+65/1/44+

LIFASR5 Contrôle

Contrôle du mardi 12/04/23 - 45 minutes

Nom : D'Oeuf

Prénom : John

No. étu. : p0123456 = 10123456

Numéro d'étudiant :

☐0	☒0	☐0	☐0	☐0	☐0	☐0	☐0
☒1	☐1	☒1	☐1	☐1	☐1	☐1	☐1
☐2	☐2	☐2	☒2	☐2	☐2	☐2	☐2
☐3	☐3	☐3	☐3	☒3	☐3	☐3	☐3
☐4	☐4	☐4	☐4	☐4	☒4	☐4	☐4
☐5	☐5	☐5	☐5	☐5	☐5	☒5	☐5
☐6	☐6	☐6	☐6	☐6	☐6	☐6	☒6
☐7	☐7	☐7	☐7	☐7	☐7	☐7	☐7
☐8	☐8	☐8	☐8	☐8	☐8	☐8	☐8
☐9	☐9	☐9	☐9	☐9	☐9	☐9	☐9

Utilisez un stylo noir (pas au crayon de bois), et répondez uniquement dans les cadres prévus à cet effet.

Aucun document n'est autorisé. Les téléphones, ordinateurs, et toutes communication avec les autres étudiants sont interdits. Seule l'antisèche fournie est autorisée.

Dans tout le sujet, les programmes sont donnés sans les `includes` et les `using namespace` : on suppose que l'on saurait les ajouter pour compiler, mais on ne s'en préoccupe pas ici.

1 Autour des pipes

Sujet A

Question 1 On considère le programme ci-dessous.

```
1  int main(void) {
2    int p[2];
3    pipe(p); // p[0] pour la lecture, p[1] pour l'écriture
4    if(fork() == 0) { // processus fils
5      close(p[0]);
6      for(char c = '0'; c < '9'; c++)
7        write(p[1], &c, 1);
8    }
9    else { // processus père
10     char c;
11     while(read(p[0], &c, 1) == 1)
12       cout << "(père) je lis " << c << endl << flush;
13     close(p[0]);
14     wait(NULL);
15   }
16   return 0;
17 }
```

A l'exécution, ce programme fonctionne normalement en affichant les chiffres de 0 à 9, mais il ne rend jamais la main dans le shell dans lequel on l'a lancé. Vous devez expliquer précisément pourquoi. On rappelle la règle "tout descripteur de fichier ouvert doit être fermé dès que possible" : votre explication doit donc être plus précise qu'un simple rappel de cette règle.

☐0 ☐1 ☐2 ☐3

Le père ne ferme pas de son côté l'entrée en écriture du pipe (p[1]) avant de le mettre à lire dedans. Ainsi, même quand le fils est terminé (plus précisément, il reste à l'état zombie), il reste un écrivain sur le pipe, et donc le père reste bloqué en attente de lecture à la ligne 11.

2 Autour des signaux

Question 2 On considère le programme ci-dessous, qui compte le nombre de nombres premiers entre 2 et l'entier naturel `n` défini dans le code.

```
1  unsigned int n = 1000000000;
2
3  bool stop = false;
4
5  int main(void) {
6    // TODO 1
7    unsigned int i, j, k = 0;
8    for(i = 2; i <= n; i++) {
9      for(j = 2; j < i; j++)
10       if(i % j == 0) break;
11     if(j == i) k++; // i est un nombre premier
12     // TODO 2
13   }
14   std::cout << "phi(" << i-1 << ") = " << k << endl << flush;
15   return EXIT_SUCCESS;
16 }
```

Comme le calcul (assez naïf) jusqu'à 10^9 risque d'être assez long, on souhaite pouvoir arrêter l'exécution du programme en cours de route, et afficher le résultat partiel obtenu. Vous devez pour cela mettre en place un gestionnaire pour le signal `SIGUSR1`. Commencez par écrire une fonction nommée `sigusr1_handler`, que vous utiliserez à la question suivante avec `sigaction()` pour installer votre gestionnaire de signal, et qui fait simplement passer la variable globale `stop` à `true`.

☐0 ☐1 ☐2 ☐3

```
void sigusr1_handler(int t){
    stop = true;
}
```

Question 3 Donnez le code que vous devez ajouter au niveau du commentaire `TODO 1` pour que `sigusr1_handler()` soit appelée lors de la réception de `SIGUSR1`. Donnez également le code à ajouter au niveau du `TODO 2` pour que, après la réception de `SIGUSR1`, le programme se termine en affichant le résultat intermédiaire du calcul. ☐0 ☐1 ☐2 ☐3

```
TODO 1:  struct sigaction sa;
         sigaction(SIGUSR1, NULL, &sa);
         sa.sa_handler = sigusr1_handler;
         sigaction(SIGUSR1, &sa, NULL);

TODO 2 : if(stop) break;
```

Question 4 Supposons que vous ayez lancé le programme sur votre système Linux, et que son PID est 1984. Si vous souhaitez l'arrêter en cours de cours en connaissant le résultat intermédiaire, quelle commande shell pouvez-vous utiliser ?

☐0 ☐1 ☐2 ☐3

```
kill -USR1 1984
```

3 Autre thème

Question 5 On considère le programme ci-dessous, qui est compilé en un exécutable nommé `./sys` (c'est ce nom qui sera utilisé par la suite).

```
1 int main(void) {
2   system("ps -l");
3   cout << "La commande est terminée, je me termine !" << endl;
4   return 0;
5 }
```

La fonction `system()` est utilisée pour lancer la commande `ps -l`. Voici un extrait de la page de manuel de cette fonction :

```
1 SYNOPSIS: int system(const char *command);
2 DESCRIPTION: The system() library function uses fork() to create a child process that executes the shell
3   command specified in command using execl() as follows: execl("/bin/sh", "sh", "-c", command, (char *) 0);
4   system() returns after the command has been completed.
```

Lors d'une exécution, ce programme donne le résultat suivant :

```
1 UID        PID  PPID  C STIME TTY          TIME CMD
2 nlouvet   5253  2532  0 09:49 pts/1    00:00:00 bash
3 nlouvet   6001  5253  0 10:23 pts/1    00:00:00 ./sys
4 nlouvet   6002  6001  0 10:23 pts/1    00:00:00 sh -c ps -l
5 nlouvet   6003  6002  0 10:23 pts/1    00:00:00 ps -l
6 La commande est terminée, je me termine !
```

La commande `ps -l` est une commande que l'on peut utiliser dans un shell, pour lister les processus en cours d'exécution dans le terminal. La commande `sh` est un shell comme le `bash` (aux fonctionnalités un peu réduites). On voit dans l'affiche produit par le `ps -l` du programme, et on comprend d'après l'antisèche, que l'appel `system("ps -l")` provoque l'exécution de `sh -c ps -l` : que fait cette commande ? ☐0 ☐1 ☐2 ☐3

On voit que la commande "ps -l" (PID 6003) est fille de la commande "sh -c ps-l" (PID 6002). On en déduit que cette dernière commande lance un shell sh, qui crée un fils qui est recouvert (appel exec()) par "ps -l".

Question 6 En notant `p1 -> p2` pour indiquer que le processus `p2` est fils du processus `p1`, indiquez : quel processus est le fils du processus `bash`, celui de `./sys`, et celui de `sh -c ps -l` ? ☐0 ☐1 ☐2 ☐3

bash (5253) ⟶ ./sys (6001) ⟶ sh -c ps-l (6002) ⟶ ps-l (6003)

Question 7 Expliquez comment se déroule l'appel `system("ps -f")` : quels sont les appels systèmes utilisés, quelle primitive système est utilisée pour que l'appel à `system()` ne se termine qu'après que la commande passée en paramètre soit terminée ? (Il ne vous est pas demandé de coder la fonction). ☐0 ☐1 ☐2 ☐3

L'appel system("ps...") crée un fils avec fork() : par un appel à une primitive de la famille exec(), ce fils se transforme (se recouvre) en "sh -c ps-l". Ce shell crée lui même un fils, qui exécute (enfin !) "ps-l" par recouvrement (à nouveau exec()). La primitive waitpid() est utilisée par chaque père pour attendre la terminaison de son fils.

+65/4/41+

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 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);
    /* the rest is useless */
};
```