont des wrappers sur `execv`/`execl()` où le chemin `path` du programme est construit à partir de :

- `path[]` seul, s'il commence par un slash (`/`),
- `path[]` et du premier répertoire de la var d'env `PATH` de `environ` qui permet de lancer le programme, sinon.

```
#include <unistd.h>
int execvp (char const path[], char * const argv[]);
int execlp (char const path[], char const * argv0, ...);
```

Exemple de recouvrement via `execv()`

Reprenons notre exemple de recouvrement par `/bin/date` :

```c
// File: execv.c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char * argv[]) {
  char * path= (argc == 1+1) ? argv[1] : "/bin/date";
  printf ("LC_ALL is <%s>\n", getenv ("LC_ALL"));
  execv (path, (char*[]) { path, "-d", "12/25/2018", NULL } );
  perror ("execv()");
  return 13;
}
```

```
$ LC_ALL=fr_FR.UTF-8 ./execv /bin/date ; echo "exit status $?"
LC_ALL is <fr_FR.UTF-8>
mardi 25 décembre 2018, 00:00:00 (UTC+0100)
exit status 0
```

```
$ LC_ALL=en_US.UTF-8 ./execv /bin/date ; echo "exit status $?"
LC_ALL is <en_US.UTF-8>
Tue Dec 25 00:00:00 CET 2018
exit status 0
```

```
$ LC_ALL=en_US.UTF-8 ./execv date ; echo "exit status $?"
LC_ALL is <en_US.UTF-8>
execv(): No such file or directory
exit status 13
```

Exemple de recouvrement via `execl()`

Reprenons notre exemple de recouvrement par `/bin/date` :

```c
// File: execl.c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char * argv[]) {
  char * path= (argc == 1+1) ? argv[1] : "/bin/date";
  printf ("LC_ALL is <%s>\n", getenv ("LC_ALL"));
  execl (path, path, "-d", "12/25/2018", (char *) NULL );
  perror ("execl()");
  return 13;
}
```

```
$ LC_ALL=fr_FR.UTF-8 ./execl /bin/date ; echo "exit status $?"
LC_ALL is <fr_FR.UTF-8>
mardi 25 décembre 2018, 00:00:00 (UTC+0100)
exit status 0
```

```
$ LC_ALL=en_US.UTF-8 ./execl /bin/date ; echo "exit status $?"
LC_ALL is <en_US.UTF-8>
Tue Dec 25 00:00:00 CET 2018
exit status 0
```

```
$ LC_ALL=en_US.UTF-8 ./execl date ; echo "exit status $?"
LC_ALL is <en_US.UTF-8>
execl(): No such file or directory
exit status 13
```

Exemple de recouvrement via `execlp()`

Reprenons notre exemple de recouvrement par `/bin/date` :

```c
// File: execl.c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char * argv[]) {
  char * path= (argc == 1+1) ? argv[1] : "/bin/date";
  printf ("LC_ALL is <%s>, PATH is <%s>\n", getenv ("LC_ALL"), getenv ("PATH"));
  execlp (path, path, "-d", "12/25/2018", (char *) NULL );
  perror ("execlp()");
  return 13;
}
```

```
$ LC_ALL=fr_FR.UTF-8 PATH="" ./execlp /bin/date ; echo "exit status $?"
LC_ALL is <fr_FR.UTF-8>, PATH is <>
mardi 25 décembre 2018, 00:00:00 (UTC+0100)
exit status 0
```

```
$ LC_ALL=en_US.UTF-8 PATH="/bin:/usr/bin" ./execlp date ; echo "exit status $?"
LC_ALL is <en_US.UTF-8>, PATH is </bin:/usr/bin>
Tue Dec 25 00:00:00 CET 2018
exit status 0
```

```
$ LC_ALL=en_US.UTF-8 PATH="" ./execlp date ; echo "exit status $?"
LC_ALL is <en_US.UTF-8>, PATH is <>
execlp(): No such file or directory
exit status 13
```

La fonction `system()`

La fonction `system(cmd)` lance une commande `cmd` dans un shell.
Elle combine `fork()`, `execl()` et `wait()` pour lancer `sh -c cmd`.

```
$ man 3 system
```

```
SYNOPSIS
      #include <stdlib.h>
      int system(const char *command);

DESCRIPTION
       The  system()  library  function uses fork()  to create a child process
       that executes the shell command specified in command using execl():

       execl("/bin/sh", "sh", "-c", command, (char *) NULL);

RETURN VALUE
       The return value of system() is one of the following:

       *  If command is NULL, then a nonzero value if a shell is available, or
          0 if no shell is available.
       *  If a child process could not be created, or its status could not  be
          retrieved, the return value is -1.
       *  If  a  shell  could  not  be executed in the child process, then the
          return value is as though the  child  shell  terminated  by  calling
          _exit() with the status 127.
       *  If  all  system calls succeed, then the return value is the termina-
          tion status of the child shell used to execute command.  (The termi-
          nation  status of a shell is the termination status of the last com-
          mand it executes.)

       In the last two cases, the return value is a "wait status" that can  be
       examined using the macros WIFEXITED(), WEXITSTATUS(), and so on.
```