Votre nom :
Votre prénom :
Votre no. d'étudiant :

Contrôle de LIFASR5 — 24 minutes — sujet 1

le mardi 26/03/2019

Toute communication (orale, téléphonique, par messagerie, télépathique, etc.) avec les autres étudiants est interdite. Tout document est interdit, tout comme toute utilisation d'un ordinateur, d'un téléphone, d'une montre connectée, etc. Le non-respect de ces consignes sera sanctionné par une exclusion de l'épreuve et la note de 0. Vous disposez uniquement de l'antisèche fournie, qui contient toutes les informations utiles. Ce sujet est à rendre complété : n'oubliez pas de noter votre nom, votre prénom et votre no. étudiant.

Sujet 1 Donnez une fonction `int main(void)` dans laquelle :

- le processus initial, appelé `PERE`, crée (`fork()`) un processus appelé `FILS` ;
- processus `FILS` crée (`fork()`) un processus appelé `PETIT-FILS` ;
- le processus `PETIT-FILS` se met en sommeil pour 5s (`sleep(5)`), puis se termine ;
- le processus `FILS` attend que `PETIT-FILS` se termine (`waitpid(...)`), puis se termine lui-même (`return EXIT_SUCCESS`) ;
- `PERE` attend que `FILS` se termine (`waitpid()`), puis se termine (`return EXIT_SUCCESS`).

Veillez à bien prendre en compte les cas d'erreurs pour les appels systèmes, en appelant la fonction `void exit_error(void)` pour mettre fin au processus courant. Utilisez la valeur de retour de `fork()`, ainsi que la fonction `getpid()` de façon à ce que votre programme produise l'affichage suivant :

```
nlouvet@nlbook:~/$ ./ex1
(PERE) j'attends FILS
(FILS) j'ai pour PID 10417
(FILS) je vais créer PETIT-FILS
(FILS) j'attends PETIT-FILS
(PETIT-FILS) j'ai pour PID 10418
(PETIT-FILS) je me mets en sommeil pour 5s
(PETIT-FILS) je me termine
(FILS) PETIT-FILS de PID 10418 est terminé, je me termine
(PERE) FILS de PID 10417 est terminé, je me termine
nlouvet@nlbook:~/$
```

Solution:

```cpp
#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <string.h>

using namespace std;

void exit_error() {
  cerr << "Erreur : " << strerror(errno) << endl;
  exit(EXIT_FAILURE);
}

int main(void) {

  int pid1 = fork();
  if(pid1 == -1) exit_error();

  if(pid1 == 0) { // processus fils
    cout << "(FILS) j'ai pour PID " << getpid() << endl;
    cout << "(FILS) je vais créer PETIT-FILS" << endl;

    int pid2 = fork();
    if(pid2 == -1) exit_error();

    if(pid2 == 0) { // processus petit-fils
      cout << "(PETIT-FILS) j'ai pour PID " << getpid() << endl;
      cout << "(PETIT-FILS) je me mets en sommeil pour 5s" << endl;
      sleep(5);
      cout << "(PETIT-FILS) je me termine" << endl;
      return EXIT_SUCCESS;
    }

    // processus fils
    cout << "(FILS) j'attends PETIT-FILS" << endl;
    int res = waitpid(pid2, NULL, 0);
    if(res == -1) exit_error();
    cout << "(FILS) PETIT-FILS de PID " << res << " est terminé, je me termine" << endl;
    return EXIT_SUCCESS;
  }

  // processus père
  cout << "(PERE) j'attends FILS" << endl;
  if(waitpid(pid1, NULL, 0) == -1) exit_error();
  cout << "(PERE) FILS de PID " << pid1 << " est terminé, je me termine" << endl;

  return EXIT_SUCCESS;
}
```

Votre nom :
Votre prénom :
Votre no. d'étudiant :

Contrôle de LIFASR5 — 24 minutes — sujet 2

le mardi 26/03/2019

Toute communication (orale, téléphonique, par messagerie, télépathique, etc.) avec les autres étudiants est interdite. Tout document est interdit, tout comme toute utilisation d'un ordinateur, d'un téléphone, d'une montre connectée, etc. Le non-respect de ces consignes sera sanctionné par une exclusion de l'épreuve et la note de 0. Vous disposez uniquement de l'antisèche fournie, qui contient toutes les informations utiles. Ce sujet est à rendre complété : n'oubliez pas de noter votre nom, votre prénom et votre no. étudiant.

Sujet 2 Vous devez compléter le code suivant de façon à ce que, lorsque le processus père reçoit le signal `SIGINT`,

- le gestionnaire `sigint_handler()` affiche le message « `(PERE) je dois tout fermer` », puis fasse passer la variable booléenne `stop` à `true` ;
- le père sorte de la boucle d'attente `do pause(); while(!stop);` et envoie le signal `SIGQUIT` au fils, ce qui provoque sa terminaison (vous n'avez rien de particulier à faire ici) ;
- le père prenne en compte la mort de son fils avec `waitpid()`.

En outre, vous utilisez la valeur de retour de `fork()`, ainsi que la fonction `getpid()` de façon à ce que votre programme produise l'affichage suivant (le `^C` correspond à l'instant où le signal `SIGINT` est envoyé au père) :

```
nlouvet@nlbook:~/$ ./ex3
(PERE) mon PID est 10715
(PERE) je vais créer un fils
(FILS) j'ai pour PID 10716
(FILS) je fais des trucs
^C(PERE, de PID 10715) je dois tout fermer
(PERE) mon fils de PID 10716 est teminé, je me termine
nlouvet@nlbook:~/$
```

Veillez à bien prendre en compte les cas d'erreurs pour les appels systèmes, en appelant la fonction `void exit_error(void)` pour mettre fin au processus courant.

```
bool stop = false; // variable booléenne initialisée à false
void sigint_handler(int sig) {
  ..........................................................
  ..........................................................
}


int main(void) {
  ..........................................................
  ..........................................................
  int pid = fork();
  if(pid == -1) exit_error();
  if(pid == 0) { // processus fils
    struct sigaction sa;
    if(sigaction(SIGINT, NULL, &sa) == -1) exit_error();
    sa.sa_handler = SIG_IGN;
    if(sigaction(SIGINT, &sa, NULL) == -1) exit_error();
    ..........................................................
    cout << "(FILS) je fais des trucs" << endl;
    do_things();
    cout << "(FILS) je me termine" << endl;
    return EXIT_SUCCESS;
  }
  // processus père
  ..........................................................
  ..........................................................
  ..........................................................
  ..........................................................
  do pause(); while(!stop); // boucle d'attente
  if(kill(pid, SIGQUIT) == -1) exit_error();
  ..........................................................
  ..........................................................
  return EXIT_SUCCESS;
}
```

Solution:

```cpp
#include <iostream>
#include <atomic>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <errno.h>
#include <string.h>

using namespace std;

void exit_error() {
  cerr << "Erreur : " << strerror(errno) << endl;
  exit(EXIT_FAILURE);
}

atomic<bool> stop(false);

// gestionnaire du signal SIGINT.
void sigint_handler(int sig) {
  cout << "(PERE, de PID " << getpid() << ") je dois tout fermer" << endl << flush;
  stop = true;
}

int main(void) {
  cout << "(PERE) mon PID est " << getpid() << endl;
  cout << "(PERE) je vais créer un fils" << endl;

  int pid = fork();

  if(pid == -1) exit_error();

  if(pid == 0) { // processus fils
    struct sigaction sa;
    if(sigaction(SIGINT, NULL, &sa) == -1) exit_error();
    sa.sa_handler = SIG_IGN;
    if(sigaction(SIGINT, &sa, NULL) == -1) exit_error();

    cout << "(FILS) j'ai pour PID " << getpid() << endl;
    cout << "(FILS) je fais des trucs" << endl;
    pause();
    cout << "(FILS) je me termine" << endl;

    return EXIT_SUCCESS;
  }

  // processus père

  struct sigaction sa;
  if(sigaction(SIGINT, NULL, &sa) == -1) exit_error();
  sa.sa_handler = sigint_handler;
  if(sigaction(SIGINT, &sa, NULL) == -1) exit_error();

  do pause(); while(!stop);

  if(kill(pid, SIGQUIT) == -1) exit_error();
  if(waitpid(pid, NULL, 0) == -1) exit_error();

  cout << "(PERE) mon fils de PID " << pid << " est teminé, je me termine" << endl;

  return EXIT_SUCCESS;
}
```

Votre nom :
Votre prénom :
Votre no. d'étudiant :

Contrôle de LIFASR5 — 24 minutes — sujet 3

le mardi 26/03/2019

Toute communication (orale, téléphonique, par messagerie, télépathique, etc.) avec les autres étudiants est interdite. Tout document est interdit, tout comme toute utilisation d'un ordinateur, d'un téléphone, d'une montre connectée, etc. Le non-respect de ces consignes sera sanctionné par une exclusion de l'épreuve et la note de 0. Vous disposez uniquement de l'antisèche fournie, qui contient toutes les informations utiles. Ce sujet est à rendre complété : n'oubliez pas de noter votre nom, votre prénom et votre no. étudiant.

Sujet 3 Donnez une fonction `int main(void)` dans laquelle :

- le processus père crée (`fork()`) deux processus fils, `FILS1` et `FILS2` ;
- le processus `FILS1` se met en sommeil pour 5s (`sleep(5)`) puis se termine ;
- le processus `FILS2` se met en sommeil pour 3s (`sleep(3)`) puis se termine (`return EXIT_SUCCESS`) ;
- le père attend que `FILS1` puis `FILS2` se terminent (`waitpid(...)`), puis se termine lui-même (`return EXIT_SUCCESS`).

Veillez à bien prendre en compte les cas d'erreurs pour les appels systèmes, en appelant la fonction `void exit_error(void)` pour mettre fin au processus courant. Utilisez la valeur de retour de `fork()`, ainsi que la fonction `getpid()` de façon à ce que votre programme produise l'affichage suivant :

```
nlouvet@nlbook:~/$ ./ex2
(PERE) j'attends le FILS1
(FILS2) j'ai pour PID 10369
(FILS1) j'ai pour PID 10368
(FILS2) je me mets en sommeil pour 3s
(FILS1) je me mets en sommeil pour 5s
(FILS2) je me termine
(FILS1) je me termine
(PERE) FILS1 de PID 10368 terminé
(PERE) j'attends le FILS2
(PERE) FILS2 de PID 10369 terminé
(PERE) je me termine
nlouvet@nlbook:~/$
```

Solution:

```cpp
#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <string.h>

using namespace std;

void exit_error() {
  cerr << "Erreur : " << strerror(errno) << endl;
  exit(EXIT_FAILURE);
}

int main(void) {

  int pid1 = fork();
  if(pid1 == -1) exit_error();
  if(pid1 == 0) { // processus fils1
    cout << "(FILS1) j'ai pour PID " << getpid() << endl;
    cout << "(FILS1) je me mets en sommeil pour 5s" << endl;
    sleep(5);
    cout << "(FILS1) je me termine" << endl;
    return EXIT_SUCCESS;
  }

  int pid2 = fork();
  if(pid2 == -1) exit_error();
  if(pid2 == 0) { // processus fils2
    cout << "(FILS2) j'ai pour PID " << getpid() << endl;
    cout << "(FILS2) je me mets en sommeil pour 3s" << endl;
    sleep(5);
    cout << "(FILS2) je me termine" << endl;
    return EXIT_SUCCESS;
  }

  // processus père
  cout << "(PERE) j'attends le FILS1" << endl;
  if(waitpid(pid1, NULL, 0) == -1) exit_error();
  cout << "(PERE) FILS1 de PID " << pid1 << " terminé" << endl;

  cout << "(PERE) j'attends le FILS2" << endl;
  if(waitpid(pid2, NULL, 0) == -1) exit_error();
  cout << "(PERE) FILS2 de PID " << pid2 << " terminé" << endl;

  cout << "(PERE) je me termine" << endl;

  return EXIT_SUCCESS;
}
```

Votre nom :
Votre prénom :
Votre no. d'étudiant :

Contrôle de LIFASR5 — 24 minutes — sujet 4

le mardi 26/03/2019

Toute communication (orale, téléphonique, par messagerie, télépathique, etc.) avec les autres étudiants est interdite. Tout document est interdit, tout comme toute utilisation d'un ordinateur, d'un téléphone, d'une montre connectée, etc. Le non-respect de ces consignes sera sanctionné par une exclusion de l'épreuve et la note de 0. Vous disposez uniquement de l'antisèche fournie, qui contient toutes les informations utiles. Ce sujet est à rendre complété : n'oubliez pas de noter votre nom, votre prénom et votre no. étudiant.

Sujet 4 Vous devez compléter le code suivant de façon à ce que :

- le programme principal crée un tube (primitive `pipe()`) avant la création d'un fils avec `fork()`
- le processus `FILS` envoie 10 caractères au processus `PERE` (les caractères `'a'` à `'j'`, pour fixer les idées), à 1s d'intervalle (`sleep(1)`) ; via le tube, ferme son descripteur de fichier en écriture sur le tube, puis se termine ;
- le processus `PERE` lit tous les caractères que son fils lui envoie, les affiche sur la sortie standard, et ce jusqu'à ce que le fils ferme son descripteur de fichier en écriture sur le tube ; le processus `PERE` prend alors en compte la fin de son fils avec `waitpid(...)`, puis se termine.

Veillez à bien prendre en compte les cas d'erreurs pour les appels systèmes, en appelant la fonction `void exit_error(void)` pour mettre fin au processus courant. Veillez à fermer les descripteurs de fichiers non-utilisés dès que possible. Voici un exemple d'exécution du programme :

```
nlouvet@nlbook:~/$ ./ex5
(PERE) j'attends des nouvelle du FILS
(FILS) j'ai pour PID 10169
(PERE) j'ai reçu un caractère : a
...
(PERE) j'ai reçu un caractère : j
(FILS) je me termine
(PERE) FILS de PID 10169 est terminé, je me termine
nlouvet@nlbook:~/$
```

Voici le programme à compléter :

```
int main(void) {
  int res;
  char c;
  ..........................................................
  ..........................................................
  ..........................................................

  int pid = fork();
  if(pid == -1) exit_error();

  if(pid == 0) {
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    ..........................................................
    return EXIT_SUCCESS;
  }
  ..........................................................
  ..........................................................
  ..........................................................
  ..........................................................
  ..........................................................
  ..........................................................
  ..........................................................
  ..........................................................
  return EXIT_SUCCESS;
}
```

Solution:

```
#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <string.h>

using namespace std;

void exit_error() {
  cerr << "Erreur : " << strerror(errno) << endl;
  exit(EXIT_FAILURE);
}

int main(void) {
  int res;
  char c;

  int pipefd[2];
  // pipefd[0] pour la lecture
  // pipefd[1] pour l'écriture
  if(pipe(pipefd) == -1) exit_error();

  int pid = fork();
  if(pid == -1) exit_error();

  if(pid == 0) {
    // processus fils
    cout << "(FILS) j'ai pour PID " << getpid() << endl;
    close(pipefd[0]);
    for(int i=0; i<10; i++) {
      c = 'a' + i;
      if(write(pipefd[1], &c, 1) == -1) exit_error();
      sleep(1);
    }
    close(pipefd[1]);
    cout << "(FILS) je me termine" << endl;
    return EXIT_SUCCESS;
  }

  // processus père
  cout << "(PERE) j'attends des nouvelle du FILS" << endl;
  close(pipefd[1]);
  while((res = read(pipefd[0], &c, 1)) == 1)
    cout << "(PERE) j'ai reçu un caractère : " << c << endl;
  if(res == -1) exit_error();
  close(pipefd[0]);
  if(waitpid(pid, NULL, 0) == -1) exit_error();
  cout << "(PERE) le FILS est terminé, je me termine" << endl;

  return EXIT_SUCCESS;
}
```

Antisèche :

NAME open -open a file
SYNOPSIS int open(const char *pathname, int flags);
DESCRIPTION
The open() system call opens the file specified by pathname. The return value of open() is a file descriptor, a small, nonnegative integer that is used in subsequent system calls (read(), write(), etc.) to refer to the open file. The argument flags must include one of the following access modes: O_RDONLY, O_WRONLY, or O_RDWR. These request opening the file read-only, write-only, or read/write, respectively.
RETURN VALUE
open() returns the new file descriptor, or -1 if an error occurred (in which case, errno is set appropriately).

==========

NAME read -read from a file descriptor
SYNOPSIS ssize_t read(int fd, void *buf, size_t count);
DESCRIPTION
read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.
RETURN VALUE
On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because read() was interrupted by a signal. On error, -1 is returned, and errno is set appropriately.

==========

NAME write -write to a file descriptor
SYNOPSIS ssize_t write(int fd, const void *buf, size_t count);
DESCRIPTION
write() writes up to count bytes from the buffer starting at buf to the file referred to by the file descriptor fd.
RETURN VALUE
On success, the number of bytes written is returned (zero indicates nothing was written). It is not an error if this number is smaller than the number of bytes requested; this may happen for example because the disk device was filled. On error, -1 is returned, and errno is set appropriately.

==========

NAME fork -create a child process
SYNOPSIS 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.
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.

==========

NAME waitpid -wait for process to change state
SYNOPSIS pid_t waitpid(pid_t pid, int *wstatus, int options);
DESCRIPTION
waitpid() is used to wait for state changes in a child of the calling process. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state.

If pid $>$ 0, then the call will wait for the children whose PID equals pid.

If wstatus is not NULL, then waitpid() stores status in informations in the int it points to. If wstatus is NULL, then this parameter is ignored. The value of options is an OR of zero or more of the following constants : WNOHANG, WUNTRACED, WCONTINUED.
RETURN VALUE
On success, waitpid() returns the process ID of the child whose state has changed; if WNOHANG was specified and one or more child(ren) specified by pid exist, but have not yet changed state, then 0 is returned. On error, -1 is returned.

==========

NAME getpid, getppid -get process identification
SYNOPSIS pid_t getpid(void);
pid_t getppid(void);
DESCRIPTION
getpid() returns the process ID (PID) of the calling process. getppid() returns the process ID of the parent of the calling process. This will be either the ID of the process that created this process using fork(), or, if that process has already terminated, the ID of the process to which this process has been reparented.
ERRORS
These functions are always successful.

==========

NAME pipe -create pipe
SYNOPSIS int pipe(int pipefd[2]);
DESCRIPTION
pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array pipefd is used to return two file descriptors referring to the ends of the pipe. pipefd[0] refers to the read end of the pipe. pipefd[1] refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.

If all file descriptors referring to the write end of a pipe have been closed, then an attempt to read from the pipe will see end-of- file and will return 0. If all file descriptors referring to the read end of a pipe have been closed, then a write will cause a SIGPIPE signal to be generated for the calling process. If the calling process is ignoring this signal, then write fails with the error EPIPE. An application that uses pipe and fork should use suitable close calls to close unnecessary duplicate file descriptors; this ensures that end-of- file and SIGPIPE/EPIPE are delivered when appropriate.
RETURN VALUE
On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

==========

NAME sigaction -examine and change a signal action
SYNOPSIS int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
DESCRIPTION
The sigaction() system call is used to change the action taken by a process on receipt of a specific signal. signum specifies the signal and can be any valid signal except SIGKILL and SIGSTOP. If act is non-NULL, the new action for signal signum is installed from act. If oldact is non-NULL, the previous action is saved in oldact. The sigaction structure is defined as something like:

```
struct sigaction {
  void     (*sa_handler)(int);
  ... ;
};
```

sa_handler specifies the action to be associated with signum and may be SIG_DFL for the default action, SIG_IGN to ignore this signal, or a pointer to a signal handling function. This function receives the signal number as its only argument.
RETURN VALUE
sigaction() returns 0 on success; on error, -1 is returned, and errno is set to indicate the error.