SY5 – Systèmes d'exploitation
Examen de 1re session – 4 janvier 2023
Durée : 3 heures

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

On considère une commande « `zombies` » dont l'exécution provoque la situation suivante :

```
poulalho@lulu:SY5$ ./zombies & sleep 5 ; ps j
   PPID     PID    PGID     SID TTY      STAT   UID   TIME COMMAND
1952538 1952539 1952539 1952539 pts/20   Ss    8159   0:00 -bash
1952539 1963818 1963818 1952539 pts/20   S     8159   0:00 zombies
1963818 1963822 1963818 1952539 pts/20   S     8159   0:00 zombies
1963822 1963823 1963818 1952539 pts/20   Z     8159   0:00 [sleep] <defunct>
1963822 1963824 1963818 1952539 pts/20   Z     8159   0:00 [sleep] <defunct>
1963822 1963825 1963818 1952539 pts/20   Z     8159   0:00 [sleep] <defunct>
1963822 1963826 1963818 1952539 pts/20   Z     8159   0:00 [sleep] <defunct>
1952539 1963861 1963861 1952539 pts/20   R+    8159   0:00 ps j
poulalho@lulu:SY5$ sleep 5 ; ps j
   PPID     PID    PGID     SID TTY      STAT   UID   TIME COMMAND
1952538 1952539 1952539 1952539 pts/20   Ss    8159   0:00 -bash
1952539 1963818 1963818 1952539 pts/20   S     8159   0:00 zombies
1952539 1963932 1963932 1952539 pts/20   R+    8159   0:00 ps j
```

1. Dessiner la généalogie des processus créés par l'exécution de « `zombies` ».
2. Expliquer précisément l'état de chacun de ces processus au bout de 5 secondes (c'est-à-dire lors de la première exécution de « `ps j` »).
3. Que s'est-il passé durant les 5 secondes suivantes (c'est-à-dire entre la première et la deuxième exécution de « `ps j` ») ?
4. Écrire un programme `zombies.c` ayant exactement le comportement observé.

Exercice 2 : recherche de liens corrompus

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 bibliothèques `ftw` et `fts`.

Écrire un programme `check-lnk.c` qui liste tous les liens symboliques contenus dans ***l'arborescence*** dont le répertoire courant est la racine, et dont la cible ***n'est pas*** une référence valide. Dans le cas où un lien symbolique pointe vers un répertoire, ce lien ne sera pas suivi. Le format attendu est le suivant :

```
lien corrompu : ./toto -> tutu
lien corrompu : ./A/tata -> ../tata
lien corrompu : ./A/B/titi -> /usr/bin/titi
```

(la fonction `readlink()` sera utile pour l'affichage : si `path` est un lien symbolique, `readlink(path, buf, bufsize)` place son conntenu dans `buf`)

Exercice 3 : à table !

Écrire un programme `a-table.c` qui crée 11 processus, un père et ses 10 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 et 10 secondes, le père appelle ses fils en leur envoyant un signal. Les fils réagissent à son appel mais pas immédiatement : au bout d'un temps aléatoire de 0 à 5 secondes (dépendant du fils), chacun affiche son `pid` suivi du message `"Oui, j'arrive!"`, puis termine. Le père affiche alors le message `"Ah, enfin,"`, suivi du `pid` du fils concerné. Il termine lorsque tous les fils sont arrivés à table.

Expliquer en particulier quand le gestionnaire de signal doit être mis en place, et comment obtenir des temps aléatoires indépendants.

Exercice 4 : décompte d'appels système

On rappelle que l'exécution de « `strace` *cmd* » provoque deux affichages : sur la sortie standard, celui de la commande *cmd*, et sur la sortie *erreur* standard, une trace des appels systèmes effectués, au format suivant :

```
[...]
openat(AT_FDCWD, "Test", O_RDONLY|O_NONBLOCK|O_CLOEXEC|O_DIRECTORY) = 3
fstat(3, {st_mode=S_IFDIR|0755, st_size=4096, ...})                  = 0
getdents64(3, 0x556707f122e0 /* 4 entries */, 32768)                 = 96
lstat("Test/.", {st_mode=S_IFDIR|0755, st_size=4096, ...})           = 0
lstat("Test/..", {st_mode=S_IFDIR|0755, st_size=4096, ...})          = 0
lstat("Test/toto", {st_mode=S_IFREG|0644, st_size=0, ...})           = 0
lstat("Test/tutu", {st_mode=S_IFLNK|0777, st_size=4, ...})           = 0
[...]
+++ exited with 0 +++
```

On considèrera pour simplifier qu'à l'exception de la dernière ligne, chaque ligne de l'affichage correspond exactement à un appel système. Écrire un programme `compte-appels.c` prenant en argument un nom de commande *cmd*, et utilisant « `strace` » pour compter les appels système effectués lors d'une exécution de *cmd*. Hormis *cmd*, votre programme ne doit exécuter aucune autre commande que « `strace` ». Le seul affichage produit doit être le nombre d'appels système calculé ; en particulier, l'éventuelle sortie standard de *cmd* ne doit pas apparaître. L'usage de la bibliothèque `stdio` doit être limité à la gestion des erreurs et à l'affichage final.

Exercice 5 : autour des tampons de la bibliothèque standard

On considère le programme suivant :

```
int main() {
    printf("pif ");
    sleep(1);
    printf("paf\n");
    sleep(1);
    if (fork() == 0)
        printf("pouf ");
    else {
        sleep(1);
        printf("bang\n");
    }
    exit(0);
}
```

En exécutant ce programme via « `strace` », avec l'option « `-f` » pour suivre les deux processus, et en redirigeant la sortie sur un autre terminal pour ne pas polluer l'affichage, on obtient le résultat suivant :

```
poulalho@lulu:SY5$ strace -f ./a.out > /dev/pts/5
execve("./a.out", ["./a.out"], 0x7ffdcd079d68 /* 26 vars */) = 0
[...]
fstat(1, {st_mode=S_IFCHR|0620, [...]}) = 0
[...]
clock_nanosleep([...], tv_sec=1, [...]) = 0
write(1, "pif paf\n", 8)                = 8
clock_nanosleep([...], tv_sec=1, [...]) = 0
clone([...])                            = 117038
[pid 117037] clock_nanosleep([...], tv_sec=1, [...]},  <unfinished ...>
[pid 117038] write(1, "pouf ", 5)       = 5
[pid 117038] exit_group(0)              = ?
[pid 117038] +++ exited with 0 +++
<... clock_nanosleep resumed>)          = 0
write(1, "bang\n", 5)                   = 5
exit_group(0)                           = ?
+++ exited with 0 +++
```

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

Exercice 6 : un problème de synchronisation

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

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

2. Proposer une modification minimale du programme permettant d'assurer que l'affichage soit toujours `"ping pong pang "` (sauf en cas d'erreur de `fork()`, naturellement).

3. Cette solution est-elle adaptable pour assurer un autre affichage (sans changer la généalogie, ni le message affiché par chaque processus) ?

4. Proposer une solution pour assurer l'affichage `"pang pong ping "`, sans changer l'ordre des `fork` et des `printf`, en synchronisant les processus à l'aide de tube(s) anonyme(s).

5. Proposer une autre solution de synchronisation reposant sur l'envoi de signaux. Expliquer en particulier quand le(s) gestionnaire(s) doi(ven)t être mis en place. S'il y a un risque de blocage, expliquer comment l'éviter.

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 lstat(const char *pathname, struct stat *statbuf);
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 */
};
ssize_t readlink(const char *restrict path, char *restrict buf, size_t bufsize);
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;
}
```