Votre nom :
Votre prénom :
Votre no. d'étudiant :

Contrôle de LIFASR5 — 60 minutes

le mardi 16/03/2021 à 14h en amphis Thémis 10 et 11.

Toute communication (orale, téléphonique, par messagerie, etc.) avec les autres étudiants est interdite. Tout document est interdit, tout comme toute utilisation d'un ordinateur, d'un téléphone, d'une montre connectée, etc. Vous disposez uniquement de l'antisèche fournie, qui contient toutes les informations utiles. Ce sujet est à rendre complété : n'oubliez pas de noter votre nom, votre prénom et votre no. étudiant. **Après avoir parcouru tout le sujet, traitez d'abord les exercices qui vous semblent les plus abordables en fonctions de vos compétences !**

---

1 Autour des processus (7 points)

1. (4 points) On considère le programme suivant.

```
int main(void) {
    cout << "(p0: " << getpid() << ") ";
    cout << "(p1: " << getppid() << ") ";

    int ret1 = fork(); // 1er fork()
    if(ret1 == 0) {
        cout << "(p2: " << getpid() << ") ";
        cout << "(p3: " << getppid() << ") ";
        return EXIT_SUCCESS;
    }
    else {
      waitpid(ret1, NULL, 0);
      cout << "(p4: " << ret1 << ") ";
      cout << "(p5: " << getpid() << ") ";
      cout << "(p6: " << getppid() << ") ";
    }

    int ret2 = fork(); // 2ème fork()
    if(ret2 > 0) {
      waitpid(ret2, NULL, 0);
      cout << "(p7: " << ret2 << ") ";
      cout << "(p8: " << getpid() << ") ";
      cout << "(p9: " << getppid() << ") ";
      return EXIT_SUCCESS;
    }

    cout << "(p10: " << ret2 << ") ";
    cout << "(p11: " << getpid() << ") ";
    cout << "(p12: " << getppid() << ") ";

    return EXIT_SUCCESS;
}
```

On suppose que :
- le shell qui lance le programme a pour PID 4481,
- le processus principal a pour PID 7644,
- le 1er `fork()` crée un processus de PID 7645,
- le 2ème `fork()` crée un processus de PID 7646,
- tous les affichages ont lieu immédiatement après l'exécution de la ligne correspondante (on a simplifié : il faudrait ajouter `<< flush` à la fin de chaque ligne d'affichage).

On rappelle que :
- `getpid()` retourne le PID du processus appelant,
- `getppid()` retourne le PID du père du processus appelant.

Dans le cadre ci-dessous, donnez un diagramme représentant l'exécution des processus (`main()`, `fils 1` et `fils 2`) de ce programme. Différentes solutions sont acceptables : l'important est que votre diagramme soit propre, clair et lisible !

Donnez ensuite sur les lignes pointillées l'affichage produit sur la sortie standard lors de l'exécution du programme.

.....

.....

2. (3 points) **On vous donne dans ce paragraphe tous les détails utiles quant à la gestion des signaux : lisez-le bien !** Dans le processus créé, on met en place un gestionnaire pour le signal `SIGCONT`. Le processus se met ensuite en attente de la réception d'un signal quelconque avec `pause()`. Lorsqu'il recevra `SIGCONT` :
— il exécutera la fonction `handler()` de gestion du signal,
— il sortira de l'appel à `pause()`, et reprendra son exécution juste après cet appel.
Notez que l'autre processus en jeu dans ce programme envoie le signal `SIGCONT` avec `kill(ret, SIGCONT)`.

```
void handler(int s) {
  cout << "(p0) " << flush;
}

int main(void) {
    cout << "(p1: " << getpid() << ") ";
    int ret = fork();
    if(ret == 0) {
        struct sigaction act;
        sigaction(SIGCONT, NULL, &act);
        act.sa_handler = handler;
        sigaction(SIGCONT, &act, NULL); // on installe un gestionnaire pour SIGCONT
        pause(); // pause() bloque le processus en attente d'un signal quelconque
        cout << "(p2: " << getpid() << ") ";
        cout << "(p3: " << getppid() << ") ";
    }
    else {
        cout << "(p4: " << ret << ") ";
        cout << "(p5: " << getpid() << ") ";
        cout << "(p6: " << getppid() << ") ";
        sleep(1);
        kill(ret, SIGCONT); // envoi du signal SIGCONT au processus de PID ret
        waitpid(ret, NULL, 0);
    }
    cout << "(p7: " << getpid() << ") ";
    return EXIT_SUCCESS;
}
```

On suppose que tous les affichages ont lieu immédiatement après l'exécution de la ligne correspondante, et que :
— le shell dans lequel on lance le programme a pour PID 4481,
— le processus principal a pour PID 8863, et lors de l'appel à `fork()`, le processus créé a pour PID 8864,
Donnez l'affichage produit par le programme.

.....

.....

.....

2 Lecture et écriture sur des descripteurs de fichiers (7 points)

Exercice 1 : Vous êtes en train d'écrire un programme pour compter le nombre de caractères dans un fichier régulier. Vous êtes arrivé.e à la solution suivante, qui fonctionne bien : vous ne gérez pas les cas d'erreur pour `read()`, mais ça n'est pas grave : dans cet exercice, on suppose toujours que `read()` retourne toujours un entier différent de -1 !

```
1  int main(int argc, char *argv[]) {
2    int fd = open(argv[1], O_RDONLY);
3    if(fd == -1) {
4      cerr << "Impossible d'ouvrir " << argv[1] << " en lecture" << endl;
5      return EXIT_FAILURE;
6    }
7
8    int nbrd = 0; // nombre de caractères lus lors d'un appel à read()
9    int nbch = 0; // décompte du nombre de caractères
10   char c;
11   do {
12     nbrd = read(fd, &c, 1);
13     nbch += nbrd;
14   } while(nbrd == 1);
15   cout << "Nombre de caractères dans le fichier : " << nbch << endl;
16
17   close(fd);
18   return EXIT_SUCCESS;
19 }
```

1. (0,5 points) Quelles valeurs peut prendre `nbrd` dans la boucle du programme précédent, et dans quels cas ?

.....

.....

2. (0,5 points) On utilise `read()` pour lire au plus `k` > 0 caractères dans un fichier et on note `nbrd` la valeur de retour de l'appel (que l'on suppose toujours différente de -1) : comment interpréter les cas $0 <$ `nbrd` $\leq k$ et `nbrd` $= 0$?

.....

.....

3. (2 points) Vous devez modifier les lignes 11 à 14 de votre programme de façon à ne plus lire dans le fichier caractère par caractère, mais par blocs d'au plus 128 caractères. Vous devez pour cela définir un tableau de 128 `char`, et modifier votre code en conséquence.

.....

.....

.....

.....

.....

Exercice 2 : Vous avez reçu comme consigne d'écrire une fonction qui permet de copier un fichier. La fonction reçoit pour cela :

— un descripteur `fdin` sur le fichier d'entrée, supposé correctement ouvert pour la lecture,
— un descripteur sur le fichier de sortie `fdout`, supposé correctement ouvert pour l'écriture.

Voici l'état actuel de votre travail :

```
1  void copy(int fdin, int fdout) {
2    char *buf = new char[256];
3    int nbrd;
4    int nbwr;
5    do {
6      nbrd = read(fdin, buf, 256);
7      nbwr = 0;
8      while(???) {
9        ??? write(fdout, ???, ???);
10     }
11   } while(???);
12   delete[] buf;
13 }
```

Comme dans l'exercice précédent, on suppose que les appels à `read()` ne produisent jamais d'erreur, et on fait la même chose pour `write()`. On rappelle que chaque qu'une octet est équivalent à un `char`.

1. (0,5 points) En cas de succès, que retourne un appel à `write()` ?

.....

2. (1,5 points) La boucle des lignes 8 à 10 doit vous permettre d'écrire vers `fdout` les `nbrd` octets lus par le `read()` de la ligne 5. Supposez que juste avant l'appel à `write()` de la ligne 9 vous ayez déjà écrit les `nbwr` premiers octets parmi ces `nbrd` octets :
   a) À quelle condition vous reste-t-il des octets à écrire ?
   b) Combien vous en reste-t-il à écrire,
   c) à partir de quelle adresse en mémoire sont-ils rangés ?

.....

.....

.....

3. (2 points) Donnez le code complet des lignes 5 à 11 (pas plus de 7 lignes de code).

.....

.....

.....

.....

.....

.....

.....

3 Système de fichiers (6 points)

Le système de fichier FAT12 utilise une table d'allocation : il s'agit d'un tableau d'entiers sur 12 bits, avec des indices codés sur 12 bits. FAT12 alloue les blocs par chaînage dans la table d'allocation : chaque entrée de la table d'allocation contient une valeur qui est : soit l'adresse du bloc suivant d'un fichier, soit une valeur spéciale, par exemple une valeur marquant la fin du fichier. Voici la signification des valeurs possibles (elles sont données en hexadécimal) :

valeur	signification
`0x000`	le bloc est libre et peut être utilisé pour l'allocation de blocs
`0x001` à `0xFEF`	adresse du bloc suivant
`0xFF0` à `0xFF6`	le bloc est réservé par l'algorithme d'allocation de blocs
`0xFF7`	le bloc est défectueux et ne peut plus être utilisé pour l'allocation de blocs
`0xFF8` à `0xFFF`	dernier bloc d'un fichier

1. (0,5 points) Donnez le nombre de valeurs possibles pour les entrées de la table d'allocation, sous la forme d'une puissance de 2 puis en décimal.

   .....

2. (0,5 points) Combien de blocs peut comporter au maximum un fichier dans ce système de fichier ? Précisez votre calcul, et donnez votre réponse en décimal.

   .....

   .....

   .....

3. (1 point) La taille en octets des blocs dans le système FAT12 est codée sur 2 octets. Quelle est la taille maximale d'un fichier ? Donnez le calcul à faire pour avoir le résultat exact et une estimation en Mio.

   .....

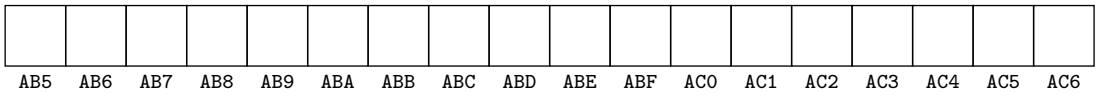
   .....

   .....

4. (2 points) Voici un extrait de la table d'allocation d'un disque, dans laquelle les adresses et les valeurs sont en hexadécimal :

adresse	AB5	AB6	AB7	AB8	AB9	ABA	ABB	ABC	ABD	ABE	ABF	AC0	AC1	AC2	AC3	AC4	AC5	AC6
valeur	AB6	AB7	FF8	FF8	ABA	ABB	000	FF7	000	FF7	000	AC1	AC2	AB9	FF8	AC3	AC4	AC5

   Complétez le schéma ci-dessous en :
   — en précisant bien les chaînages entre les blocs à l'aide de fléches,
   — en attribuant une lettre (`A, B, C`...) à chaque fichier présent,
   — en marquant les blocs qui terminent un fichier par une croix (×) et les blocs défectueux par un tiret (−),
   — en laissant vide les cases correspondant aux blocs libres.



5. (2 points) On doit ajouter 3 blocs à la fin du fichier commençant avec le bloc d'adresse `0xAC6` : expliquez quels sont les chaînages à modifier dans la table d'allocation décrite pour allouer ces blocs (plusieurs réponses sont possibles).

   .....

   .....

   .....

   .....

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 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 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);
  ... ;
};
```

sa_handler specifies the action to be associated with signum and may be SIG_DFL for the default action, SIG_IGN to ignore this signal, or a pointer to a signal handling function. This function receives the signal number as its only argument.
RETURN VALUE
sigaction() returns 0 on success; on error, -1 is returned, and errno is set to indicate the error.