BROILLET VIRGILE
Note: 6.5/20 (score total : 9/28)

+62/1/54+

LIFSE ECA

Contrôle du mardi 25/05/23 - 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   fork();
3   fork();
4   fork();
5   std::cout << "*" << std::endl;
6   return 0;
7 }
```

0/1

☐ 4 ☒ 8 ☐ 5 ■ 7 ☐ 9 ☐ 6 ☐ 3 ☐ 2

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 = 2;
3    int ret = fork();
4    if (ret == 0) { // processus fils
5      sleep(1);
6      cout << a << endl;
7    }
8    else { // processus père
9      a = 4;
10     cout << a << endl;
11   }
12   return EXIT_SUCCESS;
13 }
```

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

■ 0 ☐ 1 ☐ 2 ☐ 3

0/3

La valeur affichée pour a est 4 car il est pré-défini par le père à 4.

1 Lecture ligne par ligne

On considère le programme suivant qui lit les caractères sur son entrée standard, ligne par ligne (la fin d'une ligne est marquée par un `'\n'`), et les caractères lus sont rangés dans un tableau `buffer` de `char` de taille `BUFLEN` (`BUFLEN`

est une macro qui a une valeur entière strictement positive). Dès que `buffer` est rempli ou que `'\n'` est lu, la ligne courante est affichée sur la sortie standard. On suppose que le cas d'erreur de l'appel à `read()` ne se produit pas.

```
1  int main(void) {
2    char c, buffer[BUFLEN];
3    int r, nbrd;
4
5    do {
6      nbrd = 0;
7      while(nbrd < BUFLEN-1) {
8        r = read(STDIN_FILENO, &c, 1);
9        if(r == -1) {
10         cerr << "Une erreur de lecture s'est produite" << endl;
11         return 1;
12       }
13       buffer[nbrd] = c;
14       if((r == 0) || (c == '\n')) break;
15       else nbrd++;
16     }
17     buffer[nbrd] = '\0';
18     cout << "Vous avez entré : " << buffer << endl;
19   }
20   while(r != 0);
21   return 0;
22 }
```

Question 3 `STDIN_FILENO` est

1/1

- ☐ un flux de la classe `ostream`
- ☐ un pointeur de fichier de type `FILE*`
- ☒ un descripteur de fichier de type `int`

Question 4 À la ligne 17, `nbrd` a pour valeur

0/1

- ☒ `BUFLEN`
- ☒ le nombre de caractères disponibles dans `buffer`
- ☐ `BUFLEN-1`

Question 5 Le programme sort de la boucle des lignes 5 à 20 pour exécuter le `return 0` quand

1/1

- ☐ une ligne vide est entrée par l'utilisateur
- ☒ la fin de fichier est lue sur l'entrée standard
- ☐ la combinaison de touche `Ctrl+C` est utilisée

Question 6 Écrivez le code d'une fonction d'entête `int rdline(int sd, char[BUFLEN] buf)` pour recevoir ligne par ligne des caractères sur une socket de dialogue `sd`, en utilisant la primitive `recv()`. Lors d'un appel à cette fonction, les caractères reçus sur `sd` sont rangés dans le tableau `buf` passé en paramètre, jusqu'à ce que le tableau soit rempli ou qu'un retour à la ligne `'\n'` soit rencontré. L'éventuel caractère `'\n'` reçu ne doit pas être placé dans le tableau, et la chaîne doit se terminer par un `'\0'`. L'appel retourne le nombre caractères reçus (hors `'\0'`), 0 si la socket a été fermée pendant l'appel, et `-1` en cas d'échec.

☐0 ☒1 ☐2 ☐3 — 1/3

```
int rdline(int sd, char buf[BUFLEN]) {
  int r, nb; char c;
  do {
    nb = 0;
    r = recv(sd, buf+nb, BUFLEN-nb, 0);
    buf[BUFLEN-nb] = c;
    if(r == -1) { return -1; }
    else if(c == '\n') { buf[BUFLEN-nb] = '\0'; }
    else nb++;
  while(r != 0); return 0;
}
```

Question 7 Vous devez écrire un serveur TCP/IP, qui se mettra à l'écoute sur le port 9999, et recevra les clients successivement. Vous disposez de deux fonctions pour manipuler les sockets :

- `int create_server_socket(const char* port);`
  ↪ retourne une socket côté serveur, à l'écoute sur le port passé en paramètre.

- `int accept_connection(int s);`
  ↪ se met en attente bloquante d'une connexion sur la socket d'écoute s, et retourne la socket de dialogue créée.

Dès qu'un client se connectera, le serveur lira les caractères envoyés par le client grâce à la fonction `rdline()` de la question précédente, et affichera le texte reçu, ligne par ligne, sur sa sortie standard (celle du serveur). Quand le client fermera la connexion, le serveur la fermera également de son côté, puis se remettre en attente du client suivant

Écrivez ci-dessous le code de la fonction `main()` de votre serveur (ne prévoyez pas la terminaison de votre serveur, ne vous préoccupez pas des fichiers d'en-tête, ne définissez pas d'autres fonctions, ne vous préoccupez pas de la gestion des échecs).

0 1 2 3 1/3

```
int main(void){ int s, sd ; char *port = "9999";
  int CEN = 9999; char buf[CEN]; int i=0;
  s = create_server_socket(port);
  while(true){
    sd = accept_connection(s);
    Rdline(sd, buf[CEN]);
    do{
      cout << buf[i];
      if(buf[i] == '\n'){ cout << endl; }
      if(buf[i] == '\0'){ cout << "." << endl; }
    i++ }while(i != CEN); close(sd); } close(s);
return 0; }
```

2 Autour des tubes

On considère le programme suivant où le processus fils lit sur le tube et affiche les entiers lus jusqu'à ce qu'il ne lise plus rien.

```
1  int main(void) {
2
3    int tube[2];
4    pipe(tube); // tube[0] pour la lecture, tube[1] pour l'écriture
5    int pid = fork();
6
7    if (pid > 0) { // code du père
8    } else { // code du fils
9      int i;
10     while(read(tube[0], &i, sizeof(int)) == sizeof(int)){
11       cout << i << endl;
12     }
13     close(tube[0]);
14
15   }
16
17   return 0;
18 }
```

Question 8 On souhaite que le processus père écrive sur le tube les 20 premiers entiers naturels avant de se terminer. Donnez le code correspondant.

☐0 ☒1 ☐2 ☐3 1/3

```
int tube[2]; pipe(tube); int pid = fork(); int i=?;
if(pid > 0){// proc père
do{ write(tube[1], &i, sizeof(int);
    i++;
}while(i != 21);
} else {break;} // proc fils
close(tube[1]);} return 0;
```

Question 9 Après avoir écrit le code du père on s'aperçoit que le fils ne termine jamais. Pourquoi ? Précisez la correction à apporter.

☒0 ☐1 ☐2 ☐3 0/3

Le fils ne se termine jamais il faudrait lui mettre un break comme fait à la question 8.

Question 10 Que se passe t-il si un processus essaie d'écrire sur un tube alors que plus aucun descripteur en lecture sur ce tube n'est ouvert ?

☒0 ☐1 ☐2 ☐3 0/3

L'écriture est impossible, le processus ne se terminera jamais, il est à l'état "zombie".

3 Pour finir...

Question 11 On considère l'extrait de code ci-dessous dans lequel un processus père fait notamment un `fork()` pour créer un fils.

```
1  int main(void) {
2    int p = fork();
3    if(p == -1) exit_error("fork");
4    if(p == 0) {
5      sleep(10);
6      return 0;
7    }
8    sleep(10);
9    waitpid(p, NULL, 0);
10   return 0;
11 }
```

À l'exécution, quel va être le temps d'exécution (approximatif) de ce programme ?

0/1 ☐ 30s ☐ 15s ☐ 5s ☒ 10s ☒ 20s

Question 12 Pourquoi doit-on s'embêter avec des tubes (pipes) ou des sockets pour faire communiquer les processus entre-eux, même s'ils sont sur le même système ?

☐ pour éviter les échanges de données trop importants entre processus
1/1 ☒ parce qu'il n'y a pas (*a priori*) de mémoire partagée entre processus
☐ les solutions les plus compliquées sont toujours les meilleures

Question 13 Pour établir une connexion TCP/IP avec un serveur, il suffit au client de connaître

☒ l'adresse IP et le port auquel il doit se connecter
1/1 ☐ l'URL (Uniform Resource Locator) du serveur sur le web
☐ le PID du processus serveur

Question 14 La primitive `accept()` de l'API POSIX pour les sockets sert

☐ au client pour accepter la réponse d'un serveur
1/1 ☐ au client pour le connecter à un serveur
☒ au serveur pour se mettre en attente de nouveaux clients

Question 15 Dans un shell, lorsque l'on entre le chemin d'un programme exécutable sur la ligne de commande, ce processus shell

☒ crée un fils qui est chargé de lancer le programme avec une primitive de la famille `exec()`
0/1 ☒ appelle une primitive de la famille `exec()` pour exécuter lui-même le programme
☐ utilise la primitive `system()` pour exécuter le programme

Question 16 Observez le résultat de la commande ci-dessous :

```
1 nlouvet@ninjutsu:~$ ps -f
2 UID        PID  PPID  C STIME TTY          TIME CMD
3 nlouvet  11084  5576  0 16:12 pts/2    00:00:00 bash
4 nlouvet  11149 11084  0 16:13 pts/2    00:00:00 xclock
5 nlouvet  11150 11084  0 16:14 pts/2    00:00:00 bash
6 nlouvet  11194 11150  0 16:15 pts/2    00:00:00 xcalc
7 nlouvet  11275 11150  0 16:21 pts/2    00:00:00 ps -f
```

Quels processus sont des fils du `bash` d'identifiant 11084 ?

☐ `xclock` et `xcalc`
☐ `xcalc` et `ps`
1/1 ☒ `xclock` et `bash`
☐ `bash` et `ps`

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.