SY5 – Systèmes d'exploitation
Examen de 1re session – 6 janvier 2022
Durée : 2 heures 30

Aucun document autorisé excepté une feuille A4 manuscrite
Appareils électroniques éteints et rangés

Tous les programmes demandés doivent être écrits en C, de la manière la plus lisible possible, c'est-à-dire ***bien indentés*** *et* ***commentés*** *aux endroits où cela paraît nécessaire. En revanche, s'agissant d'une épreuve sur papier, il ne vous est pas demandé un code irréprochable ; en particulier il est inutile d'indiquer les inclusions nécessaires, et vous pouvez vous contenter d'une gestion minimale des erreurs (par exemple en utilisant la fonction* `assert()`*, ou même simplement avec un commentaire* `/* erreur à gérer */`*).*

Exercice 1 : histoires de famille

1. Si le fragment de code ci-dessous s'exécute sans erreur, combien de *nouveaux* processus crée-t-il (en plus du processus initial) ?

```
fork(); fork(); fork();
```

2. Que faut-il changer pour créer uniquement 3 fils ? Ou au contraire 3 descendants de 3 générations différentes ?

3. Écrire un programme `zombies.c` dont l'exécution permet d'observer (durablement) la situation suivante (les lignes non pertinentes ont été supprimées) :

```
poulalho@lulu:SY5$ ./zombies & sleep 5; ps
    PID TTY          TIME CMD
3743177 pts/19   00:00:00 zombies
3743181 pts/19   00:00:00 sleep <defunct>
3743182 pts/19   00:00:00 sleep <defunct>
3743183 pts/19   00:00:00 sleep <defunct>
```

4. Que se passera-t-il à la mort du processus exécutant « `zombies` » ?

5. Comment modifier le programme pour créer le même nombre de processus, effectuant les mêmes tâches (en particulier, le processus principal doit continuer à s'exécuter), sans produire de zombies observables ?

Exercice 2 : à table !

1. Écrire un programme `a-table.c` qui crée 5 processus, un père et ses fils. Le père prépare le repas pendant que les fils se reposent. Lorsque tout est prêt, ce qui lui prend un temps aléatoire entre 0 (oui, 0) et 5 secondes, le père appelle ses fils en leur envoyant un signal. Les fils affichent alors leur `pid` suivi du message `"Oui papa, j'arrive!"`, puis terminent. Expliquer en particulier quand le gestionnaire doit être mis en place.

2. Modifier le programme pour reproduire un comportement plus réaliste : le père est obligé de réitérer son appel plusieurs fois, à intervalles réguliers (1 seconde par exemple) car les fils ne réagissent pas immédiatement – le fils aîné vient dès le 1er appel, le 2e attend le 2e appel, etc.

Exercice 3 : autour des tampons de la bibliothèque standard

On considère le programme suivant :

```
int main() {
  printf("salut!\n");
  sleep(1); printf("coucou, ");
  sleep(1); printf("me revoilou!");
  sleep(1); printf("\n");
  sleep(1); printf("un p'tit tour et puis s'en va...");
  exit(0);
}
```

Son exécution via « `strace` » donne le résultat suivant (la sortie est redirigée sur un autre terminal pour ne pas polluer l'affichage) :

```
poulalho@lulu:SY5$ strace ./a.out >/dev/pts/28
execve("./a.out", ["./a.out"], 0x7ffecf6dded0 /* 26 vars */) = 0
[...]
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x1c), ...}) = 0
brk(NULL)                               = 0x55de5391c000
brk(0x55de5393d000)                     = 0x55de5393d000
write(1, "salut!\n", 7)                 = 7
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7fff4b2d0a90) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7fff4b2d0a90) = 0
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7fff4b2d0a90) = 0
write(1, "coucou, me revoilou!\n", 21)  = 21
clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7fff4b2d0a90) = 0
write(1, "un p'tit tour et puis s'en va...", 32) = 32
exit_group(0)                           = ?
+++ exited with 0 +++
```

1. Expliquer cette trace, en particulier les différents appels à `write()`.
2. Que se passerait-il si on remplaçait l'appel final à `exit()` par un appel à `_exit()` ?
3. Comment rétablir l'alternance entre les écritures et les pauses ?
4. Qu'y aurait-il de différent dans chaque cas si la sortie était redirigée sur autre chose qu'un terminal (un fichier ordinaire ou un tube, par exemple) ?

Exercice 4 : parcours de répertoires

Dans cet exercice, il est interdit d'exécuter une autre commande que le programme à écrire (en particulier, pas de « `find` »), et d'utiliser les bilbiothèques `ftw` et `fts`.

1. Écrire un programme `cpt-dir.c` qui parcourt le répertoire courant et affiche le nombre de sous-répertoires [1] qu'il contient.
2. Modifier ce programme pour qu'il compte tous les répertoires de l'arborescence dont la racine est le répertoire courant (y compris celui-ci).

(On supposera que l'arborescence ne contient pas de liens symboliques)

1. au sens strict, c'est-à-dire situés à la profondeur suivante dans l'arborescence

Exercice 5 : liste des ouvertures de fichiers

La commande « `lsof` » permet de lister les entrées de la table des fichiers ouverts ; avec en paramètre le nom d'un fichier, elle se limite aux ouvertures de celui-ci. Par exemple, si je crée un tube nommé `/tmp/montube` puis que je lance deux processus exécutant respectivement « `cat /tmp/montube` » et « `cat > /tmp/montube` » depuis deux terminaux différents, j'obtiens :

```
poulalho@lulu:SY5$ lsof /tmp/montube
COMMAND     PID     USER   FD   TYPE DEVICE SIZE/OFF  NODE NAME
cat     3656042 poulalho    3r  FIFO   0,65      0t0 56215 /tmp/montube
cat     3656063 poulalho    1w  FIFO   0,65      0t0 56215 /tmp/montube
```

Un même processus peut apparaître plusieurs fois, par exemple si un job exécutant « `vim` » tourne dans le terminal `/dev/pts/19` d'où a été lancé le processus `3656063`, cela donne :

```
poulalho@lulu:SY5$ lsof /dev/pts/19
COMMAND     PID     USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
bash    3621352 poulalho    0u   CHR 136,19      0t0   22 /dev/pts/19
bash    3621352 poulalho    1u   CHR 136,19      0t0   22 /dev/pts/19
bash    3621352 poulalho    2u   CHR 136,19      0t0   22 /dev/pts/19
bash    3621352 poulalho  255u   CHR 136,19      0t0   22 /dev/pts/19
vim     3655729 poulalho    0u   CHR 136,28      0t0   22 /dev/pts/19
vim     3655729 poulalho    1u   CHR 136,28      0t0   22 /dev/pts/19
vim     3655729 poulalho    2u   CHR 136,28      0t0   22 /dev/pts/19
cat     3656063 poulalho    0u   CHR 136,19      0t0   22 /dev/pts/19
cat     3656063 poulalho    2u   CHR 136,19      0t0   22 /dev/pts/19
```

En particulier, la colonne « `FD` » indique les descripteurs et les modes associés à chaque ouverture (« `u` » indiquant une ouverture en lecture et en écriture).

1. Expliquer les valeurs correspondant aux ouvertures de `/tmp/montube` et `/dev/pts/19` par les deux processus « `cat` », en précisant qui est le lecteur et qui est l'écrivain.

L'option « `-t` » permet de n'obtenir que la liste (sans doublon) des processus possédant (au moins) une ouverture du fichier :

```
poulalho@lulu:SY5$ lsof -t /tmp/montube
3656042
3656063
poulalho@lulu:SY5$ lsof -t /dev/pts/19
3621352
3655729
3656063
```

2. Écrire un programme `cpof.c` utilisant « `lsof` » pour compter les processus possédant une ouverture sur un fichier dont le nom est passé en paramètre. Par exemple, dans la situation précédente, on attend l'affichage suivant :

```
poulalho@lulu:SY5$ ./cpof /tmp/montube
2
poulalho@lulu:SY5$ ./cpof /dev/pts/19
3
```

Votre programme ne doit exécuter aucune autre commande que « `lsof` ». L'usage de la bibliothèque `stdio` doit être limité à la gestion des erreurs et à l'affichage final.

Exercice 6 : (encore) les tampons... et un problème de synchronisation

1. Quels affichages le programme suivant peut-il produire ?

```
int main() {
  printf("ping ");
  if (fork() == 0) printf("pong ");
  else printf("pang");
}
```

2. Proposer une modification minimale permettant d'assurer que l'affichage soit toujours [2] `"ping pong pang"` (l'affichage intermédiaire `"pong "` étant toujours réalisé par le fils).

3. Cette solution est-elle adaptable en inversant les rôles (c'est-à-dire que le fils réalise les affichages `"ping "` et `"pang"`) ?

4. Proposer une solution pour inverser les rôles en synchronisant les deux processus à l'aide de tube(s) anonyme(s).

5. Proposer une autre solution de synchronisation reposant sur l'envoi de signaux. Expliquer où réside le risque d'interblocage, et dites en quelques mots ce qu'il faudrait faire pour l'éviter (il n'est pas demandé de le faire effectivement).

Petit mémento

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
struct dirent {    /* contient entre autres : */
    ino_t  d_ino;      /* Inode number */
    char   d_name[];   /* Null-terminated filename */
};
int stat(const char *pathname, struct stat *statbuf);
struct stat {   /* contient entre autres : */
    dev_t    st_dev;       /* ID of device containing file */
    ino_t    st_ino;       /* Inode number */
    mode_t   st_mode;      /* File type and mode */
};
int execlp(const char *file, const char *arg, ... /* (char *) NULL */);
int execvp(const char *file, char *const argv[]);
int kill(pid_t pid, int sig);
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
struct sigaction {  /* contient entre autres : */
  void    (*sa_handler)(int);
  sigset_t  sa_mask;
  int       sa_flags;
}
```

2. sauf en cas d'erreur de `fork()`, naturellement