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

Contrôle de LIFASR5 — 51 minutes

le lundi 01/04/2019

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.

---

1 Autour des processus

1. On considère le programme suivant. Comme d'habitude, `exit_error()` termine le processus appelant en affichant une erreur. On suppose que le shell dans lequel on lance le programme a pour PID 150, que le processus `main()` a pour PID 610, le processus créé par le premier `fork()` a pour PID 611, et celui créé par le deuxième `fork()` a pour PID 612. Donnez une trace possible des messages qui s'affichent sur la sortie standard à l'exécution du programme.

```
int main(void) {
    cout << "(P0) processus de PID " << getpid() << endl;
    cout << "(P0) mon père a pour PID " << getppid() << endl;

    int pid1 = fork();
    if(pid1 == -1) exit_error();
    if(pid1 == 0) {
        cout << "(P1) processus de PID " << getpid() << endl;
        cout << "(P1) mon père a pour PID " << getppid() << endl;
        sleep(2);
        cout << "(P1) je crée un fils" << endl;

        int pid2 = fork();
        if(pid2 == -1) exit_error();
        if(pid2 == 0) {
            cout << "(P2) processus de PID " << getpid() << endl;
            cout << "(P2) mon père a pour PID " << getppid() << endl;
            sleep(4);
            cout << "(P2) je me termine" << endl;
            exit(EXIT_SUCCESS);
        }
        cout << "(P1) j'attends la fin P2 de PID " << pid2 << endl;
        if(waitpid(pid2, NULL, 0) == -1) exit_error();
        cout << "(P1) je me termine" << endl;
        exit(EXIT_SUCCESS);
    }
    cout << "(P0) j'attends la fin P1 de PID " << pid1 << endl;
    if(waitpid(pid1, NULL, 0) == -1) exit_error();
    cout << "(P0) je me termine" << endl;
    return EXIT_SUCCESS;
}
```

Solution:

```
(P0) processus de PID 610
(P0) mon père a pour PID 150
(P0) j'attends la fin P1 de PID 611
(P1) processus de PID 611
(P1) mon père a pour PID 610
(P1) je crée un fils
(P1) j'attends la fin P2 de PID 612
(P2) processus de PID 612
(P2) mon père a pour PID 611
(P2) je me termine
(P1) je me termine
(P1) je me termine
```

2. Même chose (avec les mêmes hypothèses), pour le programme ci-dessous.

```
int main(void) {
    cout << "(P0) processus de PID " << getpid() << endl;

    int pid1 = fork();
    if(pid1 == -1) exit_error();
    if(pid1 == 0) {
        cout << "(P1) processus de PID " << getpid() << endl;
        cout << "(P1) mon père a pour PID " << getppid() << endl;
        sleep(2);
        exit(EXIT_SUCCESS);
    }

    int pid2 = fork();
    if(pid2 == -1) exit_error();
    if(pid2 == 0) {
        cout << "(P2) processus de PID " << getpid() << endl;
        cout << "(P2) mon père a pour PID " << getppid() << endl;
        sleep(4);
        exit(EXIT_SUCCESS);
    }

    if(waitpid(pid2, NULL, 0) == -1) exit_error();
    cout << "(P0) le processus P2 est terminé" << endl;

    if(waitpid(pid1, NULL, 0) == -1) exit_error();
    cout << "(P0) le processus P1 est terminé" << endl;

    return EXIT_SUCCESS;
}
```

Solution:

```
(P0) processus de PID 610
(P1) processus de PID 611
(P1) mon père a pour PID 610
(P2) processus de PID 612
(P2) mon père a pour PID 610
(P0) le processus P2 est terminé
(P0) le processus P1 est terminé
```

2 Signaux

On considère le programme ci-dessous.

```
bool quit = false;
int n = 0;

void handler(int signum) {
    // signum est le numéro du signal pour lequel le gestionnaire a été appelé.
    switch(signum) {
        case SIGUSR1:
            n++;
            break;
        case SIGUSR2:
            n--;
            break;
        case SIGQUIT:
            quit = true;
            break;
        default:
            cerr << "Erreur : cela ne devrait pas se produire." << endl << flush;
            exit(EXIT_FAILURE);
    }
}

int main(void) {
    // Installation du gestionnaire de signal pour trois signaux différents.
    struct sigaction sa;
    if(sigemptyset(&sa.sa_mask) == -1) exit_error();
    sa.sa_flags = 0;
    sa.sa_handler = handler;
    if(sigaction(SIGUSR1, &sa, NULL) == -1) exit_error();
    if(sigaction(SIGUSR2, &sa, NULL) == -1) exit_error();
    if(sigaction(SIGQUIT, &sa, NULL) == -1) exit_error();

    // bloucle d'attente de signaux et d'affichage.
    while(true) {
```

```
        pause(); // attente d'un signal
        cout << "n = " << n << endl;
        if(quit) break;
    }
    cout << "Fin avec n = " << n << endl;
    return(EXIT_SUCCESS);
}
```

1. On suppose que le programme, compilé sous le nom de `ex3`, est en cours d'exécution dans un terminal, et on lui envoie des signaux avec la suite de commandes suivante depuis un shell, dans un autre terminal :

   ```
   $ pkill -USR1 ex3
   $ pkill -USR1 ex3
   $ pkill -USR2 ex3
   $ pkill -QUIT ex3
   ```

   Donnez une trace de l'affichage du programme dans son terminal.

   Solution: Le programme affiche :

   ```
   n = 1
   n = 2
   n = 1
   n = 1
   Fin avec n = 1
   ```

   Ensuite, le programme est terminé.

2. Le gestionnaire de signal prévu par défaut lors de la réception du signal `SIGTERM` provoque la terminaison du processus qui le reçoit. Donnez une trace de l'exécution du programme si on lui envoie les signaux suivants :

   ```
   $ pkill -USR2 ex3
   $ pkill -USR1 ex3
   $ pkill -USR1 ex3
   $ pkill -TERM ex3
   ```

   Solution: Le programme affiche :

   ```
   n = -1
   n = 0
   n = 1
   ```

   Ensuite, le programme est terminé.

3 Lecture et écriture sur des descripteurs de fichiers

Vous êtes en train d'écrire le programme d'une commande `mycat`, qui prend comme paramètre un nom de fichier, et qui envoie le contenu de ce fichier vers la sortie standard. On rappelle d'ailleurs que que `STDOUT_FILENO` *est le descripteur de fichier de la sortie standard.* Il ne vous reste plus que deux parties à compléter, marquées par le commentaire `// À FAIRE` au niveau des lignes 35 et 39.

```
1  void print_error(const string &msg = "") {
2      if(msg == "") cerr << "Erreur~: " << strerror(errno) << endl;
3      else cerr << "Erreur, " << msg << "~: " << strerror(errno) << endl;
4  }
5
6  void exit_error(const string &msg = "") {
7      print_error(msg);
8      exit(EXIT_FAILURE);
9  }
10
11 void exit_usage(const string &name) {
12     cerr << "Usage~: " << name << " fichier" << endl;
13     cerr << "Envoie le contenu de fichier vers la sortie standard" << endl;
14     exit(EXIT_FAILURE);
15 }
16
17 int main(int argc, char *argv[]) {
18     int fd; // descripteur de fichier
19     char buf[16]; // servira de buffer
20
21     if(argc < 2) exit_usage(string(argv[0]));
```

```
22
23      fd = open(argv[1], O_RDONLY);
24      if(fd == -1) exit_error("ouverture du fichier");
25
26      int nbrd, nbwr, nbrem;
27
28      do {
29          nbrem = nbrd = read(fd, buf, 16);
30          // sauf en cas d'erreur, ici :
31          // * nbrd est le nombre d'octets lus,
32          // * nbrem est le nombre d'octets restant à écrire,
33          // dans l'itération courante de la boucle do-while.
34
35          // À FAIRE
36
37          char *p = buf;
38          while(nbrem > 0) {
39              // À FAIRE
40          }
41
42      } while(nbrd > 0);
43
44      close(fd);
45
46      return EXIT_SUCCESS;
47  }
```

1. Quelle valeur retourne la primitive `read()` en cas d'erreur ? Donnez le code à insérer à la ligne 35 pour détecter l'erreur, afficher un message d'erreur, fermer le descripteur de fichier ouvert, et terminer le programme avec un code de retour adapté. *Indication : notre solution fait 4 lignes*

   Solution:

   ```
   if(nbrd == -1) {
     print_error("lors de la lecture");
     close(fd);
     return EXIT_FAILURE;
   }
   ```

2. Vous avez décidé d'utiliser un buffer de 16 octets que vous avez déclaré avec `char buf[16];` à la ligne 19. Entre les lignes 28 et 42, vous avez commencé à écrire une boucle `do-while` de lecture dans le fichier d'entrée. Vous devez maintenant écrire le code à insérer au niveau de la ligne 39 de façon à ce que les octets placés dans le buffer à chaque itération de la boucle `do-while` soient envoyés vers la sortie standard. N'oubliez pas de gérer convenablement les éventuelles erreurs d'écriture. *Indication : notre solution fait 7 lignes.*

   Solution:

   ```
   char *p = buf;
   while(nbrem > 0) {
       // nbrem est ici le nombre d'octets du buffer restant à écrire.
       // p pointe vers le 1er octet du buffer à écrire.
       nbwr = write(STDOUT_FILENO, p, nbrem);
       // sauf en cas d'erreur, nbwr est ici le nombre d'octets qui
       // viennent d'être écrits.
       if(nbwr == -1) {
         print_error("lors de l'écriture");
         close(fd);
         return EXIT_FAILURE;
       }
       // on met à jour nbrem et le pointeur p.
       nbrem -= nbwr;
       p += nbwr;
   }
   ```

3. Que retourne le `read(fd, buf, 16)` de la ligne 29 lorsque tout le contenu du fichier d'entrée a été lu ? Dans ce cas, qu'est-ce qui est écrit sur la sortie standard, et comment le programme sort-il de la boucle `do-while` des lignes 28 à 42 ? *Une explication en langue naturelle est attendue, il est inutile de recopier le code ici.*

   Solution: `read(fd, buf, 16)` retourne 0 lorsqu'il n'y a plus rien à lire sur le descripteur de fcihier `fd`. Donc dans ce cas `nbrem` est nulle : le programme n'entre pas dans la boucle `while(nbrem > 0) ...` et n'écrit donc rien sur la sortie standard. Comme la variable `nbrd` est nulle aussi, le programme sort de la boucle `do-while` car la condition `nbrd > 0` est désormais fausse.

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.