+1/1/60+

LIFSE ECI

ECI du mardi 12/03/24 - 45 minutes

Nom : Hopper

Prénom : Grace

No. étu. : 17543210

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.

ATTENTION, CE NE SONT QUE DES ELEMENTS DE CORRECTION, IL PEUT RESTER DES ERREURS...

1 Autour de read/write (8 points)

On considère l'extrait de programme C/C++ suivant, qui permet de copier le contenu d'un fichier régulier `source` vers un fichier nommé `destination`.

```
1  const unsigned int LEN=256;
2  int nbrem, nbrd, nbwr;
3  int fdin, fdout;
4  char buf[LEN];
5
6  fdin = open("source", O_RDONLY);
7  fdout = open("destination", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);
8
9  do {
10   nbrem = nbrd = read(fdin, buf, LEN);
11   if(nbrd == -1) return EXIT_FAILURE;
12   while(nbrem > 0) {
13     nbwr += write(fdout, buf+nbrem, nbrem);
14     if(nbwr == -1) return EXIT_FAILURE;
15     nbrem -= nbwr;
16   }
17 } while(nbrd > 0);
18
19 close(fdin);
20 close(fdout);
```

nbwr = 0;

nbwr

= nbrd - nbwr;

Il y avait des erreurs dans le programme ci-contre. Une façon de les corriger est indiquée.
La notation a été adaptée pour prendre en compte ces erreurs, surtout à la question 6 qui était la plus affectée.

Question 1 Jusqu'à combien d'octets peuvent être lus par le `read()` de la ligne 10, et écrit par le `write()` de la ligne 13 ? ☐0 ☐1 ☐2 ☐3

Au plus LEN = 256 octets peuvent être lus par chaque appel à read() à la ligne 10. Si n octets sont lus à la ligne 10, alors l'appel à write() ligne 13 tente d'en écrire n : au plus n octets seront écris.

Question 2 Juste après l'exécution de la ligne 10, que représentent `nbrd` et `nbrem` ? ☐0 ☐1 ☐2 ☐3

nbrd est le nombre d'octets lus sur fdin au début de chaque itération de la boucle do ... while(nbrd>0) par le read() de la ligne 10. nbrem est le nombre d'octets restant ensuite à écrire sur fdout.

Question 3 À la ligne 13, avant l'appel à `write()`, que représente `buf+nbwr` ? ☐0 ☐1 ☐2 ☐3

Il s'agit de l'adresse en mémoire à laquelle write() doit aller chercher les octets qu'il doit tenter d'écrire sur le descripeur de fichier fdout.

ATTENTION, CE NE SONT QUE DES ELEMENTS DE CORRECTION, IL PEUT RESTER DES ERREURS...

Question 4 Comment se termine l'exécution si le fichier `source` n'existe pas ? ☐0 ☐1 ☐2 ☐3

Si le fichier source n'existe pas, alors open() retourne -1 à la ligne 6 (voir dans l'antisèche). Mais cette valeur de retour n'est pas testée, et le programme continue son exécution avec fdin = -1. Mais -1 n'est pas un descripteur de fichier valide : à la ligne 10, la lecture sur fdin avec read() échoue, et retour -1 (voir à nouveau l'antisèche). Comme désormais nbrd = -1, le programme se termine à la ligne 11 en retournant EXIT_FAILURE (c'est-à-dire 1).

Question 5 On suppose que le fichier `source` contient exactement 42 octets. Dans le meilleur des cas, combien de fois arrive t-on, au moins, à la ligne 10 ? Même question si `source` contient exactement $n \geq 0$ octets. Justifiez vos réponses. ☐0 ☐1 ☐2 ☐3

Si le fichier source contient 42 octets, dans le meilleur des cas ces 42 octets sont lus dans le buffer de LEN = 256 octets à la ligne 10, et le programme arrive à la ligne 17 avec nbrd = 42 ; on revient donc au début de la boucle do while(nbrd > 0), et on effectue un nouvel appel à read() qui retourne 0, car on a atteint le fin de fichier : en tout, on effectue donc 2 passages à la ligne 10.

Si le fichier contient n octets, on effectue autant de passages que nécessaires pour lire ces n octets (quotient dans la division entière de n par LEN plus 1), plus un passage pour constater que la fin de fichier est atteinte. On effectue donc (il y a bien entendu d'autres formules équivalentes) :
(quotient dans la division euclidienne de n par LEN) + 2 passages.

Question 6 Écrivez un programme (sans vous occuper des fichiers d'en-tête) qui ouvre un fichier dont le nom est passé comme premier argument du programme sur la ligne de commande, et envoie son contenu sur la sortie standard. On rappelle que `STDOUT_FILENO` est une macro qui donne le descripteur de fichier pour la sortie standard. ☐0 ☐1 ☐2 ☐3 ☐4 ☐5

Notez que la question suivante rappelle l'en-tête de la fonction main() qui permet de récupérer les arguments passés sur la ligne de commande. STDOUT_FILENO est disponible par défaut, on n'a pas besoin ni d'ouvrir ni de fermer ce descripteur de fichier.

```
int main(int argc, char *argv[]) {
  const unsigned int LEN = 256;
  int nbrem, nbrd, nbwr;
  int fdin;
  char buf[LEN];

  if(argc != 2) return EXIT_FAILURE;
  fdin = open(argv[1], O_RDONLY);
  do {
    nbrem = nbrd = read(fdin, buf, LEN);
    if(nbrd == -1) return EXIT_FAILURE;
    nbwr = 0;
    while(nbrem > 0) {
      nbwr += write(STDOUT_FILENO, buf+nbwr, nbrem);
      if(nbwr == -1) return EXIT_FAILURE;
      nbrem = nbrd - nbwr;
    }
  } while(nbrd > 0);
  close(fdin);

  return EXIT_SUCCESS;
}
```

ATTENTION, CE NE SONT QUE DES ELEMENTS DE CORRECTION, IL PEUT RESTER DES ERREURS...

2 Autour des processus (6 points)

Question 7 On considère le code suivant. Quelle valeur est affichée pour la variable `a` pour le processus parent ? Pourquoi ?

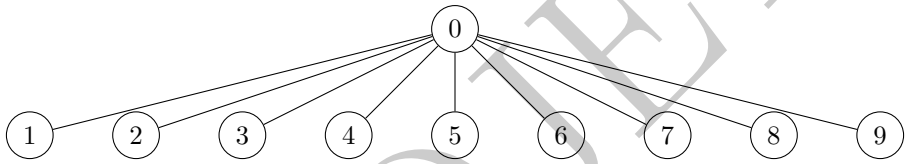
☐0 ☐1 ☐2 ☐3

```
1  int main(int argc, char *argv[]) {
2    int a = 1;
3    int ret = fork();
4    if(ret == 0) { // processus enfant
5      a = 2;
6      cout << a << endl;
7    }
8    else { // processus parent
9      waitpid(ret, NULL, 0);
10     cout << a << endl;
11   }
12   return EXIT_SUCCESS;
13 }
```

On suppose que l'appel à fork() réussit ; dans cet exemple, le processus parent attend la terminaison de son enfant avec waitpid() avant d'afficher la valeur de la variable a.

Après un appel réussi à fork(), l'enfant reçoit une copie de la mémoire du parent, mais ces deux espaces mémoires sont entièrement séparés : la modification de la valeur de a effectuée du côté du fils n'a pas d'impact sur la valeur de a du côté du parent. Le parent affiche donc 1.

Question 8 Écrivez un programme C/C++, en utilisant obligatoirement une boucle `for`, permettant de créer l'arborescence de processus suivante:



Chaque enfant affiche son numéro (pas son PID, le numéro indiqué sur la figure) et puis se termine. Le processus principal doit attendre individuellement la terminaison de chacun de ses enfants une fois qu'il les a tous créés.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5

Le père utilise un tableau pour ranger le PID de ses enfants à leur création avec fork() : comme ça, il pourra les attendre individuellement avec waipid() grâce à leurs PIDs. Notez que la question précédente donne un exemple d'utilisation de fork(), et un exemple d'appel à waitpid().

```
int main(void) {
  int pid[10]; // indices 0 à 9 ; l'indice 0 ne sera pas utilisé

  // le père crée ses 9 enfants, en rangeant leurs PIDs dans
  // un tableau pour pouvoir les attendre individuellement
  for(int i = 1; i <= 9; i++) {
    pid[i] = fork(); // le père range le PID du fils créé
    if(pid[i] == 0) { // processus enfant avec pour numéro i
      std::cout << "numéro " << i << std::endl;
      return 0; // le processus enfant numéro i se termine
    }
  }

  // le père attend la terminaison de chacun de ses fils
  for(int i = 1; i <= 9; i++)
    waipid(pid[i], NULL, 0);

  // le père se termine
  return 0;
}
```

ATTENTION, CE NE SONT QUE DES ELEMENTS DE CORRECTION, IL PEUT RESTER DES ERREURS...

3 Processus et signaux (6 points)

Question 9 On souhaite qu'un programme créé un enfant qui effectue en boucle `faire_des_choses()` jusqu'à recevoir le signal `SIGUSR1`. À la réception de ce signal, l'enfant doit se terminer après l'exécution courante de `faire_des_choses()`. Donnez un *pseudo-code* (comme on a vu en CM et en TD), en précisant bien les appels systèmes utilisés, pour ce programme.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5

Soit CONTINUER une variable booléenne globale intialisée à Vrai

Soit gest() une fonction gestionnaire du signal SIGUSR1 :
* un appel à gest() fait passer CONTINUER à Faux

Dans le programme principal :
* le programme principal crée un enfant à l'aide de fork()
* dans le processus enfant créé :
  - on installe gest() comme fonction gestionnaire pour le signal SIGUSR1 avec sigaction()
  - tant que la variable CONTINUER est à Vrai, répèter :
    appel à faire_des_choses()
  - retourner 0
* le processus principal (parent) attend la terminaison de son fils avec waitpid()
* il retourne 0

Question 10 On suppose que le parent doit envoyer le signal `SIGUSR1` à son fils : quel appel doit-il effectuer ?

☐0 ☐1 ☐2 ☐3

On suppose que le programme principal (parent) a bien conservé le PID de son fils dans une variable pid lors de sa création ; le parent doit envoyer le signal SIGUSR1 à son fils avec un appel à kill(pid, SIGUSR1);

Question 11 On suppose désormais que le signal `SIGUSR1` doit être envoyé depuis un terminal. Comment doit-on procéder ? Expliquez brièvement (une phrase) les informations nécessaires à l'envoi du signal.

☐0 ☐1 ☐2 ☐3

Il faut d'abord retrouver le PID du processus fils depuis le terminal, par exemple en utilisant la commande ps -l (l'option est un L en minuscule...)

On peut ensuite envoyer le signal au processus avec la commande kill -USR1 PID, en indiquant bien le PID du processus concerné.

ATTENTION, CE NE SONT QUE DES ELEMENTS DE CORRECTION, IL PEUT RESTER DES ERREURS...

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 */
};
```

==========

NAME kill − send signal to a process
SYNOPSIS int kill(pid_t pid, int sig);
DESCRIPTION
The kill() system call can be used to send any signal to any process group or process.
If pid is positive, then signal sig is sent to the process with the ID specified by pid.