BROILLET VIRGILE
Note: 10/20 (score total : 15/30)

+13/1/48+

LIFSE ECA Session 2

Contrôle du mardi 04/07/23 - 60 minutes

Nom : BROILLET

Prénom : Virgile

No. étu. : 12103804

Numéro d'étudiant :

0	0	0	■	0	0	■	0
■	1	■	1	1	1	1	1
2	■	2	2	2	2	2	2
3	3	3	3	■	3	3	3
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
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 autorisé ; téléphones, ordinateurs, communications interdits.

Question 1 On considère l'extrait de programme suivant, dans lequel tous les appels à `fork()` réussissent ; combien d'étoiles sont affichées à l'exécution du programme ?

```
1  int main(void) {
2    if(fork() > 0) {
3      fork();
4    }
5    fork();
6    std::cout << "*" << std::endl;
7    return 0;
8  }
```

0/1

☐ 2 ☒ 6 ■ 4 ☐ 9 ☐ 5 ☐ 8 ☐ 3 ☐ 7

Question 2 On considère le code ci-dessous dans lequel un processus père fait notamment un `fork()` pour créer un fils.

```
1  int main(int argc, char *argv[]) {
2    int a = 4;
3    if (fork() > 0) { // processus père
4      a = 2;
5      cout << a << endl;
6    }
7    else { // processus fils
8      sleep(1);
9      cout << a << endl;
10   }
11   return EXIT_SUCCESS;
12 }
```

Quelle est la valeur affichée pour `a` pour le processus fils ? Pourquoi ?

☐0 ☐1 ☐2 ■3 3/3

La valeur affichée pour le processus fils est 4 car les différents processus ont chacun leur mémoire.

1 Transfert de données à l'aide d'un tube

On considère le programme suivant, dans lequel le processus principal crée un pipe (tube), puis deux processus fils, `fils1` et `fils2`. Ce programme utilise la primitive `dup2()`, que vous ne connaissez pas forcément ; mais dans ce

programme, cette primitive est utilisée deux fois, pour que :

- `fils1` remplace sa sortie standard par `tube[1]` (tout ce qu'il écrit sur sa sortie est envoyé dans le tube) ;
- `fils2` remplace son entrée standard par `tube[0]` (tout ce qu'il lit sur son entrée est lu depuis le tube).

```
1  int main(void) {
2    int tube[2];
3    pipe(tube); // tube[0] pour la lecture, tube[1] pour l'écriture
4
5    int pid1 = fork();
6    if(pid1 == 0) { // processus fils1
7      cout << "fils1" << endl;
8      dup2(tube[1], STDOUT_FILENO); // STDOUT_FILENO devient l'entrée tube[1]
9      close(tube[0]);
10     close(tube[1]);
11     cout << "L'araignée Gipsy" << endl;
12     cout << "Monte âla gouttière" << endl;
13     cout << "Tiens voilâ la pluie" << endl;
14     cout << "Gipsy tombe par terre" << endl;
15     cout << "Mais le soleil va chasser la pluie" << endl;
16     return 0;
17   }
18
19   int pid2 = fork();
20   if(pid2 == 0) { // processus fils2
21     cout << "fils2 ->";
22     dup2(tube[0], STDIN_FILENO); // STDIN_FILENO devient l'entrée tube[0]
23     close(tube[0]);
24     close(tube[1]);
25     execl("/usr/bin/wc", "/usr/bin/wc", "-l", (char*)NULL);
26     cout << "erreur" << endl;
27     return 1;
28   }
29
30   close(tube[0]);
31   close(tube[1]); // ici
32   waitpid(pid1, NULL, 0);
33   waitpid(pid2, NULL, 0);
34   return 0;
35 }
```

Question 3 Dans le programme, comment le processus `fils1` fait-il pour envoyer des caractères vers sa sortie standard (c'est-à-dire en fait vers `tube[1]`) ?

☐0 ☐1 ■2 ☐3 2/3

Grâce à dup2(tube[1], STDOUT_FILENO) qui remplace la sortie par tube[1]

Question 4 Dans le `fils2`, la commande `wc -l` (commande pour compter le nombre de lignes) est lancée à l'aide de la primitive de recouvrement `execl()`. A l'exécution, que va afficher `fils2` sur sa sortie standard, et pourquoi ?

■0 ☐1 ☐2 ☐3 0/3

Le fils2 affichera le message "erreur" car il a précédemment fermer tube[0] qui était devenu son entrée standard.

Question 5 Pourquoi le processus père fait-il deux appels à `waitpid()` ?

☐0 ☐1 ☐2 ■3 3/3

Le père attend la fermeture de ses 2 fils.

Question 6 Le programme s'exécute parfaitement sans erreur, et `fils2` n'affiche pas le message "erreur", ce qui indique qu'il ne passe jamais par la ligne `cout << "erreur" << endl`. Expliquez pourquoi.

☐0 ☐1 ■2 ☐3 2/3

Cela veut dire que ce fils2 connait exactement compter ce nombre de lignes.

Question 7 Si l'on supprime le `close(tube[1])` au niveau du commentaire "ici" (ligne 31) dans le processus père, alors le `fils2` reste bloqué indéfiniment en attente de lecture sur le tube : expliquez pourquoi ?

☐0 ☐1 ■2 ☐3 2/3

Le tube[1] d'écriture n'est alors jamais fermé ce fils 2 attend indéfiniment de nouvelle ligne pour se mettre à jour. Il reste ici [illegible] zombie

2 Un serveur

Question 8 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(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, et 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.

☐0 ☐1 ■2 ☐3 ☐4 ☐5 2/5

```
int main(void){
int s, sd;
s = create_server_socket("8080");
while(true){
    sd = accept_connection(s);
    dial(sd);
    close(sd);
}
return 0;
}
```

3 Pour finir...

Question 9 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);` qui écrit un-à-un les caractères sur `fd`. Votre fonction doit retourner 0 en cas de succès, -1 en cas d'échec.

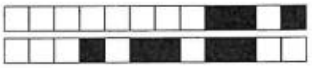
☐0 ■1 ☐2 ☐3 1/3

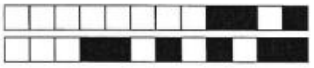
```
int writech0(int fd, const char *v, int n){
  int r; char * buf[n];
  for(int i=0; i<n; i++){ buf[i]=v;
     r = write(fd, buf, size_of-(char *));
     if(r==-1){return -1;}
  }
  return 0;
}
```

Question 10 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).

■0 ☐1 ☐2 ☐3 0/3

```
int writech(int fd, const char * v, int n, int bs){
   int r;
   while( bs != 0){
   : if(writech0(fd, v, n)==0){ n--;}
   : else{ return -1;}
   :}
   if(write(fd, '\n', n)==0){return 0;}
   else{ return -1;}
}
```





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 close – close a file descriptor
SYNOPSIS int close(int fd);
DESCRIPTION
close() closes a file descriptor (for a regular file , a pipe or a socket), so that it no longer refers to any file and may be reused.
RETURN VALUE
close() returns zero on success. On error, –1 is returned, and errno is set appropriately.

==========

NAME recv – receive a message from a socket
SYNOPSIS ssize_t recv(int sockfd, void *buf, size_t len, int flags);
DESCRIPTION
The recv() call is used to receive messages from a socket. It may be used to receive data on connection–oriented sockets. The only difference between recv() and read() is the presence of flags . With a zero flags argument, recv() is generally equivalent to read().

==========

NAME send – send a message on a socket
SYNOPSIS ssize_t send(int sockfd, const void *buf, size_t len, int flags);
DESCRIPTION
The system call send() is used to transmit a message to another socket. The send() call may be used only when the socket is in a connected state (so that the intended recipient is known). The only difference between send() and write() is the presence of flags . With a zero flags argument, send() is equivalent to write().

==========

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 PID 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 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.