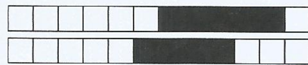


+62/1/56+

LIFASR5 Contrôle

Contrôle du mardi 12/04/23 - 45 minutes

Nom : L'Eponge

Prénom : Bob

No. étu. : p01210012 = 10121012

Numéro d'étudiant :

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 B

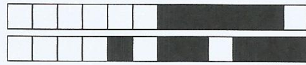
Question 1 On considère le programme ci-dessous.

```
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 père
5      close(p[0]);
6      for(char c = '0'; c < '9'; c++)
7        write(p[1], &c, 1);
8      wait(NULL);
9    }
10   else { // processus fils
11     close(p[1]);
12     char c;
13     while(read(p[0], &c, 1) == 1)
14       cout << "(fils) je lis " << c << endl << flush;
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 jamais p[1], donc il reste toujours un écrivain sur le pipe. Le fils (même après avoir lu toutes les données envoyée par le père) reste donc bloqué en lecture sur p[0] à la ligne 13, et ne se termine jamais. Le père quant à lui attend la terminaison de son fils à la ligne 8, mais cela n'arrivera jamais !



2 Autour des signaux

Question 2 On considère le programme à compléter ci-dessous, qui doit afficher les nombres premiers entre 2 et l'entier naturel n défini dans le code.

```
 1 int main(void) {
 2   unsigned int n = 100000;
 3   int rf = fork();
 4   if(rf == 0) { // processus fils
 5     // TODO 1f
 6     unsigned int i, j;
 7     for(i = 2; i <= n; i++) {
 8       for(j = 2; j < i; j++) if(i % j == 0) break;
 9       if(j == i) cout << i << " " << flush; // i est un nombre premier
10     }
11   }
12   else { // processus père
13     // TODO 1p
14     // TODO 2
15   }
16   return EXIT_SUCCESS;
17 }
```

Comme le calcul risque d'occuper beaucoup le CPU, le processus père va stopper régulièrement l'exécution de son fils, pour ne lui laisser continuer son exécution qu'à des intervalles de temps espacés. Mais le père et le fils doivent commencer par se synchroniser : donnez le code à ajouter au niveau du `TODO 1p` pour que le père stoppe sa propre exécution en s'envoyant le signal `SIGSTOP`, puis le code à ajouter au niveau du `TODO 1f` pour que le fils fasse reprendre l'exécution du père en lui envoyant `SIGCONT`.

☐0 ☐1 ☐2 ☐3

TODO1p : kill(getpid(), SIGSTOP);

TODO1f : kill(getppid(), SIGCONT);

Question 3 Quand le père a repris son exécution, il doit entrer dans une boucle : il stoppe l'exécution de son fils avec le signal `SIGSTOP`, s'endort pendant 9000μs, fait reprendre l'exécution de son fils avec `SIGCONT`, s'endort pendant 1000μs, puis reprend au début de boucle (pour endormir le père pendant tμs, utilisez `usleep(t)`). De plus, à chaque itération, le père teste si son fils est terminé l'aide de `waitpid(p, NULL, WNOHANG)` : cet appel non-bloquant retourne `p` si le fils de PID `p` est terminé, une valeur différente de `p` sinon. Quand le fils est terminé, le père se termine aussi en affichant `calcul terminé`. Donnez le code à ajouter au niveau du `TODO2` pour réaliser tout cela.

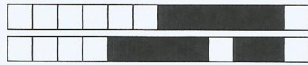
☐0 ☐1 ☐2 ☐3

```
do {
   kill(rf, SIGSTOP);
   usleep(9000);
   kill(rf, SIGCONT);
   usleep(1000);
} while (waitpid(rf, NULL, WNOHANG) != rf);
cout << "calcul terminé" << endl;
```

Question 4 En supposant que le fils ne s'exécute que sur un seul CPU, quelle est au maximum la proportion de temps CPU consommée par le processus fils ? Justifiez, et donnez un résultat en pourcents.

☐0 ☐1 ☐2 ☐3

Sur 9000µs + 1000 µs, le fils ne s'exécute que 1000µs, et "dort" le reste du temps. La proportion du temps où il s'exécute est donc : $\frac{1000\mu s}{1000\mu s + 9000\mu s} = 10\%$.



3 Autre thème

Question 5 Dans le programme suivant, le processus principal (père) crée ~~5~~ 3 processus fils, à des intervalles de temps aléatoires. Ces fils se mettent en sommeil pour une durée aléatoire.

```
1  int k = 0;
2
3  void handler(int s) {
4    cout << "père : SIGCHLD " << endl;
5    waitpid(-1, NULL, 0);
6    k++;
7  }
8
9  int main(void) {
10   struct sigaction act;
11   sigaction(SIGCHLD, NULL, &act);
12   act.sa_handler = handler;
13   sigaction(SIGCHLD, &act, NULL);
14   for(int i = 0; i < 3; i++) {
15     sleep(rand()%10); // sommeil d'une durée maximale de 9 secondes
16     if(fork() == 0) { // processus fils
17       cout << "fils " << i << endl;
18       sleep(rand()%10); // sommeil d'une durée maximale de 9 secondes
19       return 0;
20     }
21   }
22   do {
23     pause(); // mise en attente d'un signal
24   } while(k < 5); 3
25   cout << "père : je me termine" << endl;
26   return 0;
27 }
```

(ligne 24 : 5 barré à la main et remplacé par 3)

Rappelez quel est le signal envoyé au processus père lors de la terminaison de l'un de ses fils, et expliquez comment ce signal est géré par le processus père dans ce programme.

☐0 ☐1 ☐2 ☐3

SIGCHLD : à la réception de ce signal, la fonction handler() est appelée pour prend en compte la fin du fils.

Question 6 Expliquez pourquoi, dans ce programme, le processus père attend bien la terminaison de tous ses fils avant de se terminer lui-même.

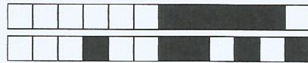
☐0 ☐1 ☐2 ☐3

Il se met en attente de la terminaison de tous ses fils avec la boucle des lignes 22 à 24 : il se remet en pause tant qu'il n'a pas reçu 3 fois SIGCHLD, c'est-à-dire tant que k<3.

Question 7 Donnez deux exemples différents d'affichages possibles produits par le programme.

☐0 ☐1 ☐2 ☐3

Exemple 1	Exemple 2
fils 0	fils 0
fils 1	fils 1
fils 2	père : SIGCHLD
père : SIGCHLD	fils 2
père : ———	père : SIGCHLD
père : ———	père : ———
père : je me termine	père : je me termine



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 */
};
```