+1/1/60+

LIFSE Contrôle
Contrôle du mardi 21/03/23 - 45 minutes

Nom : Keaton
Prénom : Buster
No. étu. : 04101895

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 de read/write (8 points)

Vous devez écrire un programme qui formate un fichier contenant une chaine de caractères (multiple de 80) qui ne contient pas de saut de ligne (caractère \n) ni de caractère nul. L'objectif est d'afficher le texte à l'écran (sortie standard) 80 caractères par 80 caractères pour plus de lisibilité. On vous fournit un code qui lit le fichier caractère par caractère et qui affiche à l'écran un caractère par ligne en utilisant `std::cout`,

```
1  int main(int argc, char *argv[]) {
2    int fdin, nbrd;
3    char c;
4    if((fdin = open(argv[1], O_RDONLY)) < 0) { // Ouverture du fichier source dont le nom est argv[1]
5      cerr << "Erreur : " << strerror(errno) << endl;
6      return 1;
7    }
8    do {
9      nbrd = read(fdin, &c, 1);
10     if (nbrd > 0){
11       cout << c << endl;
12     }
13   } while(nbrd > 0);
14   close(fdin);
15   return EXIT_SUCCESS;
16 }
```

Question 1 On suppose que chaque appel à la fonction `read` permet de lire exactement le nombre d'octets passés en paramètre. Écrivez une nouvelle version du code (boucle lignes 8 à 13) permettant d'afficher les caractères 80 par 80.

□0 □1 □2 □3

```
char tab[81];
do {
  nbrd = read(fdin, tab, 80);
  if (nbrd > 0){
    tab[nbrd] = '\0';
    cout << tab << endl;
  }
}
```

Question 2 On se place désormais dans le comportement usuel de la fonction `read` que vous pouvez retrouver dans l'antisèche. Que faut-il faire pour s'assurer que l'on affiche bien les caractères par paquet de 80 (*i.e.*, que chaque appel à `cout << tab` affiche 80 caractères) ? Vous pouvez soit donner une explication soit le code.

☐0 ☐1 ☐2 ☐3

Il faut s'assurer que l'on a bien lu 80 caractères avant de lancer l'affichage. Il faut pour cela utiliser nbrd comme un compteur initialisé à 0, et auquel on ajoute la valeur de retour de read(fdin, tab+nbrd, 80-nbrd), tant que nbrd n'a pas atteint 80.

Question 3 On souhaite désormais afficher à l'écran en utilisant directement l'appel système `write()`. Quelles modifications devez-vous apporter à votre code ? Vous pouvez soit donner une explication soit le code.

☐0 ☐1 ☐2 ☐3

Une fois que l'on a lu 80 caractères à l'aide de la boucle de la question précédente, il faut mettre en place une boucle similaire pour les écritures. On initialise pour cela un compteur nbwr 0, et on lui ajoute la valeur de retour de write(STDOUT_FILENO, tab+nbwr, 80-nbwr) tant que nbwr n'atteint pas 80.

2 Autour des processus (8 points)

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

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

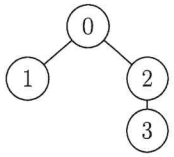
☐0 ☐1 ☐2 ☐3

L'affectation effectuée dans le père ne change pas la valeur de a dans le fils, car les processus ont chacuns leurs mémoires. Le fils conserve la valeur qu'avait a avant le fork(), et affiche donc 1.

Question 5 On considère un programme dans lequel deux processus sont créés, d'identifiants respectifs $\texttt{pid}_1$ et $\texttt{pid}_2$. Les deux processus effectuent des affichages dans le même terminal, sans retour à la ligne : le processus d'identifiant $\texttt{pid}_1$ affiche les entiers de 1 à 5, et celui d'identifiant $\texttt{pid}_2$ affiche les lettres de `a` à `e`. Donnez trois exemples de l'affichage que nous pourrions obtenir à l'exécution du programme ; pourquoi différents affichages sont possibles ?

☐0 ☐1 ☐2 ☐3

Différents affichages sont possibles car les deux processus affichent de façon concurrente, sans se synchroniser.
Trois exemples : 12ab34cd5e, 123abc45de,
12345abcde



Question 6 Donnez un programme permettant de créer l'arborescence de processus suivante:

☐0 ☐1 ☐2 ☐3

```
int main (void) {//proc. 0
  int pid1, pid2, pid3;
  if((pid1 = fork()) == 0) { //proc. 1
    return 0;
  }
  if( (pid2 = fork()) == 0) {//proc. 2
    if ((pid3 = fork()) == 0) {//proc. 3
      return 0;
    }
    return 0;
  }
  return 0;
}
```

Question 7
Comment le processus 0 doit-il procéder pour attendre ses deux fils ? Et s'il souhaite les attendre dans un ordre particulier (2 avant 1) ?

☐0 ☐1 ☐2 ☐3

Sans ordre, le processus 0 peut appeler deux fois wait(NULL), ou bien deux fois waitpid(-1, NULL, 0). Dans un ordre précis, il peut appeler waitpid(pid1, NULL, 0); ou l'inverse.
waitpid(pid2, NULL, 0);

Question 8
Quel est l'état du processus 1 s'il termine avant le processus 2 et que le père commence par attendre le processus 2 ? Pourquoi ?

☐0 ☐1 ☐2 ☐3

Entre l'instant où un processus se termine, et celui où son père prend en compte sa terminaison avec wait() ou waitpid(), il reste à l'état zombie.
C'est ce qui arrive au processus 1 ici.

3 Signaux (4 points)

Question 9
On souhaite qu'un programme affiche ``j'ai reçu un SIGUSR1'' lorsqu'il reçoit le signal SIGUSR1. Donnez un pseudo code du programme correspondant.

☐0 ☐1 ☐2 ☐3

* On définit une procédure void handle(int) pour afficher le message
* Dans le programme principal :
  - On installe handle comme gestionnaire pour le signal SIGUSR1, à l'aide de sigaction().
  - On met le programme en attente d'un signal.

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 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 */
};
```