Télécom Physique Strasbourg - Université de Strasbourg
Examen de Programmation Système
FIP 2A - 2013-2014
Durée : 1h45

Adlane HABED

CETTE COPIE APPARTIENT À

Nom : _____

Prénom : _____

d'anonymat : _____

ATTENTION

- Ecrire vos nom et prénom dès le début de l'épreuve.
- Aucun document autorisé.
- Vérifier que le sujet contient 13 pages (celle-ci comprise).
- Barème approximatif : partie I (8 pts), partie II (12 pts).
- Répondre à toutes les questions.
- Durée : 1h45

Partie I : "Vrai" ou "Faux" (≈8 points)

Cocher la bonne réponse :

1. Dans la librairie standard du C, le mode de "buffering" (l'utilisation ou non de mémoire tampon pour les opérations de lecture et d'écriture concernant les fichiers) dépend de la fonction utilisée.

   ☐ Vrai ☐ Faux

2. Les appels système qui concernent les entrées-sorties sous Unix sont toujours en mode "unbuffered" (n'utilise pas de mémoire tampon).

   ☐ Vrai ☐ Faux

3. Les processus orphelins gardent toutes leurs ressources (segments de données, de code et de pile).

   ☐ Vrai ☐ Faux

4. Si un processus père termine avant son fils, le processus fils devient un processus zombie.

   ☐ Vrai ☐ Faux

5. Un processus orphelin ne peut pas devenir un processus zombie.

   ☐ Vrai ☐ Faux

6. Les processus zombies gardent toutes leurs ressources (segments de données, de code et de pile).

   ☐ Vrai ☐ Faux

7. Le PPID d'un processus orphelin est toujours 1.

   ☐ Vrai ☐ Faux

8. Les appels système de la famille `exec` ne retournent jamais au programme appelant.

   ☐ Vrai ☐ Faux

9. Lorsqu'un processus fait un appel à `fork()`, une copie du processus est créée et toutes les variables locales sont partagées entre le processus père et le fils.

☐ Vrai ☐ Faux

10. L'appel à `wait()` bloque le processus appelant jusqu'à la réception du signal `SIGCHLD`.

☐ Vrai ☐ Faux

11. L'appel système `signal()` est utilisé par un processus pour envoyer des signaux à d'autres processus.

☐ Vrai ☐ Faux

12. L'appel système `kill()` est utilisé uniquement pour l'arrêt d'un processus.

☐ Vrai ☐ Faux

13. Un tube créé par l'appel système `pipe()` permet une communication full-duplex entre des processus.

☐ Vrai ☐ Faux

14. Un appel `read()` sur un pipe anonyme (c'est-à-dire créé avec `pipe()`), vide mais non fermé, bloque le processus appelant.

☐ Vrai ☐ Faux

15. Les FIFO ne peuvent être utilisés qu'entre des processus qui s'exécutent sur le même ordinateur.

☐ Vrai ☐ Faux

16. Un FIFO permet une communication half-duplex entre des processus n'ayant pas nécessairement le même propriétaire.

☐ Vrai ☐ Faux

17. L'appel `open()` en mode lecture sur un FIFO bloque le processus appelant jusqu'à ce qu'un autre processus ouvre le FIFO en écriture.

☐ Vrai ☐ Faux

18. Les sockets ne peuvent être utilisées qu'entre des processus qui s'exécutent sur des ordinateurs distants.

☐ Vrai ☐ Faux

19. La fonction `connect()` de la librairie des sockets doit être appelée par le client et le serveur pour permettre d'établir une connexion TCP/IP.

☐ Vrai ☐ Faux

20. La fonction `listen()` de la librairie des sockets bloque le serveur jusqu'à réception d'une requête de connexion.

☐ Vrai ☐ Faux

21. La fonction `exit()` envoie un le signal `SIGCHLD` au processus père.

☐ Vrai ☐ Faux

22. Lorsque le processus père execute `exit(-1)`, le père et tous ses processus fils se terminent.

☐ Vrai ☐ Faux

23. `alarm(0)` lance un signal `SIGALRM` immediatement (après 0 secondes) au processus appelant.

☐ Vrai ☐ Faux

24. Si `alarm(4)` est executé avant un appel à `fork`, le signal `SIGALRM` sera envoyé au processus père et au processus fils après 4 secondes.

☐ Vrai ☐ Faux

25. Le programme `myprog`, executé à travers `execl` ne sera certainement pas affecté par le signal `SIGALRM`

```
alarm(5);
execl("./myprog","myprog", NULL);
```

☐ Vrai ☐ Faux

Partie II - Questions-Réponses (≈12 Points)

On supposera que tous les `#include` des fichiers entête sont présents. Répondre précisément aux questions suivantes :

1. Lequels des blocs "BLOCK A", "BLOCK B", "BLOCK C" et "BLOCK D" seront executés par le processus fils? Quels sont ceux qui seront executés par le processus père?

```
int main(int argc, char *argv[]){
    int pid = fork();
    if(pid == -1){
        /* BLOCK A */
        ...
    }
    else if(pid > 0){
        /* BLOCK B */
        ...
    }
    else{
        /* BLOCK C */
        ...
    }
    /* BLOCK D */
    ...
}
```

Réponse :

2. Quelle est la différence entre un appel de fonction et l'execution d'un executable avec une des fonctions de la famille `exec`?

Réponse :

3. Examiner le fragment de code ci-dessous et signaler toute erreur pouvant s'y trouver. On supposera qu'un executable `foobar`, prenant un entier en argument de ligne de commande, se trouve dans le répertoire courant. Expliquer pour quoi il s'agit d'une erreur (ou d'erreurs) et comment la (les) corriger.

```
int instance;
for ( instance = 0; instance < 10; ){
    pid_t pid, cpid;
    if ( ( pid = fork() ) < 0 )
        perror ( "Could not fork a child" );
    if ( pid )
        cpid = wait ( &status );
    execl ( ".", "foobar", (char *)( instance++ ), (char *)0 );
}
```

Réponse :

4. Quel est le rôle des fonctions `wait` et `waitpid`? Quel est la différence entre ces deux fonctions?

Réponse :

5. Considerer le fichier `myfile` suivant :

   `-r--r--r--  3 habed None 0 Dec 1 14:00 myfile`

   Quel effet aura l'appel `unlink("./myfile")` de l'utilisateur `habed` à partir du réportoire contenant `myfile` ?

   Réponse :

6. Quel sera l'effet de `lseek(fd, 10, SEEK_END)` sur le processus appelant ?

   Réponse :

7. Ecrire quelques lignes de code qui auront pour effet la création d'un processus zombie.

   Réponse :

8. Ecrire quelques lignes de code qui auront pour effet la création d'un processus orphelin.

Réponse :

9. Considérer la fonction `myFork` suivante :

```
pid_t myFork(){
    static int count=0;
    count++;
    if(count <= 3) return(fork());
    else return(-1);
}
```

Si on utilise `myFork` au lieu de `fork` dans un programme, quel est le nombre maximal de processus que ce même programme pourra créer ?

Réponse :

10. Combien de processus vont executer la boucle infinie dans le programme suivant ?

```
main(){
    int a = 2;
    if(!fork())
        if(!fork())
            execlp("ls", "ls", NULL);
    printf("Infinite loop\n");
    while(1)
        sleep(1);
}
```

Réponse :

11. Qu'affiche le programme suivant ?

```
main(){
    int a = 2;
    if(fork()==0){
        sleep(1);
        a = a*2;
        fork();
    }
    printf("%d\n",a);
}
```

Réponse :

12. Qu'affiche le programme suivant ?

```
main(int argc, char *argv[]){
    off_t n; //off_t is a long integer
    int fd, i;
    char data[20][100];

    unlink("data.txt");
    fd=open("data.txt", O_CREAT|O_WRONLY);
    for(i=0; i<20;i++)
        write(fd, data[i], 100);
    n=lseek(fd,0, SEEK_CUR);
    printf("Value=%d\n", (int)n);
    close(fd);
}
```

Réponse :

13. Apporter les changements nécessaires pour que le programme suivant ne soit jamais interrompu par un signal `SIGCHLD` ?

```
void alarm_handler(int dummy){

    alarm(3);
}
int main(int argc, char *argv[]){

    signal(SIGCHLD, alarm_handler);
    alarm(3);

    while(1){
        printf("I am working\n");
        sleep(1);

    }
}
```

Réponse :

14. Donner un exemple de programme synchronisant un processus père et un processus fils par l'envoi de signaux. Le père et le fils executent des boucles infinis.

Réponse :

15. Trouver et corriger les erreurs dans le programme suivant :

```
void child(int *);
void parent(int *);
int main(int argc, char *argv[]){
    int fd[2];
    if(pipe(fd) == -1)exit(1);
    if(fork() == 0) child(fd);
    else parent(fd);
    exit(0);
}

void parent(int *fd){
    char ch;

    do{
        read(fd[0], &ch, 1);
        printf("%c", ch);
        if(ch == '\n')break;
    }while(1);
}

void child(int *fd){
    char message='a';

    write(fd[1], &message, 1);
}
```

Réponse :