Examen de LIFASR5, le 28/06/2019 (90 minutes)

Toutes les communications et tous les documents sont interdits. Seule l'antisèche fournie est autorisée. Vous devez répondre lisiblement sur votre copie, indenter votre code, et éviter le plus possible les ratures. Le correcteur ne passera pas de temps sur les réponses confuses et leur attribuera 0.

Exercice 1 (4 pts). On vous demande d'écrire le code d'un serveur TCP, acceptant des connexions sur le port 8080, et permettant de gérer le dialogue avec **plusieurs clients simultanément**. Vous disposez déjà d'une fonction `void dial(int sd);` permettant de gérer les échanges avec un client, via la socket de dialogue `sd`. Vous devez donner la fonction `void main(void);` du serveur, en utilisant les appels systèmes adéquats, ainsi que les fonctions habituelles de la `socklib` que nous avons utilisées en TP :

- `int create_server_socket(const char* port);`
- `int accept_connection(int s);`

Dans cet exercice :

- le processus père n'a pas à se préoccuper de la mort de ses fils,
- vous ne vous préoccuperez pas de la gestion des erreurs, ni de la fin du serveur.

Par contre, vous veillerez à ce que les descripteurs de fichiers soient fermés au bon moment.

Solution: J'attends quelque chose du genre :

```
int main(void) {
  int se = create_server_socket("8080");
  for(;;) {
    int sd = accept_connection(se);
    if(fork() == 0) {
      close(se);
      dial(sd);
      close(se);
      return EXIT_SUCCESS;
    }
    close(sd);
  }
  return EXIT_SUCCESS;
}
```

Exercice 2 (6 pts).

1. (3 pts) Comment écrire un (potentiellement long) tableau de `n` caractères ($n > 0$) de type `char *` via un descripteur de fichier `fd` (sur un disque dur par exemple) ? Donnez une fonction `int writech0(int fd, const char *v, int n);` pour cela, en écrivant un-à-un les caractères sur `fd`. Votre fonction doit retourner 0 en cas de succès, -1 en cas d'échec.

   Solution: On va au plus simple :

   ```
   int writech0(int fd, const char *v, int n) {
     int nbwr, rem = n;
     const char *p = v;

     while(rem > 0) {
      nbwr = write(fd, p, 1);
      if(nbwr == -1) return -1;
      rem -= nbwr;
      p++;
     }
     return 0;
   }
   ```

   Dans ce cas, l'appel `write(fd, p, 1)` retourne soit 1 (on avance d'un caractère), soit 0 (on n'avance pas), soit -1 (il y a une erreur, que l'on ne gère pas pour l'instant).

2. (1 pt) Donnez un exemple d'appel de votre fonction, pour envoyer le contenu d'un tableau de 4 caractères sur la sortie standard.

   Solution:

   ```
   char t[] = "toto";
   if(writech0(STDOUT_FILENO, t, strlen(t)-1) == -1) return EXIT_FAILURE;
   ```

3. (2 pts) Vous devez maintenant écrire une fonction

   `int writech(int fd, const char *v, int n, int bs);`

   qui joue le même rôle que `writech0()`, mais dans laquelle les caractères sont écrits sur `fd` avec `write()` par bloc de `bs` caractères (`bs` > 0).

Solution:

```
int writech(int fd, const char *v, int n, int bs) {
  int nbwr, rem = n;
  const char *p = v;

  while(rem >= bs) {
   nbwr = write(fd, p, bs);
   if(nbwr == -1) return -1;
   rem -= nbwr;
   p += nbwr;
  }
  while(rem > 0) {
   nbwr = write(fd, p, rem);
   if(nbwr == -1) return -1;
   rem -= nbwr;
   p += nbwr;
  }
  return 0;
}
```

Exercice 3 (10 pts). Vous devez écrire un programme qui se duplique pour lancer une commande externe : le fils doit exécuter le programme `gaston`. Or ce programme (`gaston`) a tendance à s'arrêter en s'envoyant lui même le signal `SIGSTOP`. Votre programme doit donc guetter l'arrêt de son fils pour le relancer en lui envoyant le signal `SIGCONT`. Vous allez utiliser le fait que lorsqu'un processus est stoppé par le signal `SIGSTOP`, son père est prévenu par le signal `SIGCHLD` (comme lorsqu'il se termine). De plus, la primitive `waitpid`, grâce à l'option `WUNTRACED` (expliquée dans l'antisèche), permet de savoir si le fils a reçu le signal `SIGSTOP`. Votre programme doit prendre en compte les points suivants :

- Le processus principal doit se dupliquer avec `fork()`.
- Le fils créé doit être remplacé à l'aide de `execl()` (voir dans l'antisèche), par le programme `./gaston`.
- Tant que le fils travaille, le père reste bloqué en attente de signaux avec l'appel système `pause()`.
- Le père doit prendre en compte les arrêts de son fils en gérant le signal `SIGCHLD` : quand le fils a été stoppé (et uniquement dans ce cas), le père doit le réveiller en lui envoyant le signal `SIGCONT` avec `kill()` (voir l'antisèche) ; quand le fils s'est terminé, le père doit se terminer aussi. Pour que le PID du fils soit utilisable dans le gestionnaire de signal, vous utiliserez une variable globale `pid_t pid`, initialisée dans le programme à 0, mais qui prendra pour valeur le PID dès qu'il aura été crée.
- Le programme doit se terminer proprement lorsque le père reçoit le signal `SIGINT` : pour cela, le gestionnaire de signaux doit faire passer à 1 une variable globale `int stop` initialisée à 0. Le père doit aussi envoyer le signal `SIGQUIT` à son fils si celui-ci n'est pas déjà terminé.

Voici le squelette du programme que vous devez écrire :

```
int stop = 0;
pid_t pid = 0

void handler(int sig) {
  // à compléter
}

int main(void) {
  // à compléter
  return EXIT_SUCCESS;
}
```

En cas d'erreur, faites appel à la fonction `void exit_error(string msg = "")` qui affiche un message d'erreur optionnel, et termine le processus appelant.

Solution: Le handler peut ressembler à :

```
void handler(int sig) {
 if(sig == SIGCHLD) {
  if(pid != 0) {
   int stat;
   if(waitpid(pid, &stat, WUNTRACED|WNOHANG) == -1) exit_error("waitpid");

   if(WIFSTOPPED(stat)) {
    if(kill(pid, SIGCONT) != 0) exit_error("kill");
```

```
    }
    else if(WIFEXITED(stat)) { // ici, il faudrait affiner, mais ce serait déjà bien à l'écrit...
      stop = 1;
    }
    else exit_error("SIGCHLD raison inconnue");
  }
}
else if(sig == SIGINT) {
  if(pid != 0) {
    if(kill(pid, SIGQUIT) != 0) exit_error("kill");
  }
  stop = 1;
}
else exit_error("signal inconnu");
}
```

1. (5pts) Écrivez la fonction `handler()` pour la gestion des signaux : vous installerez plus tard cette fonction pour la gestion de `SIGCHLD` et de `SIGINT`. Vous devez suivre les consignes données précédemment. Indications et barème :
   - Dans le cas de `SIGCHLD` (0,5 pt) :
     - `waitpid()` n'est appelé que si le fils existe bien (0,5 pt),
     - `waitpid()` est utilisé correctement pour obtenir le status du fils (0,5 pt),
     - si le fils s'est stoppé, on lui envoie le signal `SIGCONT` (0,5 pt),
     - si le fils est terminé, le père doit se terminer (0,5 pt).
   - Dans le cas de `SIGINT` (0,5 pt) :
     - si le fils n'est pas déjà terminé, on lui envoie le signal `SIGQUIT` (0,5 pt),
     - le père doit se terminer (0,5 pt).
   - Gestion des erreurs de façon à faciliter le débugage (0,5 pt).
   - Code (non vide...) propre et bien indenté (0,5 pt).
2. (5 pts) Ecrivez la fonction principale `main()`, de façon à ce que le programme se comporte comme indiqué précédemment. Indications et barème :
   - Appel correct de `fork()` et distinction des cas (0,5 pt) ;
   - Dans le processus fils :
     - appel correct à `execl()` pour lancer `./gaston` (0,5 pt),
     - gestion pertinente du cas d'erreur (0,5 pt).
   - Dans le processus père :
     - installation du gestionnaire pour `SIGCHLD` et `SIGINT` (1 pt),
     - boucle d'attente des signaux (1 pt),
     - terminaison correcte du processus (0,5 pt).
   - Gestion des erreurs de façon à faciliter le débugage (0,5 pt).
   - Code (non vide...) propre et bien indenté (0,5 pt).

Solution: Cela doit ressembler plus ou moins à :

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

    execl("./gaston", "./gaston", NULL);
    exit_error();
  }

  if(sigaction(SIGCHLD, NULL, &sa) != 0) exit_error();
  sa.sa_handler = handler;
  if(sigaction(SIGCHLD, &sa, NULL) != 0) exit_error();
  if(sigaction(SIGINT, NULL, &sa) != 0) exit_error();
  sa.sa_handler = handler;
  if(sigaction(SIGINT, &sa, NULL) != 0) exit_error();

  // processus père
  while(1) {
    pause();
    if(stop) break;
  }

  return EXIT_SUCCESS;
}
```

Antisèche :

==========

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.

==========

COMPLEMENT POUR pid_t waitpid(pid_t pid, int *status, int options);
L'option WNOHANG permet de sortir de la fonction avec une erreur si aucun fils n'est terminé au lieu d'attendre la fin d'un fils . L'option WUNTRACED permet de sortir aussi lorsque les fils sont stoppés. Si status n'est pas NULL, des informations sur la terminaison sont stockées dans la variable pointée. Pour lire ces informations, vous devez utiliser des macros :
WIFEXITED(status) renvoie 1 si le fils s'est terminé par exit (), 0 sinon.
WIFSIGNALED(status) renvoie 1 si le fils a été terminé par un signal, 0 sinon.
WIFSTOPPED(status) renvoie 1 si le fils a été stoppé par un signal, 0 sinon.
WEXITSTATUS(status) renvoie la valeur passée àl'appel exit() s'il a eu lieu .
WTERMSIG(status) renvoie le numéro du signal si le processus à été signalé.

==========

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 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.

==========

UTILISATION DE int kill(pid_t pid, int sig);
L'appel système kill() peut être utilisé pour envoyer n'importe quel signal à n'importe quel processus ou groupe de processus.
Si pid > 0, le signal sig est envoyé au processus dont l'identifiant est indiqué par pid.
Si pid = 0, alors sig est envoyé à tous les processus appartenant au même groupe que le processus appelant.
En cas de réussite (au moins un signal a été envoyé) 0 est renvoyé ; en cas d'échec -1 est renvoyé et errno contient le code d'erreur.

==========

UTILISATION DE int execl(const char *path, const char *arg, ...);
execl() remplace le processus courant par le processus contenu dans le fichier path. Les arguments du nouveau processus sont contenus dans la liste d'arguments variable arg. Le dernier argument de la fonction doit être NULL pour signaler la fin de la liste d'argument. Par exemple :
execl("/bin/bash", "/bin/bash", NULL), lance un bash sans argument (à part le nom du fichier exécutable).
execl("/bin/bash", "/bin/bash", "-i", NULL), lance un bash avec l'argument -i.
L'appel execl() fait partie de la famille des appels exec(), et donc retourne -1 en cas d'erreur, mais ne retourne jamais en cas de succès.